

The L^AT_EX3 Sources

The L^AT_EX3 Project*

Released 2020-10-27

Abstract

This is the reference documentation for the `expl3` programming environment. The `expl3` modules set up an experimental naming scheme for L^AT_EX commands, which allow the L^AT_EX programmer to systematically name functions and variables, and specify the argument types of functions.

The T_EX and ε -T_EX primitives are all given a new name according to these conventions. However, in the main direct use of the primitives is not required or encouraged: the `expl3` modules define an independent low-level L^AT_EX3 programming language.

At present, the `expl3` modules are designed to be loaded on top of L^AT_EX 2 ε . In time, a L^AT_EX3 format will be produced based on this code. This allows the code to be used in L^AT_EX 2 ε packages *now* while a stand-alone L^AT_EX3 is developed.

While `expl3` is still experimental, the bundle is now regarded as broadly stable. The syntax conventions and functions provided are now ready for wider use. There may still be changes to some functions, but these will be minor when compared to the scope of `expl3`.

New modules will be added to the distributed version of `expl3` as they reach maturity.

*E-mail: latex-team@latex-project.org

Contents

I	Introduction to <code>expl3</code> and this document	1
1	Naming functions and variables	1
1.1	Terminological inexactitude	3
2	Documentation conventions	4
3	Formal language conventions which apply generally	5
4	<code>TeX</code> concepts not supported by <code>L^AT_EX3</code>	6
II	The <code>l3bootstrap</code> package: Bootstrap code	7
1	Using the <code>L^AT_EX3</code> modules	7
III	The <code>l3names</code> package: Namespace for primitives	8
1	Setting up the <code>L^AT_EX3</code> programming language	8
IV	The <code>l3basics</code> package: Basic definitions	9
1	No operation functions	9
2	Grouping material	9
3	Control sequences and functions	10
3.1	Defining functions	10
3.2	Defining new functions using parameter text	11
3.3	Defining new functions using the signature	12
3.4	Copying control sequences	15
3.5	Deleting control sequences	15
3.6	Showing control sequences	15
3.7	Converting to and from control sequences	16
4	Analysing control sequences	17
5	Using or removing tokens and arguments	18
5.1	Selecting tokens from delimited arguments	20
6	Predicates and conditionals	21
6.1	Tests on control sequences	22
6.2	Primitive conditionals	22
7	Starting a paragraph	24
7.1	Debugging support	24

V	The <code>l3expan</code> package: Argument expansion	25
1	Defining new variants	25
2	Methods for defining variants	26
3	Introducing the variants	27
4	Manipulating the first argument	29
5	Manipulating two arguments	31
6	Manipulating three arguments	32
7	Unbraced expansion	33
8	Preventing expansion	33
9	Controlled expansion	35
10	Internal functions	37
VI	The <code>l3quark</code> package: Quarks	38
1	Quarks	38
2	Defining quarks	38
3	Quark tests	39
4	Recursion	39
5	An example of recursion with quarks	40
6	Scan marks	41
VII	The <code>l3tl</code> package: Token lists	43
1	Creating and initialising token list variables	43
2	Adding data to token list variables	44
3	Modifying token list variables	45
4	Reassigning token list category codes	45
5	Token list conditionals	46
6	Mapping to token lists	48
7	Using token lists	51

8	Working with the content of token lists	51
9	The first token from a token list	53
10	Using a single item	56
11	Viewing token lists	58
12	Constant token lists	58
13	Scratch token lists	59
VIII The <code>I3str</code> package: Strings		60
1	Building strings	60
2	Adding data to string variables	61
3	Modifying string variables	62
4	String conditionals	63
5	Mapping to strings	64
6	Working with the content of strings	66
7	String manipulation	69
8	Viewing strings	70
9	Constant token lists	71
10	Scratch strings	71
IX The <code>I3str-convert</code> package: string encoding conversions		72
1	Encoding and escaping schemes	72
2	Conversion functions	72
3	Conversion by expansion (for PDF contexts)	74
4	Possibilities, and things to do	74
X The <code>I3seq</code> package: Sequences and stacks		75
1	Creating and initialising sequences	75
2	Appending data to sequences	76
3	Recovering items from sequences	76

4	Recovering values from sequences with branching	78
5	Modifying sequences	79
6	Sequence conditionals	80
7	Mapping to sequences	80
8	Using the content of sequences directly	83
9	Sequences as stacks	84
10	Sequences as sets	85
11	Constant and scratch sequences	86
12	Viewing sequences	87
 XI The I3int package: Integers		88
1	Integer expressions	89
2	Creating and initialising integers	90
3	Setting and incrementing integers	91
4	Using integers	92
5	Integer expression conditionals	92
6	Integer expression loops	94
7	Integer step functions	96
8	Formatting integers	97
9	Converting from other formats to integers	98
10	Random integers	99
11	Viewing integers	100
12	Constant integers	100
13	Scratch integers	100
13.1	Direct number expansion	101
14	Primitive conditionals	101
 XII The I3flag package: Expandable flags		103
1	Setting up flags	103

2	Expandable flag commands	104
XIII The <i>l3prg</i> package: Control structures		105
1	Defining a set of conditional functions	105
2	The boolean data type	107
3	Boolean expressions	109
4	Logical loops	111
5	Producing multiple copies	112
6	Detecting <i>T_EX</i> 's mode	112
7	Primitive conditionals	113
8	Nestable recursions and mappings	113
8.1	Simple mappings	113
9	Internal programming functions	114
XIV The <i>l3sys</i> package: System/runtime functions		115
1	The name of the job	115
2	Date and time	115
3	Engine	115
4	Output format	116
5	Platform	116
6	Random numbers	116
7	Access to the shell	117
8	Loading configuration data	118
8.1	Final settings	118
XV The <i>l3clist</i> package: Comma separated lists		119
1	Creating and initialising comma lists	119
2	Adding data to comma lists	121
3	Modifying comma lists	121

4	Comma list conditionals	123
5	Mapping to comma lists	123
6	Using the content of comma lists directly	125
7	Comma lists as stacks	126
8	Using a single item	127
9	Viewing comma lists	127
10	Constant and scratch comma lists	128
XVI The <code>l3token</code> package: Token manipulation		129
1	Creating character tokens	129
2	Manipulating and interrogating character tokens	131
3	Generic tokens	134
4	Converting tokens	134
5	Token conditionals	135
6	Peeking ahead at the next token	138
7	Description of all possible tokens	141
XVII The <code>l3prop</code> package: Property lists		144
1	Creating and initialising property lists	144
2	Adding entries to property lists	145
3	Recovering values from property lists	145
4	Modifying property lists	146
5	Property list conditionals	146
6	Recovering values from property lists with branching	147
7	Mapping to property lists	148
8	Viewing property lists	149
9	Scratch property lists	149
10	Constants	150

XVIII The <code>l3msg</code> package: Messages	151
1 Creating new messages	151
2 Contextual information for messages	152
3 Issuing messages	153
3.1 Expandable error messages	155
4 Redirecting messages	156
XIX The <code>l3file</code> package: File and I/O operations	158
1 Input–output stream management	158
1.1 Reading from files	159
1.2 Writing to files	162
1.3 Wrapping lines in output	164
1.4 Constant input–output streams, and variables	165
1.5 Primitive conditionals	165
2 File operation functions	165
XX The <code>l3skip</code> package: Dimensions and skips	170
1 Creating and initialising <code>dim</code> variables	170
2 Setting <code>dim</code> variables	171
3 Utilities for dimension calculations	171
4 Dimension expression conditionals	172
5 Dimension expression loops	174
6 Dimension step functions	175
7 Using <code>dim</code> expressions and variables	176
8 Viewing <code>dim</code> variables	177
9 Constant dimensions	178
10 Scratch dimensions	178
11 Creating and initialising <code>skip</code> variables	178
12 Setting <code>skip</code> variables	179
13 Skip expression conditionals	180
14 Using <code>skip</code> expressions and variables	180

15	Viewing skip variables	180
16	Constant skips	181
17	Scratch skips	181
18	Inserting skips into the output	181
19	Creating and initialising <code>muskip</code> variables	182
20	Setting <code>muskip</code> variables	182
21	Using <code>muskip</code> expressions and variables	183
22	Viewing <code>muskip</code> variables	183
23	Constant muskips	184
24	Scratch muskips	184
25	Primitive conditional	184
 XXI The <code>I3keys</code> package: Key–value interfaces		185
1	Creating keys	186
2	Sub-dividing keys	190
3	Choice and multiple choice keys	191
4	Setting keys	193
5	Handling of unknown keys	193
6	Selective key setting	194
7	Utility functions for keys	195
8	Low-level interface for parsing key–val lists	196
 XXII The <code>I3intarray</code> package: fast global integer arrays		198
1	<code>I3intarray</code> documentation	198
1.1	Implementation notes	199
 XXIII The <code>I3fp</code> package: Floating points		200
1	Creating and initialising floating point variables	201
2	Setting floating point variables	202

3	Using floating points	202
4	Floating point conditionals	204
5	Floating point expression loops	205
6	Some useful constants, and scratch variables	207
7	Floating point exceptions	208
8	Viewing floating points	209
9	Floating point expressions	210
9.1	Input of floating point numbers	210
9.2	Precedence of operators	211
9.3	Operations	211
10	Disclaimer and roadmap	218
XXIV	The I3fparray package: fast global floating point arrays	221
1	I3fparray documentation	221
XXV	The I3cctab package: Category code tables	222
1	Creating and initialising category code tables	222
2	Using category code tables	222
3	Category code table conditionals	223
4	Constant category code tables	223
XXVI	The I3sort package: Sorting functions	224
1	Controlling sorting	224
XXVII	The I3tl-analysis package: Analysing token lists	225
1	I3tl-analysis documentation	225
XXVIII	The I3regex package: Regular expressions in T_EX	226
1	Syntax of regular expressions	226
2	Syntax of the replacement text	231
3	Pre-compiling regular expressions	233

4	Matching	233
5	Submatch extraction	234
6	Replacement	235
7	Constants and variables	235
8	Bugs, misfeatures, future work, and other possibilities	236
XXIX	The l3box package: Boxes	239
1	Creating and initialising boxes	239
2	Using boxes	239
3	Measuring and setting box dimensions	240
4	Box conditionals	241
5	The last box inserted	241
6	Constant boxes	241
7	Scratch boxes	242
8	Viewing box contents	242
9	Boxes and color	242
10	Horizontal mode boxes	242
11	Vertical mode boxes	244
12	Using boxes efficiently	245
13	Affine transformations	246
14	Primitive box conditionals	249
XXX	The l3coffins package: Coffin code layer	250
1	Creating and initialising coffins	250
2	Setting coffin content and poles	250
3	Coffin affine transformations	252
4	Joining and using coffins	252
5	Measuring coffins	253

6	Coffin diagnostics	253
7	Constants and variables	254
XXXI The <code>I3color-base</code> package: Color support		255
1	Color in boxes	255
XXXII The <code>I3luatex</code> package: LuaTeX-specific functions		256
1	Breaking out to Lua	256
2	Lua interfaces	257
XXXIII The <code>I3unicode</code> package: Unicode support functions		259
XXXIV The <code>I3text</code> package: text processing		260
1	<code>I3text</code> documentation	260
1.1	Expanding text	260
1.2	Case changing	262
1.3	Removing formatting from text	263
1.4	Control variables	263
XXXV The <code>I3legacy</code> package: Interfaces to legacy concepts		264
XXXVI The <code>I3candidates</code> package: Experimental additions to <code>I3kernel</code>		265
1	Important notice	265
2	Additions to <code>I3box</code>	265
2.1	Viewing part of a box	265
3	Additions to <code>I3expan</code>	266
4	Additions to <code>I3fp</code>	266
5	Additions to <code>I3file</code>	266
6	Additions to <code>I3flag</code>	267
7	Additions to <code>I3intarray</code>	267
7.1	Working with contents of integer arrays	267
8	Additions to <code>I3msg</code>	268

9	Additions to <code>I3prg</code>	268
10	Additions to <code>I3prop</code>	269
11	Additions to <code>I3seq</code>	270
12	Additions to <code>I3sys</code>	271
13	Additions to <code>I3tl</code>	272
14	Additions to <code>I3token</code>	273
XXXVII Implementation		274
1	<code>I3bootstrap</code> implementation	274
1.1	Lua ^{TeX} -specific code	274
1.2	The <code>\pdfstrcmp</code> primitive in X ^{TeX}	275
1.3	Loading support Lua code	275
1.4	Engine requirements	276
1.5	Extending allocators	277
1.6	The ^{La} TeX3 code environment	278
2	<code>I3names</code> implementation	279
3	Internal kernel functions	305
4	Kernel backend functions	311
5	<code>I3basics</code> implementation	312
5.1	Renaming some ^T EX primitives (again)	312
5.2	Defining some constants	314
5.3	Defining functions	315
5.4	Selecting tokens	315
5.5	Gobbling tokens from input	317
5.6	Debugging and patching later definitions	317
5.7	Conditional processing and definitions	318
5.8	Dissecting a control sequence	324
5.9	Exist or free	326
5.10	Preliminaries for new functions	328
5.11	Defining new functions	329
5.12	Copying definitions	331
5.13	Undefining functions	331
5.14	Generating parameter text from argument count	332
5.15	Defining functions from a given number of arguments	333
5.16	Using the signature to define functions	334
5.17	Checking control sequence equality	336
5.18	Diagnostic functions	336
5.19	Decomposing a macro definition	338
5.20	Doing nothing functions	338
5.21	Breaking out of mapping functions	339

5.22 Starting a paragraph	339
6 I3expn implementation	340
6.1 General expansion	340
6.2 Hand-tuned definitions	343
6.3 Last-unbraced versions	347
6.4 Preventing expansion	349
6.5 Controlled expansion	350
6.6 Emulating e-type expansion	350
6.7 Defining function variants	357
6.8 Definitions with the automated technique	368
7 I3quark implementation	369
7.1 Quarks	369
7.2 Scan marks	378
8 I3tl implementation	379
8.1 Functions	379
8.2 Constant token lists	381
8.3 Adding to token list variables	381
8.4 Internal quarks and quark-query functions	383
8.5 Reassigning token list category codes	384
8.6 Modifying token list variables	387
8.7 Token list conditionals	391
8.8 Mapping to token lists	396
8.9 Using token lists	398
8.10 Working with the contents of token lists	398
8.11 The first token from a token list	401
8.12 Token by token changes	406
8.13 Using a single item	409
8.14 Viewing token lists	412
8.15 Internal scan marks	413
8.16 Scratch token lists	413
9 I3str implementation	413
9.1 Internal auxiliaries	413
9.2 Creating and setting string variables	414
9.3 Modifying string variables	415
9.4 String comparisons	416
9.5 Mapping to strings	419
9.6 Accessing specific characters in a string	420
9.7 Counting characters	425
9.8 The first character in a string	427
9.9 String manipulation	428
9.10 Viewing strings	429

10	I3str-convert implementation	429
10.1	Helpers	429
10.1.1	Variables and constants	429
10.2	String conditionals	431
10.3	Conversions	432
10.3.1	Producing one byte or character	432
10.3.2	Mapping functions for conversions	433
10.3.3	Error-reporting during conversion	434
10.3.4	Framework for conversions	435
10.3.5	Byte unescape and escape	439
10.3.6	Native strings	440
10.3.7	clist	441
10.3.8	8-bit encodings	442
10.4	Messages	445
10.5	Escaping definitions	446
10.5.1	Unescape methods	446
10.5.2	Escape methods	451
10.6	Encoding definitions	453
10.6.1	UTF-8 support	453
10.6.2	UTF-16 support	458
10.6.3	UTF-32 support	463
10.7	PDF names and strings by expansion	466
10.7.1	ISO 8859 support	467
11	I3seq implementation	482
11.1	Allocation and initialisation	484
11.2	Appending data to either end	487
11.3	Modifying sequences	487
11.4	Sequence conditionals	490
11.5	Recovering data from sequences	492
11.6	Mapping to sequences	495
11.7	Using sequences	499
11.8	Sequence stacks	500
11.9	Viewing sequences	501
11.10	Scratch sequences	502
12	I3int implementation	502
12.1	Integer expressions	503
12.2	Creating and initialising integers	505
12.3	Setting and incrementing integers	507
12.4	Using integers	508
12.5	Integer expression conditionals	508
12.6	Integer expression loops	512
12.7	Integer step functions	513
12.8	Formatting integers	515
12.9	Converting from other formats to integers	520
12.10	Viewing integer	523
12.11	Random integers	523
12.12	Constant integers	524
12.13	Scratch integers	524

12.14	Integers for earlier modules	524
13	 3flag implementation	525
13.1	Non-expandable flag commands	525
13.2	Expandable flag commands	526
14	 3prg implementation	527
14.1	Primitive conditionals	527
14.2	Defining a set of conditional functions	527
14.3	The boolean data type	527
14.4	Internal auxiliaries	528
14.5	Boolean expressions	530
14.6	Logical loops	534
14.7	Producing multiple copies	535
14.8	Detecting T _E X's mode	537
14.9	Internal programming functions	538
15	 3sys implementation	538
15.1	Kernel code	538
15.1.1	Detecting the engine	538
15.1.2	Randomness	540
15.1.3	Platform	541
15.1.4	Configurations	541
15.1.5	Access to the shell	543
15.2	Dynamic (every job) code	545
15.2.1	The name of the job	545
15.2.2	Time and date	546
15.2.3	Random numbers	546
15.2.4	Access to the shell	547
15.2.5	Held over from 3file	548
15.3	Last-minute code	548
15.3.1	Detecting the output	548
15.3.2	Configurations	549
16	 3clist implementation	550
16.1	Removing spaces around items	551
16.2	Allocation and initialisation	552
16.3	Adding data to comma lists	554
16.4	Comma lists as stacks	555
16.5	Modifying comma lists	557
16.6	Comma list conditionals	560
16.7	Mapping to comma lists	561
16.8	Using comma lists	564
16.9	Using a single item	565
16.10	Viewing comma lists	567
16.11	Scratch comma lists	567

17	I3token implementation	568
17.1	Internal auxiliaries	568
17.2	Manipulating and interrogating character tokens	568
17.3	Creating character tokens	570
17.4	Generic tokens	579
17.5	Token conditionals	580
17.6	Peeking ahead at the next token	588
18	I3prop implementation	594
18.1	Internal auxiliaries	596
18.2	Allocation and initialisation	596
18.3	Accessing data in property lists	598
18.4	Property list conditionals	603
18.5	Recovering values from property lists with branching	604
18.6	Mapping to property lists	604
18.7	Viewing property lists	606
19	I3msg implementation	606
19.1	Internal auxiliaries	607
19.2	Creating messages	607
19.3	Messages: support functions and text	608
19.4	Showing messages: low level mechanism	609
19.5	Displaying messages	611
19.6	Kernel-specific functions	620
19.7	Expandable errors	628
20	I3file implementation	630
20.1	Input operations	630
20.1.1	Variables and constants	630
20.1.2	Stream management	631
20.1.3	Reading input	634
20.2	Output operations	637
20.2.1	Variables and constants	637
20.2.2	Internal auxiliaries	638
20.3	Stream management	638
20.3.1	Deferred writing	640
20.3.2	Immediate writing	640
20.3.3	Special characters for writing	641
20.3.4	Hard-wrapping lines to a character count	641
20.4	File operations	650
20.4.1	Internal auxiliaries	652
20.5	GetIdInfo	668
20.6	Checking the version of kernel dependencies	669
20.7	Messages	671
20.8	Functions delayed from earlier modules	671

21	I3skip implementation	672
21.1	Length primitives renamed	672
21.2	Internal auxiliaries	672
21.3	Creating and initialising <code>dim</code> variables	672
21.4	Setting <code>dim</code> variables	673
21.5	Utilities for dimension calculations	674
21.6	Dimension expression conditionals	675
21.7	Dimension expression loops	677
21.8	Dimension step functions	678
21.9	Using <code>dim</code> expressions and variables	680
21.10	Viewing <code>dim</code> variables	681
21.11	Constant dimensions	682
21.12	Scratch dimensions	682
21.13	Creating and initialising <code>skip</code> variables	682
21.14	Setting <code>skip</code> variables	683
21.15	Skip expression conditionals	684
21.16	Using <code>skip</code> expressions and variables	685
21.17	Inserting skips into the output	685
21.18	Viewing <code>skip</code> variables	685
21.19	Constant skips	686
21.20	Scratch skips	686
21.21	Creating and initialising <code>muskip</code> variables	686
21.22	Setting <code>muskip</code> variables	687
21.23	Using <code>muskip</code> expressions and variables	688
21.24	Viewing <code>muskip</code> variables	688
21.25	Constant muskips	688
21.26	Scratch muskips	688
22	I3keys Implementation	689
22.1	Low-level interface	689
22.2	Constants and variables	695
22.2.1	Internal auxiliaries	697
22.3	The key defining mechanism	697
22.4	Turning properties into actions	699
22.5	Creating key properties	705
22.6	Setting keys	710
22.7	Utilities	719
22.8	Messages	721
23	I3intarray implementation	722
23.1	Allocating arrays	722
23.2	Array items	723
23.3	Working with contents of integer arrays	725
23.4	Random arrays	727
24	I3fp implementation	728

25	<i>l3fp-aux</i> implementation	728
25.1	Access to primitives	728
25.2	Internal representation	729
25.3	Using arguments and semicolons	730
25.4	Constants, and structure of floating points	731
25.5	Overflow, underflow, and exact zero	733
25.6	Expanding after a floating point number	733
25.7	Other floating point types	735
25.8	Packing digits	738
25.9	Decimate (dividing by a power of 10)	740
25.10	Functions for use within primitive conditional branches	742
25.11	Integer floating points	744
25.12	Small integer floating points	744
25.13	Fast string comparison	745
25.14	Name of a function from its <i>l3fp-parse</i> name	745
25.15	Messages	745
26	<i>l3fp-traps</i> Implementation	746
26.1	Flags	746
26.2	Traps	746
26.3	Errors	750
26.4	Messages	750
27	<i>l3fp-round</i> implementation	751
27.1	Rounding tools	751
27.2	The <i>round</i> function	755
28	<i>l3fp-parse</i> implementation	758
28.1	Work plan	759
28.1.1	Storing results	760
28.1.2	Precedence and infix operators	761
28.1.3	Prefix operators, parentheses, and functions	764
28.1.4	Numbers and reading tokens one by one	765
28.2	Main auxiliary functions	766
28.3	Helpers	767
28.4	Parsing one number	768
28.4.1	Numbers: trimming leading zeros	774
28.4.2	Number: small significand	776
28.4.3	Number: large significand	778
28.4.4	Number: beyond 16 digits, rounding	780
28.4.5	Number: finding the exponent	782
28.5	Constants, functions and prefix operators	785
28.5.1	Prefix operators	785
28.5.2	Constants	789
28.5.3	Functions	790
28.6	Main functions	790
28.7	Infix operators	792
28.7.1	Closing parentheses and commas	794
28.7.2	Usual infix operators	796
28.7.3	Juxtaposition	797

28.7.4	Multi-character cases	797
28.7.5	Ternary operator	798
28.7.6	Comparisons	798
28.8	Tools for functions	800
28.9	Messages	802
29	I3fp-assign implementation	803
29.1	Assigning values	803
29.2	Updating values	804
29.3	Showing values	805
29.4	Some useful constants and scratch variables	805
30	I3fp-logic Implementation	806
30.1	Syntax of internal functions	806
30.2	Tests	806
30.3	Comparison	807
30.4	Floating point expression loops	810
30.5	Extrema	813
30.6	Boolean operations	815
30.7	Ternary operator	816
31	I3fp-basics Implementation	817
31.1	Addition and subtraction	817
31.1.1	Sign, exponent, and special numbers	818
31.1.2	Absolute addition	820
31.1.3	Absolute subtraction	822
31.2	Multiplication	826
31.2.1	Signs, and special numbers	826
31.2.2	Absolute multiplication	827
31.3	Division	830
31.3.1	Signs, and special numbers	830
31.3.2	Work plan	831
31.3.3	Implementing the significand division	833
31.4	Square root	838
31.5	About the sign and exponent	845
31.6	Operations on tuples	846
32	I3fp-extended implementation	847
32.1	Description of fixed point numbers	848
32.2	Helpers for numbers with extended precision	848
32.3	Multiplying a fixed point number by a short one	849
32.4	Dividing a fixed point number by a small integer	850
32.5	Adding and subtracting fixed points	851
32.6	Multiplying fixed points	852
32.7	Combining product and sum of fixed points	853
32.8	Extended-precision floating point numbers	855
32.9	Dividing extended-precision numbers	858
32.10	Inverse square root of extended precision numbers	861
32.11	Converting from fixed point to floating point	863

33	I3fp-expo implementation	865
33.1	Logarithm	865
33.1.1	Work plan	865
33.1.2	Some constants	866
33.1.3	Sign, exponent, and special numbers	866
33.1.4	Absolute ln	866
33.2	Exponential	874
33.2.1	Sign, exponent, and special numbers	874
33.3	Power	878
33.4	Factorial	884
34	I3fp-trig Implementation	886
34.1	Direct trigonometric functions	887
34.1.1	Filtering special cases	887
34.1.2	Distinguishing small and large arguments	890
34.1.3	Small arguments	891
34.1.4	Argument reduction in degrees	891
34.1.5	Argument reduction in radians	893
34.1.6	Computing the power series	900
34.2	Inverse trigonometric functions	903
34.2.1	Arctangent and arccotangent	904
34.2.2	Arcsine and arccosine	909
34.2.3	Arccosecant and arcsecant	911
35	I3fp-convert implementation	912
35.1	Dealing with tuples	912
35.2	Trimming trailing zeros	913
35.3	Scientific notation	913
35.4	Decimal representation	914
35.5	Token list representation	916
35.6	Formatting	917
35.7	Convert to dimension or integer	918
35.8	Convert from a dimension	918
35.9	Use and eval	919
35.10	Convert an array of floating points to a comma list	920
36	I3fp-random Implementation	921
36.1	Engine support	921
36.2	Random floating point	925
36.3	Random integer	925
37	I3farray implementation	930
37.1	Allocating arrays	930
37.2	Array items	931

38	<i>I3cctab</i> implementation	934
38.1	Variables	934
38.2	Allocating category code tables	935
38.3	Saving category code tables	936
38.4	Using category code tables	937
38.5	Category code table conditionals	942
38.6	Constant category code tables	943
38.7	Messages	945
39	<i>I3sort</i> implementation	946
39.1	Variables	946
39.2	Finding available \toks registers	947
39.3	Protected user commands	949
39.4	Merge sort	951
39.5	Expandable sorting	954
39.6	Messages	959
40	<i>I3tl-analysis</i> implementation	960
40.1	Internal functions	960
40.2	Internal format	960
40.3	Variables and helper functions	961
40.4	Plan of attack	963
40.5	Disabling active characters	964
40.6	First pass	964
40.7	Second pass	969
40.8	Mapping through the analysis	972
40.9	Showing the results	973
40.10	Messages	975
41	<i>I3regex</i> implementation	976
41.1	Plan of attack	976
41.2	Helpers	977
41.2.1	Constants and variables	979
41.2.2	Testing characters	980
41.2.3	Internal auxiliaries	980
41.2.4	Character property tests	984
41.2.5	Simple character escape	985
41.3	Compiling	991
41.3.1	Variables used when compiling	991
41.3.2	Generic helpers used when compiling	993
41.3.3	Mode	994
41.3.4	Framework	996
41.3.5	Quantifiers	999
41.3.6	Raw characters	1002
41.3.7	Character properties	1003
41.3.8	Anchoring and simple assertions	1004
41.3.9	Character classes	1005
41.3.10	Groups and alternations	1008
41.3.11	Catcodes and csnames	1011
41.3.12	Raw token lists with \u	1015

41.3.13 Other	1017
41.3.14 Showing regexes	1017
41.4 Building	1021
41.4.1 Variables used while building	1021
41.4.2 Framework	1022
41.4.3 Helpers for building an NFA	1023
41.4.4 Building classes	1025
41.4.5 Building groups	1026
41.4.6 Others	1031
41.5 Matching	1032
41.5.1 Variables used when matching	1033
41.5.2 Matching: framework	1035
41.5.3 Using states of the NFA	1039
41.5.4 Actions when matching	1039
41.6 Replacement	1042
41.6.1 Variables and helpers used in replacement	1042
41.6.2 Query and brace balance	1043
41.6.3 Framework	1044
41.6.4 Submatches	1046
41.6.5 Csnames in replacement	1048
41.6.6 Characters in replacement	1049
41.6.7 An error	1053
41.7 User functions	1053
41.7.1 Variables and helpers for user functions	1055
41.7.2 Matching	1056
41.7.3 Extracting submatches	1056
41.7.4 Replacement	1060
41.7.5 Storing and showing compiled patterns	1062
41.8 Messages	1062
41.9 Code for tracing	1068
42 I3box implementation	1069
42.1 Support code	1069
42.2 Creating and initialising boxes	1069
42.3 Measuring and setting box dimensions	1070
42.4 Using boxes	1071
42.5 Box conditionals	1071
42.6 The last box inserted	1071
42.7 Constant boxes	1072
42.8 Scratch boxes	1072
42.9 Viewing box contents	1072
42.10 Horizontal mode boxes	1073
42.11 Vertical mode boxes	1075
42.12 Affine transformations	1078

43	I3coffins Implementation	1087
43.1	Coffins: data structures and general variables	1087
43.2	Basic coffin functions	1088
43.3	Measuring coffins	1094
43.4	Coffins: handle and pole management	1094
43.5	Coffins: calculation of pole intersections	1097
43.6	Affine transformations	1100
43.7	Aligning and typesetting of coffins	1108
43.8	Coffin diagnostics	1113
43.9	Messages	1119
44	I3color-base Implementation	1119
45	I3luatex implementation	1121
45.1	Breaking out to Lua	1121
45.2	Messages	1122
45.3	Lua functions for internal use	1122
46	I3unicode implementation	1128
47	I3text implementation	1131
47.1	Internal auxiliaries	1131
47.2	Utilities	1132
47.3	Configuration variables	1135
47.4	Expansion to formatted text	1136
48	I3text-case implementation	1143
48.1	Case changing	1143
48.2	Case changing data for 8-bit engines	1164
49	I3text-purify implementation	1175
49.1	Purifying text	1175
49.2	Accent and letter-like data for purifying text	1180
50	I3legacy Implementation	1187
51	I3candidates Implementation	1188
51.1	Additions to I3box	1188
51.1.1	Viewing part of a box	1188
51.2	Additions to I3flag	1190
51.3	Additions to I3msg	1191
51.4	Additions to I3prg	1191
51.5	Additions to I3prop	1193
51.6	Additions to I3seq	1193
51.7	Additions to I3sys	1195
51.8	Additions to I3file	1196
51.9	Additions to I3tl	1196
51.9.1	Building a token list	1196
51.9.2	Other additions to I3tl	1200
51.10	Additions to I3token	1200

52	I3deprecation implementation	1202
52.1	Helpers and variables	1202
52.2	Patching definitions to deprecate	1203
52.3	Removed functions	1206
52.4	Loading the patches	1210
52.5	Deprecated I3box functions	1210
52.6	Deprecated I3str functions	1211
52.7	Deprecated I3seq functions	1211
52.7.1	Deprecated I3tl functions	1212
52.8	Deprecated I3token functions	1213
52.9	Deprecated I3file functions	1213
	Index	1214

Part I

Introduction to `expl3` and this document

This document is intended to act as a comprehensive reference manual for the `expl3` language. A general guide to the L^AT_EX3 programming language is found in [expl3.pdf](#).

1 Naming functions and variables

L^AT_EX3 does not use @ as a “letter” for defining internal macros. Instead, the symbols _ and : are used in internal macro names to provide structure. The name of each *function* is divided into logical units using _, while : separates the *name* of the function from the *argument specifier* (“arg-spec”). This describes the arguments expected by the function. In most cases, each argument is represented by a single letter. The complete list of arg-spec letters for a function is referred to as the *signature* of the function.

Each function name starts with the *module* to which it belongs. Thus apart from a small number of very basic functions, all `expl3` function names contain at least one underscore to divide the module name from the descriptive name of the function. For example, all functions concerned with comma lists are in module `clist` and begin `\clist_`.

Every function must include an argument specifier. For functions which take no arguments, this will be blank and the function name will end :. Most functions take one or more arguments, and use the following argument specifiers:

N and **n** These mean *no manipulation*, of a single token for N and of a set of tokens given in braces for n. Both pass the argument through exactly as given. Usually, if you use a single token for an n argument, all will be well.

c This means *csname*, and indicates that the argument will be turned into a csname before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`.

V and **v** These mean *value of variable*. The V and v specifiers are used to get the content of a variable without needing to worry about the underlying T_EX structure containing the data. A V argument will be a single token (similar to N), for example `\foo:V \MyVariable`; on the other hand, using v a csname is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.

o This means *expansion once*. In general, the V and v specifiers are favoured over o for recovering stored information. However, o is useful for correctly processing information with delimited arguments.

x The x specifier stands for *exhaustive expansion*: every token in the argument is fully expanded until only unexpandable ones remain. The T_EX `\edef` primitive carries out this type of expansion. Functions which feature an x-type argument are *not* expandable.

e The e specifier is in many respects identical to x, but with a very different implementation. Functions which feature an e-type argument may be expandable. The drawback is that e is extremely slow (often more than 200 times slower) in older engines, more precisely in non-LuaT_EX engines older than 2019.

- f** The **f** specifier stands for *full expansion*, and in contrast to **x** stops at the first non-expandable token (reading the argument from left to right) without trying to expand it. If this token is a *<space token>*, it is gobbled, and thus won't be part of the resulting argument. For example, when setting a token list variable (a macro used for storage), the sequence

```
\tl_set:Nn \l_mya_t1 { A }
\tl_set:Nn \l_myb_t1 { B }
\tl_set:Nf \l_mya_t1 { \l_mya_t1 \l_myb_t1 }
```

will leave **\l_mya_t1** with the content **A\l_myb_t1**, as **A** cannot be expanded and so terminates expansion before **\l_myb_t1** is considered.

T and F For logic tests, there are the branch specifiers **T** (*true*) and **F** (*false*). Both specifiers treat the input in the same way as **n** (no change), but make the logic much easier to see.

- p** The letter **p** indicates *TeX parameters*. Normally this will be used for delimited functions as `expl3` provides better methods for creating simple sequential arguments.
- w** Finally, there is the **w** specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some specified string).
- D** The **D** specifier means *do not use*. All of the *TeX* primitives are initially `\let` to a **D** name, and some are then given a second name. Only the kernel team should use anything with a **D** specifier!

Notice that the argument specifier describes how the argument is processed prior to being passed to the underlying function. For example, `\foo:c` will take its argument, convert it to a control sequence and pass it to `\foo:N`.

Variables are named in a similar manner to functions, but begin with a single letter to define the type of variable:

- c** Constant: global parameters whose value should not be changed.
- g** Parameters whose value should only be set globally.
- l** Parameters whose value should only be set locally.

Each variable name is then build up in a similar way to that of a function, typically starting with the module¹ name and then a descriptive part. Variables end with a short identifier to show the variable type:

clist Comma separated list.

dim “Rigid” lengths.

fp Floating-point values;

int Integer-valued count register.

¹The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the **int** module contains some scratch variables called **\l_tmpa_int**, **\l_tmpb_int**, and so on. In such a case adding the module name up front to denote the module and in the back to indicate the type, as in **\l_int_tmpa_int** would be very unreadable.

muskip “Rubber” lengths for use in mathematics.

seq “Sequence”: a data-type used to implement lists (with access at both ends) and stacks.

skip “Rubber” lengths.

str String variables: contain character data.

t1 Token list variables: placeholder for a token list.

Applying V-type or v-type expansion to variables of one of the above types is supported, while it is not supported for the following variable types:

bool Either true or false.

box Box register.

coffin A “box with handles” — a higher-level data type for carrying out **box** alignment operations.

flag Integer that can be incremented expandably.

farray Fixed-size array of floating point values.

intarray Fixed-size array of integers.

ior/iow An input or output stream, for reading from or writing to, respectively.

prop Property list: analogue of dictionary or associative arrays in other languages.

regex Regular expression.

1.1 Terminological inexactitude

A word of warning. In this document, and others referring to the `expl3` programming modules, we often refer to “variables” and “functions” as if they were actual constructs from a real programming language. In truth, `TeX` is a macro processor, and functions are simply macros that may or may not take arguments and expand to their replacement text. Many of the common variables are *also* macros, and if placed into the input stream will simply expand to their definition as well — a “function” with no arguments and a “token list variable” are almost the same.² On the other hand, some “variables” are actually registers that must be initialised and their values set and retrieved with specific functions.

The conventions of the `expl3` code are designed to clearly separate the ideas of “macros that contain data” and “macros that contain code”, and a consistent wrapper is applied to all forms of “data” whether they be macros or actually registers. This means that sometimes we will use phrases like “the function returns a value”, when actually we just mean “the macro expands to something”. Similarly, the term “execute” might be used in place of “expand” or it might refer to the more specific case of “processing in `TeX`’s stomach” (if you are familiar with the `TeXbook` parlance).

If in doubt, please ask; chances are we’ve been hasty in writing certain definitions and need to be told to tighten up our terminology.

²`TeXnically`, functions with no arguments are `\long` while token list variables are not.

2 Documentation conventions

This document is typeset with the experimental `l3doc` class; several conventions are used to help describe the features of the code. A number of conventions are used here to make the documentation clearer.

Each group of related functions is given in a box. For a function with a “user” name, this might read:

```
\ExplSyntaxOn ... \ExplSyntaxOff
```

The textual description of how the function works would appear here. The syntax of the function is shown in mono-spaced text to the right of the box. In this example, the function takes no arguments and so the name of the function is simply reprinted.

For programming functions, which use `_` and `:` in their name there are a few additional conventions: If two related functions are given with identical names but different argument specifiers, these are termed *variants* of each other, and the latter functions are printed in grey to show this more clearly. They will carry out the same function but will take different types of argument:

```
\seq_new:N <sequence>
\seq_new:c
```

When a number of variants are described, the arguments are usually illustrated only for the base function. Here, `<sequence>` indicates that `\seq_new:N` expects the name of a sequence. From the argument specifier, `\seq_new:c` also expects a sequence name, but as a name rather than as a control sequence. Each argument given in the illustration should be described in the following text.

Fully expandable functions Some functions are fully expandable, which allows them to be used within an `x`-type or `e`-type argument (in plain `TEX` terms, inside an `\edef` or `\expanded`), as well as within an `f`-type argument. These fully expandable functions are indicated in the documentation by a star:

```
\cs_to_str:N *
```

As with other functions, some text should follow which explains how the function works. Usually, only the star will indicate that the function is expandable. In this case, the function expects a `(cs)`, shorthand for a `(control sequence)`.

Restricted expandable functions A few functions are fully expandable but cannot be fully expanded within an `f`-type argument. In this case a hollow star is used to indicate this:

```
\seq_map_function:NN *
```

Conditional functions Conditional (`if`) functions are normally defined in three variants, with `T`, `F` and `TF` argument specifiers. This allows them to be used for different “true”/“false” branches, depending on which outcome the conditional is being used to test. To indicate this without repetition, this information is given in a shortened form:

`\sys_if_engine_xetex:TF *` `\sys_if_engine_xetex:TF {<true code>} {{<false code>}}`

The underlining and italic of TF indicates that three functions are available:

- `\sys_if_engine_xetex:T`
- `\sys_if_engine_xetex:F`
- `\sys_if_engine_xetex:TF`

Usually, the illustration will use the TF variant, and so both *<true code>* and *<false code>* will be shown. The two variant forms T and F take only *<true code>* and *<false code>*, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, *all* conditional functions in the `expl3` modules should be defined in this way.

Variables, constants and so on are described in a similar manner:

`\l_tmpa_t1`

A short piece of text will describe the variable: there is no syntax illustration in this case.

In some cases, the function is similar to one in L^AT_EX 2_ε or plain T_EX. In these cases, the text will include an extra “**TeXhackers note**” section:

`\token_to_str:N *`

`\token_to_str:N <token>`

The normal description text.

TeXhackers note: Detail for the experienced T_EX or L^AT_EX 2_ε programmer. In this case, it would point out that this function is the T_EX primitive `\string`.

Changes to behaviour When new functions are added to `expl3`, the date of first inclusion is given in the documentation. Where the documented behaviour of a function changes after it is first introduced, the date of the update will also be given. This means that the programmer can be sure that any release of `expl3` after the date given will contain the function of interest with expected behaviour as described. Note that changes to code internals, including bug fixes, are not recorded in this way *unless* they impact on the expected behaviour.

3 Formal language conventions which apply generally

As this is a formal reference guide for L^AT_EX3 programming, the descriptions of functions are intended to be reasonably “complete”. However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarised here.

For tests which have a TF argument specification, the test is evaluated to give a logically TRUE or FALSE result. Depending on this result, either the *<true code>* or the *<false code>* will be left in the input stream. In the case where the test is expandable, and a predicate (_p) variant is available, the logical value determined by the test is left in the input stream: this will typically be part of a larger logical construct.

4 T_EX concepts not supported by L_AT_EX3

The T_EX concept of an “\outer” macro is *not supported* at all by L_AT_EX3. As such, the functions provided here may break when used on top of L_AT_EX 2_ε if \outer tokens are used in the arguments.

Part II

The **I3bootstrap** package

Bootstrap code

1 Using the L^AT_EX3 modules

The modules documented in `source3` are designed to be used on top of L^AT_EX 2_E and are loaded all as one with the usual `\usepackage{exp13}` or `\RequirePackage{exp13}` instructions.

As the modules use a coding syntax different from standard L^AT_EX 2_E it provides a few functions for setting it up.

```
\ExplSyntaxOn <code> \ExplSyntaxOff
```

The `\ExplSyntaxOn` function switches to a category code régime in which spaces are ignored and in which the colon (`:`) and underscore (`_`) are treated as “letters”, thus allowing access to the names of code functions and variables. Within this environment, `~` is used to input a space. The `\ExplSyntaxOff` reverts to the document category code régime.

```
\ProvidesExplPackage
\ProvidesExplClass
\ProvidesExplFile
```

Updated: 2011-08-13

```
\RequirePackage{exp13}
\ProvidesExplPackage {<package>} {<date>} {<version>} {<description>}
```

These functions act broadly in the same way as the corresponding L^AT_EX 2_E kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as L^AT_EX 2_E provides in turning on `\makeatletter` within package and class code.) The `<date>` should be given in the format `<year>/<month>/<day>`. If the `<version>` is given then it will be prefixed with `v` in the package identifier line.

```
\GetIdInfo

```

Updated: 2012-06-04

```
\RequirePackage{l3bootstrap}
\GetIdInfo $Id: <SVN info field> $ {<description>}
```

Extracts all information from a SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with `\ExplFileName` for the part of the file name leading up to the period, `\ExplFileDate` for date, `\ExplFileVersion` for version and `\ExplFileDescription` for the description.

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with `\RequirePackage` or similar are loaded with usual L^AT_EX 2_E category codes and the L^AT_EX3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the `\GetIdInfo` command you can use the information when loading a package with

```
\ProvidesExplPackage{\ExplFileName}
{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

Part III

The **I3names** package

Namespace for primitives

1 Setting up the L^AT_EX3 programming language

This module is at the core of the L^AT_EX3 programming language. It performs the following tasks:

- defines new names for all T_EX primitives;
- emulate required primitives not provided by default in LuaT_EX;
- switches to the category code régime for programming;

This module is entirely dedicated to primitives (and emulations of these), which should not be used directly within L^AT_EX3 code (outside of “kernel-level” code). As such, the primitives are not documented here: *The T_EXbook*, *T_EX by Topic* and the manuals for pdfT_EX, X_ET_EX, LuaT_EX, pT_EX and upT_EX should be consulted for details of the primitives. These are named `\tex_{<name>}:D`, typically based on the primitive’s `<name>` in pdfT_EX and omitting a leading pdf when the primitive is not related to pdf output.

Part IV

The **I3basics** package

Basic definitions

As the name suggest this package holds some basic definitions which are needed by most or all other packages in this set.

Here we describe those functions that are used all over the place. With that we mean functions dealing with the construction and testing of control sequences. Furthermore the basic parts of conditional processing are covered; conditional processing dealing with specific data types is described in the modules specific for the respective data types.

1 No operation functions

`\prg_do_nothing: *`

`\prg_do_nothing:`

An expandable function which does nothing at all: leaves nothing in the input stream after a single expansion.

`\scan_stop:`

`\scan_stop:`

A non-expandable function which does nothing. Does not vanish on expansion but produces no typeset output.

2 Grouping material

`\group_begin:`

`\group_end:`

`\group_begin:`

`\group_end:`

These functions begin and end a group for definition purposes. Assignments are local to groups unless carried out in a global manner. (A small number of exceptions to this rule will be noted as necessary elsewhere in this document.) Each `\group_begin:` must be matched by a `\group_end:`, although this does not have to occur within the same function. Indeed, it is often necessary to start a group within one function and finish it within another, for example when seeking to use non-standard category codes.

`\group_insert_after:N <token>`

Adds `<token>` to the list of `<tokens>` to be inserted when the current group level ends. The list of `<tokens>` to be inserted is empty at the beginning of a group: multiple applications of `\group_insert_after:N` may be used to build the inserted list one `<token>` at a time. The current group level may be closed by a `\group_end:` function or by a token with category code 2 (close-group), namely a `}` if standard category codes apply.

3 Control sequences and functions

As TeX is a macro language, creating new functions means creating macros. At point of use, a function is replaced by the replacement text (“code”) in which each parameter in the code (#1, #2, etc.) is replaced by the appropriate arguments absorbed by the function. In the following, *<code>* is therefore used as a shorthand for “replacement text”.

Functions which are not “protected” are fully expanded inside an **x** expansion. In contrast, “protected” functions are not expanded within **x** expansions.

3.1 Defining functions

Functions can be created with no requirement that they are declared first (in contrast to variables, which must always be declared). Declaring a function before setting up the code means that the name chosen is checked and an error raised if it is already in use. The name of a function can be checked at the point of definition using the `\cs_new...` functions: this is recommended for all functions which are defined for the first time.

There are three ways to define new functions. All classes define a function to expand to the substitution text. Within the substitution text the actual parameters are substituted for the formal parameters (#1, #2, ...).

new Create a new function with the **new** scope, such as `\cs_new:Npn`. The definition is global and results in an error if it is already defined.

set Create a new function with the **set** scope, such as `\cs_set:Npn`. The definition is restricted to the current TeX group and does not result in an error if the function is already defined.

gset Create a new function with the **gset** scope, such as `\cs_gset:Npn`. The definition is global and does not result in an error if the function is already defined.

Within each set of scope there are different ways to define a function. The differences depend on restrictions on the actual parameters and the expandability of the resulting function.

nopar Create a new function with the **nopar** restriction, such as `\cs_set_nopar:Npn`. The parameter may not contain `\par` tokens.

protected Create a new function with the **protected** restriction, such as `\cs_set_protected:Npn`. The parameter may contain `\par` tokens but the function will not expand within an **x**-type or **e**-type expansion.

Finally, the functions in Subsections 3.2 and 3.3 are primarily meant to define *base functions* only. Base functions can only have the following argument specifiers:

N and **n** No manipulation.

T and **F** Functionally equivalent to **n** (you are actually encouraged to use the family of `\prg_new_conditional:` functions described in Section 1).

p and **w** These are special cases.

The `\cs_new:` functions below (and friends) do not stop you from using other argument specifiers in your function names, but they do not handle expansion for you. You should define the base function and then use `\cs_generate_variant:Nn` to generate custom variants as described in Section 2.

3.2 Defining new functions using parameter text

```
\cs_new:Npn \cs_new:cpx
```

`\cs_new:Npn <function> <parameters> {{code}}`

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. The definition is global and an error results if the `<function>` is already defined.

```
\cs_new_nopar:Npn \cs_new_nopar:cpx
```

`\cs_new_nopar:Npn <function> <parameters> {{code}}`

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The definition is global and an error results if the `<function>` is already defined.

```
\cs_new_protected:Npn \cs_new_protected:cpx
```

`\cs_new_protected:Npn <function> <parameters> {{code}}`

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. The `<function>` will not expand within an `x`-type argument. The definition is global and an error results if the `<function>` is already defined.

```
\cs_new_protected_nopar:Npn \cs_new_protected_nopar:cpx
```

`\cs_new_protected_nopar:Npn <function> <parameters> {{code}}`

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The `<function>` will not expand within an `x`-type or `e`-type argument. The definition is global and an error results if the `<function>` is already defined.

```
\cs_set:Npn \cs_set:cpx
```

`\cs_set:Npn <function> <parameters> {{code}}`

Sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the `<function>` is restricted to the current TeX group level.

```
\cs_set_nopar:Npn \cs_set_nopar:cpx
```

`\cs_set_nopar:Npn <function> <parameters> {{code}}`

Sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The assignment of a meaning to the `<function>` is restricted to the current TeX group level.

```
\cs_set_protected:Npn \cs_set_protected:cpx
```

`\cs_set_protected:Npn <function> <parameters> {{code}}`

Sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the `<function>` is restricted to the current TeX group level. The `<function>` will not expand within an `x`-type or `e`-type argument.

```
\cs_set_protected_nopar:Npn \cs_set_protected_nopar:Npn <function> <parameters> {<code>}
\cs_set_protected_nopar:cpx
\cs_set_protected_nopar:Npx
\cs_set_protected_nopar:cpx
```

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. When the *<function>* is used the *<parameters>* absorbed cannot contain `\par` tokens. The assignment of a meaning to the *<function>* is restricted to the current TeX group level. The *<function>* will not expand within an x-type or e-type argument.

```
\cs_gset:Npn \cs_gset:cpx
\cs_gset:Npx \cs_gset:cpx
```

Globally sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the *<function>* is *not* restricted to the current TeX group level: the assignment is global.

```
\cs_gset_nopar:Npn \cs_gset_nopar:cpx
\cs_gset_nopar:Npx \cs_gset_nopar:cpx
```

```
\cs_gset_nopar:Npn <function> <parameters> {<code>}
```

Globally sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. When the *<function>* is used the *<parameters>* absorbed cannot contain `\par` tokens. The assignment of a meaning to the *<function>* is *not* restricted to the current TeX group level: the assignment is global.

```
\cs_gset_protected:Npn \cs_gset_protected:cpx
\cs_gset_protected:Npx \cs_gset_protected:cpx
```

```
\cs_gset_protected:Npn <function> <parameters> {<code>}
```

Globally sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the *<function>* is *not* restricted to the current TeX group level: the assignment is global. The *<function>* will not expand within an x-type or e-type argument.

```
\cs_gset_protected_nopar:Npn \cs_gset_protected_nopar:Npn <function> <parameters> {<code>}
\cs_gset_protected_nopar:cpx
\cs_gset_protected_nopar:Npx \cs_gset_protected_nopar:cpx
```

Globally sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. When the *<function>* is used the *<parameters>* absorbed cannot contain `\par` tokens. The assignment of a meaning to the *<function>* is *not* restricted to the current TeX group level: the assignment is global. The *<function>* will not expand within an x-type argument.

3.3 Defining new functions using the signature

```
\cs_new:Nn \cs_new:(cn|Nx|cx)
```

```
\cs_new:Nn <function> {<code>}
```

Creates *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. The definition is global and an error results if the *<function>* is already defined.

`\cs_new_nopar:Nn``\cs_new_nopar:(cn|Nx|cx)`

`\cs_new_nopar:Nn <function> {{code}}`

Creates *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. When the *<function>* is used the *<parameters>* absorbed cannot contain \par tokens. The definition is global and an error results if the *<function>* is already defined.

`\cs_new_protected:Nn``\cs_new_protected:(cn|Nx|cx)`

`\cs_new_protected:Nn <function> {{code}}`

Creates *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. The *<function>* will not expand within an x-type argument. The definition is global and an error results if the *<function>* is already defined.

`\cs_new_protected_nopar:Nn``\cs_new_protected_nopar:(cn|Nx|cx)`

`\cs_new_protected_nopar:Nn <function> {{code}}`

Creates *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. When the *<function>* is used the *<parameters>* absorbed cannot contain \par tokens. The *<function>* will not expand within an x-type or e-type argument. The definition is global and an error results if the *<function>* is already defined.

`\cs_set:Nn``\cs_set:(cn|Nx|cx)`

`\cs_set:Nn <function> {{code}}`

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the *<function>* is restricted to the current TeX group level.

`\cs_set_nopar:Nn``\cs_set_nopar:(cn|Nx|cx)`

`\cs_set_nopar:Nn <function> {{code}}`

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. When the *<function>* is used the *<parameters>* absorbed cannot contain \par tokens. The assignment of a meaning to the *<function>* is restricted to the current TeX group level.

`\cs_set_protected:Nn``\cs_set_protected:(cn|Nx|cx)`

`\cs_set_protected:Nn <function> {{code}}`

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. The *<function>* will not expand within an x-type argument. The assignment of a meaning to the *<function>* is restricted to the current TeX group level.

<code>\cs_set_protected_nopar:Nn</code>	<code>\cs_set_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_set_protected_nopar:(cn Nx cx)</code>	Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. When the <i><function></i> is used the <i><parameters></i> absorbed cannot contain \par tokens. The <i><function></i> will not expand within an x-type or e-type argument. The assignment of a meaning to the <i><function></i> is restricted to the current TeX group level.
<code>\cs_gset:Nn</code>	<code>\cs_gset:Nn <function> {<code>}</code>
<code>\cs_gset:(cn Nx cx)</code>	Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the <i><function></i> is global.
<code>\cs_gset_nopar:Nn</code>	<code>\cs_gset_nopar:Nn <function> {<code>}</code>
<code>\cs_gset_nopar:(cn Nx cx)</code>	Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. When the <i><function></i> is used the <i><parameters></i> absorbed cannot contain \par tokens. The assignment of a meaning to the <i><function></i> is global.
<code>\cs_gset_protected:Nn</code>	<code>\cs_gset_protected:Nn <function> {<code>}</code>
<code>\cs_gset_protected:(cn Nx cx)</code>	Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. The <i><function></i> will not expand within an x-type argument. The assignment of a meaning to the <i><function></i> is global.
<code>\cs_gset_protected_nopar:Nn</code>	<code>\cs_gset_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_gset_protected_nopar:(cn Nx cx)</code>	Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. When the <i><function></i> is used the <i><parameters></i> absorbed cannot contain \par tokens. The <i><function></i> will not expand within an x-type or e-type argument. The assignment of a meaning to the <i><function></i> is global.
<code>\cs_generate_from_arg_count>NNnn</code>	<code>\cs_generate_from_arg_count>NNnn <function> <creator> {<number>} {<code>}</code>
<code>\cs_generate_from_arg_count:(cNnn Ncnn)</code>	Uses the <i><creator></i> function (which should have signature Npn, for example <code>\cs_new:Npn</code>) to define a <i><function></i> which takes <i><number></i> arguments and has <i><code></i> as replacement text. The <i><number></i> of arguments is an integer expression, evaluated as detailed for <code>\int_eval:n</code> .

Updated: 2012-01-14

3.4 Copying control sequences

Control sequences (not just functions as defined above) can be set to have the same meaning using the functions described here. Making two control sequences equivalent means that the second control sequence is a *copy* of the first (rather than a pointer to it). Thus the old and new control sequence are not tied together: changes to one are not reflected in the other.

In the following text “cs” is used as an abbreviation for “control sequence”:

```
\cs_new_eq:NN  
\cs_new_eq:(Nc|cN|cc)
```

```
\cs_new_eq:NN <cs1> <cs2>  
\cs_new_eq:NN <cs1> <token>
```

Globally creates *<control sequence₁>* and sets it to have the same meaning as *<control sequence₂>* or *<token>*. The second control sequence may subsequently be altered without affecting the copy.

```
\cs_set_eq:NN  
\cs_set_eq:(Nc|cN|cc)
```

```
\cs_set_eq:NN <cs1> <cs2>  
\cs_set_eq:NN <cs1> <token>
```

Sets *<control sequence₁>* to have the same meaning as *<control sequence₂>* (or *<token>*). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the *<control sequence₁>* is restricted to the current TeX group level.

```
\cs_gset_eq:NN  
\cs_gset_eq:(Nc|cN|cc)
```

```
\cs_gset_eq:NN <cs1> <cs2>  
\cs_gset_eq:NN <cs1> <token>
```

Globally sets *<control sequence₁>* to have the same meaning as *<control sequence₂>* (or *<token>*). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the *<control sequence₁>* is *not* restricted to the current TeX group level: the assignment is global.

3.5 Deleting control sequences

There are occasions where control sequences need to be deleted. This is handled in a very simple manner.

```
\cs_undefine:N  
\cs_undefine:c
```

Updated: 2011-09-15

```
\cs_undefine:N <control sequence>
```

Sets *<control sequence>* to be globally undefined.

3.6 Showing control sequences

```
\cs_meaning:N *
```

```
\cs_meaning:c *
```

Updated: 2011-12-22

```
\cs_meaning:N <control sequence>
```

This function expands to the *meaning* of the *<control sequence>* control sequence. For a macro, this includes the *<replacement text>*.

TeXhackers note: This is TeX’s `\meaning` primitive. For tokens that are not control sequences, it is more logical to use `\token_to_meaning:N`. The `c` variant correctly reports undefined arguments.

```
\cs_show:N  
\cs_show:c
```

Updated: 2017-02-14

```
\cs_show:N <control sequence>
```

Displays the definition of the *<control sequence>* on the terminal.

TeXhackers note: This is similar to the TeX primitive `\show`, wrapped to a fixed number of characters per line.

```
\cs_log:N  
\cs_log:c
```

New: 2014-08-22
Updated: 2017-02-14

```
\cs_log:N <control sequence>
```

Writes the definition of the *<control sequence>* in the log file. See also `\cs_show:N` which displays the result in the terminal.

3.7 Converting to and from control sequences

```
\use:c *
```

```
\use:c {<control sequence name>}
```

Expands the *<control sequence name>* until only characters remain, and then converts this into a control sequence. This process requires two expansions. As in other c-type arguments the *<control sequence name>* must, when fully expanded, consist of character tokens, typically a mixture of category code 10 (space), 11 (letter) and 12 (other).

TeXhackers note: Protected macros that appear in a c-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

As an example of the `\use:c` function, both

```
\use:c { a b c }
```

and

```
\tl_new:N \l_my_tl  
\tl_set:Nn \l_my_tl { a b c }  
\use:c { \tl_use:N \l_my_tl }
```

would be equivalent to

```
\abc
```

after two expansions of `\use:c`.

```
\cs_if_exist_use:N *  
\cs_if_exist_use:c *  
\cs_if_exist_use:NTF *  
\cs_if_exist_use:cTF *
```

New: 2012-11-10

```
\cs_if_exist_use:N <control sequence>  
\cs_if_exist_use:NTF <control sequence> {<true code>} {<false code>}
```

Tests whether the *<control sequence>* is currently defined according to the conditional `\cs_if_exist:NTF` (whether as a function or another control sequence type), and if it is inserts the *<control sequence>* into the input stream followed by the *<true code>*. Otherwise the *<false code>* is used.

```
\cs:w      *
\cs_end: *
```

Converts the given *<control sequence name>* into a single control sequence token. This process requires one expansion. The content for *<control sequence name>* may be literal material or from other expandable functions. The *<control sequence name>* must, when fully expanded, consist of character tokens which are not active: typically of category code 10 (space), 11 (letter) or 12 (other), or a mixture of these.

TeXhackers note: These are the TeX primitives `\csname` and `\endcsname`.

As an example of the `\cs:w` and `\cs_end:` functions, both

```
\cs:w a b c \cs_end:
```

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\cs:w \tl_use:N \l_my_tl \cs_end:
```

would be equivalent to

```
\abc
```

after one expansion of `\cs:w`.

```
\cs_to_str:N *
```

```
\cs_to_str:N <control sequence>
```

Converts the given *<control sequence>* into a series of characters with category code 12 (other), except spaces, of category code 10. The result does *not* include the current escape token, contrarily to `\token_to_str:N`. Full expansion of this function requires exactly 2 expansion steps, and so an x-type or e-type expansion, or two o-type expansions are required to convert the *<control sequence>* to a sequence of characters in the input stream. In most cases, an f-expansion is correct as well, but this loses a space at the start of the result.

4 Analysing control sequences

```
\cs_split_function:N *
```

```
\cs_split_function:N <function>
```

New: 2018-04-06

Splits the *<function>* into the *<name>* (*i.e.* the part before the colon) and the *<signature>* (*i.e.* after the colon). This information is then placed in the input stream in three parts: the *<name>*, the *<signature>* and a logic token indicating if a colon was found (to differentiate variables from function names). The *<name>* does not include the escape character, and both the *<name>* and *<signature>* are made up of tokens with category code 12 (other).

The next three functions decompose TeX macros into their constituent parts: if the *<token>* passed is not a macro then no decomposition can occur. In the latter case, all three functions leave `\scan_stop:` in the input stream.

\cs_prefix_spec:N *New: 2019-02-27

\cs_prefix_spec:N *token*

If the *token* is a macro, this function leaves the applicable TeX prefixes in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_prefix_spec:N \next:nn
```

leaves `\long` in the input stream. If the *token* is not a macro then `\scan_stop:` is left in the input stream.

TeXhackers note: The prefix can be empty, `\long`, `\protected` or `\protected\long` with backslash replaced by the current escape character.

\cs_argument_spec:N *New: 2019-02-27

\cs_argument_spec:N *token*

If the *token* is a macro, this function leaves the primitive TeX argument specification in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1 y #2 }
\cs_argument_spec:N \next:nn
```

leaves `#1#2` in the input stream. If the *token* is not a macro then `\scan_stop:` is left in the input stream.

TeXhackers note: If the argument specification contains the string `->`, then the function produces incorrect results.

\cs_replacement_spec:N *New: 2019-02-27

\cs_replacement_spec:N *token*

If the *token* is a macro, this function leaves the replacement text in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_replacement_spec:N \next:nn
```

leaves `x#1~y#2` in the input stream. If the *token* is not a macro then `\scan_stop:` is left in the input stream.

TeXhackers note: If the argument specification contains the string `->`, then the function produces incorrect results.

5 Using or removing tokens and arguments

Tokens in the input can be read and used or read and discarded. If one or more tokens are wrapped in braces then when absorbing them the outer set is removed. At the same time, the category code of each token is set when the token is read by a function (if it

is read more than once, the category code is determined by the situation in force when first function absorbs the token).

\use:n	*	\use:n {group ₁ }
\use:nn	*	\use:nn {group ₁ } {group ₂ }
\use:nnn	*	\use:nnn {group ₁ } {group ₂ } {group ₃ }
\use:nnnn	*	\use:nnnn {group ₁ } {group ₂ } {group ₃ } {group ₄ }

As illustrated, these functions absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument are removed and the remaining tokens are left in the input stream. The category code of these tokens is also fixed by this process (if it has not already been by some other absorption). All of these functions require only a single expansion to operate, so that one expansion of

```
\use:nn { abc } { def }
```

results in the input stream containing

```
abc { def }
```

i.e. only the outer braces are removed.

TeXhackers note: The \use:n function is equivalent to L^AT_EX 2_ε's \@firstofone.

\use_i:nn	*	\use_i:nn {arg ₁ } {arg ₂ }
\use_ii:nn	*	

These functions absorb two arguments from the input stream. The function \use_i:nn discards the second argument, and leaves the content of the first argument in the input stream. \use_ii:nn discards the first argument and leaves the content of the second argument in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

TeXhackers note: These are equivalent to L^AT_EX 2_ε's \@firstoftwo and \@secondoftwo.

\use_i:nnn	*	\use_i:nnn {arg ₁ } {arg ₂ } {arg ₃ }
\use_ii:nnn	*	
\use_iii:nnn	*	

These functions absorb three arguments from the input stream. The function \use_i:nnn discards the second and third arguments, and leaves the content of the first argument in the input stream. \use_ii:nnn and \use_iii:nnn work similarly, leaving the content of second or third arguments in the input stream, respectively. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

\use_i:nnnn	*	\use_i:nnnn {arg ₁ } {arg ₂ } {arg ₃ } {arg ₄ }
\use_ii:nnnn	*	
\use_iii:nnnn	*	
\use_iv:nnnn	*	

These functions absorb four arguments from the input stream. The function \use_i:nnnn discards the second, third and fourth arguments, and leaves the content of the first argument in the input stream. \use_ii:nnnn, \use_iii:nnnn and \use_iv:nnnn work similarly, leaving the content of second, third or fourth arguments in the input stream, respectively. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

`\use_i_ii:nnn` *

`\use_i_ii:nnn {<arg1>} {<arg2>} {<arg3>}`

This function absorbs three arguments and leaves the content of the first and second in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect. An example:

```
\use_i_ii:nnn { abc } { { def } } { ghi }
```

results in the input stream containing

```
abc { def }
```

i.e. the outer braces are removed and the third group is removed.

`\use_ii_i:nn` *

New: 2019-06-02

`\use_ii_i:nn {<arg1>} {<arg2>}`

This function absorbs two arguments and leaves the content of the second and first in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect.

`\use_none:n` *

`\use_none:nn` *

`\use_none:nnn` *

`\use_none:nnnn` *

`\use_none:nnnnn` *

`\use_none:nnnnnn` *

`\use_none:nnnnnnn` *

`\use_none:nnnnnnnn` *

`\use_none:n {<group1>}`

These functions absorb between one and nine groups from the input stream, leaving nothing on the resulting input stream. These functions work after a single expansion. One or more of the `n` arguments may be an unbraced single token (i.e. an `N` argument).

TEXhackers note: These are equivalent to L^AT_EX 2 _{ε} 's `\@gobble`, `\@gobbletwo`, etc.

`\use:e` *

New: 2018-06-18

`\use:e {<expandable tokens>}`

Fully expands the `<token list>` in an `x`-type manner, but the function remains fully expandable, and parameter character (usually `#`) need not be doubled.

TEXhackers note: `\use:e` is a wrapper around the primitive `\expanded` where it is available: it requires two expansions to complete its action. When `\expanded` is not available this function is very slow.

`\use:x`

Updated: 2011-12-31

`\use:x {<expandable tokens>}`

Fully expands the `<expandable tokens>` and inserts the result into the input stream at the current location. Any hash characters (`#`) in the argument must be doubled.

5.1 Selecting tokens from delimited arguments

A different kind of function for selecting tokens from the token stream are those that use delimited arguments.

<code>\use_none_delimit_by_q_nil:w</code>	★	<code>\use_none_delimit_by_q_nil:w <balanced text> \q_nil</code>
<code>\use_none_delimit_by_q_stop:w</code>	★	<code>\use_none_delimit_by_q_stop:w <balanced text> \q_stop</code>
<code>\use_none_delimit_by_q_recursion_stop:w</code>	★	<code>\use_none_delimit_by_q_recursion_stop:w <balanced text> \q_recursion_stop</code>

Absorb the *<balanced text>* from the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

<code>\use_i_delimit_by_q_nil:nw</code>	★	<code>\use_i_delimit_by_q_nil:nw {<inserted tokens>} <balanced text></code>
<code>\use_i_delimit_by_q_stop:nw</code>	★	<code>\q_nil</code>
<code>\use_i_delimit_by_q_recursion_stop:nw</code>	★	<code>\use_i_delimit_by_q_stop:nw {<inserted tokens>} <balanced text> \q_stop</code> <code>\use_i_delimit_by_q_recursion_stop:nw {<inserted tokens>} <balanced text> \q_recursion_stop</code>

Absorb the *<balanced text>* from the input stream delimited by the marker given in the function name, leaving *<inserted tokens>* in the input stream for further processing.

6 Predicates and conditionals

LATEX3 has three concepts for conditional flow processing:

Branching conditionals Functions that carry out a test and then execute, depending on its result, either the code supplied as the *<true code>* or the *<false code>*. These arguments are denoted with T and F, respectively. An example would be

```
\cs_if_free:cTF {abc} {<true code>} {<false code>}
```

a function that turns the first argument into a control sequence (since it's marked as c) then checks whether this control sequence is still free and then depending on the result carries out the code in the second argument (true case) or in the third argument (false case).

These type of functions are known as “conditionals”; whenever a TF function is defined it is usually accompanied by T and F functions as well. These are provided for convenience when the branch only needs to go a single way. Package writers are free to choose which types to define but the kernel definitions always provide all three versions.

Important to note is that these branching conditionals with *<true code>* and/or *<false code>* are always defined in a way that the code of the chosen alternative can operate on following tokens in the input stream.

These conditional functions may or may not be fully expandable, but if they are expandable they are accompanied by a “predicate” for the same test as described below.

Predicates “Predicates” are functions that return a special type of boolean value which can be tested by the boolean expression parser. All functions of this type are expandable and have names that end with _p in the description part. For example,

```
\cs_if_free_p:N
```

would be a predicate function for the same type of test as the conditional described above. It would return “true” if its argument (a single token denoted by N) is still free for definition. It would be used in constructions like

```
\bool_if:nTF {
    \cs_if_free_p:N \l_tmpz_t1 || \cs_if_free_p:N \g_tmpz_t1
} {true code} {false code}
```

For each predicate defined, a “branching conditional” also exists that behaves like a conditional described above.

Primitive conditionals There is a third variety of conditional, which is the original concept used in plain TeX and L^AT_EX 2_ε. Their use is discouraged in `expl3` (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

<code>\c_true_bool</code>	Constants that represent <code>true</code> and <code>false</code> , respectively. Used to implement predicates.
<code>\c_false_bool</code>	

6.1 Tests on control sequences

<code>\cs_if_eq_p:NN *</code>	<code>\cs_if_eq_p:NNN <cs₁> <cs₂></code>
<code>\cs_if_eq:NNTF *</code>	<code>\cs_if_eq:NNTF <cs₁> <cs₂> {<i>true code</i>} {<i>false code</i>}</code>

Compares the definition of two *(control sequences)* and is logically `true` if they are the same, *i.e.* if they have exactly the same definition when examined with `\cs_show:N`.

<code>\cs_if_exist_p:N *</code>	<code>\cs_if_exist_p:N <control sequence></code>
<code>\cs_if_exist_p:c *</code>	<code>\cs_if_exist:NTF <control sequence> {<i>true code</i>} {<i>false code</i>}</code>
<code>\cs_if_exist:NTF *</code>	Tests whether the <i>(control sequence)</i> is currently defined (whether as a function or another control sequence type). Any definition of <i>(control sequence)</i> other than <code>\relax</code> evaluates as <code>true</code> .
<code>\cs_if_exist:cTF *</code>	

<code>\cs_if_free_p:N *</code>	<code>\cs_if_free_p:N <control sequence></code>
<code>\cs_if_free_p:c *</code>	<code>\cs_if_free:NTF <control sequence> {<i>true code</i>} {<i>false code</i>}</code>
<code>\cs_if_free:NTF *</code>	Tests whether the <i>(control sequence)</i> is currently free to be defined. This test is <code>false</code> if the <i>(control sequence)</i> currently exists (as defined by <code>\cs_if_exist:N</code>).
<code>\cs_if_free:cTF *</code>	

6.2 Primitive conditionals

The ε-T_EX engine itself provides many different conditionals. Some expand whatever comes after them and others don’t. Hence the names for these underlying functions often contains a `:w` part but higher level functions are often available. See for instance `\int_compare_p:nNn` which is a wrapper for `\if_int_compare:w`.

Certain conditionals deal with specific data types like boxes and fonts and are described there. The ones described below are either the universal conditionals or deal with control sequences. We prefix primitive conditionals with `\if_`.

\if_true:	★ \if_true: <true code> \else: <false code> \fi:
\if_false:	★ \if_false: <true code> \else: <false code> \fi:
\else:	★ \reverse_if:N <primitive conditional>
\fi:	★ \if_true: always executes <true code>, while \if_false: always executes <false code>.
\reverse_if:N	★ \reverse_if:N reverses any two-way primitive conditional. \else: and \fi: delimit the branches of the conditional. The function \or: is documented in l3int and used in case switches.

TeXhackers note: These are equivalent to their corresponding TeX primitive conditionals; \reverse_if:N is ε-TEx's \unless.

\if_meaning:w	★ \if_meaning:w <arg ₁ > <arg ₂ > <true code> \else: <false code> \fi:
---------------	--

\if_meaning:w executes <true code> when <arg₁> and <arg₂> are the same, otherwise it executes <false code>. <arg₁> and <arg₂> could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.

TeXhackers note: This is TeX's \ifx.

\if:w	★ \if:w <token ₁ > <token ₂ > <true code> \else: <false code> \fi:
\if_charcode:w	★ \if_catcode:w <token ₁ > <token ₂ > <true code> \else: <false code> \fi:
\if_catcode:w	★

These conditionals expand any following tokens until two unexpandable tokens are left. If you wish to prevent this expansion, prefix the token in question with \exp_not:N. \if_catcode:w tests if the category codes of the two tokens are the same whereas \if:w tests if the character codes are identical. \if_charcode:w is an alternative name for \if:w.

\if_cs_exist:N	★ \if_cs_exist:N <cs> <true code> \else: <false code> \fi:
\if_cs_exist:w	★ \if_cs_exist:w <tokens> \cs_end: <true code> \else: <false code> \fi:

Check if <cs> appears in the hash table or if the control sequence that can be formed from <tokens> appears in the hash table. The latter function does not turn the control sequence in question into \scan_stop!: This can be useful when dealing with control sequences which cannot be entered as a single token.

\if_mode_horizontal:	★ \if_mode_horizontal: <true code> \else: <false code> \fi:
\if_mode_vertical:	★
\if_mode_math:	★
\if_mode_inner:	★

Execute <true code> if currently in horizontal mode, otherwise execute <false code>. Similar for the other functions.

7 Starting a paragraph

\mode_leave_vertical:

New: 2017-07-04

\mode_leave_vertical:

Ensures that TeX is not in vertical (inter-paragraph) mode. In horizontal or math mode this command has no effect, in vertical mode it switches to horizontal mode, and inserts a box of width \parindent, followed by the \everypar token list.

TeXhackers note: This results in the contents of the \everypar token register being inserted, after \mode_leave_vertical: is complete. Notice that in contrast to the L^AT_EX 2_ε \leavevmode approach, no box is used by the method implemented here.

7.1 Debugging support

\debug_on:n
\debug_off:n

New: 2017-07-16
Updated: 2017-08-02

\debug_on:n { <comma-separated list> }
\debug_off:n { <comma-separated list> }

Turn on and off within a group various debugging code, some of which is also available as expl3 load-time options. The items that can be used in the <list> are

- **check-declarations** that checks all expl3 variables used were previously declared and that local/global variables (based on their name or on their first assignment) are only locally/globally assigned;
- **check-expressions** that checks integer, dimension, skip, and muskip expressions are not terminated prematurely;
- **deprecation** that makes soon-to-be-deprecated commands produce errors;
- **log-functions** that logs function definitions;
- **all** that does all of the above.

Providing these as switches rather than options allows testing code even if it relies on other packages: load all other packages, call \debug_on:n, and load the code that one is interested in testing. These functions can only be used in L^AT_EX 2_ε package mode loaded with enable-debug or another option implying it.

\debug_suspend:
\debug_resume:

New: 2017-11-28

\debug_suspend: ... \debug_resume:

SUPPRESS (locally) errors and logging from debug commands, except for the deprecation errors or warnings. These pairs of commands can be nested. This can be used around pieces of code that are known to fail checks, if such failures should be ignored. See for instance l3coffins.

Part V

The **I3expan** package

Argument expansion

This module provides generic methods for expanding TeX arguments in a systematic manner. The functions in this module all have prefix `exp`.

Not all possible variations are implemented for every base function. Instead only those that are used within the LATEX3 kernel or otherwise seem to be of general interest are implemented. Consult the module description to find out which functions are actually defined. The next section explains how to define missing variants.

1 Defining new variants

The definition of variant forms for base functions may be necessary when writing new functions or when applying a kernel function in a situation that we haven't thought of before.

Internally preprocessing of arguments is done with functions of the form `\exp_...`. They all look alike, an example would be `\exp_args:NNo`. This function has three arguments, the first and the second are a single tokens, while the third argument should be given in braces. Applying `\exp_args:NNo` expands the content of third argument once before any expansion of the first and second arguments. If `\seq_gpush:N` was not defined it could be coded in the following way:

```
\exp_args:NNo \seq_gpush:Nn
  \g_file_name_stack
  { \l_tmpa_t1 }
```

In other words, the first argument to `\exp_args:NNo` is the base function and the other arguments are preprocessed and then passed to this base function. In the example the first argument to the base function should be a single token which is left unchanged while the second argument is expanded once. From this example we can also see how the variants are defined. They just expand into the appropriate `\exp_` function followed by the desired base function, *e.g.*

```
\cs_generate_variant:Nn \seq_gpush:Nn { No }
```

results in the definition of `\seq_gpush:N`

```
\cs_new:Npn \seq_gpush:N { \exp_args:NNo \seq_gpush:Nn }
```

Providing variants in this way in style files is safe as the `\cs_generate_variant:Nn` function will only create new definitions if there is not already one available. Therefore adding such definition to later releases of the kernel will not make such style files obsolete.

The steps above may be automated by using the function `\cs_generate_variant:Nn`, described next.

2 Methods for defining variants

We recall the set of available argument specifiers.

- `N` is used for single-token arguments while `c` constructs a control sequence from its name and passes it to a parent function as an `N`-type argument.
- Many argument types extract or expand some tokens and provide it as an `n`-type argument, namely a braced multiple-token argument: `V` extracts the value of a variable, `v` extracts the value from the name of a variable, `n` uses the argument as it is, `o` expands once, `f` expands fully the front of the token list, `e` and `x` expand fully all tokens (differences are explained later).
- A few odd argument types remain: `T` and `F` for conditional processing, otherwise identical to `n`-type arguments, `p` for the parameter text in definitions, `w` for arguments with a specific syntax, and `D` to denote primitives that should not be used directly.

\cs_generate_variant:Nn *<parent control sequence>* {*<variant argument specifiers>*}

This function is used to define argument-specifier variants of the *<parent control sequence>* for L^AT_EX3 code-level macros. The *<parent control sequence>* is first separated into the *<base name>* and *<original argument specifier>*. The comma-separated list of *<variant argument specifiers>* is then used to define variants of the *<original argument specifier>* if these are not already defined. For each *<variant>* given, a function is created that expands its arguments as detailed and passes them to the *<parent control sequence>*. So for example

```
\cs_set:Npn \foo:Nn #1#2 { code here }
\cs_generate_variant:Nn \foo:Nn { c }
```

creates a new function \foo:c which expands its first argument into a control sequence name and passes the result to \foo:Nn. Similarly

```
\cs_generate_variant:Nn \foo:Nn { NV , cV }
```

generates the functions \foo:NV and \foo:cV in the same way. The \cs_generate_variant:Nn function can only be applied if the *<parent control sequence>* is already defined. If the *<parent control sequence>* is protected or if the *<variant>* involves any x argument, then the *<parent control sequence>* is also protected. The *<variant>* is created globally, as is any \exp_args:N*<variant>* function needed to carry out the expansion.

Only n and N arguments can be changed to other types. The only allowed changes are

- c variant of an N parent;
- o, V, v, f, e, or x variant of an n parent;
- N, n, T, F, or p argument unchanged.

This means the *<parent>* of a *<variant>* form is always unambiguous, even in cases where both an n-type parent and an N-type parent exist, such as for \tl_count:n and \tl_count:N.

For backward compatibility it is currently possible to make n, o, V, v, f, e, or x-type variants of an N-type argument or N or c-type variants of an n-type argument. Both are deprecated. The first because passing more than one token to an N-type argument will typically break the parent function's code. The second because programmers who use that most often want to access the value of a variable given its name, hence should use a V-type or v-type variant instead of c-type. In those cases, using the lower-level \exp_args:No or \exp_args:Nc functions explicitly is preferred to defining confusing variants.

3 Introducing the variants

The V type returns the value of a register, which can be one of t1,clist,int,skip,dim,muskip, or built-in T_EX registers. The v type is the same except it first creates a control sequence out of its argument before returning the value.

In general, the programmer should not need to be concerned with expansion control. When simply using the content of a variable, functions with a V specifier should be used. For those referred to by (cs)name, the v specifier is available for the same purpose. Only

when specific expansion steps are needed, such as when using delimited arguments, should the lower-level functions with `o` specifiers be employed.

The `e` type expands all tokens fully, starting from the first. More precisely the expansion is identical to that of TeX's `\message` (in particular `#` needs not be doubled). It was added in May 2018. In recent enough engines (starting around 2019) it relies on the primitive `\expanded` hence is fast. In older engines it is very much slower. As a result it should only be used in performance critical code if typical users will have a recent installation of the TeX ecosystem.

The `x` type expands all tokens fully, starting from the first. In contrast to `e`, all macro parameter characters `#` must be doubled, and omitting this leads to low-level errors. In addition this type of expansion is not expandable, namely functions that have `x` in their signature do not themselves expand when appearing inside `x` or `e` expansion.

The `f` type is so special that it deserves an example. It is typically used in contexts where only expandable commands are allowed. Then `x`-expansion cannot be used, and `f`-expansion provides an alternative that expands the front of the token list as much as can be done in such contexts. For instance, say that we want to evaluate the integer expression `3 + 4` and pass the result `7` as an argument to an expandable function `\example:n`. For this, one should define a variant using `\cs_generate_variant:Nn \example:n { f }`, then do

```
\example:f { \int_eval:n { 3 + 4 } }
```

Note that `x`-expansion would also expand `\int_eval:n` fully to its result `7`, but the variant `\example:x` cannot be expandable. Note also that `o`-expansion would not expand `\int_eval:n` fully to its result since that function requires several expansions. Besides the fact that `x`-expansion is protected rather than expandable, another difference between `f`-expansion and `x`-expansion is that `f`-expansion expands tokens from the beginning and stops as soon as a non-expandable token is encountered, while `x`-expansion continues expanding further tokens. Thus, for instance

```
\example:f { \int_eval:n { 1 + 2 } , \int_eval:n { 3 + 4 } }
```

results in the call

```
\example:n { 3 , \int_eval:n { 3 + 4 } }
```

while using `\example:x` or `\example:e` instead results in

```
\example:n { 3 , 7 }
```

at the cost of being protected (for `x` type) or very much slower in old engines (for `e` type). If you use `f` type expansion in conditional processing then you should stick to using TF type functions only as the expansion does not finish any `\if... \fi:` itself!

It is important to note that both `f`- and `o`-type expansion are concerned with the expansion of tokens from left to right in their arguments. In particular, `o`-type expansion applies to the first *token* in the argument it receives: it is conceptually similar to

```
\exp_after:wN <base function> \exp_after:wN { <argument> }
```

At the same time, `f`-type expansion stops at the *first* non-expandable token. This means for example that both

```
\tl_set:No \l_tmpa_tl { \g_tmpb_tl }
```

and

```
\tl_set:Nf \l_tmpa_tl { { \g_tmpb_tl } }
```

leave `\g_tmpb_tl` unchanged: `{` is the first token in the argument and is non-expandable.

It is usually best to keep the following in mind when using variant forms.

- Variants with `x`-type arguments (that are fully expanded before being passed to the `n`-type base function) are never expandable even when the base function is. Such variants cannot work correctly in arguments that are themselves subject to expansion. Consider using `f` or `e` expansion.
- In contrast, `e` expansion (full expansion, almost like `x` except for the treatment of `#`) does not prevent variants from being expandable (if the base function is). The drawback is that `e` expansion is very much slower in old engines (before 2019). Consider using `f` expansion if that type of expansion is sufficient to perform the required expansion, or `x` expansion if the variant will not itself need to be expandable.
- Finally `f` expansion only expands the front of the token list, stopping at the first non-expandable token. This may fail to fully expand the argument.

When speed is essential (for functions that do very little work and whose variants are used numerous times in a document) the following considerations apply because internal functions for argument expansion come in two flavours, some faster than others.

- Arguments that might need expansion should come first in the list of arguments.
- Arguments that should consist of single tokens `N`, `c`, `V`, or `v` should come first among these.
- Arguments that appear after the first multi-token argument `n`, `f`, `e`, or `o` require slightly slower special processing to be expanded. Therefore it is best to use the optimized functions, namely those that contain only `N`, `c`, `V`, and `v`, and, in the last position, `o`, `f`, `e`, with possible trailing `N` or `n` or `T` or `F`, which are not expanded. Any `x`-type argument causes slightly slower processing.

4 Manipulating the first argument

These functions are described in detail: expansion of multiple tokens follows the same rules but is described in a shorter fashion.

```
\exp_args:Nc *
\exp_args:cc *
```

This function absorbs two arguments (the `<function>` name and the `<tokens>`). The `<tokens>` are expanded until only characters remain, and are then turned into a control sequence. The result is inserted into the input stream *after* reinsertion of the `<function>`. Thus the `<function>` may take more than one argument: all others are left unchanged.

The `:cc` variant constructs the `<function>` name in the same manner as described for the `<tokens>`.

TeXhackers note: Protected macros that appear in a `c`-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

`\exp_args:N` *

`\exp_args:N` *function* {*tokens*} ...

This function absorbs two arguments (the *function* name and the *tokens*). The *tokens* are expanded once, and the result is inserted in braces into the input stream *after* reinsertion of the *function*. Thus the *function* may take more than one argument: all others are left unchanged.

`\exp_args:NV` *

`\exp_args:NV` *function* *variable*

This function absorbs two arguments (the names of the *function* and the *variable*). The content of the *variable* are recovered and placed inside braces into the input stream *after* reinsertion of the *function*. Thus the *function* may take more than one argument: all others are left unchanged.

`\exp_args:Nv` *

`\exp_args:Nv` *function* {*tokens*}

This function absorbs two arguments (the *function* name and the *tokens*). The *tokens* are expanded until only characters remain, and are then turned into a control sequence. This control sequence should be the name of a *variable*. The content of the *variable* are recovered and placed inside braces into the input stream *after* reinsertion of the *function*. Thus the *function* may take more than one argument: all others are left unchanged.

TeXhackers note: Protected macros that appear in a v-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

`\exp_args:Ne` *

New: 2018-05-15

`\exp_args:Ne` *function* {*tokens*}

This function absorbs two arguments (the *function* name and the *tokens*) and exhaustively expands the *tokens*. The result is inserted in braces into the input stream *after* reinsertion of the *function*. Thus the *function* may take more than one argument: all others are left unchanged.

TeXhackers note: This relies on the `\expanded` primitive when available (in LuaTeX and starting around 2019 in other engines). Otherwise it uses some fall-back code that is very much slower. As a result it should only be used in performance-critical code if typical users have a recent installation of the TeX ecosystem.

`\exp_args:Nf` *

`\exp_args:Nf` *function* {*tokens*}

This function absorbs two arguments (the *function* name and the *tokens*). The *tokens* are fully expanded until the first non-expandable token is found (if that is a space it is removed), and the result is inserted in braces into the input stream *after* reinsertion of the *function*. Thus the *function* may take more than one argument: all others are left unchanged.

```
\exp_args:Nx * \exp_args:Nx <function> {<tokens>}
```

This function absorbs two arguments (the *<function>* name and the *<tokens>*) and exhaustively expands the *<tokens>*. The result is inserted in braces into the input stream *after* reinsertion of the *<function>*. Thus the *<function>* may take more than one argument: all others are left unchanged.

5 Manipulating two arguments

```
\exp_args:NNc * \exp_args:NNc <token1> <token2> {<tokens>}
```

These optimized functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments.

```
\exp_args:NNo *  
\exp_args:NNV *  
\exp_args:NNv *  
\exp_args:NNe *  
\exp_args:NNf *  
\exp_args:Ncc *  
\exp_args:Nco *  
\exp_args:NcV *  
\exp_args:Ncv *  
\exp_args:Ncf *  
\exp_args:NVV *
```

Updated: 2018-05-15

```
\exp_args:Nnc * \exp_args:Noo <token> {<tokens1>} {<tokens2>}
```

These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions need slower processing.

```
\exp_args:Nno *  
\exp_args:NnV *  
\exp_args:Nnv *  
\exp_args:Nne *  
\exp_args:Nnf *  
\exp_args:Noc *  
\exp_args:Noo *  
\exp_args:Nof *  
\exp_args:NVo *  
\exp_args:Nfo *  
\exp_args:Nff *  
\exp_args:Nee *
```

Updated: 2018-05-15

```
\exp_args>NNx * \exp_args>NNx <token1> <token2> {<tokens>}
```

These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions are not expandable due to their x-type argument.

6 Manipulating three arguments

```
\exp_args:NNNo *
\exp_args:NNNV *
\exp_args:NNNv *
\exp_args:Nccc *
\exp_args:NcNc *
\exp_args:NcNo *
\exp_args:Ncco *
```

`\exp_args:NNNo <token1> <token2> <token3> {<tokes>}`

These optimized functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.*

```
\exp_args:NNcf *
\exp_args:NNno *
\exp_args:NNnV *
\exp_args:NNoo *
\exp_args:NNVV *
\exp_args:Ncno *
\exp_args:NcnV *
\exp_args:Ncoo *
\exp_args:NcVV *
\exp_args:Nnnnc *
\exp_args:Nnno *
\exp_args:Nnnf *
\exp_args:Nnff *
\exp_args:Nooo *
\exp_args:Noof *
\exp_args:Nffo *
\exp_args:Neee *
```

`\exp_args:NNoo <token1> <token2> {<token3>} {<tokes>}`

These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.* These functions need slower processing.

```
\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Noox
```

`\exp_args:NNnx <token1> <token2> {<tokes1>} {<tokes2>}`

These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.*

New: 2015-08-12

7 Unbraced expansion

\exp_last_unbraced:N	*	\exp_last_unbraced:Nno <i><token></i> {\i<tokens ₁ >} {\i<tokens ₂ >}
\exp_last_unbraced:NV	*	These functions absorb the number of arguments given by their specification, carry out the expansion indicated and leave the results in the input stream, with the last argument not surrounded by the usual braces. Of these, the :Nno, :Noo, :Nf and :NnNo variants need slower processing.
\exp_last_unbraced:Nv	*	
\exp_last_unbraced:Ne	*	
\exp_last_unbraced:Nf	*	
\exp_last_unbraced:NNo	*	
\exp_last_unbraced:NNV	*	
\exp_last_unbraced:NNf	*	
\exp_last_unbraced:Nco	*	
\exp_last_unbraced:NcV	*	
\exp_last_unbraced:Nno	*	
\exp_last_unbraced:Noo	*	
\exp_last_unbraced:Nfo	*	
\exp_last_unbraced:NNNo	*	
\exp_last_unbraced:NNNV	*	
\exp_last_unbraced:NNNf	*	
\exp_last_unbraced:NnNo	*	
\exp_last_unbraced:NNNNo	*	
\exp_last_unbraced:NNNNf	*	

Updated: 2018-05-15

\exp_last_unbraced:Nx	\exp_last_unbraced:Nx <i><function></i> {\i<tokens>}
-----------------------	--

This function fully expands the *<tokens>* and leaves the result in the input stream after reinsertion of the *<function>*. This function is not expandable.

\exp_last_two_unbraced:Noo	*	\exp_last_two_unbraced:Noo <i><token></i> {\i<tokens ₁ >} {\i<tokens ₂ >}
----------------------------	---	---

This function absorbs three arguments and expands the second and third once. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments, which are not wrapped in braces. This function needs special (slower) processing.

\exp_after:wN	*	\exp_after:wN <i><token₁></i> <i><token₂></i>
---------------	---	---

Carries out a single expansion of *<token₂>* (which may consume arguments) prior to the expansion of *<token₁>*. If *<token₂>* has no expansion (for example, if it is a character) then it is left unchanged. It is important to notice that *<token₁>* may be *any* single token, including group-opening and -closing tokens ({ or } assuming normal TeX category codes). Unless specifically required this should be avoided: expansion should be carried out using an appropriate argument specifier variant or the appropriate \exp_arg:N function.

TeXhackers note: This is the TeX primitive \expandafter renamed.

8 Preventing expansion

Despite the fact that the following functions are all about preventing expansion, they're designed to be used in an expandable context and hence are all marked as being 'expand-

able' since they themselves disappear after the expansion has completed.

\exp_not:N * `\exp_not:N <token>`

Prevents expansion of the *<token>* in a context where it would otherwise be expanded, for example an x-type argument or the first token in an o or e or f argument.

TeXhackers note: This is the TeX \noexpand primitive. It only prevents expansion. At the beginning of an f-type argument, a space *<token>* is removed even if it appears as \exp_not:N \c_space_token. In an x-expanding definition (\cs_new:Npx), a macro parameter introduces an argument even if it appears as \exp_not:N # 1. This differs from \exp_not:n.

\exp_not:c * `\exp_not:c {<tokens>}`

Expands the *<tokens>* until only characters remain, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited using \exp_not:N.

TeXhackers note: Protected macros that appear in a c-type argument are expanded despite being protected; \exp_not:n also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

\exp_not:n * `\exp_not:n {<tokens>}`

Prevents expansion of the *<tokens>* in an e or x-type argument. In all other cases the *<tokens>* continue to be expanded, for example in the input stream or in other types of arguments such as c, f, v. The argument of \exp_not:n must be surrounded by braces.

TeXhackers note: This is the ε-Tex \unexpanded primitive. In an x-expanding definition (\cs_new:Npx), \exp_not:n {#1} is equivalent to ##1 rather than to #1, namely it inserts the two characters # and 1. In an e-type argument \exp_not:n {#} is equivalent to #, namely it inserts the character #.

\exp_not:o * `\exp_not:o {<tokens>}`

Expands the *<tokens>* once, then prevents any further expansion in x-type or e-type arguments using \exp_not:n.

\exp_not:v * `\exp_not:v {variable}`

Recovering the content of the *variable*, then prevents expansion of this material in x-type or e-type arguments using \exp_not:n.

`\exp_not:v *` `\exp_not:v {<tokens>}`

Expands the $\langle tokens \rangle$ until only characters remains, and then converts this into a control sequence which should be a $\langle variable \rangle$ name. The content of the $\langle variable \rangle$ is recovered, and further expansion in x-type or e-type arguments is prevented using `\exp_not:n`.

TeXhackers note: Protected macros that appear in a v-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

`\exp_not:e *` `\exp_not:e {<tokens>}`

Expands $\langle tokens \rangle$ exhaustively, then protects the result of the expansion (including any tokens which were not expanded) from further expansion in e or x-type arguments using `\exp_not:n`. This is very rarely useful but is provided for consistency.

`\exp_not:f *` `\exp_not:f {<tokens>}`

Expands $\langle tokens \rangle$ fully until the first unexpandable token is found (if it is a space it is removed). Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion in x-type or e-type arguments using `\exp_not:n`.

`\exp_stop_f: *` `\foo_bar:f {<tokens>} \exp_stop_f: <more tokens> }`

Updated: 2011-06-03

This function terminates an f-type expansion. Thus if a function `\foo_bar:f` starts an f-type expansion and all of $\langle tokens \rangle$ are expandable `\exp_stop_f:` terminates the expansion of tokens even if $\langle more tokens \rangle$ are also expandable. The function itself is an implicit space token. Inside an x-type expansion, it retains its form, but when typeset it produces the underlying space (\sqcup).

9 Controlled expansion

The expl3 language makes all efforts to hide the complexity of T_EX expansion from the programmer by providing concepts that evaluate/expand arguments of functions prior to calling the “base” functions. Thus, instead of using many `\expandafter` calls and other trickery it is usually a matter of choosing the right variant of a function to achieve a desired result.

Of course, deep down T_EX is using expansion as always and there are cases where a programmer needs to control that expansion directly; typical situations are basic data manipulation tools. This section documents the functions for that level. These commands are used throughout the kernel code, but we hope that outside the kernel there will be little need to resort to them. Instead the argument manipulation methods document above should usually be sufficient.

While `\exp_after:wN` expands one token (out of order) it is sometimes necessary to expand several tokens in one go. The next set of commands provide this functionality. Be aware that it is absolutely required that the programmer has full control over the tokens to be expanded, i.e., it is not possible to use these functions to expand unknown input as part of $\langle expandable-tokens \rangle$ as that will break badly if unexpandable tokens are encountered in that place!

```
\exp:w      *
\exp_end: *
```

New: 2015-08-23

Expands *<expandable-tokens>* until reaching `\exp_end:` at which point expansion stops. The full expansion of *<expandable tokens>* has to be empty. If any token in *<expandable tokens>* or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end:` will be misinterpreted later on.³

In typical use cases the `\exp_end:` is hidden somewhere in the replacement text of *<expandable-tokens>* rather than being on the same expansion level than `\exp:w`, e.g., you may see code such as

```
\exp:w @@_case:NnTF #1 {#2} {} {}
```

where somewhere during the expansion of `\@@_case:NnTF` the `\exp_end:` gets generated.

TeXhackers note: The current implementation uses `\romannumeral` hence ignores space tokens and explicit signs + and - in the expansion of the *<expandable tokens>*, but this should not be relied upon.

```
\exp:w      *
\exp_end_continue_f:w *
```

New: 2015-08-23

Expands *<expandable-tokens>* until reaching `\exp_end_continue_f:w` (*further-tokens*)

Expands *<expandable-tokens>* until reaching `\exp_end_continue_f:w` at which point expansion continues as an f-type expansion expanding *<further-tokens>* until an unexpandable token is encountered (or the f-type expansion is explicitly terminated by `\exp_stop_f:`). As with all f-type expansions a space ending the expansion gets removed.

The full expansion of *<expandable-tokens>* has to be empty. If any token in *<expandable-tokens>* or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end_continue_f:w` will be misinterpreted later on.⁴

In typical use cases *<expandable-tokens>* contains no tokens at all, e.g., you will see code such as

```
\exp_after:wN { \exp:w \exp_end_continue_f:w #2 }
```

where the `\exp_after:wN` triggers an f-expansion of the tokens in `#2`. For technical reasons this has to happen using two tokens (if they would be hidden inside another command `\exp_after:wN` would only expand the command but not trigger any additional f-expansion).

You might wonder why there are two different approaches available, after all the effect of

```
\exp:w <expandable-tokens> \exp_end:
```

can be alternatively achieved through an f-type expansion by using `\exp_stop_f:`, i.e.

```
\exp:w \exp_end_continue_f:w <expandable-tokens> \exp_stop_f:
```

The reason is simply that the first approach is slightly faster (one less token to parse and less expansion internally) so in places where such performance really matters and where we want to explicitly stop the expansion at a defined point the first form is preferable.

³Due to the implementation you might get the character in position 0 in the current font (typically ““”) in the output without any error message!

⁴In this particular case you may get a character into the output as well as an error message.

```
\exp:w          *
\exp_end_continue_f:nw *
```

New: 2015-08-23

`\exp:w <expandable-tokens> \exp_end_continue_f:nw <further-tokens>`

The difference to `\exp_end_continue_f:w` is that we first pick up an argument which is then returned to the input stream. If `<further-tokens>` starts with space tokens then these space tokens are removed while searching for the argument. If it starts with a brace group then the braces are removed. Thus such spaces or braces will not terminate the f-type expansion.

10 Internal functions

```
\::n \cs_new:Npn \exp_args:Ncof { \::c \::o \::f \::: }
```

```
\::N
\::p
\::c
\::o
\::e
\::f
\::x
\::v
\::V
\:::
```

`\cs_new:Npn \exp_args:Ncof { \::c \::o \::f \::: }`

Internal forms for the base expansion types. These names do *not* conform to the general L^AT_EX3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

```
\::o_unbraced
\::e_unbraced
\::f_unbraced
\::x_unbraced
\::v_unbraced
\::V_unbraced
```

`\cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }`

Internal forms for the expansion types which leave the terminal argument unbraced. These names do *not* conform to the general L^AT_EX3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

Part VI

The **I3quark** package

Quarks

Two special types of constants in L^AT_EX3 are “quarks” and “scan marks”. By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`.

1 Quarks

Quarks are control sequences that expand to themselves and should therefore *never* be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, the most common use case being the ‘stop token’ (*i.e.* `\q_stop`). For example, when writing a macro to parse a user-defined date

```
\date_parse:n {19/June/1981}
```

one might write a command such as

```
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
{ <do something with the date> }
```

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function `\prop_get:NnN` to retrieve a value stored in some key of a property list, if the key does not exist then the return value is the quark `\q_no_value`. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using `\tl_if_eq:NNTF`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster. An example of the quark testing functions and their use in recursion can be seen in the implementation of `\clist_map_function:NN`.

2 Defining quarks

`\quark_new:N` *<quark>*

Creates a new *<quark>* which expands only to *<quark>*. The *<quark>* is defined globally, and an error message is raised if the name was already taken.

`\q_stop` Used as a marker for delimited arguments, such as

```
\cs_set:Npn \tmp:w #1#2 \q_stop {#1}
```

<u>\q_mark</u>	Used as a marker for delimited arguments when \q_stop is already in use.
<u>\q_nil</u>	Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself need to be tested (in contrast to \q_stop, which is only ever used as a delimiter).
<u>\q_no_value</u>	A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a “return” value by functions such as \prop_get:NnN if there is no data to return.

3 Quark tests

The method used to define quarks means that the single token (N) tests are faster than the multi-token (n) tests. The latter should therefore only be used when the argument can definitely take more than a single token.

<u>\quark_if_nil_p:N *</u>	\quark_if_nil_p:N <i><token></i>
<u>\quark_if_nil:NTF *</u>	\quark_if_nil:NTF <i><token></i> { <i>true code</i> } { <i>false code</i> }
	Tests if the <i><token></i> is equal to \q_nil.
<u>\quark_if_nil_p:n *</u>	\quark_if_nil_p:n { <i><token list></i> }
<u>\quark_if_nil_p:(o V) *</u>	\quark_if_nil:nTF { <i><token list></i> } { <i>true code</i> } { <i>false code</i> }
<u>\quark_if_nil:nTF *</u>	Tests if the <i><token list></i> contains only \q_nil (distinct from <i><token list></i> being empty or containing \q_nil plus one or more other tokens).
<u>\quark_if_no_value_p:N *</u>	\quark_if_no_value_p:N <i><token></i>
<u>\quark_if_no_value_p:c *</u>	\quark_if_no_value:NTF <i><token></i> { <i>true code</i> } { <i>false code</i> }
<u>\quark_if_no_value:NTF *</u>	Tests if the <i><token></i> is equal to \q_no_value.
<u>\quark_if_no_value_p:n *</u>	\quark_if_no_value_p:n { <i><token list></i> }
<u>\quark_if_no_value:nTF *</u>	\quark_if_no_value:nTF { <i><token list></i> } { <i>true code</i> } { <i>false code</i> }
	Tests if the <i><token list></i> contains only \q_no_value (distinct from <i><token list></i> being empty or containing \q_no_value plus one or more other tokens).

4 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below and an example is shown in Section 5.

<u>\q_recursion_tail</u>	This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.
--------------------------	---

\q_recursion_stop

This quark is added *after* the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

\quark_if_recursion_tail_stop:N * \quark_if_recursion_tail_stop:N *<token>*

Tests if *<token>* contains only the marker `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

\quark_if_recursion_tail_stop:n * \quark_if_recursion_tail_stop:n {*<token list>*}
\quark_if_recursion_tail_stop:o *

Updated: 2011-09-06

Tests if the *<token list>* contains only `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

\quark_if_recursion_tail_stop_do:Nn * \quark_if_recursion_tail_stop_do:Nn *<token>* {*<insertion>*}

Tests if *<token>* contains only the marker `\q_recursion_tail`, and if so uses `\use_i_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The *<insertion>* code is then added to the input stream after the recursion has ended.

\quark_if_recursion_tail_stop_do:nn * \quark_if_recursion_tail_stop_do:nn {*<token list>*} {*<insertion>*}
\quark_if_recursion_tail_stop_do:on *

Updated: 2011-09-06

Tests if the *<token list>* contains only `\q_recursion_tail`, and if so uses `\use_i_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The *<insertion>* code is then added to the input stream after the recursion has ended.

\quark_if_recursion_tail_break>NN * \quark_if_recursion_tail_break:nN {*<token list>*}
\quark_if_recursion_tail_break:nN * \<type>_map_break:

New: 2018-04-10

Tests if *<token list>* contains only `\q_recursion_tail`, and if so terminates the recursion using `\<type>_map_break::`. The recursion end should be marked by `\prg_break_point:Nn \<type>_map_break::`.

5 An example of recursion with quarks

Quarks are mainly used internally in the expl3 code to define recursion functions such as `\tl_map_inline:nn` and so on. Here is a small example to demonstrate how to

use quarks in this fashion. We shall define a command called `\my_map dbl:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map dbl:nn {abcd} {[--#1--#2--]~}` would produce “[*a*-*b*-] [*c*-*d*-]”. Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here’s the definition of `\my_map dbl:nn`. First of all, define the function that does the processing based on the inline function argument #2. Then initiate the recursion using an internal function. The token list #1 is terminated using `\q_recursion_tail`, with delimiters according to the type of recursion (here a pair of `\q_recursion_tail`), concluding with `\q_recursion_stop`. These quarks are used to mark the end of the token list being operated upon.

```
\cs_new:Npn \my_map dbl:nn #1#2
{
    \cs_set:Npn \__my_map dbl_fn:nn ##1 ##2 {#2}
    \__my_map dbl:nn #1 \q_recursion_tail \q_recursion_tail
    \q_recursion_stop
}
```

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

```
\cs_new:Nn \__my_map dbl:nn
{
    \quark_if_recursion_tail_stop:n {#1}
    \quark_if_recursion_tail_stop:n {#2}
    \__my_map dbl_fn:nn {#1} {#2}
```

Finally, recurse:

```
    \__my_map dbl:nn
}
```

Note that contrarily to L^AT_EX3 built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of `__my_map dbl_fn:nn`.

6 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence never expand in an expansion context and are (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by T_EX in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see `\l3regex`).

`\scan_new:N`

New: 2018-04-01

`\scan_new:N <scan mark>`

Creates a new *<scan mark>* which is set equal to `\scan_stop:`. The *<scan mark>* is defined globally, and an error message is raised if the name was already taken by another scan mark.

\s_stop Used at the end of a set of instructions, as a marker that can be jumped to using `\use-
none_delimit_by_s_stop:w`.
New: 2018-04-01

\use_none_delimit_by_s_stop:w * `\use_none_delimit_by_s_stop:w <tokens> \s_stop`
New: 2018-04-01

Removes the *<tokens>* and `\s_stop` from the input stream. This leads to a low-level T_EX error if `\s_stop` is absent.

Part VII

The **\3tl** package

Token lists

TE^X works with tokens, and **L**^A**T**^E₃ therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

```
\foo:n { a collection of \tokens }
```

or may be stored in a so-called “token list variable”, which have the suffix **tl**: a token list variable can also be used as the argument to a function, for example

```
\foo:N \l_some_tl
```

In both cases, functions are available to test and manipulate the lists of tokens, and these have the module prefix **tl**. In many cases, functions which can be applied to token list variables are paired with similar functions for application to explicit lists of tokens: the two “views” of a token list are therefore collected together here.

A token list (explicit, or stored in a variable) can be seen either as a list of “items”, or a list of “tokens”. An item is whatever **\use:n** would grab as its argument: a single non-space token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal **N** argument, or **,**, **{**, or **}** (assuming normal **T**E^X category codes). Thus for example

```
{ Hello } ~ world
```

contains six items (**Hello**, **w**, **o**, **r**, **l** and **d**), but thirteen tokens (**{**, **H**, **e**, **l**, **l**, **o**, **}**, **,**, **w**, **o**, **r**, **l** and **d**). Functions which act on items are often faster than their analogue acting directly on tokens.

1 Creating and initialising token list variables

```
\tl_new:N \tl_new:c
```

Creates a new ***tl var*** or raises an error if the name is already taken. The declaration is global. The ***tl var*** is initially empty.

```
\tl_const:Nn \tl_const:(Nx|cn|cx)
```

Creates a new constant ***tl var*** or raises an error if the name is already taken. The value of the ***tl var*** is set globally to the ***token list***.

```
\tl_clear:N \tl_clear:c
```

\tl_gclear:N **\tl_gclear:c**

Clears all entries from the ***tl var***.

<code>\tl_clear_new:N</code>	<code>\tl_clear_new:N <tl var></code>	Ensures that the $\langle tl\ var \rangle$ exists globally by applying <code>\tl_new:N</code> if necessary, then applies <code>\tl_(g)clear:N</code> to leave the $\langle tl\ var \rangle$ empty.
<code>\tl_set_eq:NN</code> <code>\tl_set_eq:(cN Nc cc)</code> <code>\tl_gset_eq:NN</code> <code>\tl_gset_eq:(cN Nc cc)</code>	<code>\tl_set_eq:NN <tl var₁> <tl var₂></code>	Sets the content of $\langle tl\ var_1 \rangle$ equal to that of $\langle tl\ var_2 \rangle$.
<code>\tl_concat:NNN</code> <code>\tl_concat:ccc</code> <code>\tl_gconcat:NNN</code> <code>\tl_gconcat:ccc</code>	<code>\tl_concat:NNN <tl var₁> <tl var₂> <tl var₃></code>	Concatenates the content of $\langle tl\ var_2 \rangle$ and $\langle tl\ var_3 \rangle$ together and saves the result in $\langle tl\ var_1 \rangle$. The $\langle tl\ var_2 \rangle$ is placed at the left side of the new token list.
New: 2012-05-18		
<code>\tl_if_exist_p:N *</code> <code>\tl_if_exist_p:c *</code> <code>\tl_if_exist:NTF *</code> <code>\tl_if_exist:cTF *</code>	<code>\tl_if_exist_p:N <tl var></code> <code>\tl_if_exist:NTF <tl var> {<true code>} {<false code>}</code>	Tests whether the $\langle tl\ var \rangle$ is currently defined. This does not check that the $\langle tl\ var \rangle$ really is a token list variable.
New: 2012-03-03		

2 Adding data to token list variables

<code>\tl_set:Nn</code>	<code>\tl_set:Nn <tl var> {<tokens>}</code>	Sets $\langle tl\ var \rangle$ to contain $\langle tokens \rangle$, removing any previous content from the variable.
<code>\tl_set:(NV Nv No Nf Nx cn cV cv co cf cx)</code>		
<code>\tl_gset:Nn</code>		
<code>\tl_gset:(NV Nv No Nf Nx cn cV cv co cf cx)</code>		
<code>\tl_put_left:Nn</code>	<code>\tl_put_left:Nn <tl var> {<tokens>}</code>	Appends $\langle tokens \rangle$ to the left side of the current content of $\langle tl\ var \rangle$.
<code>\tl_put_left:(NV No Nx cn cV co cx)</code>		
<code>\tl_gput_left:Nn</code>		
<code>\tl_gput_left:(NV No Nx cn cV co cx)</code>		
<code>\tl_put_right:Nn</code>	<code>\tl_put_right:Nn <tl var> {<tokens>}</code>	Appends $\langle tokens \rangle$ to the right side of the current content of $\langle tl\ var \rangle$.
<code>\tl_put_right:(NV No Nx cn cV co cx)</code>		
<code>\tl_gput_right:Nn</code>		
<code>\tl_gput_right:(NV No Nx cn cV co cx)</code>		

3 Modifying token list variables

```
\tl_replace_once:Nnn
\tl_replace_once:cnn
\tl_greplace_once:Nnn
\tl_greplace_once:cnn
```

Updated: 2011-08-11

```
\tl_replace_once:Nnn <tl var> {<old tokens>} {<new tokens>}
```

Replaces the first (leftmost) occurrence of *<old tokens>* in the *<tl var>* with *<new tokens>*. *<Old tokens>* cannot contain {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

```
\tl_replace_all:Nnn
\tl_replace_all:cnn
\tl_greplace_all:Nnn
\tl_greplace_all:cnn
```

Updated: 2011-08-11

```
\tl_replace_all:Nnn <tl var> {<old tokens>} {<new tokens>}
```

Replaces all occurrences of *<old tokens>* in the *<tl var>* with *<new tokens>*. *<Old tokens>* cannot contain {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern *<old tokens>* may remain after the replacement (see `\tl_remove_all:Nn` for an example).

```
\tl_remove_once:Nn
\tl_remove_once:cn
\tl_gremove_once:Nn
\tl_gremove_once:cn
```

Updated: 2011-08-11

```
\tl_remove_once:Nn <tl var> {<tokens>}
```

Removes the first (leftmost) occurrence of *<tokens>* from the *<tl var>*. *<Tokens>* cannot contain {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

```
\tl_remove_all:Nn
\tl_remove_all:cn
\tl_gremove_all:Nn
\tl_gremove_all:cn
```

Updated: 2011-08-11

```
\tl_remove_all:Nn <tl var> {<tokens>}
```

Removes all occurrences of *<tokens>* from the *<tl var>*. *<Tokens>* cannot contain {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern *<tokens>* may remain after the removal, for instance,

```
\tl_set:Nn \l_tmpa_tl {abcccd} \tl_remove_all:Nn \l_tmpa_tl {bc}
```

results in `\l_tmpa_tl` containing abcd.

4 Reassigning token list category codes

These functions allow the rescanning of tokens: re-apply TeX's tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes or using `\char_generate:nn`).

```
\tl_set_rescan:Nnn          *
\tl_set_rescan:(Nno|Nnx|cnn|cno|cnx) *
\tl_gset_rescan:Nnn         *
\tl_gset_rescan:(Nno|Nnx|cnn|cno|cnx)
```

Updated: 2015-08-11

Sets $\langle tl \ var \rangle$ to contain $\langle tokens \rangle$, applying the category code régime specified in the $\langle setup \rangle$ before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the $\langle setup \rangle$ are those in force at the point of use of $\backslash tl_set_rescan:Nnn$.) This allows the $\langle tl \ var \rangle$ to contain material with category codes other than those that apply when $\langle tokens \rangle$ are absorbed. The $\langle setup \rangle$ is run within a group and may contain any valid input, although only changes in category codes, such as uses of $\backslash cctab_select:N$, are relevant. See also $\backslash tl_rescan:nn$.

TExhackers note: The $\langle tokens \rangle$ are first turned into a string (using $\backslash tl_to_str:n$). If the string contains one or more characters with character code $\backslash newlinechar$ (set equal to $\backslash endlinechar$ unless that is equal to 32, before the user $\langle setup \rangle$), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

```
\tl_rescan:nn {<setup>} {<tokens>}
```

Updated: 2015-08-11

Rescans $\langle tokens \rangle$ applying the category code régime specified in the $\langle setup \rangle$, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the $\langle setup \rangle$ are those in force at the point of use of $\backslash tl_rescan:nn$.) The $\langle setup \rangle$ is run within a group and may contain any valid input, although only changes in category codes, such as uses of $\backslash cctab_select:N$, are relevant. See also $\backslash tl_set_rescan:Nnn$, which is more robust than using $\backslash tl_set:Nn$ in the $\langle tokens \rangle$ argument of $\backslash tl_rescan:nn$.

TExhackers note: The $\langle tokens \rangle$ are first turned into a string (using $\backslash tl_to_str:n$). If the string contains one or more characters with character code $\backslash newlinechar$ (set equal to $\backslash endlinechar$ unless that is equal to 32, before the user $\langle setup \rangle$), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

5 Token list conditionals

```
\tl_if_blank_p:n      *
\tl_if_blank_p:(e|V|o) *
\tl_if_blank:nTF      *
\tl_if_blank:(e|V|o)TF *
```

Updated: 2019-09-04

$\backslash tl_if_blank_p:n$ { $\langle token \ list \rangle$ }
 $\backslash tl_if_blank_p:(e|V|o) *$ { $\langle token \ list \rangle$ } { $\langle true \ code \rangle$ } { $\langle false \ code \rangle$ }
Tests if the $\langle token \ list \rangle$ consists only of blank spaces (*i.e.* contains no item). The test is **true** if $\langle token \ list \rangle$ is zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is **false** otherwise.

```
\tl_if_empty_p:N *
\tl_if_empty_p:c *
\tl_if_empty:NTF *
\tl_if_empty:cTF *
```

\tl_if_empty_p:N <tl var>
\tl_if_empty:N~~TF~~ <tl var> {{true code}} {{false code}}

Tests if the *<token list variable>* is entirely empty (*i.e.* contains no tokens at all).

```
\tl_if_empty_p:n   *
\tl_if_empty_p:(\V|\o) *
\tl_if_empty:nTF   *
\tl_if_empty:(\V|\o)TF *
```

\tl_if_empty_p:n {{<token list>}}
\tl_if_empty:n~~TF~~ {{<token list>}} {{true code}} {{false code}}

Tests if the *<token list>* is entirely empty (*i.e.* contains no tokens at all).

New: 2012-05-24
Updated: 2012-06-05

```
\tl_if_eq_p>NN      *
\tl_if_eq_p:(Nc|cN|cc) *
\tl_if_eq:NTF   *
\tl_if_eq:(Nc|cN|cc)TF *
```

\tl_if_eq_p:NN <tl var₁> <tl var₂>
\tl_if_eq:N~~TF~~ <tl var₁> <tl var₂> {{true code}} {{false code}}

Compares the content of two *<token list variables>* and is logically **true** if the two contain the same list of tokens (*i.e.* identical in both the list of characters they contain and the category codes of those characters). Thus for example

```
\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Nx \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:NTF \l_tmpa_tl \l_tmpb_tl { true } { false }
```

yields **false**. See also `\str_if_eq:nnTF` for a comparison that ignores category codes.

```
\tl_if_eq:NnTF *
\tl_if_eq:cnTF *
```

New: 2020-07-14

\tl_if_eq:N~~n~~TF <tl var₁> {{<token list₂>}} {{true code}} {{false code}}

Tests if the *<token list variable₁>* and the *<token list₂>* contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see `\tl_if_eq:NNTF` for an expandable version when both token lists are stored in variables, or `\str_if_eq:nnTF` if category codes are not important.

```
\tl_if_eq:nnTF *
```

\tl_if_eq:nnTF {{<token list₁>}} {{<token list₂>}} {{true code}} {{false code}}

Tests if *<token list₁>* and *<token list₂>* contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see `\tl_if_eq:NNTF` for an expandable version when token lists are stored in variables, or `\str_if_eq:nnTF` if category codes are not important.

```
\tl_if_in:NnTF *
\tl_if_in:cnTF *
```

\tl_if_in:N~~n~~TF <tl var> {{<token list>}} {{true code}} {{false code}}

Tests if the *<token list>* is found in the content of the *<tl var>*. The *<token list>* cannot contain the tokens {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

```
\tl_if_in:nnTF *
\tl_if_in:(\Vn|on|no)TF *
```

\tl_if_in:nnTF {{<token list₁>}} {{<token list₂>}} {{true code}} {{false code}}

Tests if *<token list₂>* is found inside *<token list₁>*. The *<token list₂>* cannot contain the tokens {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

```
\tl_if_novalue_p:n *
\tl_if_novalue:nTF *
```

New: 2017-11-14

```
\tl_if_novalue_p:n {\langle token list\rangle}
\tl_if_novalue:nTF {\langle token list\rangle} {\langle true code\rangle} {\langle false code\rangle}
```

Tests if the *<token list>* is exactly equal to the special `\c_novalue_tl` marker. This function is intended to allow construction of flexible document interface structures in which missing optional arguments are detected.

```
\tl_if_single_p:N *
\tl_if_single_p:c *
\tl_if_single:NTF *
\tl_if_single:cTF *
```

Updated: 2011-08-13

```
\tl_if_single_p:N {\langle tl var\rangle}
\tl_if_single:NTF {\langle tl var\rangle} {\langle true code\rangle} {\langle false code\rangle}
```

Tests if the content of the *<tl var>* consists of a single item, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\tl_count:N`.

```
\tl_if_single_p:n *
\tl_if_single:nTF *
```

Updated: 2011-08-13

```
\tl_if_single_p:n {\langle token list\rangle}
\tl_if_single:nTF {\langle token list\rangle} {\langle true code\rangle} {\langle false code\rangle}
```

Tests if the *<token list>* has exactly one item, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\tl_count:n`.

```
\tl_if_single_token_p:n *
\tl_if_single_token:nTF *
```

New: 2013-07-24

```
\tl_if_single_token_p:n {\langle token list\rangle}
\tl_if_single_token:nTF {\langle token list\rangle} {\langle true code\rangle} {\langle false code\rangle}
```

Tests if the token list consists of exactly one token, *i.e.* is either a single space character or a single “normal” token. Token groups *{...}* are not single tokens.

```
\tl_case:Nn   *
\tl_case:cn   *
\tl_case:NnTF *
\tl_case:cnTF *
```

New: 2013-07-24

```
\tl_case:NnTF {\langle test token list variable\rangle}
{
  {\langle token list variable case1\rangle} {\langle code case1\rangle}
  {\langle token list variable case2\rangle} {\langle code case2\rangle}
  ...
  {\langle token list variable casen\rangle} {\langle code casen\rangle}
}
{\langle true code\rangle}
{\langle false code\rangle}
```

This function compares the *<test token list variable>* in turn with each of the *<token list variable cases>*. If the two are equal (as described for `\tl_if_eq:NNTF`) then the associated *<code>* is left in the input stream and other cases are discarded. If any of the cases are matched, the *<true code>* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *<false code>* is inserted. The function `\tl_case:Nn`, which does nothing if there is no match, is also available.

6 Mapping to token lists

All mappings are done at the current group level, *i.e.* any local assignments made by the *<function>* or *<code>* discussed below remain in effect after the loop.

\tl_map_function:NN ☆
\tl_map_function:cN ☆

Updated: 2012-06-29

\tl_map_function:NN <tl var> <function>

Applies <function> to every <item> in the <tl var>. The <function> receives one argument for each iteration. This may be a number of tokens if the <item> was stored within braces. Hence the <function> should anticipate receiving n-type arguments. See also \tl_map_function:nN.

\tl_map_function:nN ☆
Updated: 2012-06-29

\tl_map_function:nN {<token list>} <function>

Applies <function> to every <item> in the <token list>, The <function> receives one argument for each iteration. This may be a number of tokens if the <item> was stored within braces. Hence the <function> should anticipate receiving n-type arguments. See also \tl_map_function:NN.

\tl_map_inline:Nn
\tl_map_inline:cn

Updated: 2012-06-29

\tl_map_inline:Nn <tl var> {{<inline function>}}

Applies the <inline function> to every <item> stored within the <tl var>. The <inline function> should consist of code which receives the <item> as #1. See also \tl_map_function:NN.

\tl_map_inline:nn
Updated: 2012-06-29

\tl_map_inline:nn {<token list>} {{<inline function>}}

Applies the <inline function> to every <item> stored within the <token list>. The <inline function> should consist of code which receives the <item> as #1. See also \tl_map_function:nN.

\tl_map_tokens:Nn ☆
\tl_map_tokens:cn ☆
\tl_map_tokens:nn ☆

New: 2019-09-02

\tl_map_tokens:Nn <tl var> {{<code>}}
\tl_map_tokens:nn <tokens> {{<code>}}

Analogue of \tl_map_function:NN which maps several tokens instead of a single function. The <code> receives each item in the <tl var> or <tokens> as two trailing brace groups. For instance,

\tl_map_tokens:Nn \l_my_t1 { \prg_replicate:nn { 2 } }

expands to twice each item in the <sequence>: for each item in \l_my_t1 the function \prg_replicate:nn receives 2 and <item> as its two arguments. The function \tl_map_inline:Nn is typically faster but is not expandable.

\tl_map_variable:NNn
\tl_map_variable:cNn

Updated: 2012-06-29

\tl_map_variable:NNn <tl var> <variable> {{<code>}}

Stores each <item> of the <tl var> in turn in the (token list) <variable> and applies the <code>. The <code> will usually make use of the <variable>, but this is not enforced. The assignments to the <variable> are local. Its value after the loop is the last <item> in the <tl var>, or its original value if the <tl var> is blank. See also \tl_map_inline:Nn.

\tl_map_variable:nNn
Updated: 2012-06-29

\tl_map_variable:nNn {<token list>} <variable> {{<code>}}

Stores each <item> of the <token list> in turn in the (token list) <variable> and applies the <code>. The <code> will usually make use of the <variable>, but this is not enforced. The assignments to the <variable> are local. Its value after the loop is the last <item> in the <tl var>, or its original value if the <tl var> is blank. See also \tl_map_inline:nn.

\tl_map_break: ☆

Updated: 2012-06-29**\tl_map_break:**

Used to terminate a `\tl_map_...` function before all entries in the *<token list variable>* have been processed. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
{
    \str_if_eq:nnT { #1 } { bingo } { \tl_map_break: }
    % Do something useful
}
```

See also `\tl_map_break:n`. Use outside of a `\tl_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *<tokens>* are inserted into the input stream. This depends on the design of the mapping function.

\tl_map_break:n ☆

Updated: 2012-06-29**\tl_map_break:n {<code>}**

Used to terminate a `\tl_map_...` function before all entries in the *<token list variable>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
{
    \str_if_eq:nnT { #1 } { bingo }
        { \tl_map_break:n { <code> } }
    % Do something useful
}
```

Use outside of a `\tl_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

7 Using token lists

`\tl_to_str:n *` `\tl_to_str:n {<token list>}`
`\tl_to_str:v *`

Converts the $\langle token\ list\rangle$ to a $\langle string\rangle$, leaving the resulting character tokens in the input stream. A $\langle string\rangle$ is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This function requires only a single expansion. Its argument *must* be braced.

TEXhackers note: This is the ε -TEX primitive `\detokenize`. Converting a $\langle token\ list\rangle$ to a $\langle string\rangle$ yields a concatenation of the string representations of every token in the $\langle token\ list\rangle$. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter `\escapechar`, absent if the `\escapechar` is negative or greater than the largest character code;
- the control sequence name, as defined by `\cs_to_str:N`;
- a space, unless the control sequence name is a single character whose category at the time of expansion of `\tl_to_str:n` is not “letter”.

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally #). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the `\escapechar`: for instance `\tl_to_str:n f\{a}` normally produces the three character “backslash”, “lower-case a”, “space”, but it may also produce a single “lower-case a” if the escape character is negative and a is currently not a letter.

`\tl_to_str:N *` `\tl_to_str:N <tl var>`
`\tl_to_str:c *`

Converts the content of the $\langle tl\ var\rangle$ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This $\langle string\rangle$ is then left in the input stream. For low-level details, see the notes given for `\tl_to_str:n`.

`\tl_use:N *` `\tl_use:N <tl var>`
`\tl_use:c *`

Recovering the content of a $\langle tl\ var\rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a $\langle tl\ var\rangle$ directly without an accessor function.

8 Working with the content of token lists

`\tl_count:n *` `\tl_count:n {<tokens>}`
`\tl_count:(V|o) *`

New: 2012-05-13

Counts the number of $\langle items\rangle$ in $\langle tokens\rangle$ and leaves this information in the input stream. Unbraced tokens count as one element as do each token group $\{\dots\}$. This process ignores any unprotected spaces within $\langle tokens\rangle$. See also `\tl_count:N`. This function requires three expansions, giving an $\langle integer\ denotation\rangle$.

```
\tl_count:N *
\tl_count:c *
```

New: 2012-05-13

```
\tl_count:N <tl var>
```

Counts the number of token groups in the *<tl var>* and leaves this information in the input stream. Unbraced tokens count as one element as do each token group (*{...}*). This process ignores any unprotected spaces within the *<tl var>*. See also `\tl_count:n`. This function requires three expansions, giving an *<integer denotation>*.

```
\tl_count_tokens:n *
```

New: 2019-02-25

```
\tl_count_tokens:n {<tokens>}
```

Counts the number of TeX tokens in the *<tokens>* and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of `a~{bc}` is 6.

```
\tl_reverse:n *
\tl_reverse:(V|o) *
```

Updated: 2012-01-08

```
\tl_reverse:n {<token list>}
```

Reverses the order of the *<items>* in the *<token list>*, so that *<item₁><item₂><item₃> ... <item_n>* becomes *<item_n>...<item₃><item₂><item₁>*. This process preserves unprotected space within the *<token list>*. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider `\tl_reverse_items:n`. See also `\tl_reverse:N`.

TeXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an x-type argument expansion.

```
\tl_reverse:N
\tl_reverse:c
\tl_greverse:N
\tl_greverse:c
```

Updated: 2012-01-08

```
\tl_reverse:N <tl var>
```

Reverses the order of the *<items>* stored in *<tl var>*, so that *<item₁><item₂><item₃> ... <item_n>* becomes *<item_n>...<item₃><item₂><item₁>*. This process preserves unprotected spaces within the *<token list variable>*. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. See also `\tl_reverse:n`, and, for improved performance, `\tl_reverse_items:n`.

```
\tl_reverse_items:n *
```

New: 2012-01-08

```
\tl_reverse_items:n {<token list>}
```

Reverses the order of the *<items>* stored in *<tl var>*, so that *{<item₁>}{<item₂>}{<item₃>} ... {<item_n>}* becomes *{<item_n>} ... {<item₃>}{<item₂>}{<item₁>}*. This process removes any unprotected space within the *<token list>*. Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function `\tl_reverse:n`.

TeXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an x-type argument expansion.

```
\tl_trim_spaces:n *
\tl_trim_spaces:o *
```

New: 2011-07-09

Updated: 2012-06-25

```
\tl_trim_spaces:n {<token list>}
```

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the *<token list>* and leaves the result in the input stream.

TeXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an x-type argument expansion.

```
\tl_trim_spaces_apply:nN *
\tl_trim_spaces_apply:oN *
```

New: 2018-04-12

`\tl_trim_spaces_apply:nN {⟨token list⟩} {function}`

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the *⟨token list⟩* and passes the result to the *⟨function⟩* as an n-type argument.

```
\tl_trim_spaces:N
\tl_trim_spaces:c
\tl_gtrim_spaces:N
\tl_gtrim_spaces:c
```

New: 2011-07-09

`\tl_trim_spaces:N {tl var}`

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the content of the *⟨tl var⟩*. Note that this therefore *resets* the content of the variable.

```
\tl_sort:Nn
\tl_sort:cn
\tl_gsort:Nn
\tl_gsort:cn
```

New: 2017-02-06

`\tl_sort:Nn {tl var} {⟨comparison code⟩}`

Sorts the items in the *⟨tl var⟩* according to the *⟨comparison code⟩*, and assigns the result to *⟨tl var⟩*. The details of sorting comparison are described in Section 1.

```
\tl_sort:nN *

```

New: 2017-02-06

`\tl_sort:nN {⟨token list⟩} {conditional}`

Sorts the items in the *⟨token list⟩*, using the *⟨conditional⟩* to compare items, and leaves the result in the input stream. The *⟨conditional⟩* should have signature :nnTF, and return `true` if the two items being compared should be left in the same order, and `false` if the items should be swapped. The details of sorting comparison are described in Section 1.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an x-type or e-type argument expansion.

9 The first token from a token list

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.

```
\tl_head:N      *
\tl_head:n      *
\tl_head:(v|v|f) *
```

Updated: 2012-09-09

```
\tl_head:n {\token list}
```

Leaves in the input stream the first *item* in the *token list*, discarding the rest of the *token list*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example

```
\tl_head:n { abc }
```

and

```
\tl_head:n { ~ abc }
```

both leave `a` in the input stream. If the “head” is a brace group, rather than a single token, the braces are removed, and so

```
\tl_head:n { ~ { ~ ab } c }
```

yields `ab`. A blank *token list* (see `\tl_if_blank:nTF`) results in `\tl_head:n` leaving nothing in the input stream.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an x-type argument expansion.

```
\tl_head:w *
```

```
\tl_head:w {\token list} { } \q_stop
```

Updated: 2012-09-01

Leaves in the input stream the first *item* in the *token list*, discarding the rest of the *token list*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank *token list* (which consists only of space characters) results in a low-level `TEX` error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, `\tl_if_blank:nF` may be used to avoid using the function with a “blank” argument. This function requires only a single expansion, and thus is suitable for use within an o-type expansion. In general, `\tl_head:n` should be preferred if the number of expansions is not critical.

```
\tl_tail:N      *
\tl_tail:n      *
\tl_tail:(v|v|f) *
```

Updated: 2012-09-01

```
\tl_tail:n {\token list}
```

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first *item* in the *token list*, and leaves the remaining tokens in the input stream. Thus for example

```
\tl_tail:n { a ~ {bc} d }
```

and

```
\tl_tail:n { ~ a ~ {bc} d }
```

both leave `~{bc}d` in the input stream. A blank *token list* (see `\tl_if_blank:nTF`) results in `\tl_tail:n` leaving nothing in the input stream.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an x-type argument expansion.

```
\tl_if_head_eq_catcode_p:nN * \tl_if_head_eq_catcode_p:nN {\langle token list\rangle} {test token}
\tl_if_head_eq_catcode_p:oN * \tl_if_head_eq_catcode:nNTF {\langle token list\rangle} {test token}
\tl_if_head_eq_catcode:nNTF *   {\langle true code\rangle} {\langle false code\rangle}
\tl_if_head_eq_catcode:oNTF *
```

Updated: 2012-07-09

Tests if the first *<token>* in the *<token list>* has the same category code as the *(test token)*. In the case where the *<token list>* is empty, the test is always **false**.

```
\tl_if_head_eq_charcode_p:nN * \tl_if_head_eq_charcode_p:nN {\langle token list\rangle} {test token}
\tl_if_head_eq_charcode_p:fN * \tl_if_head_eq_charcode:nNTF {\langle token list\rangle} {test token}
\tl_if_head_eq_charcode:nNTF *   {\langle true code\rangle} {\langle false code\rangle}
\tl_if_head_eq_charcode:fNTF *
```

Updated: 2012-07-09

Tests if the first *<token>* in the *<token list>* has the same character code as the *(test token)*. In the case where the *<token list>* is empty, the test is always **false**.

```
\tl_if_head_eq_meaning_p:nN * \tl_if_head_eq_meaning_p:nN {\langle token list\rangle} {test token}
\tl_if_head_eq_meaning:nNTF * \tl_if_head_eq_meaning:nNTF {\langle token list\rangle} {test token}
                                {\langle true code\rangle} {\langle false code\rangle}
```

Updated: 2012-07-09

Tests if the first *<token>* in the *<token list>* has the same meaning as the *(test token)*. In the case where *<token list>* is empty, the test is always **false**.

```
\tl_if_head_is_group_p:n * \tl_if_head_is_group_p:n {\langle token list\rangle}
\tl_if_head_is_group:nTF * \tl_if_head_is_group:nTF {\langle token list\rangle} {\langle true code\rangle} {\langle false code\rangle}
```

New: 2012-07-08

Tests if the first *<token>* in the *<token list>* is an explicit begin-group character (with category code 1 and any character code), in other words, if the *<token list>* starts with a brace group. In particular, the test is **false** if the *<token list>* starts with an implicit token such as `\c_group_begin_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

```
\tl_if_head_is_N_type_p:n * \tl_if_head_is_N_type_p:n {\langle token list\rangle}
\tl_if_head_is_N_type:nTF * \tl_if_head_is_N_type:nTF {\langle token list\rangle} {\langle true code\rangle} {\langle false code\rangle}
```

New: 2012-07-08

Tests if the first *<token>* in the *<token list>* is a normal N-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields **false**, as it does not have a “normal” first token. This function is useful to implement actions on token lists on a token by token basis.

```
\tl_if_head_is_space_p:n * \tl_if_head_is_space_p:n {\langle token list\rangle}
\tl_if_head_is_space:nTF * \tl_if_head_is_space:nTF {\langle token list\rangle} {\langle true code\rangle} {\langle false code\rangle}
```

Updated: 2012-07-08

Tests if the first *<token>* in the *<token list>* is an explicit space character (explicit token with character code 12 and category code 10). In particular, the test is **false** if the *<token list>* starts with an implicit token such as `\c_space_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

10 Using a single item

```
\tl_item:nn *
\tl_item:Nn *
\tl_item:cn *
```

New: 2014-07-17

```
\tl_item:nn {\langle token list\rangle} {\langle integer expression\rangle}
```

Indexing items in the $\langle token list \rangle$ from 1 on the left, this function evaluates the $\langle integer expression \rangle$ and leaves the appropriate item from the $\langle token list \rangle$ in the input stream. If the $\langle integer expression \rangle$ is negative, indexing occurs from the right of the token list, starting at -1 for the right-most item. If the index is out of bounds, then the function expands to nothing.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an `x`-type argument expansion.

```
\tl_rand_item:N *
\tl_rand_item:c *
\tl_rand_item:n *
```

New: 2016-12-06

```
\tl_rand_item:N {tl var}
\tl_rand_item:n {\langle token list\rangle}
```

Selects a pseudo-random item of the $\langle token list \rangle$. If the $\langle token list \rangle$ is blank, the result is empty. This is not available in older versions of X_HTeX.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an `x`-type argument expansion.

```
\tl_range:Nnn *
\tl_range:nnn *
```

New: 2017-02-17
Updated: 2017-07-15

`\tl_range:Nnn <tl var> {<start index>} {<end index>}`

`\tl_range:nnn {<token list>} {<start index>} {<end index>}`

Leaves in the input stream the items from the `<start index>` to the `<end index>` inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Here `<start index>` and `<end index>` should be `<integer expressions>`. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means ‘start counting from the left end’, and a negative index means ‘from the right end’. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$.

Spaces in between items in the actual range are preserved. Spaces at either end of the token list will be removed anyway (think to the token list being passed to `\tl_trim_spaces:n` to begin with).

Thus, with $l = 7$ as in the examples below, all of the following are equivalent and result in the whole token list

```
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 12 }
\tl_range:nnn { abcd~{e{}}fg } { -7 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { -12 } { 7 }
```

Here are some more interesting examples. The calls

```
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { 2 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { 2 } { -3 } }
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { -6 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { -6 } { -3 } }
```

are all equivalent and will print `bcd{e{}}` on the terminal; similarly

```
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { 2 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { 2 } { -3 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { -6 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { -6 } { -3 } }
```

are all equivalent and will print `bcd {e{}}` on the terminal (note the space in the middle). To the contrary,

```
\tl_range:nnn { abcd~{e{}}f } { 2 } { 4 }
```

will discard the space after ‘d’.

If we want to get the items from, say, the third to the last in a token list `<tl>`, the call is `\tl_range:nnn { <tl> } { 3 } { -1 }`. Similarly, for discarding the last item, we can do `\tl_range:nnn { <tl> } { 1 } { -2 }`.

For better performance, see `\tl_range_braced:nnn` and `\tl_range_unbraced:nnn`.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<item>` does not expand further when appearing in an `x`-type argument expansion.

11 Viewing token lists

\tl_show:N
\tl_show:c
Updated: 2015-08-01

\tl_show:N *tl var*

Displays the content of the *tl var* on the terminal.

TEXhackers note: This is similar to the TeX primitive \show, wrapped to a fixed number of characters per line.

\tl_show:n
Updated: 2015-08-07

\tl_show:n {*token list*}

Displays the *token list* on the terminal.

TEXhackers note: This is similar to the ε-TEx primitive \showtokens, wrapped to a fixed number of characters per line.

\tl_log:N
\tl_log:c
New: 2014-08-22
Updated: 2015-08-01

\tl_log:N *tl var*

Writes the content of the *tl var* in the log file. See also \tl_show:N which displays the result in the terminal.

\tl_log:n
New: 2014-08-22
Updated: 2015-08-07

\tl_log:n {*token list*}

Writes the *token list* in the log file. See also \tl_show:n which displays the result in the terminal.

12 Constant token lists

\c_empty_tl

Constant that is always empty.

\c_novalue_tl
New: 2017-11-14

A marker for the absence of an argument. This constant **tl** can safely be typeset (*cf.* \q_nil), with the result being **-NoValue-**. It is important to note that \c_novalue_tl is constructed such that it will *not* match the simple text input **-NoValue-**, *i.e.* that

```
\tl_if_eq:NnTF \c_novalue_tl { -NoValue- }
```

is logically **false**. The \c_novalue_tl marker is intended for use in creating document-level interfaces, where it serves as an indicator that an (optional) argument was omitted. In particular, it is distinct from a simple empty **tl**.

\c_space_tl

An explicit space character contained in a token list (compare this with \c_space_token). For use where an explicit space is required.

13 Scratch token lists

`\l_tmpa_t1`
`\l_tmpb_t1`

Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_t1`
`\g_tmpb_t1`

Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Part VIII

The `\3str` package: Strings

`TEX` associates each character with a category code: as such, there is no concept of a “string” as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense “ignoring” category codes: this is done by treating token lists as strings in a `TEX` sense.

A `TEX` string (and thus an `\exp3` string) is a series of characters which have category code 12 (“other”) with the exception of space characters which have category code 10 (“space”). Thus at a technical level, a `TEX` string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using `\t1_to_str:n` for internal processing, and do not treat a token list or the corresponding string representation differently.

As a string is a subset of the more general token list, it is sometimes unclear when one should be used over the other. Use a string variable for data that isn’t primarily intended for typesetting and for which a level of protection from unwanted expansion is suitable. This data type simplifies comparison of variables since there are no concerns about expansion of their contents.

The functions `\cs_to_str:N`, `\t1_to_str:n`, `\t1_to_str:N` and `\token_to_str:N` (and variants) generate strings from the appropriate input: these are documented in `\3basics`, `\3tl` and `\3token`, respectively.

Most expandable functions in this module come in three flavours:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

1 Building strings

`\str_new:N`
`\str_new:c`
New: 2015-09-18

`\str_new:N <str var>`

Creates a new `<str var>` or raises an error if the name is already taken. The declaration is global. The `<str var>` is initially empty.

`\str_const:Nn`
`\str_const:(NV|Nx|cn|cV|cx)`
New: 2015-09-18
Updated: 2018-07-28

`\str_const:Nn <str var> {<token list>}`

Creates a new constant `<str var>` or raises an error if the name is already taken. The value of the `<str var>` is set globally to the `<token list>`, converted to a string.

```
\str_clear:N  
\str_clear:c  
\str_gclear:N  
\str_gclear:c
```

New: 2015-09-18

```
\str_clear:N <str var>
```

Clears the content of the *<str var>*.

```
\str_clear_new:N  
\str_clear_new:c
```

New: 2015-09-18

```
\str_clear_new:N <str var>
```

Ensures that the *<str var>* exists globally by applying `\str_new:N` if necessary, then applies `\str_(g)clear:N` to leave the *<str var>* empty.

```
\str_set_eq:NN  
\str_set_eq:(cN|Nc|cc)  
\str_gset_eq:NN  
\str_gset_eq:(cN|Nc|cc)
```

New: 2015-09-18

```
\str_set_eq:NN <str var1> <str var2>
```

Sets the content of *<str var₁>* equal to that of *<str var₂>*.

```
\str_concat:NNN  
\str_concat:ccc  
\str_gconcat:NNN  
\str_gconcat:ccc
```

New: 2017-10-08

```
\str_concat:NNN <str var1> <str var2> <str var3>
```

Concatenates the content of *<str var₂>* and *<str var₃>* together and saves the result in *<str var₁>*. The *<str var₂>* is placed at the left side of the new string variable. The *<str var₂>* and *<str var₃>* must indeed be strings, as this function does not convert their contents to a string.

2 Adding data to string variables

```
\str_set:Nn  
\str_set:(NV|Nx|cn|cV|cx)  
\str_gset:Nn  
\str_gset:(NV|Nx|cn|cV|cx)
```

New: 2015-09-18
Updated: 2018-07-28

```
\str_set:Nn <str var> {<token list>}
```

Converts the *<token list>* to a *<string>*, and stores the result in *<str var>*.

```
\str_put_left:Nn  
\str_put_left:(NV|Nx|cn|cV|cx)  
\str_gput_left:Nn  
\str_gput_left:(NV|Nx|cn|cV|cx)
```

New: 2015-09-18
Updated: 2018-07-28

```
\str_put_left:Nn <str var> {<token list>}
```

Converts the *<token list>* to a *<string>*, and prepends the result to *<str var>*. The current contents of the *<str var>* are not automatically converted to a string.

```
\str_put_right:Nn
\str_put_right:(NV|Nx|cn|cV|cx)
\str_gput_right:Nn
\str_gput_right:(NV|Nx|cn|cV|cx)
```

New: 2015-09-18
Updated: 2018-07-28

```
\str_put_right:Nn <str var> {<token list>}
```

Converts the *<token list>* to a *<string>*, and appends the result to *<str var>*. The current contents of the *<str var>* are not automatically converted to a string.

3 Modifying string variables

```
\str_replace_once:Nnn
\str_replace_once:cnn
\str_greplace_once:Nn
\str_greplace_once:cnn
```

New: 2017-10-08

```
\str_replace_once:Nnn <str var> {<old>} {<new>}
```

Converts the *<old>* and *<new>* token lists to strings, then replaces the first (leftmost) occurrence of *<old string>* in the *<str var>* with *<new string>*.

```
\str_replace_all:Nnn
\str_replace_all:cnn
\str_greplace_all:Nn
\str_greplace_all:cnn
```

New: 2017-10-08

```
\str_replace_all:Nnn <str var> {<old>} {<new>}
```

Converts the *<old>* and *<new>* token lists to strings, then replaces all occurrences of *<old string>* in the *<str var>* with *<new string>*. As this function operates from left to right, the pattern *<old string>* may remain after the replacement (see `\str_remove_all:Nn` for an example).

```
\str_remove_once:Nn
\str_remove_once:cn
\str_gremove_once:Nn
\str_gremove_once:cn
```

New: 2017-10-08

```
\str_remove_once:Nn <str var> {<token list>}
```

Converts the *<token list>* to a *<string>* then removes the first (leftmost) occurrence of *<string>* from the *<str var>*.

```
\str_remove_all:Nn
\str_remove_all:cn
\str_gremove_all:Nn
\str_gremove_all:cn
```

New: 2017-10-08

```
\str_remove_all:Nn <str var> {<token list>}
```

Converts the *<token list>* to a *<string>* then removes all occurrences of *<string>* from the *<str var>*. As this function operates from left to right, the pattern *<string>* may remain after the removal, for instance,

```
\str_set:Nn \l_tmpa_str {abbccd} \str_remove_all:Nn \l_tmpa_str
{bc}
```

results in `\l_tmpa_str` containing abcd.

4 String conditionals

```
\str_if_exist_p:N *
\str_if_exist_p:c *
\str_if_exist:NTF *
\str_if_exist:cTF *

```

Tests whether the *<str var>* is currently defined. This does not check that the *<str var>* really is a string.

New: 2015-09-18

```
\str_if_empty_p:N *
\str_if_empty_p:c *
\str_if_empty:NTF *
\str_if_empty:cTF *

```

Tests if the *(string variable)* is entirely empty (*i.e.* contains no characters at all).

New: 2015-09-18

```
\str_if_eq_p>NN      *
\str_if_eq_p:(Nc|cN|cc) *
\str_if_eq:NNTF   *
\str_if_eq:(Nc|cN|cc)TF *

```

Compares the content of two *<str variables>* and is logically **true** if the two contain the same characters in the same order. See `\tl_if_eq:NNTF` to compare tokens (including their category codes) rather than characters.

New: 2015-09-18

```
\str_if_eq_p:nn          * \str_if_eq_p:nnn {{tl1}} {{tl2}}
\str_if_eq_p:(Vn|on|no|nV|VV|vn|nv|ee) * \str_if_eq:nnTF {{tl1}} {{tl2}} {{true code}} {{false code}}
\str_if_eq:nnTF          *
\str_if_eq:(Vn|on|no|nV|VV|vn|nv|ee)TF *
```

Updated: 2018-06-18

Compares the two *<token lists>* on a character by character basis (namely after converting them to strings), and is **true** if the two *<strings>* contain the same characters in the same order. Thus for example

```
\str_if_eq_p:no { abc } { \tl_to_str:n { abc } }
```

is logically **true**. See `\tl_if_eq:nnTF` to compare tokens (including their category codes) rather than characters.

```
\str_if_in:NnTF
\str_if_in:cNnTF

```

Converts the *<token list>* to a *<string>* and tests if that *<string>* is found in the content of the *<str var>*.

New: 2017-10-08

```
\str_if_in:nnTF

```

Converts both *<token lists>* to *<strings>* and tests whether *<string₂>* is found inside *<string₁>*.

New: 2017-10-08

```
\str_case:nn      *
\str_case:(Vn|on|nV|nv)  *
\str_case:nnTF   *
\str_case:(Vn|on|nV|nv)TF *
```

New: 2013-07-24
Updated: 2015-02-28

```
\str_case:nnTF {{test string}}
{
  {{string case1}} {{code case1}}
  {{string case2}} {{code case2}}
  ...
  {{string casen}} {{code casen}}
}
{{true code}}
{{false code}}
```

Compares the *<test string>* in turn with each of the *<string cases>* (all token lists are converted to strings). If the two are equal (as described for `\str_if_eq:nnTF`) then the associated *<code>* is left in the input stream and other cases are discarded. If any of the cases are matched, the *<true code>* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *<false code>* is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

```
\str_case_e:nn   *
\str_case_e:nnTF *
```

New: 2018-06-19

```
\str_case_e:nnTF {{test string}}
{
  {{string case1}} {{code case1}}
  {{string case2}} {{code case2}}
  ...
  {{string casen}} {{code casen}}
}
{{true code}}
{{false code}}
```

Compares the full expansion of the *<test string>* in turn with the full expansion of the *<string cases>* (all token lists are converted to strings). If the two full expansions are equal (as described for `\str_if_eq:nnTF`) then the associated *<code>* is left in the input stream and other cases are discarded. If any of the cases are matched, the *<true code>* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *<false code>* is inserted. The function `\str_case_e:nn`, which does nothing if there is no match, is also available. The *<test string>* is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

5 Mapping to strings

All mappings are done at the current group level, *i.e.* any local assignments made by the *<function>* or *<code>* discussed below remain in effect after the loop.

```
\str_map_function:NN ☆
\str_map_function:cN ☆
```

New: 2017-11-14

```
\str_map_function:NN <str var> <function>
```

Applies *<function>* to every *<character>* in the *<str var>* including spaces. See also `\str_map_function:nN`.

```
\str_map_function:nN ☆
```

New: 2017-11-14

```
\str_map_function:nN {{token list}} <function>
```

Converts the *<token list>* to a *<string>* then applies *<function>* to every *<character>* in the *<string>* including spaces. See also `\str_map_function:NN`.

\str_map_inline:Nn
\str_map_inline:cN

New: 2017-11-14

\str_map_inline:Nn <str var> {<inline function>}

Applies the *<inline function>* to every *<character>* in the *<str var>* including spaces. The *<inline function>* should consist of code which receives the *<character>* as #1. See also \str_map_function:NN.

\str_map_inline:nn

New: 2017-11-14

\str_map_inline:nn {<token list>} {<inline function>}

Converts the *<token list>* to a *<string>* then applies the *<inline function>* to every *<character>* in the *<string>* including spaces. The *<inline function>* should consist of code which receives the *<character>* as #1. See also \str_map_function:NN.

\str_map_variable>NNn
\str_map_variable:cNn

New: 2017-11-14

\str_map_variable>NNn <str var> <variable> {<code>}

Stores each *<character>* of the *<string>* (including spaces) in turn in the (string or token list) *<variable>* and applies the *<code>*. The *<code>* will usually make use of the *<variable>*, but this is not enforced. The assignments to the *<variable>* are local. Its value after the loop is the last *<character>* in the *<string>*, or its original value if the *<string>* is empty. See also \str_map_inline:Nn.

\str_map_variable:nNn

New: 2017-11-14

\str_map_variable:nNn {<token list>} <variable> {<code>}

Converts the *<token list>* to a *<string>* then stores each *<character>* in the *<string>* (including spaces) in turn in the (string or token list) *<variable>* and applies the *<code>*. The *<code>* will usually make use of the *<variable>*, but this is not enforced. The assignments to the *<variable>* are local. Its value after the loop is the last *<character>* in the *<string>*, or its original value if the *<string>* is empty. See also \str_map_inline:Nn.

\str_map_break: ☆

New: 2017-10-08

\str_map_break:

Used to terminate a \str_map_... function before all characters in the *<string>* have been processed. This normally takes place within a conditional statement, for example

```
\str_map_inline:Nn \l_my_str
{
    \str_if_eq:nnT { #1 } { bingo } { \str_map_break: }
    % Do something useful
}
```

See also \str_map_break:n. Use outside of a \str_map_... scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before continuing with the code that follows the loop. This depends on the design of the mapping function.

\str_map_break:n ☆

New: 2017-10-08

\str_map_break:n {<code>}

Used to terminate a \str_map_... function before all characters in the *⟨string⟩* have been processed, inserting the *⟨code⟩* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\str_map_inline:Nn \l_my_str
{
    \str_if_eq:nnT { #1 } { bingo }
        { \str_map_break:n { <code> } }
    % Do something useful
}
```

Use outside of a \str_map_... scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *⟨code⟩* is inserted into the input stream. This depends on the design of the mapping function.

6 Working with the content of strings

\str_use:N ★ \str_use:N <str var>

\str_use:c ★

New: 2015-09-18

Recovers the content of a *⟨str var⟩* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a *⟨str⟩* directly without an accessor function.

\str_count:N ★ \str_count:n {<token list>}

\str_count:c ★

\str_count:n ★

\str_count_ignore_spaces:n ★

New: 2015-09-18

Leaves in the input stream the number of characters in the string representation of *⟨token list⟩*, as an integer denotation. The functions differ in their treatment of spaces. In the case of \str_count:N and \str_count:n, all characters including spaces are counted. The \str_count_ignore_spaces:n function leaves the number of non-space characters in the input stream.

\str_count_spaces:N ★ \str_count_spaces:n {<token list>}

\str_count_spaces:c ★

\str_count_spaces:n ★

New: 2015-09-18

Leaves in the input stream the number of space characters in the string representation of *⟨token list⟩*, as an integer denotation. Of course, this function has no *_ignore_spaces* variant.

```
\str_head:N      * \str_head:n {\<token list>}
\str_head:c      *
\str_head:n      *
\str_head_ignore_spaces:n *
```

New: 2015-09-18

Converts the $\langle token\ list\rangle$ into a $\langle string\rangle$. The first character in the $\langle string\rangle$ is then left in the input stream, with category code “other”. The functions differ if the first character is a space: \str_head:N and \str_head:n return a space token with category code 10 (blank space), while the $\text{\str_head_ignore_spaces:n}$ function ignores this space character and leaves the first non-space character in the input stream. If the $\langle string\rangle$ is empty (or only contains spaces in the case of the $_ignore_spaces$ function), then nothing is left on the input stream.

```
\str_tail:N      * \str_tail:n {\<token list>}
\str_tail:c      *
\str_tail:n      *
\str_tail_ignore_spaces:n *
```

New: 2015-09-18

Converts the $\langle token\ list\rangle$ to a $\langle string\rangle$, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: \str_tail:N and \str_tail:n only trim that space, while $\text{\str_tail_ignore_spaces:n}$ removes the first non-space character and any space before it. If the $\langle token\ list\rangle$ is empty (or blank in the case of the $_ignore_spaces$ variant), then nothing is left on the input stream.

```
\str_item:Nn      * \str_item:nn {\<token list>} {\<integer expression>}
\str_item:nn      *
\str_item_ignore_spaces:nn *
```

New: 2015-09-18

Converts the $\langle token\ list\rangle$ to a $\langle string\rangle$, and leaves in the input stream the character in position $\langle integer\ expression\rangle$ of the $\langle string\rangle$, starting at 1 for the first (left-most) character. In the case of \str_item:Nn and \str_item:nn , all characters including spaces are taken into account. The $\text{\str_item_ignore_spaces:nn}$ function skips spaces when counting characters. If the $\langle integer\ expression\rangle$ is negative, characters are counted from the end of the $\langle string\rangle$. Hence, -1 is the right-most character, etc.

```

\str_range:Nnn          * \str_range:nnn {\langle token list\rangle} {\langle start index\rangle} {\langle end index\rangle}
\str_range:cnn          *
\str_range:nnn          *
\str_range_ignore_spaces:nnn *

```

New: 2015-09-18

Converts the $\langle token\ list\rangle$ to a $\langle string\rangle$, and leaves in the input stream the characters from the $\langle start\ index\rangle$ to the $\langle end\ index\rangle$ inclusive. Spaces are preserved and counted as items (contrast this with $\backslash tl_range:nnn$ where spaces are not counted as items and are possibly discarded from the output).

Here $\langle start\ index\rangle$ and $\langle end\ index\rangle$ should be integer denotations. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', a negative index means 'start counting from the right end'. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$. For instance,

```

\iow_term:x { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { 0 } { -1 } }

```

prints **bcd**, **cde**, **ef**, and an empty line to the terminal. The $\langle start\ index\rangle$ must always be smaller than or equal to the $\langle end\ index\rangle$: if this is not the case then no output is generated. Thus

```

\iow_term:x { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:x { \str_range:nnn { abcdef } { -1 } { -4 } }

```

both yield empty strings.

The behavior of $\backslash str_range_ignore_spaces:nnn$ is similar, but spaces are removed before starting the job. The input

```

\iow_term:x { \str_range:nnn { abcdefg } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdefg } { 2 } { -3 } }
\iow_term:x { \str_range:nnn { abcdefg } { -6 } { 5 } }
\iow_term:x { \str_range:nnn { abcdefg } { -6 } { -3 } }

\iow_term:x { \str_range:nnn { abc~efg } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abc~efg } { 2 } { -3 } }
\iow_term:x { \str_range:nnn { abc~efg } { -6 } { 5 } }
\iow_term:x { \str_range:nnn { abc~efg } { -6 } { -3 } }

\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { -3 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { -3 } }

```

```
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { -3 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { -3 } }
```

will print four instances of `bcde`, four instances of `bc e` and eight instances of `bcde`.

7 String manipulation

```
\str_lowercase:n *
\str_lowercase:f *
\str_uppercase:n *
\str_uppercase:f *
```

New: 2019-11-26

`\str_lowercase:n {⟨tokens⟩}`
`\str_uppercase:n {⟨tokens⟩}`

Converts the input `⟨tokens⟩` to their string representation, as described for `\tl_to_str:n`, and then to the lower or upper case representation using a one-to-one mapping as described by the Unicode Consortium file `UnicodeData.txt`.

These functions are intended for case changing programmatic data in places where upper/lower case distinctions are meaningful. One example would be automatically generating a function name from user input where some case changing is needed. In this situation the input is programmatic, not textual, case does have meaning and a language-independent one-to-one mapping is appropriate. For example

```
\cs_new_protected:Npn \myfunc:nn #1#2
{
    \cs_set_protected:cpx
    {
        user
        \str_uppercase:f { \tl_head:n {#1} }
        \str_lowercase:f { \tl_tail:n {#1} }
    }
    { #2 }
}
```

would be used to generate a function with an auto-generated name consisting of the upper case equivalent of the supplied name followed by the lower case equivalent of the rest of the input.

These functions should *not* be used for

- Caseless comparisons: use `\str_foldcase:n` for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the `\text_lowercase:n(n)`, `\text_uppercase:n(n)` and `\text_titlecase:n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

TeXhackers note: As with all `expl3` functions, the input supported by `\str_foldcase:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfTeX *only* the Latin alphabet characters A–Z are case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both XeTeX and LuaTeX.

```
\str_foldcase:n *
```

New: 2019-11-26

```
\str_foldcase:n {\langle tokens\rangle}
```

Converts the input $\langle tokens \rangle$ to their string representation, as described for `\tl_to_str:n`, and then folds the case of the resulting $\langle string \rangle$ to remove case information. The result of this process is left in the input stream.

String folding is a process used for material such as identifiers rather than for “text”. The folding provided by `\str_foldcase:n` follows the mappings provided by the [Unicode Consortium](#), who [state](#):

Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by `\str_foldcase:n` follows the “full” scheme defined by the Unicode Consortium (*e.g.* SSfolds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (*i.e.* I always folds to i and not to ı).

TEXhackers note: As with all `expl3` functions, the input supported by `\str_foldcase:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfTEX only the Latin alphabet characters A–Z are case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both XeTeX and LuaTeX, subject only to the fact that XeTeX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

8 Viewing strings

```
\str_show:N  
\str_show:c  
\str_show:n
```

New: 2015-09-18

```
\str_show:N {str var}
```

Displays the content of the $\langle str var \rangle$ on the terminal.

```
\str_log:N  
\str_log:c  
\str_log:n
```

New: 2019-02-15

```
\str_log:N {str var}
```

Writes the content of the $\langle str var \rangle$ in the log file.

9 Constant token lists

```
\c_ampersand_str  
\c_atsign_str  
\c_backslash_str  
\c_left_brace_str  
\c_right_brace_str  
\c_circumflex_str  
\c_colon_str  
\c_dollar_str  
\c_hash_str  
\c_percent_str  
\c_tilde_str  
\c_underscore_str
```

Constant strings, containing a single character token, with category code 12.

New: 2015-09-19

10 Scratch strings

```
\l_tmpa_str  
\l_tmpb_str
```

Scratch strings for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

```
\g_tmpa_str  
\g_tmpb_str
```

Scratch strings for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Part IX

The `I3str-convert` package: string encoding conversions

1 Encoding and escaping schemes

Traditionally, string encodings only specify how strings of characters should be stored as bytes. However, the resulting lists of bytes are often to be used in contexts where only a restricted subset of bytes are permitted (*e.g.*, PDF string objects, URLs). Hence, storing a string of characters is done in two steps.

- The code points (“character codes”) are expressed as bytes following a given “encoding”. This can be UTF-16, ISO 8859-1, *etc.* See Table 1 for a list of supported encodings.⁵
- Bytes are translated to \TeX tokens through a given “escaping”. Those are defined for the most part by the `pdf` file format. See Table 2 for a list of escaping methods supported.⁵

2 Conversion functions

```
\str_set_convert:Nnnn
\str_gset_convert:Nnnn
```

```
\str_set_convert:Nnnn <str var> {<string>} {<name 1>} {<name 2>}
```

This function converts the $\langle string \rangle$ from the encoding given by $\langle name 1 \rangle$ to the encoding given by $\langle name 2 \rangle$, and stores the result in the $\langle str var \rangle$. Each $\langle name \rangle$ can have the form $\langle encoding \rangle$ or $\langle encoding \rangle / \langle escaping \rangle$, where the possible values of $\langle encoding \rangle$ and $\langle escaping \rangle$ are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty $\langle name \rangle$ indicates the use of “native” strings, 8-bit for `pdf\TeX`, and Unicode strings for the other two engines.

For example,

```
\str_set_convert:Nnnn \l_foo_str { Hello! } { } { utf16/hex }
```

results in the variable `\l_foo_str` holding the string `FEFF00480065006C006C006F0021`. This is obtained by converting each character in the (native) string `Hello!` to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark “`FEFF`”, which can be avoided by specifying the encoding `utf16be/hex`.

An error is raised if the $\langle string \rangle$ is not valid according to the $\langle escaping 1 \rangle$ and $\langle encoding 1 \rangle$, or if it cannot be reencoded in the $\langle encoding 2 \rangle$ and $\langle escaping 2 \rangle$ (for instance, if a character does not exist in the $\langle encoding 2 \rangle$). Erroneous input is replaced by the Unicode replacement character “`FFFD`”, and characters which cannot be reencoded are replaced by either the replacement character “`FFFD`” if it exists in the $\langle encoding 2 \rangle$, or an encoding-specific replacement character, or the question mark character.

⁵Encodings and escapings will be added as they are requested.

Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the encoding in this list.

$\langle Encoding \rangle$	description
<code>utf8</code>	UTF-8
<code>utf16</code>	UTF-16, with byte-order mark
<code>utf16be</code>	UTF-16, big-endian
<code>utf16le</code>	UTF-16, little-endian
<code>utf32</code>	UTF-32, with byte-order mark
<code>utf32be</code>	UTF-32, big-endian
<code>utf32le</code>	UTF-32, little-endian
<code>iso88591, latin1</code>	ISO 8859-1
<code>iso88592, latin2</code>	ISO 8859-2
<code>iso88593, latin3</code>	ISO 8859-3
<code>iso88594, latin4</code>	ISO 8859-4
<code>iso88595</code>	ISO 8859-5
<code>iso88596</code>	ISO 8859-6
<code>iso88597</code>	ISO 8859-7
<code>iso88598</code>	ISO 8859-8
<code>iso88599, latin5</code>	ISO 8859-9
<code>iso885910, latin6</code>	ISO 8859-10
<code>iso885911</code>	ISO 8859-11
<code>iso885913, latin7</code>	ISO 8859-13
<code>iso885914, latin8</code>	ISO 8859-14
<code>iso885915, latin9</code>	ISO 8859-15
<code>iso885916, latin10</code>	ISO 8859-16
<code>clist</code>	comma-list of integers
<code>(empty)</code>	native (Unicode) string

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the escaping in this list.

$\langle Escaping \rangle$	description
<code>bytes, or empty</code>	arbitrary bytes
<code>hex, hexadecimal</code>	byte = two hexadecimal digits
<code>name</code>	see <code>\pdfescape name</code>
<code>string</code>	see <code>\pdfescape string</code>
<code>url</code>	encoding used in URLs

```
\str_set_convert:NnnnTF
\str_gset_convert:NnnnTF
```

```
\str_set_convert:NnnnTF <str var> {<string>} {<name 1>} {<name 2>} {<true code>}
{<false code>}
```

As `\str_set_convert:Nnnn`, converts the `<string>` from the encoding given by `<name 1>` to the encoding given by `<name 2>`, and assigns the result to `<str var>`. Contrarily to `\str_set_convert:Nnnn`, the conditional variant does not raise errors in case the `<string>` is not valid according to the `<name 1>` encoding, or cannot be expressed in the `<name 2>` encoding. Instead, the `<false code>` is performed.

3 Conversion by expansion (for PDF contexts)

A small number of expandable functions are provided for use in PDF string/name contexts. These *assume UTF-8* and *no escaping* in the input.

```
\str_convert_pdfname:n *
```

```
\str_convert_pdfname:n <string>
```

As `\str_set_convert:Nnnn`, converts the `<string>` on a byte-by-byte basis with non-ASCII codepoints escaped using hashes.

4 Possibilities, and things to do

Encoding/escaping-related tasks.

- In Xe_ET_EX/LuaT_EX, would it be better to use the `~~~~....` approach to build a string from a given list of character codes? Namely, within a group, assign `0-9a-f` and all characters we want to category “other”, then assign `^` the category superscript, and use `\scantokens`.
- Change `\str_set_convert:Nnnn` to expand its last two arguments.
- Describe the internal format in the code comments. Refuse code points in `["D800, "DFFF]` in the internal representation?
- Add documentation about each encoding and escaping method, and add examples.
- The `hex` unescaping should raise an error for odd-token count strings.
- Decide what bytes should be escaped in the `url` escaping. Perhaps the characters `! ' ()*-./0123456789_` are safe, and all other characters should be escaped?
- Automate generation of 8-bit mapping files.
- Change the framework for 8-bit encodings: for decoding from 8-bit to Unicode, use 256 integer registers; for encoding, use a tree-box.
- More encodings (see Heiko’s `stringenc`). CESU?
- More escapings: ASCII85, shell escapes, lua escapes, *etc.*?

Part X

The **I3seq** package

Sequences and stacks

LATEX3 implements a “sequence” data type, which contain an ordered list of entries which may contain any *(balanced text)*. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in LATEX3. This is achieved using a number of dedicated stack functions.

1 Creating and initialising sequences

`\seq_new:N`
`\seq_new:c`

Creates a new *(sequence)* or raises an error if the name is already taken. The declaration is global. The *(sequence)* initially contains no items.

`\seq_clear:N`
`\seq_clear:c`
`\seq_gclear:N`
`\seq_gclear:c`

Clears all items from the *(sequence)*.

`\seq_clear_new:N`
`\seq_clear_new:c`
`\seq_gclear_new:N`
`\seq_gclear_new:c`

Ensures that the *(sequence)* exists globally by applying `\seq_new:N` if necessary, then applies `\seq_(g)clear:N` to leave the *(sequence)* empty.

`\seq_set_eq:NN`
`\seq_set_eq:(cN|Nc|cc)`
`\seq_gset_eq:NN`
`\seq_gset_eq:(cN|Nc|cc)`

Sets the content of *(sequence₁)* equal to that of *(sequence₂)*.

`\seq_set_from_clist:NN`
`\seq_set_from_clist:(cN|Nc|cc)`
`\seq_set_from_clist:Nn`
`\seq_set_from_clist:cn`
`\seq_gset_from_clist:NN`
`\seq_gset_from_clist:(cN|Nc|cc)`
`\seq_gset_from_clist:Nn`
`\seq_gset_from_clist:cn`

`\seq_set_from_clist:NN` *(sequence)* *(comma-list)*

New: 2014-07-17

Converts the data in the *(comma list)* into a *(sequence)*: the original *(comma list)* is unchanged.

```
\seq_const_from_clist:Nn
\seq_const_from_clist:cn
```

New: 2017-11-28

```
\seq_const_from_clist:Nn <seq var> {{comma-list}}
```

Creates a new constant $\langle \text{seq var} \rangle$ or raises an error if the name is already taken. The $\langle \text{seq var} \rangle$ is set globally to contain the items in the $\langle \text{comma list} \rangle$.

```
\seq_set_split:Nnn
\seq_set_split:NnV
\seq_gset_split:Nnn
\seq_gset_split:NnV
```

New: 2011-08-15

Updated: 2012-07-02

```
\seq_set_split:Nnn <sequence> {{delimiter}} {{token list}}
```

Splits the $\langle \text{sequence} \rangle$ into $\langle \text{items} \rangle$ separated by $\langle \text{delimiter} \rangle$, and assigns the result to the $\langle \text{sequence} \rangle$. Spaces on both sides of each $\langle \text{item} \rangle$ are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of \!l3clist functions. Empty $\langle \text{items} \rangle$ are preserved by $\text{\seq_set_split:Nnn}$, and can be removed afterwards using $\text{\seq_remove_all:Nn} \langle \text{sequence} \rangle \{\}$. The $\langle \text{delimiter} \rangle$ may not contain {, } or # (assuming TeX's normal category code régime). If the $\langle \text{delimiter} \rangle$ is empty, the $\langle \text{token list} \rangle$ is split into $\langle \text{items} \rangle$ as a $\langle \text{token list} \rangle$.

```
\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc
```

```
\seq_concat:NNN <sequence1> <sequence2> <sequence3>
```

Concatenates the content of $\langle \text{sequence}_2 \rangle$ and $\langle \text{sequence}_3 \rangle$ together and saves the result in $\langle \text{sequence}_1 \rangle$. The items in $\langle \text{sequence}_2 \rangle$ are placed at the left side of the new sequence.

```
\seq_if_exist_p:N *
\seq_if_exist_p:c *
\seq_if_exist:NTF *
\seq_if_exist:cTF *
```

New: 2012-03-03

```
\seq_if_exist_p:N <sequence>
```

```
\seq_if_exist:NTF <sequence> {{true code}} {{false code}}
```

Tests whether the $\langle \text{sequence} \rangle$ is currently defined. This does not check that the $\langle \text{sequence} \rangle$ really is a sequence variable.

```
\seq_put_left:Nn
\seq_put_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gput_left:Nn
\seq_gput_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
```

```
\seq_put_left:Nn <sequence> {{item}}
```

Appends the $\langle \text{item} \rangle$ to the left of the $\langle \text{sequence} \rangle$.

```
\seq_put_right:Nn
\seq_put_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gput_right:Nn
\seq_gput_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
```

```
\seq_put_right:Nn <sequence> {{item}}
```

Appends the $\langle \text{item} \rangle$ to the right of the $\langle \text{sequence} \rangle$.

3 Recovering items from sequences

Items can be recovered from either the left or the right of sequences. For implementation reasons, the actions at the left of the sequence are faster than those acting on the right. These functions all assign the recovered material locally, *i.e.* setting the $\langle \text{token list variable} \rangle$ used with \tl_set:Nn and *never* \tl_gset:Nn .

`\seq_get_left:NN`
`\seq_get_left:cN`

Updated: 2012-05-14

`\seq_get_left:NN <sequence> <token list variable>`

Stores the left-most item from a *<sequence>* in the *<token list variable>* without removing it from the *<sequence>*. The *<token list variable>* is assigned locally. If *<sequence>* is empty the *<token list variable>* is set to the special marker `\q_no_value`.

`\seq_get_right:NN`
`\seq_get_right:cN`

Updated: 2012-05-19

`\seq_get_right:NN <sequence> <token list variable>`

Stores the right-most item from a *<sequence>* in the *<token list variable>* without removing it from the *<sequence>*. The *<token list variable>* is assigned locally. If *<sequence>* is empty the *<token list variable>* is set to the special marker `\q_no_value`.

`\seq_pop_left:NN`
`\seq_pop_left:cN`

Updated: 2012-05-14

`\seq_pop_left:NN <sequence> <token list variable>`

Pops the left-most item from a *<sequence>* into the *<token list variable>*, i.e. removes the item from the sequence and stores it in the *<token list variable>*. Both of the variables are assigned locally. If *<sequence>* is empty the *<token list variable>* is set to the special marker `\q_no_value`.

`\seq_gpop_left:NN`
`\seq_gpop_left:cN`

Updated: 2012-05-14

`\seq_gpop_left:NN <sequence> <token list variable>`

Pops the left-most item from a *<sequence>* into the *<token list variable>*, i.e. removes the item from the sequence and stores it in the *<token list variable>*. The *<sequence>* is modified globally, while the assignment of the *<token list variable>* is local. If *<sequence>* is empty the *<token list variable>* is set to the special marker `\q_no_value`.

`\seq_pop_right:NN`
`\seq_pop_right:cN`

Updated: 2012-05-19

`\seq_pop_right:NN <sequence> <token list variable>`

Pops the right-most item from a *<sequence>* into the *<token list variable>*, i.e. removes the item from the sequence and stores it in the *<token list variable>*. Both of the variables are assigned locally. If *<sequence>* is empty the *<token list variable>* is set to the special marker `\q_no_value`.

`\seq_gpop_right:NN`
`\seq_gpop_right:cN`

Updated: 2012-05-19

`\seq_gpop_right:NN <sequence> <token list variable>`

Pops the right-most item from a *<sequence>* into the *<token list variable>*, i.e. removes the item from the sequence and stores it in the *<token list variable>*. The *<sequence>* is modified globally, while the assignment of the *<token list variable>* is local. If *<sequence>* is empty the *<token list variable>* is set to the special marker `\q_no_value`.

`\seq_item:Nn *`
`\seq_item:cN *`

New: 2014-07-17

`\seq_item:Nn <sequence> {\<integer expression>}`

Indexing items in the *<sequence>* from 1 at the top (left), this function evaluates the *<integer expression>* and leaves the appropriate item from the sequence in the input stream. If the *<integer expression>* is negative, indexing occurs from the bottom (right) of the sequence. If the *<integer expression>* is larger than the number of items in the *<sequence>* (as calculated by `\seq_count:N`) then the function expands to nothing.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* does not expand further when appearing in an x-type argument expansion.

```
\seq_rand_item:N *
\seq_rand_item:c *
```

New: 2016-12-06

```
\seq_rand_item:N <seq var>
```

Selects a pseudo-random item of the *<sequence>*. If the *<sequence>* is empty the result is empty. This is not available in older versions of X_ET_EX.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* does not expand further when appearing in an x-type argument expansion.

4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

```
\seq_get_left:NNTF
\seq_get_left:cNNTF
```

New: 2012-05-14

Updated: 2012-05-19

```
\seq_get_left:NNTF <sequence> <token list variable> {\<true code>} {\<false code>}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, stores the left-most item from the *<sequence>* in the *<token list variable>* without removing it from the *<sequence>*, then leaves the *<true code>* in the input stream. The *<token list variable>* is assigned locally.

```
\seq_get_right:NNTF
\seq_get_right:cNNTF
```

New: 2012-05-19

```
\seq_get_right:NNTF <sequence> <token list variable> {\<true code>} {\<false code>}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, stores the right-most item from the *<sequence>* in the *<token list variable>* without removing it from the *<sequence>*, then leaves the *<true code>* in the input stream. The *<token list variable>* is assigned locally.

```
\seq_pop_left:NNTF
\seq_pop_left:cNNTF
```

New: 2012-05-14

Updated: 2012-05-19

```
\seq_pop_left:NNTF <sequence> <token list variable> {\<true code>} {\<false code>}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, pops the left-most item from the *<sequence>* in the *<token list variable>*, i.e. removes the item from the *<sequence>*, then leaves the *<true code>* in the input stream. Both the *<sequence>* and the *<token list variable>* are assigned locally.

```
\seq_gpop_left:NNTF
\seq_gpop_left:cNNTF
```

New: 2012-05-14

Updated: 2012-05-19

```
\seq_gpop_left:NNTF <sequence> <token list variable> {\<true code>} {\<false code>}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, pops the left-most item from the *<sequence>* in the *<token list variable>*, i.e. removes the item from the *<sequence>*, then leaves the *<true code>* in the input stream. The *<sequence>* is modified globally, while the *<token list variable>* is assigned locally.

```
\seq_pop_right:NNTF  
\seq_pop_right:cNTF
```

New: 2012-05-19

```
\seq_pop_right:NNTF <sequence> <token list variable> {<true code>} {<false code>}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, pops the right-most item from the *<sequence>* in the *<token list variable>*, *i.e.* removes the item from the *<sequence>*, then leaves the *<true code>* in the input stream. Both the *<sequence>* and the *<token list variable>* are assigned locally.

```
\seq_gpop_right:NNTF  
\seq_gpop_right:cNTF
```

New: 2012-05-19

```
\seq_gpop_right:NNTF <sequence> <token list variable> {<true code>} {<false code>}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, pops the right-most item from the *<sequence>* in the *<token list variable>*, *i.e.* removes the item from the *<sequence>*, then leaves the *<true code>* in the input stream. The *<sequence>* is modified globally, while the *<token list variable>* is assigned locally.

5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

```
\seq_remove_duplicates:N  
\seq_remove_duplicates:c  
\seq_gremove_duplicates:N  
\seq_gremove_duplicates:c
```

```
\seq_remove_duplicates:N <sequence>
```

Removes duplicate items from the *<sequence>*, leaving the left most copy of each item in the *<sequence>*. The *<item>* comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

TeXhackers note: This function iterates through every item in the *<sequence>* and does a comparison with the *<items>* already checked. It is therefore relatively slow with large sequences.

```
\seq_remove_all:Nn  
\seq_remove_all:cn  
\seq_gremove_all:Nn  
\seq_gremove_all:cn
```

```
\seq_remove_all:Nn <sequence> {<item>}
```

Removes every occurrence of *<item>* from the *<sequence>*. The *<item>* comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

```
\seq_reverse:N  
\seq_reverse:c  
\seq_greverse:N  
\seq_greverse:c
```

New: 2014-07-18

```
\seq_reverse:N <sequence>
```

Reverses the order of the items stored in the *<sequence>*.

```
\seq_sort:Nn  
\seq_sort:cn  
\seq_gsort:Nn  
\seq_gsort:cn
```

New: 2017-02-06

```
\seq_sort:Nn <sequence> {<comparison code>}
```

Sorts the items in the *<sequence>* according to the *<comparison code>*, and assigns the result to *<sequence>*. The details of sorting comparison are described in Section 1.

```
\seq_shuffle:N  
\seq_shuffle:c  
\seq_gshuffle:N  
\seq_gshuffle:c
```

New: 2018-04-29

```
\seq_shuffle:N <seq var>
```

Sets the $\langle \text{seq} \var \rangle$ to the result of placing the items of the $\langle \text{seq} \var \rangle$ in a random order. Each item is (roughly) as likely to end up in any given position.

TeXhackers note: For sequences with more than 13 items or so, only a small proportion of all possible permutations can be reached, because the random seed `\sys_rand_seed`: only has 28-bits. The use of `\toks` internally means that sequences with more than 32767 or 65535 items (depending on the engine) cannot be shuffled.

6 Sequence conditionals

```
\seq_if_empty_p:N *  
\seq_if_empty_p:c *  
\seq_if_empty:NTF *  
\seq_if_empty:cTF *
```

Tests if the $\langle \text{sequence} \rangle$ is empty (containing no items).

```
\seq_if_in:NnTF  
\seq_if_in:(NV|Nv|No|Nx|cn|cV|cv|co|cx)TF
```

```
\seq_if_in:NnTF <sequence> {\langle item \rangle} {\langle true code \rangle} {\langle false code \rangle}
```

Tests if the $\langle \text{item} \rangle$ is present in the $\langle \text{sequence} \rangle$.

7 Mapping to sequences

All mappings are done at the current group level, *i.e.* any local assignments made by the $\langle \text{function} \rangle$ or $\langle \text{code} \rangle$ discussed below remain in effect after the loop.

```
\seq_map_function>NN *  
\seq_map_function:cN *
```

Updated: 2012-06-29

```
\seq_map_function:NN <sequence> <function>
```

Applies $\langle \text{function} \rangle$ to every $\langle \text{item} \rangle$ stored in the $\langle \text{sequence} \rangle$. The $\langle \text{function} \rangle$ will receive one argument for each iteration. The $\langle \text{items} \rangle$ are returned from left to right. To pass further arguments to the $\langle \text{function} \rangle$, see `\seq_map_tokens:Nn`. The function `\seq_map_inline:Nn` is faster than `\seq_map_function:NN` for sequences with more than about 10 items.

```
\seq_map_inline:Nn  
\seq_map_inline:cn
```

Updated: 2012-06-29

```
\seq_map_inline:Nn <sequence> {\langle inline function \rangle}
```

Applies $\langle \text{inline function} \rangle$ to every $\langle \text{item} \rangle$ stored within the $\langle \text{sequence} \rangle$. The $\langle \text{inline function} \rangle$ should consist of code which will receive the $\langle \text{item} \rangle$ as #1. The $\langle \text{items} \rangle$ are returned from left to right.

\seq_map_tokens:Nn ☆
\seq_map_tokens:cn ☆

New: 2019-08-30

\seq_map_tokens:Nn *sequence* {*code*}

Analogue of \seq_map_function:NN which maps several tokens instead of a single function. The *code* receives each item in the *sequence* as two trailing brace groups. For instance,

```
\seq_map_tokens:Nn \l_my_seq { \prg_replicate:nn { 2 } }
```

expands to twice each item in the *sequence*: for each item in \l_my_seq the function \prg_replicate:nn receives 2 and *item* as its two arguments. The function \seq_map_inline:Nn is typically faster but is not expandable.

\seq_map_variable:NNn
\seq_map_variable:(Ncn|cNn|ccn)

Updated: 2012-06-29

\seq_map_variable:NNn *sequence* *variable* {*code*}

Stores each *item* of the *sequence* in turn in the (token list) *variable* and applies the *code*. The *code* will usually make use of the *variable*, but this is not enforced. The assignments to the *variable* are local. Its value after the loop is the last *item* in the *sequence*, or its original value if the *sequence* is empty. The *items* are returned from left to right.

\seq_map_indexed_function:NN ☆

New: 2018-05-03

\seq_map_indexed_function:NN *seq var* *function*

Applies *function* to every entry in the *sequence variable*. The *function* should have signature :nn. It receives two arguments for each iteration: the *index* (namely 1 for the first entry, then 2 and so on) and the *item*.

\seq_map_indexed_inline:Nn
New: 2018-05-03

\seq_map_indexed_inline:Nn *seq var* {*inline function*}

Applies *inline function* to every entry in the *sequence variable*. The *inline function* should consist of code which receives the *index* (namely 1 for the first entry, then 2 and so on) as #1 and the *item* as #2.

\seq_map_break: ☆

Updated: 2012-06-29

\seq_map_break:

Used to terminate a `\seq_map_...` function before all entries in the *sequence* have been processed. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
    \str_if_eq:nnTF { #1 } { bingo }
        { \seq_map_break: }
        {
            % Do something useful
        }
}
```

Use outside of a `\seq_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\seq_map_break:n ☆

Updated: 2012-06-29

\seq_map_break:n {<code>}

Used to terminate a `\seq_map_...` function before all entries in the *sequence* have been processed, inserting the *code* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
    \str_if_eq:nnTF { #1 } { bingo }
        { \seq_map_break:n { <code> } }
        {
            % Do something useful
        }
}
```

Use outside of a `\seq_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *code* is inserted into the input stream. This depends on the design of the mapping function.

\seq_set_map:NNn

\seq_gset_map:NNn

New: 2011-12-22

Updated: 2020-07-16

\seq_set_map:NNn {sequence₁} {sequence₂} {<inline function>}

Applies *inline function* to every *item* stored within the *sequence₂*. The *inline function* should consist of code which will receive the *item* as #1. The sequence resulting applying *inline function* to each *item* is assigned to *sequence₁*.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level T_EX errors.

```
\seq_set_map_x:NNn
\seq_gset_map_x:NNn
```

New: 2020-07-16

```
\seq_set_map_x:NNn <sequence1> <sequence2> {<inline function>}
```

Applies *<inline function>* to every *<item>* stored within the *<sequence₂>*. The *<inline function>* should consist of code which will receive the *<item>* as #1. The sequence resulting from x-expanding *<inline function>* applied to each *<item>* is assigned to *<sequence₁>*. As such, the code in *<inline function>* should be expandable.

TEXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

```
\seq_count:N *
```

```
\seq_count:c *
```

New: 2012-07-13

```
\seq_count:N <sequence>
```

Leaves the number of items in the *<sequence>* in the input stream as an *<integer denotation>*. The total number of items in a *<sequence>* includes those which are empty and duplicates, *i.e.* every item in a *<sequence>* is unique.

8 Using the content of sequences directly

```
\seq_use:Nnnn *
```

```
\seq_use:cnnn *
```

New: 2013-05-26

```
\seq_use:Nnnn <seq var> {{separat or between two}}
{{separat or between more than two}} {{separat or between final two}}
```

Places the contents of the *<seq var>* in the input stream, with the appropriate *<separat or>* between the items. Namely, if the sequence has more than two items, the *<separat or between more than two>* is placed between each pair of items except the last, for which the *<separat or between final two>* is used. If the sequence has exactly two items, then they are placed in the input stream separated by the *<separat or between two>*. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<items>* do not expand further when appearing in an x-type argument expansion.

\seq_use:Nn *
\seq_use:cN *

New: 2013-05-26

\seq_use:Nn *seq var* {*separator*}

Places the contents of the *seq var* in the input stream, with the *separator* between the items. If the sequence has a single item, it is placed in the input stream with no *separator*, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the *items* do not expand further when appearing in an x-type argument expansion.

9 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

\seq_get:NN
\seq_get:cN

Updated: 2012-05-14

\seq_get:NN *sequence* {*token list variable*}

Reads the top item from a *sequence* into the *token list variable* without removing it from the *sequence*. The *token list variable* is assigned locally. If *sequence* is empty the *token list variable* is set to the special marker \q_no_value.

\seq_pop:NN
\seq_pop:cN

Updated: 2012-05-14

\seq_pop:NN *sequence* {*token list variable*}

Pops the top item from a *sequence* into the *token list variable*. Both of the variables are assigned locally. If *sequence* is empty the *token list variable* is set to the special marker \q_no_value.

\seq_gpop:NN
\seq_gpop:cN

Updated: 2012-05-14

\seq_gpop:NN *sequence* {*token list variable*}

Pops the top item from a *sequence* into the *token list variable*. The *sequence* is modified globally, while the *token list variable* is assigned locally. If *sequence* is empty the *token list variable* is set to the special marker \q_no_value.

\seq_get:NNTF
\seq_get:cNTF

New: 2012-05-14
Updated: 2012-05-19

\seq_get:NNTF *sequence* {*token list variable*} {*true code*} {*false code*}

If the *sequence* is empty, leaves the *false code* in the input stream. The value of the *token list variable* is not defined in this case and should not be relied upon. If the *sequence* is non-empty, stores the top item from a *sequence* in the *token list variable* without removing it from the *sequence*. The *token list variable* is assigned locally.

```
\seq_pop:NNTF
\seq_pop:cNTF
```

New: 2012-05-14
Updated: 2012-05-19

```
\seq_pop:NNTF <sequence> <token list variable> {<true code>} {<false code>}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, pops the top item from the *<sequence>* in the *<token list variable>*, *i.e.* removes the item from the *<sequence>*. Both the *<sequence>* and the *<token list variable>* are assigned locally.

```
\seq_gpop:NNTF
\seq_gpop:cNTF
```

New: 2012-05-14
Updated: 2012-05-19

```
\seq_gpop:NNTF <sequence> <token list variable> {<true code>} {<false code>}
```

If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<sequence>* is non-empty, pops the top item from the *<sequence>* in the *<token list variable>*, *i.e.* removes the item from the *<sequence>*. The *<sequence>* is modified globally, while the *<token list variable>* is assigned locally.

```
\seq_push:Nn
\seq_push:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gpush:Nn
\seq_gpush:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
```

```
\seq_push:Nn <sequence> {<item>}
```

Adds the *{<item>}* to the top of the *<sequence>*.

10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurrences of a given item. To make sure that a *<sequence variable>* only has distinct items, use `\seq_remove_duplicates:N <sequence variable>`. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set *<seq var>* are straightforward. For instance, `\seq_count:N <seq var>` expands to the number of items, while `\seq_if_in:NnTF <seq var> {<item>}` tests if the *<item>* is in the set.

Adding an *<item>* to a set *<seq var>* can be done by appending it to the *<seq var>* if it is not already in the *<seq var>*:

```
\seq_if_in:NnF <seq var> {<item>}
{ \seq_put_right:Nn <seq var> {<item>} }
```

Removing an *<item>* from a set *<seq var>* can be done using `\seq_remove_all:Nn`,

```
\seq_remove_all:Nn <seq var> {<item>}
```

The intersection of two sets *<seq var₁>* and *<seq var₂>* can be stored into *<seq var₃>* by collecting items of *<seq var₁>* which are in *<seq var₂>*.

```
\seq_clear:N <seq var3>
\seq_map_inline:Nn <seq var1>
{
\seq_if_in:NnT <seq var2> {#1}
{ \seq_put_right:Nn <seq var3> {#1} }
}
```

The code as written here only works if $\langle \text{seq } \text{var}_3 \rangle$ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence $\backslash \text{l_}\langle \text{pkg} \rangle\text{_internal_seq}$, then $\langle \text{seq } \text{var}_3 \rangle$ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets $\langle \text{seq } \text{var}_1 \rangle$ and $\langle \text{seq } \text{var}_2 \rangle$ can be stored into $\langle \text{seq } \text{var}_3 \rangle$ through

```
\seq_concat:NNN <seq var3> <seq var1> <seq var2>
\seq_remove_duplicates:N <seq var3>
```

or by adding items to (a copy of) $\langle \text{seq } \text{var}_1 \rangle$ one by one

```
\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{
\seq_if_in:NnF <seq var3> {#1}
{ \seq_put_right:Nn <seq var3> {#1} }
}
```

The second approach is faster than the first when the $\langle \text{seq } \text{var}_2 \rangle$ is short compared to $\langle \text{seq } \text{var}_1 \rangle$.

The difference of two sets $\langle \text{seq } \text{var}_1 \rangle$ and $\langle \text{seq } \text{var}_2 \rangle$ can be stored into $\langle \text{seq } \text{var}_3 \rangle$ by removing items of the $\langle \text{seq } \text{var}_2 \rangle$ from (a copy of) the $\langle \text{seq } \text{var}_1 \rangle$ one by one.

```
\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn <seq var3> {#1} }
```

The symmetric difference of two sets $\langle \text{seq } \text{var}_1 \rangle$ and $\langle \text{seq } \text{var}_2 \rangle$ can be stored into $\langle \text{seq } \text{var}_3 \rangle$ by computing the difference between $\langle \text{seq } \text{var}_1 \rangle$ and $\langle \text{seq } \text{var}_2 \rangle$ and storing the result as $\backslash \text{l_}\langle \text{pkg} \rangle\text{_internal_seq}$, then the difference between $\langle \text{seq } \text{var}_2 \rangle$ and $\langle \text{seq } \text{var}_1 \rangle$, and finally concatenating the two differences to get the symmetric differences.

```
\seq_set_eq:NN \l_\langle \text{pkg} \rangle\text{\_internal\_seq} <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn \l_\langle \text{pkg} \rangle\text{\_internal\_seq} {#1} }
\seq_set_eq:NN <seq var3> <seq var2>
\seq_map_inline:Nn <seq var1>
{ \seq_remove_all:Nn <seq var3> {#1} }
\seq_concat:NNN <seq var3> <seq var3> \l_\langle \text{pkg} \rangle\text{\_internal\_seq}
```

11 Constant and scratch sequences

\c_empty_seq Constant that is always empty.

New: 2012-07-02

\l_tmpa_seq
\l_tmpb_seq
New: 2012-04-26

Scratch sequences for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_seq
\g_tmpb_seq
New: 2012-04-26

Scratch sequences for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

12 Viewing sequences

\seq_show:N
\seq_show:c
Updated: 2015-08-01

\seq_show:N *(sequence)*
Displays the entries in the *(sequence)* in the terminal.

\seq_log:N
\seq_log:c
New: 2014-08-12
Updated: 2015-08-01

\seq_log:N *(sequence)*
Writes the entries in the *(sequence)* in the log file.

Part XI

The **I3int** package

Integers

Calculation and comparison of integer values can be carried out using literal numbers, **int** registers, constants and integers stored in token list variables. The standard operators **+**, **-**, **/** and ***** and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on *integer expressions* (“**intexpr**”).

1 Integer expressions

```
\int_eval:n *
```

```
\int_eval:n {<integer expression>}
```

Evaluates the $\langle\text{integer expression}\rangle$ and leaves the result in the input stream as an integer denotation: for positive results an explicit sequence of decimal digits not starting with 0, for negative results – followed by such a sequence, and 0 for zero. The $\langle\text{integer expression}\rangle$ should consist, after expansion, of $+$, $-$, $*$, $/$, $($, $)$ and of course integer operands. The result is calculated by applying standard mathematical rules with the following peculiarities:

- $/$ denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds $2^{31} - 1$, except in the case of scaling operations $a*b/c$, for which $a*b$ may be arbitrarily large;
- parentheses may not appear after unary $+$ or $-$, namely placing $+()$ or $-()$ at the start of an expression or after $+$, $-$, $*$, $/$ or $($ leads to an error.

Each integer operand can be either an integer variable (with no need for `\int_use:N`) or an integer denotation. For example both

```
\int_eval:n { 5 + 4 * 3 - ( 3 + 4 * 5 ) }
```

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { 5 }
\int_new:N \l_my_int
\int_set:Nn \l_my_int { 4 }
\int_eval:n { \l_my_tl + \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

evaluate to -6 because `\l_my_tl` expands to the integer denotation 5. As the $\langle\text{integer expression}\rangle$ is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and `\exp_not:n` and its variants have no effect while `\exp_not:N` may incorrectly interrupt the expression.

TeXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an $\langle\text{internal integer}\rangle$, and therefore requires suitable termination if used in a TeX-style integer assignment.

As all TeX integers, integer operands can also be dimension or skip variables, converted to integers in `sp`, or octal numbers given as ‘`\`’ followed by digits other than 8 and 9, or hexadecimal numbers given as ‘`\#`’ followed by digits or upper case letters from A to F, or the character code of some character or one-character control sequence, given as ‘ $\langle\text{char}\rangle$ ’.

`\int_eval:w *`

New: 2018-03-30

`\int_eval:w <integer expression>`

Evaluates the $\langle\text{integer expression}\rangle$ as described for `\int_eval:n`. The end of the expression is the first token encountered that cannot form part of such an expression. If that token is `\scan_stop`: it is removed, otherwise not. Spaces do *not* terminate the expression. However, spaces terminate explicit integers, and this may terminate the expression: for instance, `\int_eval:w 1+u1+u9` expands to 29 since the digit 9 is not part of the expression.

`\int_sign:n *`

New: 2018-11-03

`\int_sign:n {<intexpr>}`

Evaluates the $\langle\text{integer expression}\rangle$ then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

`\int_abs:n *`

Updated: 2012-09-26

`\int_abs:n {<integer expression>}`

Evaluates the $\langle\text{integer expression}\rangle$ as described for `\int_eval:n` and leaves the absolute value of the result in the input stream as an $\langle\text{integer denotation}\rangle$ after two expansions.

`\int_div_round:nn *`

Updated: 2012-09-26

`\int_div_round:nn {<intexpr1>} {<intexpr2>}`

Evaluates the two $\langle\text{integer expressions}\rangle$ as described earlier, then divides the first value by the second, and rounds the result to the closest integer. Ties are rounded away from zero. Note that this is identical to using `/` directly in an $\langle\text{integer expression}\rangle$. The result is left in the input stream as an $\langle\text{integer denotation}\rangle$ after two expansions.

`\int_div_truncate:nn *`

Updated: 2012-02-09

`\int_div_truncate:nn {<intexpr1>} {<intexpr2>}`

Evaluates the two $\langle\text{integer expressions}\rangle$ as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using `/` rounds to the closest integer instead. The result is left in the input stream as an $\langle\text{integer denotation}\rangle$ after two expansions.

`\int_max:nn *`

`\int_min:nn *`

Updated: 2012-09-26

`\int_max:nn {<intexpr1>} {<intexpr2>}`

`\int_min:nn {<intexpr1>} {<intexpr2>}`

Evaluates the $\langle\text{integer expressions}\rangle$ as described for `\int_eval:n` and leaves either the larger or smaller value in the input stream as an $\langle\text{integer denotation}\rangle$ after two expansions.

`\int_mod:nn *`

Updated: 2012-09-26

`\int_mod:nn {<intexpr1>} {<intexpr2>}`

Evaluates the two $\langle\text{integer expressions}\rangle$ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting `\int_div_truncate:nn {<intexpr1>} {<intexpr2>}` times $\langle\text{intexpr2}\rangle$ from $\langle\text{intexpr1}\rangle$. Thus, the result has the same sign as $\langle\text{intexpr1}\rangle$ and its absolute value is strictly less than that of $\langle\text{intexpr2}\rangle$. The result is left in the input stream as an $\langle\text{integer denotation}\rangle$ after two expansions.

2 Creating and initialising integers

`\int_new:N <integer>`

`\int_new:c`

Creates a new $\langle\text{integer}\rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle\text{integer}\rangle$ is initially equal to 0.

```
\int_const:Nn
\int_const:cn
```

Updated: 2011-10-22

```
\int_const:Nn <integer> {{integer expression}}
```

Creates a new constant *<integer>* or raises an error if the name is already taken. The value of the *<integer>* is set globally to the *<integer expression>*.

```
\int_zero:N
\int_zero:c
\int_gzero:N
\int_gzero:c
```

```
\int_zero:N <integer>
```

Sets *<integer>* to 0.

```
\int_zero_new:N
\int_zero_new:c
\int_gzero_new:N
\int_gzero_new:c
```

New: 2011-12-13

```
\int_zero_new:N <integer>
```

Ensures that the *<integer>* exists globally by applying \int_new:N if necessary, then applies \int_(g)zero:N to leave the *<integer>* set to zero.

```
\int_set_eq:NN
\int_set_eq:(cN|Nc|cc)
\int_gset_eq:NN
\int_gset_eq:(cN|Nc|cc)
```

```
\int_set_eq:NN <integer1> <integer2>
```

Sets the content of *<integer₁>* equal to that of *<integer₂>*.

```
\int_if_exist_p:N *
\int_if_exist_p:c *
\int_if_exist:NTF *
\int_if_exist:cTF *
```

New: 2012-03-03

```
\int_if_exist_p:N <int>
```

```
\int_if_exist:NTF <int> {{true code}} {{false code}}
```

Tests whether the *<int>* is currently defined. This does not check that the *<int>* really is an integer variable.

3 Setting and incrementing integers

```
\int_add:Nn
\int_add:cn
\int_gadd:Nn
\int_gadd:cn
```

Updated: 2011-10-22

```
\int_add:Nn <integer> {{integer expression}}
```

Adds the result of the *<integer expression>* to the current content of the *<integer>*.

```
\int_decr:N
\int_decr:c
\int_gdecr:N
\int_gdecr:c
```

```
\int_decr:N <integer>
```

Decreases the value stored in *<integer>* by 1.

```
\int_incr:N
\int_incr:c
\int_gincr:N
\int_gincr:c
```

```
\int_incr:N <integer>
```

Increases the value stored in *<integer>* by 1.

```
\int_set:Nn      \int_set:Nn <integer> {<integer expression>}
\int_set:cn      Sets <integer> to the value of <integer expression>, which must evaluate to an integer (as
\int_gset:Nn      described for \int_eval:n).
\int_gset:cn
```

Updated: 2011-10-22

```
\int_sub:Nn      \int_sub:Nn <integer> {<integer expression>}
\int_sub:cn      Subtracts the result of the <integer expression> from the current content of the <integer>.
\int_gsub:Nn
\int_gsub:cn
```

Updated: 2011-10-22

4 Using integers

```
\int_use:N      * \int_use:N <integer>
```

Recovers the content of an *<integer>* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where an *<integer>* is required (such as in the first and third arguments of *\int_compare:nNnTF*).

TEXhackers note: *\int_use:N* is the TeX primitive *\the*: this is one of several L^AT_EX3 names for this primitive.

5 Integer expression conditionals

```
\int_compare_p:nNn * \int_compare_p:nNn {<intexpr1>} <relation> {<intexpr2>}
\int_compare:nNnTF
  {<intexpr1>} <relation> {<intexpr2>}
  {<true code>} {<false code>}
```

This function first evaluates each of the *<integer expressions>* as described for *\int_eval:n*. The two results are then compared using the *<relation>*:

Equal	=
Greater than	>
Less than	<

This function is less flexible than *\int_compare:nTF* but around 5 times faster.

```
\int_compare_p:n *
\int_compare:nTF *

```

Updated: 2013-01-13

```
\int_compare_p:n
{
  ⟨intexpr_1⟩ ⟨relation_1⟩
  ...
  ⟨intexpr_N⟩ ⟨relation_N⟩
  ⟨intexpr_{N+1}⟩
}
\int_compare:nTF
{
  ⟨intexpr_1⟩ ⟨relation_1⟩
  ...
  ⟨intexpr_N⟩ ⟨relation_N⟩
  ⟨intexpr_{N+1}⟩
}
{⟨true code⟩} {⟨false code⟩}
```

This function evaluates the $\langle \text{integer expressions} \rangle$ as described for `\int_eval:n` and compares consecutive result using the corresponding $\langle \text{relation} \rangle$, namely it compares $\langle \text{intexpr}_1 \rangle$ and $\langle \text{intexpr}_2 \rangle$ using the $\langle \text{relation}_1 \rangle$, then $\langle \text{intexpr}_2 \rangle$ and $\langle \text{intexpr}_3 \rangle$ using the $\langle \text{relation}_2 \rangle$, until finally comparing $\langle \text{intexpr}_N \rangle$ and $\langle \text{intexpr}_{N+1} \rangle$ using the $\langle \text{relation}_N \rangle$. The test yields `true` if all comparisons are `true`. Each $\langle \text{integer expression} \rangle$ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other $\langle \text{integer expression} \rangle$ is evaluated and no other comparison is performed. The $\langle \text{relations} \rangle$ can be any of the following:

Equal	= or ==
Greater than or equal to	\geq
Greater than	$>$
Less than or equal to	\leq
Less than	$<$
Not equal	\neq

This function is more flexible than `\int_compare:nNnTF` but around 5 times slower.

```
\int_case:nn   *
\int_case:nnTF *

```

New: 2013-07-24

```
\int_case:nnTF {\test integer expression}
{
  {\intexpr case1} {\code case1}
  {\intexpr case2} {\code case2}
  ...
  {\intexpr casen} {\code casen}
}
{\true code}
{\false code}
```

This function evaluates the *<test integer expression>* and compares this in turn to each of the *<integer expression cases>*. If the two are equal then the associated *<code>* is left in the input stream and other cases are discarded. If any of the cases are matched, the *<true code>* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *<false code>* is inserted. The function `\int_case:nn`, which does nothing if there is no match, is also available. For example

```
\int_case:nnF
{ 2 * 5 }
{
  { 5 }      { Small }
  { 4 + 6 }  { Medium }
  { -2 * 10 } { Negative }
}
{ No idea! }
```

leaves “Medium” in the input stream.

```
\int_if_even_p:n *
\int_if_even:nTF *
\int_if_odd_p:n *
\int_if_odd:nTF *
```

This function first evaluates the *<integer expression>* as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate.

6 Integer expression loops

```
\int_do_until:nNnn ☆ \int_do_until:nNnn {\intexpr1} <relation> {\intexpr2} {\code}
```

Places the *<code>* in the input stream for TeX to process, and then evaluates the relationship between the two *<integer expressions>* as described for `\int_compare:nNnTF`. If the test is **false** then the *<code>* is inserted into the input stream again and a loop occurs until the *<relation>* is **true**.

```
\int_do_while:nNnn ☆ \int_do_while:nNnn {\intexpr1} <relation> {\intexpr2} {\code}
```

Places the *<code>* in the input stream for TeX to process, and then evaluates the relationship between the two *<integer expressions>* as described for `\int_compare:nNnTF`. If the test is **true** then the *<code>* is inserted into the input stream again and a loop occurs until the *<relation>* is **false**.

<code>\int_until_do:nNnn</code> ☆	<code>\int_until_do:nNnn {\langle intexpr_1 \rangle} {\langle relation \rangle} {\langle intexpr_2 \rangle} {\langle code \rangle}</code>
	Evaluates the relationship between the two <i><integer expressions></i> as described for <code>\int_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is <code>false</code> . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\int_while_do:nNnn</code> ☆	<code>\int_while_do:nNnn {\langle intexpr_1 \rangle} {\langle relation \rangle} {\langle intexpr_2 \rangle} {\langle code \rangle}</code>
	Evaluates the relationship between the two <i><integer expressions></i> as described for <code>\int_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is <code>true</code> . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>false</code> .
<code>\int_do_until:nn</code> ☆ Updated: 2013-01-13	<code>\int_do_until:nn {\langle integer relation \rangle} {\langle code \rangle}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> . If the test is <code>false</code> then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is <code>true</code> .
<code>\int_do_while:nn</code> ☆ Updated: 2013-01-13	<code>\int_do_while:nn {\langle integer relation \rangle} {\langle code \rangle}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> . If the test is <code>true</code> then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is <code>false</code> .
<code>\int_until_do:nn</code> ☆ Updated: 2013-01-13	<code>\int_until_do:nn {\langle integer relation \rangle} {\langle code \rangle}</code>
	Evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is <code>false</code> . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\int_while_do:nn</code> ☆ Updated: 2013-01-13	<code>\int_while_do:nn {\langle integer relation \rangle} {\langle code \rangle}</code>
	Evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is <code>true</code> . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>false</code> .

7 Integer step functions

```
\int_step_function:nN  ☆
\int_step_function:nnN  ☆
\int_step_function:nnnN  ☆
```

New: 2012-06-04
Updated: 2018-04-22

```
\int_step_function:nN {<final value>} <function>
\int_step_function:nnN {<initial value>} {<final value>} <function>
\int_step_function:nnnN {<initial value>} {<step>} {<final value>} <function>
```

This function first evaluates the $\langle initial \ value \rangle$, $\langle step \rangle$ and $\langle final \ value \rangle$, all of which should be integer expressions. The $\langle function \rangle$ is then placed in front of each $\langle value \rangle$ from the $\langle initial \ value \rangle$ to the $\langle final \ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$). The $\langle step \rangle$ must be non-zero. If the $\langle step \rangle$ is positive, the loop stops when the $\langle value \rangle$ becomes larger than the $\langle final \ value \rangle$. If the $\langle step \rangle$ is negative, the loop stops when the $\langle value \rangle$ becomes smaller than the $\langle final \ value \rangle$. The $\langle function \rangle$ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\int_step_function:nnnN { 1 } { 1 } { 5 } \my_func:n
```

would print

```
[I saw 1] [I saw 2] [I saw 3] [I saw 4] [I saw 5]
```

The functions $\int_step_function:nN$ and $\int_step_function:nnN$ both use a fixed $\langle step \rangle$ of 1, and in the case of $\int_step_function:nN$ the $\langle initial \ value \rangle$ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

```
\int_step_inline:nn
\int_step_inline:nnn
\int_step_inline:nnnn
```

New: 2012-06-04
Updated: 2018-04-22

```
\int_step_inline:nn {<final value>} <code>
\int_step_inline:nnn {<initial value>} {<final value>} <code>
\int_step_inline:nnnn {<initial value>} {<step>} {<final value>} <code>
```

This function first evaluates the $\langle initial \ value \rangle$, $\langle step \rangle$ and $\langle final \ value \rangle$, all of which should be integer expressions. Then for each $\langle value \rangle$ from the $\langle initial \ value \rangle$ to the $\langle final \ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream with $\#1$ replaced by the current $\langle value \rangle$. Thus the $\langle code \rangle$ should define a function of one argument ($\#1$).

The functions $\int_step_inline:nn$ and $\int_step_inline:nnn$ both use a fixed $\langle step \rangle$ of 1, and in the case of $\int_step_inline:nn$ the $\langle initial \ value \rangle$ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

```
\int_step_variable:nNn
\int_step_variable:nnNn
\int_step_variable:nnnNn
```

New: 2012-06-04
Updated: 2018-04-22

```
\int_step_variable:nNn {<final value>} <tl var> <code>
\int_step_variable:nnNn {<initial value>} {<final value>} <tl var> <code>
\int_step_variable:nnnNn {<initial value>} {<step>} {<final value>} <tl var>
<code>
```

This function first evaluates the $\langle initial \ value \rangle$, $\langle step \rangle$ and $\langle final \ value \rangle$, all of which should be integer expressions. Then for each $\langle value \rangle$ from the $\langle initial \ value \rangle$ to the $\langle final \ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream, with the $\langle tl \ var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl \ var \rangle$.

The functions $\int_step_variable:nNn$ and $\int_step_variable:nnNn$ both use a fixed $\langle step \rangle$ of 1, and in the case of $\int_step_variable:nNn$ the $\langle initial \ value \rangle$ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

\int_to_arabic:n *

Updated: 2011-10-22

\int_to_arabic:n {<integer expression>}

Places the value of the *<integer expression>* in the input stream as digits, with category code 12 (other).

\int_to_alpha:n *

\int_to_Alph:n *

Updated: 2011-09-17

\int_to_alpha:n {<integer expression>}

Evaluates the *<integer expression>* and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

```
\int_to_alpha:n { 1 }
```

places **a** in the input stream,

```
\int_to_alpha:n { 26 }
```

is represented as **z** and

```
\int_to_alpha:n { 27 }
```

is converted to **aa**. For conversions using other alphabets, use \int_to_symbols:nnn to define an alphabet-specific function. The basic \int_to_alpha:n and \int_to_Alph:n functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\int_to_symbols:nnn *

Updated: 2011-09-17

\int_to_symbols:nnn

{<integer expression>} {<total symbols>}
{<value to symbol mapping>}

This is the low-level function for conversion of an *<integer expression>* into a symbolic form (often letters). The *<total symbols>* available should be given as an integer expression. Values are actually converted to symbols according to the *<value to symbol mapping>*. This should be given as *<total symbols>* pairs of entries, a number and the appropriate symbol. Thus the \int_to_alpha:n function is defined as

```
\cs_new:Npn \int_to_alpha:n #1
{
    \int_to_symbols:nnn {#1} { 26 }
    {
        { 1 } { a }
        { 2 } { b }
        ...
        { 26 } { z }
    }
}
```

<code>\int_to_bin:n</code> *	<code>\int_to_bin:n {<integer expression>}</code>
<small>New: 2014-02-11</small>	Calculates the value of the <i><integer expression></i> and places the binary representation of the result in the input stream.
<code>\int_to_hex:n</code> *	<code>\int_to_hex:n {<integer expression>}</code>
<code>\int_to_Hex:n</code> *	<code>\int_to_hex:n {<integer expression>}</code>
<small>New: 2014-02-11</small>	Calculates the value of the <i><integer expression></i> and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for <code>\int_to_hex:n</code> and upper case ones for <code>\int_to_Hex:n</code> . The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
<code>\int_to_oct:n</code> *	<code>\int_to_oct:n {<integer expression>}</code>
<small>New: 2014-02-11</small>	Calculates the value of the <i><integer expression></i> and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
<code>\int_to_base:nn</code> *	<code>\int_to_base:nn {<integer expression>} {<base>}</code>
<code>\int_to_Base:nn</code> *	<code>\int_to_base:nn {<integer expression>} {<base>}</code>
<small>Updated: 2014-02-11</small>	Calculates the value of the <i><integer expression></i> and converts it into the appropriate representation in the <i><base></i> ; the latter may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by letters from the English alphabet: lower case letters for <code>\int_to_base:n</code> and upper case ones for <code>\int_to_Base:n</code> . The maximum <i><base></i> value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

TeXhackers note: This is a generic version of `\int_to_bin:n`, etc.

<code>\int_to_roman:n</code> ☆	<code>\int_to_roman:n {<integer expression>}</code>
<code>\int_to_Roman:n</code> ☆	<code>\int_to_roman:n {<integer expression>}</code>
<small>Updated: 2011-10-22</small>	Places the value of the <i><integer expression></i> in the input stream as Roman numerals, either lower case (<code>\int_to_roman:n</code>) or upper case (<code>\int_to_Roman:n</code>). If the value is negative or zero, the output is empty. The Roman numerals are letters with category code 11 (letter). The letters used are mdclxvi, repeated as needed: the notation with bars (such as \bar{v} for 5000) is <i>not</i> used. For instance <code>\int_to_roman:n { 8249 }</code> expands to <code>mmmmmmmmccxliv</code> .

9 Converting from other formats to integers

<code>\int_from_alpha:n</code> *	<code>\int_from_alpha:n {<letters>}</code>
<small>Updated: 2014-08-25</small>	Converts the <i><letters></i> into the integer (base 10) representation and leaves this in the input stream. The <i><letters></i> are first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with “a” equal to 1 through to “z” equal to 26. The function also accepts a leading sign, made of + and -. This is the inverse function of <code>\int_to_alpha:n</code> and <code>\int_to_Alpha:n</code> .

```
\int_from_bin:n *
```

New: 2014-02-11
Updated: 2014-08-25

```
\int_from_bin:n {\langle binary number \rangle}
```

Converts the $\langle \text{binary number} \rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle \text{binary number} \rangle$ is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by binary digits. This is the inverse function of `\int_to_bin:n`.

```
\int_from_hex:n *
```

New: 2014-02-11
Updated: 2014-08-25

```
\int_from_hex:n {\langle hexadecimal number \rangle}
```

Converts the $\langle \text{hexadecimal number} \rangle$ into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the $\langle \text{hexadecimal number} \rangle$ by upper or lower case letters. The $\langle \text{hexadecimal number} \rangle$ is first converted to a string, with no expansion. The function also accepts a leading sign, made of + and -. This is the inverse function of `\int_to_hex:n` and `\int_to_Hex:n`.

```
\int_from_oct:n *
```

New: 2014-02-11
Updated: 2014-08-25

```
\int_from_oct:n {\langle octal number \rangle}
```

Converts the $\langle \text{octal number} \rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle \text{octal number} \rangle$ is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by octal digits. This is the inverse function of `\int_to_oct:n`.

```
\int_from_roman:n *
```

Updated: 2014-08-25

```
\int_from_roman:n {\langle roman numeral \rangle}
```

Converts the $\langle \text{roman numeral} \rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle \text{roman numeral} \rangle$ is first converted to a string, with no expansion. The $\langle \text{roman numeral} \rangle$ may be in upper or lower case; if the numeral contains characters besides mdclxvi or MDCLXVI then the resulting value is -1. This is the inverse function of `\int_to_roman:n` and `\int_to_Roman:n`.

```
\int_from_base:nn *
```

Updated: 2014-08-25

```
\int_from_base:nn {\langle number \rangle} {\langle base \rangle}
```

Converts the $\langle \text{number} \rangle$ expressed in $\langle \text{base} \rangle$ into the appropriate value in base 10. The $\langle \text{number} \rangle$ is first converted to a string, with no expansion. The $\langle \text{number} \rangle$ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum $\langle \text{base} \rangle$ value is 36. This is the inverse function of `\int_to_base:nn` and `\int_to_Base:nn`.

10 Random integers

```
\int_rand:nn *
```

New: 2016-12-06
Updated: 2018-04-27

```
\int_rand:nn {\langle intexpr_1 \rangle} {\langle intexpr_2 \rangle}
```

Evaluates the two $\langle \text{integer expressions} \rangle$ and produces a pseudo-random number between the two (with bounds included). This is not available in older versions of X_ET_EX.

```
\int_rand:n *
```

New: 2018-05-05

```
\int_rand:n {\langle intexpr \rangle}
```

Evaluates the $\langle \text{integer expression} \rangle$ then produces a pseudo-random number between 1 and the $\langle \text{intexpr} \rangle$ (included). This is not available in older versions of X_ET_EX.

11 Viewing integers

<u>\int_show:N</u>	<code>\int_show:N <integer></code>
<u>\int_show:c</u>	Displays the value of the <i><integer></i> on the terminal.
 <u>\int_show:n</u>	<code>\int_show:n {<integer expression>}</code>
New: 2011-11-22	Displays the result of evaluating the <i><integer expression></i> on the terminal.
Updated: 2015-08-07	
 <u>\int_log:N</u>	<code>\int_log:N <integer></code>
<u>\int_log:c</u>	Writes the value of the <i><integer></i> in the log file.
New: 2014-08-22	
Updated: 2015-08-03	
 <u>\int_log:n</u>	<code>\int_log:n {<integer expression>}</code>
New: 2014-08-22	Writes the result of evaluating the <i><integer expression></i> in the log file.
Updated: 2015-08-07	

12 Constant integers

<u>\c_zero_int</u>	Integer values used with primitive tests and assignments: their self-terminating nature makes these more convenient and faster than literal numbers.
<u>\c_one_int</u>	
New: 2018-05-07	
 <u>\c_max_int</u>	The maximum value that can be stored as an integer.
 <u>\c_max_register_int</u>	Maximum number of registers.
 <u>\c_max_char_int</u>	Maximum character code completely supported by the engine.

13 Scratch integers

<u>\l_tmpa_int</u>	Scratch integer for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<u>\g_tmpa_int</u>	Scratch integer for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

13.1 Direct number expansion

```
\int_value:w *
```

New: 2018-03-27

`\int_value:w <integer>`

`\int_value:w <integer denotation> <optional space>`

Expands the following tokens until an `<integer>` is formed, and leaves a normalized form (no leading sign except for negative numbers, no leading digit 0 except for zero) in the input stream as category code 12 (other) characters. The `<integer>` can consist of any number of signs (with intervening spaces) followed by

- an integer variable (in fact, any TeX register except `\toks`) or
- explicit digits (or by '`<octal digits>`' or "`<hexadecimal digits>`" or '`<character>`').

In this last case expansion stops once a non-digit is found; if that is a space it is removed as in f-expansion, and so `\exp_stop_f:` may be employed as an end marker. Note that protected functions *are* expanded by this process.

This function requires exactly one expansion to produce a value, and so is suitable for use in cases where a number is required "directly". In general, `\int_eval:n` is the preferred approach to generating numbers.

TeXhackers note: This is the TeX primitive `\number`.

14 Primitive conditionals

```
\if_int_compare:w *
```

`\if_int_compare:w <integer1> <relation> <integer2>`
 `<true code>`
`\else:`
 `<false code>`
`\fi:`

Compare two integers using `<relation>`, which must be one of =, < or > with category code 12. The `\else:` branch is optional.

TeXhackers note: These are both names for the TeX primitive `\ifnum`.

```
\if_case:w *
```

`\if_case:w <integer> <case0>`
 `\or: <case1>`
 `\or: ...`
 `\else: <default>`
`\fi:`

Selects a case to execute based on the value of the `<integer>`. The first case (`(case0)`) is executed if `<integer>` is 0, the second (`(case1)`) if the `<integer>` is 1, etc. The `<integer>` may be a literal, a constant or an integer expression (e.g. using `\int_eval:n`).

TeXhackers note: These are the TeX primitives `\ifcase` and `\or`.

```
\if_int_odd:w ∗ \if_int_odd:w ⟨tokens⟩ ⟨optional space⟩
    ⟨true code⟩
\else:
    ⟨true code⟩
\fi:
```

Expands ⟨tokens⟩ until a non-numeric token or a space is found, and tests whether the resulting ⟨integer⟩ is odd. If so, ⟨true code⟩ is executed. The \else: branch is optional.

TeXhackers note: This is the TeX primitive \ifodd.

Part XII

The **l3flag** package: Expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any non-negative value, which we call its *height*. In expansion-only contexts, a flag can only be “raised”: this increases the *height* by 1. The *height* can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local. They are referenced by a *flag name* such as `str-missing`. The *flag name* is used as part of `\use:c` constructions hence is expanded at point of use. It must expand to character tokens only, with no spaces.

A typical use case of flags would be to keep track of whether an exceptional condition has occurred during expandable processing, and produce a meaningful (non-expandable) message after the end of the expandable processing. This is exemplified by `l3str-convert`, which for performance reasons performs conversions of individual characters expandably and for readability reasons produces a single error message describing incorrect inputs that were encountered.

Flags should not be used without carefully considering the fact that raising a flag takes a time and memory proportional to its height. Flags should not be used unless unavoidable.

1 Setting up flags

`\flag_new:n {flag name}`

Creates a new flag with a name given by *flag name*, or raises an error if the name is already taken. The *flag name* may not contain spaces. The declaration is global, but flags are always local variables. The *flag* initially has zero height.

`\flag_clear:n {flag name}`

The *flag*’s height is set to zero. The assignment is local.

`\flag_clear_new:n {flag name}`

Ensures that the *flag* exists globally by applying `\flag_new:n` if necessary, then applies `\flag_clear:n`, setting the height to zero locally.

`\flag_show:n {flag name}`

Displays the *flag*’s height in the terminal.

`\flag_log:n {flag name}`

Writes the *flag*’s height to the log file.

2 Expandable flag commands

`\flag_if_exist_p:n *` `\flag_if_exist:n {<flag name>}`

This function returns `true` if the `<flag name>` references a flag that has been defined previously, and `false` otherwise.

`\flag_if_raised_p:n *` `\flag_if_raised:n {<flag name>}`

This function returns `true` if the `<flag>` has non-zero height, and `false` if the `<flag>` has zero height.

`\flag_height:n *` `\flag_height:n {<flag name>}`

Expands to the height of the `<flag>` as an integer denotation.

`\flag_raise:n *` `\flag_raise:n {<flag name>}`

The `<flag>`'s height is increased by 1 locally.

Part XIII

The `\l3prg` package

Control structures

Conditional processing in L^AT_EX3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead in the input stream. After processing the input, a *state* is returned. The states returned are `<true>` and `<false>`.

L^AT_EX3 has two forms of conditional flow processing based on these states. The first form is predicate functions that turn the returned state into a boolean `<true>` or `<false>`. For example, the function `\cs_if_free_p:N` checks whether the control sequence given as its argument is free and then returns the boolean `<true>` or `<false>` values to be used in testing with `\if_predicate:w` or in functions to be described below. The second form is the kind of functions choosing a particular argument from the input stream based on the result of the testing as in `\cs_if_free:NTF` which also takes one argument (the `N`) and then executes either `true` or `false` depending on the result.

TEXhackers note: The arguments are executed after exiting the underlying `\if... \fi:` structure.

1 Defining a set of conditional functions

```
\prg_new_conditional:Npnn
\prg_set_conditional:Npnn
\prg_new_conditional:Nnn
\prg_set_conditional:Nnn
```

Updated: 2012-02-06

```
\prg_new_conditional:Npnn \<name>:\<arg spec> \<parameters> {\<conditions>} {\<code>}
\prg_new_conditional:Nnn \<name>:\<arg spec> {\<conditions>} {\<code>}
```

These functions create a family of conditionals using the same `{<code>}` to perform the test created. Those conditionals are expandable if `<code>` is. The `new` versions check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the `set` versions do no check and perform assignments locally (*cf.* `\cs_set:Npn`). The conditionals created are dependent on the comma-separated list of `<conditions>`, which should be one or more of `p`, `T`, `F` and `TF`.

```
\prg_new_protected_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Nnn
\prg_set_protected_conditional:Nnn
```

Updated: 2012-02-06

```
\prg_new_protected_conditional:Npnn \<name>:\<arg spec> \<parameters>
{\<conditions>} {\<code>}
\prg_new_protected_conditional:Nnn \<name>:\<arg spec>
{\<conditions>} {\<code>}
```

These functions create a family of protected conditionals using the same `{<code>}` to perform the test created. The `<code>` does not need to be expandable. The `new` version check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the `set` version do not (*cf.* `\cs_set:Npn`). The conditionals created are depended on the comma-separated list of `<conditions>`, which should be one or more of `T`, `F` and `TF` (not `p`).

The conditionals are defined by `\prg_new_conditional:Npnn` and friends as:

- $\langle name \rangle_p : \langle arg\ spec \rangle$ — a predicate function which will supply either a logical **true** or logical **false**. This function is intended for use in cases where one or more logical tests are combined to lead to a final outcome. This function cannot be defined for **protected** conditionals.
- $\langle name \rangle : \langle arg\ spec \rangle T$ — a function with one more argument than the original $\langle arg\ spec \rangle$ demands. The $\langle true\ branch \rangle$ code in this additional argument will be left on the input stream only if the test is **true**.
- $\langle name \rangle : \langle arg\ spec \rangle F$ — a function with one more argument than the original $\langle arg\ spec \rangle$ demands. The $\langle false\ branch \rangle$ code in this additional argument will be left on the input stream only if the test is **false**.
- $\langle name \rangle : \langle arg\ spec \rangle TF$ — a function with two more arguments than the original $\langle arg\ spec \rangle$ demands. The $\langle true\ branch \rangle$ code in the first additional argument will be left on the input stream if the test is **true**, while the $\langle false\ branch \rangle$ code in the second argument will be left on the input stream if the test is **false**.

The $\langle code \rangle$ of the test may use $\langle parameters \rangle$ as specified by the second argument to $\text{\prg_set_conditional:Npnn}$: this should match the $\langle argument\ specification \rangle$ but this is not enforced. The Nnn versions infer the number of arguments from the argument specification given (*cf.* \cs_new:Nn , *etc.*). Within the $\langle code \rangle$, the functions \prg_return_true: and $\text{\prg_return_false:}$ are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

```
\prg_set_conditional:Npnn \foo_if_bar:NN #1#2 { p , T , TF }
{
  \if_meaning:w \l_tmpa_t1 #1
    \prg_return_true:
  \else:
    \if_meaning:w \l_tmpa_t1 #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  \fi:
}
```

This defines the function \foo_if_bar_p:NN , \foo_if_bar:NNTF and \foo_if_bar:NNT but not \foo_if_bar:NNF (because F is missing from the $\langle conditions \rangle$ list). The return statements take care of resolving the remaining \else: and \fi: before returning the state. There must be a return statement for each branch; failing to do so will result in erroneous output if that branch is executed.

$\text{\prg_new_eq_conditional:NNn}$ $\text{\prg_new_eq_conditional:NNn} \langle name_1 \rangle : \langle arg\ spec_1 \rangle \langle name_2 \rangle : \langle arg\ spec_2 \rangle$
 $\text{\prg_set_eq_conditional:NNn}$ $\{ \langle conditions \rangle \}$

These functions copy a family of conditionals. The **new** version checks for existing definitions (*cf.* \cs_new_eq:NN) whereas the **set** version does not (*cf.* \cs_set_eq:NN). The conditionals copied are depended on the comma-separated list of $\langle conditions \rangle$, which should be one or more of p, T, F and TF.

```
\prg_return_true: *
\prg_return_false: *
```

These “return” functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by `\prg_set_conditional:Npn`, *etc.*, to indicate when a true or false branch should be taken. While they may appear multiple times each within the code of such conditionals, the execution of the conditional must result in the expansion of one of these two functions *exactly once*.

The return functions trigger what is internally an f-expansion process to complete the evaluation of the conditional. Therefore, after `\prg_return_true:` or `\prg_return_false:` there must be no non-expandable material in the input stream for the remainder of the expansion of the conditional code. This includes other instances of either of these functions.

<code>\prg_generate_conditional_variant:Nnn</code>	<code>\prg_generate_conditional_variant:Nnn \⟨name⟩:{arg spec}</code> <code>{variant argument specifiers} {condition specifiers}</code>
--	--

New: 2017-12-12

Defines argument-specifier variants of conditionals. This is equivalent to running `\cs_generate_variant:Nn` `\⟨conditional⟩ {variant argument specifiers}` on each `\⟨conditional⟩` described by the `\⟨condition specifiers⟩`. These base-form `\⟨conditionals⟩` are obtained from the `\⟨name⟩` and `\⟨arg spec⟩` as described for `\prg_new_conditional:Npn`, and they should be defined.

2 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (*e.g.*, draft/final) and other times you perhaps want to use it as a predicate function in an `\if_predicate:w` test. The problem of the primitive `\if_false:` and `\if_true:` tokens is that it is not always safe to pass them around as they may interfere with scanning for termination of primitive conditional processing. Therefore, we employ two canonical booleans: `\c_true_bool` or `\c_false_bool`. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations And, Or, Not, *etc.* which can then be used on both the boolean type and predicate functions.

All conditional `\bool_` functions except assignments are expandable and expect the input to also be fully expandable (which generally means being constructed from predicate functions and booleans, possibly nested).

TEXhackers note: The `bool` data type is not implemented using the `\iffalse/\iftrue` primitives, in contrast to `\newif`, *etc.*, in plain T_EX, L^TE_X 2_ε and so on. Programmers should not base use of `bool` switches on any particular expectation of the implementation.

```
\bool_new:N <boolean>
\bool_new:c
```

Creates a new `\langle boolean\rangle` or raises an error if the name is already taken. The declaration is global. The `\langle boolean\rangle` is initially `false`.

\bool_const:Nn
\bool_const:cn
New: 2017-11-28

\bool_const:Nn <boolean> {<boolexpr>}

Creates a new constant <boolean> or raises an error if the name is already taken. The value of the <boolean> is set globally to the result of evaluating the <boolexpr>.

\bool_set_false:N
\bool_set_false:c
\bool_gset_false:N
\bool_gset_false:c

\bool_set_false:N <boolean>

Sets <boolean> logically **false**.

\bool_set_true:N
\bool_set_true:c
\bool_gset_true:N
\bool_gset_true:c

\bool_set_true:N <boolean>

Sets <boolean> logically **true**.

\bool_set_eq:NN
\bool_set_eq:(cN|Nc|cc)
\bool_gset_eq:NN
\bool_gset_eq:(cN|Nc|cc)

\bool_set_eq:NN <boolean₁> <boolean₂>

Sets <boolean₁> to the current value of <boolean₂>.

\bool_set:Nn
\bool_set:cn
\bool_gset:Nn
\bool_gset:cn
Updated: 2017-07-15

\bool_set:Nn <boolean> {<boolexpr>}

Evaluates the <boolean expression> as described for \bool_if:nTF, and sets the <boolean> variable to the logical truth of this evaluation.

\bool_if_p:N *
\bool_if_p:c *
\bool_if:NTF *
\bool_if:cTF *

Updated: 2017-07-15

\bool_if_p:N <boolean>

\bool_if:NTF <boolean> {<true code>} {<false code>}

Tests the current truth of <boolean>, and continues expansion based on this result.

\bool_show:N
\bool_show:c
New: 2012-02-09
Updated: 2015-08-01

\bool_show:N <boolean>

Displays the logical truth of the <boolean> on the terminal.

\bool_show:n
New: 2012-02-09
Updated: 2017-07-15

\bool_show:n {<boolean expression>}

Displays the logical truth of the <boolean expression> on the terminal.

\bool_log:N
\bool_log:c
New: 2014-08-22
Updated: 2015-08-03

\bool_log:N <boolean>

Writes the logical truth of the <boolean> in the log file.

`\bool_log:n`

New: 2014-08-22

Updated: 2017-07-15

`\bool_log:n {<boolean expression>}`

Writes the logical truth of the *<boolean expression>* in the log file.

`\bool_if_exist_p:N *`
`\bool_if_exist_p:c *`
`\bool_if_exist:NTF *`
`\bool_if_exist:cTF *`

New: 2012-03-03

`\bool_if_exist_p:N <boolean>`
`\bool_if_exist:NTF <boolean> {<true code>} {<false code>}`

Tests whether the *<boolean>* is currently defined. This does not check that the *<boolean>* really is a boolean variable.

`\l_tmpa_bool`
`\l_tmpb_bool`

A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any L^AT_EX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_bool`
`\g_tmpb_bool`

A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any L^AT_EX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean *<true>* or *<false>* values, it seems only fitting that we also provide a parser for *<boolean expressions>*.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean *<true>* or *<false>*. It supports the logical operations And, Or and Not as the well-known infix operators `&&` and `||` and prefix `!` with their usual precedences (namely, `&&` binds more tightly than `||`). In addition to this, parentheses can be used to isolate sub-expressions. For example,

```
\int_compare_p:n { 1 = 1 } &&
(
  \int_compare_p:n { 2 = 3 } ||
  \int_compare_p:n { 4 <= 4 } ||
  \str_if_eq_p:nn { abc } { def }
) &&
! \int_compare_p:n { 2 = 4 }
```

is a valid boolean expression.

Contrarily to some other programming languages, the operators `&&` and `||` evaluate both operands in all cases, even when the first operand is enough to determine the result. This “eager” evaluation should be contrasted with the “lazy” evaluation of `\bool_lazy_...` functions.

TeXhackers note: The eager evaluation of boolean expressions is unfortunately necessary in TeX. Indeed, a lazy parser can get confused if `&&` or `||` or parentheses appear as (unbraced) arguments of some predicates. For instance, the innocuous-looking expression below would break (in a lazy parser) if `#1` were a closing parenthesis and `\l_tmpa_bool` were `true`.

```
( \l_tmpa_bool || \token_if_eq_meaning_p:NN X #1 )
```

Minimal (lazy) evaluation can be obtained using the conditionals `\bool_lazy_all:nTF`, `\bool_lazy_and:nnTF`, `\bool_lazy_any:nTF`, or `\bool_lazy_or:nnTF`, which only evaluate their boolean expression arguments when they are needed to determine the resulting truth value. For example, when evaluating the boolean expression

```
\bool_lazy_and_p:nn
{
  \bool_lazy_any_p:n
  {
    { \int_compare_p:n { 2 = 3 } }
    { \int_compare_p:n { 4 <= 4 } }
    { \int_compare_p:n { 1 = \error } } % skipped
  }
}
{ ! \int_compare_p:n { 2 = 4 } }
```

the line marked with `skipped` is not expanded because the result of `\bool_lazy_any_p:n` is known once the second boolean expression is found to be logically true. On the other hand, the last line is expanded because its logical value is needed to determine the result of `\bool_lazy_and_p:nn`.

`\bool_if_p:n *`
`\bool_if:nTF *`

Updated: 2017-07-15

`\bool_if_p:n {<boolean expression>}`
`\bool_if:nTF {<boolean expression>} {<true code>} {<false code>}`

Tests the current truth of `<boolean expression>`, and continues expansion based on this result. The `<boolean expression>` should consist of a series of predicates or boolean variables with the logical relationship between these defined using `&&` (“And”), `||` (“Or”), `!` (“Not”) and parentheses. The logical Not applies to the next predicate or group.

`\bool_lazy_all_p:n *`
`\bool_lazy_all:nTF *`

New: 2015-11-15
 Updated: 2017-07-15

`\bool_lazy_all_p:n { {<boolexpr_1>} {<boolexpr_2>} ... {<boolexpr_N>} }`
`\bool_lazy_all:nTF { {<boolexpr_1>} {<boolexpr_2>} ... {<boolexpr_N>} } {<true code>} {<false code>}`

Implements the “And” operation on the `<boolean expressions>`, hence is `true` if all of them are `true` and `false` if any of them is `false`. Contrarily to the infix operator `&&`, only the `<boolean expressions>` which are needed to determine the result of `\bool_lazy_all:nTF` are evaluated. See also `\bool_lazy_and:nnTF` when there are only two `<boolean expressions>`.

`\bool_lazy_and_p:nn *`
`\bool_lazy_and:nnTF *`

New: 2015-11-15
 Updated: 2017-07-15

`\bool_lazy_and_p:nn {<boolexpr_1>} {<boolexpr_2>}`
`\bool_lazy_and:nnTF {<boolexpr_1>} {<boolexpr_2>} {<true code>} {<false code>}`

Implements the “And” operation between two boolean expressions, hence is `true` if both are `true`. Contrarily to the infix operator `&&`, the `<boolexpr_2>` is only evaluated if it is needed to determine the result of `\bool_lazy_and:nnTF`. See also `\bool_lazy_all:nTF` when there are more than two `<boolean expressions>`.

```
\bool_lazy_any_p:n *
```

```
\bool_lazy_any:nTF *
```

New: 2015-11-15
Updated: 2017-07-15

```
\bool_lazy_any_p:n { {(boolexpr1} } {(boolexpr2} } ... {(boolexprN} } }  
\bool_lazy_any:nTF { {(boolexpr1} } {(boolexpr2} } ... {(boolexprN} } } { (true code) }  
{ (false code) }
```

Implements the “Or” operation on the *boolean expressions*, hence is **true** if any of them is **true** and **false** if all of them are **false**. Contrarily to the infix operator **||**, only the *boolean expressions* which are needed to determine the result of `\bool_lazy_any:nTF` are evaluated. See also `\bool_lazy_or:nnTF` when there are only two *boolean expressions*.

```
\bool_lazy_or_p:nn *
```

```
\bool_lazy_or:nnTF *
```

New: 2015-11-15
Updated: 2017-07-15

```
\bool_lazy_or_p:nn { {(boolexpr1} } {(boolexpr2} } }  
\bool_lazy_or:nnTF { {(boolexpr1} } {(boolexpr2} } } { (true code) } { (false code) }
```

Implements the “Or” operation between two boolean expressions, hence is **true** if either one is **true**. Contrarily to the infix operator **||**, the *boolexpr₂* is only evaluated if it is needed to determine the result of `\bool_lazy_or:nnTF`. See also `\bool_lazy_any:nTF` when there are more than two *boolean expressions*.

```
\bool_not_p:n *
```

Updated: 2017-07-15

```
\bool_not_p:n { (boolean expression) }
```

Function version of `! (boolean expression)` within a boolean expression.

```
\bool_xor_p:nn *
```

```
\bool_xor:nnTF *
```

New: 2018-05-09

```
\bool_xor_p:nn { {(boolexpr1} } {(boolexpr2} } }  
\bool_xor:nnTF { {(boolexpr1} } {(boolexpr2} } } { (true code) } { (false code) }
```

Implements an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operation.

4 Logical loops

Loops using either boolean expressions or stored boolean values.

```
\bool_do_until:Nn *
```

```
\bool_do_until:cn *
```

Updated: 2017-07-15

```
\bool_do_until:Nn { (boolean) } { (code) }
```

Places the *code* in the input stream for **TEX** to process, and then checks the logical value of the *boolean*. If it is **false** then the *code* is inserted into the input stream again and the process loops until the *boolean* is **true**.

```
\bool_do_while:Nn *
```

```
\bool_do_while:cn *
```

Updated: 2017-07-15

```
\bool_do_while:Nn { (boolean) } { (code) }
```

Places the *code* in the input stream for **TEX** to process, and then checks the logical value of the *boolean*. If it is **true** then the *code* is inserted into the input stream again and the process loops until the *boolean* is **false**.

```
\bool_until_do:Nn *
```

```
\bool_until_do:cn *
```

Updated: 2017-07-15

```
\bool_until_do:Nn { (boolean) } { (code) }
```

This function firsts checks the logical value of the *boolean*. If it is **false** the *code* is placed in the input stream and expanded. After the completion of the *code* the truth of the *boolean* is re-evaluated. The process then loops until the *boolean* is **true**.

```
\bool_while_do:Nn *
```

```
\bool_while_do:cn *
```

Updated: 2017-07-15

```
\bool_while_do:Nn { (boolean) } { (code) }
```

This function firsts checks the logical value of the *boolean*. If it is **true** the *code* is placed in the input stream and expanded. After the completion of the *code* the truth of the *boolean* is re-evaluated. The process then loops until the *boolean* is **false**.

\bool_do_until:nn ☆

Updated: 2017-07-15

\bool_do_until:nn {*boolean expression*} {*code*}

Places the *code* in the input stream for TeX to process, and then checks the logical value of the *boolean expression* as described for \bool_if:nTF. If it is **false** then the *code* is inserted into the input stream again and the process loops until the *boolean expression* evaluates to **true**.

\bool_do_while:nn ☆

Updated: 2017-07-15

\bool_do_while:nn {*boolean expression*} {*code*}

Places the *code* in the input stream for TeX to process, and then checks the logical value of the *boolean expression* as described for \bool_if:nTF. If it is **true** then the *code* is inserted into the input stream again and the process loops until the *boolean expression* evaluates to **false**.

\bool_until_do:nn ☆

Updated: 2017-07-15

\bool_until_do:nn {*boolean expression*} {*code*}

This function firsts checks the logical value of the *boolean expression* (as described for \bool_if:nTF). If it is **false** the *code* is placed in the input stream and expanded. After the completion of the *code* the truth of the *boolean expression* is re-evaluated. The process then loops until the *boolean expression* is **true**.

\bool_while_do:nn ☆

Updated: 2017-07-15

\bool_while_do:nn {*boolean expression*} {*code*}

This function firsts checks the logical value of the *boolean expression* (as described for \bool_if:nTF). If it is **true** the *code* is placed in the input stream and expanded. After the completion of the *code* the truth of the *boolean expression* is re-evaluated. The process then loops until the *boolean expression* is **false**.

5 Producing multiple copies

\prg_replicate:nn *

Updated: 2011-07-04

\prg_replicate:nn {*integer expression*} {*tokens*}

Evaluates the *integer expression* (which should be zero or positive) and creates the resulting number of copies of the *tokens*. The function is both expandable and safe for nesting. It yields its result after two expansion steps.

6 Detecting TeX's mode

\mode_if_horizontal_p: *

\mode_if_horizontal:TF *

\mode_if_horizontal_p:

\mode_if_horizontal:TF {*true code*} {*false code*}

Detects if TeX is currently in horizontal mode.

\mode_if_inner_p: *

\mode_if_inner:TF *

\mode_if_inner_p:

\mode_if_inner:TF {*true code*} {*false code*}

Detects if TeX is currently in inner mode.

\mode_if_math_p: *

\mode_if_math:TF *

\mode_if_math_p:

\mode_if_math:TF {*true code*} {*false code*}

Detects if TeX is currently in maths mode.

Updated: 2011-09-05

```
\mode_if_vertical_p: *
\mode_if_vertical:TF *
```

Detects if T_EX is currently in vertical mode.

7 Primitive conditionals

```
\if_predicate:w *
```

`\if_predicate:w <predicate> <true code> \else: <false code> \fi:`
This function takes a predicate function and branches according to the result. (In practice this function would also accept a single boolean variable in place of the *<predicate>* but to make the coding clearer this should be done through `\if_bool:N`.)

```
\if_bool:N *
```

`\if_bool:N <boolean> <true code> \else: <false code> \fi:`
This function takes a boolean variable and branches according to the result.

8 Nestable recursions and mappings

There are a number of places where recursion or mapping constructs are used in `expl3`. At a low-level, these typically require insertion of tokens at the end of the content to allow “clean up”. To support such mappings in a nestable form, the following functions are provided.

```
\prg_break_point:Nn *
```

New: 2018-03-26

`\prg_break_point:Nn \<type>_map_break: <code>`

Used to mark the end of a recursion or mapping: the functions `\<type>_map_break:` and `\<type>_map_break:n` use this to break out of the loop (see `\prg_map_break:Nn` for how to set these up). After the loop ends, the *<code>* is inserted into the input stream. This occurs even if the break functions are *not* applied: `\prg_break_point:Nn` is functionally-equivalent in these cases to `\use_i:nn`.

```
\prg_map_break:Nn *
```

New: 2018-03-26

`\prg_map_break:Nn \<type>_map_break: <user code>`

`\prg_map_break:Nn \<type>_map_break: <ending code>`

Breaks a recursion in mapping contexts, inserting in the input stream the *<user code>* after the *<ending code>* for the loop. The function breaks loops, inserting their *<ending code>*, until reaching a loop with the same *<type>* as its first argument. This `\<type>_map_break:` argument must be defined; it is simply used as a recognizable marker for the *<type>*.

For types with mappings defined in the kernel, `\<type>_map_break:` and `\<type>_map_break:n` are defined as `\prg_map_break:Nn \<type>_map_break: {}` and the same with {} omitted.

8.1 Simple mappings

In addition to the more complex mappings above, non-nestable mappings are used in a number of locations and support is provided for these.

```
\prg_break_point: *
```

New: 2018-03-27

This copy of \prg_do_nothing: is used to mark the end of a fast short-term recursion: the function \prg_break:n uses this to break out of the loop.

```
\prg_break: *
```

```
\prg_break:n *
```

New: 2018-03-27

```
\prg_break:n {<code>} ... \prg_break_point:
```

Breaks a recursion which has no *<ending code>* and which is not a user-breakable mapping (see for instance \prop_get:Nn), and inserts the *<code>* in the input stream.

9 Internal programming functions

```
\group_align_safe_begin: *
```

```
\group_align_safe_end: *
```

Updated: 2011-08-11

```
\group_align_safe_begin:  
...  
\group_align_safe_end:
```

These functions are used to enclose material in a TeX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the & token inside \halign. This is necessary to allow grabbing of tokens for testing purposes, as TeX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as \peek_after:Nw would result in a forbidden comparison of the internal \endtemplate token, yielding a fatal error. Each \group_align_safe_begin: must be matched by a \group_align_safe_end:, although this does not have to occur within the same function.

Part XIV

The `\l3sys` package: System/runtime functions

1 The name of the job

`\c_sys_jobname_str`

New: 2015-09-19
Updated: 2019-10-27

Constant that gets the “job name” assigned when TeX starts.

TeXhackers note: This copies the contents of the primitive `\jobname`. For technical reasons, the string here is not of the same internal form as other, but may be manipulated using normal string functions.

2 Date and time

`\c_sys_minute_int`
`\c_sys_hour_int`
`\c_sys_day_int`
`\c_sys_month_int`
`\c_sys_year_int`

New: 2015-09-22

The date and time at which the current job was started: these are all reported as integers.

TeXhackers note: Whilst the underlying primitives can be altered by the user, this interface to the time and date is intended to be the “real” values.

3 Engine

`\sys_if_engine_luatex_p: *`
`\sys_if_engine_luatex:TF *`
`\sys_if_engine_pdftex_p: *`
`\sys_if_engine_pdftex:TF *`
`\sys_if_engine_ptex_p: *`
`\sys_if_engine_ptex:TF *`
`\sys_if_engine_uptex_p: *`
`\sys_if_engine_uptex:TF *`
`\sys_if_engine_xetex_p: *`
`\sys_if_engine_xetex:TF *`

New: 2015-09-07

`\sys_if_engine_pdftex:TF {true code} {false code}`

Conditionals which allow engine-specific code to be used. The names follow naturally from those of the engine binaries: note that the (u)ptex tests are for ε -pTeX and ε -upTeX as expl3 requires the ε -TeX extensions. Each conditional is true for *exactly one* supported engine. In particular, `\sys_if_engine_ptex_p:` is true for ε -pTeX but false for ε -upTeX.

`\c_sys_engine_str`

New: 2015-09-19

The current engine given as a lower case string: one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

\c_sys_engine_exec_str

New: 2020-08-20

The name of the standard executable for the current TeX engine given as a lower case string: one of `luatex`, `luahbtex`, `pdftex`, `eptex`, `euptex` or `xetex`.

\c_sys_engine_format_str

New: 2020-08-20

The name of the preloaded format for the current TeX run given as a lower case string: one of `lualatex` (or `dviulatex`), `pdflatex` (or `latex`), `platex`, `uplatex` or `xelatex` for L^AT_EX, similar names for plain TeX (except `pdfTeX` in DVI mode yields `etex`), and `cont-en` for ConTeXt (i.e. the `\fmtname`).

4 Output format

\sys_if_output_dvi_p: ***\sys_if_output_dvi:TF *****\sys_if_output_pdf_p: *****\sys_if_output_pdf:TF ***

New: 2015-09-19

```
\sys_if_output_dvi:TF {<true code>} {<false code>}
```

Conditionals which give the current output mode the TeX run is operating in. This is always one of two outcomes, DVI mode or PDF mode. The two sets of conditionals are thus complementary and are both provided to allow the programmer to emphasise the most appropriate case.

\c_sys_output_str

New: 2015-09-19

The current output mode given as a lower case string: one of `dvi` or `pdf`.

5 Platform

\sys_if_platform_unix_p: ***\sys_if_platform_unix:TF *****\sys_if_platform_windows_p: *****\sys_if_platform_windows:TF ***

New: 2018-07-27

Conditionals which allow platform-specific code to be used. The names follow the Lua `os.type()` function, i.e. all Unix-like systems are `unix` (including Linux and MacOS).

\c_sys_platform_str

New: 2018-07-27

The current platform given as a lower case string: one of `unix`, `windows` or `unknown`.

6 Random numbers

\sys_rand_seed: *

New: 2017-05-27

```
\sys_rand_seed:
```

Expands to the current value of the engine's random seed, a non-negative integer. In engines without random number support this expands to 0.

```
\sys_gset_rand_seed:n
```

New: 2017-05-27

```
\sys_gset_rand_seed:n {<intexpr>}
```

Globally sets the seed for the engine's pseudo-random number generator to the *<integer expression>*. This random seed affects all `\..._rand` functions (such as `\int_rand:nn` or `\clist_rand_item:n`) as well as other packages relying on the engine's random number generator. In engines without random number support this produces an error.

TEXhackers note: While a 32-bit (signed) integer can be given as a seed, only the absolute value is used and any number beyond 2^{28} is divided by an appropriate power of 2. We recommend using an integer in $[0, 2^{28} - 1]$.

7 Access to the shell

```
\sys_get_shell:nnN
```

```
\sys_get_shell:nnNTF
```

New: 2019-09-20

```
\sys_get_shell:nnN {<shell command>} {<setup>} {<tl var>}
\sys_get_shell:nnNTF {<shell command>} {<setup>} {<tl var>} {<true code>} {<false code>}
```

Defines `<tl>` to the text returned by the *<shell command>*. The *<shell command>* is converted to a string using `\tl_to_str:n`. Category codes may need to be set appropriately via the *<setup>* argument, which is run just before running the *<shell command>* (in a group). If shell escape is disabled, the `<tl var>` will be set to `\q_no_value` in the non-branching version. Note that quote characters ("") *cannot* be used inside the *<shell command>*. The `\sys_get_shell:nnNTF` conditional returns `true` if the shell is available and no quote is detected, and `false` otherwise.

```
\c_sys_shell_escape_int
```

New: 2017-05-27

This variable exposes the internal triple of the shell escape status. The possible values are

- 0** Shell escape is disabled
- 1** Unrestricted shell escape is enabled
- 2** Restricted shell escape is enabled

```
\sys_if_shell_p: *
```

```
\sys_if_shell:TF *
```

New: 2017-05-27

```
\sys_if_shell_p:
\sys_if_shell:TF {<true code>} {<false code>}
```

Performs a check for whether shell escape is enabled. This returns true if either of restricted or unrestricted shell escape is enabled.

```
\sys_if_shell_unrestricted_p: *
```

```
\sys_if_shell_unrestricted:TF *
```

New: 2017-05-27

Performs a check for whether *unrestricted* shell escape is enabled.

```
\sys_if_shell_restricted_p: * \sys_if_shell_restricted_p:  
\sys_if_shell_restricted:TF * \sys_if_shell_restricted:TF {{true code}} {{false code}}
```

New: 2017-05-27

Performs a check for whether *restricted* shell escape is enabled. This returns false if unrestricted shell escape is enabled. Unrestricted shell escape is not considered a superset of restricted shell escape in this case. To find whether any shell escape is enabled use `\sys_if_shell:`.

```
\sys_shell_now:n \sys_shell_now:n {{tokens}}  
\sys_shell_now:x
```

Execute *tokens* through shell escape immediately.

New: 2017-05-27

```
\sys_shell_shipout:n \sys_shell_shipout:n {{tokens}}  
\sys_shell_shipout:x
```

Execute *tokens* through shell escape at shipout.

New: 2017-05-27

8 Loading configuration data

```
\sys_load_backend:n \sys_load_backend:n {{backend}}
```

New: 2019-09-12

Loads the additional configuration file needed for backend support. If the *backend* is empty, the standard backend for the engine in use will be loaded. This command may only be used once.

```
\c_sys_backend_str
```

Set to the name of the backend in use by `\sys_load_backend:n` when issued.

```
\sys_load_debug:  
\sys_load_deprecation:
```

New: 2019-09-12

Load the additional configuration files for debugging support and rolling back deprecations, respectively.

8.1 Final settings

```
\sys_finalise: \sys_finalise:  
New: 2019-10-06
```

Finalises all system-dependent functionality: required before loading a backend.

Part XV

The `\clist` package

Comma separated lists

Comma lists contain ordered data where items can be added to the left or right end of the list. This data type allows basic list manipulations such as adding/removing items, applying a function to every item, removing duplicate items, extracting a given item, using the comma list with specified separators, and so on. Sequences (defined in `\seq`) are safer, faster, and provide more features, so they should often be preferred to comma lists. Comma lists are mostly useful when interfacing with L^AT_EX 2_< or other code that expects or provides comma list data.

Several items can be added at once. To ease input of comma lists from data provided by a user outside an `\ExplSyntaxOn ... \ExplSyntaxOff` block, spaces are removed from both sides of each comma-delimited argument upon input. Blank arguments are ignored, to allow for trailing commas or repeated commas (which may otherwise arise when concatenating comma lists “by hand”). In addition, a set of braces is removed if the result of space-trimming is braced: this allows the storage of any item in a comma list. For instance,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~a~ , ~{b}~ , c~\d }
\clist_put_right:Nn \l_my_clist { ~{e~} , , {{f}} , }
```

results in `\l_my_clist` containing `a,b,c~\d,{e~},{{f}}` namely the five items `a`, `b`, `c~\d`, `e~` and `{f}`. Comma lists normally do not contain empty items so the following gives an empty comma list:

```
\clist_clear_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

and it leaves `true` in the input stream. To include an “unsafe” item (empty, or one that contains a comma, or starts or ends with a space, or is a single brace group), surround it with braces.

Almost all operations on comma lists are noticeably slower than those on sequences so converting the data to sequences using `\seq_set_from_clist:Nn` (see `\seq`) may be advisable if speed is important. The exception is that `\clist_if_in:NnTF` and `\clist_remove_duplicates:N` may be faster than their sequence analogues for large lists. However, these functions work slowly for “unsafe” items that must be braced, and may produce errors when their argument contains `{, }` or `#` (assuming the usual T_EX category codes apply). The sequence data type should thus certainly be preferred to comma lists to store such items.

1 Creating and initialising comma lists

```
\clist_new:N <comma list>
\clist_new:c
```

Creates a new `<comma list>` or raises an error if the name is already taken. The declaration is global. The `<comma list>` initially contains no items.

```
\clist_const:Nn
\clist_const:(Nx|cn|cx)

```

New: 2014-07-05

```
\clist_const:Nn <clist var> {{comma list}}
```

Creates a new constant *<clist var>* or raises an error if the name is already taken. The value of the *<clist var>* is set globally to the *<comma list>*.

```
\clist_clear:N
\clist_clear:c
\clist_gclear:N
\clist_gclear:c

```

```
\clist_clear:N <comma list>
```

Clears all items from the *<comma list>*.

```
\clist_clear_new:N
\clist_clear_new:c
\clist_gclear_new:N
\clist_gclear_new:c

```

```
\clist_clear_new:N <comma list>
```

Ensures that the *<comma list>* exists globally by applying `\clist_new:N` if necessary, then applies `\clist_(g)clear:N` to leave the list empty.

```
\clist_set_eq:NN
\clist_set_eq:(cN|Nc|cc)
\clist_gset_eq:NN
\clist_gset_eq:(cN|Nc|cc)

```

```
\clist_set_eq:NN <comma list1> <comma list2>
```

Sets the content of *<comma list₁>* equal to that of *<comma list₂>*.

```
\clist_set_from_seq:NN
\clist_set_from_seq:(cN|Nc|cc)
\clist_gset_from_seq:NN
\clist_gset_from_seq:(cN|Nc|cc)

```

```
\clist_set_from_seq:NN <comma list> <sequence>
```

New: 2014-07-17

Converts the data in the *<sequence>* into a *<comma list>*: the original *<sequence>* is unchanged. Items which contain either spaces or commas are surrounded by braces.

```
\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc

```

```
\clist_concat:NNN <comma list1> <comma list2> <comma list3>
```

Concatenates the content of *<comma list₂>* and *<comma list₃>* together and saves the result in *<comma list₁>*. The items in *<comma list₂>* are placed at the left side of the new comma list.

```
\clist_if_exist_p:N *
\clist_if_exist_p:c *
\clist_if_exist:NTF *
\clist_if_exist:cTF *

```

```
\clist_if_exist_p:N <comma list>
```

```
\clist_if_exist:NTF <comma list> {{true code}} {{false code}}
```

Tests whether the *<comma list>* is currently defined. This does not check that the *<comma list>* really is a comma list.

New: 2012-03-03

2 Adding data to comma lists

```
\clist_set:Nn          \clist_set:Nn <comma list> {\langle item_1\rangle,\dots,\langle item_n\rangle}
\clist_set:(NV|No|Nx|cn|cV|co|cx)
\clist_gset:Nn
\clist_gset:(NV|No|Nx|cn|cV|co|cx)
```

New: 2011-09-06

Sets $\langle \text{comma list} \rangle$ to contain the $\langle \text{items} \rangle$, removing any previous content from the variable. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To store some $\langle \text{tokens} \rangle$ as a single $\langle \text{item} \rangle$ even if the $\langle \text{tokens} \rangle$ contain commas or spaces, add a set of braces: `\clist_set:Nn <comma list> {\{\langle tokens \rangle\}}`.

```
\clist_put_left:Nn      \clist_put_left:Nn <comma list> {\langle item_1\rangle,\dots,\langle item_n\rangle}
\clist_put_left:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_left:Nn
\clist_gput_left:(NV|No|Nx|cn|cV|co|cx)
```

Updated: 2011-09-05

Appends the $\langle \text{items} \rangle$ to the left of the $\langle \text{comma list} \rangle$. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some $\langle \text{tokens} \rangle$ as a single $\langle \text{item} \rangle$ even if the $\langle \text{tokens} \rangle$ contain commas or spaces, add a set of braces: `\clist_put_left:Nn <comma list> {\{\langle tokens \rangle\}}`.

```
\clist_put_right:Nn     \clist_put_right:Nn <comma list> {\langle item_1\rangle,\dots,\langle item_n\rangle}
\clist_put_right:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_right:Nn
\clist_gput_right:(NV|No|Nx|cn|cV|co|cx)
```

Updated: 2011-09-05

Appends the $\langle \text{items} \rangle$ to the right of the $\langle \text{comma list} \rangle$. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some $\langle \text{tokens} \rangle$ as a single $\langle \text{item} \rangle$ even if the $\langle \text{tokens} \rangle$ contain commas or spaces, add a set of braces: `\clist_put_right:Nn <comma list> {\{\langle tokens \rangle\}}`.

3 Modifying comma lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

```
\clist_remove_duplicates:N    \clist_remove_duplicates:N <comma list>
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c
```

Removes duplicate items from the $\langle\text{comma list}\rangle$, leaving the left most copy of each item in the $\langle\text{comma list}\rangle$. The $\langle\text{item}\rangle$ comparison takes place on a token basis, as for `\t1_if_eq:nn(TF)`.

TEXhackers note: This function iterates through every item in the $\langle comma\ list \rangle$ and does a comparison with the $\langle items \rangle$ already checked. It is therefore relatively slow with large comma lists. Furthermore, it may fail if any of the items in the $\langle comma\ list \rangle$ contains $\{$, $\}$, or $\#$ (assuming the usual TeX category codes apply).

\clist_remove_all:Nn	\clist_remove_all:Nn <i>(comma list)</i> { <i>item</i> }
\clist_remove_all:cn	
\clist_gremove_all:Nn	Removes every occurrence of <i>item</i> from the
\clist_gremove_all:cn	place on a token basis, as for \tl_if_eq:nn

Updated: 2011-09-06

`\clist_reverse:N comma list`
Reverses the order of items stored in the *<comma list>*.

New: 2014-07-18

\clist_reverse:n {<comma list>}

New: 2014-07-18

Leaves the items in the $\langle comma\;list \rangle$ in the input stream in reverse order. Contrarily to other what is done for other n-type $\langle comma\;list \rangle$ arguments, braces and spaces are preserved by this process.

TeXhackers note: The result is returned within `\unexpanded`, which means that the comma list does not expand further when appearing in an `x`-type or `e`-type argument expansion.

```
\clist_sort:Nn  
\clist_sort:cn  
\clist_gsort:Nn  
\clist_gsort:cn
```

New: 2017-02-06

\clist_sort:Nn *clist var* {*comparison code*}

Sorts the items in the `<clist var>` according to the `<comparison code>`, and assigns the result to `<clist var>`. The details of sorting comparison are described in Section 1.

4 Comma list conditionals

\clist_if_empty_p:N *	\clist_if_empty_p:N <i>(comma list)</i> \clist_if_empty:N _{TF} <i>(comma list)</i> {\{true code\}} {\{false code\}}
	Tests if the <i>(comma list)</i> is empty (containing no items).
\clist_if_empty_p:n *	\clist_if_empty_p:n {\{comma list\}} \clist_if_empty:n _{TF} {\{comma list\}} {\{true code\}} {\{false code\}}

New: 2014-07-05

Tests if the *(comma list)* is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list {~,~,~,~} (without outer braces) is empty, while {~,{},{} (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

\clist_if_in:Nn _{TF}	\clist_if_in:Nn _{TF} <i>(comma list)</i> {\{item\}} {\{true code\}} {\{false code\}}
\clist_if_in:(NV No cn cV co) _{TF}	
\clist_if_in:nn _{TF}	
\clist_if_in:(nV no) _{TF}	

Updated: 2011-09-06

Tests if the *(item)* is present in the *(comma list)*. In the case of an n-type *(comma list)*, the usual rules of space trimming and brace stripping apply. Hence,

\clist_if_in:nn_{TF} { a , {b}~ , {b} , c } { b } {true} {false}

yields true.

TeXhackers note: The function may fail if the *(item)* contains {, }, or # (assuming the usual TeX category codes apply).

5 Mapping to comma lists

The functions described in this section apply a specified function to each item of a comma list. All mappings are done at the current group level, i.e. any local assignments made by the *(function)* or *(code)* discussed below remain in effect after the loop.

When the comma list is given explicitly, as an n-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if the comma list that is being mapped is {a, {b}, , {}, {c}, } then the arguments passed to the mapped function are ‘a’, ‘{b}’, an empty argument, and ‘c’.

When the comma list is given as an N-type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using n-type comma lists.

\clist_map_function:NN	\clist_map_function:NN <i>(comma list)</i> <i>(function)</i>
\clist_map_function:cN	Applies <i>(function)</i> to every <i>(item)</i> stored in the <i>(comma list)</i> . The <i>(function)</i> receives one argument for each iteration. The <i>(items)</i> are returned from left to right. The function \clist_map_inline:Nn is in general more efficient than \clist_map_function:NN.
Updated: 2012-06-29	

```
\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn
```

Updated: 2012-06-29

```
\clist_map_variable:NNn
\clist_map_variable:cNn
\clist_map_variable:nNn
```

Updated: 2012-06-29

```
\clist_map_break: ☆
```

Updated: 2012-06-29

```
\clist_map_inline:Nn {comma list} {inline function}
```

Applies *inline function* to every *item* stored within the *comma list*. The *inline function* should consist of code which receives the *item* as #1. The *items* are returned from left to right.

```
\clist_map_variable:NNn {comma list} {variable} {code}
```

Stores each *item* of the *comma list* in turn in the (token list) *variable* and applies the *code*. The *code* will usually make use of the *variable*, but this is not enforced. The assignments to the *variable* are local. Its value after the loop is the last *item* in the *comma list*, or its original value if there were no *item*. The *items* are returned from left to right.

```
\clist_map_break:
```

Used to terminate a \clist_map_... function before all entries in the *comma list* have been processed. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF {#1} {bingo}
    { \clist_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a \clist_map_... scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\clist_map_break:n ☆

Updated: 2012-06-29

\clist_map_break:n {<code>}

Used to terminate a \clist_map_... function before all entries in the *comma list* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
    { \clist_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a \clist_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

\clist_count:N ★\clist_count:c ★\clist_count:n ★

New: 2012-07-13

\clist_count:N <comma list>

Leaves the number of items in the *<comma list>* in the input stream as an *integer denotation*. The total number of items in a *<comma list>* includes those which are duplicates, *i.e.* every item in a *<comma list>* is counted.

6 Using the content of comma lists directly

\clist_use:Nnnn ★

\clist_use:cnnn ★

New: 2013-05-26

\clist_use:Nnnn <clist var> {<separator between two>}
{<separator between more than two>} {<separator between final two>}

Places the contents of the *<clist var>* in the input stream, with the appropriate *<separator>* between the items. Namely, if the comma list has more than two items, the *<separator between more than two>* is placed between each pair of items except the last, for which the *<separator between final two>* is used. If the comma list has exactly two items, then they are placed in the input stream separated by the *<separator between two>*. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

TeXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the *<items>* do not expand further when appearing in an x-type argument expansion.

```
\clist_use:Nn *
\clist_use:cn *
```

New: 2013-05-26

```
\clist_use:Nn <clist var> {<separator>}
```

Places the contents of the *<clist var>* in the input stream, with the *<separator>* between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<items>* do not expand further when appearing in an x-type argument expansion.

7 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

```
\clist_get:NN
\clist_get:cN
\clist_get:NNTF
\clist_get:cNTF
```

New: 2012-05-14
Updated: 2019-02-16

```
\clist_get:NN <comma list> <token list variable>
```

Stores the left-most item from a *<comma list>* in the *<token list variable>* without removing it from the *<comma list>*. The *<token list variable>* is assigned locally. In the non-branching version, if the *<comma list>* is empty the *<token list variable>* is set to the marker value `\q_no_value`.

```
\clist_pop:NN
\clist_pop:cN
```

Updated: 2011-09-06

```
\clist_pop:NN <comma list> <token list variable>
```

Pops the left-most item from a *<comma list>* into the *<token list variable>*, i.e. removes the item from the comma list and stores it in the *<token list variable>*. Both of the variables are assigned locally.

```
\clist_gpop:NN
\clist_gpop:cN
```

```
\clist_gpop:NN <comma list> <token list variable>
```

Pops the left-most item from a *<comma list>* into the *<token list variable>*, i.e. removes the item from the comma list and stores it in the *<token list variable>*. The *<comma list>* is modified globally, while the assignment of the *<token list variable>* is local.

```
\clist_pop:NNTF
\clist_pop:cNTF
```

New: 2012-05-14

```
\clist_pop:NNTF <comma list> <token list variable> {<true code>} {<false code>}
```

If the *<comma list>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<comma list>* is non-empty, pops the top item from the *<comma list>* in the *<token list variable>*, i.e. removes the item from the *<comma list>*. Both the *<comma list>* and the *<token list variable>* are assigned locally.

```
\clist_gpop:NNTF
\clist_gpop:cNTF
```

New: 2012-05-14

`\clist_gpop:NNTF <comma list> <token list variable> {<true code>} {<false code>}`

If the $\langle \text{comma list} \rangle$ is empty, leaves the $\langle \text{false code} \rangle$ in the input stream. The value of the $\langle \text{token list variable} \rangle$ is not defined in this case and should not be relied upon. If the $\langle \text{comma list} \rangle$ is non-empty, pops the top item from the $\langle \text{comma list} \rangle$ in the $\langle \text{token list variable} \rangle$, i.e. removes the item from the $\langle \text{comma list} \rangle$. The $\langle \text{comma list} \rangle$ is modified globally, while the $\langle \text{token list variable} \rangle$ is assigned locally.

```
\clist_push:Nn
\clist_push:(NV|No|Nx|cn|cV|co|cx)
\clist_gpush:Nn
\clist_gpush:(NV|No|Nx|cn|cV|co|cx)
```

Adds the $\{ \langle \text{items} \rangle \}$ to the top of the $\langle \text{comma list} \rangle$. Spaces are removed from both sides of each item as for any n-type comma list.

8 Using a single item

```
\clist_item:Nn *
\clist_item:cn *
\clist_item:nn *
```

New: 2014-07-17

`\clist_item:Nn <comma list> {<integer expression>}`

Indexing items in the $\langle \text{comma list} \rangle$ from 1 at the top (left), this function evaluates the $\langle \text{integer expression} \rangle$ and leaves the appropriate item from the comma list in the input stream. If the $\langle \text{integer expression} \rangle$ is negative, indexing occurs from the bottom (right) of the comma list. When the $\langle \text{integer expression} \rangle$ is larger than the number of items in the $\langle \text{comma list} \rangle$ (as calculated by `\clist_count:N`) then the function expands to nothing.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle \text{item} \rangle$ does not expand further when appearing in an x-type argument expansion.

```
\clist_rand_item:N *
\clist_rand_item:c *
\clist_rand_item:n *
```

New: 2016-12-06

`\clist_rand_item:N <clist var>
\clist_rand_item:n {<comma list>}`

Selects a pseudo-random item of the $\langle \text{comma list} \rangle$. If the $\langle \text{comma list} \rangle$ has no item, the result is empty.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle \text{item} \rangle$ does not expand further when appearing in an x-type argument expansion.

9 Viewing comma lists

```
\clist_show:N
\clist_show:c
```

Updated: 2015-08-03

`\clist_show:N <comma list>`

Displays the entries in the $\langle \text{comma list} \rangle$ in the terminal.

<code>\clist_show:n</code>	<code>\clist_show:n {\langle tokens \rangle}</code>
<small>Updated: 2013-08-03</small>	Displays the entries in the comma list in the terminal.
<code>\clist_log:N</code>	<code>\clist_log:N {comma list}</code>
<code>\clist_log:c</code>	Writes the entries in the <i>comma list</i> in the log file. See also <code>\clist_show:N</code> which displays the result in the terminal.
<small>New: 2014-08-22</small>	
<small>Updated: 2015-08-03</small>	
<code>\clist_log:n</code>	<code>\clist_log:n {\langle tokens \rangle}</code>
<small>New: 2014-08-22</small>	Writes the entries in the comma list in the log file. See also <code>\clist_show:n</code> which displays the result in the terminal.

10 Constant and scratch comma lists

<code>\c_empty_clist</code>	Constant that is always empty.
<small>New: 2012-07-02</small>	
<code>\l_tmpa_clist</code>	Scratch comma lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_clist</code>	
<small>New: 2011-09-06</small>	
<code>\g_tmpa_clist</code>	Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_clist</code>	
<small>New: 2011-09-06</small>	

Part XVI

The **I3token** package

Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in **T_EX**, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such has two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most functions we describe here can be used on control sequences, as those are tokens as well.

It is important to distinguish two aspects of a token: its “shape” (for lack of a better word), which affects the matching of delimited arguments and the comparison of token lists containing this token, and its “meaning”, which affects whether the token expands or what operation it performs. One can have tokens of different shapes with the same meaning, but not the converse.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three names for the same internal operation of **T_EX**, namely the primitive testing the next two characters for equality of their character code. They have the same meaning hence behave identically in many situations. However, **T_EX** distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below takes everything until `\if:w` as an argument, despite the presence of other copies of `\if:w` under different names.

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

A list of all possible shapes and a list of all possible meanings are given in section 7.

1 Creating character tokens

```
\char_set_active_eq:NN
\char_set_active_eq:Nc
\char_gset_active_eq:NN
\char_gset_active_eq:Nc
```

Updated: 2015-11-12

`\char_set_active_eq:NN <char> <function>`

Sets the behaviour of the `<char>` in situations where it is active (category code 13) to be equivalent to that of the `<function>`. The category code of the `<char>` is *unchanged* by this process. The `<function>` may itself be an active character.

```
\char_set_active_eq:nN
\char_set_active_eq:nc
\char_gset_active_eq:nN
\char_gset_active_eq:nc
```

New: 2015-11-12

`\char_set_active_eq:nN {<integer expression>} <function>`

Sets the behaviour of the `<char>` which has character code as given by the `<integer expression>` in situations where it is active (category code 13) to be equivalent to that of the `<function>`. The category code of the `<char>` is *unchanged* by this process. The `<function>` may itself be an active character.

\char_generate:nn *

New: 2015-09-09
Updated: 2019-01-16

\char_generate:nn {*charcode*} {*catcode*}

Generates a character token of the given *charcode* and *catcode* (both of which may be integer expressions). The *catcode* may be one of

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 11 (letter)
- 12 (other)
- 13 (active)

and other values raise an error. The *charcode* may be any one valid for the engine in use. Active characters cannot be generated in older versions of X_ET_EX.

TeXhackers note: Exactly two expansions are needed to produce the character.

\char_lowercase:N *
\char_uppercase:N *
\char_titlecase:N *
\char_foldcase:N *
\char_str_lowercase:N *
\char_str_uppercase:N *
\char_str_titlecase:N *
\char_str_foldcase:N *

New: 2020-01-09

\char_lowercase:N {*char*}

Converts the *char* to the equivalent case-changed character as detailed by the function name (see **\str_foldcase:n** and **\text_titlecase:n** for details of these terms). The case mapping is carried out with no context-dependence (*cf.* **\text_uppercase:n**, *etc.*) The **str** versions always generate “other” (category code 12) characters, whilst the standard versions generate characters with the category code of the *char* (i.e. only the character code changes).

\c_catcode_other_space_tl

New: 2011-09-05

Token list containing one character with category code 12, (“other”), and character code 32 (space).

2 Manipulating and interrogating character tokens

```
\char_set_catcode_escape:N          \char_set_catcode_letter:N <character>
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N
```

Updated: 2015-11-11

Sets the category code of the *<character>* to that indicated in the function name. Depending on the current category code of the *<token>* the escape token may also be needed:

```
\char_set_catcode_other:N \%
```

The assignment is local.

```
\char_set_catcode_escape:n          \char_set_catcode_letter:n {<integer expression>}
\char_set_catcode_group_begin:n
\char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
\char_set_catcode_alignment:n
\char_set_catcode_end_line:n
\char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
\char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n
```

Updated: 2015-11-11

Sets the category code of the *<character>* which has character code as given by the *<integer expression>*. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local.

\char_set_catcode:nn

Updated: 2015-11-11

```
\char_set_catcode:nn {\integer expression} {\integer expression}
```

These functions set the category code of the *character* which has character code as given by the *integer expression*. The first *integer expression* is the character code and the second is the category code to apply. The setting applies within the current TeX group. In general, the symbolic functions `\char_set_catcode_<type>` should be preferred, but there are cases where these lower-level functions may be useful.

\char_value_catcode:n *

```
\char_value_catcode:n {\integer expression}
```

Expands to the current category code of the *character* with character code given by the *integer expression*.

\char_show_value_catcode:n

```
\char_show_value_catcode:n {\integer expression}
```

Displays the current category code of the *character* with character code given by the *integer expression* on the terminal.

\char_set_lccode:nn

Updated: 2015-08-06

```
\char_set_lccode:nn {\integer expression} {\integer expression}
```

Sets up the behaviour of the *character* when found inside `\text_lowercase:n`, such that *character*₁ will be converted into *character*₂. The two *characters* may be specified using an *integer expression* for the character code concerned. This may include the TeX ‘*character*’ method for converting a single character into its character code:

```
\char_set_lccode:nn {\`A} {\`a} % Standard behaviour  
\char_set_lccode:nn {\`A} {\`A + 32}  
\char_set_lccode:nn {50} {60}
```

The setting applies within the current TeX group.

\char_value_lccode:n *

```
\char_value_lccode:n {\integer expression}
```

Expands to the current lower case code of the *character* with character code given by the *integer expression*.

\char_show_value_lccode:n

```
\char_show_value_lccode:n {\integer expression}
```

Displays the current lower case code of the *character* with character code given by the *integer expression* on the terminal.

\char_set_uccode:nn

Updated: 2015-08-06

```
\char_set_uccode:nn {\integer expression} {\integer expression}
```

Sets up the behaviour of the *character* when found inside `\text_uppercase:n`, such that *character*₁ will be converted into *character*₂. The two *characters* may be specified using an *integer expression* for the character code concerned. This may include the TeX ‘*character*’ method for converting a single character into its character code:

```
\char_set_uccode:nn {\`a} {\`A} % Standard behaviour  
\char_set_uccode:nn {\`A} {\`A - 32}  
\char_set_uccode:nn {60} {50}
```

The setting applies within the current TeX group.

\char_value_uccode:n *

`\char_value_uccode:n {<integer expression>}`

Expands to the current upper case code of the *character* with character code given by the *integer expression*.

\char_show_value_uccode:n

`\char_show_value_uccode:n {<integer expression>}`

Displays the current upper case code of the *character* with character code given by the *integer expression* on the terminal.

\char_set_mathcode:nn

Updated: 2015-08-06

`\char_set_mathcode:nn {<intexpr1>} {<intexpr2>}`

This function sets up the math code of *character*. The *character* is specified as an *integer expression* which will be used as the character code of the relevant character. The setting applies within the current TeX group.

\char_value_mathcode:n *

`\char_value_mathcode:n {<integer expression>}`

Expands to the current math code of the *character* with character code given by the *integer expression*.

\char_show_value_mathcode:n

`\char_show_value_mathcode:n {<integer expression>}`

Displays the current math code of the *character* with character code given by the *integer expression* on the terminal.

\char_set_sfcode:nn

Updated: 2015-08-06

`\char_set_sfcode:nn {<intexpr1>} {<intexpr2>}`

This function sets up the space factor for the *character*. The *character* is specified as an *integer expression* which will be used as the character code of the relevant character. The setting applies within the current TeX group.

\char_value_sfcode:n *

`\char_value_sfcode:n {<integer expression>}`

Expands to the current space factor for the *character* with character code given by the *integer expression*.

\char_show_value_sfcode:n

`\char_show_value_sfcode:n {<integer expression>}`

Displays the current space factor for the *character* with character code given by the *integer expression* on the terminal.

\l_char_active_seq

New: 2012-01-23

Updated: 2015-11-11

Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category *active* (catcode 13). Each entry in the sequence consists of a single escaped token, for example `\~`. Active tokens should be added to the sequence when they are defined for general document use.

\l_char_special_seq

New: 2012-01-23

Updated: 2015-11-11

Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories *letter* (catcode 11) or *other* (catcode 12). Each entry in the sequence consists of a single escaped token, for example `\\"` for the backslash or `\{` for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.

3 Generic tokens

```
\c_group_begin_token
\c_group_end_token
\c_math_toggle_token
\c_alignment_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token
```

These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.

```
\c_catcode_letter_token
\c_catcode_other_token
```

These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.

```
\c_catcode_active_tl
```

A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.

4 Converting tokens

```
\token_to_meaning:N *
\token_to_meaning:c *
```

Inserts the current meaning of the *<token>* into the input stream as a series of characters of category code 12 (other). This is the primitive TeX description of the *<token>*, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` are described as **macros**.

TeXhackers note: This is the TeX primitive `\meaning`. The *<token>* can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal TeX category codes apply) even though these are not valid N-type arguments.

```
\token_to_str:N *
\token_to_str:c *
```

Converts the given *<token>* into a series of characters with category code 12 (other). If the *<token>* is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the *<token>*). This function requires only a single expansion.

TeXhackers note: `\token_to_str:N` is the TeX primitive `\string` renamed. The *<token>* can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal TeX category codes apply) even though these are not valid N-type arguments.

5 Token conditionals

```
\token_if_group_begin_p:N * \token_if_group_begin_p:N <token>
\token_if_group_begin:NTF * \token_if_group_begin:NTF <token> {{true code}} {{false code}}
```

Tests if *<token>* has the category code of a begin group token ({ when normal TeX category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid N-type argument.

```
\token_if_group_end_p:N * \token_if_group_end_p:N <token>
\token_if_group_end:NTF * \token_if_group_end:NTF <token> {{true code}} {{false code}}
```

Tests if *<token>* has the category code of an end group token (}) when normal TeX category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.

```
\token_if_math_toggle_p:N * \token_if_math_toggle_p:N <token>
\token_if_math_toggle:NTF * \token_if_math_toggle:NTF <token> {{true code}} {{false code}}
```

Tests if *<token>* has the category code of a math shift token (\$) when normal TeX category codes are in force).

```
\token_if_alignment_p:N * \token_if_alignment_p:N <token>
\token_if_alignment:NTF * \token_if_alignment:NTF <token> {{true code}} {{false code}}
```

Tests if *<token>* has the category code of an alignment token (& when normal TeX category codes are in force).

```
\token_if_parameter_p:N * \token_if_parameter_p:N <token>
\token_if_parameter:NTF * \token_if_parameter:NTF <token> {{true code}} {{false code}}
```

Tests if *<token>* has the category code of a macro parameter token (# when normal TeX category codes are in force).

```
\token_if_math_superscript_p:N * \token_if_math_superscript_p:N <token>
\token_if_math_superscript:NTF * \token_if_math_superscript:NTF <token> {{true code}} {{false code}}
```

Tests if *<token>* has the category code of a superscript token (^ when normal TeX category codes are in force).

```
\token_if_math_subscript_p:N * \token_if_math_subscript_p:N <token>
\token_if_math_subscript:NTF * \token_if_math_subscript:NTF <token> {{true code}} {{false code}}
```

Tests if *<token>* has the category code of a subscript token (_) when normal TeX category codes are in force).

```
\token_if_space_p:N * \token_if_space_p:N <token>
\token_if_space:NTF * \token_if_space:NTF <token> {{true code}} {{false code}}
```

Tests if *<token>* has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_letter_p:N *</code>	<code>\token_if_letter_p:N <token></code>
<code>\token_if_letter:NTF *</code>	<code>\token_if_letter:NTF <token> {\{true code\}} {\{false code\}}</code>
Tests if <code><token></code> has the category code of a letter token.	
<code>\token_if_other_p:N *</code>	<code>\token_if_other_p:N <token></code>
<code>\token_if_other:NTF *</code>	<code>\token_if_other:NTF <token> {\{true code\}} {\{false code\}}</code>
Tests if <code><token></code> has the category code of an “other” token.	
<code>\token_if_active_p:N *</code>	<code>\token_if_active_p:N <token></code>
<code>\token_if_active:NTF *</code>	<code>\token_if_active:NTF <token> {\{true code\}} {\{false code\}}</code>
Tests if <code><token></code> has the category code of an active character.	
<code>\token_if_eq_catcode_p>NN *</code>	<code>\token_if_eq_catcode_p>NN <token₁> <token₂></code>
<code>\token_if_eq_catcode:NNTF *</code>	<code>\token_if_eq_catcode:NNTF <token₁> <token₂> {\{true code\}} {\{false code\}}</code>
Tests if the two <code><tokens></code> have the same category code.	
<code>\token_if_eq_charcode_p>NN *</code>	<code>\token_if_eq_charcode_p>NN <token₁> <token₂></code>
<code>\token_if_eq_charcode:NNTF *</code>	<code>\token_if_eq_charcode:NNTF <token₁> <token₂> {\{true code\}} {\{false code\}}</code>
Tests if the two <code><tokens></code> have the same character code.	
<code>\token_if_eq_meaning_p>NN *</code>	<code>\token_if_eq_meaning_p>NN <token₁> <token₂></code>
<code>\token_if_eq_meaning:NNTF *</code>	<code>\token_if_eq_meaning:NNTF <token₁> <token₂> {\{true code\}} {\{false code\}}</code>
Tests if the two <code><tokens></code> have the same meaning when expanded.	
<code>\token_if_macro_p:N *</code>	<code>\token_if_macro_p:N <token></code>
<code>\token_if_macro:NTF *</code>	<code>\token_if_macro:NTF <token> {\{true code\}} {\{false code\}}</code>
Tests if the <code><token></code> is a TeX macro. Updated: 2011-05-23	
<code>\token_if_cs_p:N *</code>	<code>\token_if_cs_p:N <token></code>
<code>\token_if_cs:NTF *</code>	<code>\token_if_cs:NTF <token> {\{true code\}} {\{false code\}}</code>
Tests if the <code><token></code> is a control sequence.	
<code>\token_if_expandable_p:N *</code>	<code>\token_if_expandable_p:N <token></code>
<code>\token_if_expandable:NTF *</code>	<code>\token_if_expandable:NTF <token> {\{true code\}} {\{false code\}}</code>
Tests if the <code><token></code> is expandable. This test returns <code>false</code> for an undefined token.	
<code>\token_if_long_macro_p:N *</code>	<code>\token_if_long_macro_p:N <token></code>
<code>\token_if_long_macro:NTF *</code>	<code>\token_if_long_macro:NTF <token> {\{true code\}} {\{false code\}}</code>
Tests if the <code><token></code> is a long macro. Updated: 2012-01-20	
<code>\token_if_protected_macro_p:N *</code>	<code>\token_if_protected_macro_p:N <token></code>
<code>\token_if_protected_macro:NTF *</code>	<code>\token_if_protected_macro:NTF <token> {\{true code\}} {\{false code\}}</code>
Tests if the <code><token></code> is a protected macro: for a macro which is both protected and long this returns <code>false</code> . Updated: 2012-01-20	

```
\token_if_protected_long_macro_p:N * \token_if_protected_long_macro_p:N <token>
\token_if_protected_long_macro:NTF * \token_if_protected_long_macro:NTF <token> {\{true code\}} {\{false code\}}
```

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is a protected long macro.

```
\token_if_chardef_p:N * \token_if_chardef_p:N <token>
\token_if_chardef:NTF * \token_if_chardef:NTF <token> {\{true code\}} {\{false code\}}
```

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a chardef.

TEXhackers note: Booleans, boxes and small integer constants are implemented as \backslash chardefs.

```
\token_if_mathchardef_p:N * \token_if_mathchardef_p:N <token>
\token_if_mathchardef:NTF * \token_if_mathchardef:NTF <token> {\{true code\}} {\{false code\}}
```

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a mathchardef.

```
\token_if_font_selection_p:N * \token_if_font_selection_p:N <token>
\token_if_font_selection:NTF * \token_if_font_selection:NTF <token> {\{true code\}} {\{false code\}}
```

New: 2020-10-27

Tests if the $\langle token \rangle$ is defined to be a font selection command.

```
\token_if_dim_register_p:N * \token_if_dim_register_p:N <token>
\token_if_dim_register:NTF * \token_if_dim_register:NTF <token> {\{true code\}} {\{false code\}}
```

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a dimension register.

```
\token_if_int_register_p:N * \token_if_int_register_p:N <token>
\token_if_int_register:NTF * \token_if_int_register:NTF <token> {\{true code\}} {\{false code\}}
```

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a integer register.

TEXhackers note: Constant integers may be implemented as integer registers, \backslash chardefs, or \backslash mathchardefs depending on their value.

```
\token_if_muskip_register_p:N * \token_if_muskip_register_p:N <token>
\token_if_muskip_register:NTF * \token_if_muskip_register:NTF <token> {\{true code\}} {\{false code\}}
```

New: 2012-02-15

Tests if the $\langle token \rangle$ is defined to be a muskip register.

```
\token_if_skip_register_p:N *
\token_if_skip_register:NTF *
```

Updated: 2012-01-20

Tests if the *<token>* is defined to be a skip register.

```
\token_if_toks_register_p:N *
\token_if_toks_register:NTF *
```

Updated: 2012-01-20

Tests if the *<token>* is defined to be a toks register (not used by L^AT_EX3).

```
\token_if_primitive_p:N *
\token_if_primitive:NTF *
```

Updated: 2020-09-11

Tests if the *<token>* is an engine primitive. In L^AT_EX this includes primitive-like commands defined using `{token.set_lua}`.

6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the “peek” functions. The generic `\peek_after:Nw` is provided along with a family of predefined tests for common cases. As peeking ahead does *not* skip spaces the predefined tests include both a space-respecting and space-skipping version.

```
\peek_after:Nw
```

Locally sets the test variable `\l_peek_token` equal to *<token>* (as an implicit token, *not* as a token list), and then expands the *<function>*. The *<token>* remains in the input stream as the next item after the *<function>*. The *<token>* here may be `\`, `{` or `}` (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

```
\peek_gafter:Nw
```

Globally sets the test variable `\g_peek_token` equal to *<token>* (as an implicit token, *not* as a token list), and then expands the *<function>*. The *<token>* remains in the input stream as the next item after the *<function>*. The *<token>* here may be `\`, `{` or `}` (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

```
\l_peek_token
```

Token set by `\peek_after:Nw` and available for testing as described above.

```
\g_peek_token
```

Token set by `\peek_gafter:Nw` and available for testing as described above.

\peek_catcode:NTF

Updated: 2012-12-20

\peek_catcode:NTF *<test token>* {\<true code>} {\<false code>}

Tests if the next *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test \token_if_eq_catcode:NNTF). Spaces are respected by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

\peek_catcode_ignore_spaces:NTF

Updated: 2012-12-20

\peek_catcode_ignore_spaces:NTF *<test token>* {\<true code>} {\<false code>}

Tests if the next non-space *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test \token_if_eq_catcode:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

\peek_catcode_remove:NTF

Updated: 2012-12-20

\peek_catcode_remove:NTF *<test token>* {\<true code>} {\<false code>}

Tests if the next *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test \token_if_eq_catcode:NNTF). Spaces are respected by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

\peek_catcode_remove_ignore_spaces:NTF

Updated: 2012-12-20

\peek_catcode_remove_ignore_spaces:NTF *<test token>* {\<true code>} {\<false code>}

Tests if the next non-space *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test \token_if_eq_catcode:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

\peek_charcode:NTF

Updated: 2012-12-20

\peek_charcode:NTF *<test token>* {\<true code>} {\<false code>}

Tests if the next *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test \token_if_eq_charcode:NNTF). Spaces are respected by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

\peek_charcode_ignore_spaces:NTF

Updated: 2012-12-20

\peek_charcode_ignore_spaces:NTF *<test token>* {\<true code>} {\<false code>}

Tests if the next non-space *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test \token_if_eq_charcode:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<code>\peek_charcode_remove:NTF</code>	<code>\peek_charcode_remove:NTF <test token> {(true code)} {(false code)}</code>
Updated: 2012-12-20	Tests if the next <i><token></i> in the input stream has the same character code as the <i><test token></i> (as defined by the test <code>\token_if_eq_charcode:NNTF</code>). Spaces are respected by the test and the <i><token></i> is removed from the input stream if the test is true. The function then places either the <i><true code></i> or <i><false code></i> in the input stream (as appropriate to the result of the test).
<code>\peek_charcode_remove_ignore_spaces:NTF</code>	<code>\peek_charcode_remove_ignore_spaces:NTF <test token> {(true code)} {(false code)}</code>
Updated: 2012-12-20	Tests if the next non-space <i><token></i> in the input stream has the same character code as the <i><test token></i> (as defined by the test <code>\token_if_eq_charcode:NNTF</code>). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the <i><token></i> is removed from the input stream if the test is true. The function then places either the <i><true code></i> or <i><false code></i> in the input stream (as appropriate to the result of the test).
<code>\peek_meaning:NTF</code>	<code>\peek_meaning:NTF <test token> {(true code)} {(false code)}</code>
Updated: 2011-07-02	Tests if the next <i><token></i> in the input stream has the same meaning as the <i><test token></i> (as defined by the test <code>\token_if_eq_meaning:NNTF</code>). Spaces are respected by the test and the <i><token></i> is left in the input stream after the <i><true code></i> or <i><false code></i> (as appropriate to the result of the test).
<code>\peek_meaning_ignore_spaces:NTF</code>	<code>\peek_meaning_ignore_spaces:NTF <test token> {(true code)} {(false code)}</code>
Updated: 2012-12-05	Tests if the next non-space <i><token></i> in the input stream has the same meaning as the <i><test token></i> (as defined by the test <code>\token_if_eq_meaning:NNTF</code>). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the <i><token></i> is left in the input stream after the <i><true code></i> or <i><false code></i> (as appropriate to the result of the test).
<code>\peek_meaning_remove:NTF</code>	<code>\peek_meaning_remove:NTF <test token> {(true code)} {(false code)}</code>
Updated: 2011-07-02	Tests if the next <i><token></i> in the input stream has the same meaning as the <i><test token></i> (as defined by the test <code>\token_if_eq_meaning:NNTF</code>). Spaces are respected by the test and the <i><token></i> is removed from the input stream if the test is true. The function then places either the <i><true code></i> or <i><false code></i> in the input stream (as appropriate to the result of the test).
<code>\peek_meaning_remove_ignore_spaces:NTF</code>	<code>\peek_meaning_remove_ignore_spaces:NTF <test token> {(true code)} {(false code)}</code>
Updated: 2012-12-05	Tests if the next non-space <i><token></i> in the input stream has the same meaning as the <i><test token></i> (as defined by the test <code>\token_if_eq_meaning:NNTF</code>). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the <i><token></i> is removed from the input stream if the test is true. The function then places either the <i><true code></i> or <i><false code></i> in the input stream (as appropriate to the result of the test).

`\peek_N_type:TF`

Updated: 2012-12-20

`\peek_N_type:TF {<true code>} {<false code>}`

Tests if the next $\langle token \rangle$ in the input stream can be safely grabbed as an N-type argument. The test is $\langle false \rangle$ if the next $\langle token \rangle$ is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in L^AT_EX3) and $\langle true \rangle$ in all other cases. Note that a $\langle true \rangle$ result ensures that the next $\langle token \rangle$ is a valid N-type argument. However, if the next $\langle token \rangle$ is for instance `\c_space_token`, the test takes the $\langle false \rangle$ branch, even though the next $\langle token \rangle$ is in fact a valid N-type argument. The $\langle token \rangle$ is left in the input stream after the $\langle true code \rangle$ or $\langle false code \rangle$ (as appropriate to the result of the test).

7 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on T_EX's state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, `\use:n` is a five-letter control sequence.
- An active character token, characterized by its character code (between 0 and 1114111 for LuaT_EX and X_ET_EX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).

There are also a few internal tokens. The following list may be incomplete in some engines.

- Expanding `\the\font` results in a token that looks identical to the command that was used to select the current font (such as `\tenrm`) but it differs from it in shape.
- A “frozen” `\relax`, which differs from the primitive in shape (but has the same meaning), is inserted when the closing `\fi` of a conditional is encountered before the conditional is evaluated.
- Expanding `\noexpand \token` (when the $\langle token \rangle$ is expandable) results in an internal token, displayed (temporarily) as `\notexpanded: \token`, whose shape coincides with the $\langle token \rangle$ and whose meaning differs from `\relax`.
- An `\outer endtemplate:` can be encountered when peeking ahead at the next token; this expands to another internal token, `end of alignment template`.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.

- In LuaTeX, there is also the strange case of “bytes” `~~~~~1100xy` where x, y are any two lowercase hexadecimal digits, so that the hexadecimal number ranges from `\text{110000}=1114112$` to `$1100ff = 1114367`. These are used to output individual bytes to files, rather than UTF-8. For the purposes of token comparisons they behave like non-expandable primitive control sequences (*not characters*) whose `\meaning` is `the_character_` followed by the given byte. If this byte is in the range `80–ff` this gives an “invalid utf-8 sequence” error: applying `\token_to_str:N` or `\token_to_meaning:N` to these tokens is unsafe. Unfortunately, they don’t seem to be detectable safely by any means except perhaps Lua code.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We call these tokens *explicit* character tokens. Category codes that a character token can have are listed below by giving a sample output of the TeX primitive `\meaning`, together with their L^AT_EX3 names and most common example:

- 1 begin-group character (`group_begin`, often `{`),
- 2 end-group character (`group_end`, often `}`),
- 3 math shift character (`math_toggle`, often `$`),
- 4 alignment tab character (`alignment`, often `&`),
- 6 macro parameter character (`parameter`, often `#`),
- 7 superscript character (`math_superscript`, often `^`),
- 8 subscript character (`math_subscript`, often `_`),
- 10 blank space (`space`, often character code 32),
- 11 the letter (`letter`, such as `A`),
- 12 the character (`other`, such as `0`).

Category code 13 (**active**) is discussed below. Input characters can also have several other category codes which do not lead to character tokens for later processing: 0 (`escape`), 5 (`end_line`), 9 (`ignore`), 14 (`comment`), and 15 (`invalid`).

The meaning of a control sequence or active character can be identical to that of any character token listed above (with any character code), and we call such tokens *implicit* character tokens. The meaning is otherwise in the following list:

- a macro, used in L^AT_EX3 for most functions and some variables (`t1`, `fp`, `seq`, ...),
- a primitive such as `\def` or `\topmark`, used in L^AT_EX3 for some functions,
- a register such as `\count123`, used in L^AT_EX3 for the implementation of some variables (`int`, `dim`, ...),
- a constant integer such as `\char"56` or `\mathchar"121`,
- a font selection command,
- undefined.

Macros can be `\protected` or not, `\long` or not (the opposite of what L^AT_EX3 calls `nopar`), and `\outer` or not (unused in L^AT_EX3). Their `\meaning` takes the form

$\langle \text{prefix} \rangle \text{ macro} : \langle \text{argument} \rangle \rightarrow \langle \text{replacement} \rangle$

where $\langle \text{prefix} \rangle$ is among `\protected\long\outer`, $\langle \text{argument} \rangle$ describes parameters that the macro expects, such as `#1#2#3`, and $\langle \text{replacement} \rangle$ describes how the parameters are manipulated, such as `\int_eval:n{#2+#1*#3}`.

Now is perhaps a good time to mention some subtleties relating to tokens with category code 10 (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code 32 and category code 10.

When a macro takes an undelimited argument, explicit space characters (with character code 32 and category code 10) are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then T_EX scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens “N-type”, as they are suitable to be used as an argument for a function with the signature `:N`.

When a macro takes a delimited argument T_EX scans ahead until finding the delimiter (outside any pairs of begin-group/end-group explicit characters), and the resulting list of tokens (with outer braces removed) becomes the argument. Note that explicit space characters at the start of the argument are *not* ignored in this case (and they prevent brace-stripping).

Part XVII

The **I3prop** package

Property lists

LATEX3 implements a “property list” data type, which contain an unordered list of entries each of which consists of a $\langle key \rangle$ and an associated $\langle value \rangle$. The $\langle key \rangle$ and $\langle value \rangle$ may both be any $\langle balanced\ text \rangle$. It is possible to map functions to property lists such that the function is applied to every key–value pair within the list.

Each entry in a property list must have a unique $\langle key \rangle$: if an entry is added to a property list which already contains the $\langle key \rangle$ then the new entry overwrites the existing one. The $\langle keys \rangle$ are compared on a string basis, using the same method as `\str_if_eq:nn`.

Property lists are intended for storing key-based information for use within code. This is in contrast to key–value lists, which are a form of *input* parsed by the `keys` module.

1 Creating and initialising property lists

`\prop_new:N`
`\prop_new:c`

`\prop_new:N` $\langle property\ list \rangle$
Creates a new $\langle property\ list \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle property\ list \rangle$ initially contains no entries.

`\prop_clear:N`
`\prop_clear:c`
`\prop_gclear:N`
`\prop_gclear:c`

`\prop_clear:N` $\langle property\ list \rangle$
Clears all entries from the $\langle property\ list \rangle$.

`\prop_clear_new:N`
`\prop_clear_new:c`
`\prop_gclear_new:N`
`\prop_gclear_new:c`

`\prop_clear_new:N` $\langle property\ list \rangle$
Ensures that the $\langle property\ list \rangle$ exists globally by applying `\prop_new:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

`\prop_set_eq:NN`
`\prop_set_eq:(cN|Nc|cc)`
`\prop_gset_eq:NN`
`\prop_gset_eq:(cN|Nc|cc)`

`\prop_set_eq:NN` $\langle property\ list_1 \rangle$ $\langle property\ list_2 \rangle$
Sets the content of $\langle property\ list_1 \rangle$ equal to that of $\langle property\ list_2 \rangle$.

`\prop_set_from_keyval:Nn`
`\prop_set_from_keyval:cn`
`\prop_gset_from_keyval:Nn`
`\prop_gset_from_keyval:cn`

`\prop_set_from_keyval:Nn` $\langle prop\ var \rangle$
{
 $\langle key_1 \rangle = \langle value_1 \rangle$,
 $\langle key_2 \rangle = \langle value_2 \rangle$, ...}

Sets $\langle prop\ var \rangle$ to contain key–value pairs given in the second argument. If duplicate keys appear only one of the values is kept.

New: 2017-11-28
Updated: 2019-08-25

```
\prop_const_from_keyval:Nn
\prop_const_from_keyval:cn
```

New: 2017-11-28
Updated: 2019-08-25

```
\prop_const_from_keyval:Nn <prop var>
{
<key1> = <value1> ,
<key2> = <value2> , ...
}
```

Creates a new constant *<prop var>* or raises an error if the name is already taken. The *<prop var>* is set globally to contain key–value pairs given in the second argument. If duplicate keys appear only one of the values is kept.

2 Adding entries to property lists

```
\prop_put:Nnn
\prop_put:(NnV|Nno|Nnx|NVn|NVV|NVx|Nvx|Non|Noo|Nxx|cnn|cnV|cno|cnx|cVn|cVV|cVx|cvx|con|coo|cxx)
\prop_gput:Nnn
\prop_gput:(NnV|Nno|Nnx|NVn|NVV|NVx|Nvx|Non|Noo|Nxx|cnn|cnV|cno|cnx|cVn|cVV|cVx|cvx|con|coo|cxx)
```

```
\prop_put:Nnn
<property list>
{{<key>}}
{{<value>}}
```

Updated: 2012-07-09

Adds an entry to the *<property list>* which may be accessed using the *<key>* and which has *<value>*. Both the *<key>* and *<value>* may contain any *<balanced text>*. The *<key>* is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored. If the *<key>* is already present in the *<property list>*, the existing entry is overwritten by the new *<value>*.

```
\prop_put_if_new:Nnn
\prop_put_if_new:cnn
\prop_gput_if_new:Nnn
\prop_gput_if_new:cnn
```

```
\prop_put_if_new:Nnn <property list> {{<key>}} {{<value>}}
```

If the *<key>* is present in the *<property list>* then no action is taken. If the *<key>* is not present in the *<property list>* then a new entry is added. Both the *<key>* and *<value>* may contain any *<balanced text>*. The *<key>* is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored.

3 Recovering values from property lists

```
\prop_get:NnN
\prop_get:(NVN|NvN|NoN|cnN|cVN|cvN|coN)
```

Updated: 2011-08-28

```
\prop_get:NnN <property list> {{<key>}} <tl var>
```

Recovering the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* is set to the special marker `\q_no_value`. The *<token list variable>* is set within the current T_EX group. See also `\prop_get:NnNTF`.

```
\prop_pop:NnN
\prop_pop:(NoN|cnN|coN)
```

Updated: 2011-08-18

```
\prop_pop:NnN <property list> {{<key>}} <tl var>
```

Recovering the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* is set to the special marker `\q_no_value`. The *<key>* and *<value>* are then deleted from the property list. Both assignments are local. See also `\prop_pop:NnNTF`.

```
\prop_gpop:NnN
\prop_gpop:(NoN|cnN|coN)
```

Updated: 2011-08-18

```
\prop_gpop:NnN <property list> {<key>} <tl var>
```

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* is set to the special marker `\q_no_value`. The *<key>* and *<value>* are then deleted from the property list. The *<property list>* is modified globally, while the assignment of the *<token list variable>* is local. See also `\prop_gpop:NnNTF`.

```
\prop_item:Nn *
\prop_item:cn *
```

New: 2014-07-17

```
\prop_item:Nn <property list> {<key>}
```

Expands to the *<value>* corresponding to the *<key>* in the *<property list>*. If the *<key>* is missing, this has an empty expansion.

TeXhackers note: This function is slower than the non-expandable analogue `\prop_get:NnN`. The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<value>* does not expand further when appearing in an x-type argument expansion.

```
\prop_count:N *
\prop_count:c *
```

```
\prop_count:N <property list>
```

Leaves the number of key–value pairs in the *<property list>* in the input stream as an *<integer denotation>*.

4 Modifying property lists

```
\prop_remove:Nn
\prop_remove:(NV|cn|cV)
\prop_gremove:Nn
\prop_gremove:(NV|cn|cV)
```

New: 2012-05-12

```
\prop_remove:Nn <property list> {<key>}
```

Removes the entry listed under *<key>* from the *<property list>*. If the *<key>* is not found in the *<property list>* no change occurs, i.e there is no need to test for the existence of a key before deleting it.

5 Property list conditionals

```
\prop_if_exist_p:N *
\prop_if_exist_p:c *
\prop_if_exist:NTF *
\prop_if_exist:cTF *
```

New: 2012-03-03

```
\prop_if_exist_p:N <property list>
\prop_if_exist:NTF <property list> {<true code>} {<false code>}
```

Tests whether the *<property list>* is currently defined. This does not check that the *<property list>* really is a property list variable.

```
\prop_if_empty_p:N *
\prop_if_empty_p:c *
\prop_if_empty:NTF *
\prop_if_empty:cTF *
```

```
\prop_if_empty_p:N <property list>
\prop_if_empty:NTF <property list> {<true code>} {<false code>}
```

Tests if the *<property list>* is empty (containing no entries).

```
\prop_if_in_p:Nn          * \prop_if_in:NnTF <property list> {\<key>} {\<true code>} {\<false code>}
\prop_if_in_p:(NV|No|cn|cV|co) *
\prop_if_in:NnTF *
\prop_if_in:(NV|No|cn|cV|co)TF *
```

Updated: 2011-09-15

Tests if the *<key>* is present in the *<property list>*, making the comparison using the method described by `\str_if_eq:nnTF`.

TeXhackers note: This function iterates through every key–value pair in the *<property list>* and is therefore slower than using the non-expandable `\prop_get:NnTF`.

6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated valued. This makes them useful for cases where different cases follow dependent on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

```
\prop_get:NnNTF
\prop_get:(NVN|NvN|NoN|cnN|cVN|cvN|coN)TF
```

Updated: 2012-05-19

If the *<key>* is not present in the *<property list>*, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<key>* is present in the *<property list>*, stores the corresponding *<value>* in the *<token list variable>* without removing it from the *<property list>*, then leaves the *<true code>* in the input stream. The *<token list variable>* is assigned locally.

```
\prop_pop:NnNTF
\prop_pop:cnNTF
```

New: 2011-08-18
Updated: 2012-05-19

If the *<key>* is not present in the *<property list>*, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<key>* is present in the *<property list>*, pops the corresponding *<value>* in the *<token list variable>*, *i.e.* removes the item from the *<property list>*. Both the *<property list>* and the *<token list variable>* are assigned locally.

```
\prop_gpop:NnNTF
\prop_gpop:cnNTF
```

New: 2011-08-18
Updated: 2012-05-19

If the *<key>* is not present in the *<property list>*, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<key>* is present in the *<property list>*, pops the corresponding *<value>* in the *<token list variable>*, *i.e.* removes the item from the *<property list>*. The *<property list>* is modified globally, while the *<token list variable>* is assigned locally.

7 Mapping to property lists

All mappings are done at the current group level, *i.e.* any local assignments made by the `<function>` or `<code>` discussed below remain in effect after the loop.

`\prop_map_function:NN` ☆
`\prop_map_function:cN` ☆
Updated: 2013-01-08

`\prop_map_function:NN <property list> <function>`

Applies `<function>` to every `<entry>` stored in the `<property list>`. The `<function>` receives two arguments for each iteration: the `<key>` and associated `<value>`. The order in which `<entries>` are returned is not defined and should not be relied upon. To pass further arguments to the `<function>`, see `\prop_map_tokens:Nn`.

`\prop_map_inline:Nn`
`\prop_map_inline:cn`
Updated: 2013-01-08

`\prop_map_inline:Nn <property list> {<inline function>}`

Applies `<inline function>` to every `<entry>` stored within the `<property list>`. The `<inline function>` should consist of code which receives the `<key>` as #1 and the `<value>` as #2. The order in which `<entries>` are returned is not defined and should not be relied upon.

`\prop_map_tokens:Nn` ☆
`\prop_map_tokens:cn` ☆

`\prop_map_tokens:Nn <property list> {<code>}`

Analogue of `\prop_map_function:NN` which maps several tokens instead of a single function. The `<code>` receives each key–value pair in the `<property list>` as two trailing brace groups. For instance,

```
\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }
```

expands to the value corresponding to `mykey`: for each pair in `\l_my_prop` the function `\str_if_eq:nnT` receives `mykey`, the `<key>` and the `<value>` as its three arguments. For that specific task, `\prop_item:Nn` is faster.

`\prop_map_break:` ☆
Updated: 2012-06-29

`\prop_map_break:`

Used to terminate a `\prop_map_...` function before all entries in the `<property list>` have been processed. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break:
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map_...` scenario leads to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\prop_map_break:n` ☆

Updated: 2012-06-29

`\prop_map_break:n {<code>}`

Used to terminate a `\prop_map_...` function before all entries in the *property list* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
    { \prop_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map_...` scenario leads to low level T_EX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

8 Viewing property lists

`\prop_show:N`

`\prop_show:c`

Updated: 2015-08-01

`\prop_show:N <property list>`

Displays the entries in the *property list* in the terminal.

`\prop_log:N`

`\prop_log:c`

New: 2014-08-12

Updated: 2015-08-01

`\prop_log:N <property list>`

Writes the entries in the *property list* in the log file.

9 Scratch property lists

`\l_tmpa_prop`

`\l_tmpb_prop`

New: 2012-06-23

Scratch property lists for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_prop`

`\g_tmpb_prop`

New: 2012-06-23

Scratch property lists for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

10 Constants

\c_empty_prop A permanently-empty property list used for internal comparisons.

Part XVIII

The `l3msg` package

Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The `l3msg` module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by `l3msg` to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example `error`, `warning` or `info`.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behaviour can be altered and messages can be entirely suppressed.

1 Creating new messages

All messages have to be created before they can be used. The text of messages is automatically wrapped to the length available in the console. As a result, formatting is only needed where it helps to show meaning. In particular, `\\\` may be used to force a new line and `_` forces an explicit space. Additionally, `\{`, `\#`, `\}`, `\%` and `\~` can be used to produce the corresponding character.

Messages may be subdivided by *one level* using the `/` character. This is used within the message filtering system to allow for example the L^AT_EX kernel messages to belong to the module L^AT_EX while still being filterable at a more granular level. Thus for example

```
\msg_new:nnnn { mymodule } { submodule / message } ...
```

will allow to filter out specifically messages from the `submodule`.

```
\msg_new:nnnn
\msg_new:nnn
```

Updated: 2011-08-16

`\msg_new:nnnn {<module>} {<message>} {<text>} {<more text>}`

Creates a `<message>` for a given `<module>`. The message is defined to first give `<text>` and then `<more text>` if the user requests it. If no `<more text>` is available then a standard text is given instead. Within `<text>` and `<more text>` four parameters (#1 to #4) can be used: these will be supplied at the time the message is used. An error is raised if the `<message>` already exists.

```
\msg_set:nnnn
\msg_set:nnn
\msg_gset:nnnn
\msg_gset:nnn
```

`\msg_set:nnnn {<module>} {<message>} {<text>} {<more text>}`

Sets up the text for a `<message>` for a given `<module>`. The message is defined to first give `<text>` and then `<more text>` if the user requests it. If no `<more text>` is available then a standard text is given instead. Within `<text>` and `<more text>` four parameters (#1 to #4) can be used: these will be supplied at the time the message is used.

\msg_if_exist_p:nn ★ \msg_if_exist_p:nn {*module*} {*message*}
\msg_if_exist:nnTF ★ \msg_if_exist:nnTF {*module*} {*message*} {*true code*} {*false code*}

New: 2012-03-03

2 Contextual information for messages

\msg_line_context: ★

\msg_line_context:

Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is proceeded by the text `on line`.

\msg_line_number: ★

\msg_line_number:

Prints the current line number when a message is given.

\msg_fatal_text:n ★

\msg_fatal_text:n {*module*}

Produces the standard text

Fatal Package *module* Error

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *module* to be included.

\msg_critical_text:n ★

\msg_critical_text:n {*module*}

Produces the standard text

Critical Package *module* Error

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *module* to be included.

\msg_error_text:n ★

\msg_error_text:n {*module*}

Produces the standard text

Package *module* Error

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *module* to be included.

\msg_warning_text:n ★

\msg_warning_text:n {*module*}

Produces the standard text

Package *module* Warning

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *module* to be included. The *type* of *module* may be adjusted: `Package` is the standard outcome: see `\msg_module_type:n`.

<code>\msg_info_text:n *</code>	<code>\msg_info_text:n {<module>}</code> Produces the standard text: <code>Package <module> Info</code>
	This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included. The <i><type></i> of <i><module></i> may be adjusted: Package is the standard outcome: see <code>\msg_module_type:n</code> .
<code>\msg_module_name:n *</code> New: 2018-10-10	<code>\msg_module_name:n {<module>}</code> Expands to the public name of the <i><module></i> as defined by <code>\g_msg_module_name_prop</code> (or otherwise leaves the <i><module></i> unchanged).
<code>\msg_module_type:n *</code> New: 2018-10-10	<code>\msg_module_type:n {<module>}</code> Expands to the description which applies to the <i><module></i> , for example a Package or Class . The information here is defined in <code>\g_msg_module_type_prop</code> , and will default to Package if an entry is not present.
<code>\msg_see_documentation_text:n *</code> Updated: 2018-09-30	<code>\msg_see_documentation_text:n {<module>}</code> Produces the standard text <code>See the <module> documentation for further information.</code>
	This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included. The name of the <i><module></i> may be altered by use of <code>\g_msg_module_documentation_prop</code>
<code>\g_msg_module_name_prop</code> New: 2018-10-10	Provides a mapping between the module name used for messages, and that for documentation. For example, L ^A T _E X3 core messages are stored in the reserved L ^A T _E X tree, but are printed as L ^A T _E X3.
<code>\g_msg_module_type_prop</code> New: 2018-10-10	Provides a mapping between the module name used for messages, and that type of module. For example, for L ^A T _E X3 core messages, an empty entry is set here meaning that they are not described using the standard Package text.

3 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments are ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments are converted to strings before being added to the message text: the x-type variants should be used to expand material.

```
\msg_fatal:nnnnnn
\msg_fatal:nnxxxx
\msg_fatal:nnnnn
\msg_fatal:nnxxx
\msg_fatal:nnnn
\msg_fatal:nnxx
\msg_fatal:nnn
\msg_fatal:nnx
\msg_fatal:nn
```

Updated: 2012-08-11

```
\msg_fatal:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}
```

Issues *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. After issuing a fatal error the TeX run halts. No PDF file will be produced in this case (DVI mode runs may produce a truncated DVI file).

```
\msg_critical:nnnnnn
\msg_critical:nnxxxx
\msg_critical:nnnnn
\msg_critical:nnxxx
\msg_critical:nnnn
\msg_critical:nnxx
\msg_critical:nnn
\msg_critical:nnx
\msg_critical:nn
```

Updated: 2012-08-11

```
\msg_critical:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}
```

Issues *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. After issuing a critical error, TeX stops reading the current input file. This may halt the TeX run (if the current file is the main file) or may abort reading a sub-file.

TeXhackers note: The TeX \endinput primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

```
\msg_error:nnnnnn
\msg_error:nnxxxx
\msg_error:nnnnn
\msg_error:nnxxx
\msg_error:nnnn
\msg_error:nnxx
\msg_error:nnn
\msg_error:nnx
\msg_error:nn
```

Updated: 2012-08-11

```
\msg_error:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}
```

Issues *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The error interrupts processing and issues the text at the terminal. After user input, the run continues.

```
\msg_warning:nnnnnn
\msg_warning:nnxxxx
\msg_warning:nnnnn
\msg_warning:nnxxx
\msg_warning:nnnn
\msg_warning:nnxx
\msg_warning:nnn
\msg_warning:nnx
\msg_warning:nn
```

Updated: 2012-08-11

```
\msg_warning:nnxxxx {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}
```

Issues *<module>* warning *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The warning text is added to the log file and the terminal, but the TeX run is not interrupted.

```
\msg_info:nnnnnn
\msg_info:nnxxxx
\msg_info:nnnnn
\msg_info:nnxxx
\msg_info:nnnn
\msg_info:nnxx
\msg_info:nnn
\msg_info:nnx
\msg_info:nn
```

Updated: 2012-08-11

```
\msg_info:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg four}
```

Issues *module* information *message*, passing *arg one* to *arg four* to the text-creating functions. The information text is added to the log file.

```
\msg_log:nnnnnn
\msg_log:nnxxxx
\msg_log:nnnnn
\msg_log:nnxxx
\msg_log:nnnn
\msg_log:nnxx
\msg_log:nnn
\msg_log:nnx
\msg_log:nn
```

Updated: 2012-08-11

```
\msg_log:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg four}
```

Issues *module* information *message*, passing *arg one* to *arg four* to the text-creating functions. The information text is added to the log file: the output is briefer than \msg_info:nnnnnn.

```
\msg_term:nnnnnn
\msg_term:nnxxxx
\msg_term:nnnnn
\msg_term:nnxxx
\msg_term:nnnn
\msg_term:nnxx
\msg_term:nnn
\msg_term:nnx
\msg_term:nn
```

New: 2020-07-16

```
\msg_term:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg four}
```

Issues *module* information *message*, passing *arg one* to *arg four* to the text-creating functions. The information text is printed on the terminal (and added to the log file): the output is similar to that of \msg_log:nnnnnn.

```
\msg_none:nnnnnn
\msg_none:nnxxxx
\msg_none:nnnnn
\msg_none:nnxxx
\msg_none:nnnn
\msg_none:nnxx
\msg_none:nnn
\msg_none:nnx
\msg_none:nn
```

Updated: 2012-08-11

```
\msg_none:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg four}
```

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

3.1 Expandable error messages

In very rare cases it may be necessary to produce errors in an expansion-only context. The functions in this section should only be used if there is no alternative approach using \msg_error:nnnnnn or other non-expandable commands from the previous section.

Despite having a similar interface as non-expandable messages, expandable errors must be handled internally very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable. As a result, short-hands such as \{ or \\ do not work, and messages must be very short (with default settings, they are truncated after approximately 50 characters). It is advisable to ensure that the message is understandable even when truncated, by putting the most important information up front. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

```
\msg_expandable_error:nnnnn * \msg_expandable_error:nnnnnn {<module>} {<message>} {<arg one>} {<arg
\msg_expandable_error:nnfff * two>} {<arg three>} {<arg four>}
\msg_expandable_error:nnnnn *
\msg_expandable_error:nnfff *
\msg_expandable_error:nnnn  *
\msg_expandable_error:nnff  *
\msg_expandable_error:nnn   *
\msg_expandable_error:nnf   *
\msg_expandable_error:nn    *
```

New: 2015-08-06
Updated: 2019-02-28

Issues an “Undefined error” message from TeX itself using the undefined control sequence \::error then prints “! <module>: ”<error message>, which should be short. With default settings, anything beyond approximately 60 characters long (or bytes in some engines) is cropped. A leading space might be removed as well.

4 Redirecting messages

Each message has a “name”, which can be used to alter the behaviour of the message when it is given. Thus we might have

```
\msg_new:nnnn { module } { my-message } { Some~text } { Some~more~text }
```

to define a message, with

```
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this raises an error. However, we could alter the behaviour with

```
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```
\msg_redirect_module:nnn { module } { error } { warning }
```

to alter only messages from that module, or even

```
\msg_redirect_name:nnn { module } { my-message } { warning }
```

to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class raises an error immediately. Infinite loops are prevented by eliminating the redirection starting from the target of the redirection that caused the loop to appear. Namely, if redirections are requested as $A \rightarrow B$, $B \rightarrow C$ and $C \rightarrow A$ in this order, then the $A \rightarrow B$ redirection is cancelled.

\msg_redirect_class:nn

Updated: 2012-04-27

```
\msg_redirect_class:nn {\<class one>} {\<class two>}
```

Changes the behaviour of messages of *<class one>* so that they are processed using the code for those of *<class two>*.

\msg_redirect_module:nnn

Updated: 2012-04-27

```
\msg_redirect_module:nnn {\<module>} {\<class one>} {\<class two>}
```

Redirects message of *<class one>* for *<module>* to act as though they were from *<class two>*. Messages of *<class one>* from sources other than *<module>* are not affected by this redirection. This function can be used to make some messages “silent” by default. For example, all of the **warning** messages of *<module>* could be turned off with:

```
\msg_redirect_module:nnn { module } { warning } { none }
```

\msg_redirect_name:nnn

Updated: 2012-04-27

```
\msg_redirect_name:nnn {\<module>} {\<message>} {\<class>}
```

Redirects a specific *<message>* from a specific *<module>* to act as a member of *<class>* of messages. No further redirection is performed. This function can be used to make a selected message “silent” without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

Part XIX

The **I3file** package

File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix `\file_...`, while others are used to work with files on a line by line basis and have prefix `\ior_...` (reading) or `\iow_...` (writing).

It is important to remember that when reading external files \TeX attempts to locate them using both the operating system path and entries in the \TeX file database (most \TeX systems use such a database). Thus the “current path” for \TeX is somewhat broader than that for other programs.

For functions which expect a $\langle\text{file name}\rangle$ argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Active characters (as declared in `\l_char_active_seq`) are *not* expanded, allowing the direct use of these in file names. Quote tokens ("") are not permitted in file names as they are reserved for internal use by some \TeX primitives.

Spaces are trimmed at the beginning and end of the file name: this reflects the fact that some file systems do not allow or interact unpredictably with spaces in these positions. When no extension is given, this will trim spaces from the start of the name only.

1 Input–output stream management

As \TeX engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in $\text{\LaTeX}3$. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

`\ior_new:N`
`\ior_new:c`
`\iow_new:N`
`\iow_new:c`
New: 2011-09-26
Updated: 2011-12-27

`\ior_new:N` $\langle\text{stream}\rangle$
`\iow_new:N` $\langle\text{stream}\rangle$

Globally reserves the name of the $\langle\text{stream}\rangle$, either for reading or for writing as appropriate. The $\langle\text{stream}\rangle$ is not opened until the appropriate `\..._open:Nn` function is used. Attempting to use a $\langle\text{stream}\rangle$ which has not been opened is an error, and the $\langle\text{stream}\rangle$ will behave as the corresponding `\c_term_...`.

`\ior_open:Nn`
`\ior_open:cn`
Updated: 2012-02-10

`\ior_open:Nn` $\langle\text{stream}\rangle$ $\{\langle\text{file name}\rangle\}$

Opens $\langle\text{file name}\rangle$ for reading using $\langle\text{stream}\rangle$ as the control sequence for file access. If the $\langle\text{stream}\rangle$ was already open it is closed before the new operation begins. The $\langle\text{stream}\rangle$ is available for access immediately and will remain allocated to $\langle\text{file name}\rangle$ until a `\ior_close:N` instruction is given or the \TeX run ends. If the file is not found, an error is raised.

\ior_open:NnTF
\ior_open:cNTF

New: 2013-01-12

`\ior_open:NnTF <stream> {<file name>} {<true code>} {<false code>}`

Opens *<file name>* for reading using *<stream>* as the control sequence for file access. If the *<stream>* was already open it is closed before the new operation begins. The *<stream>* is available for access immediately and will remain allocated to *<file name>* until a `\ior_close:N` instruction is given or the T_EX run ends. The *<true code>* is then inserted into the input stream. If the file is not found, no error is raised and the *<false code>* is inserted into the input stream.

\iow_open:Nn
\iow_open:cN

Updated: 2012-02-09

`\iow_open:Nn <stream> {<file name>}`

Opens *<file name>* for writing using *<stream>* as the control sequence for file access. If the *<stream>* was already open it is closed before the new operation begins. The *<stream>* is available for access immediately and will remain allocated to *<file name>* until a `\iow_close:N` instruction is given or the T_EX run ends. Opening a file for writing clears any existing content in the file (*i.e.* writing is *not* additive).

\ior_close:N
\ior_close:c
\iow_close:N
\iow_close:c

Updated: 2012-07-31

`\ior_close:N <stream>`
`\iow_close:N <stream>`

Closes the *<stream>*. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.

\ior_show_list:
\ior_log_list:
\iow_show_list:
\iow_log_list:

New: 2017-06-27

`\ior_show_list:`
`\ior_log_list:`
`\iow_show_list:`
`\iow_log_list:`

Display (to the terminal or log file) a list of the file names associated with each open (read or write) stream. This is intended for tracking down problems.

1.1 Reading from files

Reading from files and reading from the terminal are separate processes in `expl3`. The functions `\ior_get>NN` and `\ior_str_get>NN`, and their branching equivalents, are designed to work with *files*.

```
\ior_get:NN  
\ior_get:NNTF
```

New: 2012-06-24
Updated: 2019-03-23

```
\ior_get:NN <stream> <token list variable>  
\ior_get:NNTF <stream> <token list variable> <true code> <false code>
```

Function that reads one or more lines (until an equal number of left and right braces are found) from the file input *<stream>* and stores the result locally in the *<token list>* variable. The material read from the *<stream>* is tokenized by \TeX according to the category codes and \endlinechar in force when the function is used. Assuming normal settings, any lines which do not end in a comment character % have the line ending converted to a space, so for example input

```
a b c
```

results in a token list `a\b\c`. Any blank line is converted to the token `\par`. Therefore, blank lines can be skipped by using a test such as

```
\ior_get:NN \l_my_stream \l_tmpa_t1  
\tl_set:Nn \l_tmpb_t1 { \par }  
\tl_if_eq:NNF \l_tmpa_t1 \l_tmpb_t1  
...
```

Also notice that if multiple lines are read to match braces then the resulting token list can contain `\par` tokens. In the non-branching version, where the *<stream>* is not open the *<tl var>* is set to `\q_no_value`.

TeXhackers note: This protected macro is a wrapper around the \TeX primitive `\read`. Regardless of settings, \TeX replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code \endlinechar , omitted if \endlinechar is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

```
\ior_str_get:NN  
\ior_str_get:NNTF
```

New: 2016-12-04
Updated: 2019-03-23

```
\ior_str_get:NN <stream> <token list variable>  
\ior_str_get:NNTF <stream> <token list variable> <true code> <false code>
```

Function that reads one line from the file input *<stream>* and stores the result locally in the *<token list>* variable. The material is read from the *<stream>* as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the *<token list variable>* being empty. Unlike `\ior_get:NN`, line ends do not receive any special treatment. Thus input

```
a b c
```

results in a token list `a b c` with the letters `a`, `b`, and `c` having category code 12. In the non-branching version, where the *<stream>* is not open the *<tl var>* is set to `\q_no_value`.

TeXhackers note: This protected macro is a wrapper around the ε - \TeX primitive `\readline`. Regardless of settings, \TeX removes trailing space and tab characters (character codes 32 and 9). However, the end-line character normally added by this primitive is not included in the result of `\ior_str_get:NN`.

All mappings are done at the current group level, *i.e.* any local assignments made by the *<function>* or *<code>* discussed below remain in effect after the loop.

\ior_map_inline:NnNew: 2012-02-11

\ior_map_inline:Nn *stream* {*inline function*}

Applies the *inline function* to each set of *lines* obtained by calling \ior_get:NN until reaching the end of the file. TeX ignores any trailing new-line marker from the file it reads. The *inline function* should consist of code which receives the *line* as #1.

\ior_str_map_inline:NnNew: 2012-02-11

\ior_str_map_inline:Nn *stream* {*inline function*}

Applies the *inline function* to every *line* in the *stream*. The material is read from the *stream* as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The *inline function* should consist of code which receives the *line* as #1. Note that TeX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TeX also ignores any trailing new-line marker from the file it reads.

\ior_map_variable>NNnNew: 2019-01-13

\ior_map_variable>NNn *stream* {*tl var*} {*code*}

For each set of *lines* obtained by calling \ior_get:NN until reaching the end of the file, stores the *lines* in the *tl var* then applies the *code*. The *code* will usually make use of the *variable*, but this is not enforced. The assignments to the *variable* are local. Its value after the loop is the last set of *lines*, or its original value if the *stream* is empty. TeX ignores any trailing new-line marker from the file it reads. This function is typically faster than \ior_map_inline:Nn.

\ior_str_map_variable>NNnNew: 2019-01-13

\ior_str_map_variable>NNn *stream* {*variable*} {*code*}

For each *line* in the *stream*, stores the *line* in the *variable* then applies the *code*. The material is read from the *stream* as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The *code* will usually make use of the *variable*, but this is not enforced. The assignments to the *variable* are local. Its value after the loop is the last *line*, or its original value if the *stream* is empty. Note that TeX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TeX also ignores any trailing new-line marker from the file it reads. This function is typically faster than \ior_str_map_inline:Nn.

\ior_map_break:New: 2012-06-29

\ior_map_break:

Used to terminate a \ior_map_... function before all lines from the *stream* have been processed. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
    { \ior_map_break: }
    {
      % Do something useful
    }
}
```

Use outside of a \ior_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\ior_map_break:nNew: 2012-06-29

\ior_map_break:n {<code>}

Used to terminate a \ior_map_... function before all lines in the *stream* have been processed, inserting the *code* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
    { \ior_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a \ior_map_... scenario leads to low level T_EX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the *code* is inserted into the input stream. This depends on the design of the mapping function.

\ior_if_eof_p:N * **\ior_if_eof_p:N {<stream>}**
\ior_if_eof:NTF * **\ior_if_eof:NTF {<stream>} {<true code>} {<false code>}**Updated: 2012-02-10

Tests if the end of a file *stream* has been reached during a reading operation. The test also returns a **true** value if the *stream* is not open.

1.2 Writing to files

\iow_now:Nn
\iow_now:(Nx|cn|cx)Updated: 2012-06-05

\iow_now:Nn {<stream>} {<tokens>}

This functions writes *tokens* to the specified *stream* immediately (*i.e.* the write operation is called on expansion of \iow_now:Nn).

\iow_log:n
\iow_log:x

\iow_log:n {<tokens>}

This function writes the given *tokens* to the log (transcript) file immediately: it is a dedicated version of \iow_now:Nn.

\iow_term:n
\iow_term:x

\iow_term:n {<tokens>}

This function writes the given *tokens* to the terminal file immediately: it is a dedicated version of \iow_now:Nn.

```
\iow_shipout:Nn
\iow_shipout:(Nx|cn|cx)
```

```
\iow_shipout:Nn <stream> {\langle tokens \rangle}
```

This function writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (*i.e.* at shipout). The x-type variants expand the $\langle tokens \rangle$ at the point where the function is used but *not* when the resulting tokens are written to the $\langle stream \rangle$ (*cf.* $\backslash iow_shipout_x:Nn$). This may lead to the insertion of additional unwanted line-breaks.

TeXhackers note: When using `expl3` with a format other than L^AT_EX, new line characters inserted using $\backslash iow_newline:$ or using the line-wrapping code $\backslash iow_wrap:nnnN$ are not recognized in the argument of $\backslash iow_shipout:Nn$. This may lead to the insertion of additional unwanted line-breaks.

```
\iow_shipout_x:Nn
\iow_shipout_x:(Nx|cn|cx)
```

Updated: 2012-09-08

```
\iow_shipout_x:Nn <stream> {\langle tokens \rangle}
```

This function writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (*i.e.* at shipout). The $\langle tokens \rangle$ are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

TeXhackers note: This is a wrapper around the TeX primitive `\write`. When using `expl3` with a format other than L^AT_EX, new line characters inserted using $\backslash iow_newline:$ or using the line-wrapping code $\backslash iow_wrap:nnnN$ are not recognized in the argument of $\backslash iow_shipout:Nn$. This may lead to the insertion of additional unwanted line-breaks.

```
\iow_char:N *
```

```
\iow_char:N \langle char \rangle
```

Inserts $\langle char \rangle$ into the output stream. Useful when trying to write difficult characters such as %, {, }, *etc.* in messages, for example:

```
\iow_now:Nx \g_my_iow { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of $\backslash iow_now:Nn$).

```
\iow_newline: *
```

```
\iow_newline:
```

Function to add a new line within the $\langle tokens \rangle$ written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of $\backslash iow_now:Nn$).

TeXhackers note: When using `expl3` with a format other than L^AT_EX, the character inserted by $\backslash iow_newline:$ is not recognized by TeX, which may lead to the insertion of additional unwanted line-breaks. This issue only affects $\backslash iow_shipout:Nn$, $\backslash iow_shipout_x:Nn$ and direct uses of primitive operations.

1.3 Wrapping lines in output

\iow_wrap:nnnN
\iow_wrap:nxnN

New: 2012-06-28
Updated: 2017-12-04

\iow_wrap:nnnN {⟨text⟩} {⟨run-on text⟩} {⟨set up⟩} ⟨function⟩

This function wraps the ⟨text⟩ to a fixed number of characters per line. At the start of each line which is wrapped, the ⟨run-on text⟩ is inserted. The line character count targeted is the value of `\l_iow_line_count_int` minus the number of characters in the ⟨run-on text⟩ for all lines except the first, for which the target number of characters is simply `\l_iow_line_count_int` since there is no run-on text. The ⟨text⟩ and ⟨run-on text⟩ are exhaustively expanded by the function, with the following substitutions:

- `\\" or \iow_newline`: may be used to force a new line,
- `_` may be used to represent a forced space (for example after a control sequence),
- `\#, \%, \{, \}, \~` may be used to represent the corresponding character,
- `\iow_allow_break`: may be used to allow a line-break without inserting a space (this is experimental),
- `\iow_indent:n` may be used to indent a part of the ⟨text⟩ (not the ⟨run-on text⟩).

Additional functions may be added to the wrapping by using the ⟨set up⟩, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the ⟨text⟩ which is not to be expanded on wrapping should be converted to a string using `\token_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N`, etc.

The result of the wrapping operation is passed as a braced argument to the ⟨function⟩, which is typically a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the ⟨function⟩) consists of characters of category “other” (category code 12), with the exception of spaces which have category “space” (category code 10). This means that the output does *not* expand further when written to a file.

TeXhackers note: Internally, `\iow_wrap:nnnN` carries out an x-type expansion on the ⟨text⟩ to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` could be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the ⟨text⟩.

\iow_indent:n
New: 2011-09-21

\iow_indent:n {⟨text⟩}

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents ⟨text⟩ by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the ⟨text⟩. In case the indented ⟨text⟩ should appear on separate lines from the surrounding text, use `\\"` to force line breaks.

\l_iow_line_count_int
New: 2012-06-24

The maximum number of characters in a line to be written by the `\iow_wrap:nnnN` function. This value depends on the TeX system in use: the standard value is 78, which is typically correct for unmodified TeXlive and MiKTeX systems.

1.4 Constant input–output streams, and variables

`\g_tmpa_iow`

`\g_tmpb_iow`

New: 2017-12-11

Scratch input stream for global use. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\c_log_iow`

`\c_term_iow`

Constant output streams for writing to the log and to the terminal (plus the log), respectively.

`\g_tmpa_iow`

`\g_tmpb_iow`

New: 2017-12-11

Scratch output stream for global use. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

1.5 Primitive conditionals

`\if_eof:w *`

```
\if_eof:w <stream>
  <true code>
\else:
  <false code>
\fi:
```

Tests if the `<stream>` returns “end of file”, which is true for non-existent files. The `\else:` branch is optional.

TeXhackers note: This is the T_EX primitive `\ifeof`.

2 File operation functions

`\g_file_curr_dir_str`

`\g_file_curr_name_str`

`\g_file_curr_ext_str`

New: 2017-06-21

Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (*i.e.* if it is in the T_EX search path), and does *not* end in / other than the case that it is exactly equal to the root directory. The `<name>` and `<ext>` parts together make up the file name, thus the `<name>` part may be thought of as the “job name” for the current file. Note that T_EX does not provide information on the `<ext>` part for the main (top level) file and that this file always has an empty `<dir>` component. Also, the `<name>` here will be equal to `\c_sys_jobname_str`, which may be different from the real file name (if set using `--jobname`, for example).

\l_file_search_path_seqNew: 2017-06-18

Each entry is the path to a directory which should be searched when seeking a file. Each path can be relative or absolute, and should not include the trailing slash. The entries are not expanded when used so may contain active characters but should not feature any variable content. Spaces need not be quoted.

TeXhackers note: When working as a package in L^AT_EX 2 ε , `expl3` will automatically append the current `\input@path` to the set of values from `\l_file_search_path_seq`.

\file_if_exist:nTFUpdated: 2012-02-10

```
\file_if_exist:nTF {\<file name>} {\<true code>} {\<false code>}
```

Searches for `<file name>` using the current TeX search path and the additional paths controlled by `\l_file_search_path_seq`.

**\file_get:nnN
\file_get:nnNTF**New: 2019-01-16
Updated: 2019-02-16

```
\file_get:nnN {\<filename>} {\<setup>} {tl}  
\file_get:nnNTF {\<filename>} {\<setup>} {tl} {\<true code>} {\<false code>}
```

Defines `{tl}` to the contents of `<filename>`. Category codes may need to be set appropriately via the `<setup>` argument. The non-branching version sets the `{tl}` to `\q_no_value` if the file is not found. The branching version runs the `<true code>` after the assignment to `{tl}` if the file is found, and `<false code>` otherwise.

**\file_get_full_name:nN
\file_get_full_name:VN
\file_get_full_name:nNTF
\file_get_full_name:VNTF**Updated: 2019-02-16

```
\file_get_full_name:nN {\<file name>} {tl}  
\file_get_full_name:nNTF {\<file name>} {tl} {\<true code>} {\<false code>}
```

Searches for `<file name>` in the path as detailed for `\file_if_exist:nTF`, and if found sets the `{tl var}` the fully-qualified name of the file, *i.e.* the path and file name. This includes an extension `.tex` when the given `<file name>` has no extension but the file found has that extension. In the non-branching version, the `{tl var}` will be set to `\q_no_value` in the case that the file does not exist.

**\file_full_name:n ☆
\file_full_name:V ☆**New: 2019-09-03

```
\file_full_name:n {\<file name>}
```

Searches for `<file name>` in the path as detailed for `\file_if_exist:nTF`, and if found leaves the fully-qualified name of the file, *i.e.* the path and file name, in the input stream. This includes an extension `.tex` when the given `<file name>` has no extension but the file found has that extension. If the file is not found on the path, the expansion is empty.

```
\file_parse_full_name:nNNN  
\file_parse_full_name:VNNN
```

New: 2017-06-23
Updated: 2020-06-24

```
\file_parse_full_name:nNNN {\full name} <dir> <name> <ext>
```

Parses the *full name* and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The *dir*: everything up to the last / (path separator) in the *file path*. As with system PATH variables and related functions, the *dir* does *not* include the trailing / unless it points to the root directory. If there is no path (only a file name), *dir* is empty.
- The *name*: everything after the last / up to the last ., where both of those characters are optional. The *name* may contain multiple . characters. It is empty if *full name* consists only of a directory name.
- The *ext*: everything after the last . (including the dot). The *ext* is empty if there is no . after the last /.

Before parsing, the *full name* is expanded until only non-expandable tokens remain, except that active characters are also not expanded. Quotes ("") are invalid in file names and are discarded from the input.

```
\file_parse_full_name:n *
```

New: 2020-06-24

```
\file_parse_full_name:n {\full name}
```

Parses the *full name* as described for \file_parse_full_name:nNNN, and leaves *dir*, *name*, and *ext* in the input stream, each inside a pair of braces.

```
\file_parse_full_name_apply:nN * \file_parse_full_name_apply:nN {\full name} <function>
```

New: 2020-06-24

Parses the *full name* as described for \file_parse_full_name:nNNN, and passes *dir*, *name*, and *ext* as arguments to *function*, as an n-type argument each, in this order.

```
\file_hex_dump:n ☆  
\file_hex_dump:nnn ☆
```

New: 2019-11-19

```
\file_hex_dump:n {\file name}  
\file_hex_dump:nnn {\file name} {\start index} {\end index}
```

Searches for *file name* using the current TeX search path and the additional paths controlled by \l_file_search_path_seq. It then expands to leave the hexadecimal dump of the file content in the input stream. The file is read as bytes, which means that in contrast to most TeX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty. The {\start index} and {\end index} values work as described for \str_range:nnn.

```
\file_get_hex_dump:nN  
\file_get_hex_dump:nNTF  
\file_get_hex_dump:nnnN  
\file_get_hex_dump:nnnNTF
```

New: 2019-11-19

```
\file_get_hex_dump:nN {\file name} <tl var>  
\file_get_hex_dump:nnnN {\file name} {\start index} {\end index} <tl var>
```

Sets the *tl var* to the result of applying \file_hex_dump:n/\file_hex_dump:nnn to the *file*. If the file is not found, the *tl var* will be set to \q_no_value.

\file_mdfive_hash:n ☆

New: 2019-09-03

```
\file_mdfive_hash:n {\file name}
```

Searches for *file name* using the current T_EX search path and the additional paths controlled by `\l_file_search_path_seq`. It then expands to leave the MD5 sum generated from the contents of the file in the input stream. The file is read as bytes, which means that in contrast to most T_EX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty.

\file_get_mdfive_hash:nN

\file_get_mdfive_hash:nNTF

New: 2017-07-11

Updated: 2019-02-16

```
\file_get_mdfive_hash:nN {\file name} {tl var}
```

Sets the *tl var* to the result of applying `\file_mdfive_hash:n` to the *file*. If the file is not found, the *tl var* will be set to `\q_no_value`.

\file_size:n ☆

New: 2019-09-03

```
\file_size:n {\file name}
```

Searches for *file name* using the current T_EX search path and the additional paths controlled by `\l_file_search_path_seq`. It then expands to leave the size of the file in bytes in the input stream. When the file is not found, the result of expansion is empty.

\file_get_size:nN

\file_get_size:nNTF

New: 2017-07-09

Updated: 2019-02-16

```
\file_get_size:nN {\file name} {tl var}
```

Sets the *tl var* to the result of applying `\file_size:n` to the *file*. If the file is not found, the *tl var* will be set to `\q_no_value`. This is not available in older versions of X_ET_EX.

\file_timestamp:n ☆

New: 2019-09-03

```
\file_timestamp:n {\file name}
```

Searches for *file name* using the current T_EX search path and the additional paths controlled by `\l_file_search_path_seq`. It then expands to leave the modification timestamp of the file in the input stream. The timestamp is of the form D:*year*(*month*)*day*(*hour*)*minute*(*second*)*offset*, where the latter may be Z (UTC) or *plus-minus*(*hours*)*minutes*. When the file is not found, the result of expansion is empty. This is not available in older versions of X_ET_EX.

\file_get_timestamp:nN

\file_get_timestamp:nNTF

New: 2017-07-09

Updated: 2019-02-16

```
\file_get_timestamp:nN {\file name} {tl var}
```

Sets the *tl var* to the result of applying `\file_timestamp:n` to the *file*. If the file is not found, the *tl var* will be set to `\q_no_value`. This is not available in older versions of X_ET_EX.

```
\file_compare_timestamp_p:nNn ★ \file_compare_timestamp:nNn {⟨file-1⟩} ⟨comparator⟩ {⟨file-2⟩} {⟨true code⟩} {⟨false code⟩}
```

New: 2019-05-13

Updated: 2019-09-20

Compares the file stamps on the two ⟨files⟩ as indicated by the ⟨comparator⟩, and inserts either the ⟨true code⟩ or ⟨false case⟩ as required. A file which is not found is treated as older than any file which is found. This allows for example the construct

```
\file_compare_timestamp:nNnT { source-file } > { derived-file }
{
    % Code to regenerate derived file
}
```

to work when the derived file is entirely absent. The timestamp of two absent files is regarded as different. This is not available in older versions of X_HT_EX.

```
\file_input:n
```

Updated: 2017-06-26

```
\file_input:n {⟨file name⟩}
```

Searches for ⟨file name⟩ in the path as detailed for \file_if_exist:nTF, and if found reads in the file as additional L_AT_EX source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.

```
\file_if_exist_input:n
```

```
\file_if_exist_input:nF
```

New: 2014-07-02

```
\file_if_exist_input:n {⟨file name⟩}
```

```
\file_if_exist_input:nF {⟨file name⟩} {⟨false code⟩}
```

Searches for ⟨file name⟩ using the current T_EX search path and the additional paths controlled by \file_path_include:n. If found then reads in the file as additional L_AT_EX source as described for \file_input:n, otherwise inserts the ⟨false code⟩. Note that these functions do not raise an error if the file is not found, in contrast to \file_input:n.

```
\file_input_stop:
```

New: 2017-07-07

```
\file_input_stop:
```

Ends the reading of a file started by \file_input:n or similar before the end of the file is reached. Where the file reading is being terminated due to an error, \msg_critical:nn(nn) should be preferred.

TeXhackers note: This function must be used on a line on its own: T_EX reads files line-by-line and so any additional tokens in the “current” line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn’t help as it is the line where it is used that counts!

```
\file_show_list:
```

```
\file_log_list:
```

These functions list all files loaded by L_AT_EX 2_E commands that populate \@filelist or by \file_input:n. While \file_show_list: displays the list in the terminal, \file_log_list: outputs it to the log file only.

Part XX

The **I3skip** package

Dimensions and skips

LATEX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in `mu`). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

1 Creating and initialising `dim` variables

`\dim_new:N` `\dim_new:c`

Creates a new `<dimension>` or raises an error if the name is already taken. The declaration is global. The `<dimension>` is initially equal to 0 pt.

`\dim_const:Nn` `\dim_const:cn`

New: 2012-03-05

`\dim_const:<dimension> {<dimension expression>}`

Creates a new constant `<dimension>` or raises an error if the name is already taken. The value of the `<dimension>` is set globally to the `<dimension expression>`.

`\dim_zero:N`
`\dim_zero:c`
`\dim_gzero:N`
`\dim_gzero:c`

`\dim_zero:<dimension>`

Sets `<dimension>` to 0 pt.

`\dim_zero_new:N`
`\dim_zero_new:c`
`\dim_gzero_new:N`
`\dim_gzero_new:c`

New: 2012-01-07

`\dim_zero_new:<dimension>`

Ensures that the `<dimension>` exists globally by applying `\dim_new:N` if necessary, then applies `\dim_(g)zero:N` to leave the `<dimension>` set to zero.

`\dim_if_exist_p:N` *
`\dim_if_exist_p:c` *
`\dim_if_exist:NTF` *
`\dim_if_exist:cTF` *

`\dim_if_exist_p:<dimension>`

`\dim_if_exist:<dimension> {<true code>} {<false code>}`

Tests whether the `<dimension>` is currently defined. This does not check that the `<dimension>` really is a dimension variable.

New: 2012-03-03

2 Setting `dim` variables

`\dim_add:Nn` `\dim_add:Nn <dimension> {<dimension expression>}`
`\dim_add:cn` Adds the result of the `<dimension expression>` to the current content of the `<dimension>`.
`\dim_gadd:Nn`
`\dim_gadd:cn`

Updated: 2011-10-22

`\dim_set:Nn` `\dim_set:Nn <dimension> {<dimension expression>}`
`\dim_set:cn` Sets `<dimension>` to the value of `<dimension expression>`, which must evaluate to a length with units.
`\dim_gset:Nn`
`\dim_gset:cn`

Updated: 2011-10-22

`\dim_set_eq:NN`
`\dim_set_eq:(cN|Nc|cc)`
`\dim_gset_eq:NN`
`\dim_gset_eq:(cN|Nc|cc)`

`\dim_set_eq:NN <dimension1> <dimension2>`
Sets the content of `<dimension1>` equal to that of `<dimension2>`.

`\dim_sub:Nn` `\dim_sub:Nn <dimension> {<dimension expression>}`
`\dim_sub:cn` Subtracts the result of the `<dimension expression>` from the current content of the `<dimension>`.
`\dim_gsub:Nn`
`\dim_gsub:cn`

Updated: 2011-10-22

3 Utilities for dimension calculations

`\dim_abs:n` `\dim_abs:n {<dimexpr>}`
Updated: 2012-09-26

Converts the `<dimexpr>` to its absolute value, leaving the result in the input stream as a `<dimension denotation>`.

`\dim_max:nn` `\dim_max:nn {<dimexpr1>} {<dimexpr2>}`
`\dim_min:nn` `\dim_min:nn {<dimexpr1>} {<dimexpr2>}`

New: 2012-09-09
Updated: 2012-09-26

Evaluates the two `<dimension expressions>` and leaves either the maximum or minimum value in the input stream as appropriate, as a `<dimension denotation>`.

\dim_ratio:nn \star

Updated: 2011-10-22

\dim_ratio:nn { $\langle dimexpr_1 \rangle$ } { $\langle dimexpr_2 \rangle$ }

Parses the two $\langle dimension\ expressions \rangle$ and converts the ratio of the two to a form suitable for use inside a $\langle dimension\ expression \rangle$. This ratio is then left in the input stream, allowing syntax such as

```
\dim_set:Nn \l_my_dim
{ 10 pt * \dim_ratio:nn { 5 pt } { 10 pt } }
```

The output of \dim_ratio:nn on full expansion is a ratio expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Nx \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

displays 327680/655360 on the terminal.

4 Dimension expression conditionals

\dim_compare_p:nNn \star
\dim_compare:nNnTF \star

\dim_compare_p:nNn { $\langle dimexpr_1 \rangle$ } { $\langle relation \rangle$ } { $\langle dimexpr_2 \rangle$ }
\dim_compare:nNnTF
{ $\langle dimexpr_1 \rangle$ } { $\langle relation \rangle$ } { $\langle dimexpr_2 \rangle$ }
{ $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }

This function first evaluates each of the $\langle dimension\ expressions \rangle$ as described for \dim_eval:n. The two results are then compared using the $\langle relation \rangle$:

Equal	=
Greater than	>
Less than	<

This function is less flexible than \dim_compare:nTF but around 5 times faster.

```
\dim_compare_p:n *
\dim_compare:nTF *

```

Updated: 2013-01-13

```
\dim_compare_p:n
{
  ⟨dimexpr_1⟩ ⟨relation_1⟩
  ...
  ⟨dimexpr_N⟩ ⟨relation_N⟩
  ⟨dimexpr_{N+1}⟩
}
\dim_compare:nTF
{
  ⟨dimexpr_1⟩ ⟨relation_1⟩
  ...
  ⟨dimexpr_N⟩ ⟨relation_N⟩
  ⟨dimexpr_{N+1}⟩
}
{⟨true code⟩} {⟨false code⟩}
```

This function evaluates the $\langle \text{dimension expressions} \rangle$ as described for `\dim_eval:n` and compares consecutive result using the corresponding $\langle \text{relation} \rangle$, namely it compares $\langle \text{dimexpr}_1 \rangle$ and $\langle \text{dimexpr}_2 \rangle$ using the $\langle \text{relation}_1 \rangle$, then $\langle \text{dimexpr}_2 \rangle$ and $\langle \text{dimexpr}_3 \rangle$ using the $\langle \text{relation}_2 \rangle$, until finally comparing $\langle \text{dimexpr}_N \rangle$ and $\langle \text{dimexpr}_{N+1} \rangle$ using the $\langle \text{relation}_N \rangle$. The test yields `true` if all comparisons are `true`. Each $\langle \text{dimension expression} \rangle$ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other $\langle \text{dimension expression} \rangle$ is evaluated and no other comparison is performed. The $\langle \text{relations} \rangle$ can be any of the following:

Equal	= or ==
Greater than or equal to	\geq
Greater than	$>$
Less than or equal to	\leq
Less than	$<$
Not equal	\neq

This function is more flexible than `\dim_compare:nNnTF` but around 5 times slower.

```
\dim_case:nn   *
\dim_case:nnTF *

```

New: 2013-07-24

```
\dim_case:nnTF {\test dimension expression}
{
  {\dimexpr case_1} {\code case_1}
  {\dimexpr case_2} {\code case_2}
  ...
  {\dimexpr case_n} {\code case_n}
}
{\true code}
{\false code}
```

This function evaluates the *<test dimension expression>* and compares this in turn to each of the *(dimension expression cases)*. If the two are equal then the associated *<code>* is left in the input stream and other cases are discarded. If any of the cases are matched, the *<true code>* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *<false code>* is inserted. The function `\dim_case:nn`, which does nothing if there is no match, is also available. For example

```
\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
{ 2 \l_tmpa_dim }
{
  { 5 pt }      { Small }
  { 4 pt + 6 pt } { Medium }
  { - 10 pt }    { Negative }
}
{ No idea! }
```

leaves “Medium” in the input stream.

5 Dimension expression loops

```
\dim_do_until:nNnn ☆ \dim_do_until:nNnn {\dimexpr_1} {\relation} {\dimexpr_2} {\code}
```

Places the *<code>* in the input stream for TeX to process, and then evaluates the relationship between the two *(dimension expressions)* as described for `\dim_compare:nNnTF`. If the test is `false` then the *<code>* is inserted into the input stream again and a loop occurs until the *<relation>* is `true`.

```
\dim_do_while:nNnn ☆ \dim_do_while:nNnn {\dimexpr_1} {\relation} {\dimexpr_2} {\code}
```

Places the *<code>* in the input stream for TeX to process, and then evaluates the relationship between the two *(dimension expressions)* as described for `\dim_compare:nNnTF`. If the test is `true` then the *<code>* is inserted into the input stream again and a loop occurs until the *<relation>* is `false`.

```
\dim_until_do:nNnn ☆ \dim_until_do:nNnn {\dimexpr_1} {\relation} {\dimexpr_2} {\code}
```

Evaluates the relationship between the two *(dimension expressions)* as described for `\dim_compare:nNnTF`, and then places the *<code>* in the input stream if the *<relation>* is `false`. After the *<code>* has been processed by TeX the test is repeated, and a loop occurs until the test is `true`.

`\dim_while_do:nNnn` ☆

`\dim_while_do:nNnn {\langle dimexpr_1\rangle} {\langle relation\rangle} {\langle dimexpr_2\rangle} {\langle code\rangle}`

Evaluates the relationship between the two *dimension expressions* as described for `\dim_compare:nNnTF`, and then places the *code* in the input stream if the *relation* is `true`. After the *code* has been processed by TeX the test is repeated, and a loop occurs until the test is `false`.

`\dim_do_until:nn` ☆

Updated: 2013-01-13

`\dim_do_until:nn {\langle dimension relation\rangle} {\langle code\rangle}`

Places the *code* in the input stream for TeX to process, and then evaluates the *dimension relation* as described for `\dim_compare:nTF`. If the test is `false` then the *code* is inserted into the input stream again and a loop occurs until the *relation* is `true`.

`\dim_do_while:nn` ☆

Updated: 2013-01-13

`\dim_do_while:nn {\langle dimension relation\rangle} {\langle code\rangle}`

Places the *code* in the input stream for TeX to process, and then evaluates the *dimension relation* as described for `\dim_compare:nTF`. If the test is `true` then the *code* is inserted into the input stream again and a loop occurs until the *relation* is `false`.

`\dim_until_do:nn` ☆

Updated: 2013-01-13

`\dim_until_do:nn {\langle dimension relation\rangle} {\langle code\rangle}`

Evaluates the *dimension relation* as described for `\dim_compare:nTF`, and then places the *code* in the input stream if the *relation* is `false`. After the *code* has been processed by TeX the test is repeated, and a loop occurs until the test is `true`.

`\dim_while_do:nn` ☆

Updated: 2013-01-13

`\dim_while_do:nn {\langle dimension relation\rangle} {\langle code\rangle}`

Evaluates the *dimension relation* as described for `\dim_compare:nTF`, and then places the *code* in the input stream if the *relation* is `true`. After the *code* has been processed by TeX the test is repeated, and a loop occurs until the test is `false`.

6 Dimension step functions

`\dim_step_function:nnnN` ☆

New: 2018-02-18

`\dim_step_function:nnnN {\langle initial value\rangle} {\langle step\rangle} {\langle final value\rangle} {\langle function\rangle}`

This function first evaluates the *initial value*, *step* and *final value*, all of which should be dimension expressions. The *function* is then placed in front of each *value* from the *initial value* to the *final value* in turn (using *step* between each *value*). The *step* must be non-zero. If the *step* is positive, the loop stops when the *value* becomes larger than the *final value*. If the *step* is negative, the loop stops when the *value* becomes smaller than the *final value*. The *function* should absorb one argument.

`\dim_step_inline:nnnn`

New: 2018-02-18

`\dim_step_inline:nnnn {\langle initial value\rangle} {\langle step\rangle} {\langle final value\rangle} {\langle code\rangle}`

This function first evaluates the *initial value*, *step* and *final value*, all of which should be dimension expressions. Then for each *value* from the *initial value* to the *final value* in turn (using *step* between each *value*), the *code* is inserted into the input stream with #1 replaced by the current *value*. Thus the *code* should define a function of one argument (#1).

```
\dim_step_variable:nnnNn
```

New: 2018-02-18

```
\dim_step_variable:nnnNn
  {<initial value>} {<step>} {<final value>} {<tl var>} {<code>}
```

This function first evaluates the *<initial value>*, *<step>* and *<final value>*, all of which should be dimension expressions. Then for each *<value>* from the *<initial value>* to the *<final value>* in turn (using *<step>* between each *<value>*), the *<code>* is inserted into the input stream, with the *<tl var>* defined as the current *<value>*. Thus the *<code>* should make use of the *<tl var>*.

7 Using `dim` expressions and variables

```
\dim_eval:n *
```

Updated: 2011-10-22

```
\dim_eval:n {<dimension expression>}
```

Evaluates the *<dimension expression>*, expanding any dimensions and token list variables within the *<expression>* to their content (without requiring `\dim_use:N`/`\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a *<dimension denotation>* after two expansions. This is expressed in points (pt), and requires suitable termination if used in a TeX-style assignment as it is *not* an *<internal dimension>*.

```
\dim_sign:n *
```

New: 2018-11-03

```
\dim_sign:n {<dimexpr>}
```

Evaluates the *<dimexpr>* then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

```
\dim_use:N *
```

```
\dim_use:c *
```

```
\dim_use:N {<dimension>}
```

Recovers the content of a *<dimension>* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a *<dimension>* is required (such as in the argument of `\dim_eval:n`).

TEXhackers note: `\dim_use:N` is the TeX primitive `\the:` this is one of several L^AT_EX3 names for this primitive.

```
\dim_to_decimal:n *
```

New: 2014-07-15

```
\dim_to_decimal:n {<dimexpr>}
```

Evaluates the *<dimension expression>*, and leaves the result, expressed in points (pt) in the input stream, with *no units*. The result is rounded by TeX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal:n { 1bp }
```

leaves 1.00374 in the input stream, *i.e.* the magnitude of one “big point” when converted to (TeX) points.

\dim_to_decimal_in_bp:n *

New: 2014-07-15

\dim_to_decimal_in_bp:n {*dimexpr*}

Evaluates the *dimension expression*, and leaves the result, expressed in big points (*bp*) in the input stream, with *no units*. The result is rounded by TeX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

\dim_to_decimal_in_bp:n { 1pt }

leaves 0.99628 in the input stream, *i.e.* the magnitude of one (TeX) point when converted to big points.

\dim_to_decimal_in_sp:n *

New: 2015-05-18

\dim_to_decimal_in_sp:n {*dimexpr*}

Evaluates the *dimension expression*, and leaves the result, expressed in scaled points (*sp*) in the input stream, with *no units*. The result is necessarily an integer.

\dim_to_decimal_in_unit:nn * \dim_to_decimal_in_unit:nn {*dimexpr*₁} {*dimexpr*₂}

New: 2014-07-15

Evaluates the *dimension expressions*, and leaves the value of *dimexpr*₁, expressed in a unit given by *dimexpr*₂, in the input stream. The result is a decimal number, rounded by TeX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

\dim_to_decimal_in_unit:nn { 1bp } { 1mm }

leaves 0.35277 in the input stream, *i.e.* the magnitude of one big point when converted to millimetres.

Note that this function is not optimised for any particular output and as such may give different results to \dim_to_decimal_in_bp:n or \dim_to_decimal_in_sp:n. In particular, the latter is able to take a wider range of input values as it is not limited by the ability to calculate a ratio using ε-Tex primitives, which is required internally by \dim_to_decimal_in_unit:nn.

\dim_to_fp:n *

New: 2012-05-08

\dim_to_fp:n {*dimexpr*}

Expands to an internal floating point number equal to the value of the *dimexpr* in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, \dim_to_fp:n can be used to speed up parts of a computation where a low precision and a smaller range are acceptable.

8 Viewing dim variables

\dim_show:N\dim_show:N *dimension*

\dim_show:cDisplays the value of the *dimension* on the terminal.

<u>\dim_show:n</u>	<code>\dim_show:n {<i>dimension expression</i>}</code>
New: 2011-11-22	Displays the result of evaluating the <i>dimension expression</i> on the terminal.
Updated: 2015-08-07	
<u>\dim_log:N</u>	<code>\dim_log:N {<i>dimension</i>}</code>
<u>\dim_log:c</u>	Writes the value of the <i>dimension</i> in the log file.
New: 2014-08-22	
Updated: 2015-08-03	
<u>\dim_log:n</u>	<code>\dim_log:n {<i>dimension expression</i>}</code>
New: 2014-08-22	Writes the result of evaluating the <i>dimension expression</i> in the log file.
Updated: 2015-08-07	

9 Constant dimensions

<u>\c_max_dim</u>	The maximum value that can be stored as a dimension. This can also be used as a component of a skip.
<u>\c_zero_dim</u>	A zero length as a dimension. This can also be used as a component of a skip.

10 Scratch dimensions

<u>\l_tmpa_dim</u>	Scratch dimension for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<u>\g_tmpa_dim</u>	Scratch dimension for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

11 Creating and initialising skip variables

<u>\skip_new:N</u>	<code>\skip_new:N {<i>skip</i>}</code>
<u>\skip_new:c</u>	Creates a new <i>skip</i> or raises an error if the name is already taken. The declaration is global. The <i>skip</i> is initially equal to 0 pt.

\skip_const:Nn
\skip_const:cn
New: 2012-03-05

\skip_const:Nn *skip* {*skip expression*}

Creates a new constant *skip* or raises an error if the name is already taken. The value of the *skip* is set globally to the *skip expression*.

\skip_zero:N
\skip_zero:c
\skip_gzero:N
\skip_gzero:c

\skip_zero:N *skip*

Sets *skip* to 0 pt.

\skip_zero_new:N
\skip_zero_new:c
\skip_gzero_new:N
\skip_gzero_new:c

New: 2012-01-07

\skip_zero_new:N *skip*

Ensures that the *skip* exists globally by applying \skip_new:N if necessary, then applies \skip_(g)zero:N to leave the *skip* set to zero.

\skip_if_exist_p:N *
\skip_if_exist_p:c *
\skip_if_exist:NTF *
\skip_if_exist:cTF *

New: 2012-03-03

\skip_if_exist_p:N *skip*

\skip_if_exist:NTF *skip* {*true code*} {*false code*}

Tests whether the *skip* is currently defined. This does not check that the *skip* really is a skip variable.

12 Setting skip variables

\skip_add:Nn
\skip_add:cn
\skip_gadd:Nn
\skip_gadd:cn

Updated: 2011-10-22

\skip_add:Nn *skip* {*skip expression*}

Adds the result of the *skip expression* to the current content of the *skip*.

\skip_set:Nn
\skip_set:cn
\skip_gset:Nn
\skip_gset:cn

Updated: 2011-10-22

\skip_set:Nn *skip* {*skip expression*}

Sets *skip* to the value of *skip expression*, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).

\skip_set_eq:NN
\skip_set_eq:(cN|Nc|cc)
\skip_gset_eq:NN
\skip_gset_eq:(cN|Nc|cc)

\skip_set_eq:NN *skip*₁ *skip*₂

Sets the content of *skip*₁ equal to that of *skip*₂.

\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn

Updated: 2011-10-22

\skip_sub:Nn *skip* {*skip expression*}

Subtracts the result of the *skip expression* from the current content of the *skip*.

13 Skip expression conditionals

```
\skip_if_eq_p:nn *
\skip_if_eq:nnTF *
```

`\skip_if_eq_p:nn` {*skipexpr₁*} {*skipexpr₂*}
`\skip_if_eq:nnTF`
 {*skipexpr₁*} {*skipexpr₂*}
 {*true code*} {*false code*}

This function first evaluates each of the *skip expressions* as described for `\skip_eval:n`. The two results are then compared for exact equality, *i.e.* both the fixed and rubber components must be the same for the test to be true.

```
\skip_if_finite_p:n *
\skip_if_finite:nTF *
```

New: 2012-03-05

```
\skip_if_finite_p:n {skipexpr}
\skip_if_finite:nTF {skipexpr} {true code} {false code}
```

Evaluates the *skip expression* as described for `\skip_eval:n`, and then tests if all of its components are finite.

14 Using skip expressions and variables

```
\skip_eval:n *
```

Updated: 2011-10-22

```
\skip_eval:n {skip expression}
```

Evaluates the *skip expression*, expanding any skips and token list variables within the *expression* to their content (without requiring `\skip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a *glue denotation* after two expansions. This is expressed in points (pt), and requires suitable termination if used in a TeX-style assignment as it is *not* an *internal glue*.

```
\skip_use:N *
\skip_use:c *
```

```
\skip_use:N {skip}
```

Recovers the content of a *skip* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a *dimension* or *skip* is required (such as in the argument of `\skip_eval:n`).

TeXhackers note: `\skip_use:N` is the TeX primitive `\the:` this is one of several L^AT_EX3 names for this primitive.

15 Viewing skip variables

```
\skip_show:N
```

Updated: 2015-08-03

```
\skip_show:N {skip}
```

Displays the value of the *skip* on the terminal.

```
\skip_show:n
```

New: 2011-11-22

Updated: 2015-08-07

```
\skip_show:n {skip expression}
```

Displays the result of evaluating the *skip expression* on the terminal.

<code>\skip_log:N</code>	<code>\skip_log:N <skip></code>
<code>\skip_log:c</code>	Writes the value of the <code><skip></code> in the log file.

New: 2014-08-22
Updated: 2015-08-03

<code>\skip_log:n</code>	<code>\skip_log:n {<skip expression>}</code>
	Writes the result of evaluating the <code><skip expression></code> in the log file.

New: 2014-08-22
Updated: 2015-08-07

16 Constant skips

<code>\c_max_skip</code>	The maximum value that can be stored as a skip (equal to <code>\c_max_dim</code> in length), with no stretch nor shrink component.
--------------------------	--

Updated: 2012-11-02

<code>\c_zero_skip</code>	A zero length as a skip, with no stretch nor shrink component.
---------------------------	--

Updated: 2012-11-01

17 Scratch skips

<code>\l_tmpa_skip</code>	Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_skip</code>	

Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

<code>\g_tmpa_skip</code>	Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_skip</code>	

Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

18 Inserting skips into the output

<code>\skip_horizontal:N</code>	<code>\skip_horizontal:N <skip></code>
<code>\skip_horizontal:c</code>	<code>\skip_horizontal:n {<skipexpr>}</code>
<code>\skip_horizontal:n</code>	Inserts a horizontal <code><skip></code> into the current list. The argument can also be a <code><dim></code> .

Updated: 2011-10-22

`\skip_horizontal:N` is the TeX primitive `\hskip` renamed.

```
\skip_vertical:N  
\skip_vertical:c  
\skip_vertical:n
```

Updated: 2011-10-22

```
\skip_vertical:N <skip>  
\skip_vertical:n {<skipexpr>}
```

Inserts a vertical *<skip>* into the current list. The argument can also be a *(dim)*.

TeXhackers note: `\skip_vertical:N` is the TeX primitive `\vskip` renamed.

19 Creating and initialising `muskip` variables

```
\muskip_new:N  
\muskip_new:c
```

```
\muskip_new:N <muskip>
```

Creates a new *<muskip>* or raises an error if the name is already taken. The declaration is global. The *<muskip>* is initially equal to 0 mu.

```
\muskip_const:Nn  
\muskip_const:cn
```

New: 2012-03-05

```
\muskip_const:Nn <muskip> {<muskip expression>}
```

Creates a new constant *<muskip>* or raises an error if the name is already taken. The value of the *<muskip>* is set globally to the *<muskip expression>*.

```
\muskip_zero:N  
\muskip_zero:c  
\muskip_gzero:N  
\muskip_gzero:c
```

```
\skip_zero:N <muskip>
```

Sets *<muskip>* to 0 mu.

```
\muskip_zero_new:N  
\muskip_zero_new:c  
\muskip_gzero_new:N  
\muskip_gzero_new:c
```

New: 2012-01-07

```
\muskip_zero_new:N <muskip>
```

Ensures that the *<muskip>* exists globally by applying `\muskip_new:N` if necessary, then applies `\muskip_(g)zero:N` to leave the *<muskip>* set to zero.

```
\muskip_if_exist_p:N *  
\muskip_if_exist_p:c *  
\muskip_if_exist:NTF *  
\muskip_if_exist:cTF *
```

New: 2012-03-03

```
\muskip_if_exist_p:N <muskip>
```

```
\muskip_if_exist:NTF <muskip> {<true code>} {<false code>}
```

Tests whether the *<muskip>* is currently defined. This does not check that the *<muskip>* really is a muskip variable.

20 Setting `muskip` variables

```
\muskip_add:Nn  
\muskip_add:cn  
\muskip_gadd:Nn  
\muskip_gadd:cn
```

Updated: 2011-10-22

```
\muskip_add:Nn <muskip> {<muskip expression>}
```

Adds the result of the *<muskip expression>* to the current content of the *<muskip>*.

```
\muskip_set:Nn
\muskip_set:cn
\muskip_gset:Nn
\muskip_gset:cn
```

Updated: 2011-10-22

```
\muskip_set:Nn <muskip> {<muskip expression>}
```

Sets *<muskip>* to the value of *<muskip expression>*, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).

```
\muskip_set_eq:NN
\muskip_set_eq:(cN|Nc|cc)
\muskip_gset_eq:NN
\muskip_gset_eq:(cN|Nc|cc)
```

Sets the content of *<muskip₁>* equal to that of *<muskip₂>*.

```
\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn
```

Updated: 2011-10-22

```
\muskip_sub:Nn <muskip> {<muskip expression>}
```

Subtracts the result of the *<muskip expression>* from the current content of the *<muskip>*.

21 Using `muskip` expressions and variables

```
\muskip_eval:n *
```

Updated: 2011-10-22

```
\muskip_eval:n {<muskip expression>}
```

Evaluates the *<muskip expression>*, expanding any skips and token list variables within the *<expression>* to their content (without requiring `\muskip_use:N`/`\t1_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a *<muglue denotation>* after two expansions. This is expressed in mu, and requires suitable termination if used in a TeX-style assignment as it is *not* an *<internal muglue>*.

```
\muskip_use:N *
\muskip_use:c *
```

```
\muskip_use:N <muskip>
```

Recovering the content of a *<skip>* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a *<dimension>* is required (such as in the argument of `\muskip_eval:n`).

TeXhackers note: `\muskip_use:N` is the TeX primitive `\the:` this is one of several L^AT_EX3 names for this primitive.

22 Viewing `muskip` variables

```
\muskip_show:N
\muskip_show:c
```

Updated: 2015-08-03

```
\muskip_show:N <muskip>
```

Displays the value of the *<muskip>* on the terminal.

<u>\muskip_show:n</u>	<code>\muskip_show:n {<muskip expression>}</code>
New: 2011-11-22	Displays the result of evaluating the <i><muskip expression></i> on the terminal.
Updated: 2015-08-07	
<u>\muskip_log:N</u>	<code>\muskip_log:N <muskip></code>
<u>\muskip_log:c</u>	<code>\muskip_log:c</code>
New: 2014-08-22	Writes the value of the <i><muskip></i> in the log file.
Updated: 2015-08-03	
<u>\muskip_log:n</u>	<code>\muskip_log:n {<muskip expression>}</code>
New: 2014-08-22	Writes the result of evaluating the <i><muskip expression></i> in the log file.
Updated: 2015-08-07	

23 Constant muskips

<u>\c_max_muskip</u>	The maximum value that can be stored as a muskip, with no stretch nor shrink component.
<u>\c_zero_muskip</u>	A zero length as a muskip, with no stretch nor shrink component.

24 Scratch muskips

<u>\l_tmpa_muskip</u>	Scratch muskip for local assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<u>\g_tmpa_muskip</u>	Scratch muskip for global assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

25 Primitive conditional

<u>\if_dim:w *</u>	<code>\if_dim:w <dimen1> <relation> <dimen2> <true code> \else: <false> \fi:</code>
	Compare two dimensions. The <i><relation></i> is one of <i><</i> , <i>=</i> or <i>></i> with category code 12.

TEXhackers note: This is the T_EX primitive `\ifdim`.

Part XXI

The **I3keys** package

Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

```
\MyModuleSetup{  
    key-one = value one,  
    key-two = value two  
}
```

or

```
\MyModuleMacro[  
    key-one = value one,  
    key-two = value two  
]{argument}
```

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }  
{  
    key-one .code:n = code including parameter #1,  
    key-two .tl_set:N = \l_mymodule_store_tl  
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }  
{  
    key-one = value one,  
    key-two = value two  
}
```

At a document level, `\keys_set:nn` is used within a document function, for example

```
\DeclareDocumentCommand \MyModuleSetup { m }  
{ \keys_set:nn { mymodule } { #1 } }  
\DeclareDocumentCommand \MyModuleMacro { o m }  
{  
    \group_begin:  
        \keys_set:nn { mymodule } { #1 }  
        % Main code for \MyModuleMacro  
    \group_end:  
}
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As discussed in section 2, it is suggested that the character / is reserved for sub-division of keys into logical groups. Functions and variables are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
{
    \l_mymodule_tmp_tl .code:n = code
}
```

creates a key called `\l_mymodule_tmp_tl`, and not one called `key`.

1 Creating keys

`\keys_define:nn`

Updated: 2017-11-14

`\keys_define:nn {<module>} {<keyval list>}`

Parses the `<keyval list>` and defines the keys listed there for `<module>`. The `<module>` name is treated as a string. In practice the `<module>` should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The `<keyval list>` should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```
\keys_define:nn { mymodule }
{
    keyname .code:n = Some~code~using~#1,
    keyname .value_required:n = true
}
```

where the properties of the key begin from the `.` after the key name.

The various properties available take either no arguments at all, or require one or more arguments. This is indicated in the name of the property using an argument specification. In the following discussion, each property is illustrated attached to an arbitrary `<key>`, which when used may be supplied with a `<value>`. All key *definitions* are local.

Key properties are applied in the reading order and so the ordering is significant. Key properties which define “actions”, such as `.code:n`, `.tl_set:N`, etc., override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so they replace one another. However, properties covering non-exclusive behaviours may be given in any order. Thus for example the following definitions are equivalent.

```
\keys_define:nn { mymodule }
{
    keyname .code:n          = Some~code~using~#1,
    keyname .value_required:n = true
}
\keys_define:nn { mymodule }
{
```

```

keyname .value_required:n = true,
keyname .code:n           = Some~code~using~#1
}

```

Note that with the exception of the special `.undefine:` property, all key properties define the key within the current TeX scope.

```
.bool_set:N
.bool_set:c
.bool_gset:N
.bool_gset:c
```

Updated: 2013-07-08

```
.bool_set_inverse:N
.bool_set_inverse:c
.bool_gset_inverse:N
.bool_gset_inverse:c
```

New: 2011-08-28

Updated: 2013-07-08

```
.choice:
```

Sets `<key>` to act as a choice key. Each valid choice for `<key>` must then be created, as discussed in section 3.

```
.choices:mm
.choices:(Vn|on|xn)
```

New: 2011-08-21

Updated: 2013-07-10

```
.clist_set:N
.clist_set:c
.clist_gset:N
.clist_gset:c
```

New: 2011-09-11

```
.code:n
```

Updated: 2013-07-10

```
.cs_set:Np
.cs_set:cp
.cs_set_protected:Np
.cs_set_protected:cp
.cs_gset:Np
.cs_gset:cp
.cs_gset_protected:Np
.cs_gset_protected:cp
```

New: 2020-01-11

`<key> .bool_set:N = <boolean>`

Defines `<key>` to set `<boolean>` to `<value>` (which must be either `true` or `false`). If the variable does not exist, it will be created globally at the point that the key is set up.

`<key> .bool_set_inverse:N = <boolean>`

Defines `<key>` to set `<boolean>` to the logical inverse of `<value>` (which must be either `true` or `false`). If the `<boolean>` does not exist, it will be created globally at the point that the key is set up.

`<key> .choice:`

Sets `<key>` to act as a choice key. Each valid choice for `<key>` must then be created, as discussed in section 3.

`<key> .choices:mm = {<choices>} {<code>}`

Sets `<key>` to act as a choice key, and defines a series `<choices>` which are implemented using the `<code>`. Inside `<code>`, `\l_keys_choice_t1` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of `<choices>` (indexed from 1). Choices are discussed in detail in section 3.

`<key> .clist_set:N = <comma list variable>`

Defines `<key>` to set `<comma list variable>` to `<value>`. Spaces around commas and empty items will be stripped. If the variable does not exist, it is created globally at the point that the key is set up.

`<key> .code:n = {<code>}`

Stores the `<code>` for execution when `<key>` is used. The `<code>` can include one parameter (#1), which will be the `<value>` given for the `<key>`.

`<key> .cs_set:Np = <control sequence> <arg. spec.>`

Defines `<key>` to set `<control sequence>` to have `<arg. spec.>` and replacement text `<value>`.

.default:n
.default:(v|o|x)
Updated: 2013-07-09

`<key> .default:n = {<default>}`

Creates a `<default>` value for `<key>`, which is used if no value is given. This will be used if only the key name is given, but not if a blank `<value>` is given:

```
\keys_define:nn { mymodule }
{
    key .code:n      = Hello~#1,
    key .default:n = World
}
\keys_set:nn { mymodule }
{
    key = Fred, % Prints 'Hello Fred'
    key,          % Prints 'Hello World'
    key = ,       % Prints 'Hello '
}
```

The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value does not trigger an error.

.dim_set:N
.dim_set:c
.dim_gset:N
.dim_gset:c
Updated: 2020-01-17

`<key> .dim_set:N = <dimension>`

Defines `<key>` to set `<dimension>` to `<value>` (which must a dimension expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.fp_set:N
.fp_set:c
.fp_gset:N
.fp_gset:c
Updated: 2020-01-17

`<key> .fp_set:N = <floating point>`

Defines `<key>` to set `<floating point>` to `<value>` (which must a floating point expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.groups:n
New: 2013-07-14

`<key> .groups:n = {<groups>}`

Defines `<key>` as belonging to the `<groups>` declared. Groups provide a “secondary axis” for selectively setting keys, and are described in Section 6.

.inherit:n
New: 2016-11-22

`<key> .inherit:n = {<parents>}`

Specifies that the `<key>` path should inherit the keys listed as `<parents>`. For example, after setting

```
\keys_define:nn { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:nn { } { bar .inherit:n = foo }
```

setting

```
\keys_set:nn { bar } { test = a }
```

will be equivalent to

```
\keys_set:nn { foo } { test = a }
```

.initial:n
.initial:(V|o|x)
Updated: 2013-07-09

`<key> .initial:n = {<value>}`

Initialises the `<key>` with the `<value>`, equivalent to

`\keys_set:nn {<module>} { <key> = <value> }`

.int_set:N
.int_set:c
.int_gset:N
.int_gset:c
Updated: 2020-01-17

`<key> .int_set:N = <integer>`

Defines `<key>` to set `<integer>` to `<value>` (which must be an integer expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.meta:n
Updated: 2013-07-10

`<key> .meta:n = {<keyval list>}`

Makes `<key>` a meta-key, which will set `<keyval list>` in one go. The `<keyval list>` can refer as #1 to the value given at the time the `<key>` is used (or, if no value is given, the `<key>`'s default value).

.meta:nn
New: 2013-07-10

`<key> .meta:nn = {<path>} {<keyval list>}`

Makes `<key>` a meta-key, which will set `<keyval list>` in one go using the `<path>` in place of the current one. The `<keyval list>` can refer as #1 to the value given at the time the `<key>` is used (or, if no value is given, the `<key>`'s default value).

.multichoice:
New: 2011-08-21

`<key> .multichoice:`

Sets `<key>` to act as a multiple choice key. Each valid choice for `<key>` must then be created, as discussed in section 3.

.multichoice:nn
.multichoice:(Vn|on|xn)
New: 2011-08-21
Updated: 2013-07-10

`<key> .multichoice:nn {<choices>} {<code>}`

Sets `<key>` to act as a multiple choice key, and defines a series `<choices>` which are implemented using the `<code>`. Inside `<code>`, `\l_keys_choice_t1` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of `<choices>` (indexed from 1). Choices are discussed in detail in section 3.

.muskip_set:N
.muskip_set:c
.muskip_gset:N
.muskip_gset:c
New: 2019-05-05
Updated: 2020-01-17

`<key> .muskip_set:N = <muskip>`

Defines `<key>` to set `<muskip>` to `<value>` (which must be a muskip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.prop_put:N
.prop_put:c
.prop_gput:N
.prop_gput:c
New: 2019-01-31

`<key> .prop_put:N = <property list>`

Defines `<key>` to put the `<value>` onto the `<property list>` stored under the `<key>`. If the variable does not exist, it is created globally at the point that the key is set up.

```
.skip_set:N  
.skip_set:c  
.skip_gset:N  
.skip_gset:c
```

Updated: 2020-01-17

`<key> .skip_set:N = <skip>`

Defines `<key>` to set `<skip>` to `<value>` (which must be a skip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

```
.tl_set:N  
.tl_set:c  
.tl_gset:N  
.tl_gset:c
```

`<key> .tl_set:N = <token list variable>`

Defines `<key>` to set `<token list variable>` to `<value>`. If the variable does not exist, it is created globally at the point that the key is set up.

```
.tl_set_x:N  
.tl_set_x:c  
.tl_gset_x:N  
.tl_gset_x:c
```

`<key> .tl_set_x:N = <token list variable>`

Defines `<key>` to set `<token list variable>` to `<value>`, which will be subjected to an x-type expansion (*i.e.* using `\tl_set:Nx`). If the variable does not exist, it is created globally at the point that the key is set up.

```
.undefined:
```

New: 2015-07-14

`<key> .undefined:`

Removes the definition of the `<key>` within the current scope.

```
.value_forbidden:n  
New: 2015-07-14
```

`<key> .value_forbidden:n = true|false`

Specifies that `<key>` cannot receive a `<value>` when used. If a `<value>` is given then an error will be issued. Setting the property `false` cancels the restriction.

```
.value_required:n  
New: 2015-07-14
```

`<key> .value_required:n = true|false`

Specifies that `<key>` must receive a `<value>` when used. If a `<value>` is not given then an error will be issued. Setting the property `false` cancels the restriction.

2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several sub-groups for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { mymodule / subgroup }  
  { key .code:n = code }
```

or to the key name:

```
\keys_define:nn { mymodule }  
  { subgroup / key .code:n = code }
```

As illustrated, the best choice of token for sub-dividing keys in this way is `/`. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name `mymodule/subgroup/key`.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

3 Choice and multiple choice keys

The l3keys system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. “Multiple” choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the `.choice:` property:

```
\keys_define:nn { mymodule }
  { key .choice: }
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependant only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the `.choices:nn` property.

```
\keys_define:nn { mymodule }
{
  key .choices:nn =
    { choice-a, choice-b, choice-c }
  {
    You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
    which~is~in~position~\int_use:N \l_keys_choice_int \c_space_tl
    in~the~list.
  }
}
```

The index `\l_keys_choice_int` in the list of choices starts at 1.

\l_keys_choice_int \l_keys_choice_tl

Inside the code block for a choice generated using `.choices:nn`, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1. Note that, as with standard key code generated using `.code:n`, the value passed to the key (i.e. the choice name) is also available as `#1`.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

```
\keys_define:nn { mymodule }
{
  key .choice:,,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}
```

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_tl` or `\l_keys_choice_int`. These variables do not have defined

behaviour when used outside of code created using `.choices:nn` (*i.e.* anything might happen).

It is possible to allow choice keys to take values which have not previously been defined by adding code for the special `unknown` choice. The general behavior of the `unknown` key is described in Section 5. A typical example in the case of a choice would be to issue a custom error message:

```
\keys_define:nn { mymodule }
{
    key .choice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
    key / unknown .code:n =
        \msg_error:nnxxxx { mymodule } { unknown-choice }
            { key } % Name of choice key
            { choice-a , choice-b , choice-c } % Valid choices
            { \exp_not:n {#1} } % Invalid choice given
    %
    %
}
```

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoices:nn`. As with mutually exclusive choices, multiple choices are define as sub-keys. Thus both

```
\keys_define:nn { mymodule }
{
    key .multichoices:nn =
        { choice-a, choice-b, choice-c }
    {
        You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
        which~is~in~position~
        \int_use:N \l_keys_choice_int \c_space_tl
        in~the~list.
    }
}
```

and

```
\keys_define:nn { mymodule }
{
    key .multichoice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
}
```

are valid.

When a multiple choice key is set

```
\keys_set:nn { mymodule }
{
    key = { a , b , c } % 'key' defined as a multiple choice
}
```

each choice is applied in turn, equivalent to a `clist` mapping or to applying each value individually:

```
\keys_set:nn { mymodule }
{
    key = a ,
    key = b ,
    key = c ,
}
```

Thus each separate choice will have passed to it the `\l_keys_choice_tl` and `\l_keys_choice_int` in exactly the same way as described for `.choices:nn`.

4 Setting keys

`\keys_set:nn`
`\keys_set:(nV|nv|no)`

Updated: 2017-11-14

`\keys_set:nn {<module>} {<keyval list>}`

Parses the `<keyval list>`, and sets those keys which are defined for `<module>`. The behaviour on finding an unknown key can be set by defining a special `unknown` key: this is illustrated later.

`\l_keys_key_str`
`\l_keys_path_str`
`\l_keys_value_tl`

Updated: 2020-02-08

For each key processed, information of the full *path* of the key, the *name* of the key and the *value* of the key is available within three token list variables. These may be used within the code of the key.

The *value* is everything after the `=`, which may be empty if no value was given. This is stored in `\l_keys_value_tl`, and is not processed in any way by `\keys_set:nn`.

The *path* of the key is a “full” description of the key, and is unique for each key. It consists of the module and full key name, thus for example

```
\keys_set:nn { mymodule } { key-a = some-value }
```

has path `mymodule/key-a` while

```
\keys_set:nn { mymodule } { subset / key-a = some-value }
```

has path `mymodule/subset/key-a`. This information is stored in `\l_keys_path_str`.

The *name* of the key is the part of the path after the last `/`, and thus is not unique. In the preceding examples, both keys have name `key-a` despite having different paths. This information is stored in `\l_keys_key_str`.

5 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` looks for a special `unknown` key for the same module, and if this is not defined raises an error indicating that

the key name was unknown. This mechanism can be used for example to issue custom error texts.

```
\keys_define:nn { mymodule }
{
    unknown .code:n =
        You~tried~to~set~key~'\l_keys_key_str'~to~'#1'.
}
```

<code>\keys_set_known:nn</code> <code>\keys_set_known:(nV nv no)</code> <code>\keys_set_known:nnN</code> <code>\keys_set_known:(nVN nvN noN)</code> <code>\keys_set_known:nnnN</code> <code>\keys_set_known:(nVnN nvN nonN)</code>	<code>\keys_set_known:nn {\langle module \rangle} {\langle keyval list \rangle}</code> <code>\keys_set_known:nnN {\langle module \rangle} {\langle keyval list \rangle} {\langle tl \rangle}</code> <code>\keys_set_known:nnnN {\langle module \rangle} {\langle keyval list \rangle} {\langle root \rangle} {\langle tl \rangle}</code>
---	--

New: 2011-08-23
Updated: 2019-01-29

These functions set keys which are known for the *<module>*, and simply ignore other keys. The `\keys_set_known:nn` function parses the *<keyval list>*, and sets those keys which are defined for *<module>*. Any keys which are unknown are not processed further by the parser. In addition, `\keys_set_known:nnN` stores the key–value pairs in the *<tl>* in comma-separated form (*i.e.* an edited version of the *<keyval list>*). When a *<root>* is given (`\keys_set_known:nnnN`), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

6 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using

```
\keys_define:nn { mymodule }
{
    key-one   .code:n = { \my_func:n {#1} } ,
    key-two   .tl_set:N = \l_my_a_tl          ,
    key-three .tl_set:N = \l_my_b_tl          ,
    key-four  .fp_set:N = \l_my_a_fp          ,
}
```

the use of `\keys_set:nn` attempts to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to *groups*: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```
\keys_define:nn { mymodule }
{
    key-one   .code:n = { \my_func:n {#1} } ,
    key-one   .groups:n = { first }          ,
    key-two   .tl_set:N = \l_my_a_tl          ,
```

```

key-two .groups:n = { first }
key-three .tl_set:N = \l_my_b_t1
key-three .groups:n = { second }
key-four .fp_set:N = \l_my_a_fp
}

```

assigns `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made “active”, or by marking one or more groups to be ignored in key setting.

<code>\keys_set_filter:nnn</code> <code>\keys_set_filter:(nnV nnv nno)</code> <code>\keys_set_filter:nnnN</code> <code>\keys_set_filter:(nnVN nnvN nnoN)</code> <code>\keys_set_filter:nnnnN</code> <code>\keys_set_filter:(nnVnN nnvnN nnonN)</code>	<code>\keys_set_filter:nnn {\langle module \rangle} {\langle groups \rangle} {\langle keyval list \rangle}</code> <code>\keys_set_filter:nnnN {\langle module \rangle} {\langle groups \rangle} {\langle keyval list \rangle} {\langle tl \rangle}</code> <code>\keys_set_filter:nnnnN {\langle module \rangle} {\langle groups \rangle} {\langle keyval list \rangle} {\langle root \rangle}</code> <code>{t1}</code>
--	---

New: 2013-07-14

Updated: 2019-01-29

Activates key filtering in an “opt-out” sense: keys assigned to any of the `\langle groups \rangle` specified are ignored. The `\langle groups \rangle` are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set. The key-value pairs for each key which is filtered out are stored in the `\langle tl \rangle` in a comma-separated form (*i.e.* an edited version of the `\langle keyval list \rangle`). The `\keys_set_filter:nnn` version skips this stage.

Use of `\keys_set_filter:nnnN` can be nested, with the correct residual `\langle keyval list \rangle` returned at each stage. In the version which takes a `\langle root \rangle` argument, the key list is returned relative to that point in the key tree. In the cases without a `\langle root \rangle` argument, only the key names and values are returned.

<code>\keys_set_groups:nnn</code> <code>\keys_set_groups:(nnV nnv nno)</code>	<code>\keys_set_groups:nnn {\langle module \rangle} {\langle groups \rangle} {\langle keyval list \rangle}</code>
--	---

New: 2013-07-14

Updated: 2017-05-27

Activates key filtering in an “opt-in” sense: only keys assigned to one or more of the `\langle groups \rangle` specified are set. The `\langle groups \rangle` are given as a comma-separated list. Unknown keys are not assigned to any group and are thus never set.

7 Utility functions for keys

<code>\keys_if_exist_p:nn *</code> <code>\keys_if_exist:nnTF *</code>	<code>\keys_if_exist_p:nn {\langle module \rangle} {\langle key \rangle}</code> <code>\keys_if_exist:nnTF {\langle module \rangle} {\langle key \rangle} {\langle true code \rangle} {\langle false code \rangle}</code>
--	---

Tests if the `\langle key \rangle` exists for `\langle module \rangle`, *i.e.* if any code has been defined for `\langle key \rangle`.

<pre>\keys_if_choice_exist_p:nnn * \keys_if_choice_exist:nnnTF *</pre>	<pre>\keys_if_choice_exist_p:nnn {\module} {\key} {\choice} \keys_if_choice_exist:nnnTF {\module} {\key} {\choice} {\true code} {\false code}</pre>
--	---

New: 2011-08-21
Updated: 2017-11-14

Tests if the *<choice>* is defined for the *<key>* within the *<module>*, *i.e.* if any code has been defined for *<key>/<choice>*. The test is **false** if the *<key>* itself is not defined.

<pre>\keys_show:nn</pre>	<pre>\keys_show:nn {\module} {\key}</pre>
--------------------------	---

Displays in the terminal the information associated to the *<key>* for a *<module>*, including the function which is used to actually implement it.

<pre>\keys_log:nn</pre>	<pre>\keys_log:nn {\module} {\key}</pre>
-------------------------	--

Writes in the log file the information associated to the *<key>* for a *<module>*. See also `\keys_show:nn` which displays the result in the terminal.

8 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in special circumstances you may wish to use a lower-level approach. The low-level parsing system converts a *<key-value list>* into *<keys>* and associated *<values>*. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double # tokens or expand any input. Active tokens = and , appearing at the outer level of braces are converted to category “other” (12) so that the parser does not “miss” any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces have exactly one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

\keyval_parse:NNn *

Updated: 2020-02-20

\keyval_parse:NNn <function₁> <function₂> {<key-value list>}

Parses the <key-value list> into a series of <keys> and associated <values>, or keys alone (if no <value> was given). <function₁> should take one argument, while <function₂> should absorb two arguments. After \keyval_parse:NNn has parsed the <key-value list>, <function₁> is used to process keys given with no value and <function₂> is used to process keys given with a value. The order of the <keys> in the <key-value list> is preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn  
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\function:nn { key1 } { value1 }  
\function:nn { key2 } { value2 }  
\function:nn { key3 } { }  
\function:n { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the <key> and <value>, then one *outer* set of braces is removed from the <key> and <value> as part of the processing.

TeXhackers note: The result is returned within \exp_not:n, which means that the converted input stream does not expand further when appearing in an x-type or e-type argument expansion.

Part XXII

The **l3intarray** package: fast global integer arrays

1 **l3intarray** documentation

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast **l3seq**, where access time is linear). These arrays have several important features

- The size of the array is fixed and must be given at point of initialisation
- The absolute value of each entry has maximum $2^{30} - 1$ (*i.e.* one power lower than the usual `\c_max_int` ceiling of $2^{31} - 1$)

The use of **intarray** data is therefore recommended for cases where the need for fast access is of paramount importance.

```
\intarray_new:Nn
\intarray_new:cN
```

New: 2018-03-29

```
\intarray_new:Nn <intarray var> {<size>}
```

Evaluates the integer expression `<size>` and allocates an *<integer array variable>* with that number of (zero) entries. The variable name should start with `\g_` because assignments are always global.

```
\intarray_count:N *
\intarray_count:c *
```

New: 2018-03-29

```
\intarray_count:N <intarray var>
```

Expands to the number of entries in the *<integer array variable>*. Contrarily to `\seq_count:N` this is performed in constant time.

```
\intarray_gset:Nnn
\intarray_gset:cnn
```

New: 2018-03-29

```
\intarray_gset:Nnn <intarray var> {<position>} {<value>}
```

Stores the result of evaluating the integer expression `<value>` into the *<integer array variable>* at the (integer expression) `<position>`. If the `<position>` is not between 1 and the `\intarray_count:N`, or the `<value>`'s absolute value is bigger than $2^{30} - 1$, an error occurs. Assignments are always global.

```
\intarray_const_from_clist:Nn
\intarray_const_from_clist:cN
```

New: 2018-05-04

```
\intarray_const_from_clist:Nn <intarray var> <intexpr list>
```

Creates a new constant *<integer array variable>* or raises an error if the name is already taken. The *<integer array variable>* is set (globally) to contain as its items the results of evaluating each *<integer expression>* in the *<comma list>*.

```
\intarray_gzero:N
\intarray_gzero:c
```

New: 2018-05-04

```
\intarray_gzero:N <intarray var>
```

Sets all entries of the *<integer array variable>* to zero. Assignments are always global.

```
\intarray_item:Nn *
\intarray_item:cn *
```

New: 2018-03-29

```
\intarray_item:Nn <intarray var> {<position>}
```

Expands to the integer entry stored at the (integer expression) *<position>* in the *<integer array variable>*. If the *<position>* is not between 1 and the `\intarray_count:N`, an error occurs.

```
\intarray_rand_item:N *
\intarray_rand_item:c *
```

New: 2018-05-05

```
\intarray_rand_item:N <intarray var>
```

Selects a pseudo-random item of the *<integer array>*. If the *<integer array>* is empty, produce an error.

```
\intarray_show:N
\intarray_show:c
\intarray_log:N
\intarray_log:c
```

New: 2018-05-04

```
\intarray_show:N <intarray var>
\intarray_log:N <intarray var>
```

Displays the items in the *<integer array variable>* in the terminal or writes them in the log file.

1.1 Implementation notes

It is a wrapper around the `\fontdimen` primitive, used to store arrays of integers (with a restricted range: absolute value at most $2^{30} - 1$). In contrast to `\seq` sequences the access to individual entries is done in constant time rather than linear time, but only integers can be stored. More precisely, the primitive `\fontdimen` stores dimensions but the `\intarray` package transparently converts these from/to integers. Assignments are always global.

While LuaTeX's memory is extensible, other engines can "only" deal with a bit less than 4×10^6 entries in all `\fontdimen` arrays combined (with default TeXLive settings).

Part XXIII

The **I3fp** package: Floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x / y , square root \sqrt{x} , and parentheses.
- Comparison operators: $x < y$, $x \leq y$, $x >? y$, $x != y$ etc.
- Boolean logic: sign $\text{sign } x$, negation $\text{!} x$, conjunction $x \&& y$, disjunction $x \parallel y$, ternary operator $x ? y : z$.
- Exponentials: $\exp x$, $\ln x$, x^y , $\log_b x$.
- Integer factorial: $\text{fact } x$.
- Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and $\text{sind } x$, $\text{cosd } x$, $\text{tand } x$, $\text{cotd } x$, $\text{seed } x$, $\text{cscd } x$ expecting their arguments in degrees.
- Inverse trigonometric functions: $\text{asin } x$, $\text{acos } x$, $\text{atan } x$, $\text{acot } x$, $\text{asec } x$, $\text{acsc } x$ giving a result in radians, and $\text{asind } x$, $\text{acosd } x$, $\text{atand } x$, $\text{cotd } x$, $\text{asecd } x$, $\text{acsed } x$ giving a result in degrees.

(not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\text{sech } x$, $\text{csch } x$, and $\text{asinh } x$, $\text{acosh } x$, $\text{atanh } x$, $\text{acoth } x$, $\text{asech } x$, $\text{acsch } x$.

- Extrema: $\max(x_1, x_2, \dots)$, $\min(x_1, x_2, \dots)$, $\text{abs}(x)$.
- Rounding functions, controlled by two optional values, n (number of places, 0 by default) and t (behavior on a tie, NaN by default):
 - $\text{trunc}(x, n)$ rounds towards zero,
 - $\text{floor}(x, n)$ rounds towards $-\infty$,
 - $\text{ceil}(x, n)$ rounds towards $+\infty$,
 - $\text{round}(x, n, t)$ rounds to the closest value, with ties rounded to an even value by default, towards zero if $t = 0$, towards $+\infty$ if $t > 0$ and towards $-\infty$ if $t < 0$.

And (not yet) modulo, and “quantize”.

- Random numbers: $\text{rand}()$, $\text{randint}(m, n)$.
- Constants: pi , deg (one degree in radians).
- Dimensions, automatically expressed in points, e.g., pc is 12.

- Automatic conversion (no need for $\langle type \rangle_use:N$) of integer, dimension, and skip variables to floating point numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.
- Tuples: (x_1, \dots, x_n) that can be stored in variables, added together, multiplied or divided by a floating point number, and nested.

Floating point numbers can be given either explicitly (in a form such as $1.234e-34$, or -0.0001), or as a stored floating point variable, which is automatically replaced by its current value. A “floating point” is a floating point number or a tuple thereof. See section 9.1 for a description of what a floating point is, section 9.2 for details about how an expression is parsed, and section 9.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 7.

An example of use could be the following.

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
= \ExplSyntaxOn \fp_to_decimal:n {sin(3.5)/2 + 2e-3} $.
```

The operation `round` can be used to limit the result’s precision. Adding `+0` avoids the possibly undesirable output `-0`, replacing it by `+0`. However, the `l3fp` module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

```
\documentclass{article}
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand{\calcnum}{m}
  { \num{ \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnum{2 pi * sin ( 2.3 ^ 5 ) }
\end{document}
```

See the documentation of `siunitx` for various options of `\num`.

1 Creating and initialising floating point variables

`\fp_new:N`

`\fp_new:c`

Updated: 2012-05-08

`\fp_new:N <fp var>`

Creates a new $\langle fp \ var \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle fp \ var \rangle$ is initially `+0`.

`\fp_const:Nn`

`\fp_const:cn`

Updated: 2012-05-08

`\fp_const:Nn <fp var> {<floating point expression>}`

Creates a new constant $\langle fp \ var \rangle$ or raises an error if the name is already taken. The $\langle fp \ var \rangle$ is set globally equal to the result of evaluating the $\langle floating \ point \ expression \rangle$.

`\fp_zero:N`

`\fp_zero:c`

`\fp_gzero:N`

`\fp_gzero:c`

Updated: 2012-05-08

`\fp_zero:N <fp var>`

Sets the $\langle fp \ var \rangle$ to `+0`.

```
\fp_zero_new:N  
\fp_zero_new:c  
\fp_gzero_new:N  
\fp_gzero_new:c
```

Updated: 2012-05-08

```
\fp_zero_new:N <fp var>
```

Ensures that the $\langle fp \ var \rangle$ exists globally by applying `\fp_new:N` if necessary, then applies `\fp_(g)zero:N` to leave the $\langle fp \ var \rangle$ set to +0.

Updated: 2012-05-08

2 Setting floating point variables

```
\fp_set:Nn  
\fp_set:cn  
\fp_gset:Nn  
\fp_gset:cn
```

Updated: 2012-05-08

```
\fp_set:Nn <fp var> {floating point expression}
```

Sets $\langle fp \ var \rangle$ equal to the result of computing the $\langle floating \ point \ expression \rangle$.

```
\fp_set_eq>NN  
\fp_set_eq:(cN|Nc|cc)  
\fp_gset_eq>NN  
\fp_gset_eq:(cN|Nc|cc)
```

Updated: 2012-05-08

```
\fp_set_eq>NN <fp var1> <fp var2>
```

Sets the floating point variable $\langle fp \ var_1 \rangle$ equal to the current value of $\langle fp \ var_2 \rangle$.

```
\fp_add:Nn  
\fp_add:cn  
\fp_gadd:Nn  
\fp_gadd:cn
```

Updated: 2012-05-08

```
\fp_add:Nn <fp var> {floating point expression}
```

Adds the result of computing the $\langle floating \ point \ expression \rangle$ to the $\langle fp \ var \rangle$. This also applies if $\langle fp \ var \rangle$ and $\langle floating \ point \ expression \rangle$ evaluate to tuples of the same size.

```
\fp_sub:Nn  
\fp_sub:cn  
\fp_gsub:Nn  
\fp_gsub:cn
```

Updated: 2012-05-08

```
\fp_sub:Nn <fp var> {floating point expression}
```

Subtracts the result of computing the $\langle floating \ point \ expression \rangle$ from the $\langle fp \ var \rangle$. This also applies if $\langle fp \ var \rangle$ and $\langle floating \ point \ expression \rangle$ evaluate to tuples of the same size.

3 Using floating points

```
\fp_eval:n  *
```

New: 2012-05-08
Updated: 2012-07-08

```
\fp_eval:n {floating point expression}
```

Evaluates the $\langle floating \ point \ expression \rangle$ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and `NaN` trigger an “invalid operation” exception. For a tuple, each item is converted using `\fp_eval:n` and they are combined as $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots, \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items. This function is identical to `\fp_to_decimal:n`.

\fp_sign:n *

New: 2018-11-03

`\fp_sign:n {\fexpr}`

Evaluates the $\langle fexpr \rangle$ and leaves its sign in the input stream using `\fp_eval:n {sign(\result)}`: +1 for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ±0 for ± 0 . If the operand is a tuple or is NaN, then “invalid operation” occurs and the result is 0.

\fp_to_decimal:N *
\fp_to_decimal:c *
\fp_to_decimal:n *

New: 2012-05-08
Updated: 2012-07-08

`\fp_to_decimal:N {fp var}`
`\fp_to_decimal:n {\floating point expression}`

Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception. For a tuple, each item is converted using `\fp_to_decimal:n` and they are combined as $(\langle fp_1 \rangle, \ldots, \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle)$ or $()$ for fewer items.

\fp_to_dim:N *
\fp_to_dim:c *
\fp_to_dim:n *

Updated: 2016-03-22

`\fp_to_dim:N {fp var}`
`\fp_to_dim:n {\floating point expression}`

Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to `\fp_to_decimal:n`, with an additional trailing pt (both letter tokens). In particular, the result may be outside the range $[-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]$ of valid TeX dimensions, leading to overflow errors if used as a dimension. Tuples, as well as the values $\pm\infty$ and NaN, trigger an “invalid operation” exception.

\fp_to_int:N *
\fp_to_int:c *
\fp_to_int:n *

Updated: 2012-07-08

`\fp_to_int:N {fp var}`
`\fp_to_int:n {\floating point expression}`

Evaluates the $\langle floating\ point\ expression \rangle$, and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range $[-2^{31} + 1, 2^{31} - 1]$ of valid TeX integers, leading to overflow errors if used in an integer expression. Tuples, as well as the values $\pm\infty$ and NaN, trigger an “invalid operation” exception.

\fp_to_scientific:N *
\fp_to_scientific:c *
\fp_to_scientific:n *

New: 2012-05-08
Updated: 2016-03-22

`\fp_to_scientific:N {fp var}`
`\fp_to_scientific:n {\floating point expression}`

Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result in scientific notation:

$\langle optional\ -\rangle \langle digit \rangle . \langle 15\ digits \rangle e \langle optional\ sign \rangle \langle exponent \rangle$

The leading $\langle digit \rangle$ is non-zero except in the case of ± 0 . The values $\pm\infty$ and NaN trigger an “invalid operation” exception. Normal category codes apply: thus the e is category code 11 (a letter). For a tuple, each item is converted using `\fp_to_scientific:n` and they are combined as $(\langle fp_1 \rangle, \ldots, \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle)$ or $()$ for fewer items.

```
\fp_to_t1:N  * \fp_to_t1:N <fp var>
\fp_to_t1:c  * \fp_to_t1:n {<floating point expression>}
```

Updated: 2016-03-22

Evaluates the $\langle\text{floating point expression}\rangle$ and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from $\text{\fp_to_scientific:n}$). Numbers in the range $[10^{-3}, 10^{16}]$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see \fp_to_decimal:n). Negative numbers start with $-$. The special values ± 0 , $\pm\infty$ and NaN are rendered as 0 , -0 , inf , $-\text{inf}$, and nan respectively. Normal category codes apply and thus inf or nan , if produced, are made up of letters. For a tuple, each item is converted using \fp_to_t1:n and they are combined as $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots, \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items.

```
\fp_use:N  * \fp_use:N <fp var>
\fp_use:c  * \fp_use:c <fp var>
```

Updated: 2012-07-08

Inserts the value of the $\langle\text{fp var}\rangle$ into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception. For a tuple, each item is converted using \fp_to_decimal:n and they are combined as $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots, \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items. This function is identical to \fp_to_decimal:N .

4 Floating point conditionals

```
\fp_if_exist_p:N  * \fp_if_exist_p:N <fp var>
\fp_if_exist_p:c  * \fp_if_exist:NTF <fp var> {{true code}} {{false code}}
```

Tests whether the $\langle\text{fp var}\rangle$ is currently defined. This does not check that the $\langle\text{fp var}\rangle$ really is a floating point variable.

Updated: 2012-05-08

```
\fp_compare_p:nNn  * \fp_compare_p:nNn <fpexpr_1> <relation> <fpexpr_2>
\fp_compare:nNnTF  * \fp_compare:nNnTF <fpexpr_1> <relation> <fpexpr_2> {{true code}} {{false code}}
```

Compares the $\langle\text{fpexpr}_1\rangle$ and the $\langle\text{fpexpr}_2\rangle$, and returns **true** if the $\langle\text{relation}\rangle$ is obeyed. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x ? y$ (“not ordered”). The last case occurs exactly if one or both operands is NaN or is a tuple, unless they are equal tuples. Note that a NaN is distinct from any value, even another NaN , hence $x = x$ is not true for a NaN . To test if a value is NaN , compare it to an arbitrary number with the “not ordered” relation.

```
\fp_compare:nNnTF { <value> } ? { 0 }
{ } % <value> is nan
{ } % <value> is not nan
```

Tuples are equal if they have the same number of items and items compare equal (in particular there must be no NaN). At present any other comparison with tuples yields **?** (not ordered). This is experimental.

This function is less flexible than \fp_compare:nTF but slightly faster. It is provided for consistency with $\text{\int_compare:nNnTF}$ and $\text{\dim_compare:nNnTF}$.

\fp_compare_p:n ★ \fp_compare:nTF ★

Updated: 2013-12-14

```
\fp_compare_p:n
{
  ⟨fpexpr₁⟩ ⟨relation₁⟩
  ...
  ⟨fpexprₙ⟩ ⟨relationₙ⟩
  ⟨fpexpr_{n+1}⟩
}
\fp_compare:nTF
{
  ⟨fpexpr₁⟩ ⟨relation₁⟩
  ...
  ⟨fpexprₙ⟩ ⟨relationₙ⟩
  ⟨fpexpr_{n+1}⟩
}
{⟨true code⟩} {⟨false code⟩}
```

Evaluates the *<floating point expressions>* as described for \fp_eval:n and compares consecutive result using the corresponding *<relation>*, namely it compares *<fpexpr₁>* and *<fpexpr₂>* using the *<relation₁>*, then *<fpexpr₂>* and *<fpexpr₃>* using the *<relation₂>*, until finally comparing *<fpexpr_N>* and *<fpexpr_{N+1}>* using the *<relation_N>*. The test yields **true** if all comparisons are **true**. Each *<floating point expression>* is evaluated only once. Contrarily to \int_compare:nTF, all *<floating point expressions>* are computed, even if one comparison is **false**. Two floating points *x* and *y* may obey four mutually exclusive relations: *x < y*, *x = y*, *x > y*, or *x?y* (“not ordered”). The last case occurs exactly if one or both operands is **NaN** or is a tuple, unless they are equal tuples. Each *<relation>* can be any (non-empty) combination of **<**, **=**, **>**, and **?**, plus an optional leading **!** (which negates the *<relation>*), with the restriction that the *<relation>* may not start with **?**, as this symbol has a different meaning (in combination with **:**) within floating point expressions. The comparison *x < relation y* is then **true** if the *<relation>* does not start with **!** and the actual relation (**<**, **=**, **>**, or **?**) between *x* and *y* appears within the *<relation>*, or on the contrary if the *<relation>* starts with **!** and the relation between *x* and *y* does not appear within the *<relation>*. Common choices of *<relation>* include **>=** (greater or equal), **!=** (not equal), **!?** or **<=>** (comparable).

This function is more flexible than \fp_compare:nNnTF and only slightly slower.

5 Floating point expression loops

\fp_do_until:nNnn ★

New: 2012-08-16

\fp_do_until:nNnn {⟨fpexpr₁⟩} ⟨relation⟩ {⟨fpexpr₂⟩} {⟨code⟩}

Places the *<code>* in the input stream for TeX to process, and then evaluates the relationship between the two *<floating point expressions>* as described for \fp_compare:nNnTF. If the test is **false** then the *<code>* is inserted into the input stream again and a loop occurs until the *<relation>* is **true**.

\fp_do_while:nNnn ★

New: 2012-08-16

\fp_do_while:nNnn {⟨fpexpr₁⟩} ⟨relation⟩ {⟨fpexpr₂⟩} {⟨code⟩}

Places the *<code>* in the input stream for TeX to process, and then evaluates the relationship between the two *<floating point expressions>* as described for \fp_compare:nNnTF. If the test is **true** then the *<code>* is inserted into the input stream again and a loop occurs until the *<relation>* is **false**.

\fp_until_do:nNnn ☆New: 2012-08-16

\fp_until_do:nNnn {*fpexpr₁*} *relation* {*fpexpr₂*} {*code*}

Evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nNnTF, and then places the *code* in the input stream if the *relation* is **false**. After the *code* has been processed by TeX the test is repeated, and a loop occurs until the test is **true**.

\fp_while_do:nNnn ☆New: 2012-08-16

\fp_while_do:nNnn {*fpexpr₁*} *relation* {*fpexpr₂*} {*code*}

Evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nNnTF, and then places the *code* in the input stream if the *relation* is **true**. After the *code* has been processed by TeX the test is repeated, and a loop occurs until the test is **false**.

\fp_do_until:nn ☆New: 2012-08-16

Updated: 2013-12-14

\fp_do_until:nn { *fpexpr₁* *relation* *fpexpr₂* } {*code*}

Places the *code* in the input stream for TeX to process, and then evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nTF. If the test is **false** then the *code* is inserted into the input stream again and a loop occurs until the *relation* is **true**.

\fp_do_while:nn ☆New: 2012-08-16

Updated: 2013-12-14

\fp_do_while:nn { *fpexpr₁* *relation* *fpexpr₂* } {*code*}

Places the *code* in the input stream for TeX to process, and then evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nTF. If the test is **true** then the *code* is inserted into the input stream again and a loop occurs until the *relation* is **false**.

\fp_until_do:nn ☆New: 2012-08-16

Updated: 2013-12-14

\fp_until_do:nn { *fpexpr₁* *relation* *fpexpr₂* } {*code*}

Evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nTF, and then places the *code* in the input stream if the *relation* is **false**. After the *code* has been processed by TeX the test is repeated, and a loop occurs until the test is **true**.

\fp_while_do:nn ☆New: 2012-08-16

Updated: 2013-12-14

\fp_while_do:nn { *fpexpr₁* *relation* *fpexpr₂* } {*code*}

Evaluates the relationship between the two *floating point expressions* as described for \fp_compare:nTF, and then places the *code* in the input stream if the *relation* is **true**. After the *code* has been processed by TeX the test is repeated, and a loop occurs until the test is **false**.

\fp_step_function:nnnN ☆
\fp_step_function:nnnc ☆

New: 2016-11-21
Updated: 2016-12-06

\fp_step_function:nnnN {<initial value>} {<step>} {<final value>} <function>

This function first evaluates the *<initial value>*, *<step>* and *<final value>*, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The *<function>* is then placed in front of each *<value>* from the *<initial value>* to the *<final value>* in turn (using *<step>* between each *<value>*). The *<step>* must be non-zero. If the *<step>* is positive, the loop stops when the *<value>* becomes larger than the *<final value>*. If the *<step>* is negative, the loop stops when the *<value>* becomes smaller than the *<final value>*. The *<function>* should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }  
\fp_step_function:nnnN { 1.0 } { 0.1 } { 1.5 } \my_func:n
```

would print

```
[I saw 1.0] [I saw 1.1] [I saw 1.2] [I saw 1.3] [I saw 1.4] [I saw 1.5]
```

TEXhackers note: Due to rounding, it may happen that adding the *<step>* to the *<value>* does not change the *<value>*; such cases give an error, as they would otherwise lead to an infinite loop.

\fp_step_inline:nnnn

New: 2016-11-21
Updated: 2016-12-06

\fp_step_inline:nnnn {<initial value>} {<step>} {<final value>} {<code>}

This function first evaluates the *<initial value>*, *<step>* and *<final value>*, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each *<value>* from the *<initial value>* to the *<final value>* in turn (using *<step>* between each *<value>*), the *<code>* is inserted into the input stream with *#1* replaced by the current *<value>*. Thus the *<code>* should define a function of one argument (*#1*).

\fp_step_variable:nnnN

New: 2017-04-12

\fp_step_variable:nnnN
{<initial value>} {<step>} {<final value>} {tl var} {<code>}

This function first evaluates the *<initial value>*, *<step>* and *<final value>*, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each *<value>* from the *<initial value>* to the *<final value>* in turn (using *<step>* between each *<value>*), the *<code>* is inserted into the input stream, with the *{tl var}* defined as the current *<value>*. Thus the *<code>* should make use of the *{tl var}*.

6 Some useful constants, and scratch variables

\c_zero_fp
\c_minus_zero_fp

New: 2012-05-08

Zero, with either sign.

\c_one_fp

New: 2012-05-08

One as an fp: useful for comparisons in some places.

`\c_inf_fp`
`\c_minus_inf_fp`

Infinity, with either sign. These can be input directly in a floating point expression as `inf` and `-inf`.

New: 2012-05-08

`\c_e_fp`

The value of the base of the natural logarithm, $e = \exp(1)$.

Updated: 2012-05-08

`\c_pi_fp`

The value of π . This can be input directly in a floating point expression as `pi`.

Updated: 2013-11-17

`\c_one_degree_fp`

New: 2012-05-08

Updated: 2013-11-17

The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as `deg`.

`\l_tmpa_fp`
`\l_tmpb_fp`

Scratch floating points for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_fp`
`\g_tmpb_fp`

Scratch floating points for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

7 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

“Exceptions” may occur when performing some floating point operations, such as $0 / 0$, or $10 ** 1e9999$. The relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0 .
- *Invalid operation* occurs for operations with no defined outcome, for instance $0/0$ or $\sin(\infty)$, and results in a `NaN`. It also occurs for conversion functions whose target type does not have the appropriate infinite or `NaN` value (*e.g.*, `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating functions at poles, *e.g.*, $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.

(not yet) *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in L^AT_EX3.

To each exception we associate a “flag”: `fp_overflow`, `fp_underflow`, `fp_invalid_operation` and `fp_division_by_zero`. The state of these flags can be tested and modified with commands from `\f3flag`.

By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions raise the corresponding flag but do not trigger an error. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

`\fp_trap:nn` `\fp_trap:nn {<exception>} {<trap type>}`

New: 2012-07-19

Updated: 2017-02-13

All occurrences of the `<exception>` (`overflow`, `underflow`, `invalid_operation` or `division_by_zero`) within the current group are treated as `<trap type>`, which can be

- `none`: the `<exception>` will be entirely ignored, and leave no trace;
- `flag`: the `<exception>` will turn the corresponding flag on when it occurs;
- `error`: additionally, the `<exception>` will halt the T_EX run and display some information about the current operation in the terminal.

This function is experimental, and may be altered or removed.

`flag_fp_overflow` Flags denoting the occurrence of various floating-point exceptions.

`flag_fp_underflow`

`flag_fp_invalid_operation`

`flag_fp_division_by_zero`

8 Viewing floating points

`\fp_show:N` `\fp_show:N {fp var}`
`\fp_show:c` `\fp_show:n {floating point expression}`

Evaluates the `<floating point expression>` and displays the result in the terminal.

New: 2012-05-08

Updated: 2015-08-07

`\fp_log:N` `\fp_log:N {fp var}`
`\fp_log:c` `\fp_log:n {floating point expression}`

Evaluates the `<floating point expression>` and writes the result in the log file.

New: 2014-08-22

Updated: 2015-08-07

9 Floating point expressions

9.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm m \cdot 10^n$, a floating point number, with integer $1 \leq m \leq 10^{16}$, and $-10000 \leq n \leq 10000$;
- ± 0 , zero, with a given sign;
- $\pm\infty$, infinity, with a given sign;
- **NaN**, is “not a number”, and can be either quiet or signalling (*not yet*: this distinction is currently unsupported);

Normal floating point numbers are stored in base 10, with up to 16 significant figures.

On input, a normal floating point number consists of:

- $\langle sign \rangle$: a possibly empty string of + and - characters;
- $\langle significand \rangle$: a non-empty string of digits together with zero or one dot;
- $\langle exponent \rangle$ optionally: the character e or E, followed by a possibly empty string of + and - tokens, and a non-empty string of digits.

The sign of the resulting number is + if $\langle sign \rangle$ contains an even number of -, and - otherwise, hence, an empty $\langle sign \rangle$ denotes a non-negative input. The stored significand is obtained from $\langle significand \rangle$ by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input $\langle significand \rangle$ has at most 16 digits. The stored $\langle exponent \rangle$ is obtained by combining the input $\langle exponent \rangle$ (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting $\langle exponent \rangle$ is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm\infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if $\langle significand \rangle$ contains no non-zero digit (*i.e.*, consists only in characters 0, and an optional period), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to +0, but that is not guaranteed to remain true.

The $\langle significand \rangle$ must be non-empty, so e1 and e-1 are not valid floating point numbers. Note that the latter could be mistaken with the difference of “e” and 1. To avoid confusions, the base of natural logarithms cannot be input as e and should be input as `exp(1)` or `\c_e_fp` (which is faster).

Special numbers are input as follows:

- `inf` represents $+\infty$, and can be preceded by any $\langle sign \rangle$, yielding $\pm\infty$ as appropriate.
- `nan` represents a (quiet) non-number. It can be preceded by any sign, but that sign is ignored.
- Any unrecognizable string triggers an error, and produces a **NaN**.
- Note that commands such as `\infty`, `\pi`, or `\sin` do not work in floating point expressions. They may silently be interpreted as completely unexpected numbers, because integer constants (allowed in expressions) are commonly stored as mathematical characters.

9.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (`sin`, `ln`, *etc*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Implicit multiplication by juxtaposition (`2pi`) when neither factor is in parentheses.
- Binary `*` and `/`, implicit multiplication by juxtaposition with parentheses (for instance `3(4+5)`).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc*.
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).
- Comma (to build tuples).

The precedence of operations can be overridden using parentheses. In particular, the precedence of juxtaposition implies that

$$\begin{aligned} 1/2\text{pi} &= 1/(2\pi), \\ 1/2\text{pi}(\text{pi} + \text{pi}) &= (2\pi)^{-1}(\pi + \pi) \simeq 1, \\ \text{sin}2\text{pi} &= \sin(2)\pi \neq 0, \\ 2^{\wedge}2\text{max}(3,5) &= 2^2 \max(3,5) = 20, \\ \text{1in}/\text{1cm} &= (\text{1in})/(\text{1cm}) = 2.54. \end{aligned}$$

Functions are called on the value of their argument, contrarily to TeX macros.

9.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is `false` if it is ± 0 , and `true` otherwise, including when it is `Nan` or a tuple such as $(0, 0)$. Tuples are only supported to some extent by operations that work with truth values (`?:`, `||`, `&&`, `!`), by comparisons (`!<=>?`), and by `+`, `-`, `*`, `/`. Unless otherwise specified, providing a tuple as an argument of any other operation yields the “invalid operation” exception and a `Nan` result.

?: \fp_eval:n { ⟨operand₁⟩ ? ⟨operand₂⟩ : ⟨operand₃⟩ }

The ternary operator **?:** results in ⟨⟨operand₂⟩ if ⟨⟨operand₁⟩ is true (not ±0), and ⟨⟨operand₃⟩ if ⟨⟨operand₁⟩ is false (±0). All three ⟨⟨operands⟩ are evaluated in all cases; they may be tuples. The operator is right associative, hence

```
\fp_eval:n
{
  1 + 3 > 4 ? 1 :
  2 + 4 > 5 ? 2 :
  3 + 5 > 6 ? 3 : 4
}
```

first tests whether $1 + 3 > 4$; since this isn't true, the branch following : is taken, and $2 + 4 > 5$ is compared; since this is true, the branch before : is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

|| \fp_eval:n { ⟨operand₁⟩ || ⟨operand₂⟩ }

If ⟨⟨operand₁⟩ is true (not ±0), use that value, otherwise the value of ⟨⟨operand₂⟩. Both ⟨⟨operands⟩ are evaluated in all cases; they may be tuples. In ⟨⟨operand₁⟩ || ⟨⟨operand₂⟩ || ... || ⟨⟨operands_n⟩, the first true (nonzero) ⟨⟨operand⟩ is used and if all are zero the last one (±0) is used.

&& \fp_eval:n { ⟨operand₁⟩ && ⟨operand₂⟩ }

If ⟨⟨operand₁⟩ is false (equal to ±0), use that value, otherwise the value of ⟨⟨operand₂⟩. Both ⟨⟨operands⟩ are evaluated in all cases; they may be tuples. In ⟨⟨operand₁⟩ && ⟨⟨operand₂⟩ && ... && ⟨⟨operands_n⟩, the first false (±0) ⟨⟨operand⟩ is used and if none is zero the last one is used.

```
<
=
>
?
_____
Updated: 2013-12-14
```

```
\fp_eval:n
{
  ⟨operand1⟩ ⟨relation1⟩
  ...
  ⟨operandN⟩ ⟨relationN⟩
  ⟨operandN+1⟩
}
```

Each ⟨⟨relation⟩ consists of a non-empty string of <, =, >, and ?, optionally preceded by !, and may not start with ?. This evaluates to +1 if all comparisons ⟨⟨operand_i⟩ ⟨⟨relation_i⟩ ⟨⟨operand_{i+1}⟩ are true, and +0 otherwise. All ⟨⟨operands⟩ are evaluated (once) in all cases. See \fp_compare:nTF for details.

+ \fp_eval:n { ⟨operand₁⟩ + ⟨operand₂⟩ }
- \fp_eval:n { ⟨operand₁⟩ - ⟨operand₂⟩ }

Computes the sum or the difference of its two ⟨⟨operands⟩. The “invalid operation” exception occurs for $\infty - \infty$. “Underflow” and “overflow” occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the “invalid operation” exception occurs and the result is NaN.

- `\fp_eval:n { <operand1> * <operand2> }`
`\fp_eval:n { <operand1> / <operand2> }`

Computes the product or the ratio of its two *operands*. The “invalid operation” exception occurs for ∞/∞ , $0/0$, or $0 * \infty$. “Division by zero” occurs when dividing a finite non-zero number by ± 0 . “Underflow” and “overflow” occur when appropriate. When *operand₁* is a tuple and *operand₂* is a floating point number, each item of *operand₁* is multiplied or divided by *operand₂*. Multiplication also supports the case where *operand₁* is a floating point number and *operand₂* a tuple. Other combinations yield an “invalid operation” exception and a NaN result.
- `\fp_eval:n { + <operand> }`
`\fp_eval:n { - <operand> }`
`\fp_eval:n { ! <operand> }`

The unary + does nothing, the unary - changes the sign of the *operand* (for a tuple, of all its components), and ! *operand* evaluates to 1 if *operand* is false (is ± 0) and 0 otherwise (this is the not boolean function). Those operations never raise exceptions.
- `\fp_eval:n { <operand1> ** <operand2> }`
`\fp_eval:n { <operand1> ^ <operand2> }`

Raises *operand₁* to the power *operand₂*. This operation is right associative, hence $2 ** 2 ** 3 = 2^{2^3} = 256$. If *operand₁* is negative or -0 then: the result’s sign is + if the *operand₂* is infinite and $(-1)^p$ if the *operand₂* is $p/5^q$ with *p*, *q* integers; the result is $+0$ if `abs(<operand1>)^<operand2>` evaluates to zero; in other cases the “invalid operation” exception occurs because the sign cannot be determined. “Division by zero” occurs when raising ± 0 to a finite strictly negative power. “Underflow” and “overflow” occur when appropriate. If either operand is a tuple, “invalid operation” occurs.
- abs `\fp_eval:n { abs(<fpexpr>) }`

Computes the absolute value of the *fpexpr*. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases. See also `\fp_abs:n`.
- exp `\fp_eval:n { exp(<fpexpr>) }`

Computes the exponential of the *fpexpr*. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.
- fact `\fp_eval:n { fact(<fpexpr>) }`

Computes the factorial of the *fpexpr*. If the *fpexpr* is an integer between -0 and 3248 included, the result is finite and correctly rounded. Larger positive integers give $+\infty$ with “overflow”, while `fact(+∞) = +∞` and `fact(nan) = nan` with no exception. All other inputs give NaN with the “invalid operation” exception.
- ln `\fp_eval:n { ln(<fpexpr>) }`

Computes the natural logarithm of the *fpexpr*. Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for $\ln(-0)$. “Division by zero” occurs when evaluating $\ln(+0) = -\infty$. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

logb	\star	<code>\fp_eval:n { logb(<fexpr>) }</code>
New: 2018-11-03		Determines the exponent of the $\langle fexpr \rangle$, namely the floor of the base-10 logarithm of its absolute value. “Division by zero” occurs when evaluating $\text{logb}(\pm 0) = -\infty$. Other special values are $\text{logb}(\pm\infty) = +\infty$ and $\text{logb}(\text{NaN}) = \text{NaN}$. If the operand is a tuple or is NaN , then “invalid operation” occurs and the result is NaN .
max		<code>\fp_eval:n { max(<fexpr1> , <fexpr2> , ...) }</code>
min		<code>\fp_eval:n { min(<fexpr1> , <fexpr2> , ...) }</code>
		Evaluates each $\langle fexpr \rangle$ and computes the largest (smallest) of those. If any of the $\langle fexpr \rangle$ is a NaN or tuple, the result is NaN . If any operand is a tuple, “invalid operation” occurs; these operations do not raise exceptions in other cases.
round		<code>\fp_eval:n { round(<fexpr>) }</code>
trunc		<code>\fp_eval:n { round(<fexpr1> , <fexpr2>) }</code>
ceil		<code>\fp_eval:n { round(<fexpr1> , <fexpr2> , <fexpr3>) }</code>
floor		
New: 2013-12-14		Only round accepts a third argument. Evaluates $\langle fexpr_1 \rangle = x$ and $\langle fexpr_2 \rangle = n$ and $\langle fexpr_3 \rangle = t$ then rounds x to n places. If n is an integer, this rounds x to a multiple of 10^{-n} ; if $n = +\infty$, this always yields x ; if $n = -\infty$, this yields one of ± 0 , $\pm\infty$, or NaN ; if $n = \text{NaN}$, this yields NaN ; if n is neither $\pm\infty$ nor an integer, then an “invalid operation” exception is raised. When $\langle fexpr_2 \rangle$ is omitted, $n = 0$, i.e., $\langle fexpr_1 \rangle$ is rounded to an integer. The rounding direction depends on the function.
Updated: 2015-08-08		
		<ul style="list-style-type: none"> • round yields the multiple of 10^{-n} closest to x, with ties (x half-way between two such multiples) rounded as follows. If t is nan (or not given) the even multiple is chosen (“ties to even”), if $t = \pm 0$ the multiple closest to 0 is chosen (“ties to zero”), if t is positive/negative the multiple closest to $\infty/-\infty$ is chosen (“ties towards positive/negative infinity”). • floor yields the largest multiple of 10^{-n} smaller or equal to x (“round towards negative infinity”); • ceil yields the smallest multiple of 10^{-n} greater or equal to x (“round towards positive infinity”); • trunc yields a multiple of 10^{-n} with the same sign as x and with the largest absolute value less than that of x (“round towards zero”).
		“Overflow” occurs if x is finite and the result is infinite (this can only happen if $\langle fexpr_2 \rangle < -9984$). If any operand is a tuple, “invalid operation” occurs.
sign		<code>\fp_eval:n { sign(<fexpr>) }</code>
		Evaluates the $\langle fexpr \rangle$ and determines its sign: $+1$ for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0 , and NaN for NaN . If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.

```
sin          \fp_eval:n { sin( <fpexpr> ) }
cos          \fp_eval:n { cos( <fpexpr> ) }
tan          \fp_eval:n { tan( <fpexpr> ) }
cot          \fp_eval:n { cot( <fpexpr> ) }
csc          \fp_eval:n { csc( <fpexpr> ) }
sec          \fp_eval:n { sec( <fpexpr> ) }
```

Updated: 2013-11-17

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$ given in radians. For arguments given in degrees, see `sind`, `cosd`, etc. Note that since π is irrational, $\sin(8\pi)$ is not quite zero, while its analogue $\text{sind}(8 \times 180)$ is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```
sind          \fp_eval:n { sind( <fpexpr> ) }
cosd          \fp_eval:n { cosd( <fpexpr> ) }
tand          \fp_eval:n { tand( <fpexpr> ) }
cotd          \fp_eval:n { cotd( <fpexpr> ) }
cscd          \fp_eval:n { cscd( <fpexpr> ) }
secd          \fp_eval:n { secd( <fpexpr> ) }
```

New: 2013-11-02

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$ given in degrees. For arguments given in radians, see `sin`, `cos`, etc. Note that since π is irrational, $\sin(8\pi)$ is not quite zero, while its analogue $\text{sind}(8 \times 180)$ is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```
asin          \fp_eval:n { asin( <fpexpr> ) }
acos          \fp_eval:n { acos( <fpexpr> ) }
acsc          \fp_eval:n { acsc( <fpexpr> ) }
asec          \fp_eval:n { asec( <fpexpr> ) }
```

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fpexpr \rangle$ and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for `asin` and `acsc` and $[0, \pi]$ for `acos` and `asec`. For a result in degrees, use `asind`, etc. If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```
asind          \fp_eval:n { asind( <fpexpr> ) }
acosd          \fp_eval:n { acosd( <fpexpr> ) }
acscd          \fp_eval:n { acscd( <fpexpr> ) }
asecd          \fp_eval:n { aecd( <fpexpr> ) }
```

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fpexpr \rangle$ and returns the result in degrees, in the range $[-90, 90]$ for `asin` and `acsc` and $[0, 180]$ for `acos` and `asec`. For a result in radians, use `asin`, etc. If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

atan	\fp_eval:n { atan(<fpexpr>) }
acot	\fp_eval:n { atan(<fpexpr1> , <fpexpr2>) }
New: 2013-11-02	\fp_eval:n { acot(<fpexpr>) }
	\fp_eval:n { acot(<fpexpr1> , <fpexpr2>) }

Those functions yield an angle in radians: **atand** and **acotd** are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-\pi/2, \pi/2]$, and arccotangent in the range $[0, \pi]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by π depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by π . Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm\infty$, the resulting angle is one of $\{\pm\pi/4, \pm 3\pi/4\}$ depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

atand	\fp_eval:n { atand(<fpexpr>) }
acotd	\fp_eval:n { atand(<fpexpr1> , <fpexpr2>) }
New: 2013-11-02	\fp_eval:n { acotd(<fpexpr>) }
	\fp_eval:n { acotd(<fpexpr1> , <fpexpr2>) }

Those functions yield an angle in degrees: **atand** and **acotd** are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-90, 90]$, and arccotangent in the range $[0, 180]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by 180 depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm\infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

sqrt	\fp_eval:n { sqrt(<fpexpr>) }
------	---------------------------------

New: 2013-12-14

Computes the square root of the $\langle fpexpr \rangle$. The “invalid operation” is raised when the $\langle fpexpr \rangle$ is negative or is a tuple; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\text{NaN}} = \text{NaN}$.

rand `\fp_eval:n { rand() }`

New: 2016-12-05

Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded. This is not available in older versions of X_ET_EX. The random seed can be queried using `\sys_rand_seed:` and set using `\sys_gset_rand_seed:n`.

TeXhackers note: This is based on pseudo-random numbers provided by the engine's primitive `\pdfuniformdeviate` in pdfT_EX, pT_EX, upT_EX and `\uniformdeviate` in LuaT_EX and X_ET_EX. The underlying code is based on Metapost, which follows an additive scheme recommended in Section 3.6 of "The Art of Computer Programming, Volume 2".

While we are more careful than `\uniformdeviate` to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.

randint `\fp_eval:n { randint(<fpexpr>) }`
`\fp_eval:n { randint(<fpexpr1> , <fpexpr2>) }`

New: 2016-12-05

Produces a pseudo-random integer between 1 and $\langle fpexpr \rangle$ or between $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$ inclusive. The bounds must be integers in the range $(-10^{16}, 10^{16})$ and the first must be smaller or equal to the second. See `rand` for important comments on how these pseudo-random numbers are generated.

inf The special values $+\infty$, $-\infty$, and NaN are represented as `inf`, `-inf` and `nan` (see `\c_-inf_fp`, `\c_minus_inf_fp` and `\c_nan_fp`).

pi The value of π (see `\c_pi_fp`).

deg The value of 1° in radians (see `\c_one_degree_fp`).

<u>em</u>	Those units of measurement are equal to their values in <code>pt</code> , namely
<u>ex</u>	
<u>in</u>	$1 \text{ in} = 72.27 \text{ pt}$
<u>pt</u>	$1 \text{ pt} = 1 \text{ pt}$
<u>pc</u>	$1 \text{ pc} = 12 \text{ pt}$
<u>cm</u>	$1 \text{ cm} = \frac{1}{2.54} \text{ in} = 28.45275590551181 \text{ pt}$
<u>mm</u>	$1 \text{ mm} = \frac{1}{25.4} \text{ in} = 2.845275590551181 \text{ pt}$
<u>dd</u>	$1 \text{ dd} = 0.376065 \text{ mm} = 1.07000856496063 \text{ pt}$
<u>cc</u>	$1 \text{ cc} = 12 \text{ dd} = 12.84010277952756 \text{ pt}$
<u>nd</u>	$1 \text{ nd} = 0.375 \text{ mm} = 1.066978346456693 \text{ pt}$
<u>nc</u>	$1 \text{ nc} = 12 \text{ nd} = 12.80374015748031 \text{ pt}$
<u>bp</u>	$1 \text{ bp} = \frac{1}{72} \text{ in} = 1.00375 \text{ pt}$
<u>sp</u>	$1 \text{ sp} = 2^{-16} \text{ pt} = 1.52587890625 \times 10^{-5} \text{ pt}.$

The values of the (font-dependent) units `em` and `ex` are gathered from TeX when the surrounding floating point expression is evaluated.

<u>true</u>	Other names for 1 and +0.
<u>false</u>	

\fp_abs:n *

New: 2012-05-14
Updated: 2012-07-08

`\fp_abs:n {<floating point expression>}`

Evaluates the `<floating point expression>` as described for `\fp_eval:n` and leaves the absolute value of the result in the input stream. If the argument is $\pm\infty$, NaN or a tuple, “invalid operation” occurs. Within floating point expressions, `abs()` can be used; it accepts $\pm\infty$ and NaN as arguments.

\fp_max:nn *

\fp_min:nn *

New: 2012-09-26

`\fp_max:nn {{fp expression 1}} {{fp expression 2}}`

Evaluates the `<floating point expressions>` as described for `\fp_eval:n` and leaves the resulting larger (`max`) or smaller (`min`) value in the input stream. If the argument is a tuple, “invalid operation” occurs, but no other case raises exceptions. Within floating point expressions, `max()` and `min()` can be used.

10 Disclaimer and roadmap

The package may break down if the escape character is among 0123456789_+, or if it receives a TeX primitive conditional affected by `\exp_not:N`.

The following need to be done. I’ll try to time-order the items.

- Function to count items in a tuple (and to determine if something is a tuple).
- Decide what exponent range to consider.

- Support signalling `nan`.
- Modulo and remainder, and rounding function `quantize` (and its friends analogous to `trunc`, `ceil`, `floor`).
- `\fp_format:nn {⟨fexpr⟩} {⟨format⟩}`, but what should `⟨format⟩` be? More general pretty printing?
- Add `and`, `or`, `xor`? Perhaps under the names `all`, `any`, and `xor`?
- Add $\log(x, b)$ for logarithm of x in base b .
- `hypot` (Euclidean length). Cartesian-to-polar transform.
- Hyperbolic functions `cosh`, `sinh`, `tanh`.
- Inverse hyperbolics.
- Base conversion, input such as `0xAB.CDEF`.
- Factorial (not with !), gamma function.
- Improve coefficients of the `sin` and `tan` series.
- Treat upper and lower case letters identically in identifiers, and ignore underscores.
- Add an `array(1,2,3)` and `i=complex(0,1)`.
- Provide an experimental `map` function? Perhaps easier to implement if it is a single character, `@sin(1,2)`?
- Provide an `isnan` function analogue of `\fp_if_nan:nTF`?
- Support keyword arguments?

`Pgfmath` also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs, and tests to add.

- Check that functions are monotonic when they should.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- Logarithms of numbers very close to 1 are inaccurate.
- When rounding towards $-\infty$, `\dim_to_fp:n {0pt}` should return -0 , not $+0$.
- The result of $(\pm 0) + (\pm 0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.
- `0e9999999999` gives a `TEX` “number too large” error.
- Subnormals are not implemented.

Possible optimizations/improvements.

- Document that `l3trial/l3fp-types` introduces tools for adding new types.
- In subsection 9.1, write a grammar.

- It would be nice if the `parse` auxiliaries for each operation were set up in the corresponding module, rather than centralizing in `l3fp-parse`.
- Some functions should get an `_o` ending to indicate that they expand after their result.
- More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.
- The code for the `ternary` set of functions is ugly.
- There are many `\~` missing in the doc to avoid bad line-breaks.
- The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking $c = 2000/(\lfloor 200x \rfloor + 1) \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of t . However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (`l3fp-basics`).
- Understand and document `_fp_basics_pack_weird_low>NNNNw` and `_fp_basics_pack_weird_high>NNNNNNNNw` better. Move the other `basics_pack` auxiliaries to `l3fp-aux` under a better name.
- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan's articles, some previous TeX fp packages, the international standards,...
- Also take into account the “inexact” exception?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)?

Part XXIV

The **I3fparray** package: fast global floating point arrays

1 I3fparray documentation

For applications requiring heavy use of floating points, this module provides arrays which can be accessed in constant time (contrast **I3seq**, where access time is linear). The interface is very close to that of **I3intarray**. The size of the array is fixed and must be given at point of initialisation

\fparray_new:Nn

New: 2018-05-05

\fparray_new:Nn *fparray var* {*size*}

Evaluates the integer expression *size* and allocates an *floating point array variable* with that number of (zero) entries. The variable name should start with **\g_** because assignments are always global.

\fparray_count:N *

New: 2018-05-05

\fparray_count:N *fparray var*

Expands to the number of entries in the *floating point array variable*. This is performed in constant time.

\fparray_gset:Nnn

New: 2018-05-05

\fparray_gset:Nnn *fparray var* {*position*} {*value*}

Stores the result of evaluating the floating point expression *value* into the *floating point array variable* at the (integer expression) *position*. If the *position* is not between 1 and the **\fparray_count:N**, an error occurs. Assignments are always global.

\fparray_gzero:N

New: 2018-05-05

\fparray_gzero:N *fparray var*

Sets all entries of the *floating point array variable* to +0. Assignments are always global.

\fparray_item:Nn *
\fparray_item_to_tl:Nn *

New: 2018-05-05

\fparray_item:Nn *fparray var* {*position*}

Applies **\fp_use:N** or **\fp_to_tl:N** (respectively) to the floating point entry stored at the (integer expression) *position* in the *floating point array variable*. If the *position* is not between 1 and the **\fparray_count:N**, an error occurs.

Part XXV

The **l3cctab** package

Category code tables

A category code table enables rapid switching of all category codes in one operation. For LuaTeX, this is possible over the entire Unicode range. For other engines, only the 8-bit range (0–255) is covered by such tables.

1 Creating and initialising category code tables

\cctab_new:N
\cctab_new:c

Updated: 2020-07-02

\cctab_new:N *category code table*

Creates a new *category code table* variable or raises an error if the name is already taken. The declaration is global. The *category code table* is initialised with the codes as used by iniTeX.

\cctab_const:Nn
\cctab_const:cn

Updated: 2020-07-07

\cctab_const:Nn *category code table* {*category code set up*}

Creates a new *category code table*, applies (in a group) the *category code set up* on top of iniTeX settings, then saves them globally as a constant table. The *category code set up* can include a call to \cctab_select:N.

\cctab_gset:Nn
\cctab_gset:cn

Updated: 2020-07-07

\cctab_gset:Nn *category code table* {*category code set up*}

Starting from the iniTeX category codes, applies (in a group) the *category code set up*, then saves them globally in the *category code table*. The *category code set up* can include a call to \cctab_select:N.

2 Using category code tables

\cctab_begin:N
\cctab_begin:c

Updated: 2020-07-02

\cctab_begin:N *category code table*

Switches locally the category codes in force to those stored in the *category code table*. The prevailing codes before the function is called are added to a stack, for use with \cctab_end:. This function does not start a TeX group.

\cctab_end:

Updated: 2020-07-02

\cctab_end:

Ends the scope of a *category code table* started using \cctab_begin:N, returning the codes to those in force before the matching \cctab_begin:N was used. This must be used within the same TeX group (and at the same TeX group level) as the matching \cctab_begin:N.

\cctab_select:N

New: 2020-05-19

Updated: 2020-07-02

\cctab_select:N *category code table*

Selects the *category code table* for the scope of the current group. This is in particular useful in the *setup* arguments of \tl_set_rescan:Nnn, \tl_rescan:nn, \cctab_const:Nn, and \cctab_gset:Nn.

3 Category code table conditionals

```
\cctab_if_exist_p:N *          \cctab_if_exist_p:N <category code table>
\cctab_if_exist_p:c *          \cctab_if_exist:NTF <category code table> {\<true code>} {\<false code>}
\cctab_if_exist:NTF *          Tests whether the <category code table> is currently defined. This does not check that the
\cctab_if_exist:cTF *          <category code table> really is a category code table.
```

4 Constant category code tables

\c_code_cctab
Updated: 2020-07-10

Category code table for the `expl3` code environment; this does *not* include `o`, which is retained as an “other” character.

\c_document_cctab
Updated: 2020-07-08

Category code table for a standard L^AT_EX document, as set by the L^AT_EX kernel. In particular, the upper-half of the 8-bit range will be set to “active” with pdfL^AT_EX *only*. No babel shorthands will be activated.

\c_initex_cctab
Updated: 2020-07-02

Category code table as set up by iniL^AT_EX.

\c_other_cctab
Updated: 2020-07-02

Category code table where all characters have category code 12 (other).

\c_str_cctab
Updated: 2020-07-02

Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space).

Part XXVI

The **I3sort** package

Sorting functions

1 Controlling sorting

L^AT_EX3 comes with a facility to sort list variables (sequences, token lists, or comma-lists) according to some user-defined comparison. For instance,

```
\clist_set:Nn \l_foo_clist { 3 , 01 , -2 , 5 , +1 }
\clist_sort:Nn \l_foo_clist
{
    \int_compare:nNnTF { #1 } > { #2 }
        { \sort_return_swapped: }
        { \sort_return_same: }
}
```

results in `\l_foo_clist` holding the values `{ -2 , 01 , +1 , 3 , 5 }` sorted in non-decreasing order.

The code defining the comparison should call `\sort_return_swapped:` if the two items given as `#1` and `#2` are not in the correct order, and otherwise it should call `\sort_return_same:` to indicate that the order of this pair of items should not be changed.

For instance, a *(comparison code)* consisting only of `\sort_return_same:` with no test yields a trivial sort: the final order is identical to the original order. Conversely, using a *(comparison code)* consisting only of `\sort_return_swapped:` reverses the list (in a fairly inefficient way).

TEXhackers note: The current implementation is limited to sorting approximately 20000 items (40000 in LuaT_EX), depending on what other packages are loaded.

Internally, the code from I3sort stores items in `\toks` registers allocated locally. Thus, the *(comparison code)* should not call `\newtoks` or other commands that allocate new `\toks` registers. On the other hand, altering the value of a previously allocated `\toks` register is not a problem.

`\sort_return_same:`
`\sort_return_swapped:`

New: 2017-02-06

```
\seq_sort:Nn <seq var>
{ ... \sort_return_same: or \sort_return_swapped: ... }
```

Indicates whether to keep the order or swap the order of two items that are compared in the sorting code. Only one of the `\sort_return_...` functions should be used by the code, according to the results of some tests on the items `#1` and `#2` to be compared.

Part XXVII

The **\3tl-analysis** package: Analysing token lists

1 \3tl-analysis documentation

This module mostly provides internal functions for use in the `\3regex` module. However, it provides as a side-effect a user debugging function, very similar to the `\ShowTokens` macro from the `ted` package.

```
\tl_analysis_show:N  
\tl_analysis_show:n
```

New: 2018-04-09

```
\tl_analysis_show:n {\langle token list\rangle}
```

Displays to the terminal the detailed decomposition of the `\langle token list\rangle` into tokens, showing the category code of each character token, the meaning of control sequences and active characters, and the value of registers.

```
\tl_analysis_map_inline:nn  
\tl_analysis_map_inline:Nn
```

New: 2018-04-09

```
\tl_analysis_map_inline:nn {\langle token list\rangle} {\langle inline function\rangle}
```

Applies the `\langle inline function\rangle` to each individual `\langle token\rangle` in the `\langle token list\rangle`. The `\langle inline function\rangle` receives three arguments:

- `\langle tokens\rangle`, which both `\o-`expand and `\x-`expand to the `\langle token\rangle`. The detailed form of `\langle token\rangle` may change in later releases.
- `\langle char code\rangle`, a decimal representation of the character code of the token, `-1` if it is a control sequence (with `\langle catcode\rangle 0`).
- `\langle catcode\rangle`, a capital hexadecimal digit which denotes the category code of the `\langle token\rangle` (`0`: control sequence, `1`: begin-group, `2`: end-group, `3`: math shift, `4`: alignment tab, `6`: parameter, `7`: superscript, `8`: subscript, `A`: space, `B`: letter, `C`:other, `D`:active).

As all other mappings the mapping is done at the current group level, *i.e.* any local assignments made by the `\langle inline function\rangle` remain in effect after the loop.

Part XXVIII

The `\3regex` package: Regular expressions in `TEX`

The `\3regex` package provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on token lists. The syntax of regular expressions is mostly a subset of the PCRE syntax (and very close to POSIX), with some additions due to the fact that `TEX` manipulates tokens rather than characters. For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After

```
\tl_set:Nn \l_my_tl { That~cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
```

the token list variable `\l_my_tl` holds the text “`This cat.`”, where the first occurrence of “`at`” was replaced by “`is`”. A more complicated example is a pattern to emphasize each word and add a comma after it:

```
\regex_replace_all:nnN { \w+ } { \c{emph}\cB{\0 }\cE\, } \l_my_tl
```

The `\w` sequence represents any “word” character, and `+` indicates that the `\w` sequence should be repeated as many times as possible (at least once), hence matching a word in the input token list. In the replacement text, `\0` denotes the full match (here, a word). The command `\emph` is inserted using `\c{emph}`, and its argument `\0` is put between braces `\cB{` and `\cE}`.

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using `\regex_const:Nn`. For example,

```
\regex_const:Nn \c_foo_regex { \c{begin} \cB. (\c[^BE].*) \cE. }
```

stores in `\c_foo_regex` a regular expression which matches the starting marker for an environment: `\begin`, followed by a begin-group token (`\cB.`), then any number of tokens which are neither begin-group nor end-group character tokens (`\c[^BE].*`), ending with an end-group token (`\cE.`). As explained in the next section, the parentheses “capture” the result of `\c[^BE].*`, giving us access to the name of the environment when doing replacements.

1 Syntax of regular expressions

We start with a few examples, and encourage the reader to apply `\regex_show:n` to these regular expressions.

- `Cat` matches the word “Cat” capitalized in this way, but also matches the beginning of the word “Cattle”: use `\bCat\b` to match a complete word only.
- `[abc]` matches one letter among “a”, “b”, “c”; the pattern `(a|b|c)` matches the same three possible letters (but see the discussion of submatches below).
- `[A-Za-z]*` matches any number (due to the quantifier `*`) of Latin letters (not accented).

- `\c{[A-Za-z]*}` matches a control sequence made of Latin letters.
- `_[^_]*_` matches an underscore, any number of characters other than underscore, and another underscore; it is equivalent to `_.*?_` where `.` matches arbitrary characters and the lazy quantifier `*?` means to match as few characters as possible, thus avoiding matching underscores.
- `[\+/-]?\d+` matches an explicit integer with at most one sign.
- `[\+/-\,]*\d+\,*` matches an explicit integer with any number of `+` and `-` signs, with spaces allowed except within the mantissa, and surrounded by spaces.
- `[\+/-\,]*(\d+|\d*\.\d+)\,*` matches an explicit integer or decimal number; using `[.,]` instead of `\.` would allow the comma as a decimal marker.
- `[\+/-\,]*(\d+|\d*\.\d+)\,*((?i)pt|in|[cem]m|ex|[bs]p|[dn]d|[pcn]c)\,*` matches an explicit dimension with any unit that `\TeX` knows, where `(?i)` means to treat lowercase and uppercase letters identically.
- `[\+/-\,]*((?i)nan|inf|(\d+|\d*\.\d+)(\,*\e[\+/-\,]*\d+)?)\,*` matches an explicit floating point number or the special values `nan` and `inf` (with signs and spaces allowed).
- `[\+/-\,]*(\d+|\cC.)\,*` matches an explicit integer or control sequence (without checking whether it is an integer variable).
- `\G.*?\K` at the beginning of a regular expression matches and discards (due to `\K`) everything between the end of the previous match (`\G`) and what is matched by the rest of the regular expression; this is useful in `\regex_replace_all:nnN` when the goal is to extract matches or submatches in a finer way than with `\regex_extract_all:nnN`.

While it is impossible for a regular expression to match only integer expressions, `[\+/-\,]*\d+)*([\+/-*/] [\+/-\,]*\d+)*` matches among other things all valid integer expressions (made only with explicit integers). One should follow it with further testing.

Most characters match exactly themselves, with an arbitrary category code. Some characters are special and must be escaped with a backslash (*e.g.*, `*` matches a star character). Some escape sequences of the form backslash–letter also have a special meaning (for instance `\d` matches any digit). As a rule,

- every alphanumeric character (`A–Z`, `a–z`, `0–9`) matches exactly itself, and should not be escaped, because `\A`, `\B`, ... have special meanings;
- non-alphanumeric printable ascii characters can (and should) always be escaped: many of them have special meanings (*e.g.*, use `\(`, `\)`, `\?`, `\.`);
- spaces should always be escaped (even in character classes);
- any other character may be escaped or not, without any effect: both versions match exactly that character.

Note that these rules play nicely with the fact that many non-alphanumeric characters are difficult to input into TeX under normal category codes. For instance, `\abc\%` matches the characters `\abc%` (with arbitrary category codes), but does not match the control sequence `\abc` followed by a percent character. Matching control sequences can be done using the `\c{\langle regex\rangle}` syntax (see below).

Any special character which appears at a place where its special behaviour cannot apply matches itself instead (for instance, a quantifier appearing at the beginning of a string), after raising a warning.

Characters.

`\x{hh...}` Character with hex code hh...

`\xhh` Character with hex code hh.

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

`\f` Form-feed (hex 0C).

`\n` New line (hex 0A).

`\r` Carriage return (hex 0D).

`\t` Horizontal tab (hex 09).

Character types.

- . A single period matches any token.

`\d` Any decimal digit.

`\h` Any horizontal space character, equivalent to `[\ \^\^I]`: space and tab.

`\s` Any space character, equivalent to `[\ \^\^I\^\^J\^\^L\^\^M]`.

`\v` Any vertical space character, equivalent to `[\^\^J\^\^K\^\^L\^\^M]`. Note that `\^\^K` is a vertical space, but not a space, for compatibility with Perl.

`\w` Any word character, *i.e.*, alphanumerics and underscore, equivalent to the explicit class `[A-Za-z0-9_]`.

`\D` Any token not matched by `\d`.

`\H` Any token not matched by `\h`.

`\N` Any token other than the `\n` character (hex 0A).

`\S` Any token not matched by `\s`.

`\V` Any token not matched by `\v`.

`\W` Any token not matched by `\w`.

Of those, ., `\D`, `\H`, `\N`, `\S`, `\V`, and `\W` match arbitrary control sequences.

Character classes match exactly one token in the subject.

[...] Positive character class. Matches any of the specified tokens.

[^...] Negative character class. Matches any token other than the specified characters.

x-y Within a character class, this denotes a range (can be used with escaped characters).

[:<name>:] Within a character class (one more set of brackets), this denotes the POSIX character class <name>, which can be `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `word`, or `xdigit`.

[:^<name>:] Negative POSIX character class.

For instance, `[a-oq-z\cc.]` matches any lowercase latin letter except p, as well as control sequences (see below for a description of \c).

Quantifiers (repetition).

? 0 or 1, greedy.

?? 0 or 1, lazy.

* 0 or more, greedy.

*? 0 or more, lazy.

+ 1 or more, greedy.

+? 1 or more, lazy.

{n} Exactly n.

{n,} n or more, greedy.

{n,}? n or more, lazy.

{n,m} At least n, no more than m, greedy.

{n,m}? At least n, no more than m, lazy.

Anchors and simple assertions.

\b Word boundary: either the previous token is matched by \w and the next by \W, or the opposite. For this purpose, the ends of the token list are considered as \W.

\B Not a word boundary: between two \w tokens or two \W tokens (including the boundary).

^or \A Start of the subject token list.

\$, \Z or \z End of the subject token list.

\G Start of the current match. This is only different from ^ in the case of multiple matches: for instance `\regex_count:nnN { \G a } { aaba } \l_tmpa_int` yields 2, but replacing \G by ^ would result in \l_tmpa_int holding the value 1.

Alternation and capturing groups.

A|B|C Either one of A, B, or C.

(...) Capturing group.

(?:...) Non-capturing group.

(?|...) Non-capturing group which resets the group number for capturing groups in each alternative. The following group is numbered with the first unused group number.

The \c escape sequence allows to test the category code of tokens, and match control sequences. Each character category is represented by a single uppercase letter:

- C for control sequences;
- B for begin-group tokens;
- E for end-group tokens;
- M for math shift;
- T for alignment tab tokens;
- P for macro parameter tokens;
- U for superscript tokens (up);
- D for subscript tokens (down);
- S for spaces;
- L for letters;
- O for others; and
- A for active characters.

The \c escape sequence is used as follows.

\c{\langle regex\rangle} A control sequence whose csname matches the *<regex>*, anchored at the beginning and end, so that \c{begin} matches exactly \begin, and nothing else.

\cX Applies to the next object, which can be a character, character property, class, or group, and forces this object to only match tokens with category X (any of CBEMTPUDSLOA). For instance, \cL[A-Z]\d matches uppercase letters and digits of category code letter, \cC. matches any control sequence, and \cO(abc) matches abc where each character has category other.

\c[XYZ] Applies to the next object, and forces it to only match tokens with category X, Y, or Z (each being any of CBEMTPUDSLOA). For instance, \c[LSO](..) matches two tokens of category letter, space, or other.

\c[^XYZ] Applies to the next object and prevents it from matching any token with category X, Y, or Z (each being any of CBEMTPUDSLOA). For instance, \c[^O]\d matches digits which have any category different from other.

The category code tests can be used inside classes; for instance, [\cO\d \c[LO][A-F]] matches what TeX considers as hexadecimal digits, namely digits with category other, or uppercase letters from A to F with category either letter or other. Within a group affected by a category code test, the outer test can be overridden by a nested test: for instance, \cL(ab\cO*cd) matches ab*cd where all characters are of category letter, except * which has category other.

The \u escape sequence allows to insert the contents of a token list directly into a regular expression or a replacement, avoiding the need to escape special characters.

Namely, `\u{<tl var name>}` matches the exact contents of the token list $\langle tl\ var \rangle$. Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\tl_to_str:v`. Quantifiers are not supported directly: use a group.

The option `(?i)` makes the match case insensitive (identifying `A-Z` with `a-z`; no Unicode support yet). This applies until the end of the group in which it appears, and can be reverted using `(?-i)`. For instance, in `(?i)(a(?-i)b|c)d`, the letters `a` and `d` are affected by the `i` option. Characters within ranges and classes are affected individually: `(?i)[Y-\\"]` is equivalent to `[YZ\[\\yz]`, and `(?i)[^aeiou]` matches any character which is not a vowel. Neither character properties, nor `\c{...}` nor `\u{...}` are affected by the `i` option.

In character classes, only `[`, `^`, `-`, `]`, `\` and spaces are special, and should be escaped. Other non-alphanumeric characters can still be escaped without harm. Any escape sequence which matches a single character (`\d`, `\D`, etc.) is supported in character classes. If the first character is `^`, then the meaning of the character class is inverted; `^` appearing anywhere else in the range is not special. If the first character (possibly following a leading `^`) is `]` then it does not need to be escaped since ending the range there would make it empty. Ranges of characters can be expressed using `-`, for instance, `[\D 0-5]` and `[^6-9]` are equivalent.

Capturing groups are a means of extracting information about the match. Parenthesized groups are labelled in the order of their opening parenthesis, starting at 1. The contents of those groups corresponding to the “best” match (leftmost longest) can be extracted and stored in a sequence of token lists using for instance `\regex_extract_once:nnNTF`.

The `\K` escape sequence resets the beginning of the match to the current position in the token list. This only affects what is reported as the full match. For instance,

```
\regex_extract_all:nnN { a \K . } { a123aaxyz } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{1}` and `{a}`: the true matches are `{a1}` and `{aa}`, but they are trimmed by the use of `\K`. The `\K` command does not affect capturing groups: for instance,

```
\regex_extract_once:nnN { (. \K c)+ \d } { acbc3 } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{c3}` and `{bc}`: the true match is `{acbc3}`, with first submatch `{bc}`, but `\K` resets the beginning of the match to the last position where it appears.

2 Syntax of the replacement text

Most of the features described in regular expressions do not make sense within the replacement text. Backslash introduces various special constructions, described further below:

- `\0` is the whole match;
- `\1` is the submatch that was matched by the first (capturing) group `(...)`; similarly for `\2`, ..., `\9` and `\g{<number>}`;
- `_` inserts a space (spaces are ignored when not escaped);

- `\a`, `\e`, `\f`, `\n`, `\r`, `\t`, `\xhh`, `\x{hhh}` correspond to single characters as in regular expressions;
- `\c{<cs name>}` inserts a control sequence;
- `\c<category><character>` (see below);
- `\u{<tl var name>}` inserts the contents of the `<tl var>` (see below).

Characters other than backslash and space are simply inserted in the result (but since the replacement text is first converted to a string, one should also escape characters that are special for TeX, for instance use `\#`). Non-alphanumeric characters can always be safely escaped with a backslash.

For instance,

```
\tl_set:Nn \l_my_t1 { Hello,~world! }
\regex_replace_all:nnN { ([er]?1|o) . } { (\O--\1) } \l_my_t1
```

results in `\l_my_t1` holding `H(e1l--el)(o,--o) w(or--o)(ld--l)!`

The submatches are numbered according to the order in which the opening parenthesis of capturing groups appear in the regular expression to match. The n -th submatch is empty if there are fewer than n capturing groups or for capturing groups that appear in alternatives that were not used for the match. In case a capturing group matches several times during a match (due to quantifiers) only the last match is used in the replacement text. Submatches always keep the same category codes as in the original token list.

The characters inserted by the replacement have category code 12 (other) by default, with the exception of space characters. Spaces inserted through `_` have category code 10, while spaces inserted through `\x20` or `\x{20}` have category code 12. The escape sequence `\c` allows to insert characters with arbitrary category codes, as well as control sequences.

`\cX(...)` Produces the characters “...” with category X , which must be one of CBEMTPUDSLOA as in regular expressions. Parentheses are optional for a single character (which can be an escape sequence). When nested, the innermost category code applies, for instance `\cL(Hello\cS\ world)!` gives this text with standard category codes.

`\c{<text>}` Produces the control sequence with csname `<text>`. The `<text>` may contain references to the submatches `\0`, `\1`, and so on, as in the example for `\u` below.

The escape sequence `\u{<tl var name>}` allows to insert the contents of the token list with name `<tl var name>` directly into the replacement, giving an easier control of category codes. When nested in `\c{...}` and `\u{...}` constructions, the `\u` and `\c` escape sequences perform `\tl_to_str:v`, namely extract the value of the control sequence and turn it into a string. Matches can also be used within the arguments of `\c` and `\u`. For instance,

```
\tl_set:Nn \l_my_one_t1 { first }
\tl_set:Nn \l_my_two_t1 { \emph{second} }
\tl_set:Nn \l_my_t1 { one , two , one , one }
\regex_replace_all:nnN { [^,]+ } { \u{\l_my_\0_t1} } \l_my_t1
```

results in `\l_my_t1` holding `first,\emph{second},first,first.`

3 Pre-compiling regular expressions

If a regular expression is to be used several times, it is better to compile it once rather than doing it each time the regular expression is used. The compiled regular expression is stored in a variable. All of the `l3regex` module's functions can be given their regular expression argument either as an explicit string or as a compiled regular expression.

`\regex_new:N`

New: 2017-05-26

`\regex_new:N <regex var>`

Creates a new `<regex var>` or raises an error if the name is already taken. The declaration is global. The `<regex var>` is initially such that it never matches.

`\regex_set:Nn`
`\regex_gset:Nn`
`\regex_const:Nn`

New: 2017-05-26

`\regex_set:Nn <regex var> {{<regex>}}`

Stores a compiled version of the `<regular expression>` in the `<regex var>`. For instance, this function can be used as

`\regex_new:N \l_my_regex`

`\regex_set:Nn \l_my_regex { my\ (simple\)? reg(ex|ular\ expression) }`

The assignment is local for `\regex_set:Nn` and global for `\regex_gset:Nn`. Use `\regex_const:Nn` for compiled expressions which never change.

`\regex_show:n`
`\regex_show:N`

New: 2017-05-26

`\regex_show:n {{<regex>}}`

Shows how `l3regex` interprets the `<regex>`. For instance, `\regex_show:n {{\A X|Y}}` shows

`+--branch`

`anchor at start (\A)`

`char code 88`

`+--branch`

`char code 89`

indicating that the anchor `\A` only applies to the first branch: the second branch is not anchored to the beginning of the match.

4 Matching

All regular expression functions are available in both `:n` and `:N` variants. The former require a “standard” regular expression, while the latter require a compiled expression as generated by `\regex_(g)set:Nn`.

`\regex_match:nnTF`
`\regex_match:NnTF`

New: 2017-05-26

`\regex_match:nnTF {{<regex>}} {{<token list>}} {{<true code>}} {{<false code>}}`

Tests whether the `<regular expression>` matches any part of the `<token list>`. For instance,

`\regex_match:nnTF { b [cde]* } { abecdcx } { TRUE } { FALSE }`

`\regex_match:nnTF { [b-dq-w] } { example } { TRUE } { FALSE }`

leaves `TRUE` then `FALSE` in the input stream.

```
\regex_count:nnN
\regex_count:NnN
```

New: 2017-05-26

```
\regex_count:nnN {<regex>} {<token list>} <int var>
```

Sets $\langle \text{int var} \rangle$ within the current TeX group level equal to the number of times $\langle \text{regular expression} \rangle$ appears in $\langle \text{token list} \rangle$. The search starts by finding the left-most longest match, respecting greedy and lazy (non-greedy) operators. Then the search starts again from the character following the last character of the previous match, until reaching the end of the token list. Infinite loops are prevented in the case where the regular expression can match an empty token list: then we count one match between each pair of characters. For instance,

```
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbabababb } \l_foo_int
```

results in $\backslash l_foo_int$ taking the value 5.

5 Submatch extraction

```
\regex_extract_once:nnN
\regex_extract_once:nnNTF
\regex_extract_once:NnN
\regex_extract_once:NnNTF
```

New: 2017-05-26

```
\regex_extract_once:nnN {<regex>} {<token list>} <seq var>
\regex_extract_once:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}
```

Finds the first match of the $\langle \text{regular expression} \rangle$ in the $\langle \text{token list} \rangle$. If it exists, the match is stored as the first item of the $\langle \text{seq var} \rangle$, and further items are the contents of capturing groups, in the order of their opening parenthesis. The $\langle \text{seq var} \rangle$ is assigned locally. If there is no match, the $\langle \text{seq var} \rangle$ is cleared. The testing versions insert the $\langle \text{true code} \rangle$ into the input stream if a match was found, and the $\langle \text{false code} \rangle$ otherwise.

For instance, assume that you type

```
\regex_extract_once:nnNTF { \A(La)?TeX(!*)\Z } { LaTeX!!! } \l_foo_seq
{ true } { false }
```

Then the regular expression (anchored at the start with $\backslash A$ and at the end with $\backslash Z$) must match the whole token list. The first capturing group, $(La)?$, matches La , and the second capturing group, $(!*)$, matches $!!!$. Thus, $\backslash l_foo_seq$ contains as a result the items $\{\text{LaTeX}\}\!\!\!$, $\{\text{La}\}$, and $\{!!!\}$, and the true branch is left in the input stream. Note that the n -th item of $\backslash l_foo_seq$, as obtained using $\backslash \text{seq_item}:Nn$, correspond to the submatch numbered $(n - 1)$ in functions such as $\backslash \text{regex_replace}_\text{once:nnN}$.

```
\regex_extract_all:nnN
\regex_extract_all:nnNTF
\regex_extract_all:NnN
\regex_extract_all:NnNTF
```

New: 2017-05-26

```
\regex_extract_all:nnN {<regex>} {<token list>} <seq var>
\regex_extract_all:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}
```

Finds all matches of the $\langle \text{regular expression} \rangle$ in the $\langle \text{token list} \rangle$, and stores all the submatch information in a single sequence (concatenating the results of multiple $\backslash \text{regex_extract}_\text{once:nnN}$ calls). The $\langle \text{seq var} \rangle$ is assigned locally. If there is no match, the $\langle \text{seq var} \rangle$ is cleared. The testing versions insert the $\langle \text{true code} \rangle$ into the input stream if a match was found, and the $\langle \text{false code} \rangle$ otherwise. For instance, assume that you type

```
\regex_extract_all:nnNTF { \w+ } { Hello,~world! } \l_foo_seq
{ true } { false }
```

Then the regular expression matches twice, the resulting sequence contains the two items $\{\text{Hello}\}$ and $\{\text{world}\}$, and the true branch is left in the input stream.

```
\regex_split:nnN
\regex_split:nnNTF
\regex_split:NnN
\regex_split:NnNTF
```

New: 2017-05-26

```
\regex_split:nnN {<regular expression>} {<token list>} <seq var>
\regex_split:nnNTF {<regular expression>} {<token list>} <seq var> {{true code}}
{{false code}}
```

Splits the *<token list>* into a sequence of parts, delimited by matches of the *<regular expression>*. If the *<regular expression>* has capturing groups, then the token lists that they match are stored as items of the sequence as well. The assignment to *<seq var>* is local. If no match is found the resulting *<seq var>* has the *<token list>* as its sole item. If the *<regular expression>* matches the empty token list, then the *<token list>* is split into single tokens. The testing versions insert the *<true code>* into the input stream if a match was found, and the *<false code>* otherwise. For example, after

```
\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
{ true } { false }
```

the sequence *\l_path_seq* contains the items *{the}*, *{path}*, *{for}*, *{this}*, and *{file.tex}*, and the *true* branch is left in the input stream.

6 Replacement

```
\regex_replace_once:nnN
\regex_replace_once:nnNTF
\regex_replace_once:NnN
\regex_replace_once:NnNTF
```

New: 2017-05-26

```
\regex_replace_once:nnN {<regular expression>} {{replacement}} <tl var>
\regex_replace_once:nnNTF {<regular expression>} {{replacement}} <tl var> {{true code}}
{{false code}}
```

Searches for the *<regular expression>* in the *<token list>* and replaces the first match with the *<replacement>*. The result is assigned locally to *<tl var>*. In the *<replacement>*, *\0* represents the full match, *\1* represent the contents of the first capturing group, *\2* of the second, *etc.*

```
\regex_replace_all:nnN
\regex_replace_all:nnNTF
\regex_replace_all:NnN
\regex_replace_all:NnNTF
```

New: 2017-05-26

```
\regex_replace_all:nnN {<regular expression>} {{replacement}} <tl var>
\regex_replace_all:nnNTF {<regular expression>} {{replacement}} <tl var> {{true code}}
{{false code}}
```

Replaces all occurrences of the *<regular expression>* in the *<token list>* by the *<replacement>*, where *\0* represents the full match, *\1* represent the contents of the first capturing group, *\2* of the second, *etc.* Every match is treated independently, and matches cannot overlap. The result is assigned locally to *<tl var>*.

7 Constants and variables

```
\l_tmpa_regex
\l_tmpb_regex
```

New: 2017-12-11

Scratch regex for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

```
\g_tmpa_regex
\g_tmpb_regex
```

New: 2017-12-11

Scratch regex for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

8 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Rewrite the documentation in a more ordered way, perhaps add a BNF?
Additional error-checking to come.
 - Clean up the use of messages.
 - Cleaner error reporting in the replacement phase.
 - Add tracing information.
 - Detect attempts to use back-references and other non-implemented syntax.
 - Test for the maximum register `\c_max_register_int`.
 - Find out whether the fact that `\W` and friends match the end-marker leads to bugs.
Possibly update `_regex_item_reverse:n`.
 - The empty cs should be matched by `\c{}`, not by `\c{csname.?endcsname\s?}`.
- Code improvements to come.
 - Shift arrays so that the useful information starts at position 1.
 - Only build `\c{...}` once.
 - Use arrays for the left and right state stacks when compiling a regex.
 - Should `_regex_action_free_group:n` only be used for greedy `{n,}` quantifier?
(I think not.)
 - Quantifiers for `\u` and assertions.
 - When matching, keep track of an explicit stack of `current_state` and `current_submatches`.
 - If possible, when a state is reused by the same thread, kill other subthreads.
 - Use an array rather than `\l_regex_balance_t1` to build the function `_regex_replacement_balance_one_match:n`.
 - Reduce the number of epsilon-transitions in alternatives.
 - Optimize simple strings: use less states (`abcade` should give two states, for `abc` and `ade`). [Does that really make sense?]
 - Optimize groups with no alternative.
 - Optimize states with a single `_regex_action_free:n`.
 - Optimize the use of `_regex_action_success:` by inserting it in state 2 directly instead of having an extra transition.
 - Optimize the use of `\int_step_...` functions.

- Groups don't capture within regexes for csnames; optimize and document.
- Better "show" for anchors, properties, and catcode tests.
- Does \K really need a new state for itself?
- When compiling, use a boolean `in_cs` and less magic numbers.
- Instead of checking whether the character is special or alphanumeric using its character code, check if it is special in regexes with `\cs_if_exist` tests.

The following features are likely to be implemented at some point in the future.

- General look-ahead/behind assertions.
- Regex matching on external files.
- Conditional subpatterns with look ahead/behind: "if what follows is [...], then [...]".
- `(*...)` and `(?...)` sequences to set some options.
- UTF-8 mode for pdfTEX.
- Newline conventions are not done. In particular, we should have an option for `.` not to match newlines. Also, `\A` should differ from `^`, and `\Z`, `\z` and `$` should differ.
- Unicode properties: `\p{...}` and `\P{...}`; `\X` which should match any "extended" Unicode sequence. This requires to manipulate a lot of data, probably using tree-boxes.
- Provide a syntax such as `\ur{l_my_regex}` to use an already-compiled regex in a more complicated regex. This makes regexes more easily composable.
- Allowing `\u{l_my_t1}` in more places, for instance as the number of repetitions in a quantifier.

The following features of PCRE or Perl may or may not be implemented.

- Callout with `(?C...)` or other syntax: some internal code changes make that possible, and it can be useful for instance in the replacement code to stop a regex replacement when some marker has been found; this raises the question of a potential `\regex_break`: and then of playing well with `\t1_map_break`: called from within the code in a regex. It also raises the question of nested calls to the regex machinery, which is a problem since `\fontdimen` are global.
- Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn't it?
- Named subpatterns: TeX programmers have lived so far without any need for named macro parameters.

The following features of PCRE or Perl will definitely not be implemented.

- Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.

- Recursion: this is a non-regular feature.
- Atomic grouping, possessive quantifiers: those tools, mostly meant to fix catastrophic backtracking, are unnecessary in a non-backtracking algorithm, and difficult to implement.
- Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking algorithm, in particular because the corresponding group should be treated as atomic.
- Backtracking control verbs: intrinsically tied to backtracking.
- `\ddd`, matching the character with octal code `ddd`: we already have `\x{...}` and the syntax is confusingly close to what we could have used for backreferences (`\1`, `\2`, ...), making it harder to produce useful error message.
- `\cx`, similar to `\TeX`'s own `\^{\^}x`.
- Comments: `\TeX` already has its own system for comments.
- `\Q...\\E` escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- `\C` single byte in UTF-8 mode: `Xe\TeX` and `Lua\TeX` serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

Part XXIX

The **I3box** package

Boxes

There are three kinds of box operations: horizontal mode denoted with prefix `\hbox_`, vertical mode with prefix `\vbox_`, and the generic operations working in both modes with prefix `\box_`.

1 Creating and initialising boxes

`\box_new:N` `\box_new:c`

Creates a new `<box>` or raises an error if the name is already taken. The declaration is global. The `<box>` is initially void.

`\box_clear:N`
`\box_clear:c`
`\box_gclear:N`
`\box_gclear:c`

`\box_clear_new:N` `\box_clear_new:c`
`\box_gclear_new:N`
`\box_gclear_new:c`

Ensures that the `<box>` exists globally by applying `\box_new:N` if necessary, then applies `\box_(g)clear:N` to leave the `<box>` empty.

`\box_set_eq:NN`
`\box_set_eq:(cN|Nc|cc)`
`\box_gset_eq:NN`
`\box_gset_eq:(cN|Nc|cc)`

`\box_set_eq:NN` `\box_set_eq:(cN|Nc|cc)`

Sets the content of `<box1>` equal to that of `<box2>`.

`\box_if_exist_p:N` *
`\box_if_exist_p:c` *

`\box_if_exist_p:N` `\box_if_exist_p:c`

Tests whether the `<box>` is currently defined. This does not check that the `<box>` really is a box.

New: 2012-03-03

2 Using boxes

`\box_use:N` `\box_use:c`

Inserts the current content of the `<box>` onto the current list for typesetting. An error is raised if the variable does not exist or if it is invalid.

TeXhackers note: This is the TeX primitive `\copy`.

```
\box_move_right:nn  
\box_move_left:nn
```

```
\box_move_right:nn {<dimexpr>} {<box function>}
```

This function operates in vertical mode, and inserts the material specified by the *<box function>* such that its reference point is displaced horizontally by the given *<dimexpr>* from the reference point for typesetting, to the right or left as appropriate. The *<box function>* should be a box operation such as `\box_use:N \<box>` or a “raw” box specification such as `\vbox:n { xyz }`.

```
\box_move_up:nn  
\box_move_down:nn
```

```
\box_move_up:nn {<dimexpr>} {<box function>}
```

This function operates in horizontal mode, and inserts the material specified by the *<box function>* such that its reference point is displaced vertically by the given *<dimexpr>* from the reference point for typesetting, up or down as appropriate. The *<box function>* should be a box operation such as `\box_use:N \<box>` or a “raw” box specification such as `\vbox:n { xyz }`.

3 Measuring and setting box dimensions

```
\box_dp:N  
\box_dp:c
```

```
\box_dp:N <box>
```

Calculates the depth (below the baseline) of the *<box>* in a form suitable for use in a *(dimension expression)*.

TeXhackers note: This is the TeX primitive `\dp`.

```
\box_ht:N  
\box_ht:c
```

```
\box_ht:N <box>
```

Calculates the height (above the baseline) of the *<box>* in a form suitable for use in a *(dimension expression)*.

TeXhackers note: This is the TeX primitive `\ht`.

```
\box_wd:N  
\box_wd:c
```

```
\box_wd:N <box>
```

Calculates the width of the *<box>* in a form suitable for use in a *(dimension expression)*.

TeXhackers note: This is the TeX primitive `\wd`.

```
\box_set_dp:Nn  
\box_set_dp:cn  
\box_gset_dp:Nn  
\box_gset_dp:cn
```

Updated: 2019-01-22

```
\box_set_dp:Nn <box> {<dimension expression>}
```

Set the depth (below the baseline) of the *<box>* to the value of the *{<dimension expression>}*.

```
\box_set_ht:Nn  
\box_set_ht:cn  
\box_gset_ht:Nn  
\box_gset_ht:cn
```

Updated: 2019-01-22

```
\box_set_ht:Nn <box> {<dimension expression>}
```

Set the height (above the baseline) of the *<box>* to the value of the *{<dimension expression>}*.

```
\box_set_wd:Nn
\box_set_wd:cn
\box_gset_wd:Nn
\box_gset_wd:cn
```

Updated: 2019-01-22

```
\box_set_wd:Nn <box> {<dimension expression>}
```

Set the width of the *<box>* to the value of the *{<dimension expression>}*.

```
\box_if_empty_p:N *
\box_if_empty_p:c *
\box_if_empty:NTF *
\box_if_empty:cTF *
```

```
\box_if_empty_p:N <box>
```

```
\box_if_empty:NTF <box> {<true code>} {<false code>}
```

Tests if *<box>* is a empty (equal to `\c_empty_box`).

```
\box_if_horizontal_p:N *
\box_if_horizontal_p:c *
\box_if_horizontal:NTF *
\box_if_horizontal:cTF *
```

```
\box_if_horizontal_p:N <box>
```

```
\box_if_horizontal:NTF <box> {<true code>} {<false code>}
```

Tests if *<box>* is a horizontal box.

```
\box_if_vertical_p:N *
\box_if_vertical_p:c *
\box_if_vertical:NTF *
\box_if_vertical:cTF *
```

```
\box_if_vertical_p:N <box>
```

```
\box_if_vertical:NTF <box> {<true code>} {<false code>}
```

Tests if *<box>* is a vertical box.

5 The last box inserted

```
\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c
```

```
\box_set_to_last:N <box>
```

Sets the *<box>* equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the *<box>* is always void as it is not possible to recover the last added item.

6 Constant boxes

```
\c_empty_box
```

Updated: 2012-11-04

This is a permanently empty box, which is neither set as horizontal nor vertical.

TeXhackers note: At the TeX level this is a void box.

7 Scratch boxes

\l_tmpa_box
\l_tmpb_box

Updated: 2012-11-04

Scratch boxes for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_box
\g_tmpb_box

Scratch boxes for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

8 Viewing box contents

\box_show:N
\box_show:c

Updated: 2012-05-11

\box_show:N <box>

Shows full details of the content of the <box> in the terminal.

\box_show:Nnn
\box_show:cnn

New: 2012-05-11

\box_show:Nnn <box> {\<intexpr₁>} {\<intexpr₂>}

Display the contents of <box> in the terminal, showing the first <intexpr₁> items of the box, and descending into <intexpr₂> group levels.

\box_log:N
\box_log:c

New: 2012-05-11

\box_log:N <box>

Writes full details of the content of the <box> to the log.

\box_log:Nnn
\box_log:cnn

New: 2012-05-11

\box_log:Nnn <box> {\<intexpr₁>} {\<intexpr₂>}

Writes the contents of <box> to the log, showing the first <intexpr₁> items of the box, and descending into <intexpr₂> group levels.

9 Boxes and color

All L^AT_EX3 boxes are “color safe”: a color set inside the box stops applying after the end of the box has occurred.

10 Horizontal mode boxes

\hbox:n

Updated: 2017-04-05

\hbox:n {\<contents>}

Typesets the <contents> into a horizontal box of natural width and then includes this box in the current list for typesetting.

`\hbox_to_wd:nn`

Updated: 2017-04-05

`\hbox_to_wd:nn {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the *⟨contents⟩* into a horizontal box of width *⟨dimexpr⟩* and then includes this box in the current list for typesetting.

`\hbox_to_zero:n`

Updated: 2017-04-05

`\hbox_to_zero:n {⟨contents⟩}`

Typesets the *⟨contents⟩* into a horizontal box of zero width and then includes this box in the current list for typesetting.

`\hbox_set:Nn`

`\hbox_set:cN`

`\hbox_gset:Nn`

`\hbox_gset:cN`

Updated: 2017-04-05

`\hbox_set:Nn ⟨box⟩ {⟨contents⟩}`

Typesets the *⟨contents⟩* at natural width and then stores the result inside the *⟨box⟩*.

`\hbox_set_to_wd:Nnn`

`\hbox_set_to_wd:cnn`

`\hbox_gset_to_wd:Nnn`

`\hbox_gset_to_wd:cnn`

Updated: 2017-04-05

`\hbox_set_to_wd:Nnn ⟨box⟩ {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the *⟨contents⟩* to the width given by the *⟨dimexpr⟩* and then stores the result inside the *⟨box⟩*.

`\hbox_overlap_center:n`

New: 2020-08-25

`\hbox_overlap_center:n {⟨contents⟩}`

Typesets the *⟨contents⟩* into a horizontal box of zero width such that material protrudes equally to both sides of the insertion point.

`\hbox_overlap_right:n`

Updated: 2017-04-05

`\hbox_overlap_right:n {⟨contents⟩}`

Typesets the *⟨contents⟩* into a horizontal box of zero width such that material protrudes to the right of the insertion point.

`\hbox_overlap_left:n`

Updated: 2017-04-05

`\hbox_overlap_left:n {⟨contents⟩}`

Typesets the *⟨contents⟩* into a horizontal box of zero width such that material protrudes to the left of the insertion point.

`\hbox_set:Nw`

`\hbox_set:cw`

`\hbox_set_end:`

`\hbox_gset:Nw`

`\hbox_gset:cw`

`\hbox_gset_end:`

Updated: 2017-04-05

`\hbox_set:Nw ⟨box⟩ ⟨contents⟩ \hbox_set_end:`

Typesets the *⟨contents⟩* at natural width and then stores the result inside the *⟨box⟩*. In contrast to `\hbox_set:Nn` this function does not absorb the argument when finding the *⟨content⟩*, and so can be used in circumstances where the *⟨content⟩* may not be a simple argument.

`\hbox_set_to_wd:Nnw`

`\hbox_set_to_wd:cnw`

`\hbox_gset_to_wd:Nnw`

`\hbox_gset_to_wd:cnw`

New: 2017-06-08

`\hbox_set_to_wd:Nnw ⟨box⟩ {⟨dimexpr⟩} {⟨contents⟩} \hbox_set_end:`

Typesets the *⟨contents⟩* to the width given by the *⟨dimexpr⟩* and then stores the result inside the *⟨box⟩*. In contrast to `\hbox_set_to_wd:Nnn` this function does not absorb the argument when finding the *⟨content⟩*, and so can be used in circumstances where the *⟨content⟩* may not be a simple argument

`\hbox_unpack:N`
`\hbox_unpack:c`

`\hbox_unpack:N <box>`

Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

TeXhackers note: This is the TeX primitive `\unhcopy`.

11 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box has no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are `_top` boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter is typically non-zero.

`\vbox:n`

Updated: 2017-04-05

`\vbox:n {\langle contents \rangle}`

Typesets the $\langle contents \rangle$ into a vertical box of natural height and includes this box in the current list for typesetting.

`\vbox_top:n`

Updated: 2017-04-05

`\vbox_top:n {\langle contents \rangle}`

Typesets the $\langle contents \rangle$ into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box is equal to that of the *first* item added to the box.

`\vbox_to_ht:nn`

Updated: 2017-04-05

`\vbox_to_ht:nn {\langle dimexpr \rangle} {\langle contents \rangle}`

Typesets the $\langle contents \rangle$ into a vertical box of height $\langle dimexpr \rangle$ and then includes this box in the current list for typesetting.

`\vbox_to_zero:n`

Updated: 2017-04-05

`\vbox_to_zero:n {\langle contents \rangle}`

Typesets the $\langle contents \rangle$ into a vertical box of zero height and then includes this box in the current list for typesetting.

`\vbox_set:Nn`

`\vbox_set:cn`

`\vbox_gset:Nn`

`\vbox_gset:cn`

Updated: 2017-04-05

`\vbox_set:Nn <box> {\langle contents \rangle}`

Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$.

`\vbox_set_top:Nn`

`\vbox_set_top:cn`

`\vbox_gset_top:Nn`

`\vbox_gset_top:cn`

Updated: 2017-04-05

`\vbox_set_top:Nn <box> {\langle contents \rangle}`

Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. The baseline of the box is equal to that of the *first* item added to the box.

```
\vbox_set_to_ht:Nnn
\vbox_set_to_ht:cnn
\vbox_gset_to_ht:Nnn
\vbox_gset_to_ht:cnn
```

Updated: 2017-04-05

```
\vbox_set:Nw <box> {<dimexpr>} {<contents>}
```

Typesets the *<contents>* to the height given by the *<dimexpr>* and then stores the result inside the *<box>*.

```
\vbox_set:Nw
\vbox_set:cw
\vbox_set_end:
\vbox_gset:Nw
\vbox_gset:cw
\vbox_gset_end:
```

Updated: 2017-04-05

```
\vbox_set:Nw <box> {<contents>} \vbox_set_end:
```

Typesets the *<contents>* at natural height and then stores the result inside the *<box>*. In contrast to `\vbox_set:Nn` this function does not absorb the argument when finding the *<content>*, and so can be used in circumstances where the *<content>* may not be a simple argument.

```
\vbox_set_to_ht:Nnw
\vbox_set_to_ht:cnw
\vbox_gset_to_ht:Nnw
\vbox_gset_to_ht:cnw
```

New: 2017-06-08

```
\vbox_set_to_ht:Nnw <box> {<dimexpr>} {<contents>} \vbox_set_end:
```

Typesets the *<contents>* to the height given by the *<dimexpr>* and then stores the result inside the *<box>*. In contrast to `\vbox_set_to_ht:Nnn` this function does not absorb the argument when finding the *<content>*, and so can be used in circumstances where the *<content>* may not be a simple argument

```
\vbox_set_split_to_ht>NNn
\vbox_set_split_to_ht:(cNn|Ncn|ccn)
\vbox_gset_split_to_ht>NNn
\vbox_gset_split_to_ht:(cNn|Ncn|ccn)
```

Updated: 2018-12-29

```
\vbox_set_split_to_ht>NNn <box1> <box2> {<dimexpr>}
```

Sets *<box₁>* to contain material to the height given by the *<dimexpr>* by removing content from the top of *<box₂>* (which must be a vertical box).

```
\vbox_unpack:N <box>
\vbox_unpack:c
```

Unpacks the content of the vertical *<box>*, retaining any stretching or shrinking applied when the *<box>* was set.

TeXhackers note: This is the TeX primitive `\unvcopy`.

12 Using boxes efficiently

The functions above for using box contents work in exactly the same way as for any other `expl3` variable. However, for efficiency reasons, it is also useful to have functions which *drop* box contents on use. When a box is dropped, the box becomes empty at the group level *where the box was originally set* rather than necessarily *at the current group level*. For example, with

```
\hbox_set:Nn \l_tmpa_box { A }
\group_begin:
\hbox_set:Nn \l_tmpa_box { B }
\group_begin:
```

```

\box_use_drop:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box

```

the first use of `\box_show:N` will show an entirely cleared (void) box, and the second will show the letter A in the box.

These functions should be preferred when the content of the box is no longer required after use. Note that due to the unusual scoping behaviour of `drop` functions they may be applied to both local and global boxes: the latter will naturally be set and thus cleared at a global level.

`\box_use_drop:N`
`\box_use_drop:c`

`\box_use_drop:N <box>`

Inserts the current content of the `<box>` onto the current list for typesetting then drops the box content. An error is raised if the variable does not exist or if it is invalid. This function may be applied to local or global boxes.

TeXhackers note: This is the `\box` primitive.

`\box_set_eq_drop:NN`
`\box_set_eq_drop:(cN|Nc|cc)`

New: 2019-01-17

`\box_set_eq_drop:NN <box1> <box2>`

Sets the content of `<box1>` equal to that of `<box2>`, then drops `<box2>`.

`\box_gset_eq_drop:NN`
`\box_gset_eq_drop:(cN|Nc|cc)`

New: 2019-01-17

`\box_gset_eq_drop:NN <box1> <box2>`

Sets the content of `<box1>` globally equal to that of `<box2>`, then drops `<box2>`.

`\hbox_unpack_drop:N`
`\hbox_unpack_drop:c`

New: 2019-01-17

`\hbox_unpack_drop:N <box>`

Unpacks the content of the horizontal `<box>`, retaining any stretching or shrinking applied when the `<box>` was set. The original `<box>` is then dropped.

TeXhackers note: This is the TeX primitive `\unhbox`.

`\vbox_unpack_drop:N`
`\vbox_unpack_drop:c`

New: 2019-01-17

`\vbox_unpack_drop:N <box>`

Unpacks the content of the vertical `<box>`, retaining any stretching or shrinking applied when the `<box>` was set. The original `<box>` is then dropped.

TeXhackers note: This is the TeX primitive `\unvbox`.

13 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in TeX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing

of boxed material can best be handled by modifying boxes. These transformations are described here.

```
\box_autosize_to_wd_and_ht:Nnn      \box_autosize_to_wd_and_ht:Nnn <box> {\<x-size>} {\<y-size>}  
\box_autosize_to_wd_and_ht:cnn  
\box_gautosize_to_wd_and_ht:Nnn  
\box_gautosize_to_wd_and_ht:cnn
```

New: 2017-04-04

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to fit within the given $\langle x\text{-size} \rangle$ (horizontally) and $\langle y\text{-size} \rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y\text{-size} \rangle$ is the height only: it does not include any depth. The updated $\langle box \rangle$ is an **hbox**, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. The final size of the $\langle box \rangle$ is the smaller of $\{\langle x\text{-size} \rangle\}$ and $\{\langle y\text{-size} \rangle\}$, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y\text{-size} \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_autosize_to_wd_and_ht_plus_dp:Nnn    \box_autosize_to_wd_and_ht_plus_dp:Nnn <box> {\<x-size>}  
\box_autosize_to_wd_and_ht_plus_dp:cnn     {\<y-size>}  
\box_gautosize_to_wd_and_ht_plus_dp:Nnn  
\box_gautosize_to_wd_and_ht_plus_dp:cnn
```

New: 2017-04-04

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to fit within the given $\langle x\text{-size} \rangle$ (horizontally) and $\langle y\text{-size} \rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y\text{-size} \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an **hbox**, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. The final size of the $\langle box \rangle$ is the smaller of $\{\langle x\text{-size} \rangle\}$ and $\{\langle y\text{-size} \rangle\}$, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y\text{-size} \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_resize_to_ht:Nn <box> {\<y-size>}
```

Resizes the $\langle box \rangle$ to $\langle y\text{-size} \rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y\text{-size} \rangle$ is a dimension expression. The $\langle y\text{-size} \rangle$ is the height only: it does not include any depth. The updated $\langle box \rangle$ is an **hbox**, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle y\text{-size} \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y\text{-size} \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_resize_to_ht_plus_dp:Nn      \box_resize_to_ht_plus_dp:Nn <box> {<y-size>}
\box_resize_to_ht_plus_dp:cn
\box_gresize_to_ht_plus_dp:Nn
\box_gresize_to_ht_plus_dp:cn
```

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle y\text{-size} \rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y\text{-size} \rangle$ is a dimension expression. The $\langle y\text{-size} \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle y\text{-size} \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y\text{-size} \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_resize_to_wd:Nn      \box_resize_to_wd:Nn <box> {<x-size>}
```

Resizes the $\langle box \rangle$ to $\langle x\text{-size} \rangle$ (horizontally), scaling the vertical size by the same amount; $\langle x\text{-size} \rangle$ is a dimension expression. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle x\text{-size} \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle x\text{-size} \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_resize_to_wd_and_ht:Nnn    \box_resize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}
```

```
\box_resize_to_wd_and_ht:cnn
```

```
\box_gresize_to_wd_and_ht:Nnn
```

```
\box_gresize_to_wd_and_ht:cnn
```

New: 2014-07-03

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle x\text{-size} \rangle$ (horizontally) and $\langle y\text{-size} \rangle$ (vertically): both of the sizes are dimension expressions. The $\langle y\text{-size} \rangle$ is the height only and does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y\text{-size} \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_resize_to_wd_and_ht_plus_dp:Nnn   \box_resize_to_wd_and_ht_plus_dp:Nnn <box> {<x-size>} {<y-size>}
```

```
\box_resize_to_wd_and_ht_plus_dp:cnn
```

```
\box_gresize_to_wd_and_ht_plus_dp:Nnn
```

```
\box_gresize_to_wd_and_ht_plus_dp:cnn
```

New: 2017-04-06

Updated: 2019-01-22

Resizes the $\langle box \rangle$ to $\langle x\text{-size} \rangle$ (horizontally) and $\langle y\text{-size} \rangle$ (vertically): both of the sizes are dimension expressions. The $\langle y\text{-size} \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y\text{-size} \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_rotate:Nn
\box_rotate:cn
\box_grotate:Nn
\box_grotate:cn
```

Updated: 2019-01-22

```
\box_scale:Nnn
\box_scale:cnn
\box_gscale:Nnn
\box_gscale:cnn
```

Updated: 2019-01-22

```
\box_rotate:Nn <box> {<angle>}
```

Rotates the $\langle box \rangle$ by $\langle angle \rangle$ (in degrees) anti-clockwise about its reference point. The reference point of the updated box is moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the rotation is applied.

```
\box_scale:Nnn <box> {<x-scale>} {<y-scale>}
```

Scales the $\langle box \rangle$ by factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the scaling is applied. Negative scalings cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-scale \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

14 Primitive box conditionals

```
\if_hbox:N *
  \if_hbox:N <box>
    <true code>
  \else:
    <false code>
  \fi:
```

Tests is $\langle box \rangle$ is a horizontal box.

TEXhackers note: This is the TeX primitive `\ifhbox`.

```
\if_vbox:N *
  \if_vbox:N <box>
    <true code>
  \else:
    <false code>
  \fi:
```

Tests is $\langle box \rangle$ is a vertical box.

TEXhackers note: This is the TeX primitive `\ifvbox`.

```
\if_box_empty:N *
  \if_box_empty:N <box>
    <true code>
  \else:
    <false code>
  \fi:
```

Tests is $\langle box \rangle$ is an empty (void) box.

TEXhackers note: This is the TeX primitive `\ifvoid`.

Part XXX

The **l3coffins** package

Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the `xcoffins` module (in the `l3experimental` bundle).

1 Creating and initialising coffins

`\coffin_new:N`
`\coffin_new:c`
New: 2011-08-17

`\coffin_new:N <coffin>`

Creates a new `<coffin>` or raises an error if the name is already taken. The declaration is global. The `<coffin>` is initially empty.

`\coffin_clear:N`
`\coffin_clear:c`
`\coffin_gclear:N`
`\coffin_gclear:c`
New: 2011-08-17
Updated: 2019-01-21

`\coffin_clear:N <coffin>`

Clears the content of the `<coffin>`.

`\coffin_set_eq:NN`
`\coffin_set_eq:(Nc|cN|cc)`
`\coffin_gset_eq:NN`
`\coffin_gset_eq:(Nc|cN|cc)`

New: 2011-08-17
Updated: 2019-01-21

`\coffin_set_eq:NN <coffin1> <coffin2>`

Sets both the content and poles of `<coffin1>` equal to those of `<coffin2>`.

`\coffin_if_exist_p:N *`
`\coffin_if_exist_p:c *`
`\coffin_if_exist:NTF *`
`\coffin_if_exist:cTF *`
New: 2012-06-20

`\coffin_if_exist_p:N <box>`

`\coffin_if_exist:NTF <box> {\<true code>} {\<false code>}`

Tests whether the `<coffin>` is currently defined.

`\hcoffin_set:Nn`
`\hcoffin_set:cn`
`\hcoffin_gset:Nn`
`\hcoffin_gset:cn`
New: 2011-08-17
Updated: 2019-01-21

`\hcoffin_set:Nn <coffin> {\<material>}`

Typesets the `<material>` in horizontal mode, storing the result in the `<coffin>`. The standard poles for the `<coffin>` are then set up based on the size of the typeset material.

```
\hcoffin_set:Nw
\hcoffin_set:cw
\hcoffin_set_end:
\hcoffin_gset:Nw
\hcoffin_gset:cw
\hcoffin_gset_end:
```

New: 2011-09-10

Updated: 2019-01-21

```
\hcoffin_set:Nw <coffin> <material> \hcoffin_set_end:
```

Typesets the *<material>* in horizontal mode, storing the result in the *<coffin>*. The standard poles for the *<coffin>* are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

```
\vcoffin_set:Nnn
\vcoffin_set:cnn
\vcoffin_gset:Nnn
\vcoffin_gset:cnn
```

New: 2011-08-17

Updated: 2019-01-21

```
\vcoffin_set:Nnn <coffin> {\<width>} {\<material>}
```

Typesets the *<material>* in vertical mode constrained to the given *<width>* and stores the result in the *<coffin>*. The standard poles for the *<coffin>* are then set up based on the size of the typeset material.

```
\vcoffin_set:Nnw
\vcoffin_set:cnw
\vcoffin_set_end:
\vcoffin_gset:Nnw
\vcoffin_gset:cnw
\vcoffin_gset_end:
```

New: 2011-09-10

Updated: 2019-01-21

```
\vcoffin_set:Nnw <coffin> {\<width>} <material> \vcoffin_set_end:
```

Typesets the *<material>* in vertical mode constrained to the given *<width>* and stores the result in the *<coffin>*. The standard poles for the *<coffin>* are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

```
\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn
\coffin_gset_horizontal_pole:Nnn
\coffin_gset_horizontal_pole:cnn
```

```
\coffin_set_horizontal_pole:Nnn <coffin>
{\<pole>} {\<offset>}
```

New: 2012-07-20

Updated: 2019-01-21

Sets the *<pole>* to run horizontally through the *<coffin>*. The *<pole>* is placed at the *<offset>* from the bottom edge of the bounding box of the *<coffin>*. The *<offset>* should be given as a dimension expression.

```
\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn
\coffin_gset_vertical_pole:Nnn
\coffin_gset_vertical_pole:cnn
```

```
\coffin_set_vertical_pole:Nnn <coffin> {\<pole>} {\<offset>}
```

New: 2012-07-20

Updated: 2019-01-21

Sets the *<pole>* to run vertically through the *<coffin>*. The *<pole>* is placed at the *<offset>* from the left-hand edge of the bounding box of the *<coffin>*. The *<offset>* should be given as a dimension expression.

3 Coffin affine transformations

```
\coffin_resize:Nnn
\coffin_resize:cnn
\coffin_gresize:Nnn
\coffin_gresize:cnn
```

Updated: 2019-01-23

```
\coffin_rotate:Nn
\coffin_rotate:cn
\coffin_grotate:Nn
\coffin_grotate:cn
```

```
\coffin_scale:Nnn
\coffin_scale:cnn
\coffin_gscale:Nnn
\coffin_gscale:cnn
```

Updated: 2019-01-23

`\coffin_resize:Nnn` $\langle \text{coffin} \rangle$ $\{\langle \text{width} \rangle\}$ $\{\langle \text{total-height} \rangle\}$

Resized the $\langle \text{coffin} \rangle$ to $\langle \text{width} \rangle$ and $\langle \text{total-height} \rangle$, both of which should be given as dimension expressions.

`\coffin_rotate:Nn` $\langle \text{coffin} \rangle$ $\{\langle \text{angle} \rangle\}$

Rotates the $\langle \text{coffin} \rangle$ by the given $\langle \text{angle} \rangle$ (given in degrees counter-clockwise). This process rotates both the coffin content and poles. Multiple rotations do not result in the bounding box of the coffin growing unnecessarily.

`\coffin_scale:Nnn` $\langle \text{coffin} \rangle$ $\{\langle \text{x-scale} \rangle\}$ $\{\langle \text{y-scale} \rangle\}$

Scales the $\langle \text{coffin} \rangle$ by a factors $\langle \text{x-scale} \rangle$ and $\langle \text{y-scale} \rangle$ in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

```
\coffin_attach:NnnNnnnn
\coffin_attach:(cnnNnnnn|Nnnccnnnn|cnncnnnn)
\coffin_gattach:NnnNnnnn
\coffin_gattach:(cnnNnnnn|Nnnccnnnn|cnncnnnn)
```

Updated: 2019-01-22

```
\coffin_attach:NnnNnnnn
\coffin_attach:{\langle \text{coffin}_1 \rangle \{\langle \text{coffin}_1-\text{pole}_1 \rangle\} \{\langle \text{coffin}_1-\text{pole}_2 \rangle\}}
\coffin_attach:{\langle \text{coffin}_2 \rangle \{\langle \text{coffin}_2-\text{pole}_1 \rangle\} \{\langle \text{coffin}_2-\text{pole}_2 \rangle\}}
\{\langle \text{x-offset} \rangle\} \{\langle \text{y-offset} \rangle\}
```

This function attaches $\langle \text{coffin}_2 \rangle$ to $\langle \text{coffin}_1 \rangle$ such that the bounding box of $\langle \text{coffin}_1 \rangle$ is not altered, i.e. $\langle \text{coffin}_2 \rangle$ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating $\langle \text{handle}_1 \rangle$, the point of intersection of $\langle \text{coffin}_1\text{-pole}_1 \rangle$ and $\langle \text{coffin}_1\text{-pole}_2 \rangle$, and $\langle \text{handle}_2 \rangle$, the point of intersection of $\langle \text{coffin}_2\text{-pole}_1 \rangle$ and $\langle \text{coffin}_2\text{-pole}_2 \rangle$. $\langle \text{coffin}_2 \rangle$ is then attached to $\langle \text{coffin}_1 \rangle$ such that the relationship between $\langle \text{handle}_1 \rangle$ and $\langle \text{handle}_2 \rangle$ is described by the $\langle \text{x-offset} \rangle$ and $\langle \text{y-offset} \rangle$. The two offsets should be given as dimension expressions.

```
\coffin_join:NnnNnnnn
\coffin_join:(cnnNnnnn|Nnnccnnnn|cnncnnnn)
\coffin_gjoin:NnnNnnnn
\coffin_gjoin:(cnnNnnnn|Nnnccnnnn|cnncnnnn)
```

Updated: 2019-01-22

```
\coffin_join:NnnNnnnn
\coffin_join:{\langle \text{coffin}_1 \rangle \{\langle \text{coffin}_1-\text{pole}_1 \rangle\} \{\langle \text{coffin}_1-\text{pole}_2 \rangle\}}
\coffin_join:{\langle \text{coffin}_2 \rangle \{\langle \text{coffin}_2-\text{pole}_1 \rangle\} \{\langle \text{coffin}_2-\text{pole}_2 \rangle\}}
\{\langle \text{x-offset} \rangle\} \{\langle \text{y-offset} \rangle\}
```

This function joins $\langle \text{coffin}_2 \rangle$ to $\langle \text{coffin}_1 \rangle$ such that the bounding box of $\langle \text{coffin}_1 \rangle$ may expand. The new bounding box covers the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating $\langle \text{handle}_1 \rangle$, the point of intersection of $\langle \text{coffin}_1\text{-pole}_1 \rangle$ and $\langle \text{coffin}_1\text{-pole}_2 \rangle$, and $\langle \text{handle}_2 \rangle$, the point of intersection of $\langle \text{coffin}_2\text{-pole}_1 \rangle$ and $\langle \text{coffin}_2\text{-pole}_2 \rangle$. $\langle \text{coffin}_2 \rangle$ is then attached to $\langle \text{coffin}_1 \rangle$ such that the relationship between $\langle \text{handle}_1 \rangle$ and $\langle \text{handle}_2 \rangle$ is described by the $\langle \text{x-offset} \rangle$ and $\langle \text{y-offset} \rangle$. The two offsets should be given as dimension expressions.

```
\coffin_typeset:Nnnnn  
\coffin_typeset:cnnnn
```

Updated: 2012-07-20

```
\coffin_typeset:Nnnnnn <coffin> {\<pole1>} {\<pole2>}  
{\<x-offset>} {\<y-offset>}
```

Typesetting is carried out by first calculating $\langle handle \rangle$, the point of intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the $\langle handle \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

5 Measuring coffins

```
\coffin_dp:N  
\coffin_dp:c
```

```
\coffin_dp:N <coffin>
```

Calculates the depth (below the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

```
\coffin_ht:N  
\coffin_ht:c
```

```
\coffin_ht:N <coffin>
```

Calculates the height (above the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

```
\coffin_wd:N  
\coffin_wd:c
```

```
\coffin_wd:N <coffin>
```

Calculates the width of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

6 Coffin diagnostics

```
\coffin_display_handles:Nn  
\coffin_display_handles:cn
```

Updated: 2011-09-02

```
\coffin_display_handles:Nn <coffin> {\<color>}
```

This function first calculates the intersections between all of the $\langle poles \rangle$ of the $\langle coffin \rangle$ to give a set of $\langle handles \rangle$. It then prints the $\langle coffin \rangle$ at the current location in the source, with the position of the $\langle handles \rangle$ marked on the coffin. The $\langle handles \rangle$ are labelled as part of this process: the locations of the $\langle handles \rangle$ and the labels are both printed in the $\langle color \rangle$ specified.

```
\coffin_mark_handle:Nnnn  
\coffin_mark_handle:cnnn
```

Updated: 2011-09-02

```
\coffin_mark_handle:Nnnn <coffin> {\<pole1>} {\<pole2>} {\<color>}
```

This function first calculates the $\langle handle \rangle$ for the $\langle coffin \rangle$ as defined by the intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. It then marks the position of the $\langle handle \rangle$ on the $\langle coffin \rangle$. The $\langle handle \rangle$ are labelled as part of this process: the location of the $\langle handle \rangle$ and the label are both printed in the $\langle color \rangle$ specified.

```
\coffin_show_structure:N  
\coffin_show_structure:c
```

Updated: 2015-08-01

```
\coffin_show_structure:N <coffin>
```

This function shows the structural information about the $\langle coffin \rangle$ in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.

Notice that the poles of a coffin are defined by four values: the x and y co-ordinates of a point that the pole passes through and the x - and y -components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.

\coffin_log_structure:N
\coffin_log_structure:c

New: 2014-08-22
Updated: 2015-08-01

\coffin_log_structure:N *(coffin)*

This function writes the structural information about the *(coffin)* in the log file. See also \coffin_show_structure:N which displays the result in the terminal.

7 Constants and variables

\c_empty_coffin

A permanently empty coffin.

\l_tmpa_coffin
\l_tmpb_coffin

New: 2012-06-19

Scratch coffins for local assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_coffin
\g_tmpb_coffin

New: 2019-01-24

Scratch coffins for global assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Part XXXI

The **l3color-base** package

Color support

This module provides support for color in L^AT_EX3. At present, the material here is mainly intended to support a small number of low-level requirements in other l3kernel modules.

1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

```
\color_group_begin:  
\color_group_end:  
New: 2011-09-03
```

```
\color_group_begin:  
...  
\color_group_end:
```

Creates a color group: one used to “trap” color settings. This grouping is built in to for example `\hbox_set:Nn`.

```
\color_ensure_current:  
New: 2011-09-03
```

```
\color_ensure_current:
```

Ensures that material inside a box uses the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin: ... \color_group_end:` group.

Part XXXII

The **l3luatex** package: LuaTeX-specific functions

The LuaTeX engine provides access to the Lua programming language, and with it access to the “internals” of TeX. In order to use this within the framework provided here, a family of functions is available. When used with pdftEX, pTeX, upTeX or XeTeX these raise an error: use `\sys_if_engine_luatex:T` to avoid this. Details on using Lua with the LuaTeX engine are given in the LuaTeX manual.

1 Breaking out to Lua

`\lua_now:n *`
`\lua_now:e *`
New: 2018-06-18

`\lua_now:n {<token list>}`

The `<token list>` is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting `<Lua input>` is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter executes the `<Lua input>` immediately, and in an expandable manner.

TeXhackers note: `\lua_now:e` is a macro wrapper around `\directlua`: when LuaTeX is in use two expansions are required to yield the result of the Lua code.

`\lua_shipout_e:n`
`\lua_shipout:n`
New: 2018-06-18

`\lua_shipout:n {<token list>}`

The `<token list>` is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting `<Lua input>` is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the `<Lua input>` during the page-building routine: no TeX expansion of the `<Lua input>` will occur at this stage.

In the case of the `\lua_shipout_e:n` version the input is fully expanded by TeX in an e-type manner during the shipout operation.

TeXhackers note: At a TeX level, the `<Lua input>` is stored as a “whatsit”.

`\lua_escape:n *`
`\lua_escape:e *`
New: 2015-06-29

`\lua_escape:n {<token list>}`

Converts the `<token list>` such that it can safely be passed to Lua: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to `\n` and `\r`, respectively.

TeXhackers note: `\lua_escape:e` is a macro wrapper around `\luaescapestring`: when LuaTeX is in use two expansions are required to yield the result of the Lua code.

2 Lua interfaces

As well as interfaces for \TeX , there are a small number of Lua functions provided here.

`ltx.utils` Most public interfaces provided by the module are stored within the `ltx.utils` table.

`l3kernel` For compatibility reasons, there are also some deprecated interfaces provided in the `l3kernel` table. These do not return their result as Lua values but instead print them to \TeX .

`l3kernel.charcat`(*charcode*, *catcode*)

Constructs a character of *charcode* and *catcode* and returns the result to \TeX .

`l3kernel.elapsedtime`

Returns the CPU time in *scaled seconds* since the start of the \TeX run or since `l3kernel.resettimer` was issued. This only measures the time used by the CPU, not the real time, e.g., waiting for user input.

`ltx.utils.fileread`
`l3kernel.fileread`

Returns the uppercase hexadecimal representation of the content of the *file* read as bytes. If the *length* is given, only this part of the file is returned; similarly, one may specify the *offset* from the start of the file. If the *length* is not given, the entire file is read starting at the *offset*.

`ltx.utils.filemd5sum`
`l3kernel.filemd5sum`

Returns the MD5 sum of the file contents read as bytes; note that the result will depend on the nature of the line endings used in the file, in contrast to normal \TeX behaviour. If the *file* is not found, nothing is returned with *no error raised*.

`ltx.utils.filemoddate`
`l3kernel.filemoddate`

Returns the date/time of last modification of the *file* in the format

`D:<year><month><day><hour><minute><second><offset>`

where the latter may be Z (UTC) or *plus-minus**hours*:*minutes*. If the *file* is not found, nothing is returned with *no error raised*.

`ltx.utils.filesize`
`l3kernel.filesize`

Returns the size of the *file* in bytes. If the *file* is not found, nothing is returned with *no error raised*.

13kernel.resettimer	<code>13kernel.resettimer()</code>
	Resets the timer used by <code>13kernel.elapsetime</code> .
13kernel.shellescape	<code>13kernel.shellescape(<cmd>)</code>
	Executes the <code><cmd></code> and prints to the log as for pdfTeX.
13kernel.strcmp	<code>13kernel.strcmp(<str one>, <str two>)</code>
	Compares the two strings and returns 0 to TeX if the two are identical.

Part XXXIII

The **I3unicode** package: Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. At present, it provides no public functions.

Part XXXIV

The **l3text** package: text processing

1 l3text documentation

This module deals with manipulation of (formatted) text; such material is comprised of a restricted set of token list content. The functions provided here concern conversion of textual content for example in case changing, generation of bookmarks and extraction to tags. All of the major functions operate by expansion. Begin-group and end-group tokens in the $\langle text \rangle$ are normalized and become $\{$ and $\}$, respectively.

1.1 Expanding text

`\text_expand:n *` `\text_expand:n {\langle text \rangle}`

New: 2020-01-02

Takes user input $\langle text \rangle$ and expands the content. Protected commands (typically formatting) are left in place, and no processing takes place of math mode material (as delimited by pairs given in `\l_text_math_delims_t1` or as the argument to commands listed in `\l_text_math_arg_t1`). Commands which are neither engine- nor L^AT_EX protected are expanded exhaustively. Any commands listed in `\l_text_expand_exclude_t1`, `\l_text_accents_t1` and `\l_text_letterlike_t1` are excluded from expansion.

`\text_declare_expand_equivalent:Nn` `\text_declare_expand_equivalent:Nn {\langle cmd \rangle {\langle replacement \rangle}}`

`\text_declare_expand_equivalent:cn`

New: 2020-01-22

Declares that the $\langle replacement \rangle$ tokens should be used whenever the $\langle cmd \rangle$ (a single token) is encountered. The $\langle replacement \rangle$ tokens should be expandable.

1.2 Case changing

```
\text_lowercase:n      *
\text_uppercase:n     *
\text_titlecase:n     *
\text_titlecase:first:n *
\text_lowercase:nn    *
\text_uppercase:nn    *
\text_titlecase:nn    *
\text_titlecase:first:nn *
```

New: 2019-11-20
Updated: 2020-02-24

```
* \text_uppercase:n  {\langle tokens \rangle}
* \text_uppercase:nn {\langle language \rangle} {\langle tokens \rangle}
```

Takes user input $\langle text \rangle$ first applies `\text_expand`, then transforms the case of character tokens as specified by the function name. The category code of letters are not changed by this process (at least where they can be represented by the engine as a single token: 8-bit engines may require active characters).

Upper- and lowercase have the obvious meanings. Titlecasing may be regarded informally as converting the first character of the $\langle tokens \rangle$ to uppercase and the rest to lowercase. However, the process is more complex than this as there are some situations where a single lowercase character maps to a special form, for example `ij` in Dutch which becomes `IJ`. The `titlecase_first` variant does not attempt any case changing at all after the first letter has been processed.

Importantly, notice that these functions are intended for working with user *text for typesetting*. For case changing programmatic data see the `l3str` module and discussion there of `\str_lowercase:n`, `\str_uppercase:n` and `\str_foldcase:n`.

Case changing does not take place within math mode material so for example

```
\text_uppercase:n { Some~text~$y = mx + c$~with~{Braces} }
```

becomes

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

The arguments of commands listed in `\l_text_case_exclude_arg_t1` are excluded from case changing; the latter are entirely non-textual content (such as labels).

As is generally true for `expl3`, these functions are designed to work with Unicode input only. As such, UTF-8 input is assumed for *all* engines. When used with Xe^{TEX} or Lua^{TEX} a full range of Unicode transformations are enabled. Specifically, the standard mappings here follow those defined by the [Unicode Consortium](#) in `UnicodeData.txt` and `SpecialCasing.txt`. In the case of 8-bit engines, mappings are provided for characters which can be represented in output typeset using the T1, T2 and LGR font encodings. Thus for example `ä` can be case-changed using pdf^{TEX}. For p^{TEX} only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Language-sensitive conversions are enabled using the $\langle language \rangle$ argument, and follow Unicode Consortium guidelines. Currently, the languages recognised for special handling are as follows.

- Azeri and Turkish (`az` and `tr`). The case pairs I/i-dotless and I-dot/i are activated for these languages. The combining dot mark is removed when lowercasing I-dot and introduced when upper casing i-dotless.
- German (`de-alt`). An alternative mapping for German in which the lowercase *Eszett* maps to a *großes Eszett*. Since there is a T1 slot for the *großes Eszett* in T1, this tailoring *is* available with pdf^{TEX} as well as in the Unicode T^{EX} engines.
- Greek (`el`). Removes accents from Greek letters when uppercasing; titlecasing leaves accents in place. (At present this is implemented only for Unicode engines.)
- Lithuanian (`lt`). The lowercase letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lowercasing of the relevant uppercase letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when uppercasing in these cases. Note that *only* the accents used in Lithuanian are covered: the behaviour of other accents are not modified.
- Dutch (`nl`). Capitalisation of `ij` at the beginning of titlecased input produces `IJ` rather than `Ij`. The output retains two separate letters, thus this transformation *is* available using pdf^{TEX}.

For titlecasing, note that there are two functions available. The function `\text_titlecase:n` applies (broadly) uppercasing to the first letter of the input, then lower-

1.3 Removing formatting from text

`\text_purify:n *` `\text_purify:n {<text>}`

New: 2020-03-05
Updated: 2020-05-14

Takes user input $\langle text \rangle$ and expands as described for `\text_expand:n`, then removes all functions from the resulting text. Math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`) is left contained in a pair of \$ delimiters. Non-expandable functions present in the $\langle text \rangle$ must either have a defined equivalent (see `\text_declare_purify-equivalent:Nn`) or will be removed from the result. Implicit tokens are converted to their explicit equivalent.

`\text_declare_purify_equivalent:Nn` `\text_declare_purify_equivalent:Nn {<cmd>} {<replacement>}`
`\text_declare_purify_equivalent:Nx`

New: 2020-03-05

Declares that the $\langle replacement \rangle$ tokens should be used whenever the $\langle cmd \rangle$ (a single token) is encountered. The $\langle replacement \rangle$ tokens should be expandable.

1.4 Control variables

`\l_text_accents_tl` Lists commands which represent accents, and which are left unchanged by expansion. (Defined only for the L^AT_EX 2 _{ε} package.)

`\l_text_letterlike_tl` Lists commands which represent letters; these are left unchanged by expansion. (Defined only for the L^AT_EX 2 _{ε} package.)

`\l_text_math_arg_tl` Lists commands present in the $\langle text \rangle$ where the argument of the command should be treated as math mode material. The treatment here is similar to `\l_text_math_delims_tl` but for a command rather than paired delimiters.

`\l_text_math_delims_tl` Lists pairs of tokens which delimit (in-line) math mode content; such content *may* be excluded from processing.

`\l_text_case_exclude_arg_tl` Lists commands which are excluded from case changing.

`\l_text_expand_exclude_tl` Lists commands which are excluded from expansion.

`\l_text_titlecase_check_letter_bool`

Controls how the start of titlecasing is handled: when `true`, the first *letter* in text is considered. The standard setting is `true`.

Part XXXV

The **I3**legacy package

Interfaces to legacy concepts

There are a small number of TeX or L^AT_EX 2 _{ε} concepts which are not used in expl3 code but which need to be manipulated when working as a L^AT_EX 2 _{ε} package. To allow these to be integrated cleanly into expl3 code, a set of legacy interfaces are provided here.

`\legacy_if:nTF {name} {true code} {false code}`
`\legacy_if:nTF *`

Tests if the L^AT_EX 2 _{ε} /plain TeX conditional (generated by `\newif`) if **true** or **false** and branches accordingly. The *name* of the conditional should *omit* the leading **if**.

Part XXXVI

The **I3candidates** package

Experimental additions to **I3kernel**

1 Important notice

This module provides a space in which functions can be added to **I3kernel** (`expl3`) while still being experimental.

As such, the functions here may not remain in their current form, or indeed at all, in **I3kernel in the future.**

In contrast to the material in **I3experimental**, the functions here are all *small* additions to the kernel. We encourage programmers to test them out and report back on the **LaTeX-L** mailing list.

Thus, if you intend to use any of these functions from the candidate module in a public package offered to others for productive use (e.g., being placed on CTAN) please consider the following points carefully:

- Be prepared that your public packages might require updating when such functions are being finalized.
- Consider informing us that you use a particular function in your public package, e.g., by discussing this on the **LaTeX-L** mailing list. This way it becomes easier to coordinate any updates necessary without issues for the users of your package.
- Discussing and understanding use cases for a particular addition or concept also helps to ensure that we provide the right interfaces in the final version so please give us feedback if you consider a certain candidate function useful (or not).

We only add functions in this space if we consider them being serious candidates for a final inclusion into the kernel. However, real use sometimes leads to better ideas, so functions from this module are **not necessarily stable** and we may have to adjust them!

2 Additions to **I3box**

2.1 Viewing part of a box

\box_clip:N
\box_clip:c
\box_gclip:N
\box_gclip:c

Updated: 2019-01-23

\box_clip:N *box*

Clips the *box* in the output so that only material inside the bounding box is displayed in the output. The updated *box* is an hbox, irrespective of the nature of the *box* before the clipping is applied.

These functions require the **LaTeX3 native drivers: they do not work with the **LaTeX2 ϵ** graphics drivers!**

TeXhackers note: Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

```
\box_set_trim:Nnnnn  
\box_set_trim:cnnnn  
\box_gset_trim:Nnnnn  
\box_gset_trim:cnnnn
```

New: 2019-01-23

```
\box_set_trim:Nnnnn <box> {\<left>} {\<bottom>} {\<right>} {\<top>}
```

Adjusts the bounding box of the $\langle box \rangle$ $\langle left \rangle$ is removed from the left-hand edge of the bounding box, $\langle right \rangle$ from the right-hand edge and so fourth. All adjustments are *dimension expressions*. Material outside of the bounding box is still displayed in the output unless $\box_clip:N$ is subsequently applied. The updated $\langle box \rangle$ is an hbox, irrespective of the nature of the $\langle box \rangle$ before the trim operation is applied. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

```
\box_set_viewport:Nnnnn  
\box_set_viewport:cnnnn  
\box_gset_viewport:Nnnnn  
\box_gset_viewport:cnnnn
```

New: 2019-01-23

```
\box_set_viewport:Nnnnn <box> {\<llx>} {\<lly>} {\<urx>} {\<ury>}
```

Adjusts the bounding box of the $\langle box \rangle$ such that it has lower-left co-ordinates ($\langle llx \rangle$, $\langle lly \rangle$) and upper-right co-ordinates ($\langle urx \rangle$, $\langle ury \rangle$). All four co-ordinate positions are *dimension expressions*. Material outside of the bounding box is still displayed in the output unless $\box_clip:N$ is subsequently applied. The updated $\langle box \rangle$ is an hbox, irrespective of the nature of the $\langle box \rangle$ before the viewport operation is applied.

3 Additions to \bf I3exp

```
\exp_args_generate:n  
New: 2018-04-04  
Updated: 2019-02-08
```

```
\exp_args_generate:n {\<variant argument specifiers>}
```

Defines $\exp_args:N\langle variant \rangle$ functions for each $\langle variant \rangle$ given in the comma list $\{\langle variant argument specifiers \rangle\}$. Each $\langle variant \rangle$ should consist of the letters N, c, n, V, v, o, f, e, x, p and the resulting function is protected if the letter x appears in the $\langle variant \rangle$. This is only useful for cases where $\cs_generate_variant:Nn$ is not applicable.

4 Additions to \bf I3fp

```
\fp_if_nan_p:n *  
\fp_if_nan:nTF *
```

New: 2019-08-25

```
\fp_if_nan:n {\<fpexpr>}
```

Evaluates the $\langle fpexpr \rangle$ and tests whether the result is exactly NaN. The test returns `false` for any other result, even a tuple containing NaN.

5 Additions to \bf I3file

```
\iow_allow_break:  
New: 2018-12-29
```

```
\iow_allow_break:
```

In the first argument of $\iow_wrap:nnN$ (for instance in messages), inserts a break-point that allows a line break. In other words this is a zero-width breaking space.

```
\ior_get_term:nN  
\ior_str_get_term:nN
```

New: 2019-03-23

```
\ior_get_term:nN <prompt> <token list variable>
```

Function that reads one or more lines (until an equal number of left and right braces are found) from the terminal and stores the result locally in the *<token list>* variable. Tokenization occurs as described for `\ior_get:NN` or `\ior_str_get:NN`, respectively. When the *<prompt>* is empty, TeX will wait for input without any other indication: typically the programmer will have provided a suitable text using e.g. `\iow_term:n`. Where the *<prompt>* is given, it will appear in the terminal followed by an =, e.g.

```
prompt=
```

```
\ior_shell_open:Nn  
\ior_shell_close:Nn
```

New: 2019-05-08

```
\ior_shell_open:Nn <stream> {<shell command>}
```

Opens the *pseudo*-file created by the output of the *<shell command>* for reading using *<stream>* as the control sequence for access. If the *<stream>* was already open it is closed before the new operation begins. The *<stream>* is available for access immediately and will remain allocated to *<shell command>* until a `\ior_close:N` instruction is given or the TeX run ends. If piped system calls are disabled an error is raised.

For details of handling of the *<shell command>*, see `\sys_get_shell:nnN(TF)`.

6 Additions to `I3flag`

```
\flag_raise_if_clear:n *
```

New: 2018-04-02

```
\flag_raise_if_clear:n {<flag name>}
```

Ensures the *<flag>* is raised by making its height at least 1, locally.

7 Additions to `I3intarray`

```
\intarray_gset_rand:Nnn  
\intarray_gset_rand:cnn  
\intarray_gset_rand:Nn  
\intarray_gset_rand:cn
```

New: 2018-05-05

```
\intarray_gset_rand:Nnn <intarray var> {<minimum>} {<maximum>}  
\intarray_gset_rand:Nn <intarray var> {<maximum>}
```

Evaluates the integer expressions *<minimum>* and *<maximum>* then sets each entry (independently) of the *<integer array variable>* to a pseudo-random number between the two (with bounds included). If the absolute value of either bound is bigger than $2^{30} - 1$, an error occurs. Entries are generated in the same way as repeated calls to `\int_rand:nn` or `\int_rand:n` respectively, in particular for the second function the *<minimum>* is 1. Assignments are always global. This is not available in older versions of XeTeX.

7.1 Working with contents of integer arrays

```
\intarray_to_clist:N *
```

New: 2018-05-04

```
\intarray_to_clist:N <intarray var>
```

Converts the *<intarray>* to integer denotations separated by commas. All tokens have category code other. If the *<intarray>* has no entry the result is empty; otherwise the result has one fewer comma than the number of items.

8 Additions to **l3msg**

\msg_show_eval:Nn

\msg_log_eval:Nn

New: 2017-12-04

\msg_show_eval:Nn *function* {\i}

Shows or logs the *expression* (turned into a string), an equal sign, and the result of applying the *function* to the {\i} (with f-expansion). For instance, if the *function* is \int_eval:n and the *expression* is 1+2 then this logs > 1+2=3.

\msg_show:nnnnnn

\msg_show:nnxxxx

\msg_show:nnnnn

\msg_show:nnxxx

\msg_show:nnnn

\msg_show:nnxx

\msg_show:nnn

\msg_show:nnx

\msg_show:nn

\msg_show:nnnnnn {\i} {\i} {\i} {\i} {\i} {\i}

Issues *module* information *message*, passing *arg one* to *arg four* to the text-creating functions. The information text is shown on the terminal and the TeX run is interrupted in a manner similar to \tl_show:n. This is used in conjunction with \msg_show_item:n and similar functions to print complex variable contents completely. If the formatted text does not contain >~ at the start of a line, an additional line >~. will be put at the end. In addition, a final period is added if not present.

New: 2017-12-04

\msg_show_item:n

* \seq_map_function>NN {\i} \msg_show_item:n

\msg_show_item_unbraced:n

* \prop_map_function>NN {\i} \msg_show_item:nn

\msg_show_item:nn

*

\msg_show_item_unbraced:nn

*

New: 2017-12-04

Used in the text of messages for \msg_show:nnxxxx to show or log a list of items or key-value pairs. The one-argument functions are used for sequences,clist or token lists and the others for property lists. These functions turn their arguments to strings.

9 Additions to **l3prg**

\bool_set_inverse:N

\bool_set_inverse:c

\bool_gset_inverse:N

\bool_gset_inverse:c

New: 2018-05-10

\bool_set_inverse:N *boolean*

Toggles the *boolean* from true to false and conversely: sets it to the inverse of its current value.

```
\bool_case_true:n    *
\bool_case_true:nTF *
\bool_case_false:n  *
\bool_case_false:nTF *

New: 2019-02-10
```

```
\bool_case_true:nTF
{
  {⟨boolexpr case1⟩} {⟨code case1⟩}
  {⟨boolexpr case2⟩} {⟨code case2⟩}
  ...
  {⟨boolexpr casen⟩} {⟨code casen⟩}}
}
{⟨true code⟩}
{⟨false code⟩}
```

Evaluates in turn each of the *⟨boolean expression cases⟩* until the first one that evaluates to *true* or to *false*, for *\bool_case_true:n* and *\bool_case_false:n*, respectively. The *⟨code⟩* associated to this first case is left in the input stream, followed by the *⟨true code⟩*, and other cases are discarded. If none of the cases match then only the *⟨false code⟩* is inserted. The functions *\bool_case_true:n* and *\bool_case_false:n*, which do nothing if there is no match, are also available. For example

```
\bool_case_true:nF
{
  { \dim_compare_p:n { \l__mypkg_wd_dim <= 10pt } }
    { Fits }
  { \int_compare_p:n { \l__mypkg_total_int >= 10 } }
    { Many }
  { \l__mypkg_special_bool }
    { Special }
}
{ No idea! }
```

leaves “Fits” or “Many” or “Special” or “No idea!” in the input stream, in a way similar to some other language’s “if ... elseif ... elseif ... else ...”.

10 Additions to **I3prop**

```
\prop_rand_key_value:N *
\prop_rand_key_value:c *
```

New: 2016-12-06

```
\prop_rand_key_value:N <prop var>
```

Selects a pseudo-random key–value pair from the *⟨property list⟩* and returns {⟨key⟩} and {⟨value⟩}. If the *⟨property list⟩* is empty the result is empty. This is not available in older versions of X_HT_EX.

TeXhackers note: The result is returned within the *\unexpanded* primitive (*\exp_not:n*), which means that the *⟨value⟩* does not expand further when appearing in an *x*-type argument expansion.

11 Additions to `I3seq`

`\seq_mapthread_function:NNN` ☆ `\seq_mapthread_function:NNN <seq1> <seq2> <function>`
`\seq_mapthread_function:(NcN|cNN|ccN)` ☆

Applies `<function>` to every pair of items `<seq1-item>-<seq2-item>` from the two sequences, returning items from both sequences from left to right. The `<function>` receives two n-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

`\seq_set_filter:NNn` `\seq_gset_filter:NNn` `\seq_set_filter:NNn <sequence1> <sequence2> {<inline boolexpr>}`

Evaluates the `<inline boolexpr>` for every `<item>` stored within the `<sequence2>`. The `<inline boolexpr>` receives the `<item>` as #1. The sequence of all `<items>` for which the `<inline boolexpr>` evaluated to `true` is assigned to `<sequence1>`.

TEXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

`\seq_set_from_function:NnN` `\seq_set_from_function:NnN <seq var> {<loop code>} <function>`
`\seq_gset_from_function:NnN`

New: 2018-04-06

Sets the `<seq var>` equal to a sequence whose items are obtained by x-expanding `<loop code>` `<function>`. This expansion must result in successive calls to the `<function>` with no nonexpandable tokens in between. More precisely the `<function>` is replaced by a wrapper function that inserts the appropriate separators between items in the sequence. The `<loop code>` must be expandable; it can be for example `\tl_map_function:NN <tl var>` or `\clist_map_function:nN {<clist>}` or `\int_step_function:nnnN {<initial value>} {<step>} {<final value>}`.

`\seq_set_from_inline_x:Nnn` `\seq_set_from_inline_x:Nnn <seq var> {<loop code>} {<inline code>}`
`\seq_gset_from_inline_x:Nnn`

New: 2018-04-06

Sets the `<seq var>` equal to a sequence whose items are obtained by x-expanding `<loop code>` applied to a `<function>` derived from the `<inline code>`. A `<function>` is defined, that takes one argument, x-expands the `<inline code>` with that argument as #1, then adds appropriate separators to turn the result into an item of the sequence. The x-expansion of `<loop code>` `<function>` must result in successive calls to the `<function>` with no nonexpandable tokens in between. The `<loop code>` must be expandable; it can be for example `\tl_map_function:NN <tl var>` or `\clist_map_function:nN {<clist>}` or `\int_step_function:nnnN {<initial value>} {<step>} {<final value>}`, but not the analogous “inline” mappings.

12 Additions to **I3sys**

```
\c_sys_engine_version_str
```

New: 2018-05-02

The version string of the current engine, in the same form as given in the banner issued when running a job. For pdfTeX and LuaTeX this is of the form

$\langle major \rangle.\langle minor \rangle.\langle revision \rangle$

For XeTeX, the form is

$\langle major \rangle.\langle minor \rangle$

For pTeX and upTeX, only releases since TeX Live 2018 make the data available, and the form is more complex, as it comprises the pTeX version, the upTeX version and the e-pTeX version.

$p\langle major \rangle.\langle minor \rangle.\langle revision \rangle-u\langle major \rangle.\langle minor \rangle-\langle epTeX \rangle$

where the u part is only present for upTeX.

```
\sys_if_rand_exist_p: *
```

New: 2017-05-27

```
\sys_if_rand_exist_p:  
\sys_if_rand_exist:TF *
```

Tests if the engine has a pseudo-random number generator. Currently this is the case in pdfTeX, LuaTeX, pTeX, upTeX and recent releases of XeTeX.

13 Additions to l3tl

```
\tl_range_braced:Nnn    *
\tl_range_braced:cnn   *
\tl_range_braced:nnn   *
\tl_range_unbraced:Nnn *
\tl_range_unbraced:cnn *
\tl_range_unbraced:nnn *
```

New: 2017-07-15

Leaves in the input stream the items from the *<start index>* to the *<end index>* inclusive, using the same indexing as `\tl_range:nnn`. Spaces are ignored. Regardless of whether items appear with or without braces in the *<token list>*, the `\tl_range_braced:nnn` function wraps each item in braces, while `\tl_range_unbraced:nnn` does not (overall it removes an outer set of braces). For instance,

```
\iow_term:x { \tl_range_braced:nnn { abcd~{e{}}f } { 2 } { 5 } }
\iow_term:x { \tl_range_braced:nnn { abcd~{e{}}f } { -4 } { -1 } }
\iow_term:x { \tl_range_braced:nnn { abcd~{e{}}f } { -2 } { -1 } }
\iow_term:x { \tl_range_braced:nnn { abcd~{e{}}f } { 0 } { -1 } }
```

prints `{b}{c}{d}{e{}}`, `{c}{d}{e{}}{f}`, `{e{}}{f}`, and an empty line to the terminal, while

```
\iow_term:x { \tl_range_unbraced:nnn { abcd~{e{}}f } { 2 } { 5 } }
\iow_term:x { \tl_range_unbraced:nnn { abcd~{e{}}f } { -4 } { -1 } }
\iow_term:x { \tl_range_unbraced:nnn { abcd~{e{}}f } { -2 } { -1 } }
\iow_term:x { \tl_range_unbraced:nnn { abcd~{e{}}f } { 0 } { -1 } }
```

prints `bcd{}`, `cde{}`, `e{}`, and an empty line to the terminal. Because braces are removed, the result of `\tl_range_unbraced:nnn` may have a different number of items as for `\tl_range:nnn` or `\tl_range_braced:nnn`. In cases where preserving spaces is important, consider the slower function `\tl_range:nnn`.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* does not expand further when appearing in an `x`-type argument expansion.

```
\tl_build_begin:N
\tl_build_gbegin:N
```

New: 2018-04-01

`\tl_build_begin:N <tl var>`

Clears the *<tl var>* and sets it up to support other `\tl_build_...` functions, which allow accumulating large numbers of tokens piece by piece much more efficiently than standard l3tl functions. Until `\tl_build_end:N <tl var>` is called, applying any function from l3tl other than `\tl_build_...` will lead to incorrect results. The `begin` and `gbegin` functions must be used for local and global *<tl var>* respectively.

```
\tl_build_clear:N
\tl_build_gclear:N
```

New: 2018-04-01

`\tl_build_clear:N <tl var>`

Clears the *<tl var>* and sets it up to support other `\tl_build_...` functions. The `clear` and `gclear` functions must be used for local and global *<tl var>* respectively.

```
\tl_build_put_left:Nn
\tl_build_put_left:Nx
\tl_build_gput_left:Nn
\tl_build_gput_left:Nx
\tl_build_put_right:Nn
\tl_build_put_right:Nx
\tl_build_gput_right:Nn
\tl_build_gput_right:Nx
```

New: 2018-04-01

```
\tl_build_put_left:Nn <tl var> {<tokens>}
\tl_build_put_right:Nn <tl var> {<tokens>}
```

Adds *<tokens>* to the left or right side of the current contents of *<tl var>*. The *<tl var>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `put` and `gput` functions must be used for local and global *<tl var>* respectively. The `right` functions are about twice faster than the `left` functions.

```
\tl_build_get:NN
```

New: 2018-04-01

```
\tl_build_get:N <tl var1> <tl var2>
```

Stores the contents of the *<tl var1>* in the *<tl var2>*. The *<tl var1>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The *<tl var2>* is a “normal” token list variable, assigned locally using `\tl_set:Nn`.

```
\tl_build_end:N
\tl_build_gend:N
```

New: 2018-04-01

```
\tl_build_end:N <tl var>
```

Gets the contents of *<tl var>* and stores that into the *<tl var>* using `\tl_set:Nn` or `\tl_gset:Nn`. The *<tl var>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `end` and `gend` functions must be used for local and global *<tl var>* respectively. These functions completely remove the setup code that enabled *<tl var>* to be used for other `\tl_build_...` functions.

14 Additions to `I3token`

```
\c_catcode_active_space_tl
```

New: 2017-08-07

Token list containing one character with category code 13, (“active”), and character code 32 (space).

```
\char_to_utfviii_bytes:n ∗
```

New: 2020-01-09

```
\char_to_utfviii_bytes:n {<codepoint>}
```

Converts the (Unicode) *<codepoint>* to UTF-8 bytes. The expansion of this function comprises four brace groups, each of which will contain a hexadecimal value: the appropriate byte. As UTF-8 is a variable-length, one or more of the groups may be empty: the bytes read in the logical order, such that a two-byte codepoint will have groups #1 and #2 filled and #3 and #4 empty.

```
\char_to_nfd:N ☆
```

New: 2020-01-02

```
\char_to_nfd:N <char>
```

Converts the *<char>* to the Unicode Normalization Form Canonical Decomposition. The category code of the generated character is the same as the *<char>*. With 8-bit engines, no change is made to the character.

```
\peek_catcode_collect_inline:Nn          \peek_catcode_collect_inline:Nn <test token> {{inline code}}
\peekCharCode_collect_inline:Nn          \peekCharCode_collect_inline:Nn <test token> {{inline code}}
\peek_meaning_collect_inline:Nn          \peek_meaning_collect_inline:Nn <test token> {{inline code}}
```

New: 2018-09-23

Collects and removes tokens from the input stream until finding a token that does not match the $\langle test\ token \rangle$ (as defined by the test `\token_if_eq_catcode:NNTF` or `\token_if_eq_charcode:NNTF` or `\token_if_eq_meaning:NNTF`). The collected tokens are passed to the $\langle inline\ code \rangle$ as #1. When begin-group or end-group tokens (usually { or }) are collected they are replaced by implicit `\c_group_begin_token` and `\c_group_end_token`, and when spaces (including `\c_space_token`) are collected they are replaced by explicit spaces.

For example the following code prints “Hello” to the terminal and leave “, world!” in the input stream.

```
\peek_catcode_collect_inline:Nn A { \iow_term:n {#1} } Hello,~world!
```

Another example is that the following code tests if the next token is *, ignoring intervening spaces, but putting them back using #1 if there is no *.

```
\peek_meaning_collect_inline:Nn \c_space_token
{ \peekCharCode:NTF * { star } { no~star #1 } }
```

```
\peek_remove_spaces:n \peek_remove_spaces:n {<code>}
```

New: 2018-10-01

Removes explicit and implicit space tokens (category code 10 and character code 32) from the input stream, then inserts $\langle code \rangle$.

Part XXXVII

Implementation

1 l3bootstrap implementation

```
1  {*package}
2  @@=kernel
```

1.1 LuaTeX-specific code

Depending on the versions available, the L^AT_EX format may not have the raw \Umath primitive names available. We fix that globally: it should cause no issues. Older LuaT_EX versions do not have a pre-built table of the primitive names here so sort one out ourselves. These end up globally-defined but at that is true with a newer format anyway and as they all start \U this should be reasonably safe.

```
3  \begingroup
4    \expandafter\ifx\csname directlua\endcsname\relax
5    \else
6      \directlua{%
7        local i
8        local t = { }
```

```

9      for _,i in pairs(tex.extraprimitives("luatex")) do
10     if string.match(i,"^U") then
11       if not string.match(i,"^Uchar$") then %%
12         table.insert(t,i)
13       end
14     end
15   end
16   tex.enableprimitives("", t)
17 }%
18 \fi
19 \endgroup

```

1.2 The `\pdfstrcmp` primitive in X_ET_EX

Only pdfT_EX has a primitive called `\pdfstrcmp`. The X_ET_EX version is just `\strcmp`, so there is some shuffling to do. As this is still a real primitive, using the pdfT_EX name is “safe”.

```

20 \begingroup\expandafter\expandafter\expandafter\endgroup
21   \expandafter\ifx\csname pdfstrcmp\endcsname\relax
22     \let\pdfstrcmp\strcmp
23   \fi

```

1.3 Loading support Lua code

When LuaT_EX is used there are various pieces of Lua code which need to be loaded. The code itself is defined in l3luatex and is extracted into a separate file. Thus here the task is to load the Lua code both now and (if required) at the start of each job.

```

24 \begingroup\expandafter\expandafter\expandafter\endgroup
25 \expandafter\ifx\csname directlua\endcsname\relax
26 \else
27   \ifnum\luatexversion<110 %
28   \else

```

For LuaT_EX we make sure the basic support is loaded: this is only necessary in plain.

Additionally we just ensure that T_EX has seen the csnames `\prg_return_true:` and `\prg_return_false:` before the Lua code builds these tokens.

```

29   \begingroup\expandafter\expandafter\expandafter\endgroup
30   \expandafter\ifx\csname newcatcodetable\endcsname\relax
31     \input{ltluatex}%
32   \fi
33   \begingroup\edef\ignored{%
34     \expandafter\noexpand\csname prg_return_true:\endcsname
35     \expandafter\noexpand\csname prg_return_false:\endcsname
36   }\endgroup
37   \directlua{require("expl3")}%

```

As the user might be making a custom format, no assumption is made about matching package mode with only loading the Lua code once. Instead, a query to Lua reveals what mode is in operation.

```

38   \ifnum 0%
39     \directlua{
40       if status.ini_version then
41         tex.write("1")

```

```

42         end
43     }>0 %
44     \everyjob\expandafter{%
45         \the\expandafter\everyjob
46         \csname\detokenize{lua_now:n}\endcsname{require("expl3")}%
47     }%
48     \fi
49     \fi
50 \fi

```

1.4 Engine requirements

The code currently requires ε - \TeX and functionality equivalent to `\pdfstrcmp`, and also driver and Unicode character support. This is available in a reasonably-wide range of engines.

For $\text{\LaTeX}_\text{\TeX}$, we require at least Lua 5.3 and the `token.set_lua` function. This is available at least since $\text{\LaTeX}_\text{\TeX}$ 1.10.

```

51 \begingroup
52   \def\next{\endgroup}%
53   \def\ShortText{Required primitives not found}%
54   \def\LongText{%
55     \%
56     LaTeX3 requires the e-\TeX primitives and additional functionality as
57     described in the README file.
58     \LineBreak
59     These are available in the engines\LineBreak
60     - pdfTeX v1.40\LineBreak
61     - XeTeX v0.99992\LineBreak
62     - LuaTeX v1.10\LineBreak
63     - e-(u)pTeX mid-2012\LineBreak
64     or later.\LineBreak
65     \LineBreak
66   }%
67 \ifnum0%
68   \expandafter\ifx\csname pdfstrcmp\endcsname\relax
69   \else
70     \expandafter\ifx\csname pdftexversion\endcsname\relax
71       \expandafter\ifx\csname Ucharcat\endcsname\relax
72         \expandafter\ifx\csname kanjiskip\endcsname\relax
73           \else
74             1%
75           \fi
76         \else
77           1%
78         \fi
79       \else
80         \ifnum\pdftexversion<140 \else 1\fi
81       \fi
82   \fi
83   \expandafter\ifx\csname directlua\endcsname\relax
84   \else
85     \ifnum\luatexversion<110 \else 1\fi
86   \fi

```

```

87     =0 %
88     \newlinechar`\^J %
89     \def\LineBreak{\noexpand\MessageBreak}%
90     \expandafter\ifx\csname PackageError\endcsname\relax
91         \def\LineBreak{^J}%
92         \def\PackageError#1#2#3%
93             {%
94                 \errhelp{#3}%
95                 \errmessage{#1 Error: #2}%
96             }%
97         \fi
98         \edef\next
99             {%
100                 \noexpand\PackageError{expl3}{\ShortText}%
101                 {\LongText Loading of expl3 will abort!}%
102                 \endgroup
103                 \noexpand\endinput
104             }%
105         \fi
106     \next

```

1.5 Extending allocators

The ability to extend \TeX 's allocation routine to allow for ε - \TeX has been around since 1997 in the `etex` package. Loading this support is delayed until here as we are now sure that the ε - \TeX extensions and `\pdfstrcmp` or equivalent are available. Thus there is no danger of an “uncontrolled” error if the engine requirements are not met.

For $\text{\LaTeX}2\varepsilon$ we need to make sure that the extended pool is being used: `expl3` uses a lot of registers. For formats from 2015 onward there is nothing to do as this is automatic. For older formats, the `etex` package needs to be loaded to do the job. In that case, some inserts are reserved also as these have to be from the standard pool. Note that `\reserveinserts` is `\outer` and so is accessed here by `csname`. In earlier versions, loading `etex` was done directly and so `\reserveinserts` appeared in the code: this then required a `\relax` after `\RequirePackage` to prevent an error with “unsafe” definitions as seen for example with `capoptions`. The optional loading here is done using a group and `\ifx` test as we are not quite in the position to have a single name for `\pdfstrcmp` just yet.

```

107 \begingroup
108   \def\@tempa{LaTeX2e}%
109   \def\next{}%
110   \ifx\fntname\@tempa
111     \expandafter\ifx\csname extrafloats\endcsname\relax
112       \def\next
113           {%
114               \RequirePackage{etex}%
115               \csname reserveinserts\endcsname{32}%
116           }%
117       \fi
118   \fi
119 \expandafter\endgroup
120 \next

```

1.6 The L^AT_EX3 code environment

The code environment is now set up.

\ExplSyntaxOff Before changing any category codes, in package mode we need to save the situation before loading. Note the set up here means that once applied \ExplSyntaxOff becomes a “do nothing” command until \ExplSyntaxOn is used.

```
121 \protected\edef\ExplSyntaxOff
122   {%
123     \protected\def\noexpand\ExplSyntaxOff{}%
124     \catcode 9 = \the\catcode 9\relax
125     \catcode 32 = \the\catcode 32\relax
126     \catcode 34 = \the\catcode 34\relax
127     \catcode 38 = \the\catcode 38\relax
128     \catcode 58 = \the\catcode 58\relax
129     \catcode 94 = \the\catcode 94\relax
130     \catcode 95 = \the\catcode 95\relax
131     \catcode 124 = \the\catcode 124\relax
132     \catcode 126 = \the\catcode 126\relax
133     \endlinechar = \the\endlinechar\relax
134     \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
135   }%
```

(End definition for \ExplSyntaxOff. This function is documented on page 7.)

The code environment is now set up.

```
136 \catcode 9 = 9\relax
137 \catcode 32 = 9\relax
138 \catcode 34 = 12\relax
139 \catcode 38 = 4\relax
140 \catcode 58 = 11\relax
141 \catcode 94 = 7\relax
142 \catcode 95 = 11\relax
143 \catcode 124 = 12\relax
144 \catcode 126 = 10\relax
145 \endlinechar = 32\relax
```

\l__kernel_expl_bool The status for experimental code syntax: this is on at present.

```
146 \chardef\l__kernel_expl_bool = 1\relax
```

(End definition for \l__kernel_expl_bool.)

\ExplSyntaxOn The idea here is that multiple \ExplSyntaxOn calls are not going to mess up category codes, and that multiple calls to \ExplSyntaxOff are also not wasting time. Applying \ExplSyntaxOn alters the definition of \ExplSyntaxOff and so in package mode this function should not be used until after the end of the loading process!

```
147 \protected \def \ExplSyntaxOn
148   {
149     \bool_if:NF \l__kernel_expl_bool
150     {
151       \cs_set_protected:Npx \ExplSyntaxOff
152       {
153         \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
154         \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
155         \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
```

```

156      \char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
157      \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
158      \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
159      \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
160      \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
161      \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
162      \tex_endlinechar:D =
163          \tex_the:D \tex_endlinechar:D \scan_stop:
164      \bool_set_false:N \l__kernel_expl_bool
165      \cs_set_protected:Npn \ExplSyntaxOff { }
166  }
167  }
168  \char_set_catcode_ignore:n      { 9 } % tab
169  \char_set_catcode_ignore:n      { 32 } % space
170  \char_set_catcode_other:n      { 34 } % double quote
171  \char_set_catcode_alignment:n   { 38 } % ampersand
172  \char_set_catcode_letter:n     { 58 } % colon
173  \char_set_catcode_math_superscript:n { 94 } % circumflex
174  \char_set_catcode_letter:n     { 95 } % underscore
175  \char_set_catcode_other:n     { 124 } % pipe
176  \char_set_catcode_space:n     { 126 } % tilde
177  \tex_endlinechar:D = 32 \scan_stop:
178  \bool_set_true:N \l__kernel_expl_bool
179  }

```

(End definition for `\ExplSyntaxOn`. This function is documented on page 7.)

```
180 </package>
```

2 **I3names** implementation

```
181 <*package & tex>
```

The prefix here is `kernel`. A few places need `@@` to be left as is; this is obtained as `0000`.

```
182 <@0=kernel>
```

The code here simply renames all of the primitives to new, internal, names.

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded csname in the hash table.

```
183 \let \tex_global:D \global
184 \let \tex_let:D \let
```

Everything is inside a (rather long) group, which keeps `__kernel_primitive:NN` trapped.

```
185 \begingroup
```

A temporary function to actually do the renaming.

```
186 \long \def \__kernel_primitive:NN #1#2
187   { \tex_global:D \tex_let:D #2 #1 }
```

(End definition for `__kernel_primitive:NN`.)

To allow extracting “just the names”, a bit of DocStrip fiddling.

```
188 </package & tex>
```

```
189 <*names | tex>
```

```
190 <*names | package>
```

In the current incarnation of this package, all T_EX primitives are given a new name of the form `\tex_<oldname>:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

```

191  \__kernel_primitive:NN \
192  \__kernel_primitive:NN \
193  \__kernel_primitive:NN \

```

Now all the other primitives.

```

194  \__kernel_primitive:NN \above           \tex_above:D
195  \__kernel_primitive:NN \abovedisplayshortskip \tex_abovedisplayshortskip:D
196  \__kernel_primitive:NN \abovedisplayskip   \tex_abovedisplayskip:D
197  \__kernel_primitive:NN \abovewithdelims  \tex_abovewithdelims:D
198  \__kernel_primitive:NN \accent          \tex_accent:D
199  \__kernel_primitive:NN \adjdemerits    \tex_adjdemerits:D
200  \__kernel_primitive:NN \advance         \tex_advance:D
201  \__kernel_primitive:NN \afterassignment \tex_afterassignment:D
202  \__kernel_primitive:NN \aftergroup     \tex_aftergroup:D
203  \__kernel_primitive:NN \atop          \tex_atop:D
204  \__kernel_primitive:NN \atopwithdelims \tex_atopwithdelims:D
205  \__kernel_primitive:NN \badness        \tex_badness:D
206  \__kernel_primitive:NN \baselineskip   \tex_baselineskip:D
207  \__kernel_primitive:NN \batchmode      \tex_batchmode:D
208  \__kernel_primitive:NN \begingroup     \tex_begingroup:D
209  \__kernel_primitive:NN \belowdisplayshortskip \tex_belowdisplayshortskip:D
210  \__kernel_primitive:NN \belowdisplayskip \tex_belowdisplayskip:D
211  \__kernel_primitive:NN \binoppenalty   \tex_binoppenalty:D
212  \__kernel_primitive:NN \botmark       \tex_botmark:D
213  \__kernel_primitive:NN \box           \tex_box:D
214  \__kernel_primitive:NN \boxmaxdepth   \tex_boxmaxdepth:D
215  \__kernel_primitive:NN \brokenpenalty \tex_brokenpenalty:D
216  \__kernel_primitive:NN \catcode       \tex_catcode:D
217  \__kernel_primitive:NN \char          \tex_char:D
218  \__kernel_primitive:NN \chardef      \tex_chardef:D
219  \__kernel_primitive:NN \cleaders     \tex_cleaders:D
220  \__kernel_primitive:NN \closein     \tex_closein:D
221  \__kernel_primitive:NN \closeout    \tex_closeout:D
222  \__kernel_primitive:NN \clubpenalty  \tex_clubpenalty:D
223  \__kernel_primitive:NN \copy         \tex_copy:D
224  \__kernel_primitive:NN \count       \tex_count:D
225  \__kernel_primitive:NN \countdef    \tex_countdef:D
226  \__kernel_primitive:NN \cr          \tex_cr:D
227  \__kernel_primitive:NN \crrc         \tex_crrc:D
228  \__kernel_primitive:NN \csname     \tex_csname:D
229  \__kernel_primitive:NN \day         \tex_day:D
230  \__kernel_primitive:NN \deadcycles  \tex_deadcycles:D
231  \__kernel_primitive:NN \def         \tex_def:D
232  \__kernel_primitive:NN \defaulthyphenchar \tex_defaulthyphenchar:D
233  \__kernel_primitive:NN \defaultskewchar \tex_defaultskewchar:D
234  \__kernel_primitive:NN \delcode     \tex_delcode:D
235  \__kernel_primitive:NN \delimiter   \tex_delimiter:D
236  \__kernel_primitive:NN \delimiterfactor \tex_delimiterfactor:D
237  \__kernel_primitive:NN \delimitershortfall \tex_delimitershortfall:D
238  \__kernel_primitive:NN \dimen      \tex_dimen:D

```

```

239  \__kernel_primitive:NN \dimendef          \tex_dimendef:D
240  \__kernel_primitive:NN \discretionary      \tex_discretionary:D
241  \__kernel_primitive:NN \displayindent      \tex_displayindent:D
242  \__kernel_primitive:NN \displaylimits      \tex_displaylimits:D
243  \__kernel_primitive:NN \displaystyle       \tex_displaystyle:D
244  \__kernel_primitive:NN \displaywidowpenalty \tex_displaywidowpenalty:D
245  \__kernel_primitive:NN \displaywidth       \tex_displaywidth:D
246  \__kernel_primitive:NN \divide             \tex_divide:D
247  \__kernel_primitive:NN \doublehyphendemerits \tex_doublehyphendemerits:D
248  \__kernel_primitive:NN \dp                 \tex_dp:D
249  \__kernel_primitive:NN \dump               \tex_dump:D
250  \__kernel_primitive:NN \edef               \tex_edef:D
251  \__kernel_primitive:NN \else               \tex_else:D
252  \__kernel_primitive:NN \emergencystretch   \tex_emergencystretch:D
253  \__kernel_primitive:NN \end                \tex_end:D
254  \__kernel_primitive:NN \endcsname         \tex_endcsname:D
255  \__kernel_primitive:NN \endgroup           \tex_endgroup:D
256  \__kernel_primitive:NN \endinput           \tex_endinput:D
257  \__kernel_primitive:NN \endlinechar        \tex_endlinechar:D
258  \__kernel_primitive:NN \eqno               \tex_eqno:D
259  \__kernel_primitive:NN \errhelp            \tex_errhelp:D
260  \__kernel_primitive:NN \errmessage         \tex_errmessage:D
261  \__kernel_primitive:NN \errorcontextlines  \tex_errorcontextlines:D
262  \__kernel_primitive:NN \errorstopmode      \tex_errorstopmode:D
263  \__kernel_primitive:NN \escapechar         \tex_escapechar:D
264  \__kernel_primitive:NN \everycr            \tex_everycr:D
265  \__kernel_primitive:NN \everydisplay        \tex_everydisplay:D
266  \__kernel_primitive:NN \everyhbox           \tex_everyhbox:D
267  \__kernel_primitive:NN \everyjob            \tex_everyjob:D
268  \__kernel_primitive:NN \everymath           \tex_everymath:D
269  \__kernel_primitive:NN \everypar            \tex_everypar:D
270  \__kernel_primitive:NN \everyvbox           \tex_everyvbox:D
271  \__kernel_primitive:NN \exhyphenpenalty    \tex_exhyphenpenalty:D
272  \__kernel_primitive:NN \expandafter        \tex_expandafter:D
273  \__kernel_primitive:NN \fam                \tex_fam:D
274  \__kernel_primitive:NN \fi                 \tex_fi:D
275  \__kernel_primitive:NN \finalhyphendemerits \tex_finalhyphendemerits:D
276  \__kernel_primitive:NN \firstmark          \tex_firstmark:D
277  \__kernel_primitive:NN \floatingpenalty     \tex_floatingpenalty:D
278  \__kernel_primitive:NN \font                \tex_font:D
279  \__kernel_primitive:NN \fontdimen          \tex_fontdimen:D
280  \__kernel_primitive:NN \fontname           \tex_fontname:D
281  \__kernel_primitive:NN \futurelet          \tex_futurelet:D
282  \__kernel_primitive:NN \gdef               \tex_gdef:D
283  \__kernel_primitive:NN \global              \tex_global:D
284  \__kernel_primitive:NN \globaldefs         \tex_globaldefs:D
285  \__kernel_primitive:NN \halign              \tex_halign:D
286  \__kernel_primitive:NN \hangafter          \tex_hangafter:D
287  \__kernel_primitive:NN \hangindent         \tex_hangindent:D
288  \__kernel_primitive:NN \hbadness           \tex_hbadness:D
289  \__kernel_primitive:NN \hbox                \tex_hbox:D
290  \__kernel_primitive:NN \hfil               \tex_hfil:D
291  \__kernel_primitive:NN \hfill               \tex_hfill:D
292  \__kernel_primitive:NN \hfilneg            \tex_hfilneg:D

```

```

293  \__kernel_primitive:NN \hfuzz          \tex_hfuzz:D
294  \__kernel_primitive:NN \hoffset         \tex_hoffset:D
295  \__kernel_primitive:NN \holdinginserts \tex_holdinginserts:D
296  \__kernel_primitive:NN \hrule           \tex_hrule:D
297  \__kernel_primitive:NN \hsize           \tex_hsize:D
298  \__kernel_primitive:NN \hskip            \tex_hskip:D
299  \__kernel_primitive:NN \hss             \tex_hss:D
300  \__kernel_primitive:NN \ht              \tex_ht:D
301  \__kernel_primitive:NN \hyphenation    \tex_hyphenation:D
302  \__kernel_primitive:NN \hyphenchar     \tex_hyphenchar:D
303  \__kernel_primitive:NN \hyphenpenalty   \tex_hyphenpenalty:D
304  \__kernel_primitive:NN \if               \tex_if:D
305  \__kernel_primitive:NN \ifcase          \tex_ifcase:D
306  \__kernel_primitive:NN \ifcat           \tex_ifcat:D
307  \__kernel_primitive:NN \ifdim           \tex_ifdim:D
308  \__kernel_primitive:NN \ifeof           \tex_ifeof:D
309  \__kernel_primitive:NN \iffalse          \tex_iffalse:D
310  \__kernel_primitive:NN \ifhbox          \tex_ifhbox:D
311  \__kernel_primitive:NN \ifhmode         \tex_ifhmode:D
312  \__kernel_primitive:NN \ifinner         \tex_ifinner:D
313  \__kernel_primitive:NN \ifmmode          \tex_ifmmode:D
314  \__kernel_primitive:NN \ifnum            \tex_ifnum:D
315  \__kernel_primitive:NN \ifodd            \tex_ifodd:D
316  \__kernel_primitive:NN \iftrue           \tex_iftrue:D
317  \__kernel_primitive:NN \ifvbox           \tex_ifvbox:D
318  \__kernel_primitive:NN \ifvmode          \tex_ifvmode:D
319  \__kernel_primitive:NN \ifvoid           \tex_ifvoid:D
320  \__kernel_primitive:NN \ifx              \tex_ifx:D
321  \__kernel_primitive:NN \ignorespaces    \tex_ignorespaces:D
322  \__kernel_primitive:NN \immediate        \tex_immediate:D
323  \__kernel_primitive:NN \indent           \tex_indent:D
324  \__kernel_primitive:NN \input             \tex_input:D
325  \__kernel_primitive:NN \inputlineno      \tex_inputlineno:D
326  \__kernel_primitive:NN \insert            \tex_insert:D
327  \__kernel_primitive:NN \insertpenalties  \tex_insertpenalties:D
328  \__kernel_primitive:NN \interlinepenalty  \tex_interlinepenalty:D
329  \__kernel_primitive:NN \jobname          \tex_jobname:D
330  \__kernel_primitive:NN \kern              \tex_kern:D
331  \__kernel_primitive:NN \language         \tex_language:D
332  \__kernel_primitive:NN \lastbox          \tex_lastbox:D
333  \__kernel_primitive:NN \lastkern         \tex_lastkern:D
334  \__kernel_primitive:NN \lastpenalty      \tex_lastpenalty:D
335  \__kernel_primitive:NN \lastskip         \tex_lastskip:D
336  \__kernel_primitive:NN \lccode           \tex_lccode:D
337  \__kernel_primitive:NN \leaders           \tex_leaders:D
338  \__kernel_primitive:NN \left             \tex_left:D
339  \__kernel_primitive:NN \lefthyphenmin    \tex_lefthyphenmin:D
340  \__kernel_primitive:NN \leftskip         \tex_leftskip:D
341  \__kernel_primitive:NN \leqno            \tex_leqno:D
342  \__kernel_primitive:NN \let               \tex_let:D
343  \__kernel_primitive:NN \limits            \tex_limits:D
344  \__kernel_primitive:NN \linepenalty       \tex_linepenalty:D
345  \__kernel_primitive:NN \lineskip          \tex_lineskip:D
346  \__kernel_primitive:NN \lineskiplimit    \tex_lineskiplimit:D

```

```

347  \__kernel_primitive:NN \long          \tex_long:D
348  \__kernel_primitive:NN \looseness      \tex_looseness:D
349  \__kernel_primitive:NN \lower          \tex_lower:D
350  \__kernel_primitive:NN \lowercase      \tex_lowercase:D
351  \__kernel_primitive:NN \mag           \tex_mag:D
352  \__kernel_primitive:NN \mark          \tex_mark:D
353  \__kernel_primitive:NN \mathaccent     \tex_mathaccent:D
354  \__kernel_primitive:NN \mathbin        \tex_mathbin:D
355  \__kernel_primitive:NN \mathchar       \tex_mathchar:D
356  \__kernel_primitive:NN \mathchardef    \tex_mathchardef:D
357  \__kernel_primitive:NN \mathchoice      \tex_mathchoice:D
358  \__kernel_primitive:NN \mathclose      \tex_mathclose:D
359  \__kernel_primitive:NN \mathcode       \tex_mathcode:D
360  \__kernel_primitive:NN \mathinner      \tex_mathinner:D
361  \__kernel_primitive:NN \mathop         \tex_mathop:D
362  \__kernel_primitive:NN \mathopen       \tex_mathopen:D
363  \__kernel_primitive:NN \mathord        \tex_mathord:D
364  \__kernel_primitive:NN \mathpunct      \tex_mathpunct:D
365  \__kernel_primitive:NN \mathrel       \tex_mathrel:D
366  \__kernel_primitive:NN \mathsurround   \tex_mathsurround:D
367  \__kernel_primitive:NN \maxdeadcycles \tex_maxdeadcycles:D
368  \__kernel_primitive:NN \maxdepth      \tex_maxdepth:D
369  \__kernel_primitive:NN \meaning      \tex_meaning:D
370  \__kernel_primitive:NN \medmuskip     \tex_medmuskip:D
371  \__kernel_primitive:NN \message       \tex_message:D
372  \__kernel_primitive:NN \mkern         \tex_mkern:D
373  \__kernel_primitive:NN \month         \tex_month:D
374  \__kernel_primitive:NN \moveleft      \tex_moveleft:D
375  \__kernel_primitive:NN \moveright     \tex_moveright:D
376  \__kernel_primitive:NN \mskip        \tex_msip:D
377  \__kernel_primitive:NN \multiply      \tex_multiply:D
378  \__kernel_primitive:NN \muskip       \tex_muskip:D
379  \__kernel_primitive:NN \muskipdef    \tex_muskipdef:D
380  \__kernel_primitive:NN \newlinechar   \tex_newlinechar:D
381  \__kernel_primitive:NN \noalign       \tex_noalign:D
382  \__kernel_primitive:NN \noboundary    \tex_noboundary:D
383  \__kernel_primitive:NN \noexpand      \tex_noexpand:D
384  \__kernel_primitive:NN \noindent      \tex_noindent:D
385  \__kernel_primitive:NN \nolimits      \tex_nolimits:D
386  \__kernel_primitive:NN \nonscript     \tex_nonscript:D
387  \__kernel_primitive:NN \nonstopmode   \tex_nonstopmode:D
388  \__kernel_primitive:NN \nulldelimiterspace \tex_nulldelimiterspace:D
389  \__kernel_primitive:NN \nullfont      \tex_nullfont:D
390  \__kernel_primitive:NN \number        \tex_number:D
391  \__kernel_primitive:NN \omit          \tex_omit:D
392  \__kernel_primitive:NN \openin        \tex_openin:D
393  \__kernel_primitive:NN \openout       \tex_openout:D
394  \__kernel_primitive:NN \or            \tex_or:D
395  \__kernel_primitive:NN \outer         \tex_outer:D
396  \__kernel_primitive:NN \output        \tex_output:D
397  \__kernel_primitive:NN \outputpenalty \tex_outputpenalty:D
398  \__kernel_primitive:NN \over          \tex_over:D
399  \__kernel_primitive:NN \overfullrule  \tex_overfullrule:D
400  \__kernel_primitive:NN \overline      \tex_overline:D

```

```

401  \__kernel_primitive:NN \overwithdelims
402  \__kernel_primitive:NN \pagedepth
403  \__kernel_primitive:NN \pagefillstretch
404  \__kernel_primitive:NN \pagefillstretch
405  \__kernel_primitive:NN \pagefilstretch
406  \__kernel_primitive:NN \pagegoal
407  \__kernel_primitive:NN \pageshrink
408  \__kernel_primitive:NN \pagestretch
409  \__kernel_primitive:NN \pagetotal
410  \__kernel_primitive:NN \par
411  \__kernel_primitive:NN \parfillskip
412  \__kernel_primitive:NN \parindent
413  \__kernel_primitive:NN \parshape
414  \__kernel_primitive:NN \parskip
415  \__kernel_primitive:NN \patterns
416  \__kernel_primitive:NN \pausing
417  \__kernel_primitive:NN \penalty
418  \__kernel_primitive:NN \postdisplaypenalty
419  \__kernel_primitive:NN \predisplaypenalty
420  \__kernel_primitive:NN \predisplaysize
421  \__kernel_primitive:NN \pretolerance
422  \__kernel_primitive:NN \prevdepth
423  \__kernel_primitive:NN \prevgraf
424  \__kernel_primitive:NN \radical
425  \__kernel_primitive:NN \raise
426  \__kernel_primitive:NN \read
427  \__kernel_primitive:NN \relax
428  \__kernel_primitive:NN \relpenalty
429  \__kernel_primitive:NN \right
430  \__kernel_primitive:NN \righthyphenmin
431  \__kernel_primitive:NN \rightskip
432  \__kernel_primitive:NN \romannumeral
433  \__kernel_primitive:NN \scriptfont
434  \__kernel_primitive:NN \scriptscriptfont
435  \__kernel_primitive:NN \scriptscriptstyle
436  \__kernel_primitive:NN \scriptspace
437  \__kernel_primitive:NN \scriptstyle
438  \__kernel_primitive:NN \scrollmode
439  \__kernel_primitive:NN \setbox
440  \__kernel_primitive:NN \setlanguage
441  \__kernel_primitive:NN \sfcode
442  \__kernel_primitive:NN \shipout
443  \__kernel_primitive:NN \show
444  \__kernel_primitive:NN \showbox
445  \__kernel_primitive:NN \showboxbreadth
446  \__kernel_primitive:NN \showboxdepth
447  \__kernel_primitive:NN \showlists
448  \__kernel_primitive:NN \showthe
449  \__kernel_primitive:NN \skewchar
450  \__kernel_primitive:NN \skip
451  \__kernel_primitive:NN \skipdef
452  \__kernel_primitive:NN \spacefactor
453  \__kernel_primitive:NN \spaceskip
454  \__kernel_primitive:NN \span

```

\tex_overwithdelims:D
\tex_pagedepth:D
\tex_pagefillstretch:D
\tex_pagefillstretch:D
\tex_pagefilstretch:D
\tex_pagegoal:D
\tex_pageshrink:D
\tex_pagestretch:D
\tex_pagetotal:D
\tex_par:D
\tex_parfillskip:D
\tex_parindent:D
\tex_parshape:D
\tex_parskip:D
\tex_patterns:D
\tex_pausing:D
\tex_penalty:D
\tex_postdisplaypenalty:D
\tex_predisplaypenalty:D
\tex_predisplaysize:D
\tex_pretolerance:D
\tex_prevdepth:D
\tex_prevgraf:D
\tex_radical:D
\tex_raise:D
\tex_read:D
\tex_relax:D
\tex_relpolicy:D
\tex_right:D
\tex_righthyphenmin:D
\tex_rightsip:D
\tex_roman numeral:D
\tex_scriptfont:D
\tex_scripts scriptfont:D
\tex_scripts scriptstyle:D
\tex_scriptspace:D
\tex_scriptstyle:D
\tex_scrollmode:D
\tex_setbox:D
\tex_setlanguage:D
\tex_sfcode:D
\tex_shipout:D
\tex_show:D
\tex_showbox:D
\tex_showboxbreadth:D
\tex_showboxdepth:D
\tex_showlists:D
\tex_showthe:D
\tex_skewchar:D
\tex_skip:D
\tex_skipdef:D
\tex_spacefactor:D
\tex_spaceskip:D
\tex_span:D

```

455  \__kernel_primitive:NN \special           \tex_special:D
456  \__kernel_primitive:NN \splitbotmark      \tex_splitbotmark:D
457  \__kernel_primitive:NN \splitfirstmark    \tex_splitfirstmark:D
458  \__kernel_primitive:NN \splitmaxdepth     \tex_splitmaxdepth:D
459  \__kernel_primitive:NN \splittopskip      \tex_splittopskip:D
460  \__kernel_primitive:NN \string            \tex_string:D
461  \__kernel_primitive:NN \tabskip           \tex_tabskip:D
462  \__kernel_primitive:NN \textfont          \tex_textfont:D
463  \__kernel_primitive:NN \textstyle          \tex_textstyle:D
464  \__kernel_primitive:NN \the               \tex_the:D
465  \__kernel_primitive:NN \thickmuskip       \tex_thickmuskip:D
466  \__kernel_primitive:NN \thinmuskip        \tex_thinmuskip:D
467  \__kernel_primitive:NN \time              \tex_time:D
468  \__kernel_primitive:NN \toks              \tex_toks:D
469  \__kernel_primitive:NN \toksdef           \tex_toksdef:D
470  \__kernel_primitive:NN \tolerance         \tex_tolerance:D
471  \__kernel_primitive:NN \topmark           \tex_topmark:D
472  \__kernel_primitive:NN \topskip           \tex_topskip:D
473  \__kernel_primitive:NN \tracingcommands   \tex_tracingcommands:D
474  \__kernel_primitive:NN \tracinglostchars  \tex_tracinglostchars:D
475  \__kernel_primitive:NN \tracingmacros     \tex_tracingmacros:D
476  \__kernel_primitive:NN \tracingonline     \tex_tracingonline:D
477  \__kernel_primitive:NN \tracingoutput     \tex_tracingoutput:D
478  \__kernel_primitive:NN \tracingpages      \tex_tracingpages:D
479  \__kernel_primitive:NN \tracingparagraphs  \tex_tracingparagraphs:D
480  \__kernel_primitive:NN \tracingrestores   \tex_tracingrestores:D
481  \__kernel_primitive:NN \tracingstats      \tex_tracingstats:D
482  \__kernel_primitive:NN \uccode            \tex_uccode:D
483  \__kernel_primitive:NN \uchyph             \tex_uchyph:D
484  \__kernel_primitive:NN \underline          \tex_underline:D
485  \__kernel_primitive:NN \uhbox              \tex_uhbox:D
486  \__kernel_primitive:NN \unhcropy          \tex_unhcropy:D
487  \__kernel_primitive:NN \unkern             \tex_unkern:D
488  \__kernel_primitive:NN \unpenalty          \tex_unpenalty:D
489  \__kernel_primitive:NN \unskip              \tex_unskip:D
490  \__kernel_primitive:NN \unvbox              \tex_unvbox:D
491  \__kernel_primitive:NN \unvcopy             \tex_unvcopy:D
492  \__kernel_primitive:NN \uppercase          \tex_uppercase:D
493  \__kernel_primitive:NN \vadjust             \tex_vadjust:D
494  \__kernel_primitive:NN \valign              \tex_valign:D
495  \__kernel_primitive:NN \vbadness            \tex_vbadness:D
496  \__kernel_primitive:NN \vbox                \tex_vbox:D
497  \__kernel_primitive:NN \vcenter             \tex_vcenter:D
498  \__kernel_primitive:NN \vfil                \tex_vfil:D
499  \__kernel_primitive:NN \vfill               \tex_vfill:D
500  \__kernel_primitive:NN \vfiltneg           \tex_vfiltneg:D
501  \__kernel_primitive:NN \vfuzz               \tex_vfuzz:D
502  \__kernel_primitive:NN \voffset             \tex_voffset:D
503  \__kernel_primitive:NN \vrule               \tex_vrule:D
504  \__kernel_primitive:NN \vsized              \tex_vsized:D
505  \__kernel_primitive:NN \vskip               \tex_vskip:D
506  \__kernel_primitive:NN \vsplit              \tex_vsplit:D
507  \__kernel_primitive:NN \vss                 \tex_vss:D
508  \__kernel_primitive:NN \vtop                \tex_vtop:D

```

509	_kernel_primitive:NN \wd	\tex_wd:D
510	_kernel_primitive:NN \widowpenalty	\tex_widowpenalty:D
511	_kernel_primitive:NN \write	\tex_write:D
512	_kernel_primitive:NN \xdef	\tex_xdef:D
513	_kernel_primitive:NN \xleaders	\tex_xleaders:D
514	_kernel_primitive:NN \xspaceskip	\tex_xspaceskip:D
515	_kernel_primitive:NN \year	\tex_year:D
Primitives introduced by ε -TeX.		
516	_kernel_primitive:NN \beginL	\tex_beginL:D
517	_kernel_primitive:NN \beginR	\tex_beginR:D
518	_kernel_primitive:NN \botmarks	\tex_botmarks:D
519	_kernel_primitive:NN \clubpenalties	\tex_clubpenalties:D
520	_kernel_primitive:NN \currentgrouplevel	\tex_currentgrouplevel:D
521	_kernel_primitive:NN \currentgroupype	\tex_currentgroupype:D
522	_kernel_primitive:NN \currentifbranch	\tex_currentifbranch:D
523	_kernel_primitive:NN \currentiflevel	\tex_currentiflevel:D
524	_kernel_primitive:NN \currentiftype	\tex_currentiftype:D
525	_kernel_primitive:NN \detokenize	\tex_detokenize:D
526	_kernel_primitive:NN \dimexpr	\tex_dimexpr:D
527	_kernel_primitive:NN \displaywidowpenalties	\tex_displaywidowpenalties:D
528	_kernel_primitive:NN \endL	\tex_endL:D
529	_kernel_primitive:NN \endR	\tex_endR:D
530	_kernel_primitive:NN \eTeXrevision	\tex_eTeXrevision:D
531	_kernel_primitive:NN \eTeXversion	\tex_eTeXversion:D
532	_kernel_primitive:NN \everyeof	\tex_everyeof:D
533	_kernel_primitive:NN \firstmarks	\tex_firstmarks:D
534	_kernel_primitive:NN \fontchardp	\tex_fontchardp:D
535	_kernel_primitive:NN \fontcharht	\tex_fontcharht:D
536	_kernel_primitive:NN \fontcharic	\tex_fontcharic:D
537	_kernel_primitive:NN \fontcharwd	\tex_fontcharwd:D
538	_kernel_primitive:NN \glueexpr	\tex_glueexpr:D
539	_kernel_primitive:NN \glueshrink	\tex_glueshrink:D
540	_kernel_primitive:NN \glueshrinkorder	\tex_glueshrinkorder:D
541	_kernel_primitive:NN \gluestretch	\tex_gluestretch:D
542	_kernel_primitive:NN \gluestretchorder	\tex_gluestretchorder:D
543	_kernel_primitive:NN \gluetomu	\tex_gluetomu:D
544	_kernel_primitive:NN \ifcsname	\tex_ifcsname:D
545	_kernel_primitive:NN \ifdefined	\tex_ifdefined:D
546	_kernel_primitive:NN \iffontchar	\tex_iffontchar:D
547	_kernel_primitive:NN \interactionmode	\tex_interactionmode:D
548	_kernel_primitive:NN \interlinepenalties	\tex_interlinepenalties:D
549	_kernel_primitive:NN \lastlinefit	\tex_lastlinefit:D
550	_kernel_primitive:NN \lastnodetype	\tex_lastnodetype:D
551	_kernel_primitive:NN \marks	\tex_marks:D
552	_kernel_primitive:NN \middle	\tex_middle:D
553	_kernel_primitive:NN \muexpr	\tex_muexpr:D
554	_kernel_primitive:NN \mutoglue	\tex_mutoglue:D
555	_kernel_primitive:NN \numexpr	\tex_numexpr:D
556	_kernel_primitive:NN \pagediscards	\tex_pagediscards:D
557	_kernel_primitive:NN \parshapedimen	\tex_parshapedimen:D
558	_kernel_primitive:NN \parshapeindent	\tex_parshapeindent:D
559	_kernel_primitive:NN \parshapelen	\tex_parshapelen:D
560	_kernel_primitive:NN \predisplaydirection	\tex_predisplaydirection:D
561	_kernel_primitive:NN \protected	\tex_protected:D

```

562  \__kernel_primitive:NN \readline          \tex_readline:D
563  \__kernel_primitive:NN \savinghyphcodes \tex_savinghyphcodes:D
564  \__kernel_primitive:NN \savingvdiscards \tex_savingvdiscards:D
565  \__kernel_primitive:NN \scantokens       \tex_scantokens:D
566  \__kernel_primitive:NN \showgroups      \tex_showgroups:D
567  \__kernel_primitive:NN \showifs         \tex_showifs:D
568  \__kernel_primitive:NN \showtokens      \tex_showtokens:D
569  \__kernel_primitive:NN \splitbotmarks   \tex_splitbotmarks:D
570  \__kernel_primitive:NN \splitediscards  \tex_splitediscards:D
571  \__kernel_primitive:NN \splitfirstmarks \tex_splitfirstmarks:D
572  \__kernel_primitive:NN \TeXXeTstate     \tex_TeXXeTstate:D
573  \__kernel_primitive:NN \topmarks        \tex_topmarks:D
574  \__kernel_primitive:NN \tracingassigns  \tex_tracingassigns:D
575  \__kernel_primitive:NN \tracinggroups   \tex_tracinggroups:D
576  \__kernel_primitive:NN \tracingifs      \tex_tracingifs:D
577  \__kernel_primitive:NN \tracingnesting  \tex_tracingnesting:D
578  \__kernel_primitive:NN \tracingscantokens \tex_tracingscantokens:D
579  \__kernel_primitive:NN \unexpanded      \tex_unexpanded:D
580  \__kernel_primitive:NN \unless          \tex_unless:D
581  \__kernel_primitive:NN \widowpenalties \tex_widowpenalties:D

```

Post- ε -TEX primitives do not always end up with the same name in all engines, if indeed they are available cross-engine anyway. We therefore take the approach of preferring the shortest name that makes sense. First, we deal with the primitives introduced by pdfTEX which directly relate to PDF output: these are copied with the names unchanged.

```

582  \__kernel_primitive:NN \pdfannot        \tex_pdfannot:D
583  \__kernel_primitive:NN \pdfcatalog      \tex_pdfcatalog:D
584  \__kernel_primitive:NN \pdfcompresslevel \tex_pdfcompresslevel:D
585  \__kernel_primitive:NN \pdfcolorstack   \tex_pdfcolorstack:D
586  \__kernel_primitive:NN \pdfcolorstackinit \tex_pdfcolorstackinit:D
587  \__kernel_primitive:NN \pdfcreationdate \tex_pdfcreationdate:D
588  \__kernel_primitive:NN \pdfdecimaldigits \tex_pdfdecimaldigits:D
589  \__kernel_primitive:NN \pdfdest         \tex_pdfdest:D
590  \__kernel_primitive:NN \pdfdestmargin   \tex_pdfdestmargin:D
591  \__kernel_primitive:NN \pdfendlink      \tex_pdfendlink:D
592  \__kernel_primitive:NN \pdfendthread    \tex_pdfendthread:D
593  \__kernel_primitive:NN \pdffontattr     \tex_pdffontattr:D
594  \__kernel_primitive:NN \pdffontname     \tex_pdffontname:D
595  \__kernel_primitive:NN \pdffontobjnum   \tex_pdffontobjnum:D
596  \__kernel_primitive:NN \pdfgamma        \tex_pdfgamma:D
597  \__kernel_primitive:NN \pdfimageapplygamma \tex_pdfimageapplygamma:D
598  \__kernel_primitive:NN \pdfimagegamma    \tex_pdfimagegamma:D
599  \__kernel_primitive:NN \pdfgentounicode \tex_pdfgentounicode:D
600  \__kernel_primitive:NN \pdfglyptounicode \tex_pdfglyptounicode:D
601  \__kernel_primitive:NN \pdfhorigin      \tex_pdfhorigin:D
602  \__kernel_primitive:NN \pdfimagehicolor \tex_pdfimagehicolor:D
603  \__kernel_primitive:NN \pdfimageresolution \tex_pdfimageresolution:D
604  \__kernel_primitive:NN \pdfincludechars \tex_pdfincludechars:D
605  \__kernel_primitive:NN \pdfinclusioncopyfonts \tex_pdfinclusioncopyfonts:D
606  \__kernel_primitive:NN \pdfinclusionerrorlevel \tex_pdfinclusionerrorlevel:D
607  \__kernel_primitive:NN \pdfinfo          \tex_pdfinfo:D
608  \__kernel_primitive:NN \pdflastannot    \tex_pdflastannot:D
609  \__kernel_primitive:NN \pdflastlink     \tex_pdflastlink:D

```

```

611  \__kernel_primitive:NN \pdflastobj                \tex_pdflastobj:D
612  \__kernel_primitive:NN \pdflastxform              \tex_pdflastxform:D
613  \__kernel_primitive:NN \pdflastximage              \tex_pdflastximage:D
614  \__kernel_primitive:NN \pdflastximagecolordepth   \tex_pdflastximagecolordepth:D
615    \tex_pdflastximagecolordepth:D
616  \__kernel_primitive:NN \pdflastximagepages        \tex_pdflastximagepages:D
617  \__kernel_primitive:NN \pdflinkmargin            \tex_pdflinkmargin:D
618  \__kernel_primitive:NN \pdfliteral                \tex_pdfliteral:D
619  \__kernel_primitive:NN \pdfmajorversion          \tex_pdfmajorversion:D
620  \__kernel_primitive:NN \pdfminorversion          \tex_pdfminorversion:D
621  \__kernel_primitive:NN \pdfnames                 \tex_pdfnames:D
622  \__kernel_primitive:NN \pdfobj                  \tex_pdfobj:D
623  \__kernel_primitive:NN \pdfobjcompresslevel      \tex_pdfobjcompresslevel:D
624  \__kernel_primitive:NN \pdfoutline               \tex_pdfoutline:D
625  \__kernel_primitive:NN \pdfoutput                \tex_pdfoutput:D
626  \__kernel_primitive:NN \pdfpageattr              \tex_pdfpageattr:D
627  \__kernel_primitive:NN \pdfpagesattr            \tex_pdfpagesattr:D
628  \__kernel_primitive:NN \pdfpagebox               \tex_pdfpagebox:D
629  \__kernel_primitive:NN \pdfpageref                \tex_pdfpageref:D
630  \__kernel_primitive:NN \pdfpageresources         \tex_pdfpageresources:D
631  \__kernel_primitive:NN \pdfpagesattr            \tex_pdfpagesattr:D
632  \__kernel_primitive:NN \pdfrefobj               \tex_pdfrefobj:D
633  \__kernel_primitive:NN \pdfrefxform             \tex_pdfrefxform:D
634  \__kernel_primitive:NN \pdfrefximage            \tex_pdfrefximage:D
635  \__kernel_primitive:NN \pdfrestore               \tex_pdfrestore:D
636  \__kernel_primitive:NN \pdfretval                \tex_pdfretval:D
637  \__kernel_primitive:NN \pdfsave                 \tex_pdfsave:D
638  \__kernel_primitive:NN \pdfsetmatrix            \tex_pdfsetmatrix:D
639  \__kernel_primitive:NN \pdfstartlink           \tex_pdfstartlink:D
640  \__kernel_primitive:NN \pdfstartthread          \tex_pdfstartthread:D
641  \__kernel_primitive:NN \pdfsuppressptexinfo       \tex_pdfsuppressptexinfo:D
642  \__kernel_primitive:NN \pdfthread               \tex_pdfthread:D
643  \__kernel_primitive:NN \pdfthreadmargin         \tex_pdfthreadmargin:D
644  \__kernel_primitive:NN \pdftrailer              \tex_pdftrailer:D
645  \__kernel_primitive:NN \pdfuniqueresname        \tex_pdfuniqueresname:D
646  \__kernel_primitive:NN \pdfvorigin              \tex_pdfvorigin:D
647  \__kernel_primitive:NN \pdfxf orm               \tex_pdfxf orm:D
648  \__kernel_primitive:NN \pdfxf ormname           \tex_pdfxf ormname:D
649  \__kernel_primitive:NN \pdfximage               \tex_pdfximage:D
650  \__kernel_primitive:NN \pdfximagebbox           \tex_pdfximagebbox:D

```

These are not related to PDF output and either already appear in other engines without the \pdf_ prefix, or might reasonably do so at some future stage. We therefore drop the leading pdf here.

```

651  \__kernel_primitive:NN \ifpdfabsdim             \tex_ifabsdim:D
652  \__kernel_primitive:NN \ifpdfabsnum            \tex_ifabsnum:D
653  \__kernel_primitive:NN \ifpdfprimitive         \tex_ifprimitive:D
654  \__kernel_primitive:NN \pdfadjustspacing       \tex_adjustspacing:D
655  \__kernel_primitive:NN \pdfcopyfont            \tex_copyfont:D
656  \__kernel_primitive:NN \pdfdraftmode           \tex_draftmode:D
657  \__kernel_primitive:NN \pdfeachlinedepth       \tex_eachlinedepth:D
658  \__kernel_primitive:NN \pdfeachlineheight     \tex_eachlineheight:D
659  \__kernel_primitive:NN \pdfelapsedtime         \tex_elapsedtime:D
660  \__kernel_primitive:NN \pdffirstlineheight     \tex_firstlineheight:D

```

```

661  \__kernel_primitive:NN \pdffontexpand          \tex_fontexpand:D
662  \__kernel_primitive:NN \pdffontsize           \tex_fontsize:D
663  \__kernel_primitive:NN \pdfignoreddimen      \tex_ignoreddimen:D
664  \__kernel_primitive:NN \pdfinsertht          \tex_insertht:D
665  \__kernel_primitive:NN \pdflastlinedepth     \tex_lastlinedepth:D
666  \__kernel_primitive:NN \pdflastxpos          \tex_lastxpos:D
667  \__kernel_primitive:NN \pdflastypos          \tex_lastypos:D
668  \__kernel_primitive:NN \pdfmapfile           \tex_mapfile:D
669  \__kernel_primitive:NN \pdfmapline            \tex_mapline:D
670  \__kernel_primitive:NN \pdfnoligatures       \tex_noligatures:D
671  \__kernel_primitive:NN \pdfnormaldeviate     \tex_normaldeviate:D
672  \__kernel_primitive:NN \pdfpageheight        \tex_pageheight:D
673  \__kernel_primitive:NN \pdfpagewidth         \tex_pagewidth:D
674  \__kernel_primitive:NN \pdfpkmode            \tex_pkmode:D
675  \__kernel_primitive:NN \pdfpkresolution      \tex_pkresolution:D
676  \__kernel_primitive:NN \pdfprimitive          \tex_primitive:D
677  \__kernel_primitive:NN \pdfprotrudechars     \tex_protrudechars:D
678  \__kernel_primitive:NN \pdfpxdimen           \tex_pxdimen:D
679  \__kernel_primitive:NN \pdfrandomseed        \tex_randomseed:D
680  \__kernel_primitive:NN \pdfresettimer        \tex_resettimer:D
681  \__kernel_primitive:NN \pdfsavepos           \tex_savepos:D
682  \__kernel_primitive:NN \pdfsetrandomseed      \tex_setrandomseed:D
683  \__kernel_primitive:NN \pdfshellescape       \tex_shellescape:D
684  \__kernel_primitive:NN \pdftracingfonts      \tex_tracingfonts:D
685  \__kernel_primitive:NN \pdfuniformdeviate    \tex_uniformdeviate:D

```

The version primitives are not related to PDF mode but are pdfTeX-specific, so again are carried forward unchanged.

```

686  \__kernel_primitive:NN \pdftexbanner          \tex_pdftexbanner:D
687  \__kernel_primitive:NN \pdftexrevision        \tex_pdftexrevision:D
688  \__kernel_primitive:NN \pdftexversion         \tex_pdftexversion:D

```

These ones appear in pdfTeX but don't have pdf in the name at all: no decisions to make.

```

689  \__kernel_primitive:NN \efcode                \tex_efcode:D
690  \__kernel_primitive:NN \ifincsname           \tex_ifincsname:D
691  \__kernel_primitive:NN \leftmarginkern       \tex_leftmarginkern:D
692  \__kernel_primitive:NN \letterspacefont      \tex_letterspacefont:D
693  \__kernel_primitive:NN \lpcode                \tex_lpcode:D
694  \__kernel_primitive:NN \quitvmode            \tex_quitvmode:D
695  \__kernel_primitive:NN \rightmarginkern      \tex_rightmarginkern:D
696  \__kernel_primitive:NN \rancode               \tex_rancode:D
697  \__kernel_primitive:NN \synctex              \tex_synctex:D
698  \__kernel_primitive:NN \tagcode               \tex_tagcode:D

```

Post pdfTeX primitive availability gets more complex. Both XeTeX and LuaTeX have varying names for some primitives from pdfTeX. Particularly for LuaTeX tracking all of that would be hard. Instead, we now check that we only save primitives if they actually exist.

```

699  </names | package>
700  /*package)
701  \tex_long:D \tex_def:D \use_i:nn #1#2 {#2}
702  \tex_long:D \tex_def:D \use_none:n #1 { }
703  \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
704  {

```

```

705   \tex_ifdefined:D #1
706     \tex_expandafter:D \use_ii:nn
707   \tex_fi:D
708     \use_none:n { \tex_global:D \tex_let:D #2 #1 }
709   }
710 
```

`</package>`

`(*names | package)`

Some pdfTeX primitives are handled here because they got dropped in LuaTeX but the corresponding internal names are emulated later. The Lua code is already loaded at this point, so we shouldn't overwrite them.

712 __kernel_primitive:NN \pdfstrcmp	\tex_strcmp:D
713 __kernel_primitive:NN \pdffilesize	\tex_filesize:D
714 __kernel_primitive:NN \pdfmdfivesum	\tex_mdfivesum:D
715 __kernel_primitive:NN \pdffilemoddate	\tex_filemoddate:D
716 __kernel_primitive:NN \pdffiledump	\tex_filedump:D

XeTeX-specific primitives. Note that XeTeX's `\strcmp` is handled earlier and is “rolled up” into `\pdfstrcmp`. A few cross-compatibility names which lack the `pdf` of the original are handled later.

717 __kernel_primitive:NN \suppressfontnotfounderror	\tex_suppressfontnotfounderror:D
718 \tex_suppressfontnotfounderror:D	
719 __kernel_primitive:NN \XeTeXcharclass	\tex_XeTeXcharclass:D
720 __kernel_primitive:NN \XeTeXcharglyph	\tex_XeTeXcharglyph:D
721 __kernel_primitive:NN \XeTeXcountfeatures	\tex_XeTeXcountfeatures:D
722 __kernel_primitive:NN \XeTeXcountglyphs	\tex_XeTeXcountglyphs:D
723 __kernel_primitive:NN \XeTeXcountselectors	\tex_XeTeXcountselectors:D
724 __kernel_primitive:NN \XeTeXcountvariations	\tex_XeTeXcountvariations:D
725 __kernel_primitive:NN \XeTeXdefaultencoding	\tex_XeTeXdefaultencoding:D
726 __kernel_primitive:NN \XeTeXdashbreakstate	\tex_XeTeXdashbreakstate:D
727 __kernel_primitive:NN \XeTeXfeaturecode	\tex_XeTeXfeaturecode:D
728 __kernel_primitive:NN \XeTeXfeaturename	\tex_XeTeXfeaturename:D
729 __kernel_primitive:NN \XeTeXfindfeaturebyname	
730 \tex_XeTeXfindfeaturebyname:D	
731 __kernel_primitive:NN \XeTeXfindselectorbyname	
732 \tex_XeTeXfindselectorbyname:D	
733 __kernel_primitive:NN \XeTeXfindvariationbyname	
734 \tex_XeTeXfindvariationbyname:D	
735 __kernel_primitive:NN \XeTeXfirstfontchar	\tex_XeTeXfirstfontchar:D
736 __kernel_primitive:NN \XeTeXfonttype	\tex_XeTeXfonttype:D
737 __kernel_primitive:NN \XeTeXgenerateactualtext	
738 \tex_XeTeXgenerateactualtext:D	
739 __kernel_primitive:NN \XeTeXglyph	\tex_XeTeXglyph:D
740 __kernel_primitive:NN \XeTeXglyphbounds	\tex_XeTeXglyphbounds:D
741 __kernel_primitive:NN \XeTeXglyphindex	\tex_XeTeXglyphindex:D
742 __kernel_primitive:NN \XeTeXglyphname	\tex_XeTeXglyphname:D
743 __kernel_primitive:NN \XeTeXinputencoding	\tex_XeTeXinputencoding:D
744 __kernel_primitive:NN \XeTeXinputnormalization	
745 \tex_XeTeXinputnormalization:D	
746 __kernel_primitive:NN \XeTeXinterchartokenstate	
747 \tex_XeTeXinterchartokenstate:D	
748 __kernel_primitive:NN \XeTeXinterchartoks	\tex_XeTeXinterchartoks:D
749 __kernel_primitive:NN \XeTeXisdefaultselector	
750 \tex_XeTeXisdefaultselector:D	
751 __kernel_primitive:NN \XeTeXisexclusivefeature	

```

752     \tex_XeTeXisexclusivefeature:D
753     \_kernel_primitive:NN \XeTeXlastfontchar      \tex_XeTeXlastfontchar:D
754     \_kernel_primitive:NN \XeTeXlinebreakskip    \tex_XeTeXlinebreakskip:D
755     \_kernel_primitive:NN \XeTeXlinebreaklocale   \tex_XeTeXlinebreaklocale:D
756     \_kernel_primitive:NN \XeTeXlinebreakpenalty  \tex_XeTeXlinebreakpenalty:D
757     \_kernel_primitive:NN \XeTeXOTcountfeatures   \tex_XeTeXOTcountfeatures:D
758     \_kernel_primitive:NN \XeTeXOTcountlanguages  \tex_XeTeXOTcountlanguages:D
759     \_kernel_primitive:NN \XeTeXOTcountsheets    \tex_XeTeXOTcountsheets:D
760     \_kernel_primitive:NN \XeTeXOTfeaturetag     \tex_XeTeXOTfeaturetag:D
761     \_kernel_primitive:NN \XeTeXOTlanguage tag   \tex_XeTeXOTlanguage tag:D
762     \_kernel_primitive:NN \XeTeXOTscripttag     \tex_XeTeXOTscripttag:D
763     \_kernel_primitive:NN \XeTeXpdf file       \tex_XeTeXpdf file:D
764     \_kernel_primitive:NN \XeTeXpdf pagecount   \tex_XeTeXpdf pagecount:D
765     \_kernel_primitive:NN \XeTeXpic file       \tex_XeTeXpic file:D
766     \_kernel_primitive:NN \XeTeXrevision       \tex_XeTeXrevision:D
767     \_kernel_primitive:NN \XeTeXselectorname    \tex_XeTeXselectorname:D
768     \_kernel_primitive:NN \XeTeXtracingfonts   \tex_XeTeXtracingfonts:D
769     \_kernel_primitive:NN \XeTeXupwardsmode    \tex_XeTeXupwardsmode:D
770     \_kernel_primitive:NN \XeTeXuseglyphmetrics \tex_XeTeXuseglyphmetrics:D
771     \_kernel_primitive:NN \XeTeXvariation      \tex_XeTeXvariation:D
772     \_kernel_primitive:NN \XeTeXvariationdefault \tex_XeTeXvariationdefault:D
773     \_kernel_primitive:NN \XeTeXvariationmax   \tex_XeTeXvariationmax:D
774     \_kernel_primitive:NN \XeTeXvariationmin   \tex_XeTeXvariationmin:D
775     \_kernel_primitive:NN \XeTeXvariationname  \tex_XeTeXvariationname:D
776     \_kernel_primitive:NN \XeTeXversion       \tex_XeTeXversion:D

```

Primitives from pdfTeX that XeTeX renames: also helps with LuaTeX.

```

777     \_kernel_primitive:NN \creationdate        \tex_creationdate:D
778     \_kernel_primitive:NN \elapsedtime       \tex_elapsedtime:D
779     \_kernel_primitive:NN \filedump          \tex_filedump:D
780     \_kernel_primitive:NN \filemoddate      \tex_filemoddate:D
781     \_kernel_primitive:NN \filesize          \tex_filesize:D
782     \_kernel_primitive:NN \mdfivesum        \tex_mdfivesum:D
783     \_kernel_primitive:NN \ifprimitive     \tex_ifprimitive:D
784     \_kernel_primitive:NN \primitive        \tex_primitive:D
785     \_kernel_primitive:NN \resettimer       \tex_resettimer:D
786     \_kernel_primitive:NN \shellescape      \tex_shellescape:D

```

Primitives from LuaTeX, some of which have been ported back to XeTeX.

```

787     \_kernel_primitive:NN \alignmark         \tex_alignmark:D
788     \_kernel_primitive:NN \aligntab         \tex_aligntab:D
789     \_kernel_primitive:NN \attribute        \tex_attribute:D
790     \_kernel_primitive:NN \attributedef    \tex_attributedef:D
791     \_kernel_primitive:NN \automaticdiscretionary \tex_automaticdiscretionary:D
792     \_kernel_primitive:NN \automatichyphenmode \tex_automatichyphenmode:D
793     \_kernel_primitive:NN \automatichyphenpenalty \tex_automatichyphenpenalty:D
794     \_kernel_primitive:NN \begincsname     \tex_begincsname:D
795     \_kernel_primitive:NN \bodydir         \tex_bodydir:D
796     \_kernel_primitive:NN \bodydirection   \tex_bodydirection:D
797     \_kernel_primitive:NN \boxdir          \tex_boxdir:D
798     \_kernel_primitive:NN \boxdirection    \tex_boxdirection:D
799     \_kernel_primitive:NN \breakafterdirmode \tex_breakafterdirmode:D
800     \_kernel_primitive:NN \catcodetable   \tex_catcodetable:D
801

```

```

803  \__kernel_primitive:NN \clearmarks           \tex_clearmarks:D
804  \__kernel_primitive:NN \crampeddisplaystyle \tex_crampeddisplaystyle:D
805  \__kernel_primitive:NN \crampedscriptstyle   \tex_crampedscriptstyle:D
806    \tex_crampedscriptscriptstyle:D
807  \__kernel_primitive:NN \crampedscriptstyle   \tex_crampedscriptstyle:D
808  \__kernel_primitive:NN \crampedtextstyle    \tex_crampedtextstyle:D
809  \__kernel_primitive:NN \csstring            \tex_csstring:D
810  \__kernel_primitive:NN \directlua           \tex_directlua:D
811  \__kernel_primitive:NN \dviextension        \tex_dviextension:D
812  \__kernel_primitive:NN \dvifeedback         \tex_dvifeedback:D
813  \__kernel_primitive:NN \dvivariable         \tex_dvivariable:D
814  \__kernel_primitive:NN \eTeXglueshrinkorder \tex_eTeXglueshrinkorder:D
815  \__kernel_primitive:NN \eTeXgluestretchorder \tex_eTeXgluestretchorder:D
816  \__kernel_primitive:NN \etoksapp            \tex_etoksapp:D
817  \__kernel_primitive:NN \etokspre             \tex_etokspre:D
818  \__kernel_primitive:NN \exceptionpenalty    \tex_exceptionpenalty:D
819  \__kernel_primitive:NN \explicithyphenpenalty \tex_explicithyphenpenalty:D
820  \__kernel_primitive:NN \expanded              \tex_expanded:D
821  \__kernel_primitive:NN \explicitdiscretionary \tex_explicitdiscretionary:D
822  \__kernel_primitive:NN \firstvalidlanguage   \tex_firstvalidlanguage:D
823  \__kernel_primitive:NN \fontid                \tex_fontid:D
824  \__kernel_primitive:NN \formatname          \tex_formatname:D
825  \__kernel_primitive:NN \hjcode               \tex_hjcode:D
826  \__kernel_primitive:NN \hpack                 \tex_hpack:D
827  \__kernel_primitive:NN \hyphenationbounds   \tex_hyphenationbounds:D
828  \__kernel_primitive:NN \hyphenationmin      \tex_hyphenationmin:D
829  \__kernel_primitive:NN \hyphenpenaltymode   \tex_hyphenpenaltymode:D
830  \__kernel_primitive:NN \gleaders             \tex_gleaders:D
831  \__kernel_primitive:NN \ifcondition         \tex_ifcondition:D
832  \__kernel_primitive:NN \immediateassigned   \tex_immediateassigned:D
833  \__kernel_primitive:NN \immediateassignment \tex_immediateassignment:D
834  \__kernel_primitive:NN \initcatcodetable   \tex_initcatcodetable:D
835  \__kernel_primitive:NN \lastnamedcs        \tex_lastnamedcs:D
836  \__kernel_primitive:NN \latelua              \tex_latelua:D
837  \__kernel_primitive:NN \lateluafunction     \tex_lateluafunction:D
838  \__kernel_primitive:NN \leftghost            \tex_leftghost:D
839  \__kernel_primitive:NN \letcharcode          \tex_letcharcode:D
840  \__kernel_primitive:NN \linedir              \tex_linedir:D
841  \__kernel_primitive:NN \linedirection       \tex_linedirection:D
842  \__kernel_primitive:NN \localbrokenpenalty  \tex_localbrokenpenalty:D
843  \__kernel_primitive:NN \localinterlinepenalty \tex_localinterlinepenalty:D
844  \__kernel_primitive:NN \luabytocode         \tex_luabytocode:D
845  \__kernel_primitive:NN \luabytocodecall    \tex_luabytocodecall:D
846  \__kernel_primitive:NN \luacopyinputnodes   \tex_luacopyinputnodes:D
847  \__kernel_primitive:NN \luadef               \tex_luadef:D
848  \__kernel_primitive:NN \localleftbox        \tex_localleftbox:D
849  \__kernel_primitive:NN \localrightbox       \tex_localrightbox:D
850  \__kernel_primitive:NN \luaescapestring    \tex_luaescapestring:D
851  \__kernel_primitive:NN \luafunction         \tex_luafunction:D
852  \__kernel_primitive:NN \luafunctioncall    \tex_luafunctioncall:D
853  \__kernel_primitive:NN \luatexbanner       \tex_luatexbanner:D
854  \__kernel_primitive:NN \luatexrevision     \tex_luatexrevision:D
855  \__kernel_primitive:NN \luatexversion      \tex_luatexversion:D
856  \__kernel_primitive:NN \mathdelimitersmode  \tex_mathdelimitersmode:D

```

```

857  \__kernel_primitive:NN \mathdir          \tex_mathdir:D
858  \__kernel_primitive:NN \mathdirection   \tex_mathdirection:D
859  \__kernel_primitive:NN \mathdisplayskipmode \tex_mathdisplayskipmode:D
860  \__kernel_primitive:NN \matheqnogapstep  \tex_matheqnogapstep:D
861  \__kernel_primitive:NN \mathnolimitsmode   \tex_mathnolimitsmode:D
862  \__kernel_primitive:NN \mathoption        \tex_mathoption:D
863  \__kernel_primitive:NN \mathpenaltiesmode \tex_mathpenaltiesmode:D
864  \__kernel_primitive:NN \mathrulesfam      \tex_mathrulesfam:D
865  \__kernel_primitive:NN \mathscriptsmode   \tex_mathscriptsmode:D
866  \__kernel_primitive:NN \mathscriptboxmode \tex_mathscriptboxmode:D
867  \__kernel_primitive:NN \mathscriptcharche mode \tex_mathscriptcharche mode:D
868  \__kernel_primitive:NN \mathstyle         \tex_mathstyle:D
869  \__kernel_primitive:NN \mathsurroundmode   \tex_mathsurroundmode:D
870  \__kernel_primitive:NN \mathsurroundskip   \tex_mathsurroundskip:D
871  \__kernel_primitive:NN \nohrule          \tex_nohtilde:D
872  \__kernel_primitive:NN \nokerns          \tex_nokerns:D
873  \__kernel_primitive:NN \noligs           \tex_noligs:D
874  \__kernel_primitive:NN \nospaces         \tex_nospaces:D
875  \__kernel_primitive:NN \novrule          \tex_novrule:D
876  \__kernel_primitive:NN \outputbox        \tex_outputbox:D
877  \__kernel_primitive:NN \pagebottomoffset \tex_pagebottomoffset:D
878  \__kernel_primitive:NN \pagedir          \tex_pagedir:D
879  \__kernel_primitive:NN \pagedirection    \tex_pagedirection:D
880  \__kernel_primitive:NN \pageleftoffset   \tex_pageleftoffset:D
881  \__kernel_primitive:NN \pagerightoffset  \tex_pagerightoffset:D
882  \__kernel_primitive:NN \pagetopoffset    \tex_pagetopoffset:D
883  \__kernel_primitive:NN \pardir          \tex_pardir:D
884  \__kernel_primitive:NN \pardirection    \tex_pardirection:D
885  \__kernel_primitive:NN \pdfextension    \tex_pdfextension:D
886  \__kernel_primitive:NN \pdffeedback    \tex_pdffeedback:D
887  \__kernel_primitive:NN \pdfvariable     \tex_pdfvariable:D
888  \__kernel_primitive:NN \postexhyphenchar \tex_postexhyphenchar:D
889  \__kernel_primitive:NN \posthyphenchar   \tex_posthyphenchar:D
890  \__kernel_primitive:NN \prebinopenpenalty \tex_prebinopenpenalty:D
891  \__kernel_primitive:NN \predisplaygapfactor \tex_predisplaygapfactor:D
892  \__kernel_primitive:NN \preexhyphenchar  \tex_preexhyphenchar:D
893  \__kernel_primitive:NN \prehyphenchar    \tex_prehyphenchar:D
894  \__kernel_primitive:NN \prerelpenalty   \tex_prerelpenalty:D
895  \__kernel_primitive:NN \rightghost      \tex_rightghost:D
896  \__kernel_primitive:NN \savecatcodetable \tex_savecatcodetable:D
897  \__kernel_primitive:NN \scantexttokens  \tex_scantexttokens:D
898  \__kernel_primitive:NN \setfontid       \tex_setfontid:D
899  \__kernel_primitive:NN \shapemode       \tex_shapemode:D
900  \__kernel_primitive:NN \suppressifcsnameerror \tex_suppressifcsnameerror:D
901  \__kernel_primitive:NN \suppresslongerror \tex_suppresslongerror:D
902  \__kernel_primitive:NN \suppressmathparerror \tex_suppressmathparerror:D
903  \__kernel_primitive:NN \suppressoutererror \tex_suppressoutererror:D
904  \__kernel_primitive:NN \suppressprimitiveerror \tex_suppressprimitiveerror:D
905  \__kernel_primitive:NN \tex_suppressprimitiveerror:D
906  \__kernel_primitive:NN \textdir          \tex_textdir:D
907  \__kernel_primitive:NN \textdirection   \tex_textdirection:D
908  \__kernel_primitive:NN \toksapp         \tex_toksapp:D
909  \__kernel_primitive:NN \tokspre         \tex_tokspre:D
910  \__kernel_primitive:NN \tpack           \tex_tpck:D

```

```

911  \__kernel_primitive:NN \vpack           \tex_vpack:D
Primitives from pdfTeX that LuaTeX renames.
912  \__kernel_primitive:NN \adjustspacing    \tex_adjustspacing:D
913  \__kernel_primitive:NN \copyfont         \tex_copyfont:D
914  \__kernel_primitive:NN \draftmode        \tex_draftmode:D
915  \__kernel_primitive:NN \expandglyphsinfont \tex_fontexpand:D
916  \__kernel_primitive:NN \ifabsdim         \tex_ifabsdim:D
917  \__kernel_primitive:NN \ifabsnum         \tex_ifabsnum:D
918  \__kernel_primitive:NN \ignoreregularitiesinfont \tex_ignoreregularitiesinfont:D
919  \__kernel_primitive:NN \insertht          \tex_insertht:D
920  \__kernel_primitive:NN \lastsavedboxresourceindex \tex_pdflastxform:D
921  \__kernel_primitive:NN \lastsavedimageresourceindex \tex_pdflastximage:D
922  \__kernel_primitive:NN \lastsavedimageresourcepages \tex_pdflastximagepages:D
923  \__kernel_primitive:NN \lastsavedimageresourcepages \tex_pdflastximage:D
924  \__kernel_primitive:NN \lastxpos           \tex_lastxpos:D
925  \__kernel_primitive:NN \lastypos           \tex_lastypos:D
926  \__kernel_primitive:NN \normaldeviate     \tex_normaldeviate:D
927  \__kernel_primitive:NN \outputmode        \tex_pdfoutput:D
928  \__kernel_primitive:NN \pageheight        \tex_pageheight:D
929  \__kernel_primitive:NN \pagewidth         \tex_pagewidth:D
930  \__kernel_primitive:NN \protrudechars    \tex_protrudechars:D
931  \__kernel_primitive:NN \pxdimen           \tex_pxdimen:D
932  \__kernel_primitive:NN \randomseed        \tex_randomseed:D
933  \__kernel_primitive:NN \useboxresource    \tex_pdfrefxform:D
934  \__kernel_primitive:NN \useimageresource   \tex_pdfrefximage:D
935  \__kernel_primitive:NN \savepos            \tex_savepos:D
936  \__kernel_primitive:NN \saveboxresource    \tex_pdfxform:D
937  \__kernel_primitive:NN \saveimageresource   \tex_pdfximage:D
938  \__kernel_primitive:NN \setrandomseed      \tex_setrandomseed:D
939  \__kernel_primitive:NN \tracingfonts       \tex_tracingfonts:D
940  \__kernel_primitive:NN \uniformdeviate     \tex_uniformdeviate:D
941  \__kernel_primitive:NN \uniformdeviate     \tex_uniformdeviate:D
942  \__kernel_primitive:NN \uniformdeviate     \tex_uniformdeviate:D

```

The set of Unicode math primitives were introduced by XeTeX and LuaTeX in a somewhat complex fashion: a few first as \XeTeX... which were then renamed with LuaTeX having a lot more. These names now all start \U... and mainly \Umath....

```

943  \__kernel_primitive:NN \Uchar           \tex_Uchar:D
944  \__kernel_primitive:NN \Ucharcat        \tex_Ucharcat:D
945  \__kernel_primitive:NN \Udelcode        \tex_Udelcode:D
946  \__kernel_primitive:NN \Udelcodenum    \tex_Udelcodenum:D
947  \__kernel_primitive:NN \Udelimiter     \tex_Udelimiter:D
948  \__kernel_primitive:NN \Udelimiterover  \tex_Udelimiterover:D
949  \__kernel_primitive:NN \Udelimiterunder \tex_Udelimiterunder:D
950  \__kernel_primitive:NN \Uhextensible   \tex_Uhextensible:D
951  \__kernel_primitive:NN \Umathaccent     \tex_Umathaccent:D
952  \__kernel_primitive:NN \Umathaxis        \tex_Umathaxis:D
953  \__kernel_primitive:NN \Umathbinbinspacing \tex_Umathbinbinspacing:D
954  \__kernel_primitive:NN \Umathbinclosespacing \tex_Umathbinclosespacing:D
955  \__kernel_primitive:NN \Umathbininnerspacing \tex_Umathbininnerspacing:D
956  \__kernel_primitive:NN \Umathbinopenspacing \tex_Umathbinopenspacing:D
957  \__kernel_primitive:NN \Umathbinopspacing  \tex_Umathbinopspacing:D
958  \__kernel_primitive:NN \Umathbinordspacing \tex_Umathbinordspacing:D
959  \__kernel_primitive:NN \Umathbinpunctspacing \tex_Umathbinpunctspacing:D

```

```

960  \__kernel_primitive:NN \Umathbinrelspacing      \tex_Umathbinrelspacing:D
961  \__kernel_primitive:NN \Umathchar           \tex_Umathchar:D
962  \__kernel_primitive:NN \Umathcharclass     \tex_Umathcharclass:D
963  \__kernel_primitive:NN \Umathchardef       \tex_Umathchardef:D
964  \__kernel_primitive:NN \Umathcharfam       \tex_Umathcharfam:D
965  \__kernel_primitive:NN \Umathcharnum      \tex_Umathcharnum:D
966  \__kernel_primitive:NN \Umathcharnumdef    \tex_Umathcharnumdef:D
967  \__kernel_primitive:NN \Umathcharslot      \tex_Umathcharslot:D
968  \__kernel_primitive:NN \Umathclosebinspacing \tex_Umathclosebinspacing:D
969  \__kernel_primitive:NN \Umathcloseclosespacing
970          \tex_Umathcloseclosespacing:D
971  \__kernel_primitive:NN \Umathcloseinnerspacing
972          \tex_Umathcloseinnerspacing:D
973  \__kernel_primitive:NN \Umathcloseopenspacing \tex_Umathcloseopenspacing:D
974  \__kernel_primitive:NN \Umathcloseopspacing   \tex_Umathcloseopspacing:D
975  \__kernel_primitive:NN \Umathcloseordspacing  \tex_Umathcloseordspacing:D
976  \__kernel_primitive:NN \Umathclosepunctspacing
977          \tex_Umathclosepunctspacing:D
978  \__kernel_primitive:NN \Umathcloserelspacing   \tex_Umathcloserelspacing:D
979  \__kernel_primitive:NN \Umathcode            \tex_Umathcode:D
980  \__kernel_primitive:NN \Umathcodenum        \tex_Umathcodenum:D
981  \__kernel_primitive:NN \Umathconnectoroverlapmin
982          \tex_Umathconnectoroverlapmin:D
983  \__kernel_primitive:NN \Umathfractiondelsize  \tex_Umathfractiondelsize:D
984  \__kernel_primitive:NN \Umathfractiondenomdown
985          \tex_Umathfractiondenomdown:D
986  \__kernel_primitive:NN \Umathfractiondenomvgap
987          \tex_Umathfractiondenomvgap:D
988  \__kernel_primitive:NN \Umathfractionnumup    \tex_Umathfractionnumup:D
989  \__kernel_primitive:NN \Umathfractionnumvgap  \tex_Umathfractionnumvgap:D
990  \__kernel_primitive:NN \Umathfractionrule     \tex_Umathfractionrule:D
991  \__kernel_primitive:NN \Umathinnerbinspacing  \tex_Umathinnerbinspacing:D
992  \__kernel_primitive:NN \Umathinnerclosespacing
993          \tex_Umathinnerclosespacing:D
994  \__kernel_primitive:NN \Umathinnerinnerspacing
995          \tex_Umathinnerinnerspacing:D
996  \__kernel_primitive:NN \Umathinneropenspacing \tex_Umathinneropenspacing:D
997  \__kernel_primitive:NN \Umathinneropspacing   \tex_Umathinneropspacing:D
998  \__kernel_primitive:NN \Umathinnerordspacing  \tex_Umathinnerordspacing:D
999  \__kernel_primitive:NN \Umathinnerpunctspacing
1000         \tex_Umathinnerpunctspacing:D
1001  \__kernel_primitive:NN \Umathinnerrelopsacing \tex_Umathinnerrelopsacing:D
1002  \__kernel_primitive:NN \Umathlimitabovebgap  \tex_Umathlimitabovebgap:D
1003  \__kernel_primitive:NN \Umathlimitabovekern  \tex_Umathlimitabovekern:D
1004  \__kernel_primitive:NN \Umathlimitabovevgap  \tex_Umathlimitabovevgap:D
1005  \__kernel_primitive:NN \Umathlimitbelowbgap  \tex_Umathlimitbelowbgap:D
1006  \__kernel_primitive:NN \Umathlimitbelowkern  \tex_Umathlimitbelowkern:D
1007  \__kernel_primitive:NN \Umathlimitbelowvgap  \tex_Umathlimitbelowvgap:D
1008  \__kernel_primitive:NN \Umathnolimitssubfactor \tex_Umathnolimitssubfactor:D
1009  \__kernel_primitive:NN \Umathnolimitsupfactor  \tex_Umathnolimitsupfactor:D
1010  \__kernel_primitive:NN \Umathopbinspacing    \tex_Umathopbinspacing:D
1011  \__kernel_primitive:NN \Umathopclosespacing  \tex_Umathopclosespacing:D
1012  \__kernel_primitive:NN \Umathopenbinspacing  \tex_Umathopenbinspacing:D
1013  \__kernel_primitive:NN \Umathopenclosespacing \tex_Umathopenclosespacing:D

```

```

1014  \__kernel_primitive:NN \Umathopeninnerspacing \tex_Umathopeninnerspacing:D
1015  \__kernel_primitive:NN \Umathopenopenspacing \tex_Umathopenopenspacing:D
1016  \__kernel_primitive:NN \Umathopenopspaceing \tex_Umathopenopspaceing:D
1017  \__kernel_primitive:NN \Umathopenordspacing \tex_Umathopenordspacing:D
1018  \__kernel_primitive:NN \Umathopenpunctspacing \tex_Umathopenpunctspacing:D
1019  \__kernel_primitive:NN \Umathopenrelspacing \tex_Umathopenrelspacing:D
1020  \__kernel_primitive:NN \Umathoperatorsize \tex_Umathoperatorsize:D
1021  \__kernel_primitive:NN \Umathopinnerspacing \tex_Umathopinnerspacing:D
1022  \__kernel_primitive:NN \Umathopopenspacing \tex_Umathopopenspacing:D
1023  \__kernel_primitive:NN \Umathopopspaceing \tex_Umathopopspaceing:D
1024  \__kernel_primitive:NN \Umathopordspacing \tex_Umathopordspacing:D
1025  \__kernel_primitive:NN \Umathoppunctspacing \tex_Umathoppunctspacing:D
1026  \__kernel_primitive:NN \Umathoprelepspaceing \tex_Umathoprelepspaceing:D
1027  \__kernel_primitive:NN \Umathordbinspaceing \tex_Umathordbinspaceing:D
1028  \__kernel_primitive:NN \Umathordclosespaceing \tex_Umathordclosespaceing:D
1029  \__kernel_primitive:NN \Umathorddinnerspacing \tex_Umathorddinnerspacing:D
1030  \__kernel_primitive:NN \Umathordopenspaceing \tex_Umathordopenspaceing:D
1031  \__kernel_primitive:NN \Umathordopspacing \tex_Umathordopspacing:D
1032  \__kernel_primitive:NN \Umathordordspacing \tex_Umathordordspacing:D
1033  \__kernel_primitive:NN \Umathordpunctspacing \tex_Umathordpunctspacing:D
1034  \__kernel_primitive:NN \Umathordrelepspaceing \tex_Umathordrelepspaceing:D
1035  \__kernel_primitive:NN \Umathoverbarkern \tex_Umathoverbarkern:D
1036  \__kernel_primitive:NN \Umathoverbarrule \tex_Umathoverbarrule:D
1037  \__kernel_primitive:NN \Umathoverbarvgap \tex_Umathoverbarvgap:D
1038  \__kernel_primitive:NN \Umathoverdelimitergap
        \tex_Umathoverdelimitergap:D
1039  \__kernel_primitive:NN \Umathoverdelimitervgap
        \tex_Umathoverdelimitervgap:D
1040  \__kernel_primitive:NN \Umathpunctbinspaceing \tex_Umathpunctbinspaceing:D
1041  \__kernel_primitive:NN \Umathpunctclosespaceing
        \tex_Umathpunctclosespaceing:D
1042  \__kernel_primitive:NN \Umathpunctinnerspacing \tex_Umathpunctinnerspacing:D
1043  \__kernel_primitive:NN \Umathpuncttinnerrspacing \tex_Umathpuncttinnerrspacing:D
1044  \__kernel_primitive:NN \Umathpuncttopspaceing \tex_Umathpuncttopspaceing:D
1045  \__kernel_primitive:NN \Umathpuncttopspacing \tex_Umathpuncttopspacing:D
1046  \__kernel_primitive:NN \Umathpunctordspacing \tex_Umathpunctordspacing:D
1047  \__kernel_primitive:NN \Umathpunctpunctspacing \tex_Umathpunctpunctspacing:D
1048  \__kernel_primitive:NN \Umathpunctrelspacing \tex_Umathpunctrelspacing:D
1049  \__kernel_primitive:NN \Umathquad \tex_Umathquad:D
1050  \__kernel_primitive:NN \Umathradicaldegreeafter
        \tex_Umathradicaldegreeafter:D
1051  \__kernel_primitive:NN \Umathradicaldegreebefore
        \tex_Umathradicaldegreebefore:D
1052  \__kernel_primitive:NN \Umathradicaldegreeraise
        \tex_Umathradicaldegreeraise:D
1053  \__kernel_primitive:NN \Umathradicalkern \tex_Umathradicalkern:D
1054  \__kernel_primitive:NN \Umathradicalrule \tex_Umathradicalrule:D
1055  \__kernel_primitive:NN \Umathradicalvgap \tex_Umathradicalvgap:D
1056  \__kernel_primitive:NN \Umathrelbinspaceing \tex_Umathrelbinspaceing:D
1057  \__kernel_primitive:NN \Umathrelclosespaceing \tex_Umathrelclosespaceing:D
1058  \__kernel_primitive:NN \Umathrelinnerspacing \tex_Umathrelinnerspacing:D
1059  \__kernel_primitive:NN \Umathrelopeninspace \tex_Umathrelopeninspace:D
1060  \__kernel_primitive:NN \Umathrelopspaceing \tex_Umathrelopspaceing:D

```

```

1068  \__kernel_primitive:NN \Umathrelordspacing      \tex_Umathrelordspacing:D
1069  \__kernel_primitive:NN \Umathrelpunctspacing   \tex_Umathrelpunctspacing:D
1070  \__kernel_primitive:NN \Umathrelrelspacing     \tex_Umathrelrelspacing:D
1071  \__kernel_primitive:NN \Umathskewedfractionhgap
1072    \tex_Umathskewedfractionhgap:D
1073  \__kernel_primitive:NN \Umathskewedfractionvgap
1074    \tex_Umathskewedfractionvgap:D
1075  \__kernel_primitive:NN \Umathspaceafterscript \tex_Umathspaceafterscript:D
1076  \__kernel_primitive:NN \Umathstackdenomdown  \tex_Umathstackdenomdown:D
1077  \__kernel_primitive:NN \Umathstacknumup      \tex_Umathstacknumup:D
1078  \__kernel_primitive:NN \Umathstackvgap       \tex_Umathstackvgap:D
1079  \__kernel_primitive:NN \Umathsubshiftdown   \tex_Umathsubshiftdown:D
1080  \__kernel_primitive:NN \Umathsubshiftdrop   \tex_Umathsubshiftdrop:D
1081  \__kernel_primitive:NN \Umathsubsupshiftdown \tex_Umathsubsupshiftdown:D
1082  \__kernel_primitive:NN \Umathsubsupvgap     \tex_Umathsubsupvgap:D
1083  \__kernel_primitive:NN \Umathsubtopmax     \tex_Umathsubtopmax:D
1084  \__kernel_primitive:NN \Umathsupbottommin  \tex_Umathsupbottommin:D
1085  \__kernel_primitive:NN \Umathsupshiftdrop  \tex_Umathsupshiftdrop:D
1086  \__kernel_primitive:NN \Umathsupshiftup    \tex_Umathsupshiftup:D
1087  \__kernel_primitive:NN \Umathsupsubbottommax \tex_Umathsupsubbottommax:D
1088  \__kernel_primitive:NN \Umathunderbarkern \tex_Umathunderbarkern:D
1089  \__kernel_primitive:NN \Umathunderbarrule \tex_Umathunderbarrule:D
1090  \__kernel_primitive:NN \Umathunderbarvgap \tex_Umathunderbarvgap:D
1091  \__kernel_primitive:NN \Umathunderdelimitergap
1092    \tex_Umathunderdelimitergap:D
1093  \__kernel_primitive:NN \Umathunderdelimitervgap
1094    \tex_Umathunderdelimitervgap:D
1095  \__kernel_primitive:NN \Unosubscript        \tex_Unosubscript:D
1096  \__kernel_primitive:NN \Unosuperscript      \tex_Unosuperscript:D
1097  \__kernel_primitive:NN \Uoverdelimiter     \tex_Uoverdelimiter:D
1098  \__kernel_primitive:NN \Uradical          \tex_Uradical:D
1099  \__kernel_primitive:NN \Uroot              \tex_Uroot:D
1100  \__kernel_primitive:NN \Uskewed            \tex_Uskewed:D
1101  \__kernel_primitive:NN \Uskewedwithdelims \tex_Uskewedwithdelims:D
1102  \__kernel_primitive:NN \Ustack              \tex_Ustack:D
1103  \__kernel_primitive:NN \Ustartdisplaymath \tex_Ustartdisplaymath:D
1104  \__kernel_primitive:NN \Ustartmath         \tex_Ustartmath:D
1105  \__kernel_primitive:NN \Ustopdisplaymath  \tex_Ustopdisplaymath:D
1106  \__kernel_primitive:NN \Ustopmath          \tex_Ustopmath:D
1107  \__kernel_primitive:NN \Usubscript         \tex_Usubscript:D
1108  \__kernel_primitive:NN \Usuperscript       \tex_Usuperscript:D
1109  \__kernel_primitive:NN \Uunderdelimiter   \tex_Uunderdelimiter:D
1110  \__kernel_primitive:NN \Uvextensible      \tex_Uvextensible:D

```

Primitives from pTeX.

```

1111  \__kernel_primitive:NN \autospacing        \tex_autospacing:D
1112  \__kernel_primitive:NN \autoxspacing      \tex_autoxspacing:D
1113  \__kernel_primitive:NN \currentcjktoken \tex_currentcjktoken:D
1114  \__kernel_primitive:NN \currentspacingmode \tex_currentspacingmode:D
1115  \__kernel_primitive:NN \currentxspacingmode \tex_currentxspacingmode:D
1116  \__kernel_primitive:NN \disinhibitglue   \tex_disinhibitglue:D
1117  \__kernel_primitive:NN \dtou              \tex_dtou:D
1118  \__kernel_primitive:NN \epTeXinputencoding \tex_epTeXinputencoding:D
1119  \__kernel_primitive:NN \epTeXversion      \tex_epTeXversion:D
1120  \__kernel_primitive:NN \euc                \tex_euc:D

```

```

1121  \__kernel_primitive:NN \hfi          \tex_hfi:D
1122  \__kernel_primitive:NN \ifdbox       \tex_ifdbox:D
1123  \__kernel_primitive:NN \ifddir       \tex_ifddir:D
1124  \__kernel_primitive:NN \ifjfont       \tex_ifjfont:D
1125  \__kernel_primitive:NN \ifmbox       \tex_ifmbox:D
1126  \__kernel_primitive:NN \ifmdir       \tex_ifmdir:D
1127  \__kernel_primitive:NN \iftbox       \tex_iftbox:D
1128  \__kernel_primitive:NN \iftfont      \tex_iftfont:D
1129  \__kernel_primitive:NN \iftdir       \tex_iftdir:D
1130  \__kernel_primitive:NN \ifybox       \tex_ifybox:D
1131  \__kernel_primitive:NN \ifydir       \tex_ifydir:D
1132  \__kernel_primitive:NN \inhibitglue   \tex_inhibitglue:D
1133  \__kernel_primitive:NN \inhibitxspcode \tex_inhibitxspcode:D
1134  \__kernel_primitive:NN \jcharwidowpenalty \tex_jcharwidowpenalty:D
1135  \__kernel_primitive:NN \jfam          \tex_jfam:D
1136  \__kernel_primitive:NN \jfont          \tex_jfont:D
1137  \__kernel_primitive:NN \jis            \tex_jis:D
1138  \__kernel_primitive:NN \kanjiskip     \tex_kanjiskip:D
1139  \__kernel_primitive:NN \kansuji       \tex_kansuji:D
1140  \__kernel_primitive:NN \kansujichar   \tex_kansujichar:D
1141  \__kernel_primitive:NN \kcatcode      \tex_kcatcode:D
1142  \__kernel_primitive:NN \kuten          \tex_kuten:D
1143  \__kernel_primitive:NN \lastnodechar  \tex_lastnodechar:D
1144  \__kernel_primitive:NN \lastnodesubtype \tex_lastnodesubtype:D
1145  \__kernel_primitive:NN \noautospacing  \tex_noautospacing:D
1146  \__kernel_primitive:NN \noautoxspacing \tex_noautoxspacing:D
1147  \__kernel_primitive:NN \pagefiststretch \tex_pagefiststretch:D
1148  \__kernel_primitive:NN \postbreakpenalty \tex_postbreakpenalty:D
1149  \__kernel_primitive:NN \prebreakpenalty  \tex_prebreakpenalty:D
1150  \__kernel_primitive:NN \ptexminorversion \tex_ptexminorversion:D
1151  \__kernel_primitive:NN \ptexrevision    \tex_ptexrevision:D
1152  \__kernel_primitive:NN \ptexversion     \tex_ptexversion:D
1153  \__kernel_primitive:NN \readpapersizespecial \tex_readpapersizespecial:D
1154  \__kernel_primitive:NN \scriptbaselineshiftfactor
1155    \tex_scriptbaselineshiftfactor:D
1156  \__kernel_primitive:NN \scriptscriptbaselineshiftfactor
1157    \tex_scriptscriptbaselineshiftfactor:D
1158  \__kernel_primitive:NN \showmode        \tex_showmode:D
1159  \__kernel_primitive:NN \sjis           \tex_sjis:D
1160  \__kernel_primitive:NN \tate            \tex_tate:D
1161  \__kernel_primitive:NN \tbaselineshift  \tex_tbaselineshift:D
1162  \__kernel_primitive:NN \textbaselineshiftfactor
1163    \tex_textbaselineshiftfactor:D
1164  \__kernel_primitive:NN \tfont           \tex_tfont:D
1165  \__kernel_primitive:NN \xkanjiskip     \tex_xkanjiskip:D
1166  \__kernel_primitive:NN \xspcode         \tex_xspcode:D
1167  \__kernel_primitive:NN \ybaselineshift  \tex_ybaselineshift:D
1168  \__kernel_primitive:NN \yoko            \tex_yoko:D
1169  \__kernel_primitive:NN \vfi             \tex_vfi:D

Primitives from upTeX.
1170  \__kernel_primitive:NN \currentcjktoken \tex_currentcjktoken:D
1171  \__kernel_primitive:NN \disablecjktoken \tex_disablecjktoken:D
1172  \__kernel_primitive:NN \enablecjktoken  \tex_enablecjktoken:D
1173  \__kernel_primitive:NN \forcecjktoken   \tex_forcecjktoken:D

```

```

1174  \__kernel_primitive:NN \kchar          \tex_kchar:D
1175  \__kernel_primitive:NN \kchardef       \tex_kchardef:D
1176  \__kernel_primitive:NN \kuten         \tex_kuten:D
1177  \__kernel_primitive:NN \ucs            \tex_ucs:D
1178  \__kernel_primitive:NN \uptexrevision \tex_uptexrevision:D
1179  \__kernel_primitive:NN \uptexversion   \tex_uptexversion:D

```

Omega primitives provided by pTeX (listed separately mainly to allow understanding of their source).

```

1180  \__kernel_primitive:NN \odelcode        \tex_odelcode:D
1181  \__kernel_primitive:NN \odelimiter      \tex_odelimiter:D
1182  \__kernel_primitive:NN \omathaccent     \tex_omathaccent:D
1183  \__kernel_primitive:NN \omathchar       \tex_omathchar:D
1184  \__kernel_primitive:NN \omathchardef    \tex_omathchardef:D
1185  \__kernel_primitive:NN \omathcode       \tex_omathcode:D
1186  \__kernel_primitive:NN \oradical        \tex_oradical:D

```

End of the “just the names” part of the source.

```

1187 </names | package>
1188 </names | tex>
1189 <*package>
1190 <*tex>

```

The job is done: close the group (using the primitive renamed!).

```
1191 \tex_endgroup:D
```

LATEX 2 ε moves a few primitives, so these are sorted out. In newer versions of LATEX 2 ε , expl3 is loaded rather early, so only some primitives are already renamed, so we need two tests here. At the beginning of the LATEX 2 ε format, the primitives \end and \input are renamed, and only later on the other ones.

```

1192 \tex_ifdefined:D \@@end
1193  \tex_let:D \tex_end:D                  \@@end
1194  \tex_let:D \tex_input:D                \@@input
1195 \tex_fi:D

```

If \@@@hyph is defined, we are loading expl3 in a pre-2020/10/01 release of LATEX 2 ε , so a few other primitives have to be tested as well.

```

1196 \tex_ifdefined:D \@@hyph
1197  \tex_let:D \tex_everydisplay:D        \frozen@everydisplay
1198  \tex_let:D \tex_everymath:D          \frozen@everymath
1199  \tex_let:D \tex_hyphen:D            \@@hyph
1200  \tex_let:D \tex_italiccorrection:D \@@italiccorr
1201  \tex_let:D \tex_underline:D         \@@underline

```

The \shipout primitive is particularly tricky as a number of packages want to hook in here. First, we see if a sufficiently-new kernel has saved a copy: if it has, just use that. Otherwise, we need to check each of the possible packages/classes that might move it: here, we are looking for those which do *not* delay action to the \AtBeginDocument hook. (We cannot use \primitive as that doesn’t allow us to make a direct copy of the primitive *itself*.) As we know that LATEX 2 ε is in use, we use it’s \@tfor loop here.

```

1202  \tex_ifdefined:D \@@shipout
1203    \tex_let:D \tex_shipout:D \@@shipout
1204  \tex_fi:D
1205  \tex_begingroup:D
1206    \tex_eodef:D \l_tmpa_t1 { \tex_string:D \shipout }

```

```

1207   \tex_edef:D \l_tmpb_t1 { \tex_meaning:D \shipout }
1208   \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1209   \tex_else:D
1210     \tex_expandafter:D \otfor \tex_expandafter:D \otempa \tex_string:D :=
1211       \CROP@shipout
1212       \dup@shipout
1213       \GPTorg@shipout
1214       \LL@shipout
1215       \mem@oldshipout
1216       \open@shipout
1217       \pgfpages@originalshipout
1218       \pr@shipout
1219       \Shipout
1220       \verso@orig@shipout
1221       \do
1222         {
1223           \tex_edef:D \l_tmpb_t1
1224             { \tex_expandafter:D \tex_meaning:D \otempa }
1225           \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1226             \tex_global:D \tex_expandafter:D \tex_let:D
1227               \tex_expandafter:D \tex_shipout:D \otempa
1228             \tex_fi:D
1229           }
1230         \tex_fi:D
1231       \tex_endgroup:D

```

Some tidying up is needed for `\(pdf)tracingfonts`. Newer `LuaTeX` has this simply as `\tracingfonts`, but that is overwritten by the `LATEX 2 ε` kernel. So any spurious definition has to be removed, then the real version saved either from the `pdftEX` name or from `LuaTeX`. In the latter case, we leave `\@@tracingfonts` available: this might be useful and almost all `LATEX 2 ε` users will have `expl3` loaded by `fontspec`. (We follow the usual kernel convention that `@@` is used for saved primitives.)

```

1232   \tex_let:D \tex_tracingfonts:D \tex_undefined:D
1233   \tex_ifdefined:D \pdftracingfonts
1234     \tex_let:D \tex_tracingfonts:D \pdftracingfonts
1235   \tex_else:D
1236     \tex_ifdefined:D \tex_directlua:D
1237       \tex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
1238     \tex_let:D \tex_tracingfonts:D \@@tracingfonts
1239     \tex_fi:D
1240   \tex_fi:D
1241 \tex_fi:D

```

That is also true for the `LuaTeX` primitives under `LATEX 2 ε` (depending on the format-building date). There are a few primitives that get the right names anyway so are missing here!

```

1242 \tex_ifdefined:D \luatexsuppressfontnotfounderror
1243   \tex_let:D \tex_alignmark:D          \luatexalignmark
1244   \tex_let:D \tex_aligntab:D          \luatexaligntab
1245   \tex_let:D \tex_attribute:D        \luatexattribute
1246   \tex_let:D \tex_attributedef:D      \luatexattributedef
1247   \tex_let:D \tex_catcodetable:D    \luatexcatcodetable
1248   \tex_let:D \tex_clearmarks:D      \luatexclearmarks
1249   \tex_let:D \tex_crampeddisplaystyle:D \luatexcrampeddisplaystyle

```

```

1250 \tex_let:D \tex_crampedscriptscriptstyle:D           \luatexcrampedscriptscriptstyle
1251   \luatexcrampedscriptscriptstyle
1252 \tex_let:D \tex_crampedscriptstyle:D    \luatexcrampedscriptstyle
1253 \tex_let:D \tex_crampedtextstyle:D    \luatexcrampedtextstyle
1254 \tex_let:D \tex_fontid:D          \luatexfontid
1255 \tex_let:D \tex_formatname:D       \luatexformatname
1256 \tex_let:D \tex_gleaders:D        \luatexleaders
1257 \tex_let:D \tex_initcatcodetable:D \luatexinitcatcodetable
1258 \tex_let:D \tex_latelua:D         \luatexlatelua
1259 \tex_let:D \tex_luaescapestring:D \luatexluaescapestring
1260 \tex_let:D \tex_luafunction:D     \luatexluafunction
1261 \tex_let:D \tex_mathstyle:D       \luatexmathstyle
1262 \tex_let:D \tex_nokerns:D         \luatexnokerns
1263 \tex_let:D \tex_noligs:D         \luatexnoligs
1264 \tex_let:D \tex_outputbox:D       \luatexoutputbox
1265 \tex_let:D \tex_pageleftoffset:D \luatexpageleftoffset
1266 \tex_let:D \tex_pagetopoffset:D \luatexpagetopoffset
1267 \tex_let:D \tex_postexhyphenchar:D \luatexpostexhyphenchar
1268 \tex_let:D \tex_posthyphenchar:D \luatexposthyphenchar
1269 \tex_let:D \tex_preehyphenchar:D \luatexpreehyphenchar
1270 \tex_let:D \tex_prehyphenchar:D \luatexprehyphenchar
1271 \tex_let:D \tex_savecatcodetable:D \luatexsavecatcodetable
1272 \tex_let:D \tex_scantexttokens:D \luatexscantexttokens
1273 \tex_let:D \tex_suppressifcsnameerror:D
1274   \luatexsuppressifcsnameerror
1275 \tex_let:D \tex_suppresslongerror:D \luatexsuppresslongerror
1276 \tex_let:D \tex_suppressmathparerror:D
1277   \luatexsuppressmathparerror
1278 \tex_let:D \tex_suppressoutererror:D \luatexsuppressoutererror
1279 \tex_let:D \tex_Uchar:D          \luatexUchar
1280 \tex_let:D \tex_suppressfontnotfounderror:D
1281   \luatexsuppressfontnotfounderror

```

Which also covers those slightly odd ones.

```

1282 \tex_let:D \tex_bodydir:D          \luatexbodydir
1283 \tex_let:D \tex_boxdir:D          \luatexboxdir
1284 \tex_let:D \tex_leftghost:D        \luatexleftghost
1285 \tex_let:D \tex_localbrokenpenalty:D \luatexlocalbrokenpenalty
1286 \tex_let:D \tex_localinterlinepenalty:D
1287   \luatexlocalinterlinepenalty
1288 \tex_let:D \tex_localeleftbox:D    \luatexlocaleleftbox
1289 \tex_let:D \tex_localrightbox:D    \luatexlocalrightbox
1290 \tex_let:D \tex_mathdir:D         \luatexmathdir
1291 \tex_let:D \tex_pagebottomoffset:D \luatexpagebottomoffset
1292 \tex_let:D \tex_pagedir:D         \luatexpagedir
1293 \tex_let:D \tex_pageheight:D       \luatexpageheight
1294 \tex_let:D \tex_pagerightoffset:D \luatexpagerightoffset
1295 \tex_let:D \tex_pagewidth:D        \luatexpagewidth
1296 \tex_let:D \tex_pardir:D          \luatexpardir
1297 \tex_let:D \tex_rightghost:D      \luatexrightghost
1298 \tex_let:D \tex_textdir:D         \luatextextdir
1299 \tex_fi:D

```

Only pdf \TeX and Lua \TeX define \pdfmapfile and \pdfmapline: Tidy up the fact that some format-building processes leave a couple of questionable decisions about that!

```

1300 \tex_ifnum:D 0
1301   \tex_ifdefined:D \tex_pdftexversion:D 1 \tex_hi:D
1302   \tex_ifdefined:D \tex_luatexversion:D 1 \tex_hi:D
1303   = 0 %
1304   \tex_let:D \tex_mapfile:D \tex_undefined:D
1305   \tex_let:D \tex_mapline:D \tex_undefined:D
1306 \tex_hi:D

```

A few packages do unfortunate things to date-related primitives.

```

1307 \tex_begingroup:D
1308   \tex_edef:D \l_tmpa_t1 { \tex_meaning:D \tex_time:D }
1309   \tex_edef:D \l_tmpb_t1 { \tex_string:D \time }
1310   \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1311   \tex_else:D
1312     \tex_global:D \tex_let:D \tex_time:D \tex_undefined:D
1313     \tex_hi:D
1314     \tex_edef:D \l_tmpa_t1 { \tex_meaning:D \tex_day:D }
1315     \tex_edef:D \l_tmpb_t1 { \tex_string:D \day }
1316     \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1317     \tex_else:D
1318       \tex_global:D \tex_let:D \tex_day:D \tex_undefined:D
1319       \tex_hi:D
1320       \tex_edef:D \l_tmpa_t1 { \tex_meaning:D \tex_month:D }
1321       \tex_edef:D \l_tmpb_t1 { \tex_string:D \month }
1322       \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1323       \tex_else:D
1324         \tex_global:D \tex_let:D \tex_month:D \tex_undefined:D
1325         \tex_hi:D
1326         \tex_edef:D \l_tmpa_t1 { \tex_meaning:D \tex_year:D }
1327         \tex_edef:D \l_tmpb_t1 { \tex_string:D \year }
1328         \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1329         \tex_else:D
1330           \tex_global:D \tex_let:D \tex_year:D \tex_undefined:D
1331           \tex_hi:D
1332 \tex_endgroup:D

```

Up to v0.80, LuaTE_X defines the pdfTE_X version data: rather confusing. Removing them means that \tex_pdftexversion:D is a marker for pdfTE_X alone: useful in engine-dependent code later.

```

1333 \tex_ifdefined:D \tex_luatexversion:D
1334   \tex_let:D \tex_pdftexbanner:D \tex_undefined:D
1335   \tex_let:D \tex_pdftexrevision:D \tex_undefined:D
1336   \tex_let:D \tex_pdftexversion:D \tex_undefined:D
1337 \tex_hi:D

```

cslatex moves a couple of primitives which we recover here; as there is no other marker, we can only work by looking for the names.

```

1338 \tex_ifdefined:D \orieveryjob
1339   \tex_let:D \tex_everyjob:D \orieveryjob
1340 \tex_hi:D
1341 \tex_ifdefined:D \oripdfoutput
1342   \tex_let:D \tex_pdfoutput:D \oripdfoutput
1343 \tex_hi:D

```

For ConTeXt, two tests are needed. Both Mark II and Mark IV move several primitives: these are all covered by the first test, again using \end as a marker. For Mark IV,

a few more primitives are moved: they are implemented using some Lua code in the current ConTeXt.

```

1344 \tex_ifdefined:D \normalend
1345   \tex_let:D \tex_end:D      \normalend
1346   \tex_let:D \tex_everyjob:D \normaleveryjob
1347   \tex_let:D \tex_input:D   \normalinput
1348   \tex_let:D \tex_language:D \normallanguage
1349   \tex_let:D \tex_mathop:D  \normalmathop
1350   \tex_let:D \tex_month:D   \normalmonth
1351   \tex_let:D \tex_outer:D   \normalouter
1352   \tex_let:D \tex_over:D   \normalover
1353   \tex_let:D \tex_vcenter:D \normalvcenter
1354   \tex_let:D \tex_unexpanded:D \normalunexpanded
1355   \tex_let:D \tex_expanded:D \normalexpanded
1356 \tex_fi:D
1357 \tex_ifdefined:D \normalitaliccorrection
1358   \tex_let:D \tex_hoffset:D    \normalhoffset
1359   \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
1360   \tex_let:D \tex_voffset:D   \normalvoffset
1361   \tex_let:D \tex_showtokens:D \normalshowtokens
1362   \tex_let:D \tex_bodydir:D   \spac_directions_normal_body_dir
1363   \tex_let:D \tex_pagedir:D  \spac_directions_normal_page_dir
1364 \tex_fi:D
1365 \tex_ifdefined:D \normalleft
1366   \tex_let:D \tex_left:D     \normalleft
1367   \tex_let:D \tex_middle:D   \normalmiddle
1368   \tex_let:D \tex_right:D   \normalright
1369 \tex_fi:D
1370 </tex>

```

In LuaTeX, we additionally emulate some primitives using Lua code.

```
1371 /*lua*/

```

\tex_strcmp:D Compare two strings, expanding to 0 if they are equal, -1 if the first one is smaller and 1 if the second one is smaller. Here “smaller” refers to codepoint order which does not correspond to the user expected order for most non-ASCII strings.

```

1372 local minus_tok = token.new(string.byte'-', 12)
1373 local zero_tok = token.new(string.byte'0', 12)
1374 local one_tok = token.new(string.byte'1', 12)
1375 luacmd('tex_strcmp:D', function()
1376   local first = scan_string()
1377   local second = scan_string()
1378   if first < second then
1379     put_next(minus_tok, one_tok)
1380   else
1381     put_next(first == second and zero_tok or one_tok)
1382   end
1383 end, 'global')

```

(End definition for \tex_strcmp:D. This function is documented on page ??.)

\tex_Ucharcat:D Creating arbitrary chars using `tex.cprint`. The alternative approach using `token.put_next(token.create)` would be about 10% slower.

```

1384 local cprint = tex.cprint
1385 luacmd('tex_Ucharcat:D', function()
1386   local charcode = scan_int()
1387   local catcode = scan_int()
1388   cprint(catcode, utf8_char(charcode))
1389 end, 'global')

```

(End definition for \tex_Ucharcat:D. This function is documented on page ??.)

\tex_filesize:D Wrap the function from ltxutils.

```

1390 luacmd('tex_filesize:D', function()
1391   local size = filesize(scan_string())
1392   if size then write(size) end
1393 end, 'global')

```

(End definition for \tex_filesize:D. This function is documented on page ??.)

\tex_mdfivesum:D There are two cases: Either hash a file or a string. Both are already implemented in l3luatex or built-in.

```

1394 luacmd('tex_mdfivesum:D', function()
1395   local hash
1396   if scan_keyword"file" then
1397     hash = filemd5sum(scan_string())
1398   else
1399     hash = md5_HEX(scan_string())
1400   end
1401   if hash then write(hash) end
1402 end, 'global')

```

(End definition for \tex_mdfivesum:D. This function is documented on page ??.)

\tex_filemoddate:D A primitive for getting the modification date of a file.

```

1403 luacmd('tex_filemoddate:D', function()
1404   local date = filemoddate(scan_string())
1405   if date then write(date) end
1406 end, 'global')

```

(End definition for \tex_filemoddate:D. This function is documented on page ??.)

\tex_filedump:D An emulated primitive for getting a hexdump from a (partial) file. The length has a default of 0. This is consistent with pdftEX, but it effectively makes the primitive useless without an explicit length. Therefore we allow the keyword whole to be used instead of a length, indicating that the whole remaining file should be read.

```

1407 luacmd('tex_filedump:D', function()
1408   local offset = scan_keyword'offset' and scan_int() or nil
1409   local length = scan_keyword'length' and scan_int()
1410     or not scan_keyword'whole' and 0 or nil
1411   local data = filedump(scan_string(), offset, length)
1412   if data then write(data) end
1413 end, 'global')

```

(End definition for \tex_filedump:D. This function is documented on page ??.)

1414 </lua>

1415 </package>

3 Internal kernel functions

```
\__kernel_chk_cs_exist:N \__kernel_chk_cs_exist:c
```

This function is only created if debugging is enabled. It checks that $\langle cs \rangle$ exists according to the criteria for $\backslash cs_if_exist:N$, and if not raises a kernel-level error.

```
\__kernel_chk_defined:NT
```

If $\langle variable \rangle$ is not defined (according to $\backslash cs_if_exist:NT$), this triggers an error, otherwise the $\langle true\ code \rangle$ is run.

```
\__kernel_chk_expr:nNnN
```

This function is only created if debugging is enabled. By default it is equivalent to $\backslash use_i:nnnn$. When expression checking is enabled, it leaves in the input stream the result of $\backslash tex_the:D \langle eval \rangle \langle expr \rangle \backslash tex_relax:D$ after checking that no token was left over. If any token was not taken as part of the expression, there is an error message displaying the result of the evaluation as well as the $\langle caller \rangle$. For instance $\langle eval \rangle$ can be $\backslash _int_eval:w$ and $\langle caller \rangle$ can be $\backslash int_eval:n$ or $\backslash int_set:Nn$. The argument $\langle convert \rangle$ is empty except for mu expressions where it is $\backslash tex_mutoglu:D$, used for internal purposes.

```
\__kernel_cs_parm_from_arg_count:nnF \__kernel_cs_parm_from_arg_count:nnF {\langle follow-on \rangle} {\langle args \rangle} {\langle false code \rangle}
```

Evaluates the number of $\langle args \rangle$ and leaves the $\langle follow-on \rangle$ code followed by a brace group containing the required number of primitive parameter markers (#1, etc.). If the number of $\langle args \rangle$ is outside the range [0, 9], the $\langle false code \rangle$ is inserted instead of the $\langle follow-on \rangle$.

```
\__kernel_dependency_version_check:Nn \__kernel_dependency_version_check:Nn {\langle date \rangle} {\langle file \rangle}  
\__kernel_dependency_version_check:nn \__kernel_dependency_version_check:nn {\langle date \rangle} {\langle file \rangle}
```

Checks if the loaded version of the expl3 kernel is at least $\langle date \rangle$, required by $\langle file \rangle$. If the kernel date is older than $\langle date \rangle$, the loading of $\langle file \rangle$ is aborted and an error is raised.

```
\__kernel_deprecation_code:nn \__kernel_deprecation_code:nn {\langle error code \rangle} {\langle working code \rangle}
```

Stores both an $\langle error \rangle$ and $\langle working \rangle$ definition for given material such that they can be exchanged by $\backslash debug_on:$ and $\backslash debug_off::$.

```
\__kernel_exp_not:w *
```

```
\__kernel_exp_not:w {\langle expandable tokens \rangle} {\langle content \rangle}
```

Carries out expansion on the $\langle expandable\ tokens \rangle$ before preventing further expansion of the $\langle content \rangle$ as for $\backslash exp_not:n$. Typically, the $\langle expandable\ tokens \rangle$ will alter the nature of the $\langle content \rangle$, i.e. allow it to be generated in some way.

$\backslash l_{__kernel_expl_bool}$ A boolean which records the current code syntax status: `true` if currently inside a code environment. This variable should only be set by $\backslash ExplSyntaxOn/\ExplSyntaxOff$.

(End definition for $\backslash l_{__kernel_expl_bool}$.)

$\backslash c_{__kernel_expl_date_tl}$ A token list containing the release date of the l3kernel preloaded in L^AT_EX 2_< used to check if dependencies match.

(End definition for `\c_kernel_expl_date_tl`.)

`_kernel_file_missing:n` `_kernel_file_missing:n {<name>}`

Expands the `<name>` as per `_kernel_file_name_sanitize:nN` then produces an error message indicating that this file was not found.

`_kernel_file_name_sanitize:nN` `_kernel_file_name_sanitize:nN {<name>} <str var>`

For converting a `<name>` to a string where active characters are treated as strings.

`_kernel_file_input_push:n` `_kernel_file_input_push:n {<name>}`
`_kernel_file_input_pop:` `_kernel_file_input_pop:`

Used to push and pop data from the internal file stack: needed only in package mode, where interfacing with the L^AT_EX 2_E kernel is necessary.

`_kernel_int_add:nnn *` `_kernel_int_add:nnn {<integer1>} {<integer2>} {<integer3>}`

Expands to the result of adding the three `<integers>` (which must be suitable input for `\int_eval:w`), avoiding intermediate overflow. Overflow occurs only if the overall result is outside $[-2^{31}+1, 2^{31}-1]$. The `<integers>` may be of the form `\int_eval:w ... \scan_stop:` but may be evaluated more than once.

`_kernel_intarray_gset:Nnn` `_kernel_intarray_gset:Nnn {intarray var} {<index>} {<value>}`

New: 2018-03-31

Faster version of `\intarray_gset:Nnn`. Stores the `<value>` into the `<integer array variable>` at the `<position>`. The `<index>` and `<value>` must be suitable for a direct assignment to a T_EX count register, for instance expanding to an integer denotation or obtained through the primitive `\numexpr` (which may be un-terminated). No bound checking is performed: the caller is responsible for ensuring that the `<position>` is between 1 and the `\intarray_count:N`, and the `<value>`'s absolute value is at most $2^{30} - 1$. Assignments are always global.

`_kernel_intarray_item:Nn *` `_kernel_intarray_item:Nn {intarray var} {<index>}`

New: 2018-03-31

Faster version of `\intarray_item:Nn`. Expands to the integer entry stored at the `<index>` in the `<integer array variable>`. The `<index>` must be suitable for a direct assignment to a T_EX count register and must be between 1 and the `\intarray_count:N`, lest a low-level T_EX error occur.

`_kernel_ioreopen:Nn` `_kernel_ioreopen:Nn {stream} {<file name>}`

This function has identical syntax to the public version. However, it does not take precautions against active characters in the `<file name>`, and it does not attempt to add a `<path>` to the `<file name>`: it is therefore intended to be used by higher-level functions which have already fully expanded the `<file name>` and which need to perform multiple open or close operations. See for example the implementation of `\file_get_full_name:nN`,

`_kernel_iow_with:Nnn`

`_kernel_iow_with:Nnn <integer> {<value>} {<code>}`

If the *<integer>* is equal to the *<value>* then this function simply runs the *<code>*. Otherwise it saves the current value of the *<integer>*, sets it to the *<value>*, runs the *<code>*, and restores the *<integer>* to its former value. This is used to ensure that the `\newlinechar` is 10 when writing to a stream, which lets `\iow_newline:` work, and that `\errorcontextlines` is -1 when displaying a message.

`_kernel_msg_new:nnnn`
`_kernel_msg_new:nnn`

`_kernel_msg_new:nnnn {<module>} {<message>} {<text>} {<more text>}`

Creates a kernel *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (#1 to #4) can be used: these will be supplied and expanded at the time the message is used. An error is raised if the *<message>* already exists.

`_kernel_msg_set:nnnn`
`_kernel_msg_set:nnn`

`_kernel_msg_set:nnnn {<module>} {<message>} {<text>} {<more text>}`

Sets up the text for a kernel *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (#1 to #4) can be used: these will be supplied and expanded at the time the message is used.

`_kernel_msg_fatal:nnnnnn`
`_kernel_msg_fatal:nnxxxx`
`_kernel_msg_fatal:nnnnn`
`_kernel_msg_fatal:nnxxx`
`_kernel_msg_fatal:nnnn`
`_kernel_msg_fatal:nnxx`
`_kernel_msg_fatal:nnn`
`_kernel_msg_fatal:nnx`
`_kernel_msg_fatal:nn`

`_kernel_msg_fatal:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}`

Issues kernel *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. After issuing a fatal error the TeX run halts. Cannot be redirected.

`_kernel_msg_critical:nnnnnn`
`_kernel_msg_critical:nnxxxx`
`_kernel_msg_critical:nnnnn`
`_kernel_msg_critical:nnxxx`
`_kernel_msg_critical:nnnn`
`_kernel_msg_critical:nnxx`
`_kernel_msg_critical:nnn`
`_kernel_msg_critical:nnx`
`_kernel_msg_critical:nn`

`_kernel_msg_critical:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}`

Issues kernel *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. After issuing a critical error, TeX stops reading the current input file. Cannot be redirected.

<code>__kernel_msg_error:nnnnnn __kernel_msg_error:nnxxxx __kernel_msg_error:nnnnn __kernel_msg_error:nnxx __kernel_msg_error:nnnn __kernel_msg_error:nnx __kernel_msg_error:nn __kernel_msg_error:nnx __kernel_msg_error:nn</code>	<code>__kernel_msg_error:nnnnnnn {\<module>} {\<message>} {\<arg one>} {\<arg two>} {\<arg three>} {\<arg four>}</code>	Issues kernel <i><module></i> error <i><message></i> , passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The error stops processing and issues the text at the terminal. After user input, the run continues. Cannot be redirected.
<code>__kernel_msg_warning:nnnnnn __kernel_msg_warning:nnxxxx __kernel_msg_warning:nnnnn __kernel_msg_warning:nnxx __kernel_msg_warning:nnnn __kernel_msg_warning:nnx __kernel_msg_warning:nn __kernel_msg_warning:nnx __kernel_msg_warning:nn</code>	<code>__kernel_msg_warning:nnnnnnn {\<module>} {\<message>} {\<arg one>} {\<arg two>} {\<arg three>} {\<arg four>}</code>	Issues kernel <i><module></i> warning <i><message></i> , passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The warning text is added to the log file, but the TeX run is not interrupted.
<code>__kernel_msg_info:nnnnnn __kernel_msg_info:nnxxxx __kernel_msg_info:nnnnn __kernel_msg_info:nnxx __kernel_msg_info:nnnn __kernel_msg_info:nnx __kernel_msg_info:nn __kernel_msg_info:nnx __kernel_msg_info:nn</code>	<code>__kernel_msg_info:nnnnnnn {\<module>} {\<message>} {\<arg one>} {\<arg two>} {\<arg three>} {\<arg four>}</code>	Issues kernel <i><module></i> information <i><message></i> , passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The information text is added to the log file.
<code>__kernel_msg_expandable_error:nnnnnn * __kernel_msg_expandable_error:nnffff * __kernel_msg_expandable_error:nnnnn * __kernel_msg_expandable_error:nnfff * __kernel_msg_expandable_error:nnnn * __kernel_msg_expandable_error:nnff * __kernel_msg_expandable_error:nnn * __kernel_msg_expandable_error:nnf * __kernel_msg_expandable_error:nn *</code>	<code>__kernel_msg_expandable_error:nnnnnnn {\<module>} {\<message>} {\<arg one>} {\<arg two>} {\<arg three>} {\<arg four>}</code>	Issues an error, passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The resulting string must be much shorter than a line, otherwise it is cropped.
<code>\g_kernel_prg_map_int</code>	This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions <code>\<type>_map_1:w</code> , <code>\<type>_map_2:w</code> , etc., labelled by <code>\g_kernel_prg_map_int</code> hold functions to be mapped over various list datatypes in inline and variable mappings.	

(End definition for `\g_kernel_prg_map_int`.)

`_kernel_quark_new_test:N _kernel_quark_new_test:N \<name>:<arg spec>`

Defines a quark-test function `\<name>:<arg spec>` which tests if its argument is `\q__\<namespace>\recursion_tail`, then acts accordingly, as described below for each possible `<arg spec>`.

The `<namespace>` is determined as the first (nonempty) `_`-delimited word in `<name>` and is used internally in the definition of auxiliaries. The function `_kernel_quark_new_test:N` does *not* define the `\q__\<namespace>\recursion_tail` and `\q__\<namespace>\recursion_stop` quarks. They should be manually defined with `\quark_new:N`.

There are 6 different types of quark-test functions. Which one is defined depends on the `<arg spec>`, which *must* be one of the options listed now. Four of them are modeled after `\quark_if_recursion_tail:(N|n)` and `\quark_if_recursion_tail_do:(N|n)n`.

`n` defines `\<name>:n` such that it checks if #1 contains only `\q__\<namespace>\recursion_tail`, and if so consumes all tokens up to `\q__\<namespace>\recursion_stop` (*c.f.* `\quark_if_recursion_tail_stop:n`).

`nn` defines `\<name>:nn` such that it checks if #1 contains only `\q__\<namespace>\recursion_tail`, and if so consumes all tokens up to `\q__\<namespace>\recursion_stop`, then executes the code #2 after that (*c.f.* `\quark_if_recursion_tail_stop_do:nn`).

`N` defines `\<name>:N` such that it checks if #1 is `\q__\<namespace>\recursion_tail`, and if so consumes all tokens up to `\q__\<namespace>\recursion_stop` (*c.f.* `\quark_if_recursion_tail_stop:N`).

`Nn` defines `\<name>:Nn` such that it checks if #1 is `\q__\<namespace>\recursion_tail`, and if so consumes all tokens up to `\q__\<namespace>\recursion_stop`, then executes the code #2 after that (*c.f.* `\quark_if_recursion_tail_stop_do:Nn`).

The last two are modeled after `\quark_if_recursion_tail_break:(n|N)N`, and in those cases the quark `\q__\<namespace>\recursion_stop` is not used (and thus needs not be defined).

`nN` defines `\<name>:nN` such that it checks if #1 contains only `\q__\<namespace>\recursion_tail`, and if so uses the `\<type>\map_break:` function #2.

`NN` defines `\<name>:NN` such that it checks if #1 is `\q__\<namespace>\recursion_tail`, and if so uses the `\<type>\map_break:` function #2.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

_kernel_quark_new_conditional:Nn _kernel_quark_new_conditional:Nn
_<namespace>_quark_if_<name>:<arg spec> {<conditions>}

Defines a collection of quark conditionals that test if their argument is the quark `\q_<namespace>_<name>` and perform suitable actions. The `<conditions>` are a comma-separated list of one or more of p, T, F, and TF, and one conditional is defined for each `<condition>` in the list, as described for `\prg_new_conditional:Npnn`. The conditionals are defined using `\prg_new_conditional:Npnn`, so that their name is obtained by adding p, T, F, or TF to the base name `__<namespace>_quark_if_<name>:<arg spec>`.

The first argument of `_kernel_quark_new_conditional:Nn` must contain `_quark_if_` and `:`, as these markers are used to determine the `name` of the quark `\q_--<namespace>_<name>` to be tested. This quark should be manually defined with `\quark_new:N`, as `_kernel_quark_new_conditional:Nn` does *not* define it.

The function `_kernel_quark_new_conditional:Nn` can define 2 different types of quark conditionals. Which one is defined depends on the *(arg spec)*, which *must* be one of the following options, modeled after `\quark_if_nil:(N|n)(TF)`.

n defines __<namespace>_quark_if_<name>:n(TF) such that it checks if #1 contains only \q_<namespace>_<name>, and executes the proper conditional branch.

`N` defines `_<namespace>_quark_if_<name>:N(TF)` such that it checks if `#1` is `\q_<namespace>_<name>`, and executes the proper conditional branch.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

`\c_kernel_randint_max_int` Maximal allowed argument to `__kernel_randint:n`. Equal to $2^{17} - 1$.

(End definition for `\c_kernel_randint_max_int`.)

__kernel_randint:n __kernel_randint:n {<max>}

Used in an integer expression this gives a pseudo-random number between 1 and $\langle max \rangle$ included. One must have $\langle max \rangle \leq 2^{17} - 1$. The $\langle max \rangle$ must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent).

`_kernel_randint:nn _kernel_randint:nn {<min>} {<max>}`

Used in an integer expression this gives a pseudo-random number between $\langle min \rangle$ and $\langle max \rangle$ included. The $\langle min \rangle$ and $\langle max \rangle$ must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent). For small ranges $R = \langle max \rangle - \langle min \rangle + 1 \leq 2^{17} - 1$, $\langle min \rangle - 1 + \text{__kernel_randint:n}\{R\}$ is faster.

__kernel_register_show:N __kernel_register_show:N <register>

_kernel_register_show:c

Used to show the contents of a **TeX** register at the terminal, formatted such that internal parts of the mechanism are not visible.

```
\ kernel register log:N <register>
```

Used to write the contents of a **TEX** register to the log file in a form similar to `_-_kernel_register_log:N` (`register`)

`__kernel_str_to_other:n *` `__kernel_str_to_other:n {<token list>}`

Converts the $\langle token\ list\rangle$ to a $\langle other\ string\rangle$, where spaces have category code “other”. This function can be f-expanded without fear of losing a leading space, since spaces do not have category code 10 in its result. It takes a time quadratic in the character count of the string.

`__kernel_str_to_other_fast:n ☆` `__kernel_str_to_other_fast:n {<token list>}`

Same behaviour `__kernel_str_to_other:n` but only restricted-expandable. It takes a time linear in the character count of the string.

`__kernel_tl_to_str:w *` `__kernel_tl_to_str:w {expandable tokens} {<tokens>}`

Carries out expansion on the $\langle expandable\ tokens\rangle$ before conversion of the $\langle tokens\rangle$ to a string as describe for `\tl_to_str:n`. Typically, the $\langle expandable\ tokens\rangle$ will alter the nature of the $\langle tokens\rangle$, i.e. allow it to be generated in some way. This function requires only a single expansion.

`__kernel_tl_set:Nx` `__kernel_tl_gset:Nx` `__kernel_tl_set:Nx {tl var} {<tokens>}`

Fully expands $\langle tokens\rangle$ and assigns the result to $\langle tl\ var\rangle$. $\langle tokens\rangle$ must be given in braces and there must be no token between $\langle tl\ var\rangle$ and $\langle tokens\rangle$.

4 Kernel backend functions

These functions are required to pass information to the backend. The nature of these means that they are defined only when the relevant backend is in use.

`__kernel_backend_literal:n` `__kernel_backend_literal:n {<content>}`

Adds the $\langle content\rangle$ literally to the current vertical list as a whatsit. The nature of the $\langle content\rangle$ will depend on the backend in use.

`__kernel_backend_literal_postscript:n` `__kernel_backend_literal_postscript:n {<PostScript>}`

Adds the $\langle PostScript\rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

`__kernel_backend_literal_pdf:n` `__kernel_backend_literal_pdf:n {<PDF instructions>}`

Adds the $\langle PDF\ instructions\rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

`__kernel_backend_literal_svg:n` `__kernel_backend_literal_svg:n {<SVG instructions>}`

Adds the $\langle SVG\ instructions\rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

```
\_\_kernel_backend_postscript:n  \_\_kernel_backend_postscript:n {\langle PostScript\rangle}
\_\_kernel_backend_postscript:x
```

Adds the $\langle PostScript \rangle$ to the current vertical list as a whatsit. The PostScript reference point is adjusted to match the current position. The PostScript is inserted inside a SDict begin/end pair.

```
\_\_kernel_backend_align_begin:  \_\_kernel_backend_align_begin:
\_\_kernel_backend_align_end:    {\langle PostScript literals\rangle}
\_\_kernel_backend_align_end:
```

Arranges to align the PostScript and DVI current positions and scales.

```
\_\_kernel_backend_scope_begin:  \_\_kernel_backend_scope_begin:
\_\_kernel_backend_scope_end:    {\langle content\rangle}
\_\_kernel_backend_scope_end:
```

Creates a scope for instructions at the backend level.

```
\_\_kernel_backend_matrix:n  \_\_kernel_backend_matrix:n {\langle matrix\rangle}
\_\_kernel_backend_matrix:x  Applies the {\langle matrix\rangle} to the current transformation matrix.
```

```
\g_\_\_kernel_backend_header_bool
```

Specifies whether to write headers for the backend.

```
\l_\_\_kernel_color_stack_int  The color stack used in pdfTEX and LuaTEX for the main color.
```

5 **I3**basics implementation

1416 $\langle *package \rangle$

5.1 Renaming some **T_EX** primitives (again)

Having given all the **T_EX** primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but we do a few now, just to get started.⁶

```
\if_true: Then some conditionals.
\if_false: 1417 \tex_let:D \if_true:          \tex_iftrue:D
\or:        1418 \tex_let:D \if_false:        \tex_iffalse:D
\else:      1419 \tex_let:D \or:            \tex_or:D
\fi:        1420 \tex_let:D \else:           \tex_else:D
\reverse_if:N 1421 \tex_let:D \fi:            \tex_fi:D
\if:w       1422 \tex_let:D \reverse_if:N   \tex_unless:D
\if_charcode:w 1423 \tex_let:D \if:w         \tex_if:D
\if_catcode:w 1424 \tex_let:D \if_charcode:w \tex_if:D
\if_meaning:w 1425 \tex_let:D \if_catcode:w \tex_ifcat:D
\if_bool:N   1426 \tex_let:D \if_meaning:w \tex_ifx:D
\ifodd:D    1427 \tex_let:D \if_bool:N   \tex_ifodd:D
```

⁶This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the \tex_...:D name in the cases where no good alternative exists.

(End definition for `\if_true:` and others. These functions are documented on page 23.)

`\if_mode_math:` T_EX lets us detect some if its modes.
`\if_mode_horizontal:` 1428 `\tex_let:D \if_mode_math: \tex_ifmmode:D`
`\if_mode_vertical:` 1429 `\tex_let:D \if_mode_horizontal: \tex_ifhmode:D`
`\if_mode_inner:` 1430 `\tex_let:D \if_mode_vertical: \tex_ifvmode:D`
1431 `\tex_let:D \if_mode_inner: \tex_ifinner:D`

(End definition for `\if_mode_math:` and others. These functions are documented on page 23.)

`\if_cs_exist:N` Building csnames and testing if control sequences exist.
`\if_cs_exist:w` 1432 `\tex_let:D \if_cs_exist:N \tex_ifdefined:D`
`\cs:w` 1433 `\tex_let:D \if_cs_exist:w \tex_ifcsname:D`
`\cs_end:` 1434 `\tex_let:D \cs:w \tex_csname:D`
1435 `\tex_let:D \cs_end: \tex_endcsname:D`

(End definition for `\if_cs_exist:N` and others. These functions are documented on page 23.)

`\exp_after:wN` The five `\exp_` functions are used in the l3expan module where they are described.
`\exp_not:N` 1436 `\tex_let:D \exp_after:wN \tex_expandafter:D`
`\exp_not:n` 1437 `\tex_let:D \exp_not:N \tex_noexpand:D`
1438 `\tex_let:D \exp_not:n \tex_unexpanded:D`
1439 `\tex_let:D \exp:w \tex_roman numeral:D`
1440 `\tex_chardef:D \exp_end: = 0 ~`

(End definition for `\exp_after:wN`, `\exp_not:N`, and `\exp_not:n`. These functions are documented on page 33.)

`\token_to_meaning:N` Examining a control sequence or token.
`\cs_meaning:N` 1441 `\tex_let:D \token_to_meaning:N \tex_meaning:D`
1442 `\tex_let:D \cs_meaning:N \tex_meaning:D`

(End definition for `\token_to_meaning:N` and `\cs_meaning:N`. These functions are documented on page 134.)

`\tl_to_str:n` Making strings.
`\token_to_str:N` 1443 `\tex_let:D \tl_to_str:n \tex_detokenize:D`
1444 `\tex_let:D \token_to_str:N \tex_string:D`
1445 `\tex_let:D __kernel_tl_to_str:w \tex_detokenize:D`

(End definition for `\tl_to_str:n`, `\token_to_str:N`, and `__kernel_tl_to_str:w`. These functions are documented on page 51.)

`\scan_stop:` The next three are basic functions for which there also exist versions that are safe inside alignments. These safe versions are defined in the l3prg module.
`\group_begin:`
`\group_end:` 1446 `\tex_let:D \scan_stop: \tex_relax:D`
1447 `\tex_let:D \group_begin: \tex_begingroup:D`
1448 `\tex_let:D \group_end: \tex_endgroup:D`

(End definition for `\scan_stop:`, `\group_begin:`, and `\group_end:`. These functions are documented on page 9.)

1449 `<@@=int>`

`\if_int_compare:w` For integers.
`__int_to_roman:w` 1450 `\tex_let:D \if_int_compare:w \tex_ifnum:D`
1451 `\tex_let:D __int_to_roman:w \tex_roman numeral:D`

(End definition for `\if_int_compare:w` and `_int_to_roman:w`. This function is documented on page 101.)

`\group_insert_after:N` Adding material after the end of a group.

```
1452 \tex_let:D \group_insert_after:N \tex_aftergroup:D
```

(End definition for `\group_insert_after:N`. This function is documented on page 9.)

`\exp_args:Nc` Discussed in l3expan, but needed much earlier.

`\exp_args:cc`

```
1453 \tex_long:D \tex_def:D \exp_args:Nc #1#2
1454   { \exp_after:wN #1 \cs:w #2 \cs_end: }
1455 \tex_long:D \tex_def:D \exp_args:cc #1#2
1456   { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }
```

(End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 29.)

`\token_to_meaning:c` A small number of variants defined by hand. Some of the necessary functions (`\use_i:nn`, `\use_ii:nn`, and `\exp_args:NNc`) are not defined at that point yet, but will be defined before those variants are used. The `\cs_meaning:c` command must check for an undefined control sequence to avoid defining it mistakenly.

```
1457 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
1458 \tex_long:D \tex_def:D \cs_meaning:c #
1459   {
1460     \if_cs_exist:w #1 \cs_end:
1461       \exp_after:wN \use_i:nn
1462     \else:
1463       \exp_after:wN \use_ii:nn
1464     \fi:
1465     { \exp_args:Nc \cs_meaning:N {#1} }
1466     { \tl_to_str:n {undefined} }
1467   }
1468 \tex_let:D \token_to_meaning:c = \cs_meaning:c
```

(End definition for `\token_to_meaning:N`. This function is documented on page 134.)

5.2 Defining some constants

`\c_zero_int`

We need the constant `\c_zero_int` which is used by some functions in the l3alloc module. The rest are defined in the l3int module – at least for the ones that can be defined with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the l3int module is required but it can't be used until the allocation has been set up properly!

```
1469 \tex_chardef:D \c_zero_int = 0 ~
```

(End definition for `\c_zero_int`. This variable is documented on page 100.)

`\c_max_register_int`

This is here as this particular integer is needed both in package mode and to bootstrap l3alloc, and is documented in l3int. LuaTeX and those which contain parts of the Omega extensions have more registers available than ε-TeX.

```
1470 \tex_ifdefined:D \tex_luatexversion:D
1471   \tex_chardef:D \c_max_register_int = 65 535 ~
1472 \tex_else:D
1473   \tex_ifdefined:D \tex_omathchardef:D
1474     \tex_omathchardef:D \c_max_register_int = 65535 ~
1475   \tex_else:D
```

```

1476     \tex_mathchardef:D \c_max_register_int = 32767 ~
1477     \tex_fi:D
1478 \tex_fi:D

```

(End definition for `\c_max_register_int`. This variable is documented on page 100.)

5.3 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

```

\cs_set_nopar:Npn
\cs_set_nopar:Npx
  \cs_set:Npn
  \cs_set:Npx
\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:Npx
  \cs_set_protected:Npn
  \cs_set_protected:Npx

```

```

1479 \tex_let:D \cs_set_nopar:Npn           \tex_def:D
1480 \tex_let:D \cs_set_nopar:Npx           \tex_edef:D
1481 \tex_protected:D \tex_long:D \tex_def:D \cs_set:Npn
1482 { \tex_long:D \tex_def:D }
1483 \tex_protected:D \tex_long:D \tex_def:D \cs_set:Npx
1484 { \tex_long:D \tex_edef:D }
1485 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npn
1486 { \tex_protected:D \tex_def:D }
1487 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npx
1488 { \tex_protected:D \tex_edef:D }
1489 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npn
1490 { \tex_protected:D \tex_long:D \tex_def:D }
1491 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npx
1492 { \tex_protected:D \tex_long:D \tex_edef:D }

```

(End definition for `\cs_set_nopar:Npn` and others. These functions are documented on page 11.)

```

\cs_gset_nopar:Npn
\cs_gset_nopar:Npx
  \cs_gset:Npn
  \cs_gset:Npx
\cs_gset_protected_nopar:Npn
\cs_gset_protected_nopar:Npx
  \cs_gset_protected:Npn
  \cs_gset_protected:Npx

```

```

1493 \tex_let:D \cs_gset_nopar:Npn           \tex_gdef:D
1494 \tex_let:D \cs_gset_nopar:Npx           \tex_xdef:D
1495 \cs_set_protected:Npn \cs_gset:Npn
1496 { \tex_long:D \tex_gdef:D }
1497 \cs_set_protected:Npn \cs_gset:Npx
1498 { \tex_long:D \tex_xdef:D }
1499 \cs_set_protected:Npn \cs_gset_protected_nopar:Npn
1500 { \tex_protected:D \tex_gdef:D }
1501 \cs_set_protected:Npn \cs_gset_protected_nopar:Npx
1502 { \tex_protected:D \tex_xdef:D }
1503 \cs_set_protected:Npn \cs_gset_protected:Npn
1504 { \tex_protected:D \tex_long:D \tex_gdef:D }
1505 \cs_set_protected:Npn \cs_gset_protected:Npx
1506 { \tex_protected:D \tex_long:D \tex_xdef:D }

```

(End definition for `\cs_gset_nopar:Npn` and others. These functions are documented on page 12.)

5.4 Selecting tokens

1507 `<@@=exp>`

`\l__exp_internal_tl`

Scratch token list variable for `\l3expan`, used by `\use:x`, used in defining conditionals. We don't use `t1` methods because `\l3basics` is loaded earlier.

1508 `\cs_set_nopar:Npn \l__exp_internal_tl { }`

(End definition for `\l__exp_internal_t1`.)

\use:c This macro grabs its argument and returns a csname from it.

```
1509 \cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }
```

(End definition for `\use:c`. This function is documented on page 16.)

\use:x Fully expands its argument and passes it to the input stream. Uses the reserved `\l__exp_internal_t1` which we've set up above.

```
1510 \cs_set_protected:Npn \use:x #1
1511 {
1512     \cs_set_nopar:Npx \l__exp_internal_t1 {#1}
1513     \l__exp_internal_t1
1514 }
```

(End definition for `\use:x`. This function is documented on page 20.)

```
1515 <@=use>
```

\use:e In non-LuaTeXengines older than 2019, `\expanded` is emulated.

```
1516 \cs_set:Npn \use:e #1 { \tex_expanded:D {#1} }
1517 \tex_ifdefined:D \tex_expanded:D \tex_else:D
1518     \cs_set:Npn \use:e #1 { \exp_args:N \use:n {#1} }
1519 \tex_if:D
```

(End definition for `\use:e`. This function is documented on page 20.)

```
1520 <@=exp>
```

\use:n These macros grab their arguments and return them back to the input (with outer braces removed).

```
1521 \cs_set:Npn \use:n    #1      {#1}
1522 \cs_set:Npn \use:nn   #1#2    {#1#2}
1523 \cs_set:Npn \use:nnn  #1#2#3 {#1#2#3}
1524 \cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
```

(End definition for `\use:n` and others. These functions are documented on page 19.)

\use_i:nn The equivalent to L^AT_EX 2_ε's `\@firstoftwo` and `\@secondoftwo`.

```
1525 \cs_set:Npn \use_i:nn  #1#2 {#1}
1526 \cs_set:Npn \use_i:nn  #1#2 {#2}
```

(End definition for `\use_i:nn` and `\use_i:nn`. These functions are documented on page 19.)

\use_i:nnn We also need something for picking up arguments from a longer list.

```
1527 \cs_set:Npn \use_i:nnn  #1#2#3 {#1}
1528 \cs_set:Npn \use_i:nnn  #1#2#3 {#2}
1529 \cs_set:Npn \use_i:nnn  #1#2#3 {#3}
1530 \cs_set:Npn \use_i:nnn  #1#2#3 {#1#2}
1531 \cs_set:Npn \use_i:nnn  #1#2#3#4 {#1}
1532 \cs_set:Npn \use_i:nnn  #1#2#3#4 {#2}
1533 \cs_set:Npn \use_i:nnn  #1#2#3#4 {#3}
1534 \cs_set:Npn \use_i:nnn  #1#2#3#4 {#4}
```

(End definition for `\use_i:nnn` and others. These functions are documented on page 19.)

```
\use_ii_i:nn
```

```
1535 \cs_set:Npn \use_ii_i:nn #1#2 { #2 #1 }
```

(End definition for `\use_ii_i:nn`. This function is documented on page 20.)

```
\use_none_delimit_by_q_nil:w
```

```
\use_none_delimit_by_q_stop:w
```

```
\use_none_delimit_by_q_recursion_stop:w
```

Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```
1536 \cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
```

```
1537 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
```

```
1538 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }
```

(End definition for `\use_none_delimit_by_q_nil:w`, `\use_none_delimit_by_q_stop:w`, and `\use_none_delimit_by_q_recursion_stop:w`. These functions are documented on page 21.)

```
\use_i_delimit_by_q_nil:nw
```

```
\use_i_delimit_by_q_stop:nw
```

```
\use_i_delimit_by_q_recursion_stop:nw
```

Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

```
1539 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
```

```
1540 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
```

```
1541 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw
```

```
1542 #1#2 \q_recursion_stop {#1}
```

(End definition for `\use_i_delimit_by_q_nil:nw`, `\use_i_delimit_by_q_stop:nw`, and `\use_i_delimit_by_q_recursion_stop:nw`. These functions are documented on page 21.)

5.5 Gobbling tokens from input

```
\use_none:n
```

```
\use_none:nn
```

```
\use_none:nnn
```

```
\use_none:nnnn
```

```
\use_none:nnnnn
```

```
\use_none:nnnnnn
```

```
\use_none:nnnnnnn
```

```
\use_none:nnnnnnnn
```

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of n's following the : in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:nnnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once takes care of gobbling all the tokens in one go.

```
1543 \cs_set:Npn \use_none:n #1 { }
```

```
1544 \cs_set:Npn \use_none:nn #1#2 { }
```

```
1545 \cs_set:Npn \use_none:nnn #1#2#3 { }
```

```
1546 \cs_set:Npn \use_none:nnnn #1#2#3#4 { }
```

```
1547 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5 { }
```

```
1548 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6 { }
```

```
1549 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7 { }
```

```
1550 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8 { }
```

```
1551 \cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9 { }
```

(End definition for `\use_none:n` and others. These functions are documented on page 20.)

5.6 Debugging and patching later definitions

```
1552 <@=debug>
```

```
\__kernel_if_debug:TF
```

A more meaningful test of whether debugging is enabled than messing up with guards. We can also more easily change the logic in one place then. This is needed primarily for deprecations.

```
1553 \cs_set_protected:Npn \__kernel_if_debug:TF #1#2 {#2}
```

(End definition for `__kernel_if_debug:TF`.)

```

\debug_on:n Stubs.
\debug_off:n
1554 \cs_set_protected:Npn \debug_on:n #1
1555 {
1556     \__kernel_msg_error:nnx { kernel } { enable-debug }
1557     { \tl_to_str:n { \debug_on:n {#1} } }
1558 }
1559 \cs_set_protected:Npn \debug_off:n #1
1560 {
1561     \__kernel_msg_error:nnx { kernel } { enable-debug }
1562     { \tl_to_str:n { \debug_off:n {#1} } }
1563 }

```

(End definition for `\debug_on:n` and `\debug_off:n`. These functions are documented on page 24.)

```

\debug_suspend:
\debug_resume:
1564 \cs_set_protected:Npn \debug_suspend: { }
1565 \cs_set_protected:Npn \debug_resume: { }

```

(End definition for `\debug_suspend:` and `\debug_resume:`. These functions are documented on page 24.)

`__kernel_deprecation_code:nn`
`\g__debug_deprecation_on_tl`
`\g__debug_deprecation_off_tl`

Some commands were more recently deprecated and not yet removed; only make these into errors if the user requests it. This relies on two token lists, filled up in `\l3deprecation`.

```

1566 \cs_set_nopar:Npn \g__debug_deprecation_on_tl { }
1567 \cs_set_nopar:Npn \g__debug_deprecation_off_tl { }
1568 \cs_set_protected:Npn \__kernel_deprecation_code:nn #1#2
1569 {
1570     \tl_gput_right:Nn \g__debug_deprecation_on_tl {#1}
1571     \tl_gput_right:Nn \g__debug_deprecation_off_tl {#2}
1572 }

```

(End definition for `__kernel_deprecation_code:nn`, `\g__debug_deprecation_on_tl`, and `\g__debug_deprecation_off_tl`.)

5.7 Conditional processing and definitions

```
1573 <@=prg>
```

Underneath any predicate function (`_p`) or other conditional forms (TF, etc.) is a built-in logic saying that it after all of the testing and processing must return the `<state>` this leaves `\TeX` in. Therefore, a simple user interface could be something like

```

\if_meaning:w #1#2
    \prg_return_true:
\else:
    \if_meaning:w #1#3
        \prg_return_true:
\else:
    \prg_return_false:
\fi:
\fi:

```

Usually, a `\TeX` programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the `\TeX` programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

\prg_return_true:
\prg_return_false:

The idea here is that \exp:w expands fully any \else: and \fi: that are waiting to be discarded, before reaching the \exp_end: which leaves an empty expansion. The code can then leave either the first or second argument in the input stream. This means that all of the branching code has to contain at least two tokens: see how the logical tests are actually implemented to see this.

```
1574 \cs_set:Npn \prg_return_true:
1575   { \exp_after:wN \use_i:nn \exp:w }
1576 \cs_set:Npn \prg_return_false:
1577   { \exp_after:wN \use_ii:nn \exp:w }
```

An extended state space could be implemented by including a more elaborate function in place of \use_i:nn/\use_ii:nn. Provided two arguments are absorbed then the code would work.

(End definition for \prg_return_true: and \prg_return_false:. These functions are documented on page 107.)

Private version of \use_none_delimit_by_q_recursion_stop:w.

```
1578 \cs_set:Npn \__prg_use_none_delimit_by_q_recursion_stop:w
1579   #1 \q_prg_recursion_stop { }
```

(End definition for __prg_use_none_delimit_by_q_recursion_stop:w.)

\prg_set_conditional:Npnn

\prg_new_conditional:Npnn

\prg_set_protected_conditional:Npnn

\prg_new_protected_conditional:Npnn

__prg_generate_conditional_parm>NNNpnn

The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those Npnn type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed {\name} {\signature} {\boolean} {\set_or_new} {\maybe_protected} {\parameters} {\TF,...} {\code} to the auxiliary function responsible for defining all conditionals. Note that e stands for expandable and p for protected.

```
1580 \cs_set_protected:Npn \prg_set_conditional:Npnn
1581   { \__prg_generate_conditional_parm:NNNpnn \cs_set:Npn e }
1582 \cs_set_protected:Npn \prg_new_conditional:Npnn
1583   { \__prg_generate_conditional_parm:NNNpnn \cs_new:Npn e }
1584 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn
1585   { \__prg_generate_conditional_parm:NNNpnn \cs_set_protected:Npn p }
1586 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn
1587   { \__prg_generate_conditional_parm:NNNpnn \cs_new_protected:Npn p }
1588 \cs_set_protected:Npn \__prg_generate_conditional_parm:NNNpnn #1#2#3#4#
1589   {
1590     \use:x
1591     {
1592       \__prg_generate_conditional:nnNNNnnn
1593       \cs_split_function:N #3
1594     }
1595     #1 #2 {#4}
1596   }
```

(End definition for \prg_set_conditional:Npnn and others. These functions are documented on page 105.)

\prg_set_conditional:Nnn

\prg_new_conditional:Nnn

\prg_set_protected_conditional:Nnn

\prg_new_protected_conditional:Nnn

__prg_generate_conditional_count:NNNnn

__prg_generate_conditional_count:nnNNNnn

The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. Split the base function into name and signature. The second auxiliary

generates the parameter text from the number of letters in the signature. Then feed $\{\langle name \rangle\} \{\langle signature \rangle\} \langle boolean \rangle \{\langle set or new \rangle\} \{\langle maybe protected \rangle\} \{\langle parameters \rangle\} \{TF, \dots\} \{\langle code \rangle\}$ to the auxiliary function responsible for defining all conditionals. If the $\langle signature \rangle$ has more than 9 letters, the definition is aborted since TeX macros have at most 9 arguments. The erroneous case where the function name contains no colon is captured later.

```

1597  \cs_set_protected:Npn \prg_set_conditional:Nnn
1598    { \__prg_generate_conditional_count:NNNnn \cs_set:Npn e }
1599  \cs_set_protected:Npn \prg_new_conditional:Nnn
1600    { \__prg_generate_conditional_count:NNNnn \cs_new:Npn e }
1601  \cs_set_protected:Npn \prg_set_protected_conditional:Nnn
1602    { \__prg_generate_conditional_count:NNNnn \cs_set_protected:Npn p }
1603  \cs_set_protected:Npn \prg_new_protected_conditional:Nnn
1604    { \__prg_generate_conditional_count:NNNnn \cs_new_protected:Npn p }
1605  \cs_set_protected:Npn \__prg_generate_conditional_count:NNNnn #1#2#3
1606    {
1607      \use:x
1608      {
1609        \__prg_generate_conditional_count:nnNNNnn
1610        \cs_split_function:N #3
1611      }
1612      #1 #2
1613    }
1614  \cs_set_protected:Npn \__prg_generate_conditional_count:nnNNNnn #1#2#3#4#5
1615  {
1616    \__kernel_cs_parm_from_arg_count:nnF
1617    { \__prg_generate_conditional:nnNNNnnn {#1} {#2} #3 #4 #5 }
1618    { \tl_count:n {#2} }
1619    {
1620      \__kernel_msg_error:nnxx { kernel } { bad-number-of-arguments }
1621      { \token_to_str:c { #1 : #2 } }
1622      { \tl_count:n {#2} }
1623      \use_none:nn
1624    }
1625  }

```

(End definition for $\text{\prg_set_conditional:Nnn}$ and others. These functions are documented on page 105.)

```
\__prg_generate_conditional:nnNNNnnn
\__prg_generate_conditional:NNnnnnNw
\__prg_generate_conditional_test:w
\__prg_generate_conditional_fast:nw
```

The workhorse here is going through a list of desired forms, *i.e.*, p, TF, T and F. The first three arguments come from splitting up the base form of the conditional, which gives the name, signature and a boolean to signal whether or not there was a colon in the name. In the absence of a colon, we throw an error and don't define any conditional. The fourth and fifth arguments build up the defining function. The sixth is the parameters to use (possibly empty), the seventh is the list of forms to define, the eighth is the replacement text which we will augment when defining the forms. The use of \tl_to_str:n makes the later loop more robust.

A large number of our low-level conditionals look like $\langle code \rangle \text{\prg_return_true:} \text{\else:} \text{\prg_return_false:} \text{\fi:}$ so we optimize this special case by calling $\text{__prg_generate_conditional_fast:nw} \{\langle code \rangle\}$. This passes \use_i:nn instead of \use_i_ii:nnn to functions such as $\text{__prg_generate_p_form:wNNnnnnN}$.

```

1626  \cs_set_protected:Npn \__prg_generate_conditional:nnNNNnnn #1#2#3#4#5#6#7#8
1627    {

```

```

1628     \if_meaning:w \c_false_bool #3
1629         \_kernel_msg_error:nnx { kernel } { missing-colon }
1630             { \token_to_str:c {#1} }
1631             \exp_after:wN \use_none:nn
1632     \fi:
1633     \use:x
1634     {
1635         \exp_not:N \_prg_generate_conditional>NNnnnnNw
1636         \exp_not:n { #4 #5 {#1} {#2} {#6} }
1637         \_prg_generate_conditional_test:w
1638             #8 \s_prg_mark
1639                 \_prg_generate_conditional_fast:nw
1640                 \prg_return_true: \else: \prg_return_false: \fi: \s_prg_mark
1641                     \use_none:n
1642                     \exp_not:n { {#8} \use_i_i:nnn }
1643                     \tl_to_str:n {#7}
1644                     \exp_not:n { , \q_prg_recursion_tail , \q_prg_recursion_stop }
1645     }
1646 }
1647 \cs_set:Npn \_prg_generate_conditional_test:w
1648     #1 \prg_return_true: \else: \prg_return_false: \fi: \s_prg_mark #2
1649     { #2 {#1} }
1650 \cs_set:Npn \_prg_generate_conditional_fast:nw #1#2 \exp_not:n #3
1651     { \exp_not:n { {#1} \use_i:nn } }

```

Looping through the list of desired forms. First are six arguments and seventh is the form. Use the form to call the correct type. If the form does not exist, the \use:c construction results in \relax, and the error message is displayed (unless the form is empty, to allow for {T, , F}), then \use_none:nnnnnnnn cleans up. Otherwise, the error message is removed by the variant form.

```

1652 \cs_set_protected:Npn \_prg_generate_conditional>NNnnnnNw #1#2#3#4#5#6#7#8 ,
1653 {
1654     \if_meaning:w \q_prg_recursion_tail #8
1655         \exp_after:wN \_prg_use_none_delimit_by_q_recursion_stop:w
1656     \fi:
1657     \use:c { __prg_generate_ #8 _form:wNNnnnnN }
1658         \tl_if_empty:nF {#8}
1659         {
1660             \_kernel_msg_error:nnxx
1661                 { kernel } { conditional-form-unknown }
1662                 {#8} { \token_to_str:c { #3 : #4 } }
1663         }
1664         \use_none:nnnnnnnn
1665         \s_prg_stop
1666         #1 #2 {#3} {#4} {#5} {#6} #7
1667     \_prg_generate_conditional>NNnnnnNw #1 #2 {#3} {#4} {#5} {#6} #7
1668 }

```

(End definition for _prg_generate_conditional:nnNNNnnn and others.)

_prg_generate_p_form:wNNnnnn
_prg_generate_TF_form:wNNnnnnN
_prg_generate_T_form:wNNnnnnN
_prg_generate_F_form:wNNnnnn
_prg_p_true:w

How to generate the various forms. Those functions take the following arguments: 1: junk, 2: \cs_set:Npn or similar, 3: p (for protected conditionals) or e, 4: function name, 5: signature, 6: parameter text, 7: replacement (possibly trimmed by _prg_generate_conditional_fast:nw), 8: \use_i_i:nnn or \use_i:nn (for “fast” conditionals). Remember that the logic-returning functions expect two arguments to be present

after \exp_end:: notice the construction of the different variants relies on this, and that the TF and F variants will be slightly faster than the T version. The p form is only valid for expandable tests, we check for that by making sure that the second argument is empty. For “fast” conditionals, #7 has an extra \if_. To optimize a bit further we could replace \exp_after:wN \use_i:nnn and similar by a single macro similar to __prg_p_true:w. The drawback is that if the T or F arguments are actually missing, the recovery from the runaway argument would not insert \fi: back, messing up nesting of conditionals.

```

1669 \cs_set_protected:Npn \__prg_generate_p_form:wNNnnnnN
1670     #1 \s__prg_stop #2#3#4#5#6#7#8
1671 {
1672     \if_meaning:w e #3
1673         \exp_after:wN \use_i:nn
1674     \else:
1675         \exp_after:wN \use_i:nn
1676     \fi:
1677     {
1678         #8
1679         { \exp_args:Nc #2 { #4 _p: #5 } #6 }
1680         { { #7 \exp_end: \c_true_bool \c_false_bool } }
1681         { #7 \__prg_p_true:w \fi: \c_false_bool }
1682     }
1683     {
1684         \__kernel_msg_error:nnx { kernel } { protected-predicate }
1685         { \token_to_str:c { #4 _p: #5 } }
1686     }
1687 }
1688 \cs_set_protected:Npn \__prg_generate_T_form:wNNnnnnN
1689     #1 \s__prg_stop #2#3#4#5#6#7#8
1690 {
1691     #8
1692     { \exp_args:Nc #2 { #4 : #5 T } #6 }
1693     { { #7 \exp_end: \use:n \use_none:n } }
1694     { #7 \exp_after:wN \use_i:nn \fi: \use_none:n }
1695 }
1696 \cs_set_protected:Npn \__prg_generate_F_form:wNNnnnnN
1697     #1 \s__prg_stop #2#3#4#5#6#7#8
1698 {
1699     #8
1700     { \exp_args:Nc #2 { #4 : #5 F } #6 }
1701     { { #7 \exp_end: { } } }
1702     { #7 \exp_after:wN \use_none:nn \fi: \use:n }
1703 }
1704 \cs_set_protected:Npn \__prg_generate_TF_form:wNNnnnnN
1705     #1 \s__prg_stop #2#3#4#5#6#7#8
1706 {
1707     #8
1708     { \exp_args:Nc #2 { #4 : #5 TF } #6 }
1709     { { #7 \exp_end: } }
1710     { #7 \exp_after:wN \use_i:nnn \fi: \use_i:nn }
1711 }
1712 \cs_set:Npn \__prg_p_true:w \fi: \c_false_bool { \fi: \c_true_bool }
```

(End definition for __prg_generate_p_form:wNNnnnnN and others.)

```
\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn
\_\_prg_set_eq_conditional:NNNn
```

The setting-equal functions. Split both functions and feed $\{\langle name_1 \rangle\} \{\langle signature_1 \rangle\} \langle boolean_1 \rangle \{\langle name_2 \rangle\} \{\langle signature_2 \rangle\} \langle boolean_2 \rangle \langle copying function \rangle \langle conditions \rangle$, $\backslash q_prg_recursion_tail$, $\backslash q_prg_recursion_stop$ to a first auxiliary.

```
1713 \cs_set_protected:Npn \prg_set_eq_conditional:NNn
1714   { \_\_prg_set_eq_conditional:NNNn \cs_set_eq:cc }
1715 \cs_set_protected:Npn \prg_new_eq_conditional:NNn
1716   { \_\_prg_set_eq_conditional:NNNn \cs_new_eq:cc }
1717 \cs_set_protected:Npn \_\_prg_set_eq_conditional:NNNn #1#2#3#4
1718   {
1719     \use:x
1720     {
1721       \exp_not:N \_\_prg_set_eq_conditional:nnNnnNNw
1722       \cs_split_function:N #2
1723       \cs_split_function:N #3
1724       \exp_not:N #1
1725       \tl_to_str:n {#4}
1726       \exp_not:n { , \q_prg_recursion_tail , \q_prg_recursion_stop }
1727     }
1728   }
```

(End definition for $\prg_set_eq_conditional:NNn$, $\prg_new_eq_conditional:NNn$, and $__prg_set_eq_conditional:NNNn$. These functions are documented on page 106.)

Split the function to be defined, and setup a manual clist loop over argument #6 of the first auxiliary. The second auxiliary receives twice three arguments coming from splitting the function to be defined and the function to copy. Make sure that both functions contained a colon, otherwise we don't know how to build conditionals, hence abort. Call the looping macro, with arguments $\{\langle name_1 \rangle\} \{\langle signature_1 \rangle\} \{\langle name_2 \rangle\} \{\langle signature_2 \rangle\} \langle copying function \rangle$ and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

```
1729 \cs_set_protected:Npn \_\_prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
1730   {
1731     \if_meaning:w \c_false_bool #3
1732       \_kernel_msg_error:nnx { kernel } { missing-colon }
1733       { \token_to_str:c {#1} }
1734       \exp_after:wN \_\_prg_use_none_delimit_by_q_recursion_stop:w
1735     \fi:
1736     \if_meaning:w \c_false_bool #6
1737       \_kernel_msg_error:nnx { kernel } { missing-colon }
1738       { \token_to_str:c {#4} }
1739       \exp_after:wN \_\_prg_use_none_delimit_by_q_recursion_stop:w
1740     \fi:
1741     \_\_prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
1742   }
1743 \cs_set_protected:Npn \_\_prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
1744   {
1745     \if_meaning:w \q_prg_recursion_tail #6
1746       \exp_after:wN \_\_prg_use_none_delimit_by_q_recursion_stop:w
1747     \fi:
1748     \use:c { \_\_prg_set_eq_conditional_ #6 _form:wNnnnn }
1749     \tl_if_empty:nF {#6}
1750     {
1751       \_\_kernel_msg_error:nnxx
```

```

1752          { kernel } { conditional-form-unknown }
1753          {#6} { \token_to_str:c { #1 : #2 } }
1754      }
1755      \use_none:nnnnnn
1756      \s__prg_stop
1757      #5 {#1} {#2} {#3} {#4}
1758      \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
1759  }
1760 \cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1761  { #2 { #3 _p : #4 } { #5 _p : #6 } }
1762 \cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1763  { #2 { #3 : #4 TF } { #5 : #6 TF } }
1764 \cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1765  { #2 { #3 : #4 T } { #5 : #6 T } }
1766 \cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1767  { #2 { #3 : #4 F } { #5 : #6 F } }

```

(End definition for `__prg_set_eq_conditional:nnNnnNNw` and others.)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either TT or TF to be tested using `\if:w` but this was later changed to 00 and 01, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

`\c_true_bool` Here are the canonical boolean values.

`\c_false_bool`

```

1768 \tex_chardef:D \c_true_bool = 1 ~
1769 \tex_chardef:D \c_false_bool = 0 ~

```

(End definition for `\c_true_bool` and `\c_false_bool`. These variables are documented on page 22.)

5.8 Dissecting a control sequence

1770 `<@=cs>`

`__cs_count_signature:N` `__cs_count_signature:N <function>`

Splits the `<function>` into the `<name>` (*i.e.* the part before the colon) and the `<signature>` (*i.e.* after the colon). The `<number>` of tokens in the `<signature>` is then left in the input stream. If there was no `<signature>` then the result is the marker value `-1`.

`__cs_get_function_name:N *` `__cs_get_function_name:N <function>`

Splits the `<function>` into the `<name>` (*i.e.* the part before the colon) and the `<signature>` (*i.e.* after the colon). The `<name>` is then left in the input stream without the escape character present made up of tokens with category code 12 (other).

`__cs_get_function_signature:N *` `__cs_get_function_signature:N <function>`

Splits the `<function>` into the `<name>` (*i.e.* the part before the colon) and the `<signature>` (*i.e.* after the colon). The `<signature>` is then left in the input stream made up of tokens with category code 12 (other).

__cs_tmp:w

Function used for various short-term usages, for instance defining functions whose definition involves tokens which are hard to insert normally (spaces, characters with category other).

\cs_to_str:N
__cs_to_str:N
__cs_to_str:w

This converts a control sequence into the character string of its name, removing the leading escape character. This turns out to be a non-trivial matter as there are different cases:

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of the `\escapechar` is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N\` yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and TeX reads `__cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `\tex_roman numeral:D`. The second case is that the escape character is not printable. Then the `\if:w` test is unfinished after reading a the space from `\token_to_str:N\` , and the auxiliary `__cs_to_str:w` is expanded, feeding – as a second character for the test; the test is false, and TeX skips to `\fi:`, then performs `\token_to_str:N`, and stops the `\tex_roman numeral:D` with `\c_zero_int`. The last case is that the escape character is itself a space. In this case, the `\if:w` test is true, and the auxiliary `__cs_to_str:w` comes into play, inserting `-\int_value:w`, which expands `\c_zero_int` to the character 0. The initial `\tex_roman numeral:D` then sees 0, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the expansion of `\tex_roman numeral:D`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```
1771 \cs_set:Npn \cs_to_str:N
1772 {
```

We implement the expansion scheme using `\tex_roman numeral:D` terminating it with `\c_zero_int` rather than using `\exp:w` and `\exp_end:` as we normally do. The reason is that the code heavily depends on terminating the expansion with `\c_zero_int` so we make this dependency explicit.

```
1773 \tex_roman numeral:D
1774   \if:w \token_to_str:N \ \_\_cs_to_str:w \fi:
1775   \exp_after:wN \_\_cs_to_str:N \token_to_str:N
1776 }
1777 \cs_set:Npn \_\_cs_to_str:N #1 { \c_zero_int }
1778 \cs_set:Npn \_\_cs_to_str:w #1 \_\_cs_to_str:N
1779 { - \int_value:w \fi: \exp_after:wN \c_zero_int }
```

If speed is a concern we could use `\csstring` in LuaTeX. For the empty csname that primitive gives an empty result while the current `\cs_to_str:N` gives incorrect results in all engines (this is impossible to fix without huge performance hit).

(End definition for `\cs_to_str:N`, `_cs_to_str:N`, and `_cs_to_str:w`. This function is documented on page 17.)

`\cs_split_function:N`

`_cs_split_function_auxi:w` `_cs_split_function_auxii:w` This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean `<true>` or `<false>` is returned with `<true>` for when there is a colon in the function and `<false>` if there is not.

We cannot use `:` directly as it has the wrong category code so an `x`-type expansion is used to force the conversion.

First ensure that we actually get a properly evaluated string by expanding `\cs_to_str:N` twice. If the function contained a colon, the auxiliary takes as #1 the function name, delimited by the first colon, then the signature #2, delimited by `\s__cs_mark`, then `\c_true_bool` as #3, and #4 cleans up until `\s__cs_stop`. Otherwise, the #1 contains the function name and `\s__cs_mark \c_true_bool`, #2 is empty, #3 is `\c_false_bool`, and #4 cleans up. The second auxiliary trims the trailing `\s__cs_mark` from the function name if present (that is, if the original function had no colon).

```

1780 \cs_set_protected:Npn \_cs_tmp:w #1
1781   {
1782     \cs_set:Npn \cs_split_function:N ##1
1783       {
1784         \exp_after:wN \exp_after:wN \exp_after:wN
1785         \_cs_split_function_auxi:w
1786         \cs_to_str:N ##1 \s__cs_mark \c_true_bool
1787         ##1 \s__cs_mark \c_false_bool \s__cs_stop
1788       }
1789     \cs_set:Npn \_cs_split_function_auxi:w
1790       ##1 ##2 \s__cs_mark ##3##4 \s__cs_stop
1791     { \_cs_split_function_auxii:w ##1 \s__cs_mark \s__cs_stop {##2} ##3 }
1792     \cs_set:Npn \_cs_split_function_auxii:w ##1 \s__cs_mark ##2 \s__cs_stop
1793     { {##1} }
1794   }
1795 \exp_after:wN \_cs_tmp:w \token_to_str:N :

```

(End definition for `\cs_split_function:N`, `_cs_split_function_auxi:w`, and `_cs_split_function_auxii:w`. This function is documented on page 17.)

5.9 Exist or free

A control sequence is said to *exist* (to be used) if has an entry in the hash table and its meaning is different from the primitive `\relax` token. A control sequence is said to be *free* (to be defined) if it does not already exist.

`\cs_if_exist_p:N`

`\cs_if_exist_p:c`

`\cs_if_exist:NTF`

`\cs_if_exist:cTF`

```

1796 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1797   {
1798     \if_meaning:w #1 \scan_stop:
1799       \prg_return_false:
1800     \else:
1801       \if_cs_exist:N #1
1802         \prg_return_true:
1803       \else:

```

```

1804         \prg_return_false:
1805     \fi:
1806     \fi:
1807 }

```

For the c form we firstly check if it is in the hash table and then for \scan_stop: so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```

1808 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1809 {
1810     \if_cs_exist:w #1 \cs_end:
1811         \exp_after:wN \use_i:nn
1812     \else:
1813         \exp_after:wN \use_ii:nn
1814     \fi:
1815 {
1816     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1817         \prg_return_false:
1818     \else:
1819         \prg_return_true:
1820     \fi:
1821 }
1822     \prg_return_false:
1823 }

```

(End definition for \cs_if_exist:NTF. This function is documented on page 22.)

\cs_if_free_p:N The logical reversal of the above.

```

\cs_if_free_p:c 1824 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
\cs_if_free:NTF 1825 {
\cs_if_free:cTF 1826     \if_meaning:w #1 \scan_stop:
1827         \prg_return_true:
1828     \else:
1829         \if_cs_exist:N #1
1830             \prg_return_false:
1831         \else:
1832             \prg_return_true:
1833         \fi:
1834     \fi:
1835 }
1836 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1837 {
1838     \if_cs_exist:w #1 \cs_end:
1839         \exp_after:wN \use_i:nn
1840     \else:
1841         \exp_after:wN \use_ii:nn
1842     \fi:
1843 {
1844     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1845         \prg_return_true:
1846     \else:
1847         \prg_return_false:

```

```
1848     \fi:  
1849 }  
1850 { \prg_return_true: }  
1851 }
```

`\cs_if_exist_use:N` `\cs_if_exist_use:c` The `\cs_if_exist_use:...` functions cannot be implemented as conditionals because the true branch must leave both the control sequence itself and the true code in the input stream. For the `c` variants, we are careful not to put the control sequence in the hash table if it does not exist. In LuaTeX we could use the `\lastnamedcs` primitive.

```
1852 \cs_set:Npn \cs_if_exist_use:NTF #1#2
1853 { \cs_if_exist:NTF #1 { #1 #2 } }
1854 \cs_set:Npn \cs_if_exist_use:NF #1
1855 { \cs_if_exist:NTF #1 { #1 } }
1856 \cs_set:Npn \cs_if_exist_use:NT #1 #2
1857 { \cs_if_exist:NTF #1 { #1 #2 } { } }
1858 \cs_set:Npn \cs_if_exist_use:N #1
1859 { \cs_if_exist:NTF #1 { #1 } { } }
1860 \cs_set:Npn \cs_if_exist_use:cTF #1#2
1861 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
1862 \cs_set:Npn \cs_if_exist_use:cF #1
1863 { \cs_if_exist:cTF {#1} { \use:c {#1} } }
1864 \cs_set:Npn \cs_if_exist_use:cT #1#2
1865 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
1866 \cs_set:Npn \cs_if_exist_use:c #1
1867 { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }
```

(End definition for `\cs_if_exist_use:NTF`. This function is documented on page [16](#).)

5.10 Preliminaries for new functions

We provide two kinds of functions that can be used to define control sequences. On the one hand we have functions that check if their argument doesn't already exist, they are called \..._new. The second type of defining functions doesn't check if the argument is already defined.

Before we can define them, we need some auxiliary macros that allow us to generate error messages. The next few definitions here are only temporary, they will be redefined later on.

`_kernel_msg_error:nnx` If an internal error occurs before LATEX3 has loaded l3msg then the code should issue a usable if terse error message and halt. This can only happen if a coding error is made by the team, so this is a reasonable response. Setting the `\newlinechar` is needed, to turn `^\^J` into a proper line break in plain T_EX.

```

1877         This~is~one~for~The~LaTeX3~Project:~bailing~out
1878     }
1879     \tex_end:D
1880 }
1881 \cs_set_protected:Npn \__kernel_msg_error:nnx #1#2#3
1882   { \__kernel_msg_error:nnxx {#1} {#2} {#3} { } }
1883 \cs_set_protected:Npn \__kernel_msg_error:nn #1#2
1884   { \__kernel_msg_error:nnxx {#1} {#2} { } { } }

(End definition for \__kernel_msg_error:nnxx, \__kernel_msg_error:nnx, and \__kernel_msg_error:nn.)
```

\msg_line_context: Another one from l3msg which will be altered later.

```

1885 \cs_set:Npn \msg_line_context:
1886   { on-line~ \tex_the:D \tex_inputlineno:D }
```

(End definition for \msg_line_context:. This function is documented on page 152.)

\iow_log:x We define a routine to write only to the log file. And a similar one for writing to both the log file and the terminal. These will be redefined later by l3io.

```

1887 \cs_set_protected:Npn \iow_log:x
1888   { \tex_immediate:D \tex_write:D -1 }
1889 \cs_set_protected:Npn \iow_term:x
1890   { \tex_immediate:D \tex_write:D 16 }
```

(End definition for \iow_log:n. This function is documented on page 162.)

__kernel_chk_if_free_cs:N This command is called by \cs_new_nopar:Npn and \cs_new_eq:NN etc. to make sure that the argument sequence is not already in use. If it is, an error is signalled. It checks if *<csname>* is undefined or \scan_stop:. Otherwise an error message is issued. We have to make sure we don't put the argument into the conditional processing since it may be an \if... type function!

```

1891 \cs_set_protected:Npn \__kernel_chk_if_free_cs:N #1
1892   {
1893     \cs_if_free:NF #1
1894     {
1895       \__kernel_msg_error:nnxx { kernel } { command-already-defined }
1896       { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1897     }
1898   }
1899 \cs_set_protected:Npn \__kernel_chk_if_free_cs:c
1900   { \exp_args:Nc \__kernel_chk_if_free_cs:N }
```

(End definition for __kernel_chk_if_free_cs:N.)

5.11 Defining new functions

```
1901 <@=cs>
```

\cs_new_nopar:Npn Function which check that the control sequence is free before defining it.

```

1902 \cs_set:Npn \__cs_tmp:w #1#2
1903   {
1904     \cs_set_protected:Npn #1 ##1
1905     {
1906       \__kernel_chk_if_free_cs:N ##1
1907       #2 ##1
1908     }
1909   }
```

```

1908     }
1909   }
1910   \__cs_tmp:w \cs_new_nopar:Npn           \cs_gset_nopar:Npn
1911   \__cs_tmp:w \cs_new_nopar:Npx          \cs_gset_nopar:Npx
1912   \__cs_tmp:w \cs_new:Npn                \cs_gset:Npn
1913   \__cs_tmp:w \cs_new:Npx               \cs_gset:Npx
1914   \__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1915   \__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1916   \__cs_tmp:w \cs_new_protected:Npn      \cs_gset_protected:Npn
1917   \__cs_tmp:w \cs_new_protected:Npx      \cs_gset_protected:Npx

```

(End definition for `\cs_new_nopar:Npn` and others. These functions are documented on page 11.)

`\cs_set_nopar:cpn`
`\cs_set_nopar:cpx`
`\cs_gset_nopar:cpn`
`\cs_gset_nopar:cpx`
`\cs_new_nopar:cpn`
`\cs_new_nopar:cpx`

Like `\cs_set_nopar:Npn` and `\cs_new_nopar:Npn`, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the c stands for csname argument, see the expansion module). Global versions are also provided.

`\cs_set_nopar:cpn` *⟨string⟩* *⟨rep-text⟩* turns *⟨string⟩* into a csname and then assigns *⟨rep-text⟩* to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

1918 \cs_set:Npn \__cs_tmp:w #1#2
1919   { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
1920 \__cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1921 \__cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
1922 \__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1923 \__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1924 \__cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1925 \__cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End definition for `\cs_set_nopar:Npn`. This function is documented on page 11.)

`\cs_set:cpn`
`\cs_set:cpx`
`\cs_gset:cpn`
`\cs_gset:cpx`
`\cs_new:cpn`
`\cs_new:cpx`

Variants of the `\cs_set:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```

1926 \__cs_tmp:w \cs_set:cpn \cs_set:Npn
1927 \__cs_tmp:w \cs_set:cpx \cs_set:Npx
1928 \__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
1929 \__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1930 \__cs_tmp:w \cs_new:cpn \cs_new:Npn
1931 \__cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End definition for `\cs_set:Npn`. This function is documented on page 11.)

`\cs_set_protected_nopar:cpn`
`\cs_set_protected_nopar:cpx`
`\cs_gset_protected_nopar:cpn`
`\cs_gset_protected_nopar:cpx`
`\cs_new_protected_nopar:cpn`
`\cs_new_protected_nopar:cpx`

Variants of the `\cs_set_protected_nopar:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```

1932 \__cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
1933 \__cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
1934 \__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
1935 \__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1936 \__cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1937 \__cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npx

```

(End definition for `\cs_set_protected_nopar:Npn`. This function is documented on page 12.)

```

\cs_set_protected:cnpn
\cs_set_protected:cpx
\cs_gset_protected:cnpn
\cs_gset_protected:cpx
\cs_new_protected:cnpn
\cs_new_protected:cpx

```

Variants of the `\cs_set_protected:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```

1938 \__cs_tmp:w \cs_set_protected:cnpn \cs_set_protected:Npn
1939 \__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
1940 \__cs_tmp:w \cs_gset_protected:cnpn \cs_gset_protected:Npn
1941 \__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
1942 \__cs_tmp:w \cs_new_protected:cnpn \cs_new_protected:Npn
1943 \__cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

```

(End definition for `\cs_set_protected:Npn`. This function is documented on page 11.)

5.12 Copying definitions

```

\cs_set_eq:NN
\cs_set_eq:cN
\cs_set_eq:Nc
\cs_set_eq:cc
\cs_gset_eq:NN
\cs_gset_eq:cN
\cs_gset_eq:Nc
\cs_gset_eq:cc
\cs_new_eq:NN
\cs_new_eq:cN
\cs_new_eq:Nc
\cs_new_eq:cc

```

These macros allow us to copy the definition of a control sequence to another control sequence. The = sign allows us to define funny char tokens like = itself or \sqcup with this function. For the definition of `\c_space_char{~}` to work we need the ~ after the =.

`\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it long in order to throw an “already defined” error rather than “runaway argument”.

```

1944 \cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
1945 \cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
1946 \cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
1947 \cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
1948 \cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }
1949 \cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }
1950 \cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1951 \cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }
1952 \cs_new_protected:Npn \cs_new_eq:NN #1
1953 {
1954     \__kernel_chk_if_free_cs:N #1
1955     \tex_global:D \cs_set_eq:NN #1
1956 }
1957 \cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1958 \cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
1959 \cs_new_protected:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }

```

(End definition for `\cs_set_eq:NN`, `\cs_gset_eq:NN`, and `\cs_new_eq:NN`. These functions are documented on page 15.)

5.13 Undefining functions

```

\cs_undefine:N
\cs_undefine:c

```

The following function is used to free the main memory from the definition of some function that isn’t in use any longer. The c variant is careful not to add the control sequence to the hash table if it isn’t there yet, and it also avoids nesting TeX conditionals in case #1 is unbalanced in this matter.

```

1960 \cs_new_protected:Npn \cs_undefine:N #1
1961     { \cs_gset_eq:NN #1 \tex_undefined:D }
1962 \cs_new_protected:Npn \cs_undefine:c #1
1963     {
1964         \if_cs_exist:w #1 \cs_end:
1965             \exp_after:wN \use:n
1966         \else:

```

```

1967     \exp_after:wN \use_none:n
1968     \fi:
1969     { \cs_gset_eq:cN {#1} \tex_undefined:D }
1970   }

```

(End definition for `\cs_undefine:N`. This function is documented on page 15.)

5.14 Generating parameter text from argument count

1971 $\langle @=cs \rangle$

`_kernel_cs_parm_from_arg_count:nnF`
`_cs_parm_from_arg_count_test:nnF`

LATEX3 provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text `{#1...#n}`, where n is the result of evaluating the second argument (as described in `\int_eval:n`). If the second argument gives a result outside the range [0,9], the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

```

1972 \cs_set_protected:Npn \_kernel_cs_parm_from_arg_count:nnF #1#2
1973   {
1974     \exp_args:Nx \_cs_parm_from_arg_count_test:nnF
1975     {
1976       \exp_after:wN \exp_not:n
1977       \if_case:w \int_eval:n {#2}
1978         {}
1979         \or: { ##1 }
1980         \or: { ##1##2 }
1981         \or: { ##1##2##3 }
1982         \or: { ##1##2##3##4 }
1983         \or: { ##1##2##3##4##5 }
1984         \or: { ##1##2##3##4##5##6 }
1985         \or: { ##1##2##3##4##5##6##7 }
1986         \or: { ##1##2##3##4##5##6##7##8 }
1987         \or: { ##1##2##3##4##5##6##7##8##9 }
1988         \else: { \c_false_bool }
1989       \fi:
1990     }
1991     {#1}
1992   }
1993 \cs_set_protected:Npn \_cs_parm_from_arg_count_test:nnF #1#2
1994   {
1995     \if_meaning:w \c_false_bool #1
1996       \exp_after:wN \use_i:nn
1997     \else:
1998       \exp_after:wN \use_i:nn
1999     \fi:
2000     { #2 {#1} }
2001   }

```

(End definition for `_kernel_cs_parm_from_arg_count:nnF` and `_cs_parm_from_arg_count_test:nnF`.)

5.15 Defining functions from a given number of arguments

2002 $\langle @@=cs \rangle$

`__cs_count_signature:N`
`__cs_count_signature:c`
`__cs_count_signature:n`
`__cs_count_signature:nnN`

Counting the number of tokens in the signature, *i.e.*, the number of arguments the function should take. Since this is not used in any time-critical function, we simply use `\tl_count:n` if there is a signature, otherwise `-1` arguments to signal an error. We need a variant form right away.

```
2003 \cs_new:Npn \__cs_count_signature:N #1
2004   { \exp_args:Nf \__cs_count_signature:n { \cs_split_function:N #1 } }
2005 \cs_new:Npn \__cs_count_signature:n #1
2006   { \int_eval:n { \__cs_count_signature:nnN #1 } }
2007 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
2008   {
2009     \if_meaning:w \c_true_bool #3
2010     \tl_count:n {#2}
2011   \else:
2012     -1
2013   \fi:
2014 }
2015 \cs_new:Npn \__cs_count_signature:c
2016   { \exp_args:Nc \__cs_count_signature:N }
```

(End definition for `__cs_count_signature:N`, `__cs_count_signature:n`, and `__cs_count_signature:nnN`.)

`\cs_generate_from_arg_count>NNnn`
`\cs_generate_from_arg_count:cNnn`
`\cs_generate_from_arg_count:Ncmn`

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since TeX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```
2017 \cs_new_protected:Npn \cs_generate_from_arg_count>NNnn #1#2#3#4
2018   {
2019     \__kernel_cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
2020   {
2021     \__kernel_msg_error:nnxx { kernel } { bad-number-of-arguments }
2022     { \token_to_str:N #1 } { \int_eval:n {#3} }
2023     \use_none:n
2024   }
2025   {#4}
2026 }
```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```
2027 \cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
2028   { \exp_args:Nc \cs_generate_from_arg_count>NNnn }
2029 \cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
2030   { \exp_args:NNc \cs_generate_from_arg_count>NNnn }
```

(End definition for `\cs_generate_from_arg_count>NNnn`. This function is documented on page 14.)

5.16 Using the signature to define functions

2031 ⟨@@=cs⟩

We can now combine some of the tools we have to provide a simple interface for defining functions, where the number of arguments is read from the signature. For instance, `\cs_set:Nn \foo_bar:nn {#1,#2}`.

We want to define `\cs_set:Nn` as

```

2032   \cs_set:Nn
2033     \cs_set:Nx
2034       \cs_set_nopar:Nn
2035       \cs_set_nopar:Nx
2036       \cs_set_protected:Nn
2037       \cs_set_protected:Nx
2038   \cs_set_protected_nopar:Nn
2039     \cs_gset:Nn
2040       \cs_gset_nopar:Nn
2041       \cs_gset_nopar:Nx
2042       \cs_gset_protected:Nn
2043       \cs_gset_protected:Nx
2044   \cs_gset_protected_nopar:Nn
2045     \cs_new:Nn
2046       \cs_new:Nx
2047       \cs_new_nopar:Nn
2048       \cs_new_nopar:Nx
2049       \cs_new_protected:Nn
2050       \cs_new_protected:Nx
2051       \cs_new_protected_nopar:Nn
2052       \cs_new_protected_nopar:Nx
2053       \use:x
2054         {
2055           \__cs_generate_from_signature:nnNNNn
2056           \cs_split_function:N #2
2057         }
2058         #1 #2
2059       \__cs_generate_from_signature:NNNn #1#2#3#4#5#6
2060     {
2061       \bool_if:NTF #3
2062         {
2063           \str_if_eq:eeF { }
2064             {
2065               \tl_map_function:nN {#2} \__cs_generate_from_signature:n
2066               {
2067                 \__kernel_msg_error:nnx { kernel } { non-base-function }
2068                 {
2069                   \token_to_str:N #5
2070                 }
2071               }
2072             {
2073               \__kernel_msg_error:nnx { kernel } { missing-colon }
2074               {
2075                 \token_to_str:N #5
2076               }
2077             }
2078           \__cs_generate_from_arg_count>NNnn
2079           #5 #4 { \tl_count:n {#2} } {#6}
2080         }
2081       {
2082         \__kernel_msg_error:nnx { kernel } { missing-colon }
2083         {
2084           \token_to_str:N #5
2085         }
2086       }
2087     }
2088   \cs_new:Npn \__cs_generate_from_signature:n #1

```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```

2032 \cs_set:Npn \__cs_tmp:w #1#2#3
2033   {
2034     \cs_new_protected:cpx { cs_ #1 : #2 }
2035     {
2036       \exp_not:N \__cs_generate_from_signature:NNn
2037       \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
2038     }
2039   }
2040 \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
2041   {
2042     \use:x
2043     {
2044       \__cs_generate_from_signature:nnNNNn
2045       \cs_split_function:N #2
2046     }
2047     #1 #2
2048   }
2049 \cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
2050   {
2051     \bool_if:NTF #3
2052       {
2053         \str_if_eq:eeF { }
2054           {
2055             \tl_map_function:nN {#2} \__cs_generate_from_signature:n
2056             {
2057               \__kernel_msg_error:nnx { kernel } { non-base-function }
2058               {
2059                 \token_to_str:N #5
2060               }
2061             }
2062           {
2063             \__kernel_msg_error:nnx { kernel } { missing-colon }
2064             {
2065               \token_to_str:N #5
2066             }
2067           }
2068         \__cs_generate_from_arg_count>NNnn
2069         #5 #4 { \tl_count:n {#2} } {#6}
2070       }
2071     {
2072       \__kernel_msg_error:nnx { kernel } { missing-colon }
2073       {
2074         \token_to_str:N #5
2075       }
2076     }
2077   }
2078 \cs_new:Npn \__cs_generate_from_signature:n #1

```

```

2068   {
2069     \if:w n #1 \else: \if:w N #1 \else:
2070     \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi:
2071   }

```

Then we define the 24 variants beginning with N.

```

2072 \__cs_tmp:w { set }           { Nn } { Npn }
2073 \__cs_tmp:w { set }           { Nx } { Npx }
2074 \__cs_tmp:w { set_nopar }     { Nn } { Npn }
2075 \__cs_tmp:w { set_nopar }     { Nx } { Npx }
2076 \__cs_tmp:w { set_protected } { Nn } { Npn }
2077 \__cs_tmp:w { set_protected } { Nx } { Npx }
2078 \__cs_tmp:w { set_protected_nopar } { Nn } { Npn }
2079 \__cs_tmp:w { set_protected_nopar } { Nx } { Npx }
2080 \__cs_tmp:w { gset }          { Nn } { Npn }
2081 \__cs_tmp:w { gset }          { Nx } { Npx }
2082 \__cs_tmp:w { gset_nopar }    { Nn } { Npn }
2083 \__cs_tmp:w { gset_nopar }    { Nx } { Npx }
2084 \__cs_tmp:w { gset_protected } { Nn } { Npn }
2085 \__cs_tmp:w { gset_protected } { Nx } { Npx }
2086 \__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
2087 \__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
2088 \__cs_tmp:w { new }          { Nn } { Npn }
2089 \__cs_tmp:w { new }          { Nx } { Npx }
2090 \__cs_tmp:w { new_nopar }     { Nn } { Npn }
2091 \__cs_tmp:w { new_nopar }     { Nx } { Npx }
2092 \__cs_tmp:w { new_protected } { Nn } { Npn }
2093 \__cs_tmp:w { new_protected } { Nx } { Npx }
2094 \__cs_tmp:w { new_protected_nopar } { Nn } { Npn }
2095 \__cs_tmp:w { new_protected_nopar } { Nx } { Npx }

```

(End definition for `\cs_set:Nn` and others. These functions are documented on page 13.)

`\cs_set:cn` The 24 c variants simply use `\exp_args:Nc`.

```

\cs_set:cx
\cs_set:nopar:cn
\cs_set:nopar:cx
\cs_set:protected:cn
\cs_set:protected:cx
\cs_set_protected_nopar:cn
\cs_set_protected_nopar:cx
\cs_gset:cn
\cs_gset:cx
\cs_gset:nopar:cn
\cs_gset:nopar:cx
\cs_gset:protected:cn
\cs_gset:protected:cx
\cs_gset_protected_nopar:cn
\cs_gset_protected_nopar:cx
\cs_new:cn
\cs_new:cx
\cs_new:nopar:cn
\cs_new:nopar:cx
\cs_new:protected:cn
\cs_new:protected:cx
\cs_new_protected_nopar:cn
\cs_new_protected_nopar:cx

```

```

2117 \__cs_tmp:w { gset_protected }      { x }
2118 \__cs_tmp:w { gset_protected_nopar } { n }
2119 \__cs_tmp:w { gset_protected_nopar } { x }
2120 \__cs_tmp:w { new }                 { n }
2121 \__cs_tmp:w { new }                 { x }
2122 \__cs_tmp:w { new_nopar }          { n }
2123 \__cs_tmp:w { new_nopar }          { x }
2124 \__cs_tmp:w { new_protected }       { n }
2125 \__cs_tmp:w { new_protected }       { x }
2126 \__cs_tmp:w { new_protected_nopar } { n }
2127 \__cs_tmp:w { new_protected_nopar } { x }

```

(End definition for `\cs_set:Nn`. This function is documented on page 13.)

5.17 Checking control sequence equality

`\cs_if_eq_p:NN` Check if two control sequences are identical.

```

2128 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
2129 {
2130   \if_meaning:w #1#2
2131     \prg_return_true: \else: \prg_return_false: \fi:
2132   }
2133 \cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
2134 \cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
2135 \cs_new:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }
2136 \cs_new:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
2137 \cs_new:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
2138 \cs_new:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
2139 \cs_new:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNT }
2140 \cs_new:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
2141 \cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
2142 \cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
2143 \cs_new:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
2144 \cs_new:Npn \cs_if_eq:ccF { \exp_args:Ncc \cs_if_eq:NNF }

```

(End definition for `\cs_if_eq:NNTF`. This function is documented on page 22.)

5.18 Diagnostic functions

2145 `<@=kernel>`

`__kernel_chk_defined:NT` Error if the variable #1 is not defined.

```

2146 \cs_new_protected:Npn \__kernel_chk_defined:NT #1#2
2147 {
2148   \cs_if_exist:NTF #1
2149   {#2}
2150   {
2151     \__kernel_msg_error:nnx { kernel } { variable-not-defined }
2152     { \token_to_str:N #1 }
2153   }
2154 }

```

(End definition for `__kernel_chk_defined:NT`.)

`_kernel_register_show:N`
`_kernel_register_show:c`
`_kernel_register_log:N`
`_kernel_register_log:c`
`_kernel_register_show_aux:NN`
`_kernel_register_show_aux:nNN`

Simply using the `\showthe` primitive does not allow for line-wrapping, so instead use `\tl_show:n` and `\tl_log:n` (defined in `l3tl` and that performs line-wrapping). This displays `>~(variable)=<value>`. We expand the value before-hand as otherwise some integers (such as `\currentgrouplevel` or `\currentgrouptype`) altered by the line-wrapping code would show wrong values.

```

2155 \cs_new_protected:Npn \_kernel_register_show:N
2156   { \_kernel_register_show_aux:NN \tl_show:n }
2157 \cs_new_protected:Npn \_kernel_register_show:c
2158   { \exp_args:Nc \_kernel_register_show:N }
2159 \cs_new_protected:Npn \_kernel_register_log:N
2160   { \_kernel_register_show_aux:NN \tl_log:n }
2161 \cs_new_protected:Npn \_kernel_register_log:c
2162   { \exp_args:Nc \_kernel_register_log:N }
2163 \cs_new_protected:Npn \_kernel_register_show_aux:NN #1#2
2164   {
2165     \_kernel_chk_defined:NT #2
2166     {
2167       \exp_args:No \_kernel_register_show_aux:nNN
2168       { \tex_the:D #2 } #2 #1
2169     }
2170   }
2171 \cs_new_protected:Npn \_kernel_register_show_aux:nNN #1#2#3
2172   { \exp_args:No #3 { \token_to_str:N #2 = #1 } }

```

(End definition for `_kernel_register_show:N` and others.)

`\cs_show:N`
`\cs_show:c`
`\cs_log:N`
`\cs_log:c`
`_kernel_show>NN`

Some control sequences have a very long name or meaning. Thus, simply using TeX's primitive `\show` could lead to overlong lines. The output of this primitive is mimicked to some extent, then the re-built string is given to `\tl_show:n` or `\tl_log:n` for line-wrapping. We must expand the meaning before passing it to the wrapping code as otherwise we would wrongly see the definitions that are in place there. To get correct escape characters, set the `\escapechar` in a group; this also localizes the assignment performed by x-expansion. The `\cs_show:c` and `\cs_log:c` commands convert their argument to a control sequence within a group to avoid showing `\relax` for undefined control sequences.

```

2173 \cs_new_protected:Npn \cs_show:N { \_kernel_show>NN \tl_show:n }
2174 \cs_new_protected:Npn \cs_show:c
2175   { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
2176 \cs_new_protected:Npn \cs_log:N { \_kernel_show>NN \tl_log:n }
2177 \cs_new_protected:Npn \cs_log:c
2178   { \group_begin: \exp_args:NNc \group_end: \cs_log:N }
2179 \cs_new_protected:Npn \_kernel_show>NN #1#2
2180   {
2181     \group_begin:
2182       \int_set:Nn \tex_escapechar:D { '\\" }
2183       \exp_args:NNx
2184     \group_end:
2185     #1 { \token_to_str:N #2 = \cs_meaning:N #2 }
2186   }

```

(End definition for `\cs_show:N`, `\cs_log:N`, and `_kernel_show>NN`. These functions are documented on page 16.)

5.19 Decomposing a macro definition

```
\cs_prefix_spec:N  
\cs_argument_spec:N  
\cs_replacement_spec:N  
\_kernel_prefix_arg_replacement:wN
```

We sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the argument specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token `\scan_stop:` is returned instead.

```
2187 \use:x  
2188 {  
2189   \exp_not:n { \cs_new:Npn \_kernel_prefix_arg_replacement:wN #1 }  
2190   \tl_to_str:n { macro : } \exp_not:n { #2 -> #3 \s__kernel_stop #4 }  
2191 }  
2192 { #4 {#1} {#2} {#3} }  
2193 \cs_new:Npn \cs_prefix_spec:N #1  
2194 {  
2195   \token_if_macro:NTF #1  
2196   {  
2197     \exp_after:wN \_kernel_prefix_arg_replacement:wN  
2198     \token_to_meaning:N #1 \s__kernel_stop \use_i:nnn  
2199   }  
2200   { \scan_stop: }  
2201 }  
2202 \cs_new:Npn \cs_argument_spec:N #1  
2203 {  
2204   \token_if_macro:NTF #1  
2205   {  
2206     \exp_after:wN \_kernel_prefix_arg_replacement:wN  
2207     \token_to_meaning:N #1 \s__kernel_stop \use_ii:nnn  
2208   }  
2209   { \scan_stop: }  
2210 }  
2211 \cs_new:Npn \cs_replacement_spec:N #1  
2212 {  
2213   \token_if_macro:NTF #1  
2214   {  
2215     \exp_after:wN \_kernel_prefix_arg_replacement:wN  
2216     \token_to_meaning:N #1 \s__kernel_stop \use_iii:nnn  
2217   }  
2218   { \scan_stop: }  
2219 }
```

(End definition for `\cs_prefix_spec:N` and others. These functions are documented on page 18.)

5.20 Doing nothing functions

```
\prg_do_nothing:
```

This does not fit anywhere else!

```
2220 \cs_new:Npn \prg_do_nothing: { }
```

(End definition for `\prg_do_nothing:`. This function is documented on page 9.)

5.21 Breaking out of mapping functions

2221 `<@=prg>`

`\prg_break_point:Nn`
`\prg_map_break:Nn`

In inline mappings, the nesting level must be reset at the end of the mapping, even when the user decides to break out. This is done by putting the code that must be performed as an argument of `_prg_break_point:Nn`. The breaking functions are then defined to jump to that point and perform the argument of `_prg_break_point:Nn`, before the user's code (if any). There is a check that we close the correct loop, otherwise we continue breaking.

```
2222 \cs_new_eq:NN \prg_break_point:Nn \use_i:nn
2223 \cs_new:Npn \prg_map_break:Nn #1#2#3 \prg_break_point:Nn #4#5
2224 {
2225     #5
2226     \if_meaning:w #1 #4
2227         \exp_after:wN \use_i:nnn
2228     \fi:
2229     \prg_map_break:Nn #1 {#2}
2230 }
```

(End definition for `\prg_break_point:Nn` and `\prg_map_break:Nn`. These functions are documented on page 113.)

`\prg_break_point:`
`\prg_break:`
`\prg_break:n`

Very simple analogues of `\prg_break_point:Nn` and `\prg_map_break:Nn`, for use in fast short-term recursions which are not mappings, do not need to support nesting, and in which nothing has to be done at the end of the loop.

```
2231 \cs_new_eq:NN \prg_break_point: \prg_do_nothing:
2232 \cs_new:Npn \prg_break: #1 \prg_break_point: { }
2233 \cs_new:Npn \prg_break:n #1#2 \prg_break_point: {#1}
```

(End definition for `\prg_break_point:`, `\prg_break:`, and `\prg_break:n`. These functions are documented on page 114.)

5.22 Starting a paragraph

`\mode_leave_vertical:`

The approach here is different to that used by L^AT_EX 2_ε or plain T_EX, which unbox a void box to force horizontal mode. That inserts the `\everypar` tokens *before* the re-inserted unboxing tokens. The approach here uses either the `\quitvmode` primitive or the equivalent protected macro. In vertical mode, the `\indent` primitive is inserted: this will switch to horizontal mode and insert `\everypar` tokens and nothing else. Unlike the L^AT_EX 2_ε version, the availability of ε-T_EX means using a mode test can be done at for example the start of an `\halign`.

```
2234 \cs_new_protected:Npn \mode_leave_vertical:
2235 {
2236     \if_mode_vertical:
2237         \exp_after:wN \tex_indent:D
2238     \fi:
2239 }
```

(End definition for `\mode_leave_vertical:`. This function is documented on page 24.)

2240 `</package>`

6 l3expan implementation

2241 `(*package)`

2242 `(@@=exp)`

`\l__exp_internal_t1` The `\exp_` module has its private variable to temporarily store the result of x-type argument expansion. This is done to avoid interference with other functions using temporary variables.

(End definition for \l__exp_internal_t1.)

`\exp_after:wN` These are defined in l3basics, as they are needed “early”. This is just a reminder of that fact!

`\exp_not:N`

`\exp_not:n`

(End definition for \exp_after:wN, \exp_not:N, and \exp_not:n. These functions are documented on page 33.)

6.1 General expansion

In this section a general mechanism for defining functions that handle arguments is defined. These general expansion functions are expandable unless x is used. (Any version of x is going to have to use one of the L^AT_EX3 names for `\cs_set:Npx` at some point, and so is never going to be expandable.)

The definition of expansion functions with this technique happens in section 6.8. In section 6.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l__exp_internal_t1` This scratch token list variable is defined in l3basics.

(End definition for \l__exp_internal_t1.)

This code uses internal functions with names that start with `\:::` to perform the expansions. All macros are long since the tokens undergoing expansion may be arbitrary user input.

An argument manipulator `\:::(Z)` always has signature `#1\:::#2#3` where #1 holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, #2 is the carried over result of the previous expansion steps and #3 is the argument about to be processed. One exception to this rule is `\:::p`, which has to grab an argument delimited by a left brace.

`_exp_arg_next:nnn` #1 is the result of an expansion step, #2 is the remaining argument manipulations and #3 is the current result of the expansion chain. This auxiliary function moves #1 back after #3 in the input stream and checks if any expansion is left to be done by calling #2. In by far the most cases we need to add a set of braces to the result of an argument manipulation so it is more effective to do it directly here. Actually, so far only the c of the final argument manipulation variants does not require a set of braces.

2243 `\cs_new:Npn _exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }`

2244 `\cs_new:Npn _exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }`

(End definition for _exp_arg_next:nnn and _exp_arg_next:Nnn.)

`\:::` The end marker is just another name for the identity function.

2245 `\cs_new:Npn \::: #1 {#1}`

(End definition for \:::. This function is documented on page 37.)

\:::n This function is used to skip an argument that doesn't need to be expanded.

```
2246 \cs_new:Npn \:::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(*End definition for \:::n. This function is documented on page 37.*)

\:::N This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
2247 \cs_new:Npn \:::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(*End definition for \:::N. This function is documented on page 37.*)

\:::p This function is used to skip an argument that is delimited by a left brace and doesn't need to be expanded. It is not wrapped in braces in the result.

```
2248 \cs_new:Npn \:::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(*End definition for \:::p. This function is documented on page 37.*)

\:::c This function is used to skip an argument that is turned into a control sequence without expansion.

```
2249 \cs_new:Npn \:::c #1 \::: #2#3
2250   { \exp_after:wN \__exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(*End definition for \:::c. This function is documented on page 37.*)

\:::o This function is used to expand an argument once.

```
2251 \cs_new:Npn \:::o #1 \::: #2#3
2252   { \exp_after:wN \__exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(*End definition for \:::o. This function is documented on page 37.*)

\:::e With the `\expanded` primitive available, just expand. Otherwise defer to `\exp_args:N` implemented later.

```
2253 \cs_if_exist:NTF \tex_expanded:D
2254   {
2255     \cs_new:Npn \:::e #1 \::: #2#3
2256       { \tex_expanded:D { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } } }
2257   }
2258   {
2259     \cs_new:Npn \:::e #1 \::: #2#3
2260       { \exp_args:N \__exp_arg_next:nnn {#3} {#1} {#2} }
2261   }
```

(*End definition for \:::e. This function is documented on page 37.*)

\:::f This function is used to expand a token list until the first unexpandable token is found. This is achieved through `\exp:w \exp_end_continue_f:w` that expands everything in its way following it. This scanning procedure is terminated once the expansion hits something non-expandable (if that is a space it is removed). We introduce `\exp_stop_f:` to mark such an end-of-expansion marker. For example, f-expanding `\cs_set_eq:Nc \aaa { b \l_tmpa_t1 b }` where `\l_tmpa_t1` contains the characters `lur` gives `\tex_let:D \aaa = \blurb` which then turns out to start with the non-expandable token `\tex_let:D`. Since the expansion of `\exp:w \exp_end_continue_f:w` is empty, we wind up with a fully expanded list, only TeX has not tried to execute any of

the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```

2262 \cs_new:Npn \:::f #1 \::: #2#3
2263 {
2264     \exp_after:wN \__exp_arg_next:nnn
2265     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2266     {#1} {#2}
2267 }
2268 \use:nn { \cs_new_eq:NN \exp_stop:f } { ~ }

```

(End definition for `\:::f` and `\exp_stop:f`. These functions are documented on page 37.)

`\:::x` This function is used to expand an argument fully. We build in the expansion of `__-exp_arg_next:nnn`.

```

2269 \cs_new_protected:Npn \:::x #1 \::: #2#3
2270 {
2271     \cs_set_nopar:Npx \l__exp_internal_tl
2272     { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } }
2273     \l__exp_internal_tl
2274 }

```

(End definition for `\:::x`. This function is documented on page 37.)

`\:::v` These functions return the value of a register, i.e., one of `tl`, `clist`, `int`, `skip`, `dim`, `\muskip`, or built-in TeX register. The `V` version expects a single token whereas `v` like `c` creates a csname from its argument given in braces and then evaluates it as if it was a `V`. The `\exp:w` sets off an expansion similar to an `f`-type expansion, which we terminate using `\exp_end:`. The argument is returned in braces.

```

2275 \cs_new:Npn \:::V #1 \::: #2#3
2276 {
2277     \exp_after:wN \__exp_arg_next:nnn
2278     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2279     {#1} {#2}
2280 }
2281 \cs_new:Npn \:::v #1 \::: #2#3
2282 {
2283     \exp_after:wN \__exp_arg_next:nnn
2284     \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2285     {#1} {#2}
2286 }

```

(End definition for `\:::v` and `\:::V`. These functions are documented on page 37.)

`__exp_eval_register:N` This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in TeX register such as `\count`. For the TeX registers we have to utilize a `\the` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\the` and when not to. What we want here is to find out whether the token expands to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the token in question when it has been prefixed with `\exp_not:N` and the token itself. If it is a macro, the prefixed `\exp_not:N` temporarily turns it into the primitive `\scan_stop:`.

```

2287 \cs_new:Npn \__exp_eval_register:N #1
2288 {
2289     \exp_after:wN \if_meaning:w \exp_not:N #1 #1

```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\scan_stop:`. In that case we throw an error. We could let `TeX` do it for us but that would result in the rather obscure

```
! You can't use '\relax' after \the.
```

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```
2290      \if_meaning:w \scan_stop: #1
2291          \__exp_eval_error_msg:w
2292      \fi:
```

The next bit requires some explanation. The function must be initiated by `\exp:w` and we want to terminate this expansion chain by inserting the `\exp_end:` token. However, we have to expand the register `#1` before we do that. If it is a `TeX` register, we need to execute the sequence `\exp_after:wN \exp_end: \tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN \exp_end: #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```
2293      \else:
2294          \exp_after:wN \use_i_i:nnn
2295      \fi:
2296      \exp_after:wN \exp_end: \tex_the:D #1
2297  }
2298 \cs_new:Npn \__exp_eval_register:c #1
2299 { \exp_after:wN \__exp_eval_register:N \cs:w #1 \cs_end: }
```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```
! Undefined control sequence.
<argument> \LaTeX3 error:
                    Erroneous variable used!
1.55 \tl_set:Nv \l_tmpa_tl {undefined_t1}

2300 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
2301 {
2302     \fi:
2303     \fi:
2304     \__kernel_msg_expandable_error:nnn { kernel } { bad-variable } {#2}
2305     \exp_end:
2306 }
```

(End definition for `__exp_eval_register:N` and `__exp_eval_error_msg:w`.)

6.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable.

`\exp_args:Nc` In `l3basics`.

`\exp_args:cc`

(End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 29.)

```

\exp_args:Nc
\exp_args:Ncc
\exp_args:Nccc
2307 \cs_new:Npn \exp_args:Nc #1#2#3
2308   { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
2309 \cs_new:Npn \exp_args:Ncc #1#2#3
2310   { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
2311 \cs_new:Npn \exp_args:Nccc #1#2#3#4
2312   {
2313     \exp_after:wN #1
2314       \cs:w #2 \exp_after:wN \cs_end:
2315       \cs:w #3 \exp_after:wN \cs_end:
2316       \cs:w #4 \cs_end:
2317   }

```

(End definition for `\exp_args:Nc`, `\exp_args:Ncc`, and `\exp_args:Nccc`. These functions are documented on page 31.)

```

\exp_args:No
\exp_args:Nno
\exp_args:NNno
2318 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
2319 \cs_new:Npn \exp_args:Nno #1#2#3
2320   { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
2321 \cs_new:Npn \exp_args:NNno #1#2#3#4
2322   { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }

```

(End definition for `\exp_args:No`, `\exp_args:Nno`, and `\exp_args:NNno`. These functions are documented on page 30.)

`\exp_args:Ne` When the `\expanded` primitive is available, use it. Otherwise use `_exp_e:nn`, defined later, to fully expand tokens.

```

2323 \cs_if_exist:NTF \tex_expanded:D
2324   {
2325     \cs_new:Npn \exp_args:Ne #1#2
2326       { \exp_after:wN #1 \tex_expanded:D { {#2} } }
2327   }
2328   {
2329     \cs_new:Npn \exp_args:Ne #1#2
2330       {
2331         \exp_after:wN #1 \exp_after:wN
2332           { \exp:w \_exp_e:nn {#2} { } }
2333       }
2334   }

```

(End definition for `\exp_args:Ne`. This function is documented on page 30.)

```

\exp_args:Nf
\exp_args:Nv
\exp_args:Nv
2335 \cs_new:Npn \exp_args:Nf #1#2
2336   { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
2337 \cs_new:Npn \exp_args:Nv #1#2
2338   {
2339     \exp_after:wN #1 \exp_after:wN
2340       { \exp:w \_exp_eval_register:c {#2} }
2341   }
2342 \cs_new:Npn \exp_args:NV #1#2
2343   {
2344     \exp_after:wN #1 \exp_after:wN
2345       { \exp:w \_exp_eval_register:N #2 }
2346   }

```

(End definition for `\exp_args:Nf`, `\exp_args:NV`, and `\exp_args:Nv`. These functions are documented on page 30.)

`\exp_args:NNV` Some more hand-tuned function with three arguments. If we forced that an o argument always has braces, we could implement `\exp_args:Nco` with less tokens and only two arguments.

```

2347 \cs_new:Npn \exp_args:NNV #1#2#3
2348 {
2349     \exp_after:wN #1
2350     \exp_after:wN #2
2351     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2352 }
2353 \cs_new:Npn \exp_args:NNv #1#2#3
2354 {
2355     \exp_after:wN #1
2356     \exp_after:wN #2
2357     \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2358 }
2359 \cs_if_exist:NTF \tex_expanded:D
2360 {
2361     \cs_new:Npn \exp_args:NNe #1#2#3
2362     {
2363         \exp_after:wN #1
2364         \exp_after:wN #2
2365         \tex_expanded:D { {#3} }
2366     }
2367 }
2368 { \cs_new:Npn \exp_args:NNe { \:::N \:::e \::: } }
2369 \cs_new:Npn \exp_args:NNf #1#2#3
2370 {
2371     \exp_after:wN #1
2372     \exp_after:wN #2
2373     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2374 }
2375 \cs_new:Npn \exp_args:Nco #1#2#3
2376 {
2377     \exp_after:wN #1
2378     \cs:w #2 \exp_after:wN \cs_end:
2379     \exp_after:wN {#3}
2380 }
2381 \cs_new:Npn \exp_args:NcV #1#2#3
2382 {
2383     \exp_after:wN #1
2384     \cs:w #2 \exp_after:wN \cs_end:
2385     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2386 }
2387 \cs_new:Npn \exp_args:Ncv #1#2#3
2388 {
2389     \exp_after:wN #1
2390     \cs:w #2 \exp_after:wN \cs_end:
2391     \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2392 }
2393 \cs_new:Npn \exp_args:Ncf #1#2#3
2394 {

```

```

2395     \exp_after:wN #1
2396     \cs:w #2 \exp_after:wN \cs_end:
2397     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2398   }
2399 \cs_new:Npn \exp_args:NVV #1#2#3
2400   {
2401     \exp_after:wN #1
2402     \exp_after:wN { \exp:w \exp_after:wN
2403       \_exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
2404     \exp_after:wN { \exp:w \_exp_eval_register:N #3 }
2405   }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page 31.)

`\exp_args:NNNV` A few more that we can hand-tune.

```

\exp_args:NNNv
\exp_args:NNNv
\exp_args:NcNc
\exp_args:NcNo
\exp_args:Ncco
2406 \cs_new:Npn \exp_args:NNNV #1#2#3#4
2407   {
2408     \exp_after:wN #1
2409     \exp_after:wN #2
2410     \exp_after:wN #3
2411     \exp_after:wN { \exp:w \_exp_eval_register:N #4 }
2412   }
2413 \cs_new:Npn \exp_args:NNNv #1#2#3#4
2414   {
2415     \exp_after:wN #1
2416     \exp_after:wN #2
2417     \exp_after:wN #3
2418     \exp_after:wN { \exp:w \_exp_eval_register:c {#4} }
2419   }
2420 \cs_new:Npn \exp_args:NcNc #1#2#3#4
2421   {
2422     \exp_after:wN #1
2423     \cs:w #2 \exp_after:wN \cs_end:
2424     \exp_after:wN #3
2425     \cs:w #4 \cs_end:
2426   }
2427 \cs_new:Npn \exp_args:NcNo #1#2#3#4
2428   {
2429     \exp_after:wN #1
2430     \cs:w #2 \exp_after:wN \cs_end:
2431     \exp_after:wN #3
2432     \exp_after:wN {#4}
2433   }
2434 \cs_new:Npn \exp_args:Ncco #1#2#3#4
2435   {
2436     \exp_after:wN #1
2437     \cs:w #2 \exp_after:wN \cs_end:
2438     \cs:w #3 \exp_after:wN \cs_end:
2439     \exp_after:wN {#4}
2440   }

```

(End definition for `\exp_args:NNNV` and others. These functions are documented on page 32.)

`\exp_args:Nx`

```

2441 \cs_new_protected:Npn \exp_args:Nx #1#2
2442   { \use:x { \exp_not:N #1 {#2} } }

```

(End definition for `\exp_args:Nx`. This function is documented on page 31.)

6.3 Last-unbraced versions

`_exp_arg_last_unbraced:nn`

```

\:::o_unbraced
\:::V_unbraced
\:::v_unbraced
\:::e_unbraced
\:::f_unbraced
\:::x_unbraced

2443 \cs_new:Npn \_exp_arg_last_unbraced:nn #1#2 { #2#1 }
2444 \cs_new:Npn \:::o_unbraced \::: #1#2
2445   { \exp_after:wN \_exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
2446 \cs_new:Npn \:::V_unbraced \::: #1#2
2447   {
2448     \exp_after:wN \_exp_arg_last_unbraced:nn
2449       \exp_after:wN { \exp:w \_exp_eval_register:N #2 } {#1}
2450   }
2451 \cs_new:Npn \:::v_unbraced \::: #1#2
2452   {
2453     \exp_after:wN \_exp_arg_last_unbraced:nn
2454       \exp_after:wN { \exp:w \_exp_eval_register:c {#2} } {#1}
2455   }
2456 \cs_if_exist:NTF \tex_expanded:D
2457   {
2458     \cs_new:Npn \:::e_unbraced \::: #1#2
2459       { \tex_expanded:D { \exp_not:n {#1} #2 } }
2460   }
2461   {
2462     \cs_new:Npn \:::e_unbraced \::: #1#2
2463       { \exp:w \_exp_e:nn {#2} {#1} }
2464   }
2465 \cs_new:Npn \:::f_unbraced \::: #1#2
2466   {
2467     \exp_after:wN \_exp_arg_last_unbraced:nn
2468       \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
2469   }
2470 \cs_new_protected:Npn \:::x_unbraced \::: #1#2
2471   {
2472     \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
2473     \l__exp_internal_tl
2474   }

```

(End definition for `_exp_arg_last_unbraced:nn` and others. These functions are documented on page 37.)

`\exp_last_unbraced:No`

`\exp_last_unbraced:NV`

`\exp_last_unbraced:Nv`

`\exp_last_unbraced:Ne`

`\exp_last_unbraced:Nf`

`\exp_last_unbraced:NNo`

`\exp_last_unbraced:NNV`

`\exp_last_unbraced:NNf`

`\exp_last_unbraced:Nco`

`\exp_last_unbraced:NcV`

`\exp_last_unbraced:NNNNo`

`\exp_last_unbraced:NNNV`

`\exp_last_unbraced:NNNf`

`\exp_last_unbraced:Nno`

`\exp_last_unbraced:Noo`

`\exp_last_unbraced:Nfo`

`\exp_last_unbraced:NnNo`

`\exp_last_unbraced:NNNNo`

`\exp_last_unbraced:NNNNf`

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

2475 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
2476 \cs_new:Npn \exp_last_unbraced:NV #1#2
2477   { \exp_after:wN #1 \exp:w \_exp_eval_register:N #2 }
2478 \cs_new:Npn \exp_last_unbraced:Nv #1#2
2479   { \exp_after:wN #1 \exp:w \_exp_eval_register:c {#2} }
2480 \cs_if_exist:NTF \tex_expanded:D
2481   {

```

```

2482     \cs_new:Npn \exp_last_unbraced:Ne #1#2
2483         { \exp_after:wN #1 \tex_expanded:D {#2} }
2484     }
2485     { \cs_new:Npn \exp_last_unbraced:Ne { \:::e_unbraced \::: } }
2486 \cs_new:Npn \exp_last_unbraced:Nf #1#2
2487     { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
2488 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
2489     { \exp_after:wN #1 \exp_after:wN #2 #3 }
2490 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
2491     {
2492         \exp_after:wN #1
2493         \exp_after:wN #2
2494         \exp:w \__exp_eval_register:N #3
2495     }
2496 \cs_new:Npn \exp_last_unbraced:NNf #1#2#3
2497     {
2498         \exp_after:wN #1
2499         \exp_after:wN #2
2500         \exp:w \exp_end_continue_f:w #3
2501     }
2502 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
2503     { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
2504 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
2505     {
2506         \exp_after:wN #1
2507         \cs:w #2 \exp_after:wN \cs_end:
2508         \exp:w \__exp_eval_register:N #3
2509     }
2510 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
2511     { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
2512 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
2513     {
2514         \exp_after:wN #1
2515         \exp_after:wN #2
2516         \exp_after:wN #3
2517         \exp:w \__exp_eval_register:N #4
2518     }
2519 \cs_new:Npn \exp_last_unbraced:NNNf #1#2#3#4
2520     {
2521         \exp_after:wN #1
2522         \exp_after:wN #2
2523         \exp_after:wN #3
2524         \exp:w \exp_end_continue_f:w #4
2525     }
2526 \cs_new:Npn \exp_last_unbraced:Nno { \:::n \:::o_unbraced \::: }
2527 \cs_new:Npn \exp_last_unbraced:Noo { \:::o \:::o_unbraced \::: }
2528 \cs_new:Npn \exp_last_unbraced:Nfo { \:::f \:::o_unbraced \::: }
2529 \cs_new:Npn \exp_last_unbraced:NnNo { \:::n \:::N \:::o_unbraced \::: }
2530 \cs_new:Npn \exp_last_unbraced:NNNNo #1#2#3#4#5
2531     { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 \exp_after:wN #4 #5 }
2532 \cs_new:Npn \exp_last_unbraced:NNNNf #1#2#3#4#5
2533     {
2534         \exp_after:wN #1
2535         \exp_after:wN #2

```

```

2536     \exp_after:wN #3
2537     \exp_after:wN #4
2538     \exp:w \exp_end_continue_f:w #5
2539   }
2540 \cs_new_protected:Npn \exp_last_unbraced:Nx { ::x_unbraced :: }

(End definition for \exp_last_unbraced:Noo and others. These functions are documented on page 33.)
```

If #2 is a single token then this can be implemented as

```

\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
  { \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }
```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```

2541 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
2542   { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
2543 \cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
2544   { \exp_after:wN #3 #2 #1 }
```

(End definition for \exp_last_two_unbraced:Noo and __exp_last_two_unbraced:noN. This function is documented on page 33.)

6.4 Preventing expansion

__kernel_exp_not:w At the kernel level, we need the primitive behaviour to allow expansion *before* the brace group.

```
2545 \cs_new_eq:NN \__kernel_exp_not:w \tex_unexpanded:D
```

(End definition for __kernel_exp_not:w.)

```

\exp_not:c All these except \exp_not:c call the kernel-internal \__kernel_exp_not:w namely
\exp_not:o \tex_unexpanded:D.
\exp_not:e 2546 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
\exp_not:f 2547 \cs_new:Npn \exp_not:o #1 { \__kernel_exp_not:w \exp_after:wN {#1} }
\exp_not:v 2548 \cs_if_exist:NTF \tex_expanded:D
\exp_not:v 2549   {
2550     \cs_new:Npn \exp_not:e #1
2551       { \__kernel_exp_not:w \tex_expanded:D { {#1} } }
2552   }
2553   {
2554     \cs_new:Npn \exp_not:e
2555       { \__kernel_exp_not:w \exp_args:N \prg_do_nothing: }
2556   }
2557 \cs_new:Npn \exp_not:f #1
2558   { \__kernel_exp_not:w \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
2559 \cs_new:Npn \exp_not:V #1
2560   {
2561     \__kernel_exp_not:w \exp_after:wN
2562       { \exp:w \__exp_eval_register:N #1 }
2563   }
2564 \cs_new:Npn \exp_not:v #1
2565   {
2566     \__kernel_exp_not:w \exp_after:wN
2567       { \exp:w \__exp_eval_register:c {#1} }
2568   }
```

(End definition for `\exp_not:c` and others. These functions are documented on page 34.)

6.5 Controlled expansion

```
2569 \exp:w
2570 \exp_end:
2571 \exp_end_continue_f:w
2572 \exp_end_continue_f:nw
```

To trigger a sequence of “arbitrarily” many expansions we need a method to invoke TeX’s expansion mechanism in such a way that (a) we are able to stop it in a controlled manner and (b) the result of what triggered the expansion in the first place is null, i.e., that we do not get any unwanted side effects. There aren’t that many possibilities in TeX; in fact the one explained below might well be the only one (as normally the result of expansion is not null).

The trick here is to make use of the fact that `\tex_roman numeral:D` expands the tokens following it when looking for a number and that its expansion is null if that number turns out to be zero or negative. So we use that to start the expansion sequence: `\exp:w` is set equal to `\tex_roman numeral:D` in `l3basics`. To stop the expansion sequence in a controlled way all we need to provide is a constant integer zero as part of expanded tokens. As this is an integer constant it immediately stops `\tex_roman numeral:D`’s search for a number. Again, the definition of `\exp_end:` as the integer constant zero is in `l3basics`. (Note that according to our specification all tokens we expand initiated by `\exp:w` are supposed to be expandable (as well as their replacement text in the expansion) so we will not encounter a “number” that actually result in a roman numeral being generated. Or if we do then the programmer made a mistake.)

If on the other hand we want to stop the initial expansion sequence but continue with an `f`-type expansion we provide the alphabetic constant ‘`^^@`’ that also represents 0 but this time TeX’s syntax for a `<number>` continues searching for an optional space (and it continues expansion doing that) — see TeXbook page 269 for details.

```
2569 \group_begin:
2570   \tex_catcode:D '^^@ = 13
2571   \cs_new_protected:Npn \exp_end_continue_f:w { '^^@ }
```

If the above definition ever appears outside its proper context the active character `^^@` will be executed so we turn this into an error. The test for existence covers the (unlikely) case that some other code has already defined `^^@`: this is true for example for `xmldtex.tex`.

```
2572 \if_cs_exist:N ^^@
2573 \else:
2574   \cs_new:Npn ^^@
2575   { \__kernel_msg_expandable_error:nn { kernel } { bad-exp-end-f } }
2576 \fi:
```

The same but grabbing an argument to remove spaces and braces.

```
2577 \cs_new:Npn \exp_end_continue_f:nw #1 { '^^@ #1 }
2578 \group_end:
```

(End definition for `\exp:w` and others. These functions are documented on page 36.)

6.6 Emulating e-type expansion

When the `\expanded` primitive is available it is used to implement `e`-type expansion; otherwise we emulate it.

```
2579 \cs_if_exist:NF \tex_expanded:D
2580   {
```

`_exp_e:nn` Repeatedly expand tokens, keeping track of fully-expanded tokens in the second argument to `_exp_e:nn`; this function eventually calls `_exp_e_end:nn` to leave `\exp_end:` in the input stream, followed by the result of the expansion. There are many special cases: spaces, brace groups, `\noexpand`, `\unexpanded`, `\the`, `\primitive`. While we use brace tricks `\if_false: { \fi:`, the expansion of this function is always triggered by `\exp:w` so brace balance is eventually restored after that is hit with a single step of expansion. Otherwise we could not nest e-type expansions within each other.

```

2581     \cs_new:Npn \_exp_e:nn #1
2582     {
2583         \if_false: { \fi:
2584             \tl_if_head_is_N_type:nTF {#1}
2585             { \_exp_e:N }
2586             {
2587                 \tl_if_head_is_group:nTF {#1}
2588                 { \_exp_e_group:n }
2589                 {
2590                     \tl_if_empty:nTF {#1}
2591                     { \exp_after:wN \_exp_e_end:nn }
2592                     { \exp_after:wN \_exp_e_space:nn }
2593                     \exp_after:wN { \if_false: } \fi:
2594                 }
2595             }
2596             #1
2597         }
2598     }
2599     \cs_new:Npn \_exp_e_end:nn #1#2 { \exp_end: #2 }

```

(End definition for `_exp_e:nn` and `_exp_e_end:nn`.)

`_exp_e_space:nn` For an explicit space character, remove it by f-expansion and put it in the (future) output.

```

2600     \cs_new:Npn \_exp_e_space:nn #1#2
2601         { \exp_args:Nf \_exp_e:nn {#1} { #2 ~ } }

```

(End definition for `_exp_e_space:nn`.)

`_exp_e_group:n` For a group, expand its contents, wrap it in two pairs of braces, and call `_exp_e_put:nn`. This function places the first item (the double-brace wrapped result) into the output. Importantly, `\tl_head:n` works even if the input contains quarks.

```

2602     \cs_new:Npn \_exp_e_group:n #1
2603     {
2604         \exp_after:wN \_exp_e_put:nn
2605         \exp_after:wN { \exp_after:wN { \exp_after:wN {
2606             \exp:w \if_false: } \fi: \_exp_e:nn {#1} { } } }
2607     }
2608     \cs_new:Npn \_exp_e_put:nn #1
2609     {
2610         \exp_args:NNo \exp_args:No \_exp_e_put:nnn
2611         { \tl_head:n {#1} } {#1}
2612     }
2613     \cs_new:Npn \_exp_e_put:nnn #1#2#3
2614         { \exp_args:No \_exp_e:nn { \use_none:n #2 } { #3 #1 } }

```

(End definition for `_exp_e_group:n`, `_exp_e_put:nn`, and `_exp_e_put:nnn`.)

```

\__exp_e:N
\__exp_e:Nnn
\__exp_e_protected:Nnn
\__exp_e_expandable:Nnn

```

For an N-type token, call `__exp_e:Nnn` with arguments the *first token*, the remaining tokens to expand and what's already been expanded. If the *first token* is non-expandable, including `\protected` (`\long` or not) macros, it is put in the result by `__exp_e_protected:Nnn`. The four special primitives `\unexpanded`, `\noexpand`, `\the`, `\primitive` are detected; otherwise the token is expanded by `__exp_e_expandable:Nnn`.

```

2615      \cs_new:Npn \__exp_e:N #1
2616      {
2617          \exp_after:wN \__exp_e:Nnn
2618          \exp_after:wN #1
2619          \exp_after:wN { \if_false: } \fi:
2620      }
2621      \cs_new:Npn \__exp_e:Nnn #1
2622      {
2623          \if_case:w
2624              \exp_after:wN \if_meaning:w \exp_not:N #1 #1 1 ~ \fi:
2625              \token_if_protected_macro:NT #1 { 1 ~ }
2626              \token_if_protected_long_macro:NT #1 { 1 ~ }
2627              \if_meaning:w \exp_not:n #1 2 ~ \fi:
2628              \if_meaning:w \exp_not:N #1 3 ~ \fi:
2629              \if_meaning:w \tex_the:D #1 4 ~ \fi:
2630              \if_meaning:w \tex_primitive:D #1 5 ~ \fi:
2631              0 ~
2632              \exp_after:wN \__exp_e_expandable:Nnn
2633              \or: \exp_after:wN \__exp_e_protected:Nnn
2634              \or: \exp_after:wN \__exp_e_unexpanded:Nnn
2635              \or: \exp_after:wN \__exp_e_noexpand:Nnn
2636              \or: \exp_after:wN \__exp_e_the:Nnn
2637              \or: \exp_after:wN \__exp_e_primitive:Nnn
2638              \fi:
2639              #1
2640      }
2641      \cs_new:Npn \__exp_e_protected:Nnn #1#2#3
2642      { \__exp_e:nn {#2} { #3 #1 } }
2643      \cs_new:Npn \__exp_e_expandable:Nnn #1#2
2644      { \exp_args:No \__exp_e:nn { #1 #2 } }

```

(End definition for `__exp_e:N` and others.)

```

\__exp_e_primitive:Nnn
\__exp_e_primitive_aux:NNw
\__exp_e_primitive_aux:NNnn
  \__exp_e_primitive_other:NNnn
  \__exp_e_primitive_other_aux:NNnn

```

We don't try hard to make sensible error recovery since the error recovery of `\tex_primitive:D` when followed by something else than a primitive depends on the engine. The only valid case is when what follows is N-type. Then distinguish special primitives `\unexpanded`, `\noexpand`, `\the`, `\primitive` from other primitives. In the “other” case, the only reasonable way to check if the primitive that follows `\tex_primitive:D` is expandable is to expand and compare the before-expansion and after-expansion results. If they coincide then probably the primitive is non-expandable and should be put in the output together with `\tex_primitive:D` (one can cook up contrived counter-examples where the true `\expanded` would have an infinite loop), and otherwise one should continue expanding.

```

2645      \cs_new:Npn \__exp_e_primitive:Nnn #1#2
2646      {
2647          \if_false: { \fi:
2648              \tl_if_head_is_N_type:nTF {#2}
2649              { \__exp_e_primitive_aux:NNw #1 }

```

```

2650      {
2651          \__kernel_msg_expandable_error:nnn { kernel } { e-type }
2652              { Missing-primitive-name }
2653              \__exp_e_primitive_aux:NNw #1 \c_empty_tl
2654      }
2655      #2
2656  }
2657 }
2658 \cs_new:Npn \__exp_e_primitive_aux:NNw #1#2
2659 {
2660     \exp_after:wN \__exp_e_primitive_aux:NNnn
2661     \exp_after:wN #1
2662     \exp_after:wN #2
2663     \exp_after:wN { \if_false: } \fi:
2664 }
2665 \cs_new:Npn \__exp_e_primitive_aux:NNnn #1#2
2666 {
2667     \exp_args:Nf \str_case_e:nnTF { \cs_to_str:N #2 }
2668     {
2669         { unexpanded } { \__exp_e_unexpanded:Nnn \exp_not:n }
2670         { noexpand } { \__exp_e_noexpand:Nnn \exp_not:N }
2671         { the } { \__exp_e_the:Nnn \tex_the:D }
2672         {
2673             \sys_if_engine_xetex:T { pdf }
2674             \sys_if_engine_luatex:T { pdf }
2675             primitive
2676             } { \__exp_e_primitive:Nnn #1 }
2677         }
2678         { \__exp_e_primitive_other:NNnn #1 #2 }
2679     }
2680 \cs_new:Npn \__exp_e_primitive_other:NNnn #1#2#3
2681 {
2682     \exp_args:No \__exp_e_primitive_other_aux:nNNnn
2683     { #1 #2 #3 }
2684     #1 #2 {#3}
2685 }
2686 \cs_new:Npn \__exp_e_primitive_other_aux:nNNnn #1#2#3#4#5
2687 {
2688     \str_if_eq:nnTF {#1} { #2 #3 #4 }
2689     { \__exp_e:nn {#4} { #5 #2 #3 } }
2690     { \__exp_e:nn {#1} {#5} }
2691 }

```

(End definition for `__exp_e_primitive:Nnn` and others.)

`__exp_e_noexpand:Nnn` The `\noexpand` primitive has no effect when followed by a token that is not N-type; otherwise `__exp_e_put:nn` can grab the next token and put it in the result unchanged.

```

2692     \cs_new:Npn \__exp_e_noexpand:Nnn #1#2
2693     {
2694         \tl_if_head_is_N_type:nTF {#2}
2695             { \__exp_e_put:nn } { \__exp_e:nn } {#2}
2696     }

```

(End definition for `__exp_e_noexpand:Nnn`.)

```
\_\_exp_e_unexpanded:Nnn
\_\_exp_e_unexpanded:nn
\_\_exp_e_unexpanded:nN
\_\_exp_e_unexpanded:N
```

The `\unexpanded` primitive expands and ignores any space, `\scan_stop:`, or token affected by `\exp_not:N`, then expects a brace group. Since we only support brace-balanced token lists it is impossible to support the case where the argument of `\unexpanded` starts with an implicit brace. Even though we want to expand and ignore spaces we cannot blindly f-expand because tokens affected by `\exp_not:N` should be discarded without being expanded further.

As usual distinguish four cases: brace group (the normal case, where we just put the item in the result), space (just f-expand to remove the space), empty (an error), or N-type `<token>`. In the last case call `__exp_e_unexpanded:nN` triggered by an f-expansion. Having a non-expandable `<token>` after `\unexpanded` is an error (we recover by passing {} to `\unexpanded`; this is different from TeX because the error recovery of `\unexpanded` changes the balance of braces), unless that `<token>` is `\scan_stop:` or a space (recall that we don't implement the case of an implicit begin-group token). An expandable `<token>` is instead expanded, unless it is `\noexpand`. The latter primitive can be followed by an expandable N-type token (removed), by a non-expandable one (kept and later causing an error), by a space (removed by f-expansion), or by a brace group or nothing (later causing an error).

```
2697     \cs_new:Npn \_\_exp_e_unexpanded:Nnn #1 { \_\_exp_e_unexpanded:nn }
2698     \cs_new:Npn \_\_exp_e_unexpanded:nn #1
2699     {
2700         \tl_if_head_is_N_type:nTF {#1}
2701         {
2702             \exp_args:Nf \_\_exp_e_unexpanded:nn
2703             { \_\_exp_e_unexpanded:nN {#1} #1 }
2704         }
2705         {
2706             \tl_if_head_is_group:nTF {#1}
2707             { \_\_exp_e_put:nn }
2708             {
2709                 \tl_if_empty:nTF {#1}
2710                 {
2711                     \kernel_msg_expandable_error:nnn
2712                     { kernel } { e-type }
2713                     { \unexpanded missing~brace }
2714                     \_\_exp_e_end:nn
2715                 }
2716                 { \exp_args:Nf \_\_exp_e_unexpanded:nn }
2717             }
2718             {#1}
2719         }
2720     }
2721     \cs_new:Npn \_\_exp_e_unexpanded:nN #1#2
2722     {
2723         \exp_after:wN \if_meaning:w \exp_not:N #2 #2
2724         \exp_after:wN \use_i:nn
2725         \else:
2726             \exp_after:wN \use_ii:nn
2727         \fi:
2728     }
2729     \token_if_eq_catcode:NNTF #2 \c_space_token
2730     { \exp_stop_f: }
2731     {
```

```

2732     \token_if_eq_meaning:NNTF #2 \scan_stop:
2733     { \exp_stop_f: }
2734     {
2735         \_kernel_msg_expandable_error:nnn
2736         { kernel } { e-type }
2737         { \unexpanded missing~brace }
2738         { }
2739     }
2740 }
2741 }
2742 {
2743     \token_if_eq_meaning:NNTF #2 \exp_not:N
2744     {
2745         \exp_args:No \tl_if_head_is_N_type:nT { \use_none:n #1 }
2746         { \_exp_e_unexpanded:N }
2747     }
2748     { \exp_after:wN \exp_stop_f: #2 }
2749 }
2750 }
2751 \cs_new:Npn \_exp_e_unexpanded:N #1
2752 {
2753     \exp_after:wN \if_meaning:w \exp_not:N #1 #1 \else:
2754     \exp_after:wN \use_i:nn
2755     \fi:
2756     \exp_stop_f: #1
2757 }

```

(End definition for `_exp_e_unexpanded:Nnn` and others.)

`_exp_e_the:Nnn` Finally implement `\the`. Followed by anything other than an N-type `<token>` this causes an error (we just let TeX make one), otherwise we test the `<token>`. If the `<token>` is expandable, expand it. Otherwise it could be any kind of register, or things like `\numexpr`, so there is no way to deal with all cases. Thankfully, only `\toks` data needs to be protected from expansion since everything else gives a string of characters. If the `<token>` is `\toks` we find a number and unpack using the `the_toks` functions. If it is a token register we unpack it in a brace group and call `_exp_e_put:nn` to move it to the result. Otherwise we unpack and continue expanding (useless but safe) since it is basically impossible to have a handle on where the result of `\the` ends.

```

2758     \cs_new:Npn \_exp_e_the:Nnn #1#2
2759     {
2760         \tl_if_head_is_N_type:nTF {#2}
2761         { \if_false: { \fi: \_exp_e_the:N #2 } }
2762         { \exp_args:No \_exp_e:nn { \tex_the:D #2 } }
2763     }
2764     \cs_new:Npn \_exp_e_the:N #1
2765     {
2766         \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2767         \exp_after:wN \use_i:nn
2768         \else:
2769         \exp_after:wN \use_ii:nn
2770         \fi:
2771     {
2772         \if_meaning:w \tex_toks:D #1
2773         \exp_after:wN \_exp_e_the_toks:wnn \int_value:w

```

```

2774   \exp_after:wN \__exp_e_the_toks:n
2775   \exp_after:wN { \int_value:w \if_false: } \fi:
2776 \else:
2777   \__exp_e_if_toks_register:N#1
2778   { \exp_after:wN \__exp_e_the_toks_reg:N }
2779   {
2780     \exp_after:wN \__exp_e:nn \exp_after:wN {
2781       \tex_the:D \if_false: } \fi:
2782     }
2783   \exp_after:wN #1
2784 \fi:
2785 }
2786 {
2787   \exp_after:wN \__exp_e_the:Nnn \exp_after:wN ?
2788   \exp_after:wN { \exp:w \if_false: } \fi:
2789   \exp_after:wN \exp_end: #1
2790 }
2791 }
2792 \cs_new:Npn \__exp_e_the_toks_reg:N #1
2793 {
2794   \exp_after:wN \__exp_e_put:nn \exp_after:wN {
2795   \exp_after:wN {
2796     \tex_the:D \if_false: } \fi: #1 }
2797 }

```

(End definition for `__exp_e_the:Nnn`, `__exp_e_the:N`, and `__exp_e_the_toks_reg:N`.)

`__exp_e_the_toks:wnn`
`__exp_e_the_toks:n`
`__exp_e_the_toks:N`

The calling function has applied `\int_value:w` so we collect digits with `__exp_e_the_toks:n` (which gets the token list as an argument) and `__exp_e_the_toks:N` (which gets the first token in case it is N-type). The digits are themselves collected into an `\int_value:w` argument to `__exp_e_the_toks:wnn`. Then that function unpacks the `\toks<number>` into the result. We include ? because `__exp_e_put:nnn` removes one item from its second argument. Note that our approach is rather crude: in cases like `\the\toks12~34` the first `\int_value:w` removes the space and we will incorrectly unpack the `\the\toks1234`.

```

2798   \cs_new:Npn \__exp_e_the_toks:wnn #1; #2
2799   {
2800     \exp_args:No \__exp_e_put:nnn
2801     { \tex_the:D \tex_toks:D #1 } { ? #2 }
2802   }
2803 \cs_new:Npn \__exp_e_the_toks:n #1
2804   {
2805     \tl_if_head_is_N_type:nTF {#1}
2806     { \exp_after:wN \__exp_e_the_toks:N \if_false: { \fi: #1 } }
2807     { ; {#1} }
2808   }
2809 \cs_new:Npn \__exp_e_the_toks:N #1
2810   {
2811     \if_int_compare:w 10 < 9 \token_to_str:N #1 \exp_stop_f:
2812     \exp_after:wN \use_i:nn
2813   \else:
2814     \exp_after:wN \use_ii:nn
2815   \fi:
2816   }

```

```

2817      #1
2818      \exp_after:wN \__exp_e_the_toks:n
2819      \exp_after:wN { \if_false: } \fi:
2820    }
2821    {
2822      \exp_after:wN ;
2823      \exp_after:wN { \if_false: } \fi: #1
2824    }
2825  }

```

(End definition for `__exp_e_the_toks:wnn`, `__exp_e_the_toks:n`, and `__exp_e_the_toks:N`.)

```

\__exp_e_if_toks_register:NTF
\__exp_e_the_XeTeXinterchartoks:
\__exp_e_the_errhelp:
\__exp_e_the_everycr:
\__exp_e_the_everydisplay:
  \__exp_e_the_everyeof:
  \__exp_e_the_everyhbox:
  \__exp_e_the_everyjob:
  \__exp_e_the_everymath:
  \__exp_e_the_everypar:
  \__exp_e_the_everyvbox:
    \__exp_e_the_output:
\__exp_e_the_pdfpageattr:
  \__exp_e_the_pdfpageresources:
\__exp_e_the_pdfpagesattr:
  \__exp_e_the_pdpkmode:

```

```

2826      \prg_new_conditional:Npnn \__exp_e_if_toks_register:N #1 { TF }
2827      {
2828        \token_if_toks_register:NTF #1 { \prg_return_true: }
2829        {
2830          \cs_if_exist:cTF
2831          {
2832            \__exp_e_the_
2833            \exp_after:wN \cs_to_str:N
2834            \token_to_meaning:N #1
2835            :
2836          } { \prg_return_true: } { \prg_return_false: }
2837        }
2838      }
2839      \cs_new_eq:NN \__exp_e_the_XeTeXinterchartoks: ?
2840      \cs_new_eq:NN \__exp_e_the_errhelp: ?
2841      \cs_new_eq:NN \__exp_e_the_everycr: ?
2842      \cs_new_eq:NN \__exp_e_the_everydisplay: ?
2843      \cs_new_eq:NN \__exp_e_the_everyeof: ?
2844      \cs_new_eq:NN \__exp_e_the_everyhbox: ?
2845      \cs_new_eq:NN \__exp_e_the_everyjob: ?
2846      \cs_new_eq:NN \__exp_e_the_everymath: ?
2847      \cs_new_eq:NN \__exp_e_the_everypar: ?
2848      \cs_new_eq:NN \__exp_e_the_everyvbox: ?
2849      \cs_new_eq:NN \__exp_e_the_output: ?
2850      \cs_new_eq:NN \__exp_e_the_pdfpageattr: ?
2851      \cs_new_eq:NN \__exp_e_the_pdfpageresources: ?
2852      \cs_new_eq:NN \__exp_e_the_pdfpagesattr: ?
2853      \cs_new_eq:NN \__exp_e_the_pdpkmode: ?

```

(End definition for `__exp_e_if_toks_register:NTF` and others.)

We are done emulating e-type argument expansion when `\expanded` is unavailable.

```
2854  }
```

6.7 Defining function variants

```
2855  ⟨@=cs⟩
```

```

\s__cs_mark Internal scan marks. No l3quark yet, so do things by hand.
\s__cs_stop
2856 \cs_new_eq:NN \s__cs_mark \scan_stop:
2857 \cs_new_eq:NN \s__cs_stop \scan_stop:

(End definition for \s__cs_mark and \s__cs_stop.)

\q__cs_recursion_stop Internal recursion quarks. No l3quark yet, so do things by hand.
2858 \cs_new:Npn \q__cs_recursion_stop { \q__cs_recursion_stop }

(End definition for \q__cs_recursion_stop.)

\_cs_use_none_delimit_by_s_stop:w Internal scan marks.
\_cs_use_i_delimit_by_s_stop:nw
\_cs_use_none_delimit_by_q_recursion_stop:w
2859 \cs_new:Npn \_cs_use_none_delimit_by_s_stop:w #1 \s__cs_stop { }
2860 \cs_new:Npn \_cs_use_i_delimit_by_s_stop:nw #1 #2 \s__cs_stop {#1}
2861 \cs_new:Npn \_cs_use_none_delimit_by_q_recursion_stop:w
2862 #1 \q__cs_recursion_stop { }

(End definition for \_cs_use_none_delimit_by_s_stop:w, \_cs_use_i_delimit_by_s_stop:nw, and
\_cs_use_none_delimit_by_q_recursion_stop:w.)

\cs_generate_variant:Nn #1 : Base form of a function; e.g., \tl_set:Nn
\cs_generate_variant:cn #2 : One or more variant argument specifiers; e.g., {Nx,c,cx}

After making sure that the base form exists, test whether it is protected or not and
define \_cs_tmp:w as either \cs_new:Npx or \cs_new_protected:Npx, which is then
used to define all the variants (except those involving x-expansion, always protected).
Split up the original base function only once, to grab its name and signature. Then
we wish to iterate through the comma list of variant argument specifiers, which we first
convert to a string: the reason is explained later.

2863 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
2864 {
2865   \_cs_generate_variant:N #1
2866   \use:x
2867   {
2868     \_cs_generate_variant:nnNN
2869     \cs_split_function:N #1
2870     \exp_not:N #1
2871     \tl_to_str:n {#2} ,
2872     \exp_not:N \scan_stop: ,
2873     \exp_not:N \q__cs_recursion_stop
2874   }
2875 }
2876 \cs_new_protected:Npn \cs_generate_variant:cn
2877 { \exp_args:Nc \cs_generate_variant:Nn }

(End definition for \cs_generate_variant:Nn. This function is documented on page 27.)

```

_cs_generate_variant:N
_cs_generate_variant:ww
_cs_generate_variant:wwNw

The goal here is to pick up protected parent functions. There are four cases: the parent function can be a primitive or a macro, and can be expandable or not. For non-expandable primitives, all variants should be protected; skipping the \else: branch is safe because non-expandable primitives cannot be TeX conditionals.

The other case where variants should be protected is when the parent function is a protected macro: then `protected` appears in the meaning before the first occurrence of `macro`. The `ww` auxiliary removes everything in the meaning string after the first `ma`. We use `ma` rather than the full `macro` because the meaning of the `\firstmark` primitive (and

four others) can contain an arbitrary string after a leading `firstmark:`. Then, look for `pr` in the part we extracted: no need to look for anything longer: the only strings we can have are an empty string, `\long`, `\protected`, `\protected\long`, `\first`, `\top`, `\bot`, `\splittop`, or `\splitbot`, with `\` replaced by the appropriate escape character. If `pr` appears in the part before `ma`, the first `\s__cs_mark` is taken as an argument of the `wwNw` auxiliary, and #3 is `\cs_new_protected:Npx`, otherwise it is `\cs_new:Npx`.

```

2878 \cs_new_protected:Npx \__cs_generate_variant:N #1
2879 {
2880     \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
2881     \exp_not:N \exp_not:N #1 #1
2882     \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx
2883 \exp_not:N \else:
2884     \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
2885     \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
2886     \s__cs_mark
2887     \s__cs_mark \cs_new_protected:Npx
2888     \tl_to_str:n { pr }
2889     \s__cs_mark \cs_new:Npx
2890     \s__cs_stop
2891     \exp_not:N \fi:
2892 }
2893 \exp_last_unbraced:NNNNo
2894 \cs_new_protected:Npn \__cs_generate_variant:ww
2895     #1 { \tl_to_str:n { ma } } #2 \s__cs_mark
2896     { \__cs_generate_variant:wwNw #1 }
2897 \exp_last_unbraced:NNNNo
2898 \cs_new_protected:Npn \__cs_generate_variant:wwNw
2899     #1 { \tl_to_str:n { pr } } #2 \s__cs_mark #3 #4 \s__cs_stop
2900     { \cs_set_eq:NN \__cs_tmp:w #3 }
```

(End definition for `__cs_generate_variant:N`, `__cs_generate_variant:ww`, and `__cs_generate_variant:wwNw`.)

`__cs_generate_variant:nnNN` #1 : Base name.
#2 : Base signature.
#3 : Boolean.
#4 : Base function.

If the boolean is `\c_false_bool`, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as #4 for efficiency.

```

2901 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
2902 {
2903     \if_meaning:w \c_false_bool #3
2904     \_kernel_msg_error:nnx { kernel } { missing-colon }
2905     { \token_to_str:c {#1} }
2906     \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
2907     \fi:
2908     \__cs_generate_variant:Nnnw #4 {#1}{#2}
2909 }
```

(End definition for `__cs_generate_variant:nnNN`.)

`__cs_generate_variant:Nnnw` #1 : Base function.
#2 : Base name.

#3 : Base signature.
#4 : Beginning of variant signature.

First check whether to terminate the loop over variant forms. Then, for each variant form, construct a new function name using the original base name, the variant signature consisting of l letters and the last $k - l$ letters of the base signature (of length k). For example, for a base function `\prop_put:Nnn` which needs a cV variant form, we want the new signature to be `cVn`.

There are further subtleties:

- In `\cs_generate_variant:Nn \foo:nnTF {xxTF}`, we must define `\foo:xxTF` using `\exp_args:Nxx`, rather than a hypothetical `\exp_args:NxxTF`. Thus, we wish to trim a common trailing part from the base signature and the variant signature.
- In `\cs_generate_variant:Nn \foo:on {ox}`, the function `\foo:ox` must be defined using `\exp_args:Nnx`, not `\exp_args:Nox`, to avoid double o expansion.
- Lastly, `\cs_generate_variant:Nn \foo:on {xn}` must trigger an error, because we do not have a means to replace o-expansion by x-expansion. More generally, we can only convert N to c, or convert n to V, v, o, f, x.

All this boils down to a few rules. Only n and N-type arguments can be replaced by `\cs_generate_variant:Nn`. Other argument types are allowed to be passed unchanged from the base form to the variant: in the process they are changed to n except for N and p-type arguments. A common trailing part is ignored.

We compare the base and variant signatures one character at a time within x-expansion. The result is given to `_cs_generate_variant:wwNN` (defined later) in the form `<processed variant signature> \s__cs_mark <errors> \s__cs_stop <base function> <new function>`. If all went well, `<errors>` is empty; otherwise, it is a kernel error message and some clean-up code.

Note the space after #3 and after the following brace group. Those are ignored by TeX when fetching the last argument for `_cs_generate_variant_loop:nNwN`, but can be used as a delimiter for `_cs_generate_variant_loop_end:nwwwNNnn`.

```

2910 \cs_new_protected:Npn \_cs_generate_variant:Nnnw #1#2#3#4 ,
2911   {
2912     \if_meaning:w \scan_stop: #4
2913     \exp_after:wN \_cs_use_none_delimit_by_q_recursion_stop:w
2914   \fi:
2915   \use:x
2916   {
2917     \exp_not:N \_cs_generate_variant:wwNN
2918     \_cs_generate_variant_loop:nNwN { }
2919     #4
2920     \_cs_generate_variant_loop_end:nwwwNNnn
2921     \s__cs_mark
2922     #3 ~
2923     { ~ { } \fi: \_cs_generate_variant_loop_long:wNNnn } ~
2924     { }
2925     \s__cs_stop
2926     \exp_not:N #1 {#2} {#4}
2927   }
2928   \_cs_generate_variant:Nnnw #1 {#2} {#3}
2929 }
```

(End definition for `__cs_generate_variant:Nnnw`.)

```
\__cs_generate_variant_loop:nNwN
\__cs_generate_variant_loop_base:N
\__cs_generate_variant_loop_same:w
\__cs_generate_variant_loop_end:nwwwNNnn
\__cs_generate_variant_loop_long:wNNnn
\__cs_generate_variant_loop_invalid>NNwNNnn
\__cs_generate_variant_loop_special>NNwNNnn
```

- #1 : Last few consecutive letters common between the base and variant (more precisely, `__cs_generate_variant_same:N <letter>` for each letter).
- #2 : Next variant letter.
- #3 : Remainder of variant form.
- #4 : Next base letter.

The first argument is populated by `__cs_generate_variant_loop_same:w` when a variant letter and a base letter match. It is flushed into the input stream whenever the two letters are different: if the loop ends before, the argument is dropped, which means that trailing common letters are ignored.

The case where the two letters are different is only allowed if the base is `N` and the variant is `c`, or when the base is `n` and the variant is `o`, `V`, `v`, `f` or `x`. Otherwise, call `__cs_generate_variant_loop_invalid>NNwNNnn` to remove the end of the loop, get arguments at the end of the loop, and place an appropriate error message as a second argument of `__cs_generate_variant:wwNN`. If the letters are distinct and the base letter is indeed `n` or `N`, leave in the input stream whatever argument #1 was collected, and the next variant letter #2, then loop by calling `__cs_generate_variant_loop:nNwN`.

The loop can stop in three ways.

- If the end of the variant form is encountered first, #2 is `__cs_generate_variant_loop_end:nwwwNNnn` (expanded by the conditional `\if:w`), which inserts some tokens to end the conditional; grabs the `<base name>` as #7, the `<variant signature>` #8, the `<next base letter>` #1 and the part #3 of the base signature that wasn't read yet; and combines those into the `<new function>` to be defined.
- If the end of the base form is encountered first, #4 is `\fi:` which ends the conditional (with an empty expansion), followed by `__cs_generate_variant_loop_long:wNNnn`, which places an error as the second argument of `__cs_generate_variant:wwNN`.
- The loop can be interrupted early if the requested expansion is unavailable, namely when the variant and base letters differ and the base is not the right one (`n` or `N` to support the variant). In that case too an error is placed as the second argument of `__cs_generate_variant:wwNN`.

Note that if the variant form has the same length as the base form, #2 is as described in the first point, and #4 as described in the second point above. The `__cs_generate_variant_loop_end:nwwwNNnn` breaking function takes the empty brace group in #4 as its first argument: this empty brace group produces the correct signature for the full variant.

```
2930 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \s__cs_mark #4
2931 {
2932     \if:w #2 #4
2933     \exp_after:wN \__cs_generate_variant_loop_same:w
2934     \else:
2935         \if:w #4 \__cs_generate_variant_loop_base:N #2 \else:
2936             \if:w 0
2937                 \if:w N #4 \else: \if:w n #4 \else: 1 \fi: \fi:
2938                 \if:w \scan_stop: \__cs_generate_variant_loop_base:N #2 1 \fi:
2939                 0
2940             \__cs_generate_variant_loop_special>NNwNNnn #4#2
```

```

2941           \else:
2942             \_\_cs_generate_variant_loop_invalid:NNwNNnn #4#2
2943           \fi:
2944           \fi:
2945           \#1
2946           \prg_do_nothing:
2947           \#2
2948             \_\_cs_generate_variant_loop:nNwN { } #3 \s\_\_cs\_mark
2949           }
2950 \cs_new:Npn \_\_cs_generate_variant_loop_base:N #1
2951   {
2952     \if:w c #1 N \else:
2953       \if:w o #1 n \else:
2954         \if:w V #1 n \else:
2955           \if:w v #1 n \else:
2956             \if:w f #1 n \else:
2957               \if:w e #1 n \else:
2958                 \if:w x #1 n \else:
2959                   \if:w n #1 n \else:
2960                     \if:w N #1 N \else:
2961                       \scan_stop:
2962                     \fi:
2963                   \fi:
2964                 \fi:
2965               \fi:
2966             \fi:
2967           \fi:
2968         \fi:
2969       \fi:
2970     \fi:
2971   \fi:
2972 \cs_new:Npn \_\_cs_generate_variant_loop_same:w
2973   #1 \prg_do_nothing: #2#3#4
2974   { #3 { #1 \_\_cs_generate_variant_same:N #2 } }
2975 \cs_new:Npn \_\_cs_generate_variant_loop_end:nwwwNNnn
2976   #1#2 \s\_\_cs\_mark #3 ~ #4 \s\_\_cs\_stop #5#6#7#8
2977   {
2978     \scan_stop: \scan_stop: \fi:
2979     \s\_\_cs\_mark \s\_\_cs\_stop
2980     \exp_not:N #6
2981     \exp_not:c { #7 : #8 #1 #3 }
2982   }
2983 \cs_new:Npn \_\_cs_generate_variant_loop_long:wNNnn #1 \s\_\_cs\_stop #2#3#4#5
2984   {
2985     \exp_not:n
2986     {
2987       \s\_\_cs\_mark
2988       \_\_kernel_msg_error:nnxx { kernel } { variant-too-long }
2989       {#5} { \token_to_str:N #3 }
2990       \use_none:nnn
2991       \s\_\_cs\_stop
2992       #3
2993       #3

```

```

2995     }
2996   }
2997 \cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
2998   #1#2 \fi: \fi: \fi: #3 \s__cs_stop #4#5#6#7
2999   {
3000     \fi: \fi: \fi:
3001     \exp_not:n
3002     {
3003       \s__cs_mark
3004       \__kernel_msg_error:nnxxxx { kernel } { invalid-variant }
3005       {#7} { \token_to_str:N #5 } {#1} {#2}
3006       \use_none:nnn
3007       \s__cs_stop
3008       #5
3009       #5
3010     }
3011   }
3012 \cs_new:Npn \__cs_generate_variant_loop_special:NNwNNnn
3013   #1#2#3 \s__cs_stop #4#5#6#7
3014   {
3015     #3 \s__cs_stop #4 #5 {#6} {#7}
3016     \exp_not:n
3017     {
3018       \__kernel_msg_error:nnxxxx
3019       { kernel } { deprecated-variant }
3020       {#7} { \token_to_str:N #5 } {#1} {#2}
3021     }
3022   }

```

(End definition for `__cs_generate_variant_loop:nNwN` and others.)

`__cs_generate_variant_same:N`

When the base and variant letters are identical, don't do any expansion. For most argument types, we can use the `n`-type no-expansion, but the `N` and `p` types require a slightly different behaviour with respect to braces. For `V`-type this function could output `N` to avoid adding useless braces but that is not a problem.

```

3023 \cs_new:Npn \__cs_generate_variant_same:N #1
3024   {
3025     \if:w N #1 #1 \else:
3026       \if:w p #1 #1 \else:
3027         \token_to_str:N n
3028         \if:w n #1 \else:
3029           \__cs_generate_variant_loop_special:NNwNNnn #1#1
3030           \fi:
3031         \fi:
3032       \fi:
3033   }

```

(End definition for `__cs_generate_variant_same:N`.)

`__cs_generate_variant:wwNN`

If the variant form has already been defined, log its existence (provided `log-functions` is active). Otherwise, make sure that the `\exp_args:N #3` form is defined, and if it contains `x`, change `__cs_tmp:w` locally to `\cs_new_protected:Npx`. Then define the variant by combining the `\exp_args:N #3` variant and the base function.

```

3034 \cs_new_protected:Npn \__cs_generate_variant:wwNN

```

```

3035     #1 \s__cs_mark #2 \s__cs_stop #3#4
3036 {
3037     #2
3038     \cs_if_free:NT #4
3039     {
3040         \group_begin:
3041             \__cs_generate_internal_variant:n {#1}
3042             \__cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
3043             \group_end:
3044     }
3045 }

```

(End definition for `__cs_generate_variant:wwNN`.)

`__cs_generate_internal_variant:n` First test for the presence of `x` (this is where working with strings makes our lives easier), as the result should be protected, and the next variant to be defined using that internal variant should be protected (done by setting `__cs_tmp:w`). Then call `__cs_generate_internal_variant>NNn` with arguments `\cs_new_protected:cpn \use:x` (for protected) or `\cs_new:cpn \tex_expanded:D` (expandable) and the signature. If `p` appears in the signature, or if the function to be defined is expandable and the primitive `\expanded` is not available, or if there are more than 8 arguments, call some fall-back code that just puts the appropriate `\:::` commands. Otherwise, call `__cs_generate_internal_one_go>NNn` to construct the `\exp_args:N...` function as a macro taking up to 9 arguments and expanding them using `\use:x` or `\tex_expanded:D`.

```

3046 \cs_new_protected:Npx \__cs_generate_internal_variant:n #1
3047 {
3048     \exp_not:N \__cs_generate_internal_variant:wwNwn
3049     #1 \s__cs_mark
3050     { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx }
3051     \cs_new_protected:cpn
3052     \use:x
3053     \token_to_str:N x \s__cs_mark
3054     { }
3055     \cs_new:cpn
3056     \exp_not:N \tex_expanded:D
3057     \s__cs_stop
3058     {#1}
3059 }
3060 \exp_last_unbraced:NNNNo
3061 \cs_new_protected:Npn \__cs_generate_internal_variant:wwNwn #1
3062     { \token_to_str:N x } #2 \s__cs_mark #3#4#5#6 \s__cs_stop #7
3063 {
3064     #3
3065     \cs_if_free:cT { exp_args:N #7 }
3066     { \__cs_generate_internal_variant>NNn #4 #5 {#7} }
3067 }
3068 \cs_set_protected:Npn \__cs_tmp:w #1
3069 {
3070     \cs_new_protected:Npn \__cs_generate_internal_variant>NNn ##1##2##3
3071     {
3072         \if_catcode:w X \use_none:nnnnnnnn ##3
3073             \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
3074             \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
3075             \prg_do_nothing: \prg_do_nothing: X

```

```

3076           \exp_after:wN \__cs_generate_internal_test:Nw \exp_after:wN ##2
3077
3078 \else:
3079   \exp_after:wN \__cs_generate_internal_test_aux:w \exp_after:wN #1
3080 \fi:
3081 ##3
3082 \s__cs_mark
3083 {
3084   \use:x
3085   {
3086     ##1 { \exp_args:N ##3 }
3087     { \__cs_generate_internal_variant_loop:n ##3 { : \use_i:nn } }
3088   }
3089   #1
3090   \s__cs_mark
3091   { \exp_not:n { \__cs_generate_internal_one_go:NNn ##1 ##2 {##3} } }
3092   \s__cs_stop
3093 }
3094 \cs_new_protected:Npn \__cs_generate_internal_test_aux:w
3095   ##1 #1 ##2 \s__cs_mark ##3 ##4 \s__cs_stop {##3}
3096 \cs_if_exist:NTF \tex_expanded:D
3097 {
3098   \cs_new_eq:NN \__cs_generate_internal_test:Nw
3099   \__cs_generate_internal_test_aux:w
3100 }
3101 {
3102   \cs_new_protected:Npn \__cs_generate_internal_test:Nw ##1
3103   {
3104     \if_meaning:w \tex_expanded:D ##1
3105       \exp_after:wN \__cs_generate_internal_test_aux:w
3106       \exp_after:wN #1
3107     \else:
3108       \exp_after:wN \__cs_generate_internal_test_aux:w
3109     \fi:
3110   }
3111 }
3112 }
3113 \exp_args:No \__cs_tmp:w { \token_to_str:N p }
3114 \cs_new_protected:Npn \__cs_generate_internal_one_go:NNn #1#2#3
3115 {
3116   \__cs_generate_internal_loop:nwnnw
3117   { \exp_not:N ##1 } 1 . { } { }
3118   #3 { ? \__cs_generate_internal_end:w } X ;
3119   23456789 { ? \__cs_generate_internal_long:w } ;
3120   #1 #2 {##3}
3121 }
3122 \cs_new_protected:Npn \__cs_generate_internal_loop:nwnnw #1#2 . #3#4#5#6 ; #7
3123 {
3124   \use_none:n #5
3125   \use_none:n #7
3126   \cs_if_exist_use:cF { \__cs_generate_internal_#5:NN }
3127   { \__cs_generate_internal_other:NN }
3128   #5 #7
3129   #7 .

```

```

3130      { #3 #1 } { #4 ## #2 }
3131      #6 ;
3132  }
3133 \cs_new_protected:Npn \__cs_generate_internal_N:NN #1#2
3134   { \__cs_generate_internal_loop:nwnnw { \exp_not:N ###2 } }
3135 \cs_new_protected:Npn \__cs_generate_internal_c:NN #1#2
3136   { \exp_args:No \__cs_generate_internal_loop:nwnnw { \exp_not:c {###2} } }
3137 \cs_new_protected:Npn \__cs_generate_internal_n:NN #1#2
3138   { \__cs_generate_internal_loop:nwnnw { { \exp_not:n {###2} } } }
3139 \cs_new_protected:Npn \__cs_generate_internal_x:NN #1#2
3140   { \__cs_generate_internal_loop:nwnnw { {###2} } }
3141 \cs_new_protected:Npn \__cs_generate_internal_other:NN #1#2
3142  {
3143    \exp_args:No \__cs_generate_internal_loop:nwnnw
3144    {
3145      \exp_after:wN
3146      {
3147        \exp:w \exp_args:Nc \exp_after:wN \exp_end:
3148        { \exp_not:#1 } {###2}
3149      }
3150    }
3151  }
3152 \cs_new_protected:Npn \__cs_generate_internal_end:w #1 . #2#3#4 ; #5 ; #6#7#8
3153   { #6 { \exp_args:N #8 } #3 { #7 {#2} } }
3154 \cs_new_protected:Npn \__cs_generate_internal_long:w #1 N #2#3 . #4#5#6#
3155  {
3156    \exp_args:Nx \__cs_generate_internal_long:nnnNNn
3157    { \__cs_generate_internal_variant_loop:n #2 #6 { : \use_i:nn } }
3158    {#4} {#5}
3159  }
3160 \cs_new:Npn \__cs_generate_internal_long:nnnNNn #1#2#3#4 ; ; #5#6#7
3161   { #5 { \exp_args:N #7 } #3 { #6 { \exp_not:n {#1} {#2} } } }

```

This command grabs char by char outputting `\::#1` (not expanded further). We avoid tests by putting a trailing `: \use_i:nn`, which leaves `\cs_end:` and removes the looping macro. The colon is in fact also turned into `\:::` so that the required structure for `\exp_args:N...` commands is correctly terminated.

```

3162 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
3163  {
3164    \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
3165    \__cs_generate_internal_variant_loop:n
3166  }

```

(End definition for `__cs_generate_internal_variant:n` and `__cs_generate_internal_variant_loop:n`.)

```

\prg_generate_conditional_variant:Nnn
\__cs_generate_variant:nnNnn
\__cs_generate_variant:w
\__cs_generate_variant:n
\__cs_generate_variant_p_form:nnn
\__cs_generate_variant_T_form:nnn
\__cs_generate_variant_F_form:nnn
\__cs_generate_variant_TF_form:nnn
3167 \cs_new_protected:Npn \prg_generate_conditional_variant:Nnn #1
3168  {
3169    \use:x
3170    {
3171      \__cs_generate_variant:nnNnn
3172      \cs_split_function:N #1
3173    }
3174  }

```

```

3175 \cs_new_protected:Npn \__cs_generate_variant:nnNnn #1#2#3#4#5
3176 {
3177     \if_meaning:w \c_false_bool #3
3178         \_kernel_msg_error:nnx { kernel } { missing-colon }
3179         { \token_to_str:c {#1} }
3180         \__cs_use_i_delimit_by_s_stop:nw
3181     \fi:
3182     \exp_after:wN \__cs_generate_variant:w
3183     \tl_to_str:n {#5} , \scan_stop: , \q__cs_recursion_stop
3184     \__cs_use_none_delimit_by_s_stop:w \s__cs_mark {#1} {#2} {#4} \s__cs_stop
3185 }
3186 \cs_new_protected:Npn \__cs_generate_variant:w
3187     #1 , #2 \s__cs_mark #3#4#5
3188 {
3189     \if_meaning:w \scan_stop: #1 \scan_stop:
3190         \if_meaning:w \q__cs_nil #1 \q__cs_nil
3191             \use_i:nnn
3192         \fi:
3193         \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
3194     \else:
3195         \cs_if_exist_use:cTF { __cs_generate_variant_#1_form:nnn }
3196         { {#3} {#4} {#5} }
3197         {
3198             \_kernel_msg_error:nnxx
3199             { kernel } { conditional-form-unknown }
3200             {#1} { \token_to_str:c {#3 : #4} }
3201         }
3202     \fi:
3203     \__cs_generate_variant:w #2 \s__cs_mark {#3} {#4} {#5}
3204 }
3205 \cs_new_protected:Npn \__cs_generate_variant_p_form:nnn #1#2
3206     { \cs_generate_variant:cn {#1 _p : #2} }
3207 \cs_new_protected:Npn \__cs_generate_variant_T_form:nnn #1#2
3208     { \cs_generate_variant:cn {#1 : #2 T} }
3209 \cs_new_protected:Npn \__cs_generate_variant_F_form:nnn #1#2
3210     { \cs_generate_variant:cn {#1 : #2 F} }
3211 \cs_new_protected:Npn \__cs_generate_variant_TF_form:nnn #1#2
3212     { \cs_generate_variant:cn {#1 : #2 TF} }

```

(End definition for `\prg_generate_conditional_variant:Nnn` and others. This function is documented on page [107](#).)

\exp_args_generate:n This function is not used in the kernel hence we can use functions that are defined in later modules. It also does not need to be fast so use inline mappings. For each requested variant we check that there are no characters besides `NnpcofVvx`, in particular that there are no spaces. Then we just call the internal function.

```

3213 \cs_new_protected:Npn \exp_args_generate:n #1
3214 {
3215     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
3216     {
3217         \str_map_inline:nn {##1}
3218         {
3219             \str_if_in:nnF { NnpcofVvx } {####1}
3220             {

```

```

3221           \_kernel_msg_error:nnnn { kernel } { invalid-exp-args }
3222           {####1} {##1}
3223           \str_map_break:n { \use_none:nn }
3224       }
3225   }
3226   \_cs_generate_internal_variant:n {##1}
3227 }
3228 }
```

(End definition for `\exp_args_generate:n`. This function is documented on page 266.)

6.8 Definitions with the automated technique

Some of these could be done more efficiently, but the complexity of coding then becomes an issue. Notice that the auto-generated functions actually take no arguments themselves.

`\exp_args:Nnc` Here are the actual function definitions, using the helper functions above. The group is used because `_cs_generate_internal_variant:n` redefines `_cs_tmp:w` locally.
`\exp_args:Nno`
`\exp_args:Nv`
`\exp_args:Nnv`
`\exp_args:Nne`
`\exp_args:Nnf`
`\exp_args:Noc`
`\exp_args:Noo`
`\exp_args:Nof`
`\exp_args:NVo`
`\exp_args:Nfo`
`\exp_args:Nff`
`\exp_args:Nee`
`\exp_args:NNx`
`\exp_args:Ncx`
`\exp_args:Nnx`
`\exp_args:Nox`
`\exp_args:Nxo`
`\exp_args:Nxx`

```

3229 \cs_set_protected:Npn \_cs_tmp:w #1
3230 {
3231     \group_begin:
3232     \exp_args:No \_cs_generate_internal_variant:n
3233         { \tl_to_str:n {#1} }
3234     \group_end:
3235 }
3236 \_cs_tmp:w { nc }
3237 \_cs_tmp:w { no }
3238 \_cs_tmp:w { nv }
3239 \_cs_tmp:w { nv }
3240 \_cs_tmp:w { ne }
3241 \_cs_tmp:w { nf }
3242 \_cs_tmp:w { oc }
3243 \_cs_tmp:w { oo }
3244 \_cs_tmp:w { of }
3245 \_cs_tmp:w { Vo }
3246 \_cs_tmp:w { fo }
3247 \_cs_tmp:w { ff }
3248 \_cs_tmp:w { ee }
3249 \_cs_tmp:w { Nx }
3250 \_cs_tmp:w { cx }
3251 \_cs_tmp:w { nx }
3252 \_cs_tmp:w { ox }
3253 \_cs_tmp:w { xo }
3254 \_cs_tmp:w { xx }
```

(End definition for `\exp_args:Nnc` and others. These functions are documented on page 31.)

`\exp_args>NNcf`
`\exp_args>NNno`
`\exp_args>NNnV`
`\exp_args>NNoo`
`\exp_args>NNVV`
`\exp_args:Ncno`
`\exp_args:NcnV`
`\exp_args:Ncoo`
`\exp_args:NcVV`
`\exp_args:NnnC`
`\exp_args:Nnno`
`\exp_args:Nnnf`
`\exp_args:Nfff`
`\exp_args:Nooo`
`\exp_args:Noof`
`\exp_args:Nffo`
`\exp_args:Neee`

```

3261 \__cs_tmp:w { cnV }
3262 \__cs_tmp:w { coo }
3263 \__cs_tmp:w { cVV }
3264 \__cs_tmp:w { nnc }
3265 \__cs_tmp:w { nno }
3266 \__cs_tmp:w { nnf }
3267 \__cs_tmp:w { nff }
3268 \__cs_tmp:w { ooo }
3269 \__cs_tmp:w { oof }
3270 \__cs_tmp:w { ffo }
3271 \__cs_tmp:w { eee }
3272 \__cs_tmp:w { NNx }
3273 \__cs_tmp:w { Nnx }
3274 \__cs_tmp:w { Nox }
3275 \__cs_tmp:w { nnx }
3276 \__cs_tmp:w { nox }
3277 \__cs_tmp:w { ccx }
3278 \__cs_tmp:w { cnx }
3279 \__cs_tmp:w { oox }

(End definition for \exp_args:Nncf and others. These functions are documented on page 32.)
```

3280 ⟨/package⟩

7 l3quark implementation

The following test files are used for this code: *m3quark001.lvt*.

3281 ⟨*package⟩

7.1 Quarks

3282 ⟨@=quark⟩

\quark_new:N Allocate a new quark.

```

3283 \cs_new_protected:Npn \quark_new:N #1
3284 {
3285     \__kernel_chk_if_free_cs:N #1
3286     \cs_gset_nopar:Npn #1 {#1}
3287 }
```

(End definition for \quark_new:N. This function is documented on page 38.)

\q_nil
\q_mark
\q_no_value
\q_stop Some “public” quarks. \q_stop is an “end of argument” marker, \q_nil is a empty value and \q_no_value marks an empty argument.

```

3288 \quark_new:N \q_nil
3289 \quark_new:N \q_mark
3290 \quark_new:N \q_no_value
3291 \quark_new:N \q_stop
```

(End definition for \q_nil and others. These variables are documented on page 39.)

\q_recursion_tail
\q_recursion_stop

Quarks for ending recursions. Only ever used there! \q_recursion_tail is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. \q_recursion_stop is placed directly after the list.

```
3292 \quark_new:N \q_recursion_tail
3293 \quark_new:N \q_recursion_stop
```

(End definition for \q_recursion_tail and \q_recursion_stop. These variables are documented on page 39.)

\s__quark

Private scan mark used in \3quark. We don't have \3scan yet, so we declare the scan mark here and add it to the scan mark pool later.

```
3294 \cs_new_eq:NN \s__quark \scan_stop:
```

(End definition for \s__quark.)

\q__quark_nil

Private quark use for some tests.

```
3295 \quark_new:N \q__quark_nil
```

(End definition for \q__quark_nil.)

\quark_if_recursion_tail_stop:N
\quark_if_recursion_tail_stop_do:Nn

When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapped up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
3296 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
3297 {
3298     \if_meaning:w \q_recursion_tail #1
3299         \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
3300     \fi:
3301 }
3302 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
3303 {
3304     \if_meaning:w \q_recursion_tail #1
3305         \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
3306     \else:
3307         \exp_after:wN \use_none:n
3308     \fi:
3309 }
```

(End definition for \quark_if_recursion_tail_stop:N and \quark_if_recursion_tail_stop_do:Nn. These functions are documented on page 40.)

\quark_if_recursion_tail_stop:n
\quark_if_recursion_tail_stop:oTF
\quark_if_recursion_tail_stop:on
\quark_if_recursion_tail_stop:off
_quark_if_recursion_tail:w

See \quark_if_nil:nTF for the details. Expanding _quark_if_recursion_tail:w once in front of the tokens chosen here gives an empty result if and only if #1 is exactly \q_recursion_tail.

```
3310 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
3311 {
3312     \tl_if_empty:oTF
3313     { \_quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
3314     { \use_none_delimit_by_q_recursion_stop:w }
3315 }
```

```

3316    }
3317 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
3318 {
3319     \tl_if_empty:cTF
3320     { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
3321     { \use_i_delimit_by_q_recursion_stop:nw }
3322     { \use_none:n }
3323 }
3324 \cs_new:Npn \__quark_if_recursion_tail:w
3325     #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
3326 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
3327 \cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }

```

(End definition for `\quark_if_recursion_tail_stop:n`, `\quark_if_recursion_tail_stop_do:nn`, and `__quark_if_recursion_tail:w`. These functions are documented on page 40.)

`\quark_if_recursion_tail_break:NN`
`\quark_if_recursion_tail_break:nN`

Analogues of the `\quark_if_recursion_tail_stop...` functions. Break the mapping using #2.

```

3328 \cs_new:Npn \quark_if_recursion_tail_break:NN #1#2
3329 {
3330     \if_meaning:w \q_recursion_tail #1
3331         \exp_after:wN #2
3332         \fi:
3333     }
3334 \cs_new:Npn \quark_if_recursion_tail_break:nN #1#2
3335 {
3336     \tl_if_empty:cT
3337     { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
3338     {#2}
3339 }

```

(End definition for `\quark_if_recursion_tail_break:NN` and `\quark_if_recursion_tail_break:nN`. These functions are documented on page 40.)

`\quark_if_nil_p:N`
`\quark_if_nil:NTF`
`\quark_if_no_value_p:N`

Here we test if we found a special quark as the first argument. We better start with `\q_no_value` as the first argument since the whole thing may otherwise loop if #1 is wrongly given a string like aabc instead of a single token.⁷

```

3340 \prg_new_conditional:Npnn \quark_if_nil:N #1 { p, T , F , TF }
3341 {
3342     \if_meaning:w \q_nil #1
3343         \prg_return_true:
3344     \else:
3345         \prg_return_false:
3346     \fi:
3347 }
3348 \prg_new_conditional:Npnn \quark_if_no_value:N #1 { p, T , F , TF }
3349 {
3350     \if_meaning:w \q_no_value #1
3351         \prg_return_true:
3352     \else:
3353         \prg_return_false:
3354     \fi:
3355 }

```

⁷It may still loop in special circumstances however!

```

3356 \prg_generate_conditional_variant:Nnn \quark_if_no_value:N
3357   { c } { p , T , F , TF }

```

(End definition for `\quark_if_nil:nTF` and `\quark_if_no_value:nTF`. These functions are documented on page 39.)

`\quark_if_nil_p:n` Let us explain `\quark_if_nil:n(TF)`. Expanding `_quark_if_nil:w` once is safe thanks to the trailing `\q_nil ??!`. The result of expanding once is empty if and only if both delimited arguments #1 and #2 are empty and #3 is delimited by the last tokens `?!`. Thanks to the leading `{}`, the argument #1 is empty if and only if the argument of `\quark_if_nil:n` starts with `\q_nil`. The argument #2 is empty if and only if this `\q_nil` is followed immediately by `?` or by `{}`?, coming either from the trailing tokens in the definition of `\quark_if_nil:n`, or from its argument. In the first case, `_quark_if_nil:w` is followed by `{ }\q_nil {} ? !\q_nil ??!`, hence #3 is delimited by the final `?!`, and the test returns `true` as wanted. In the second case, the result is not empty since the first `?!?` in the definition of `\quark_if_nil:n` stop #3. The auxiliary here is the same as `_tl_if_empty_if:o`, with the same comments applying.

```

3358 \prg_new_conditional:Npnn \quark_if_nil:n #1 { p , T , F , TF }
3359   {
3360     \_quark_if_empty_if:o
3361     { \_quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
3362     \prg_return_true:
3363   \else:
3364     \prg_return_false:
3365   \fi:
3366   }
3367 \cs_new:Npn \_quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
3368 \prg_new_conditional:Npnn \quark_if_no_value:n #1 { p , T , F , TF }
3369   {
3370     \_quark_if_empty_if:o
3371     { \_quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
3372     \prg_return_true:
3373   \else:
3374     \prg_return_false:
3375   \fi:
3376   }
3377 \cs_new:Npn \_quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
3378 \prg_generate_conditional_variant:Nnn \quark_if_nil:n
3379   { V , o } { p , TF , T , F }
3380 \cs_new:Npn \_quark_if_empty_if:o #1
3381   {
3382     \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
3383     \_kernel_tl_to_str:w \exp_after:wN {#1} \q_nil
3384   }

```

(End definition for `\quark_if_nil:nTF` and others. These functions are documented on page 39.)

`_kernel_quark_new_test:N` The function `_kernel_quark_new_test:N` defines #1 in a similar way as `\quark_if_recursion_tail_...` functions (as described below), using `\q_{namespace}_recursion_tail` as the test quark and `\q_{namespace}_recursion_stop` as the delimiter quark, where the `<namespace>` is determined as the first _-delimited part in #1.

There are six possible function types which this function can define, and which is defined depends on the signature of the function being defined:

```
:n gives an analogue of \quark_if_recursion_tail_stop:n
:nn gives an analogue of \quark_if_recursion_tail_stop_do:nn
:nN gives an analogue of \quark_if_recursion_tail_break:nN
:N gives an analogue of \quark_if_recursion_tail_stop:N
:Nn gives an analogue of \quark_if_recursion_tail_stop_do:Nn
:NN gives an analogue of \quark_if_recursion_tail_break:NN
```

Any other signature causes an error, as does a function without signature.

Similar to __kernel_quark_new_test:N, but defines quark branching conditionals like \quark_if_nil:nTF that test for the quark \q_{namespace}_{name}. The *(namespace)* and *(name)* are determined from the conditional #1, which must take the rather rigid form __{namespace}_quark_if_{name}:(*arg spec*). There are only two cases for the *(arg spec)* here:

```
:n gives an analogue of \quark_if_nil:n(TF)
:N gives an analogue of \quark_if_nil:N(TF)
```

Any other signature causes an error, as does a function without signature. We use low-level emptiness tests as !3tl is not available yet when these functions are used; thankfully we only care about whether strings are empty so a simple \if_meaning:w \q_nil *(string)* \q_nil suffices.

```
3385 \cs_new_protected:Npn \__kernel_quark_new_test:N #1
3386   { \__quark_new_test_aux:Nx #1 { \__quark_module_name:N #1 } }
3387 \cs_new_protected:Npn \__quark_new_test_aux:Nn #1 #2
3388   {
3389     \if_meaning:w \q_nil #2 \q_nil
3390       \__kernel_msg_error:nmx { kernel } { invalid-quark-function }
3391       { \token_to_str:N #1 }
3392     \else:
3393       \__quark_new_test:Nccn #1
3394       { q__#2_recursion_tail } { q__#2_recursion_stop } { __#2 }
3395     \fi:
3396   }
3397 \cs_generate_variant:Nn \__quark_new_test_aux:Nn { Nx }
3398 \cs_new_protected:Npn \__quark_new_test:NNNn #1
3399   {
3400     \exp_last_unbraced:Nf \__quark_new_test_aux:nnNNnnnn
3401     { \cs_split_function:N #1 }
3402     #1 { test }
3403   }
3404 \cs_generate_variant:Nn \__quark_new_test:NNNn { Ncc }
3405 \cs_new_protected:Npn \__kernel_quark_new_conditional:Nn #1
3406   {
3407     \__quark_new_conditional:Nxxn #1
3408     { \__quark_quark_conditional_name:N #1 }
3409     { \__quark_module_name:N #1 }
```

```

3410    }
3411 \cs_new_protected:Npn \__quark_new_conditional:Nnnn #1#2#3#4
3412 {
3413     \if_meaning:w \q_nil #2 \q_nil
3414         \_kernel_msg_error:nnx { kernel } { invalid-quark-function }
3415         { \token_to_str:N #1 }
3416     \else:
3417         \if_meaning:w \q_nil #3 \q_nil
3418             \_kernel_msg_error:nnx { kernel } { invalid-quark-function }
3419             { \token_to_str:N #1 }
3420     \else:
3421         \exp_last_unbraced:Nf \__quark_new_test_aux:nnNNnnnn
3422             { \cs_split_function:N #1 }
3423             #1 { conditional }
3424             {#2} {#3} {#4}
3425         \fi:
3426     \fi:
3427 }
3428 \cs_generate_variant:Nn \__quark_new_conditional:Nnnn { Nxx }
3429 \cs_new_protected:Npn \__quark_new_test_aux:nnNNnnnn #1 #2 #3 #4 #5
3430 {
3431     \cs_if_exist_use:cTF { __quark_new_#5_#2:Nnnn } { #4 }
3432     {
3433         \_kernel_msg_error:nnxx { kernel } { invalid-quark-function }
3434         { \token_to_str:N #4 } {#2}
3435         \use_none:nnn
3436     }
3437 }

```

(End definition for `__kernel_quark_new_test:N` and others.)

These macros implement the six possibilities mentioned above, passing the right arguments to `__quark_new_test_aux_do:nNNnnnnNNn`, which defines some auxiliaries, and then to `__quark_new_test_define_tl:nNnNNn` (`:n(n)` variants) or to `__quark_new_test_define_ifx:nNnNNn` (`:N(n)`) which define the main conditionals.

```

3438 \cs_new_protected:Npn \__quark_new_test_n:Nnnn #1 #2 #3 #4
3439 {
3440     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { none } { } { } { }
3441     \__quark_new_test_define_tl:nNnNNn #1 { }
3442 }
3443 \cs_new_protected:Npn \__quark_new_test_nn:Nnnn #1 #2 #3 #4
3444 {
3445     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
3446     \__quark_new_test_define_tl:nNnNNn #1 { \use_none:n }
3447 }
3448 \cs_new_protected:Npn \__quark_new_test_nN:Nnnn #1 #2 #3 #4
3449 {
3450     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
3451     \__quark_new_test_define_break_tl:nNNNNn #1 { }
3452 }
3453 \cs_new_protected:Npn \__quark_new_test_N:Nnnn #1 #2 #3 #4
3454 {
3455     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { none } { } { } { }
3456     \__quark_new_test_define_ifx:nNnNNn #1 { }

```

```

3457      }
3458 \cs_new_protected:Npn \__quark_new_test_Nn:Nnnn #1 #2 #3 #4
3459  {
3460    \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
3461    \__quark_new_test_define_ifx:nNnNNn #1
3462    { \else: \exp_after:wN \use_none:n }
3463  }
3464 \cs_new_protected:Npn \__quark_new_test_NN:Nnnn #1 #2 #3 #4
3465  {
3466    \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
3467    \__quark_new_test_define_break_ifx:nNNNNn #1 { }
3468  }

```

(End definition for `__quark_new_test_n:Nnnn` and others.)

`__quark_new_test_aux_do:nNNnnnnNNn`
`__quark_test_define_aux:NNNNnnNNn`

`__quark_new_test_aux_do:nNNnnnnNNn` makes the control sequence names which will be used by `__quark_test_define_aux:NNNNnnNNn`, and then later by `__quark_new-test_define_tl:nNnNNn` or `__quark_new_test_define_ifx:nNnNNn`. The control sequences defined here are analogous to `__quark_if_recursion_tail:w` and to `\use_(none|i)_delimit_by_q_recursion_stop:(|n)w`.

The name is composed by the name-space and the name of the quarks. Suppose `__kernel_quark_new_test:N` was used with:

```
\__kernel_quark_new_test:N \__test_quark_tail:n
```

then the first auxiliary will be `__test_quark_recursion_tail:w`, and the second one will be `__test_use_none_delimit_by_q_recursion_stop:w`.

Note that the actual quarks are *not* defined here. They should be defined separately using `\quark_new:N`.

```

3469 \cs_new_protected:Npn \__quark_new_test_aux_do:nNNnnnnNNn #1 #2 #3 #4 #5
3470  {
3471    \exp_args:Ncc \__quark_test_define_aux:NNNNnnNNn
3472    { #1 _quark_recursion_tail:w }
3473    { #1 _use_ #4 _delimit_by_q_recursion_stop: #5 w }
3474    #2 #3
3475  }
3476 \cs_new_protected:Npn \__quark_test_define_aux:NNNNnnNNn #1 #2 #3 #4 #5 #6 #7
3477  {
3478    \cs_gset:Npn #1 ##1 #3 ##2 ? ##3 ?! { ##1 ##2 }
3479    \cs_gset:Npn #2 ##1 #6 #4 {#5}
3480    #7 {##1} #1 #2 #3
3481  }

```

(End definition for `__quark_new_test_aux_do:nNNnnnnNNn` and `__quark_test_define_aux:NNNNnnNNn`.)

Finally, these two macros define the main conditional function using what's been set up before.

```

3482 \cs_new_protected:Npn \__quark_new_test_define_tl:nNnNNn #1 #2 #3 #4 #5 #6
3483  {
3484    \cs_new:Npn #5 #1
3485    {
3486      \tl_if_empty:oTF
3487      { #2 {} ##1 {} ?! #4 ??! }
3488      {##3} {#6}

```

```

3489     }
3490   }
3491 \cs_new_protected:Npn \__quark_new_test_define_ifx:nNnNNn #1 #2 #3 #4 #5 #6
3492   {
3493     \cs_new:Npn #5 #1
3494     {
3495       \if_meaning:w #4 ##1
3496         \exp_after:wN #3
3497         #6
3498       \fi:
3499     }
3500   }
3501 \cs_new_protected:Npn \__quark_new_test_define_break_t1:nNnNNn #1 #2 #3
3502   { \__quark_new_test_define_t1:nNnNNn {##1##2} #2 {##2} }
3503 \cs_new_protected:Npn \__quark_new_test_define_break_ifx:nNnNNn #1 #2 #3
3504   { \__quark_new_test_define_ifx:nNnNNn {##1##2} #2 {##2} }

```

(End definition for `__quark_new_test_define_t1:nNnNNn` and others.)

`__quark_new_conditional_n:Nnnn`
`__quark_new_conditional_N:Nnnn`

These macros implement the two possibilities for branching quark conditionals, passing the right arguments to `__quark_new_conditional_aux_do:NNnnn`, which defines some auxiliaries and defines the main conditionals.

```

3505 \cs_new_protected:Npn \__quark_new_conditional_n:Nnnn
3506   { \__quark_new_conditional_aux_do:NNnnn \use_i:nn }
3507 \cs_new_protected:Npn \__quark_new_conditional_N:Nnnn
3508   { \__quark_new_conditional_aux_do:NNnnn \use_ii:nn }

```

(End definition for `__quark_new_conditional_n:Nnnn` and `__quark_new_conditional_N:Nnnn`.)

Similar to the previous macros, but branching conditionals only require one auxiliary, so we take a shortcut. In `__quark_new_conditional_define:NNNNn`, #4 is `\use_i:nn` to define the n-type function (which needs an auxiliary) and is `\use_ii:nn` to define the N-type function.

```

3509 \cs_new_protected:Npn \__quark_new_conditional_aux_do:NNnnn #1 #2 #3 #4
3510   {
3511     \exp_args:Ncc \__quark_new_conditional_define:NNNNn
3512       { __ #4 _if_quark_ #3 :w } { q__ #4 _ #3 } #2 #1
3513   }
3514 \cs_new_protected:Npn \__quark_new_conditional_define:NNNNn #1 #2 #3 #4 #5
3515   {
3516     #4 { \cs_gset:Npn #1 ##1 #2 ##2 ? ##3 ?! { ##1 ##2 } } { }
3517     \exp_args:Nno \use:n { \prg_new_conditional:Npnn #3 ##1 {##5} }
3518     {
3519       #4 { \__quark_if_empty_if:o { #1 {} ##1 {} ?! #2 ??! } }
3520         { \if_meaning:w #2 ##1 }
3521       \prg_return_true: \else: \prg_return_false: \fi:
3522     }
3523   }

```

(End definition for `__quark_new_conditional_aux_do:NNnnn` and `__quark_new_conditional_define:NNNNn`.)

`__quark_module_name:N`
`__quark_module_name:w`
`__quark_module_name_loop:w`
`__quark_module_name_end:w`

`__quark_module_name:N` takes a control sequence and returns its `(module)` name, determined as the first non-empty non-single-character word, separated by `_` or `:`. These rules give the correct result for public functions `\(module)_...`, private functions `__-\(module)_...`, and variables such as `\l_\(module)_...`. If no valid module is found the

result is an empty string. The approach is to first cut off everything after the (first) : if any is present, then repeatedly grab _-delimited words until finding one of length at least 2 (we use low-level tests as |3tl| is not fully available when _kernel_quark_new_- test:N is first used. If no *<module>* is found (such as in \::n) we get the trailing marker \use_none:n {}, which expands to nothing.

```

3524  \cs_set:Npn \__quark_tmp:w #1#2
3525    {
3526      \cs_new:Npn \__quark_module_name:N ##1
3527        {
3528          \exp_last_unbraced:Nf \__quark_module_name:w
3529            { \cs_to_str:N ##1 } #1 \s__quark
3530        }
3531      \cs_new:Npn \__quark_module_name:w ##1 #1 ##2 \s__quark
3532        { \__quark_module_name_loop:w ##1 #2 \use_none:n {} #2 \s__quark }
3533      \cs_new:Npn \__quark_module_name_loop:w ##1 #2
3534        {
3535          \use_i_i:n:n \if_meaning:w \prg_do_nothing:
3536            ##1 \prg_do_nothing: \prg_do_nothing:
3537              \exp_after:wN \__quark_module_name_loop:w
3538            \else:
3539              \__quark_module_name_end:w ##1
3540            \fi:
3541        }
3542      \cs_new:Npn \__quark_module_name_end:w
3543        ##1 \fi: ##2 \s__quark { \fi: ##1 }
3544    }
3545  \exp_after:wN \__quark_tmp:w \tl_to_str:n { : _ }
```

(End definition for __quark_module_name:N and others.)

__quark_quark_conditional_name:N __quark_quark_conditional_name:w __quark_quark_conditional_name:N determines the quark name that the quark conditional function ##1 queries, as the part of the function name between _quark_if_ and the trailing :. Again we define it through __quark_tmp:w, which receives : as #1 and _quark_if_ as #2. The auxiliary __quark_quark_conditional_name:w returns the part between the first _quark_if_ and the next :, and we apply this auxiliary to the function name followed by : (in case the function name is lacking a signature), and _quark_if_: so that __quark_quark_conditional_name:N returns an empty string if _quark_if_ is not present.

```

3546  \cs_set:Npn \__quark_tmp:w #1 #2 \s__quark
3547    {
3548      \cs_new:Npn \__quark_quark_conditional_name:N ##1
3549        {
3550          \exp_last_unbraced:Nf \__quark_quark_conditional_name:w
3551            { \cs_to_str:N ##1 } #1 #2 #1 \s__quark
3552        }
3553      \cs_new:Npn \__quark_quark_conditional_name:w
3554        ##1 #2 ##2 #1 ##3 \s__quark {##2}
3555    }
3556  \exp_after:wN \__quark_tmp:w \tl_to_str:n { : _quark_if_ } \s__quark
```

(End definition for __quark_quark_conditional_name:N and __quark_quark_conditional_name:w.)

7.2 Scan marks

3557 ⟨@@=scan⟩

\g__scan_marks_tl The list of all scan marks currently declared. No l3tl yet, so define this by hand.

3558 \cs_gset:Npn \g__scan_marks_tl { }

(End definition for \g__scan_marks_tl.)

\scan_new:N Check whether the variable is already a scan mark, then declare it to be equal to \scan_stop: globally.

```
3559 \cs_new_protected:Npn \scan_new:N #1
3560 {
3561     \tl_if_in:NnTF \g__scan_marks_tl { #1 }
3562     {
3563         \__kernel_msg_error:n { kernel } { scanmark-already-defined }
3564         { \token_to_str:N #1 }
3565     }
3566     {
3567         \tl_gput_right:Nn \g__scan_marks_tl {#1}
3568         \cs_new_eq:NN #1 \scan_stop:
3569     }
3570 }
```

(End definition for \scan_new:N. This function is documented on page 41.)

\s_stop We only declare one scan mark here, more can be defined by specific modules. Can't use \scan_new:N yet because l3tl isn't loaded, so define \s_stop by hand and add it to \g__scan_marks_tl. We also add \s_quark (declared earlier) to the pool here. Since it lives in a different namespace, a little l3docstrip cheating is necessary.

```
3571 \cs_new_eq:NN \s_stop \scan_stop:
3572 \cs_gset_nopar:Npx \g__scan_marks_tl
3573 {
3574     \exp_not:o \g__scan_marks_tl
3575     \s_stop
3576     ⟨@@=quark⟩
3577     \s_quark
3578     ⟨@@=scan⟩
3579 }
```

(End definition for \s_stop. This variable is documented on page 42.)

\use_none_delimit_by_s_stop:w Similar to \use_none_delimit_by_q_stop:w.

3580 \cs_new:Npn \use_none_delimit_by_s_stop:w #1 \s_stop { }

(End definition for \use_none_delimit_by_s_stop:w. This function is documented on page 42.)

3581 ⟨/package⟩

8 **\tl** implementation

```
3582  {*package}  
3583  {@@=tl}
```

A token list variable is a **T_EX** macro that holds tokens. By using the ε -**T_EX** primitive **\unexpanded** inside a **T_EX** **\edef** it is possible to store any tokens, including **#**, in this way.

8.1 Functions

_kernel_tl_set:Nx These two are supplied to get better performance for macros which would otherwise use **\tl_set:Nx** or **\tl_gset:Nx** internally.

```
3584  \cs_new_eq:NN \_kernel_tl_set:Nx \cs_set_nopar:Npx  
3585  \cs_new_eq:NN \_kernel_tl_gset:Nx \cs_gset_nopar:Npx
```

(End definition for **_kernel_tl_set:Nx** and **_kernel_tl_gset:Nx**.)

\tl_new:N Creating new token list variables is a case of checking for an existing definition and doing the definition.

```
3586  \cs_new_protected:Npn \tl_new:N #1  
3587  {  
3588      \_kernel_chk_if_free_cs:N #1  
3589      \cs_gset_eq:NN #1 \c_empty_tl  
3590  }  
3591  \cs_generate_variant:Nn \tl_new:N { c }
```

(End definition for **\tl_new:N**. This function is documented on page 43.)

\tl_const:Nn Constants are also easy to generate. They use **\cs_gset_nopar:Npx** instead of **_kernel_tl_gset:Nx** so that the correct scope checking is applied if **\l3debug** is used.

```
3592  \cs_new_protected:Npn \tl_const:Nn #1#2  
3593  {  
3594      \_kernel_chk_if_free_cs:N #1  
3595      \cs_gset_nopar:Npx #1 { \_kernel_exp_not:w {#2} }  
3596  }  
3597  \cs_new_protected:Npn \tl_const:Nx #1#2  
3598  {  
3599      \_kernel_chk_if_free_cs:N #1  
3600      \cs_gset_nopar:Npx #1 {#2}  
3601  }  
3602  \cs_generate_variant:Nn \tl_const:Nn { c }  
3603  \cs_generate_variant:Nn \tl_const:Nx { c }
```

(End definition for **\tl_const:Nn**. This function is documented on page 43.)

\tl_clear:N Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```
3604  \cs_new_protected:Npn \tl_clear:N #1  
3605  { \tl_set_eq:NN #1 \c_empty_tl }  
3606  \cs_new_protected:Npn \tl_gclear:N #1  
3607  { \tl_gset_eq:NN #1 \c_empty_tl }  
3608  \cs_generate_variant:Nn \tl_clear:N { c }  
3609  \cs_generate_variant:Nn \tl_gclear:N { c }
```

(End definition for `\tl_clear:N` and `\tl_gclear:N`. These functions are documented on page 43.)

`\tl_clear_new:N` Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```
3610 \cs_new_protected:Npn \tl_clear_new:N #1
3611   { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
3612 \cs_new_protected:Npn \tl_gclear_new:N #1
3613   { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
3614 \cs_generate_variant:Nn \tl_clear_new:N { c }
3615 \cs_generate_variant:Nn \tl_gclear_new:N { c }
```

(End definition for `\tl_clear_new:N` and `\tl_gclear_new:N`. These functions are documented on page 44.)

`\tl_set_eq:NN` For setting token list variables equal to each other. To allow for patching, the arguments have to be explicit.

```
3616 \cs_new_protected:Npn \tl_set_eq:NN #1#2 { \cs_set_eq:NN #1 #2 }
3617 \cs_new_protected:Npn \tl_gset_eq:NN #1#2 { \cs_gset_eq:NN #1 #2 }
3618 \cs_generate_variant:Nn \tl_set_eq:NN { cN, Nc, cc }
3619 \cs_generate_variant:Nn \tl_gset_eq:NN { cN, Nc, cc }
```

(End definition for `\tl_set_eq:NN` and `\tl_gset_eq:NN`. These functions are documented on page 44.)

`\tl_concat:NNN` Concatenating token lists is easy. When checking is turned on, all three arguments must be checked: a token list #2 or #3 equal to `\scan_stop`: would lead to problems later on.

```
3620 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
3621   {
3622     \__kernel_tl_set:Nx #1
3623     {
3624       \__kernel_exp_not:w \exp_after:wN {#2}
3625       \__kernel_exp_not:w \exp_after:wN {#3}
3626     }
3627   }
3628 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
3629   {
3630     \__kernel_tl_gset:Nx #1
3631     {
3632       \__kernel_exp_not:w \exp_after:wN {#2}
3633       \__kernel_exp_not:w \exp_after:wN {#3}
3634     }
3635   }
3636 \cs_generate_variant:Nn \tl_concat:NNN { ccc }
3637 \cs_generate_variant:Nn \tl_gconcat:NNN { ccc }
```

(End definition for `\tl_concat:NNN` and `\tl_gconcat:NNN`. These functions are documented on page 44.)

`\tl_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```
3638 \prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
3639 \prg_new_eq_conditional:NNn \tl_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(End definition for `\tl_if_exist:NTF`. This function is documented on page 44.)

8.2 Constant token lists

`\c_empty_tl` Never full. We need to define that constant before using `\tl_new:N`.

```
3640 \tl_const:Nn \c_empty_tl { }
```

(End definition for `\c_empty_tl`. This variable is documented on page 58.)

`\c_novalue_tl` A special marker: as we don't have `\char_generate:nn` yet, has to be created the old-fashioned way.

```
3641 \group_begin:
3642 \tex_lccode:D 'A = '-
3643 \tex_lccode:D 'N = 'N
3644 \tex_lccode:D 'V = 'V
3645 \tex_lowercase:D
3646 {
3647   \group_end:
3648   \tl_const:Nn \c_novalue_tl { ANoValue- }
3649 }
```

(End definition for `\c_novalue_tl`. This variable is documented on page 58.)

`\c_space_tl` A space as a token list (as opposed to as a character).

```
3650 \tl_const:Nn \c_space_tl { ~ }
```

(End definition for `\c_space_tl`. This variable is documented on page 58.)

8.3 Adding to token list variables

`\tl_set:Nn` By using `\exp_not:n` token list variables can contain # tokens, which makes the token list registers provided by TeX more or less redundant. The `\tl_set:No` version is done "by hand" as it is used quite a lot.

```
3651 \cs_new_protected:Npn \tl_set:Nn #1#2
3652   { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w {#2} } }
3653 \cs_new_protected:Npn \tl_set:No #1#2
3654   { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#2} } }
3655 \cs_new_protected:Npn \tl_set:Nx #1#2
3656   { \__kernel_tl_set:Nx #1 {#2} }
3657 \cs_new_protected:Npn \tl_gset:Nn #1#2
3658   { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w {#2} } }
3659 \cs_new_protected:Npn \tl_gset:No #1#2
3660   { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#2} } }
3661 \cs_new_protected:Npn \tl_gset:Nx #1#2
3662   { \__kernel_tl_gset:Nx #1 {#2} }
3663 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
3664 \cs_generate_variant:Nn \tl_set:Nx { c }
3665 \cs_generate_variant:Nn \tl_set:Nn { c, co , cV , cv , cf }
3666 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
3667 \cs_generate_variant:Nn \tl_gset:Nx { c }
3668 \cs_generate_variant:Nn \tl_gset:Nn { c, co , cV , cv , cf }

\tl_gset:cV
\tl_gset:cv
\tl_gset:co
\tl_gset:cf
\tl_gset:cx
```

(End definition for `\tl_set:Nn` and `\tl_gset:Nn`. These functions are documented on page 44.)

\tl_put_left:Nn Adding to the left is done directly to gain a little performance.

```

3669 \cs_new_protected:Npn \tl_put_left:Nn #1#2
370   {
371     \__kernel_tl_set:Nx #1
372     { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
373   }
374 \cs_new_protected:Npn \tl_put_left:NV #1#2
375   {
376     \__kernel_tl_set:Nx #1
377     { \exp_not:V #2 \__kernel_exp_not:w \exp_after:wN {#1} }
378   }
379 \cs_new_protected:Npn \tl_put_left:No #1#2
380   {
381     \__kernel_tl_set:Nx #1
382     {
383       \__kernel_exp_not:w \exp_after:wN {#2}
384       \__kernel_exp_not:w \exp_after:wN {#1}
385     }
386   }
387 \cs_new_protected:Npn \tl_put_left:Nx #1#2
388   { \__kernel_tl_set:Nx #1 { #2 \__kernel_exp_not:w \exp_after:wN {#1} } }
389 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
390   {
391     \__kernel_tl_gset:Nx #1
392     { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
393   }
394 \cs_new_protected:Npn \tl_gput_left:NV #1#2
395   {
396     \__kernel_tl_gset:Nx #1
397     { \exp_not:V #2 \__kernel_exp_not:w \exp_after:wN {#1} }
398   }
399 \cs_new_protected:Npn \tl_gput_left:No #1#2
400   {
401     \__kernel_tl_gset:Nx #1
402     {
403       \__kernel_exp_not:w \exp_after:wN {#2}
404       \__kernel_exp_not:w \exp_after:wN {#1}
405     }
406   }
407 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
408   { \__kernel_tl_gset:Nx #1 { #2 \__kernel_exp_not:w \exp_after:wN {#1} } }
409 \cs_generate_variant:Nn \tl_put_left:Nn { c }
410 \cs_generate_variant:Nn \tl_put_left:NV { c }
411 \cs_generate_variant:Nn \tl_put_left:No { c }
412 \cs_generate_variant:Nn \tl_put_left:Nx { c }
413 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
414 \cs_generate_variant:Nn \tl_gput_left:NV { c }
415 \cs_generate_variant:Nn \tl_gput_left:No { c }
416 \cs_generate_variant:Nn \tl_gput_left:Nx { c }

```

(End definition for \tl_put_left:Nn and \tl_gput_left:Nn. These functions are documented on page 44.)

\tl_put_right:Nn The same on the right.

```

\tl_put_right:NV
\tl_put_right:No
\tl_put_right:Nx
\tl_put_right:cn
\tl_put_right:cV
\tl_put_right:co
\tl_put_right:cx
\tl_gput_right:Nn
\tl_gput_right:NV
\tl_gput_right:No
\tl_gput_right:Nx

```

```

3717 \cs_new_protected:Npn \tl_put_right:Nn #1#2
3718   { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
3719 \cs_new_protected:Npn \tl_put_right:NV #1#2
3720   {
3721     \__kernel_tl_set:Nx #1
3722     { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:V #2 }
3723   }
3724 \cs_new_protected:Npn \tl_put_right:No #1#2
3725   {
3726     \__kernel_tl_set:Nx #1
3727     {
3728       \__kernel_exp_not:w \exp_after:wN {#1}
3729       \__kernel_exp_not:w \exp_after:wN {#2}
3730     }
3731   }
3732 \cs_new_protected:Npn \tl_put_right:Nx #1#2
3733   { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#1} #2 } }
3734 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
3735   { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
3736 \cs_new_protected:Npn \tl_gput_right:NV #1#2
3737   {
3738     \__kernel_tl_gset:Nx #1
3739     { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:V #2 }
3740   }
3741 \cs_new_protected:Npn \tl_gput_right:No #1#2
3742   {
3743     \__kernel_tl_gset:Nx #1
3744     {
3745       \__kernel_exp_not:w \exp_after:wN {#1}
3746       \__kernel_exp_not:w \exp_after:wN {#2}
3747     }
3748   }
3749 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
3750   { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#1} #2 } }
3751 \cs_generate_variant:Nn \tl_put_right:Nn { c }
3752 \cs_generate_variant:Nn \tl_put_right:NV { c }
3753 \cs_generate_variant:Nn \tl_put_right:No { c }
3754 \cs_generate_variant:Nn \tl_put_right:Nx { c }
3755 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
3756 \cs_generate_variant:Nn \tl_gput_right:NV { c }
3757 \cs_generate_variant:Nn \tl_gput_right:No { c }
3758 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

(End definition for `\tl_put_right:Nn` and `\tl_gput_right:Nn`. These functions are documented on page 44.)

8.4 Internal quarks and quark-query functions

<code>\q__tl_nil</code>	Internal quarks.
<code>\q__tl_mark</code>	<code>\quark_new:N \q__tl_nil</code>
<code>\q__tl_stop</code>	<code>\quark_new:N \q__tl_mark</code>
	<code>\quark_new:N \q__tl_stop</code>

(End definition for `\q__tl_nil`, `\q__tl_mark`, and `\q__tl_stop`.)

```

\q_tl_recursion_tail Internal recursion quarks.
\q_tl_recursion_stop
 3762 \quark_new:N \q_tl_recursion_tail
 3763 \quark_new:N \q_tl_recursion_stop

(End definition for \q_tl_recursion_tail and \q_tl_recursion_stop.)

\_tl_if_recursion_tail_break:nN Functions to query recursion quarks.
\_tl_if_recursion_tail_stop_p:n
\_tl_if_recursion_tail_stop:nF
 3764 \__kernel_quark_new_test:N \_tl_if_recursion_tail_break:nN
 3765 \__kernel_quark_new_conditional:Nn \_tl_quark_if_nil:n { TF }

(End definition for \_tl_if_recursion_tail_break:nN and \_tl_if_recursion_tail_stop:nF.)

```

8.5 Reassigning token list category codes

\c_tl_rescan_marker_tl The rescanning code needs a special token list containing the same character (chosen here to be a colon) with two different category codes: it cannot appear in the tokens being rescanned since all colons have the same category code.

```

3766 \tl_const:Nx \c_tl_rescan_marker_tl { : \token_to_str:N : }

(End definition for \c_tl_rescan_marker_tl.)

```

\tl_set_rescan:Nnn \tl_set_rescan:Nno \tl_set_rescan:Nnx \tl_set_rescan:cnn \tl_set_rescan:cno \tl_set_rescan:cnx \tl_gset_rescan:Nnn \tl_gset_rescan:Nno \tl_gset_rescan:Nnx \tl_gset_rescan:cnn \tl_gset_rescan:cno \tl_gset_rescan:cnx \tl_rescan:nn __tl_rescan_aux: __tl_set_rescan:NNnn __tl_set_rescan_multi:NNn __tl_rescan:NNw In a group, after some initial setup explained below and the user setup #3 (followed by \scan_stop: to be safe), there is a call to _tl_set_rescan:nNN. This shared auxiliary defined later distinguishes single-line and multi-line “files”. In the simplest case of multi-line files, it calls (with the same arguments) _tl_set_rescan_multi:nNN, whose code is included here to help understand the approach. This function rescans its argument #1, closes the group, and performs the assignment.

One difficulty when rescanning is that \scantokens treats the argument as a file, and without the correct settings a T_EX error occurs:

```

! File ended while scanning definition of ...

```

A related minor issue is a warning due to opening a group before the \scantokens and closing it inside that temporary file; we avoid that by setting \tracingnesting. The standard solution to the “File ended” error is to grab the rescanned tokens as a delimited argument of an auxiliary, here _tl_rescan:NNw, that performs the assignment, then let T_EX “execute” the end of file marker. As usual in delimited arguments we use \prg_do_nothing: to avoid stripping an outer set braces: this is removed by using o-expanding assignments. The delimiter cannot appear within the rescanned token list because it contains twice the same character, with different catcodes.

For \tl_rescan:nn we cannot simply call _tl_set_rescan:NNnn \prg_do_nothing: \use:n because that would leave the end-of-file marker *after* the result of rescanning. If that rescanned result is code that looks further in the input stream for arguments, it would break.

For multi-line files the only subtlety is that \newlinechar should be equal to \endlinechar because \newlinechar characters become new lines and then become \endlinechar characters when writing to an abstract file and reading back. This equality is ensured by setting \newlinechar equal to \endlinechar. Prior to this, \endlinechar is set to -1 if it was 32 (in particular true after \ExplSyntaxOn) to avoid unreasonable line-breaks at every space for instance in error messages triggered by the user setup. Another side effect of reading back from the file is that spaces (catcode 10) are ignored

at the beginning of lines, and spaces and tabs (character code 32 and 9) are ignored at the end of lines.

The two `\if_false: ... \fi:` are there to prevent alignment tabs to cause a change of tabular cell while rescanning. We put the “opening” one after `\group_begin:` so that if one accidentally f-expands `\tl_set_rescan:Nnn` braces remain balanced. This is essential in e-type arguments when `\expanded` is not available.

```

3767  \cs_new_protected:Npn \tl_rescan:nn #1#2
3768  {
3769      \tl_set_rescan:Nnn \l__tl_internal_a_tl {#1} {#2}
3770      \exp_after:wN \__tl_rescan_aux:
3771      \l__tl_internal_a_tl
3772  }
3773  \exp_args:NNo \cs_new_protected:Npn \__tl_rescan_aux:
3774  { \tl_clear:N \l__tl_internal_a_tl }
3775  \cs_new_protected:Npn \tl_set_rescan:Nnn
3776  { \__tl_set_rescan:NNnn \tl_set:No }
3777  \cs_new_protected:Npn \tl_gset_rescan:Nnn
3778  { \__tl_set_rescan:NNnn \tl_gset:No }
3779  \cs_new_protected:Npn \__tl_set_rescan:NNnn #1#2#3#4
3780  {
3781      \group_begin:
3782      \if_false: { \fi:
3783          \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
3784          \int_compare:nNnT \tex_endlinechar:D = { 32 }
3785          { \int_set:Nn \tex_endlinechar:D { -1 } }
3786          \int_set_eq:NN \tex_newlinechar:D \tex_endlinechar:D
3787          #3 \scan_stop:
3788          \exp_args:No \__tl_set_rescan:nNN { \tl_to_str:n {#4} } #1 #2
3789      \if_false: } \fi:
3790  }
3791  \cs_new_protected:Npn \__tl_set_rescan_multi:nNN #1#2#3
3792  {
3793      \tex_everyeof:D \exp_after:wN { \c__tl_rescan_marker_tl }
3794      \exp_after:wN \__tl_rescan>NNw
3795      \exp_after:wN #2
3796      \exp_after:wN #3
3797      \exp_after:wN \prg_do_nothing:
3798      \tex_scantokens:D {#1}
3799  }
3800  \exp_args:Nno \use:nn
3801  { \cs_new:Npn \__tl_rescan:NNw #1#2#3 } \c__tl_rescan_marker_tl
3802  {
3803      \group_end:
3804      #1 #2 {#3}
3805  }
3806  \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno , Nnx }
3807  \cs_generate_variant:Nn \tl_set_rescan:Nnn { c , cno , cnx }
3808  \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno , Nnx }
3809  \cs_generate_variant:Nn \tl_gset_rescan:Nnn { c , cno }

```

(End definition for `\tl_set_rescan:Nnn` and others. These functions are documented on page 46.)

The function `__tl_set_rescan:nNN` calls `__tl_set_rescan_multi:nNN` or `__tl_set_rescan_single:nnNN { ' }` depending on whether its argument is a single-line

```

\__tl_set_rescan_single:nnNN
\__tl_set_rescan_single_aux:nnnnNN
\__tl_set_rescan_single_aux:w

```

fragment of code/data or is made of multiple lines by testing for the presence of a `\newlinechar` character. If `\newlinechar` is out of range, the argument is assumed to be a single line.

For a single line, no `\endlinechar` should be added, so it is set to `-1`, and spaces should not be removed. Trailing spaces and tabs are a difficult matter, as TeX removes these at a very low level. The only way to preserve them is to rescan not the argument but the argument followed by a character with a reasonable category code. Here, `11` (letter) and `12` (other) are accepted, as these are convenient, suitable for delimiting an argument, and it is very unlikely that none of the ASCII characters are in one of these categories. To avoid selecting one particular character to put at the end, whose category code may have been modified, there is a loop through characters from `'` (ASCII 39) to `~` (ASCII 127). The choice of starting point was made because this is the start of a very long range of characters whose standard category is letter or other, thus minimizing the number of steps needed by the loop (most often just a single one). If no valid character is found (very rare), fall-back on `_tl_set_rescan_multi:nNN`.

Otherwise, once a valid character is found (let us use `'` in this explanation) run some code very similar to `_tl_set_rescan_multi:nNN` but with `'` added at both ends of the input. Of course, we need to define the auxiliary `_tl_set_rescan_single:NNww` on the fly to remove the additional `'` that is just before `::` (by which we mean `\c_tl_rescan_marker_tl`). Note that the argument must be delimited by `'` with the current catcode; this is done thanks to `\char_generate:nn`. Yet another issue is that the rescaned token list may contain a comment character, in which case the `'` we expected is not there. We fix this as follows: rather than just `::` we set `\everyeof` to `::{\langle code1\rangle} ::{\langle code2\rangle} \s_tl_stop`. The auxiliary `_tl_set_rescan_single:NNww` runs the `\o-expanding assignment`, expanding either `\langle code1\rangle` or `\langle code2\rangle` before its main argument `#3`. In the typical case without comment character, `\langle code1\rangle` is expanded, removing the leading `'`. In the rarer case with comment character, `\langle code2\rangle` is expanded, calling `_tl_set_rescan_single_aux:w`, which removes the trailing `::{\langle code1\rangle}` and the leading `'`.

```

3810 \cs_new_protected:Npn \_tl_set_rescan:nNN #1
3811 {
3812     \int_compare:nNnTF \tex_newlinechar:D < 0
3813         { \use_i:i:nn }
3814         {
3815             \exp_args:Nnf \tl_if_in:nNTF {\#1}
3816                 { \char_generate:nn { \tex_newlinechar:D } { 12 } }
3817         }
3818         { \_tl_set_rescan_multi:nNN }
3819         {
3820             \int_set:Nn \tex_endlinechar:D { -1 }
3821             \_tl_set_rescan_single:nnNN { ' }
3822         }
3823     {\#1}
3824 }
3825 \cs_new_protected:Npn \_tl_set_rescan_single:nnNN #1
3826 {
3827     \int_compare:nNnTF
3828         { \char_value_catcode:n {\#1} / 2 } = 6
3829         {
3830             \exp_args:Nof \_tl_set_rescan_single_aux:nnnNN
3831             \c_tl_rescan_marker_tl

```

```

3832         { \char_generate:nN {#1} { \char_value_catcode:n {#1} } }
3833     }
3834     {
3835         \int_compare:nNnTF {#1} < { '\~ }
3836         {
3837             \exp_args:Nf \__tl_set_rescan_single:nNN
3838             { \int_eval:n { #1 + 1 } }
3839         }
3840         { \__tl_set_rescan_multi:nNN }
3841     }
3842 }
3843 \cs_new_protected:Npn \__tl_set_rescan_single_aux:nNNN #1#2#3#4#5
3844 {
3845     \tex_everyeof:D
3846     {
3847         #1 \use_none:n
3848         #2 #1 { \exp:w \__tl_set_rescan_single_aux:w }
3849         \s__tl_stop
3850     }
3851     \cs_set:Npn \__tl_rescan>NNw ##1##2##3 #2 #1 ##4 ##5 \s__tl_stop
3852     {
3853         \group_end:
3854         ##1 ##2 { ##4 ##3 }
3855     }
3856     \exp_after:wN \__tl_rescan>NNw
3857     \exp_after:wN #4
3858     \exp_after:wN #5
3859     \tex_scantokens:D { #2 #3 #2 }
3860 }
3861 \exp_args:Nno \use:nn
3862 { \cs_new:Npn \__tl_set_rescan_single_aux:w #1 }
3863 \c__tl_rescan_marker_tl #2
3864 { \use_i:nn \exp_end: #1 }

```

(End definition for `__tl_set_rescan:nNN` and others.)

8.6 Modifying token list variables

`\tl_replace_all:Nnn`
`\tl_replace_all:cnn`
`\tl_greplace_all:Nnn`
`\tl_greplace_all:cnn`
`\tl_replace_once:Nnn`
`\tl_replace_once:cnn`
`\tl_greplace_once:Nnn`
`\tl_greplace_once:cnn`

All of the `replace` functions call `__tl_replace:NnNNNnn` with appropriate arguments. The first two arguments are explained later. The next controls whether the replacement function calls itself (`__tl_replace_next:w`) or stops (`__tl_replace_wrap:w`) after the first replacement. Next comes an x-type assignment function `\tl_set:Nx` or `\tl_gset:Nx` for local or global replacements. Finally, the three arguments `{⟨tl var⟩} {⟨pattern⟩} {⟨replacement⟩}` provided by the user. When describing the auxiliary functions below, we denote the contents of the `⟨tl var⟩` by `⟨token list⟩`.

```

3865 \cs_new_protected:Npn \tl_replace_once:Nnn
3866 { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_wrap:w \__kernel_tl_set:Nx }
3867 \cs_new_protected:Npn \tl_greplace_once:Nnn
3868 { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_wrap:w \__kernel_tl_gset:Nx }
3869 \cs_new_protected:Npn \tl_replace_all:Nnn
3870 { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_next:w \__kernel_tl_set:Nx }
3871 \cs_new_protected:Npn \tl_greplace_all:Nnn
3872 { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_next:w \__kernel_tl_gset:Nx }

```

```

3873 \cs_generate_variant:Nn \tl_replace_once:Nnn { c }
3874 \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
3875 \cs_generate_variant:Nn \tl_replace_all:Nnn { c }
3876 \cs_generate_variant:Nn \tl_greplace_all:Nnn { c }

```

(End definition for `\tl_replace_all:Nnn` and others. These functions are documented on page 45.)

```

\__tl_replace:NnNNNnn
\__tl_replace_auxi:NnnNNNnn
\__tl_replace_auxii:nNNNnn
  \__tl_replace_next:w
\__tl_replace_next_aux:w
  \__tl_replace_wrap:w

```

To implement the actual replacement auxiliary `__tl_replace_auxii:nNNNnn` we need a `<delimiter>` with the following properties:

- all occurrences of the `<pattern> #6` in “`<token list> <delimiter>`” belong to the `<token list>` and have no overlap with the `<delimiter>`,
- the first occurrence of the `<delimiter>` in “`<token list> <delimiter>`” is the trailing `<delimiter>`.

We first find the building blocks for the `<delimiter>`, namely two tokens `<A>` and `` such that `<A>` does not appear in `#6` and `#6` is not `` (this condition is trivial if `#6` has more than one token). Then we consider the delimiters “`<A>`” and “`<A> <A>n <A>n `”, for $n \geq 1$, where `<A>n` denotes n copies of `<A>`, and we choose as our `<delimiter>` the first one which is not in the `<token list>`.

Every delimiter in the set obeys the first condition: `#6` does not contain `<A>` hence cannot be overlapping with the `<token list>` and the `<delimiter>`, and it cannot be within the `<delimiter>` since it would have to be in one of the two `` hence be equal to this single token (or empty, but this is an error case filtered separately). Given the particular form of these delimiters, for which no prefix is also a suffix, the second condition is actually a consequence of the weaker condition that the `<delimiter>` we choose does not appear in the `<token list>`. Additionally, the set of delimiters is such that a `<token list>` of n tokens can contain at most $O(n^{1/2})$ of them, hence we find a `<delimiter>` with at most $O(n^{1/2})$ tokens in a time at most $O(n^{3/2})$. Bear in mind that these upper bounds are reached only in very contrived scenarios: we include the case “`<A>`” in the list of delimiters to try, so that the `<delimiter>` is simply `\q__tl_mark` in the most common situation where neither the `<token list>` nor the `<pattern>` contains `\q__tl_mark`.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty `<pattern> #6` is an error, and if `#1` is absent from both the `<token list> #5` and the `<pattern> #6` then we can use it as the `<delimiter>` through `__tl_replace_auxii:nNNNnn {#1}`. Otherwise, we end up calling `__tl_replace:NnNNNnn` repeatedly with the first two arguments `\q__tl_mark {?}, \? {??}, \?? {???`, and so on, until `#6` does not contain the control sequence `#1`, which we take as our `<A>`. The argument `#2` only serves to collect `?` characters for `#1`. Note that the order of the tests means that the first two are done every time, which is wasteful (for instance, we repeatedly test for the emptiness of `#6`). However, this is rare enough not to matter. Finally, choose `` to be `\q__tl_nil` or `\q__tl_stop` such that it is not equal to `#6`.

The `__tl_replace_auxii:NnnNNNnn` auxiliary receives `{<A>}` and `{<A>n }` as its arguments, initially with $n = 1$. If “`<A> <A>n <A>n `” is in the `<token list>` then increase n and try again. Once it is not anymore in the `<token list>` we take it as our `<delimiter>` and pass this to the `auxii` auxiliary.

```

3877 \cs_new_protected:Npn \__tl_replace:NnNNNnn #1#2#3#4#5#6#7
3878   {
3879     \tl_if_empty:nTF {#6}
3880     {
3881       \__kernel_msg_error:nx { kernel } { empty-search-pattern }

```

```

3882     { \tl_to_str:n {#7} }
3883   }
3884   {
3885     \tl_if_in:onTF { #5 #6 } {#1}
3886     {
3887       \tl_if_in:nnTF {#6} {#1}
3888       { \exp_args:Nc \__tl_replace:NnNNNnn {#2} {#2?} }
3889       {
3890         \__tl_quark_if_nil:nTF {#6}
3891         { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q__tl_stop } }
3892         { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q__tl_nil } }
3893       }
3894     }
3895     { \__tl_replace_auxii:nNNNnn {#1} }
3896     #3#4#5 {#6} {#7}
3897   }
3898 }
3899 \cs_new_protected:Npn \__tl_replace_auxi:NnnNNNnn #1#2#3
3900   {
3901     \tl_if_in:NnTF #1 { #2 #3 #3 }
3902     { \__tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2} }
3903     { \__tl_replace_auxii:nNNNnn { #2 #3 #3 } }
3904   }

```

The auxiliary `__tl_replace_auxii:nNNNnn` receives the following arguments:

```

{⟨delimiter⟩} ⟨function⟩ ⟨assignment⟩
⟨tl var⟩ {⟨pattern⟩} {⟨replacement⟩}

```

All of its work is done between `\group_align_safe_begin:` and `\group_align_safe_end:` to avoid issues in alignments. It does the actual replacement within `#3 #4 {...}`, an x-expanding `⟨assignment⟩` #3 to the `⟨tl var⟩` #4. The auxiliary `__tl_replace_next:w` is called, followed by the `⟨token list⟩`, some tokens including the `⟨delimiter⟩` #1, followed by the `⟨pattern⟩` #5. This auxiliary finds an argument delimited by #5 (the presence of a trailing #5 avoids runaway arguments) and calls `__tl_replace_wrap:w` to test whether this #5 is found within the `⟨token list⟩` or is the trailing one.

If on the one hand it is found within the `⟨token list⟩`, then #1 cannot contain the `⟨delimiter⟩` #1 that we worked so hard to obtain, thus `__tl_replace_wrap:w` gets #1 as its own argument ##1, and protects it against the x-expanding assignment. It also finds `\exp_not:n` as ##2 and does nothing to it, thus letting through `\exp_not:n {⟨replacement⟩}` into the assignment. Note that `__tl_replace_next:w` and `__tl_replace_wrap:w` are always called followed by two empty brace groups. These are safe because no delimiter can match them. They prevent losing braces when grabbing delimited arguments, but require the use of `\exp_not:o` and `\use_none:nn`, rather than simply `\exp_not:n`. Afterwards, `__tl_replace_next:w` is called to repeat the replacement, or `__tl_replace_wrap:w` if we only want a single replacement. In this second case, #1 is the `⟨remaining tokens⟩` in the `⟨token list⟩` and #2 is some `⟨ending code⟩` which ends the assignment and removes the trailing tokens #5 using some `\if_false: { \fi: }` trickery because #5 may contain any delimiter.

If on the other hand the argument #1 of `__tl_replace_next:w` is delimited by the trailing `⟨pattern⟩` #5, then #1 is “{ } { } `⟨token list⟩` `⟨delimiter⟩ {⟨ending code⟩}`”, hence `__tl_replace_wrap:w` finds “{ } { } `⟨token list⟩`” as #1 and the `⟨ending code⟩`

as ##2. It leaves the *(token list)* into the assignment and unbrates the *(ending code)* which removes what remains (essentially the *(delimiter)* and *(replacement)*).

```

3905 \cs_new_protected:Npn \__tl_replace_auxii:nNNNnn #1#2#3#4#5#6
3906 {
3907     \group_align_safe_begin:
3908     \cs_set:Npn \__tl_replace_wrap:w ##1 #1 ##2
3909         { \__kernel_exp_not:w \exp_after:wN { \use_none:nn ##1 } ##2 }
3910     \cs_set:Npx \__tl_replace_next:w ##1 #5
3911         {
3912             \exp_not:N \__tl_replace_wrap:w ##1
3913             \exp_not:n { #1 }
3914             \exp_not:n { \exp_not:n {#6} }
3915             \exp_not:n { #2 { } { } }
3916         }
3917     #3 #4
3918     {
3919         \exp_after:wN \__tl_replace_next_aux:w
3920         #4
3921         #1
3922         {
3923             \if_false: { \fi: }
3924             \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
3925         }
3926         #5
3927     }
3928     \group_align_safe_end:
3929 }
3930 \cs_new:Npn \__tl_replace_next_aux:w { \__tl_replace_next:w { } { } }
3931 \cs_new_eq:NN \__tl_replace_wrap:w ?
3932 \cs_new_eq:NN \__tl_replace_next:w ?
```

(End definition for __tl_replace:NnNNNnn and others.)

\tl_remove_once:Nn Removal is just a special case of replacement.

```

\tl_remove_once:cn
\tl_gremove_once:Nn
\tl_gremove_once:cn
3933 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
3934     { \tl_replace_once:Nnn #1 {#2} { } }
3935 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
3936     { \tl_greplace_once:Nnn #1 {#2} { } }
3937 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
3938 \cs_generate_variant:Nn \tl_gremove_once:Nn { c }
```

(End definition for \tl_remove_once:Nn and \tl_gremove_once:Nn. These functions are documented on page 45.)

\tl_remove_all:Nn Removal is just a special case of replacement.

```

\tl_remove_all:cn
\tl_gremove_all:Nn
\tl_gremove_all:cn
3939 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
3940     { \tl_replace_all:Nnn #1 {#2} { } }
3941 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
3942     { \tl_greplace_all:Nnn #1 {#2} { } }
3943 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
3944 \cs_generate_variant:Nn \tl_gremove_all:Nn { c }
```

(End definition for \tl_remove_all:Nn and \tl_gremove_all:Nn. These functions are documented on page 45.)

8.7 Token list conditionals

`\tl_if_empty_p:N`

`\tl_if_empty_p:c`

`\tl_if_empty:NTF`

`\tl_if_empty:cTF`

These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```

3945 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
3946   {
3947     \if_meaning:w #1 \c_empty_tl
3948       \prg_return_true:
3949     \else:
3950       \prg_return_false:
3951     \fi:
3952   }
3953 \prg_generate_conditional_variant:Nnn \tl_if_empty:N
3954   { c } { p , T , F , TF }
```

(End definition for `\tl_if_empty:NTF`. This function is documented on page 47.)

`\tl_if_empty_p:n`

`\tl_if_empty_p:V`

`\tl_if_empty:nTF`

`\tl_if_empty:VTF`

The `\if:w` triggers the expansion of `\tl_to_str:n` which converts the argument to a string: this is empty if and only if the argument is. Then `\if:w \scan_stop: ... \scan_stop:` is `true` if and only if the string `...` is empty. It could be tempting to use `\if:w \scan_stop: #1 \scan_stop:` directly. But this fails on a token list expanding to anything starting with `\scan_stop:` leaving everything that follows in the input stream.

```

3955 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
3956   {
3957     \if:w \scan_stop: \tl_to_str:n {#1} \scan_stop:
3958       \prg_return_true:
3959     \else:
3960       \prg_return_false:
3961     \fi:
3962   }
3963 \prg_generate_conditional_variant:Nnn \tl_if_empty:n
3964   { V } { p , TF , T , F }
```

(End definition for `\tl_if_empty:nTF`. This function is documented on page 47.)

`\tl_if_empty_p:o`

`\tl_if_empty:oTF`

`_tl_if_empty_if:o`

The auxiliary function `_tl_if_empty_if:o` is for use in various token list conditionals which reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on `\tl_if_empty:nTF`, but the expansion is hard-coded for efficiency, as this auxiliary function is used in several places. We don't put `\prg_return_true:` and so on in the definition of the auxiliary, because that would prevent an optimization applied to conditionals that end with this code. Also the `\@@_if_empty_if:o` is expanded once in `\tl_if_empty:oTF` for efficiency as well (and to reduce code doubling).

```

3965 \cs_new:Npn \_tl_if_empty_if:o #1
3966   {
3967     \if:w \scan_stop: \_kernel_tl_to_str:w \exp_after:wN {#1} \scan_stop:
3968   }
3969 \exp_args:Nno \use:n
3970   { \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F } }
3971   {
3972     \_tl_if_empty_if:o {#1}
3973     \prg_return_true:
```

```

3974     \else:
3975         \prg_return_false:
3976     \fi:
3977 }

```

(End definition for `\tl_if_empty:nTF` and `_tl_if_empty_if:o`. This function is documented on page 47.)

\tl_if_blank_p:n T_EX skips spaces when reading a non-delimited arguments. Thus, a *<token list>* is blank if and only if `\use_none:n <token list>` ? is empty after one expansion. The auxiliary `_tl_if_empty_if:o` is a fast emptiness test, converting its argument to a string (after one expansion) and using the test `\if:w \scan_stop: ... \scan_stop:`.

```

3978 \exp_args:Nno \use:n
3979 { \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF } }
3980 {
3981     \_tl_if_empty_if:o { \use_none:n #1 ? }
3982         \prg_return_true:
3983     \else:
3984         \prg_return_false:
3985     \fi:
3986 }
3987 \prg_generate_conditional_variant:Nnn \tl_if_blank:n
3988 { e , V , o } { p , T , F , TF }

```

(End definition for `\tl_if_blank:nTF` and `_tl_if_blank_p:NNw`. This function is documented on page 46.)

\tl_if_eq_p:NN Returns `\c_true_bool` if and only if the two token list variables are equal.

```

3989 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
3990 {
3991     \if_meaning:w #1 #2
3992         \prg_return_true:
3993     \else:
3994         \prg_return_false:
3995     \fi:
3996 }
3997 \prg_generate_conditional_variant:Nnn \tl_if_eq:NN
3998 { Nc , c , cc } { p , TF , T , F }

```

(End definition for `\tl_if_eq:NNTF`. This function is documented on page 47.)

`\l__tl_internal_a_tl` Temporary storage.

```

3999 \tl_new:N \l__tl_internal_a_tl
4000 \tl_new:N \l__tl_internal_b_tl

```

(End definition for `\l__tl_internal_a_tl` and `\l__tl_internal_b_tl`.)

\tl_if_eq:NnTF A simple store and compare routine.

```

4001 \prg_new_protected_conditional:Npnn \tl_if_eq:Nn #1#2 { T , F , TF }
4002 {
4003     \group_begin:
4004         \tl_set:Nn \l__tl_internal_b_tl {#2}
4005         \exp_after:wN
4006     \group_end:
4007         \if_meaning:w #1 \l__tl_internal_b_tl

```

```

4008     \prg_return_true:
4009     \else:
4010     \prg_return_false:
4011     \fi:
4012   }
4013 \prg_generate_conditional_variant:Nnn \tl_if_eq:Nn { c } { TF , T , F }

```

(End definition for `\tl_if_eq:NnTF`. This function is documented on page 47.)

\tl_if_eq:nnTF A simple store and compare routine.

```

4014 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
4015   {
4016     \group_begin:
4017     \tl_set:Nn \l__tl_internal_a_tl {\#1}
4018     \tl_set:Nn \l__tl_internal_b_tl {\#2}
4019     \exp_after:wN
4020     \group_end:
4021     \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
4022       \prg_return_true:
4023     \else:
4024       \prg_return_false:
4025     \fi:
4026   }

```

(End definition for `\tl_if_eq:nnTF`. This function is documented on page 47.)

\tl_if_in:NnTF See `\tl_if_in:nnTF` for further comments. Here we simply expand the token list variable `\tl_if_in:cnTF` and pass it to `\tl_if_in:nnTF`.

```

4027 \cs_new_protected:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
4028 \cs_new_protected:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
4029 \cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
4030 \prg_generate_conditional_variant:Nnn \tl_if_in:Nn
4031   { c } { T , F , TF }

```

(End definition for `\tl_if_in:NnTF`. This function is documented on page 47.)

\tl_if_in:nnTF Once more, the test relies on the emptiness test for robustness. The function `_tl_tmpt:w` removes tokens until the first occurrence of `#2`. If this does not appear in `#1`, then `\tl_if_in:onTF` the final `#2` is removed, leaving an empty token list. Otherwise some tokens remain, and `\tl_if_in:nTF` the test is `false`. See `\tl_if_empty:nTF` for details on the emptiness test.

Treating correctly cases like `\tl_if_in:nnTF {a state}{states}`, where `#1#2` contains `#2` before the end, requires special care. To cater for this case, we insert `\{}{\}` between the two token lists. This marker may not appear in `#2` because of TeX limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the test work also for empty arguments. The `\if_false:` constructions are a faster way to do `\group_align_safe_begin:` and `\group_align_safe_end:`. The `\scan_stop:` ensures that f-expanding `\tl_if_in:nn` does not lead to unbalanced braces.

```

4032 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
4033   {
4034     \scan_stop:
4035     \if_false: { \fi:
4036     \cs_set:Npn \_tl_tmpt:w ##1 #2 { }
4037     \tl_if_empty:OTF { \_tl_tmpt:w #1 {} {} #2 }
4038       { \prg_return_false: } { \prg_return_true: }

```

```

4039     \if_false: } \fi:
4040   }
4041 \prg_generate_conditional_variant:Nnn \tl_if_in:nn
4042   { V , o , no } { T , F , TF }

```

(End definition for `\tl_if_in:nnTF`. This function is documented on page 47.)

\tl_if_novalue_p:n Tests for `-NoValue-`: this is similar to `\tl_if_in:nn` but set up to be expandable and to check the value exactly. The question mark prevents the auxiliary from losing braces.

```

4043 \cs_set_protected:Npn \__tl_tmp:w #1
4044   {
4045     \prg_new_conditional:Npnn \tl_if_novalue:n ##1
4046       { p , T , F , TF }
4047     {
4048       \str_if_eq:onTF
4049         { \__tl_if_novalue:w ? ##1 { } #1 }
4050         { ? { } #1 }
4051         { \prg_return_true: }
4052         { \prg_return_false: }
4053     }
4054     \cs_new:Npn \__tl_if_novalue:w ##1 #1 {##1}
4055   }
4056 \exp_args:No \__tl_tmp:w { \c_novalue_tl }

```

(End definition for `\tl_if_novalue:nTF` and `__tl_if_novalue:w`. This function is documented on page 48.)

\tl_if_single_p:N Expand the token list and feed it to `\tl_if_single:n`.

```

4057 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
4058 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT }
4059 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF }
4060 \cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }

```

(End definition for `\tl_if_single:NTF`. This function is documented on page 48.)

\tl_if_single_p:n **\tl_if_single:nTF** **__tl_if_single:nnw** This test is similar to `\tl_if_empty:nTF`. Expanding `\use_none:nn #1 ??` once yields an empty result if `#1` is blank, a single `?` if `#1` has a single item, and otherwise yields some tokens ending with `??`. Then, `_kernel_tl_to_str:w` makes sure there are no odd category codes. An earlier version would compare the result to a single `?` using string comparison, but the Lua call is slow in LuaTeX. Instead, `__tl_if_single:nnw` picks the second token in front of it. If `#1` is empty, this token is the trailing `?` and the `\if:w` test yields `false`. If `#1` has a single item, the token is `\scan_stop:` and the `\if:w` test yields `true`. Otherwise, it is one of the characters resulting from `\tl_to_str:n`, and the `\if:w` test yields `false`. Note that `\if:w` and `_kernel_tl_to_str:w` are primitives that take care of expansion.

```

4061 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
4062   {
4063     \if:w \scan_stop: \exp_after:wN \__tl_if_single:nnw
4064       \_kernel_tl_to_str:w
4065       \exp_after:wN { \use_none:nn #1 ?? } \scan_stop: ? \s__tl_stop
4066     \prg_return_true:
4067   \else:
4068     \prg_return_false:
4069   \fi:

```

```

4070     }
4071 \cs_new:Npn \__tl_if_single:nw #1#2#3 \s__tl_stop {#2}

```

(End definition for `\tl_if_single:nTF` and `__tl_if_single:nw`. This function is documented on page 48.)

\tl_if_single_token_p:n There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. For the next case, an empty token list is not a single token. Finally, we have a non-empty token list starting with a space or a brace group. Applying f-expansion yields an empty result if and only if the token list is a single space.

```

4072 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
4073 {
4074     \tl_if_head_is_N_type:nTF {#1}
4075     { \__tl_if_empty_if:o { \use_none:n #1 } }
4076     {
4077         \tl_if_empty:nTF {#1}
4078         { \if_false: }
4079         { \__tl_if_empty_if:o { \exp:w \exp_end_continue_f:w #1 } }
4080     }
4081     \prg_return_true:
4082 \else:
4083     \prg_return_false:
4084 \fi:
4085 }

```

(End definition for `\tl_if_single_token:nTF`. This function is documented on page 48.)

\tl_case:Nn The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit “end of recursion” marker. That is achieved by using the test input as the final case, as this is always true. The trick is then to tidy up the output such that the appropriate case code plus either the `true` or `false` branch code is inserted.

```

4086 \cs_new:Npn \tl_case:Nn #1#2
4087 {
4088     \exp:w
4089     \__tl_case:NnTF #1 {#2} { } { }
4090 }
4091 \cs_new:Npn \tl_case:NnT #1#2#3
4092 {
4093     \exp:w
4094     \__tl_case:NnTF #1 {#2} {#3} { }
4095 }
4096 \cs_new:Npn \tl_case:NnF #1#2#3
4097 {
4098     \exp:w
4099     \__tl_case:NnTF #1 {#2} { } {#3}
4100 }
4101 \cs_new:Npn \tl_case:NnTF #1#2
4102 {
4103     \exp:w
4104     \__tl_case:NnTF #1 {#2}
4105 }
4106 \cs_new:Npn \__tl_case:NnTF #1#2#3#4

```

```

4107 { \__tl_case:Nw #1 #2 #1 { } \s__tl_mark {#3} \s__tl_mark {#4} \s__tl_stop }
4108 \cs_new:Npn \__tl_case:Nw #1#2#3
4109 {
4110     \tl_if_eq:NNTF #1 #2
4111     { \__tl_case_end:nw {#3} }
4112     { \__tl_case:Nw #1 }
4113 }
4114 \cs_generate_variant:Nn \tl_case:Nn { c }
4115 \prg_generate_conditional_variant:Nnn \tl_case:Nn
4116 { c } { T , F , TF }

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 is the code to insert, #2 is the next case to check on and #3 is all of the rest of the cases code. That means that #4 is the true branch code, and #5 tidies up the spare \s__tl_mark and the false branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that #1 is empty, #2 is the first \s__tl_mark and so #4 is the false code (the true code is mopped up by #3).

4117 \cs_new:Npn \__tl_case_end:nw #1#2#3 \s__tl_mark #4#5 \s__tl_stop
4118 { \exp_end: #1 #4 }

```

(End definition for \tl_case:NnTF and others. This function is documented on page 48.)

8.8 Mapping to token lists

\tl_map_function:nN
\tl_map_function:NN
\tl_map_function:cN

```

\__tl_map_function:Nn
4119 \cs_new:Npn \tl_map_function:nN #1#2
4120 {
4121     \__tl_map_function:Nn #2 #1
4122     \q_tl_recursion_tail
4123     \prg_break_point:Nn \tl_map_break: { }
4124 }
4125 \cs_new:Npn \tl_map_function:NN
4126 { \exp_args:No \tl_map_function:nN }
4127 \cs_new:Npn \__tl_map_function:Nn #1#2
4128 {
4129     \__tl_if_recursion_tail_break:nN {#2} \tl_map_break:
4130     #1 {#2} \__tl_map_function:Nn #1
4131 }
4132 \cs_generate_variant:Nn \tl_map_function:NN { c }

(End definition for \tl_map_function:nN, \tl_map_function:NN, and \__tl_map_function:Nn. These functions are documented on page 49.)
```

\tl_map_inline:nn
\tl_map_inline:Nn
\tl_map_inline:cn

The inline functions are straight forward by now. We use a little trick with the counter \g_kernel_prg_map_int to make them nestable. We can also make use of __tl_map_function:Nn from before.

```

4133 \cs_new_protected:Npn \tl_map_inline:nn #1#2
4134 {
4135     \int_gincr:N \g_kernel_prg_map_int
4136     \cs_gset_protected:cpn
4137     { __tl_map_ \int_use:N \g_kernel_prg_map_int :w } ##1 {#2}


```

```

4138     \exp_args:Nc \__tl_map_function:Nn
4139     { __tl_map_ \int_use:N \g_kernel_prg_map_int :w }
4140     #1 \q__tl_recursion_tail
4141     \prg_break_point:Nn \tl_map_break:
4142     { \int_gdecr:N \g_kernel_prg_map_int }
4143   }
4144 \cs_new_protected:Npn \tl_map_inline:Nn
4145   { \exp_args:No \tl_map_inline:nn }
4146 \cs_generate_variant:Nn \tl_map_inline:Nn { c }

```

(End definition for `\tl_map_inline:nn` and `\tl_map_inline:Nn`. These functions are documented on page 49.)

`\tl_map_tokens:nn` Much like the function mapping.

```

4147 \cs_new:Npn \tl_map_tokens:nn #1#2
4148   {
4149     \__tl_map_tokens:nn {#2} #1
4150     \q__tl_recursion_tail
4151     \prg_break_point:Nn \tl_map_break: { }
4152   }
4153 \cs_new:Npn \tl_map_tokens:Nn
4154   { \exp_args:No \tl_map_tokens:nn }
4155 \cs_generate_variant:Nn \tl_map_tokens:Nn { c }
4156 \cs_new:Npn \__tl_map_tokens:nn #1#2
4157   {
4158     \__tl_if_recursion_tail_break:nN {#2} \tl_map_break:
4159     \use:n {#1} {#2}
4160     \__tl_map_tokens:nn {#1}
4161   }

```

(End definition for `\tl_map_tokens:nn`, `\tl_map_tokens:Nn`, and `__tl_map_tokens:nn`. These functions are documented on page 49.)

`\tl_map_variable:nNn` `\tl_map_variable:nNn` *<token list>* *<tl var>* *<action>* assigns *<tl var>* to each element and executes *<action>*. The assignment to *<tl var>* is done after the quark test so that this variable does not get set to a quark.

```

4162 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
4163   {
4164     \__tl_map_variable:Nnn #2 {#3} #1
4165     \q__tl_recursion_tail
4166     \prg_break_point:Nn \tl_map_break: { }
4167   }
4168 \cs_new_protected:Npn \tl_map_variable:NNn
4169   { \exp_args:No \tl_map_variable:nNn }
4170 \cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
4171   {
4172     \__tl_if_recursion_tail_break:nN {#3} \tl_map_break:
4173     \tl_set:Nn #1 {#3}
4174     \use:n {#2}
4175     \__tl_map_variable:Nnn #1 {#2}
4176   }
4177 \cs_generate_variant:Nn \tl_map_variable:NNn { c }

```

(End definition for `\tl_map_variable:nNn`, `\tl_map_variable:NNn`, and `__tl_map_variable:Nnn`. These functions are documented on page 49.)

\tl_map_break: The break statements use the general \prg_map_break:Nn.

```
4178 \cs_new:Npn \tl_map_break:
4179   { \prg_map_break:Nn \tl_map_break: { } }
4180 \cs_new:Npn \tl_map_break:n
4181   { \prg_map_break:Nn \tl_map_break: }
```

(End definition for \tl_map_break: and \tl_map_break:n. These functions are documented on page 50.)

8.9 Using token lists

\tl_to_str:n Another name for a primitive: defined in l3basics.

```
4182 \cs_generate_variant:Nn \tl_to_str:n { V }
```

(End definition for \tl_to_str:n. This function is documented on page 51.)

\tl_to_str:N These functions return the replacement text of a token list as a string.

```
4183 \cs_new:Npn \tl_to_str:N #1 { \__kernel_tl_to_str:w \exp_after:wN {#1} }
4184 \cs_generate_variant:Nn \tl_to_str:N { c }
```

(End definition for \tl_to_str:N. This function is documented on page 51.)

\tl_use:N Token lists which are simply not defined give a clear TeX error here. No such luck for ones equal to \scan_stop: so instead a test is made and if there is an issue an error is forced.

```
4185 \cs_new:Npn \tl_use:N #1
4186   {
4187     \tl_if_exist:NTF #1 {#1}
4188     {
4189       \__kernel_msg_expandable_error:nnn
4190         { kernel } { bad-variable } {#1}
4191     }
4192   }
4193 \cs_generate_variant:Nn \tl_use:N { c }
```

(End definition for \tl_use:N. This function is documented on page 51.)

8.10 Working with the contents of token lists

\tl_count:n Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. Spaces are ignored. __tl_count:n grabs the element and replaces it by +1. The 0 ensures that it works on an empty list.

```
4194 \cs_new:Npn \tl_count:n #1
4195   {
4196     \int_eval:n
4197       { 0 \tl_map_function:nN {#1} \__tl_count:n }
4198   }
4199 \cs_new:Npn \tl_count:N #1
4200   {
4201     \int_eval:n
4202       { 0 \tl_map_function:NN #1 \__tl_count:n }
4203   }
4204 \cs_new:Npn \__tl_count:n #1 { + 1 }
4205 \cs_generate_variant:Nn \tl_count:n { V , o }
4206 \cs_generate_variant:Nn \tl_count:N { c }
```

(End definition for `\tl_count:n`, `\tl_count:N`, and `_tl_count:n`. These functions are documented on page 51.)

`\tl_count_tokens:n`

The token count is computed through an `\int_eval:n` construction. Each `1+` is output to the *left*, into the integer expression, and the sum is ended by the `\exp_end:` inserted by `_tl_act_end:wn` (which is technically implemented as `\c_zero_int`). Somewhat a hack!

```

4207 \cs_new:Npn \tl_count_tokens:n #1
4208 {
4209     \int_eval:n
4210     {
4211         \_tl_act:NNNn
4212         \_tl_act_count_normal:N
4213         \_tl_act_count_group:n
4214         \_tl_act_count_space:
4215         {#1}
4216     }
4217 }
4218 \cs_new:Npn \_tl_act_count_normal:N #1 { 1 + }
4219 \cs_new:Npn \_tl_act_count_space: { 1 + }
4220 \cs_new:Npn \_tl_act_count_group:n #1 { 2 + \tl_count_tokens:n {#1} + }
```

(End definition for `\tl_count_tokens:n` and others. This function is documented on page 52.)

`\tl_reverse_items:n`

Reversal of a token list is done by taking one item at a time and putting it after `\s__-tl_stop`.

```

4221 \cs_new:Npn \tl_reverse_items:n #1
4222 {
4223     \_tl_reverse_items:nwNwn #1 ?
4224     \s__tl_mark \_tl_reverse_items:nwNwn
4225     \s__tl_mark \_tl_reverse_items:wn
4226     \s__tl_stop { }
4227 }
4228 \cs_new:Npn \_tl_reverse_items:nwNwn #1 #2 \s__tl_mark #3 #4 \s__tl_stop #5
4229 {
4230     #3 #2
4231     \s__tl_mark \_tl_reverse_items:nwNwn
4232     \s__tl_mark \_tl_reverse_items:wn
4233     \s__tl_stop { {#1} #5 }
4234 }
4235 \cs_new:Npn \_tl_reverse_items:wn #1 \s__tl_stop #2
4236 { \_kernel_exp_not:w \exp_after:wN { \use_none:nn #2 } }
```

(End definition for `\tl_reverse_items:n`, `_tl_reverse_items:nwNwn`, and `_tl_reverse_items:wn`. This function is documented on page 52.)

`\tl_trim_spaces:n`

Trimming spaces from around the input is deferred to an internal function whose first argument is the token list to trim, augmented by an initial `_tl_trim_mark:`, and whose second argument is a *(continuation)*, which receives as a braced argument `_tl_trim_mark: <trimmed token list>`. The control sequence `_tl_trim_mark:` expands to nothing in a single expansion. In the case at hand, we take `_kernel_exp_not:w \exp_after:wN` as our continuation, so that space trimming behaves correctly within an `x`-type expansion.

```
4237 \cs_new:Npn \tl_trim_spaces:n #1
```

```

4238  {
4239    \_tl_trim_spaces:nn
4240    { \_tl_trim_mark: #1 }
4241    { \_kernel_exp_not:w \exp_after:wN }
4242  }
4243 \cs_generate_variant:Nn \tl_trim_spaces:n { o }
4244 \cs_new:Npn \tl_trim_spaces_apply:nN #1#2
4245  { \_tl_trim_spaces:nn { \_tl_trim_mark: #1 } { \exp_args:No #2 } }
4246 \cs_generate_variant:Nn \tl_trim_spaces_apply:nN { o }
4247 \cs_new_protected:Npn \tl_trim_spaces:N #1
4248  { \_kernel_tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
4249 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
4250  { \_kernel_tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
4251 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
4252 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }

```

Trimming spaces from around the input is done using delimited arguments and quarks, and to get spaces at odd places in the definitions, we nest those in `_tl_tmp:w`, which then receives a single space as its argument: `#1` is `_`. Removing leading spaces is done with `_tl_trim_spaces_auxi:w`, which loops until `_tl_trim_mark:_` matches the end of the token list: then `##1` is the token list and `##3` is `_tl_trim_spaces_auxii:w`. This hands the relevant tokens to the loop `_tl_trim_spaces_auxiii:w`, responsible for trimming trailing spaces. The end is reached when `_s__tl_nil` matches the one present in the definition of `\tl_trim_spaces:n`. Then `_tl_trim_spaces_auxiv:w` puts the token list into a group, with a lingering `_tl_trim_mark:` at the start (which will expand to nothing in one step of expansion), and feeds this to the *(continuation)*.

```

4253 \cs_set_protected:Npn \_tl_tmp:w #1
4254  {
4255    \cs_new:Npn \_tl_trim_spaces:nn ##1
4256    {
4257      \_tl_trim_spaces_auxi:w
4258      ##1
4259      _s__tl_nil
4260      \_tl_trim_mark: #1 { }
4261      \_tl_trim_mark: \_tl_trim_spaces_auxii:w
4262      \_tl_trim_spaces_auxiii:w
4263      #1 _s__tl_nil
4264      \_tl_trim_spaces_auxiv:w
4265      _s__tl_stop
4266    }
4267    \cs_new:Npn
4268      \_tl_trim_spaces_auxi:w ##1 \_tl_trim_mark: #1 ##2 \_tl_trim_mark: ##3
4269    {
4270      ##3
4271      \_tl_trim_spaces_auxi:w
4272      \_tl_trim_mark:
4273      ##2
4274      \_tl_trim_mark: #1 {##1}
4275    }
4276    \cs_new:Npn \_tl_trim_spaces_auxii:w
4277      \_tl_trim_spaces_auxi:w \_tl_trim_mark: \_tl_trim_mark: ##1
4278    {
4279      \_tl_trim_spaces_auxiii:w

```

```

4280      ##1
4281    }
4282 \cs_new:Npn \__tl_trim_spaces_auxiii:w ##1 #1 \s__tl_nil ##2
4283    {
4284      ##2
4285      ##1 \s__tl_nil
4286      \__tl_trim_spaces_auxiii:w
4287    }
4288 \cs_new:Npn \__tl_trim_spaces_auxiv:w ##1 \s__tl_nil ##2 \s__tl_stop ##3
4289  { ##3 { ##1 } }
4290 \cs_new:Npn \__tl_trim_mark: {}
4291 }
4292 \__tl_tmp:w { ~ }

```

(End definition for `\tl_trim_spaces:n` and others. These functions are documented on page 52.)

`\tl_sort:Nn` Implemented in `l3sort`.

`\tl_sort:cn`

`\tl_gsort:Nn`

`\tl_gsort:cn`

`\tl_sort:nN`

8.11 The first token from a token list

`\tl_head:N`

`\tl_head:n`

`\tl_head:V`

`\tl_head:v`

`\tl_head:f`

`__tl_head_auxi:nw`

`__tl_head_auxii:n`

`\tl_head:w`

`__tl_tl_head:w`

`\tl_tail:N`

`\tl_tail:n`

`\tl_tail:V`

`\tl_tail:v`

`\tl_tail:f`

Finding the head of a token list expandably always strips braces, which is fine as this is consistent with for example mapping to a list. The empty brace groups in `\tl_head:n` ensure that a blank argument gives an empty result. The result is returned within the `\unexpanded` primitive. The approach here is to use `\if_false:` to allow us to use `}` as the closing delimiter: this is the only safe choice, as any other token would not be able to parse it's own code. If the `\expanded` primitive is available it is used to get a fast and safe code variant in which we don't have to ensure that the left-most token is an internal to not break in an f-type expansion. If `\expanded` isn't available, using a marker, we can see if what we are grabbing is exactly the marker, or there is anything else to deal with. If there is, there is a loop. If not, tidy up and leave the item in the output stream. More detail in <http://tex.stackexchange.com/a/70168>.

```

4293 \cs_if_exist:NTF \tex_expanded:D
4294  {
4295    \cs_new:Npn \tl_head:n #1
4296    {
4297      \__kernel_exp_not:w \tex_expanded:D
4298      { { \if_false: { \fi: \__tl_head_auxi:nw #1 { } } } }
4299    }
4300    \cs_new:Npn \__tl_head_auxi:nw #1
4301    {
4302      \__kernel_exp_not:w {#1}
4303      \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
4304    }
4305  }
4306  {
4307    \cs_new:Npn \tl_head:n #1
4308    {
4309      \__kernel_exp_not:w
4310      \if_false: { \fi: \__tl_head_auxi:nw #1 { } \s__tl_stop }
4311    }
4312    \cs_new:Npn \__tl_head_auxi:nw #1#2 \s__tl_stop

```

```

4313 {
4314   \exp_after:wN \__tl_head_auxii:n \exp_after:wN {
4315     \if_false: } \fi: {#1}
4316 }
4317 \exp_args:Nno \use:n
4318   { \cs_new:Npn \__tl_head_auxii:n #1 }
4319   {
4320     \__tl_if_empty_if:o { \use_none:n #1 }
4321     \exp_after:wN \use_ii:nnn
4322     \fi:
4323     \use_ii:nn
4324     {#1}
4325     { \if_false: { \fi: \__tl_head_auxi:nw #1 } }
4326   }
4327 }
4328 \cs_generate_variant:Nn \tl_head:n { V , v , f }
4329 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
4330 \cs_new:Npn \__tl_tl_head:w #1#2 \s__tl_stop {#1}
4331 \cs_new:Npn \tl_head:N { \exp_args:No \tl_head:n }

```

To correctly leave the tail of a token list, it's important *not* to absorb any of the tail part as an argument. For example, the simple definition

```
\cs_new:Npn \tl_tail:n #1 { \tl_tail:w #1 \q_stop }
\cs_new:Npn \tl_tail:w #1#2 \q_stop
```

would give the wrong result for `\tl_tail:n { a { bc } }` (the braces would be stripped). Thus the only safe way to proceed is to first check that there is an item to grab (*i.e.* that the argument is not blank) and assuming there is to dispose of the first item. As with `\tl_head:n`, the result is protected from further expansion by `\unexpanded`. While we could optimise the test here, this would leave some tokens “banned” in the input, which we do not have with this definition.

```

4332 \exp_args:Nno \use:n { \cs_new:Npn \tl_tail:n #1 }
4333   {
4334     \exp_after:wN \__kernel_exp_not:w
4335     \tl_if_blank:nTF {#1}
4336     { { } }
4337     { \exp_after:wN { \use_none:n #1 } }
4338   }
4339 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
4340 \cs_new:Npn \tl_tail:N { \exp_args:No \tl_tail:n }

```

(End definition for `\tl_head:N` and others. These functions are documented on page 54.)

`\tl_if_head_eq_meaning_p:nN`
`\tl_if_head_eq_meaning:nNTF`
`\tl_if_head_eq_charcode_p:nN`
`\tl_if_head_eq_charcode:nNTF`
`\tl_if_head_eq_charcode_p:fN`
`\tl_if_head_eq_charcode:fNTF`
`\tl_if_head_eq_catcode_p:nN`
`\tl_if_head_eq_catcode:nNTF`
`\tl_if_head_eq_catcode_p:oN`
`\tl_if_head_eq_catcode:oNTF`
`__tl_head_exp_not:w`
`__tl_if_head_eq_empty_arg:w`

Accessing the first token of a token list is tricky in three cases: when it has category code 1 (begin-group token), when it is an explicit space, with category code 10 and character code 32, or when the token list is empty (obviously).

Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode:nN`. Here, `\tl_head:w` yields the first token of the token list, then passed to `\exp_not:N`.

```
\if_charcode:w
  \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop
  \exp_not:N #2
```

The two first special cases are detected by testing if the token list starts with an N-type token (the extra ? sends empty token lists to the `true` branch of this test). In those cases, the first token is a character, and since we only care about its character code, we can use `\str_head:n` to access it (this works even if it is a space character). An empty argument results in `\tl_head:w` leaving two tokens: ^ and `_tl_if_head_eq_empty_arg:w` which will result in the `\if_charcode:w` test being false and remove `\exp_not:N` and #2.

```

4341 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
4342   {
4343     \if_charcode:w
4344       \tl_if_head_is_N_type:nTF { #1 ? }
4345         { \_tl_head_exp_not:w #1 { ^ \_tl_if_head_eq_empty_arg:w } \s__tl_stop }
4346         { \str_head:n {#1} }
4347       \exp_not:N #2
4348     \prg_return_true:
4349   \else:
4350     \prg_return_false:
4351   \fi:
4352 }
4353 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_charcode:nN
4354   { f } { p , TF , T , F }

```

For `\tl_if_head_eq_catcode:nN`, again we detect special cases with a `\tl_if_head_is_N_type:n`. Then we need to test if the first token is a begin-group token or an explicit space token, and produce the relevant token, either `\c_group_begin_token` or `\c_space_token`. Again, for an empty argument, a hack is used, removing the token given by the user and leaving two tokens in the input stream which will make the `\if_catcode:w` test return `false`.

```

4355 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
4356   {
4357     \if_catcode:w
4358       \tl_if_head_is_N_type:nTF { #1 ? }
4359         { \_tl_head_exp_not:w #1 { ^ \_tl_if_head_eq_empty_arg:w } \s__tl_stop }
4360         {
4361           \tl_if_head_is_group:nTF {#1}
4362             \c_group_begin_token
4363             \c_space_token
4364           }
4365         \exp_not:N #2
4366       \prg_return_true:
4367     \else:
4368       \prg_return_false:
4369     \fi:
4370   }
4371 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_catcode:nN
4372   { o } { p , TF , T , F }

```

For `\tl_if_head_eq_meaning:nN`, again, detect special cases. In the normal case, use `\tl_head:w`, with no `\exp_not:N` this time, since `\if_meaning:w` causes no expansion. With an empty argument, the test is `true`, and `\use_none:nnn` removes #2 and `\prg_return_true:` and `\else:` (it is safe this way here as in this case `\prg_new_conditional:Npnn` didn't optimize these two away). In the special cases, we know that the first token is a character, hence `\if_charcode:w` and `\if_catcode:w` together are enough. We combine them in some order, hopefully faster than the reverse. Tests are

not nested because the arguments may contain unmatched primitive conditionals.

```

4373 \prg_new_if_meaning:nN #1#2 { p , T , F , TF }
4374 {
4375     \tl_if_head_is_N_type:nTF { #1 ? }
4376         \__tl_if_head_eq_meaning_normal:nN
4377         \__tl_if_head_eq_meaning_special:nN
4378     {#1} #2
4379 }
4380 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
4381 {
4382     \exp_after:wN \if_meaning:w
4383         \__tl_if_head:w #1 { ?? \use_none:nnn } \s__tl_stop #2
4384     \prg_return_true:
4385 \else:
4386     \prg_return_false:
4387 \fi:
4388 }
4389 \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
4390 {
4391     \if_charcode:w \str_head:n {#1} \exp_not:N #2
4392         \exp_after:wN \use_ii:nn
4393 \else:
4394     \prg_return_false:
4395 \fi:
4396 \use_none:n
4397 {
4398     \if_catcode:w \exp_not:N #2
4399         \tl_if_head_is_group:nTF {#1}
4400             { \c_group_begin_token }
4401             { \c_space_token }
4402         \prg_return_true:
4403 \else:
4404     \prg_return_false:
4405 \fi:
4406 }
4407 }

```

Both `\tl_if_head_eq_charcode:nN` and `\tl_if_head_eq_catcode:nN` will need to get the first token of their argument and apply `\exp_not:N` to it. `__tl_head_exp_not:w` does exactly that.

```

4408 \cs_new:Npn \__tl_head_exp_not:w #1 #2 \s__tl_stop
4409 { \exp_not:N #1 }

```

If the argument of `\tl_if_head_eq_charcode:nN` and `\tl_if_head_eq_catcode:nN` was empty `__tl_if_head_eq_empty_arg:w` will be left in the input stream. This macro has to remove `\exp_not:N` and the following token from the input stream to make sure no unbalanced if-construct is created and leave tokens there which make the two tests return `false`.

```

4410 \cs_new:Npn \__tl_if_head_eq_empty_arg:w \exp_not:N #1
4411 { ? }

```

(End definition for `\tl_if_head_eq_meaning:nNTF` and others. These functions are documented on page 55.)

```
\tl_if_head_is_N_type_p:n
\tl_if_head_is_N_type:nTF
  \_tl_if_head_is_N_type_auxi:w
  \_tl_if_head_is_N_type_auxii:nn
  \_tl_if_head_is_N_type_auxiii:n
```

A token list can be empty, can start with an explicit space character (catcode 10 and charcode 32), can start with a begin-group token (catcode 1), or start with an N-type argument. In the first two cases, the line involving `_tl_if_head_is_N_type_auxi:w` produces `f` (and otherwise nothing). In the third case (begin-group token), the lines involving `\token_to_str:N` produce a single closing brace. The category code test is thus true exactly in the fourth case, which is what we want. One cannot optimize by moving one of the `\scan_stop:` to the beginning: if #1 contains primitive conditionals, all of its occurrences must be dealt with before the `\if:w` tries to skip the `true` branch of the conditional.

```
4412 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
4413 {
4414   \if:w
4415     \if_false: { \fi: \_tl_if_head_is_N_type_auxi:w \prg_do_nothing: #1 ~ }
4416     { \exp_after:wN { \token_to_str:N #1 } }
4417     \scan_stop: \scan_stop:
4418     \prg_return_true:
4419   \else:
4420     \prg_return_false:
4421   \fi:
4422 }
4423 \exp_args:Nno \use:n { \cs_new:Npn \_tl_if_head_is_N_type_auxi:w #1 ~ }
4424 {
4425   \tl_if_empty:ctF { #1 }
4426   { f \exp_after:wN \use_none:nn }
4427   { \exp_after:wN \_tl_if_head_is_N_type_auxii:n }
4428   \exp_after:wN { \if_false: } \fi:
4429 }
4430 \cs_new:Npn \_tl_if_head_is_N_type_auxii:n #1
4431   { \exp_after:wN \use_none:n \exp_after:wN }
```

(End definition for `\tl_if_head_is_N_type:nTF` and others. This function is documented on page 55.)

```
\tl_if_head_is_group_p:n
\tl_if_head_is_group:nTF
  \_tl_if_head_is_group_fi_false:w
```

Pass the first token of #1 through `\token_to_str:N`, then check for the brace balance. The extra ? caters for an empty argument. This could be made faster, but we need all brace tricks to happen in one step of expansion, keeping the token list brace balanced at all times.

```
4432 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
4433 {
4434   \if:w
4435     \exp_after:wN \use_none:n
4436     \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
4437     \scan_stop: \scan_stop:
4438     \_tl_if_head_is_group_fi_false:w
4439   \fi:
4440   \if_true:
4441     \prg_return_true:
4442   \else:
4443     \prg_return_false:
4444   \fi:
4445 }
4446 \cs_new:Npn \_tl_if_head_is_group_fi_false:w \fi: \if_true: { \fi: \if_false: }
```

(End definition for `\tl_if_head_is_group:nTF` and `_tl_if_head_is_group_fi_false:w`. This function is documented on page 55.)

```
\tl_if_head_is_space_p:n
\tl_if_head_is_space:nTF
\__tl_if_head_is_space:w
```

The auxiliary's argument is all that is before the first explicit space in `\prg_do_nothing:#1?~`. If that is a single `\prg_do_nothing:` the test yields `true`. Otherwise, that is more than one token, and the test yields `false`. The work is done within braces (with an `\if_false: { \fi: ... }` construction) both to hide potential alignment tab characters from TeX in a table, and to allow for removing what remains of the token list after its first space. The use of `\if:w` ensures that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

```
4447 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
4448   {
4449     \if:w
4450       \if_false: { \fi: \__tl_if_head_is_space:w \prg_do_nothing: #1 ? ~ }
4451       \scan_stop: \scan_stop:
4452       \prg_return_true:
4453     \else:
4454       \prg_return_false:
4455     \fi:
4456   }
4457 \exp_args:Nno \use:n { \cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
4458   {
4459     \__tl_if_empty_if:o {#1} \else: f \fi:
4460     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
4461   }
```

(End definition for `\tl_if_head_is_space:nTF` and `__tl_if_head_is_space:w`. This function is documented on page 55.)

8.12 Token by token changes

```
\s__tl_act_stop
```

The `__tl_act_...` functions may be applied to any token list. Hence, we use a private quark, to allow any token, even quarks, in the token list. Only `\s__tl_act_stop` may not appear in the token lists manipulated by `__tl_act:NNNn` functions.

```
4462 \scan_new:N \s__tl_act_stop
```

(End definition for `\s__tl_act_stop`.)

```
\__tl_act:NNNn
```

```
\__tl_act_output:n
```

```
\__tl_act_reverse_output:n
```

```
\__tl_act_loop:w
```

```
\__tl_act_normal:NwNNN
```

```
\__tl_act_group:nwNNN
```

```
\__tl_act_space:wwNNN
```

```
\__tl_act_end:w
```

```
\__tl_act_if_head_is_space:nTF
```

```
\__tl_act_if_head_is_space:w
```

```
\__tl_act_if_head_is_space_true:w
```

```
\__tl_use_none_delimit_by_q_act_stop:w
```

To help control the expansion, `__tl_act:NNNn` should always be preceded by `\exp:w` and ends by producing `\exp_end:` once the result has been obtained. This way no internal token of it can be accidentally end up in the input stream. Because `\s__tl_act_stop` can't appear without braces around it in the argument `#1` of `__tl_act_loop:w`, we can use this marker to set up a fast test for leading spaces.

```
4463 \cs_set_protected:Npn \__tl_tmp:w #1
4464   {
4465     \cs_new:Npn \__tl_act_if_head_is_space:nTF ##1
4466     {
4467       \__tl_act_if_head_is_space:w
4468       \s__tl_act_stop ##1 \s__tl_act_stop \__tl_act_if_head_is_space_true:w
4469       \s__tl_act_stop #1 \s__tl_act_stop \use_ii:nn
4470     }
4471     \cs_new:Npn \__tl_act_if_head_is_space:w
4472       ##1 \s__tl_act_stop #1 ##2 \s__tl_act_stop
4473     {}
4474     \cs_new:Npn \__tl_act_if_head_is_space_true:w
4475       \s__tl_act_stop #1 \s__tl_act_stop \use_ii:nn ##1 ##2
```

```

4476      {##1}
4477  }
4478 \__tl_tmp:w { ~ }

(We expand the definition \__tl_act_if_head_is_space:nTF when setting up \__tl_act_loop:w, so we can then undefine the auxiliary.) In the loop, we check how the token list begins and act accordingly. In the “group” case, we may have reached \s__tl_act_stop, the end of the list. Then leave \exp_end: and the result in the input stream, to terminate the expansion of \exp:w. Otherwise, apply the relevant function to the “arguments”, #3 and to the head of the token list. Then repeat the loop. The scheme is the same if the token list starts with an N-type or with a space, making sure that \__tl_act_space:wwNNN gobbles the space.

4479 \exp_args:Nnx \use:n { \cs_new:Npn \__tl_act_loop:w #1 \s__tl_act_stop }
4480 {
4481   \exp_not:o { \__tl_act_if_head_is_space:nTF {#1} }
4482   \exp_not:N \__tl_act_space:wwNNN
4483   {
4484     \exp_not:o { \tl_if_head_is_group:nTF {#1} }
4485     \exp_not:N \__tl_act_group:nwNNN
4486     \exp_not:N \__tl_act_normal:NwNNN
4487   }
4488   \exp_not:n {#1} \s__tl_act_stop
4489 }
4490 \cs_undefine:N \__tl_act_if_head_is_space:nTF
4491 \cs_new:Npn \__tl_act_normal:NwNNN #1 #2 \s__tl_act_stop #3
4492 {
4493   #3 #1
4494   \__tl_act_loop:w #2 \s__tl_act_stop
4495   #3
4496 }
4497 \cs_new:Npn \__tl_use_none_delimit_by_s_act_stop:w #1 \s__tl_act_stop { }
4498 \cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
4499 { \group_align_safe_end: \exp_end: #2 }
4500 \cs_new:Npn \__tl_act_group:nwNNN #1 #2 \s__tl_act_stop #3#4#5
4501 {
4502   \__tl_use_none_delimit_by_s_act_stop:w #1 \__tl_act_end:wn \s__tl_act_stop
4503   #5 {#1}
4504   \__tl_act_loop:w #2 \s__tl_act_stop
4505   #3 #4 #5
4506 }
4507 \exp_last_unbraced:NNo
4508 \cs_new:Npn \__tl_act_space:wwNNN \c_space_tl #1 \s__tl_act_stop #2#3
4509 {
4510   #3
4511   \__tl_act_loop:w #1 \s__tl_act_stop
4512   #2 #3
4513 }

\__tl_act:NNNn loops over tokens, groups, and spaces in #4. {\s@act_stop} serves as the end of token list marker, the ? after it avoids losing outer braces. The result is stored as an argument for the dummy function \__tl_act_result:n.
```

```

4514 \cs_new:Npn \__tl_act:NNNn #1#2#3#4
4515 {
4516   \group_align_safe_begin:
```

```

4517     \_\_tl\_act\_loop:w #4 { \s\_\_tl\_act\_stop } ? \s\_\_tl\_act\_stop
4518     #1 #3 #2
4519     \_\_tl\_act\_result:n { }
4520   }

```

Typically, the output is done to the right of what was already output, using `__tl_act_output:n`, but for the `__tl_act_reverse` functions, it should be done to the left.

```

4521 \cs_new:Npn \_\_tl\_act\_output:n #1 #2 \_\_tl\_act\_result:n #3
4522   { #2 \_\_tl\_act\_result:n { #3 #1 } }
4523 \cs_new:Npn \_\_tl\_act\_reverse\_output:n #1 #2 \_\_tl\_act\_result:n #3
4524   { #2 \_\_tl\_act\_result:n { #1 #3 } }

```

(End definition for `__tl_act:NNNn` and others.)

\tl_reverse:n
\tl_reverse:o
\tl_reverse:v
`__tl_reverse_normal:nN`

```

4525 \cs_new:Npn \tl\_reverse:n #1
4526   {
4527     \_\_kernel\_exp\_not:w \exp\_after:wN
4528     {
4529       \exp:w
4530       \_\_tl\_act:NNNn
4531       \_\_tl\_reverse\_normal:N
4532       \_\_tl\_reverse\_group\_preserve:n
4533       \_\_tl\_reverse\_space:
4534       {#1}
4535     }
4536   }
4537 \cs_generate_variant:Nn \tl\_reverse:n { o , v }
4538 \cs_new:Npn \_\_tl\_reverse\_normal:N
4539   { \_\_tl\_act\_reverse\_output:n }
4540 \cs_new:Npn \_\_tl\_reverse\_group\_preserve:n #1
4541   { \_\_tl\_act\_reverse\_output:n { {#1} } }
4542 \cs_new:Npn \_\_tl\_reverse\_space:
4543   { \_\_tl\_act\_reverse\_output:n { ~ } }

```

(End definition for `\tl_reverse:n` and others. This function is documented on page 52.)

\tl_reverse:N This reverses the list, leaving `\exp_stop_f:` in front, which stops the f-expansion.

```

4544 \cs_new_protected:Npn \tl\_reverse:N #1
4545   { \_\_kernel\_tl\_set:Nx #1 { \exp_args:No \tl\_reverse:n { #1 } } }
4546 \cs_new_protected:Npn \tl\_greverse:N #1
4547   { \_\_kernel\_tl\_gset:Nx #1 { \exp_args:No \tl\_reverse:n { #1 } } }
4548 \cs_generate_variant:Nn \tl\_reverse:N { c }
4549 \cs_generate_variant:Nn \tl\_greverse:N { c }

```

(End definition for `\tl_reverse:N` and `\tl_greverse:N`. These functions are documented on page 52.)

8.13 Using a single item

`\tl_item:nn`
`\tl_item:Nn`
`\tl_item:cN`

The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then `_tl_if_recursion_tail_break:nN` terminates the loop, and returns nothing at all.

```

4550 \cs_new:Npn \tl_item:nn #1#2
4551 {
4552     \exp_args:Nf \_tl_item:nn
4553         { \exp_args:Nf \_tl_item_aux:nn { \int_eval:n {#2} } {#1} }
4554     #1
4555     \q_tl_recursion_tail
4556     \prg_break_point:
4557 }
4558 \cs_new:Npn \_tl_item_aux:nn #1#2
4559 {
4560     \int_compare:nNnTF {#1} < 0
4561         { \int_eval:n { \tl_count:n {#2} + 1 + #1 } }
4562     {#1}
4563 }
4564 \cs_new:Npn \_tl_item:nn #1#2
4565 {
4566     \_tl_if_recursion_tail_break:nN {#2} \prg_break:
4567     \int_compare:nNnTF {#1} = 1
4568         { \prg_break:n { \exp_not:n {#2} } }
4569         { \exp_args:Nf \_tl_item:nn { \int_eval:n { #1 - 1 } } }
4570 }
4571 \cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
4572 \cs_generate_variant:Nn \tl_item:Nn { c }
```

(End definition for `\tl_item:nn` and others. These functions are documented on page 56.)

`\tl_rand_item:n`
`\tl_rand_item:N`
`\tl_rand_item:c`

Importantly `\tl_item:nn` only evaluates its argument once.

```

4573 \cs_new:Npn \tl_rand_item:n #1
4574 {
4575     \tl_if_blank:nF {#1}
4576         { \tl_item:nn {#1} { \int_rand:nn { 1 } { \tl_count:n {#1} } } }
4577 }
4578 \cs_new:Npn \tl_rand_item:N { \exp_args:No \tl_rand_item:n }
4579 \cs_generate_variant:Nn \tl_rand_item:N { c }
```

(End definition for `\tl_rand_item:n` and `\tl_rand_item:N`. These functions are documented on page 56.)

`\tl_range:Nnn`

`\tl_range:cnn`

`\tl_range:nnn`

`_tl_range:Nnnn`

`_tl_range:nnnn`

`_tl_range:nnNn`

`_tl_range_skip:w`

`_tl_range:w`

`_tl_range_skip_spaces:n`

`_tl_range_collect:nn`

`_tl_range_collect:ff`

`_tl_range_collect_space:nw`

`_tl_range_collect_N:nN`

`_tl_range_collect_group:nN`

To avoid checking for the end of the token list at every step, start by counting the number l of items and “normalizing” the bounds, namely clamping them to the interval $[0, l]$ and dealing with negative indices. More precisely, `_tl_range_items:nnNn` receives the number of items to skip at the beginning of the token list, the index of the last item to keep, a function which is either `_tl_range:w` or the token list itself. If nothing should be kept, leave `{}`: this stops the f-expansion of `\tl_head:f` and that function produces an empty result. Otherwise, repeatedly call `_tl_range_skip:w` to delete `#1` items from the input stream (the extra brace group avoids an off-by-one shift). For the braced version `_tl_range_braced:w` sets up `_tl_range_collect_braced:w` which stores items one by one in an argument after the semicolon. Depending on the first token of the tail, either just move it (if it is a space) or also decrement the number of items left

to find. Eventually, the result is a brace group followed by the rest of the token list, and `\tl_head:f` cleans up and gives the result in `\exp_not:n`.

```

4580 \cs_new:Npn \tl_range:Nnn { \exp_args:No \tl_range:nnn }
4581   \cs_generate_variant:Nn \tl_range:Nnn { c }
4582 \cs_new:Npn \tl_range:nnn { \_tl_range:Nnnn \_tl_range:w }
4583 \cs_new:Npn \_tl_range:Nnnn #1#2#3#4
4584   {
4585     \tl_head:f
4586     {
4587       \exp_args:Nf \_tl_range:nnnNn
4588         { \tl_count:n {#2} } {#3} {#4} #1 {#2}
4589     }
4590   }
4591 \cs_new:Npn \_tl_range:nnnNn #1#2#3
4592   {
4593     \exp_args:Nff \_tl_range:nnNn
4594     {
4595       \exp_args:Nf \_tl_range_normalize:nn
4596         { \int_eval:n {#2 - 1} } {#1}
4597     }
4598   {
4599     \exp_args:Nf \_tl_range_normalize:nn
4600       { \int_eval:n {#3} } {#1}
4601   }
4602 }
4603 \cs_new:Npn \_tl_range:nnNn #1#2#3#4
4604   {
4605     \if_int_compare:w #2 > #1 \exp_stop_f: \else:
4606       \exp_after:wN { \exp_after:wN }
4607     \fi:
4608     \exp_after:wN #3
4609     \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
4610     \exp_after:wN { \exp:w \_tl_range_skip:w #1 ; { } #4 }
4611   }
4612 \cs_new:Npn \_tl_range_skip:w #1 ; #2
4613   {
4614     \if_int_compare:w #1 > 0 \exp_stop_f:
4615       \exp_after:wN \_tl_range_skip:w
4616         \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
4617     \else:
4618       \exp_after:wN \exp_end:
4619     \fi:
4620   }
4621 \cs_new:Npn \_tl_range:w #1 ; #2
4622   {
4623     \exp_args:Nf \_tl_range_collect:nn
4624       { \_tl_range_skip_spaces:n {#2} } {#1}
4625   }
4626 \cs_new:Npn \_tl_range_skip_spaces:n #1
4627   {
4628     \tl_if_head_is_space:nTF {#1}
4629       { \exp_args:Nf \_tl_range_skip_spaces:n {#1} }
4630         { { } #1 }
4631   }

```

```

4632 \cs_new:Npn \__tl_range_collect:nn #1#2
4633 {
4634     \int_compare:nNnTF {#2} = 0
4635     {#1}
4636     {
4637         \exp_args:No \tl_if_head_is_space:nTF { \use_none:n #1 }
4638         {
4639             \exp_args:Nf \__tl_range_collect:nn
4640             { \__tl_range_collect_space:nw #1 }
4641             {#2}
4642         }
4643         {
4644             \__tl_range_collect:ff
4645             {
4646                 \exp_args:No \tl_if_head_is_N_type:nTF { \use_none:n #1 }
4647                 { \__tl_range_collect_N:nN }
4648                 { \__tl_range_collect_group:nn }
4649                 #1
4650             }
4651             { \int_eval:n { #2 - 1 } }
4652         }
4653     }
4654 }
```

4655 \cs_new:Npn __tl_range_collect_space:nw #1 ~ { { #1 ~ } }

4656 \cs_new:Npn __tl_range_collect_N:nN #1#2 { { #1 #2 } }

4657 \cs_new:Npn __tl_range_collect_group:nn #1#2 { { #1 {#2} } }

4658 \cs_generate_variant:Nn __tl_range_collect:nn { ff }

(End definition for `\tl_range:Nnn` and others. These functions are documented on page 57.)

`__tl_range_normalize:nn` This function converts an `<index>` argument into an explicit position in the token list (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the `<index>` `#1` and the string count `#2`. If `#1` is negative, replace it by `#1 + #2 + 1`, then limit to the range $[0, #2]$.

```

4659 \cs_new:Npn \__tl_range_normalize:nn #1#2
4660 {
4661     \int_eval:n
4662     {
4663         \if_int_compare:w #1 < 0 \exp_stop_f:
4664         \if_int_compare:w #1 < -#2 \exp_stop_f:
4665             0
4666         \else:
4667             #1 + #2 + 1
4668         \fi:
4669     \else:
4670         \if_int_compare:w #1 < #2 \exp_stop_f:
4671             #1
4672         \else:
4673             #2
4674         \fi:
4675     }
4676 }
```

(End definition for `__tl_range_normalize:nn`.)

8.14 Viewing token lists

\tl_show:N Showing token list variables is done after checking that the variable is defined (see __-kernel_register_show:N).

```

\tl_show:c 4678 \cs_new_protected:Npn \tl_show:N { \__tl_show>NN \tl_show:n }
\tl_log:c 4679 \cs_generate_variant:Nn \tl_show:N { c }
\__tl_show>NN 4680 \cs_new_protected:Npn \tl_log:N { \__tl_show>NN \tl_log:n }
4681 \cs_generate_variant:Nn \tl_log:N { c }
4682 \cs_new_protected:Npn \__tl_show>NN #1#2
4683 {
4684     \__kernel_chk_defined:NT #2
4685     {
4686         \exp_args:Ne #1
4687         { \token_to_str:N #2 = \__kernel_exp_not:w \exp_after:wN {#2} }
4688     }
4689 }
```

(End definition for \tl_show:N, \tl_log:N, and __tl_show>NN. These functions are documented on page 58.)

\tl_show:n Many show functions are based on \tl_show:n. The argument of \tl_show:n is line-wrapped using \iow_wrap:nnN but with a leading >~ and trailing period, both removed before passing the wrapped text to the \showtokens primitive. This primitive shows the result with a leading >~ and trailing period.

The token list \l__tl_internal_a_tl containing the result of all these manipulations is displayed to the terminal using \tex_showtokens:D and an odd \exp_after:wN which expand the closing brace to improve the output slightly. The calls to __-kernel_iow_with:Nnn ensure that the \newlinechar is set to 10 so that the \iow_newline: inserted by the line-wrapping code are correctly recognized by TeX, and that \errorcontextlines is -1 to avoid printing irrelevant context.

```

4690 \cs_new_protected:Npn \tl_show:n #1
4691     { \iow_wrap:nnN { >~ \tl_to_str:n {#1} . } { } { } \__tl_show:n }
4692 \cs_new_protected:Npn \__tl_show:n #1
4693 {
4694     \tl_set:Nf \l__tl_internal_a_tl { \__tl_show:w #1 \s__tl_stop }
4695     \__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
4696     {
4697         \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
4698         {
4699             \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
4700             { \exp_after:wN \l__tl_internal_a_tl }
4701         }
4702     }
4703 }
4704 \cs_new:Npn \__tl_show:w #1 > #2 . \s__tl_stop {#2}
```

(End definition for \tl_show:n, __tl_show:n, and __tl_show:w. This function is documented on page 58.)

\tl_log:n Logging is much easier, simply line-wrap. The >~ and trailing period is there to match the output of \tl_show:n.

```

4705 \cs_new_protected:Npn \tl_log:n #1
4706     { \iow_wrap:nnN { >~ \tl_to_str:n {#1} . } { } { } \iow_log:n }
```

(End definition for \tl_log:n. This function is documented on page 58.)

8.15 Internal scan marks

```
\s__tl_nil Internal scan marks. These are defined here at the end because the code for \scan_new:N  
\s__tl_mark depends on some l3tl functions.  
\s__tl_stop 4707 \scan_new:N \s__tl_nil  
             4708 \scan_new:N \s__tl_mark  
             4709 \scan_new:N \s__tl_stop
```

(End definition for `\s__tl_nil`, `\s__tl_mark`, and `\s__tl_stop`.)

8.16 Scratch token lists

`\g_tmpa_tl` Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```
4710 \tl_new:N \g_tmpa_tl  
4711 \tl_new:N \g_tmpb_tl
```

(End definition for `\g_tmpa_tl` and `\g_tmpb_tl`. These variables are documented on page 59.)

`\l_tmpa_tl` These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.

```
4712 \tl_new:N \l_tmpa_tl  
4713 \tl_new:N \l_tmpb_tl
```

(End definition for `\l_tmpa_tl` and `\l_tmpb_tl`. These variables are documented on page 59.)

We finally clean up a temporary control sequence that we have used at various points to set up some definitions.

```
4714 \cs_undefine:N \__tl_tmp:w  
4715 
```

9 l3str implementation

```
4716 {*package}  
4717 @@@=str}
```

9.1 Internal auxiliaries

`\s__str_mark` Internal scan marks.
`\s__str_stop` 4718 `\scan_new:N \s__str_mark`
 4719 `\scan_new:N \s__str_stop`

(End definition for `\s__str_mark` and `\s__str_stop`.)

`_str_use_none_delimit_by_s_stop:w` Functions to gobble up to a scan mark.
`_str_use_i_delimit_by_s_stop:nw` 4720 `\cs_new:Npn _str_use_none_delimit_by_s_stop:w #1 \s__str_stop { }`
 4721 `\cs_new:Npn _str_use_i_delimit_by_s_stop:nw #1 #2 \s__str_stop {#1}`

(End definition for `_str_use_none_delimit_by_s_stop:w` and `_str_use_i_delimit_by_s_stop:nw`.)

`\q__str_recursion_tail` Internal recursion quarks.
`\q__str_recursion_stop` 4722 `\quark_new:N \q__str_recursion_tail`
 4723 `\quark_new:N \q__str_recursion_stop`

(End definition for `\q__str_recursion_tail` and `\q__str_recursion_stop`.)

```
\_str_if_recursion_tail_break:NN  
\_str_if_recursion_tail_stop_do:Nn
```

Functions to query recursion quarks.

```
4724 \__kernel_quark_new_test:N \_str_if_recursion_tail_break:NN  
4725 \__kernel_quark_new_test:N \_str_if_recursion_tail_stop_do:Nn
```

(End definition for `_str_if_recursion_tail_break>NN` and `_str_if_recursion_tail_stop_do:Nn`.)

9.2 Creating and setting string variables

`\str_new:N`

A string is simply a token list. The full mapping system isn't set up yet so do things by hand.

```
\str_use:N  
\str_use:c  
\str_clear:N  
\str_clear:c  
\str_gclear:N  
\str_gclear:c  
\str_clear_new:N  
\str_clear_new:c  
\str_gclear_new:N  
\str_gclear_new:c  
\str_set_eq:NN  
\str_set_eq:cN  
\str_set_eq:Nc  
\str_set_eq:cc  
\str_gset_eq:NN  
\str_gset_eq:cN  
\str_gset_eq:Nc  
\str_gset_eq:cc  
\str_concat:NNN  
\str_concat:ccc  
\str_gconcat:NNN  
\str_gconcat:ccc
```

```
4726 \group_begin:  
4727   \cs_set_protected:Npn \_str_tmp:n #1  
4728   {  
4729     \tl_if_blank:nF {#1}  
4730     {  
4731       \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }  
4732       \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }  
4733       \_str_tmp:n  
4734     }  
4735   }  
4736   \_str_tmp:n  
4737   { new }  
4738   { use }  
4739   { clear }  
4740   { gclear }  
4741   { clear_new }  
4742   { gclear_new }  
4743   { }  
4744 \group_end:  
4745 \cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN  
4746 \cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN  
4747 \cs_generate_variant:Nn \str_set_eq:NN { c , Nc , cc }  
4748 \cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }  
4749 \cs_new_eq:NN \str_concat:NNN \tl_concat:NNN  
4750 \cs_new_eq:NN \str_gconcat:NNN \tl_gconcat:NNN  
4751 \cs_generate_variant:Nn \str_concat:NNN { ccc }  
4752 \cs_generate_variant:Nn \str_gconcat:NNN { ccc }
```

(End definition for `\str_new:N` and others. These functions are documented on page 60.)

`\str_set:Nn`

Simply convert the token list inputs to `<strings>`.

```
\str_set:NV  
\str_set:Nx  
\str_set:cn  
\str_set:cV  
\str_set:cx  
\str_gset:Nn  
\str_gset:NV  
\str_gset:Nx  
\str_gset:cn  
\str_gset:cV  
\str_gset:cx  
\str_const:Nn  
\str_const:NV  
\str_const:Nx  
\str_const:cn  
\str_const:cV  
\str_const:cx  
\str_put_left:Nn  
\str_put_left:NV  
\str_put_left:Nx
```

```
4753 \group_begin:  
4754   \cs_set_protected:Npn \_str_tmp:n #1  
4755   {  
4756     \tl_if_blank:nF {#1}  
4757     {  
4758       \cs_new_protected:cpx { str_ #1 :Nn } ##1##2  
4759       {  
4760         \exp_not:c { tl_ #1 :Nx } ##1  
4761         { \exp_not:N \tl_to_str:n {##2} }  
4762       }  
4763       \cs_generate_variant:cn { str_ #1 :Nn } { NV , Nx , cn , cV , cx }  
4764     }  
4765   }  
4766 }
```

```

4764         \_\_str\_tmp:n
4765     }
4766 }
4767 \_\_str\_tmp:n
4768 { set }
4769 { gset }
4770 { const }
4771 { put_left }
4772 { gput_left }
4773 { put_right }
4774 { gput_right }
4775 { }
4776 \group_end:

```

(End definition for `\str_set:Nn` and others. These functions are documented on page 61.)

9.3 Modifying string variables

Start by applying `\tl_to_str:n` to convert the old and new token lists to strings, and also apply `\tl_to_str:N` to avoid any issues if we are fed a token list variable. Then the code is a much simplified version of the token list code because neither the delimiter nor the replacement can contain macro parameters or braces. The delimiter `\s__str_-mark` cannot appear in the string to edit so it is used in all cases. Some x-expansion is unnecessary. There is no need to avoid losing braces nor to protect against expansion. The ending code is much simplified and does not need to hide in braces.

```

4777 \cs_new_protected:Npn \str_replace_once:Nnn
4778 { \_\_str_replace:NNNnn \prg_do_nothing: \_\_kernel_tl_set:Nx }
4779 \cs_new_protected:Npn \str_greplace_once:Nnn
4780 { \_\_str_replace:NNNnn \prg_do_nothing: \_\_kernel_tl_gset:Nx }
4781 \cs_new_protected:Npn \str_replace_all:Nnn
4782 { \_\_str_replace:NNNnn \_\_str_replace_next:w \_\_kernel_tl_set:Nx }
4783 \cs_new_protected:Npn \str_greplace_all:Nnn
4784 { \_\_str_replace:NNNnn \_\_str_replace_next:w \_\_kernel_tl_gset:Nx }
4785 \cs_generate_variant:Nn \str_replace_once:Nnn { c }
4786 \cs_generate_variant:Nn \str_greplace_once:Nnn { c }
4787 \cs_generate_variant:Nn \str_replace_all:Nnn { c }
4788 \cs_generate_variant:Nn \str_greplace_all:Nnn { c }
4789 \cs_new_protected:Npn \_\_str_replace:NNNnn #1#2#3#4#5
4790 {
4791     \tl_if_empty:nTF {#4}
4792     {
4793         \_\_kernel_msg_error:nnx { kernel } { empty-search-pattern } {#5}
4794     }
4795     {
4796         \use:x
4797         {
4798             \exp_not:n { \_\_str_replace_aux:NNNnnn #1 #2 #3 }
4799             { \tl_to_str:N #3 }
4800             { \tl_to_str:n {#4} } { \tl_to_str:n {#5} }
4801         }
4802     }
4803 }
4804 \cs_new_protected:Npn \_\_str_replace_aux:NNNnnn #1#2#3#4#5#6

```

```

4805  {
4806    \cs_set:Npn \__str_replace_next:w ##1 #5 { ##1 #6 #1 }
4807    #2 #3
4808    {
4809      \__str_replace_next:w
4810      #4
4811      \__str_use_none_delimit_by_s_stop:w
4812      #5
4813      \s__str_stop
4814    }
4815  }
4816 \cs_new_eq:NN \__str_replace_next:w ?

```

(End definition for `\str_replace_all:Nnn` and others. These functions are documented on page 62.)

`\str_remove_once:Nn` Removal is just a special case of replacement.

```

\str_remove_once:cn
\str_gremove_once:Nn
\str_gremove_once:cn
4817 \cs_new_protected:Npn \str_remove_once:Nn #1#2
4818  { \str_replace_once:Nnn #1 {#2} { } }
4819 \cs_new_protected:Npn \str_gremove_once:Nn #1#2
4820  { \str_greplace_once:Nnn #1 {#2} { } }
4821 \cs_generate_variant:Nn \str_remove_once:Nn { c }
4822 \cs_generate_variant:Nn \str_gremove_once:Nn { c }

```

(End definition for `\str_remove_once:Nn` and `\str_gremove_once:Nn`. These functions are documented on page 62.)

`\str_remove_all:Nn` Removal is just a special case of replacement.

```

\str_remove_all:cn
\str_gremove_all:Nn
\str_gremove_all:cn
4823 \cs_new_protected:Npn \str_remove_all:Nn #1#2
4824  { \str_replace_all:Nnn #1 {#2} { } }
4825 \cs_new_protected:Npn \str_gremove_all:Nn #1#2
4826  { \str_greplace_all:Nnn #1 {#2} { } }
4827 \cs_generate_variant:Nn \str_remove_all:Nn { c }
4828 \cs_generate_variant:Nn \str_gremove_all:Nn { c }

```

(End definition for `\str_remove_all:Nn` and `\str_gremove_all:Nn`. These functions are documented on page 62.)

9.4 String comparisons

`\str_if_empty_p:N` More copy-paste!

```

\str_if_empty_p:c
\str_if_empty_p:NTF
\str_if_empty:cTF
\str_if_exist_p:N
\str_if_exist_p:c
\str_if_exist:NTF
\str_if_exist:cTF
4829 \prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N
4830  { p , T , F , TF }
4831 \prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c
4832  { p , T , F , TF }
4833 \prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N
4834  { p , T , F , TF }
4835 \prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c
4836  { p , T , F , TF }

```

(End definition for `\str_if_empty:NTF` and `\str_if_exist:NTF`. These functions are documented on page 63.)

`__str_if_eq:nn` String comparisons rely on the primitive `\(pdf)strcmp`, so we define a new name for it.

```

4837 \cs_new_eq:NN \__str_if_eq:nn \tex_strcmp:D

```

(End definition for `_str_if_eq:nn`.)

\str_if_eq_p:nn Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore make life a bit clearer. The nn and xx versions are created directly as this is most efficient.

```
4838 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
4839   {
4840     \if_int_compare:w
4841       \_str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
4842       = 0 \exp_stop_f:
4843       \prg_return_true: \else: \prg_return_false: \fi:
4844   }
4845 \prg_generate_conditional_variant:Nnn \str_if_eq:nn
4846   { V , v , o , nV , no , VV , nv } { p , T , F , TF }
4847 \prg_new_conditional:Npnn \str_if_eq:ee #1#2 { p , T , F , TF }
4848   {
4849     \if_int_compare:w \_str_if_eq:nn {#1} {#2} = 0 \exp_stop_f:
4850       \prg_return_true: \else: \prg_return_false: \fi:
4851   }
```

(End definition for `\str_if_eq:nnTF`. This function is documented on page 63.)

\str_if_eq_p:NN Note that `\str_if_eq:NN` is different from `\tl_if_eq:NN` because it needs to ignore category codes.

```
4852 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
4853   {
4854     \if_int_compare:w
4855       \_str_if_eq:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }
4856       = 0 \exp_stop_f: \prg_return_true: \else: \prg_return_false: \fi:
4857   }
4858 \prg_generate_conditional_variant:Nnn \str_if_eq:NN
4859   { c , Nc , cc } { T , F , TF , p }
```

(End definition for `\str_if_eq:NNTF`. This function is documented on page 63.)

\str_if_in:NnTF Everything here needs to be detokenized but beyond that it is a simple token list test.
\str_if_in:cnTF It would be faster to fine-tune the T, F, TF variants by calling the appropriate variant of
\str_if_in:nnTF `\tl_if_in:nnTF` directly but that takes more code.

```
4860 \prg_new_protected_conditional:Npnn \str_if_in:Nn #1#2 { T , F , TF }
4861   {
4862     \use:x
4863       { \tl_if_in:nnTF { \tl_to_str:N #1 } { \tl_to_str:n {#2} } }
4864       { \prg_return_true: } { \prg_return_false: }
4865   }
4866 \prg_generate_conditional_variant:Nnn \str_if_in:Nn
4867   { c } { T , F , TF }
4868 \prg_new_protected_conditional:Npnn \str_if_in:nn #1#2 { T , F , TF }
4869   {
4870     \use:x
4871       { \tl_if_in:nnTF { \tl_to_str:n {#1} } { \tl_to_str:n {#2} } }
4872       { \prg_return_true: } { \prg_return_false: }
4873   }
```

(End definition for `\str_if_in:NnTF` and `\str_if_in:nnTF`. These functions are documented on page 63.)

```

\str_case:nn Much the same as \tl_case:nn(TF) here: just a change in the internal comparison.
\str_case:Vn
4874 \cs_new:Npn \str_case:nn #1#2
\str_case:on
4875 {
\str_case:nV
4876     \exp:w
\str_case:nv
4877     \__str_case:nnTF {#1} {#2} { } { }
\str_case:nnTF
4878 }
\str_case:VnTF
4879 \cs_new:Npn \str_case:nnT #1#2#3
\str_case:onTF
4880 {
\str_case:nVTF
4881     \exp:w
\str_case:nvTF
4882     \__str_case:nnTF {#1} {#2} {#3} { }
\str_case_e:nn
4883 }
\str_case_e:nnTF
4884 \cs_new:Npn \str_case:nnF #1#2
\str_case_e:nnTF
4885 {
\__str_case:nnTF
4886     \exp:w
\__str_case_e:nnTF
4887     \__str_case:nnTF {#1} {#2} { }
\__str_case:nw
4888 }
\__str_case_e:nw
4889 \cs_new:Npn \str_case:nnTF #1#2
\__str_case_end:nw
4890 {
4891     \exp:w
4892     \__str_case:nnTF {#1} {#2}
4893 }
4894 \cs_new:Npn \__str_case:nnTF #1#2#3#4
4895     { \__str_case:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }
4896 \cs_generate_variant:Nn \str_case:nn { V , o , nV , nv }
4897 \prg_generate_conditional_variant:Nnn \str_case:nn
4898     { V , o , nV , nv } { T , F , TF }
4899 \cs_new:Npn \__str_case:nw #1#2#3
4900 {
4901     \str_if_eq:nnTF {#1} {#2}
4902     { \__str_case_end:nw {#3} }
4903     { \__str_case:nw {#1} }
4904 }
4905 \cs_new:Npn \str_case_e:nn #1#2
4906 {
4907     \exp:w
4908     \__str_case_e:nnTF {#1} {#2} { } { }
4909 }
4910 \cs_new:Npn \str_case_e:nnT #1#2#3
4911 {
4912     \exp:w
4913     \__str_case_e:nnTF {#1} {#2} {#3} { }
4914 }
4915 \cs_new:Npn \str_case_e:nnF #1#2
4916 {
4917     \exp:w
4918     \__str_case_e:nnTF {#1} {#2} { }
4919 }
4920 \cs_new:Npn \str_case_e:nnTF #1#2
4921 {
4922     \exp:w
4923     \__str_case_e:nnTF {#1} {#2}
4924 }
4925 \cs_new:Npn \__str_case_e:nnTF #1#2#3#4
4926     { \__str_case_e:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }

```

```

4927 \cs_new:Npn \__str_case_e:nw #1#2#3
4928 {
4929     \str_if_eq:eeTF {#1} {#2}
4930     { \__str_case_end:nw {#3} }
4931     { \__str_case_e:nw {#1} }
4932 }
4933 \cs_new:Npn \__str_case_end:nw #1#2#3 \s__str_mark #4#5 \s__str_stop
4934 { \exp_end: #1 #4 }

```

(End definition for `\str_case:nnTF` and others. These functions are documented on page 64.)

9.5 Mapping to strings

```

\str_map_function:NN
\str_map_function:cN
\str_map_function:nN
\str_map_inline:Nn
\str_map_inline:cn
\str_map_inline:nn
\str_map_variable:NNn
\str_map_variable:cNn
\str_map_variable:nNn
    \str_map_break:
    \str_map_break:n
\__str_map_function:w
\__str_map_function:Nn
    \__str_map_inline>NN
\__str_map_variable:Nn

```

The inline and variable mappings are similar to the usual token list mappings but start out by turning the argument to an “other string”. Doing the same for the expandable function mapping would require `__kernel_str_to_other:n`, quadratic in the string length. To deal with spaces in that case, `__str_map_function:w` replaces the following space by a braced space and a further call to itself. These are received by `__str_map_function:Nn`, which passes the space to #1 and calls `__str_map_function:w` to deal with the next space. The space before the braced space allows to optimize the `\q__str_recursion_tail` test. Of course we need to include a trailing space (the question mark is needed to avoid losing the space when TeX tokenizes the line). At the cost of about three more auxiliaries this code could get a 9 times speed up by testing only every 9-th character for whether it is `\q__str_recursion_tail` (also by converting 9 spaces at a time in the `\str_map_function:nN` case).

For the `map_variable` functions we use a string assignment to store each character because spaces are made catcode 12 before the loop.

```

4935 \cs_new:Npn \str_map_function:nN #1#2
4936 {
4937     \exp_after:wN \__str_map_function:w
4938     \exp_after:wN \__str_map_function:Nn \exp_after:wn #2
4939     \__kernel_tl_to_str:w {#1}
4940     \q__str_recursion_tail ? ~
4941     \prg_break_point:Nn \str_map_break: { }
4942 }
4943 \cs_new:Npn \str_map_function:NN
4944 { \exp_args:No \str_map_function:nN }
4945 \cs_new:Npn \__str_map_function:w #1 ~
4946 { #1 { ~ { ~ } \__str_map_function:w } }
4947 \cs_new:Npn \__str_map_function:Nn #1#2
4948 {
4949     \if_meaning:w \q__str_recursion_tail #2
4950     \exp_after:wN \str_map_break:
4951     \fi:
4952     #1 #2 \__str_map_function:Nn #
4953 }
4954 \cs_generate_variant:Nn \str_map_function:NN { c }
4955 \cs_new_protected:Npn \str_map_inline:nn #1#2
4956 {
4957     \int_gincr:N \g__kernel_prg_map_int
4958     \cs_gset_protected:cpn
4959     { __str_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
4960     \use:x

```

```

4961   {
4962     \exp_not:N \__str_map_inline:NN
4963     \exp_not:c { __str_map_ \int_use:N \g_kernel_prg_map_int :w }
4964     \__kernel_str_to_other_fast:n {#1}
4965   }
4966   \q_str_recursion_tail
4967   \prg_break_point:Nn \str_map_break:
4968   { \int_gdecr:N \g_kernel_prg_map_int }
4969 }
4970 \cs_new_protected:Npn \str_map_inline:Nn
4971   { \exp_args:No \str_map_inline:nn }
4972 \cs_generate_variant:Nn \str_map_inline:Nn { c }
4973 \cs_new:Npn \__str_map_inline:NN #1#2
4974 {
4975   \__str_if_recursion_tail_break:NN #2 \str_map_break:
4976   \exp_args:No #1 { \token_to_str:N #2 }
4977   \__str_map_inline:NN #1
4978 }
4979 \cs_new_protected:Npn \str_map_variable:nNn #1#2#3
4980 {
4981   \use:x
4982   {
4983     \exp_not:n { \__str_map_variable:NnN #2 {#3} }
4984     \__kernel_str_to_other_fast:n {#1}
4985   }
4986   \q_str_recursion_tail
4987   \prg_break_point:Nn \str_map_break: { }
4988 }
4989 \cs_new_protected:Npn \str_map_variable:NNn
4990   { \exp_args:No \str_map_variable:nNn }
4991 \cs_new_protected:Npn \__str_map_variable:NnN #1#2#3
4992 {
4993   \__str_if_recursion_tail_break:NN #3 \str_map_break:
4994   \str_set:Nn #1 {#3}
4995   \use:n {#2}
4996   \__str_map_variable:NnN #1 {#2}
4997 }
4998 \cs_generate_variant:Nn \str_map_variable:NNn { c }
4999 \cs_new:Npn \str_map_break:
5000   { \prg_map_break:Nn \str_map_break: { } }
5001 \cs_new:Npn \str_map_break:n
5002   { \prg_map_break:Nn \str_map_break: }

```

(End definition for `\str_map_function:NN` and others. These functions are documented on page 64.)

9.6 Accessing specific characters in a string

`__kernel_str_to_other:n` First apply `\tl_to_str:n`, then replace all spaces by “other” spaces, 8 at a time, storing the converted part of the string between the `\s__str_mark` and `\s__str_stop` markers. The end is detected when `__str_to_other_loop:w` finds one of the trailing A, distinguished from any contents of the initial token list by their category. Then `__str_to_other_end:w` is called, and finds the result between `\s__str_mark` and the first A (well, there is also the need to remove a space).

```

5003 \cs_new:Npn \__kernel_str_to_other:n #1
5004 {
5005     \exp_after:wN \__str_to_other_loop:w
5006         \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_mark \s__str_stop
5007     }
5008 \group_begin:
5009 \tex_lccode:D '\* = '\ %
5010 \tex_lccode:D '\A = '\A %
5011 \tex_lowercase:D
5012 {
5013     \group_end:
5014     \cs_new:Npn \__str_to_other_loop:w
5015         #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \s__str_stop
5016     {
5017         \if_meaning:w A #8
5018             \__str_to_other_end:w
5019         \fi:
5020         \__str_to_other_loop:w
5021             #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \s__str_stop
5022         }
5023     \cs_new:Npn \__str_to_other_end:w \fi: #1 \s__str_mark #2 * A #3 \s__str_stop
5024     { \fi: #2 }
5025 }

```

(End definition for `__kernel_str_to_other:n`, `__str_to_other_loop:w`, and `__str_to_other_end:w`.)

The difference with `__kernel_str_to_other:n` is that the converted part is left in the input stream, making these commands only restricted-expandable.

```

5026 \cs_new:Npn \__kernel_str_to_other_fast:n #1
5027 {
5028     \exp_after:wN \__str_to_other_fast_loop:w \tl_to_str:n {#1} ~
5029         A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_stop
5030     }
5031 \group_begin:
5032 \tex_lccode:D '\* = '\ %
5033 \tex_lccode:D '\A = '\A %
5034 \tex_lowercase:D
5035 {
5036     \group_end:
5037     \cs_new:Npn \__str_to_other_fast_loop:w
5038         #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
5039     {
5040         \if_meaning:w A #9
5041             \__str_to_other_fast_end:w
5042         \fi:
5043             #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
5044             \__str_to_other_fast_loop:w *
5045     }
5046     \cs_new:Npn \__str_to_other_fast_end:w #1 * A #2 \s__str_stop {#1}
5047 }

```

(End definition for `__kernel_str_to_other_fast:n`, `__kernel_str_to_other_fast_loop:w`, and `__str_to_other_fast_end:w`.)

```
\str_item:Nn
\str_item:cn
\str_item:nn
\str_item_ignore_spaces:nn
\__str_item:nn
\__str_item:w
```

The `\str_item:nn` hands its argument with spaces escaped to `__str_item:nn`, and makes sure to turn the result back into a proper string (with category code 10 spaces) eventually. The `\str_item_ignore_spaces:nn` function does not escape spaces, which are thus ignored by `__str_item:nn` since everything else is done with undelimited arguments. Evaluate the `(index)` argument #2 and count characters in the string, passing those two numbers to `__str_item:w` for further analysis. If the `(index)` is negative, shift it by the `(count)` to know the how many character to discard, and if that is still negative give an empty result. If the `(index)` is larger than the `(count)`, give an empty result, and otherwise discard `(index) - 1` characters before returning the following one. The shift by `-1` is obtained by inserting an empty brace group before the string in that case: that brace group also covers the case where the `(index)` is zero.

```
5048 \cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }
5049 \cs_generate_variant:Nn \str_item:Nn { c }
5050 \cs_new:Npn \str_item:nn #1#2
5051 {
5052     \exp_args:Nf \tl_to_str:n
5053     {
5054         \exp_args:Nf \__str_item:nn
5055         { \__kernel_str_to_other:n {#1} } {#2}
5056     }
5057 }
5058 \cs_new:Npn \str_item_ignore_spaces:nn #1
5059     { \exp_args:No \__str_item:nn { \tl_to_str:n {#1} } }
5060 \cs_new:Npn \__str_item:nn #1#2
5061 {
5062     \exp_after:wN \__str_item:w
5063     \int_value:w \int_eval:n {#2} \exp_after:wN ;
5064     \int_value:w \__str_count:n {#1} ;
5065     #1 \s__str_stop
5066 }
5067 \cs_new:Npn \__str_item:w #1; #2;
5068 {
5069     \int_compare:nNnTF {#1} < 0
5070     {
5071         \int_compare:nNnTF {#1} < {-#2}
5072             { \__str_use_none_delimit_by_s_stop:w }
5073             {
5074                 \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
5075                 \exp:w \exp_after:wN \__str_skip_exp_end:w
5076                     \int_value:w \int_eval:n { #1 + #2 } ;
5077             }
5078     }
5079     {
5080         \int_compare:nNnTF {#1} > {#2}
5081             { \__str_use_none_delimit_by_s_stop:w }
5082             {
5083                 \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
5084                 \exp:w \__str_skip_exp_end:w #1 ; { }
5085             }
5086     }
5087 }
```

(End definition for `\str_item:Nn` and others. These functions are documented on page 67.)

```

\__str_skip_exp_end:w
\__str_skip_loop:wNNNNNNNNN
\__str_skip_end:w
\__str_skip_end>NNNNNNNN
```

Removes `max(#1,0)` characters from the input stream, and then leaves `\exp_end:`. This should be expanded using `\exp:w`. We remove characters 8 at a time until there are at most 8 to remove. Then we do a dirty trick: the `\if_case:w` construction leaves between 0 and 8 times the `\or:` control sequence, and those `\or:` become arguments of `__str_skip_end>NNNNNNNN`. If the number of characters to remove is 6, say, then there are two `\or:` left, and the 8 arguments of `__str_skip_end>NNNNNNNN` are the two `\or:`, and 6 characters from the input stream, exactly what we wanted to remove. Then close the `\if_case:w` conditional with `\fi:`, and stop the initial expansion with `\exp_end:` (see places where `__str_skip_exp_end:w` is called).

```

5088 \cs_new:Npn \__str_skip_exp_end:w #1;
5089 {
5090     \if_int_compare:w #1 > 8 \exp_stop_f:
5091         \exp_after:wN \__str_skip_loop:wNNNNNNNN
5092     \else:
5093         \exp_after:wN \__str_skip_end:w
5094         \int_value:w \int_eval:w
5095     \fi:
5096     #1 ;
5097 }
5098 \cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
5099 {
5100     \exp_after:wN \__str_skip_exp_end:w
5101         \int_value:w \int_eval:n { #1 - 8 } ;
5102 }
5103 \cs_new:Npn \__str_skip_end:w #1 ;
5104 {
5105     \exp_after:wN \__str_skip_end>NNNNNNNN
5106     \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or:
5107 }
5108 \cs_new:Npn \__str_skip_end>NNNNNNNN #1#2#3#4#5#6#7#8 { \fi: \exp_end: }
```

(End definition for `__str_skip_exp_end:w` and others.)

```

\str_range:Nnn
\str_range:nnn
\str_range_ignore_spaces:nnn
\__str_range:nnn
\__str_range:w
\__str_range:nnw
```

Sanitize the string. Then evaluate the arguments. At this stage we also decrement the `<start index>`, since our goal is to know how many characters should be removed. Then limit the range to be non-negative and at most the length of the string (this avoids needing to check for the end of the string when grabbing characters), shifting negative numbers by the appropriate amount. Afterwards, skip characters, then keep some more, and finally drop the end of the string.

```

5109 \cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
5110 \cs_generate_variant:Nn \str_range:Nnn { c }
5111 \cs_new:Npn \str_range:nnn #1#2#3
5112 {
5113     \exp_args:Nf \tl_to_str:n
5114     {
5115         \exp_args:Nf \__str_range:nnn
5116             { \__kernel_str_to_other:n {#1} } {#2} {#3}
5117     }
5118 }
5119 \cs_new:Npn \str_range_ignore_spaces:nnn #1
5120     { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
5121 \cs_new:Npn \__str_range:nnn #1#2#3
5122 {
```

```

5123     \exp_after:wN \__str_range:w
5124     \int_value:w \__str_count:n {#1} \exp_after:wN ;
5125     \int_value:w \int_eval:n { (#2) - 1 } \exp_after:wN ;
5126     \int_value:w \int_eval:n {#3} ;
5127     #1 \s__str_stop
5128   }
5129 \cs_new:Npn \__str_range:w #1; #2; #3;
5130   {
5131     \exp_args:Nf \__str_range:nnw
5132     { \__str_range_normalize:nn {#2} {#1} }
5133     { \__str_range_normalize:nn {#3} {#1} }
5134   }
5135 \cs_new:Npn \__str_range:nnw #1#2
5136   {
5137     \exp_after:wN \__str_collect_delimit_by_q_stop:w
5138     \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
5139     \exp:w \__str_skip_exp_end:w #1 ;
5140   }

```

(End definition for `\str_range:Nnn` and others. These functions are documented on page 68.)

`__str_range_normalize:nn` This function converts an `<index>` argument into an explicit position in the string (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the `<index>` #1 and the string count #2. If #1 is negative, replace it by `#1 + #2 + 1`, then limit to the range [0, #2].

```

5141 \cs_new:Npn \__str_range_normalize:nn #1#2
5142   {
5143     \int_eval:n
5144     {
5145       \if_int_compare:w #1 < 0 \exp_stop_f:
5146         \if_int_compare:w #1 < -#2 \exp_stop_f:
5147           0
5148         \else:
5149           #1 + #2 + 1
5150         \fi:
5151       \else:
5152         \if_int_compare:w #1 < #2 \exp_stop_f:
5153           #1
5154         \else:
5155           #2
5156         \fi:
5157       \fi:
5158     }
5159   }

```

(End definition for `__str_range_normalize:nn`.)

`__str_collect_delimit_by_q_stop:w`
`__str_collect_loop:wnNNNNNNNN`
`__str_collect_end:wn`
`__str_collect_end:nnnnnnnnnw` Collects `max(#1,0)` characters, and removes everything else until `\s__str_stop`. This is somewhat similar to `__str_skip_exp_end:w`, but accepts integer expression arguments. This time we can only grab 7 characters at a time. At the end, we use an `\if_case:w` trick again, so that the 8 first arguments of `__str_collect_end:nnnnnnnnnw` are some `\or:`, followed by an `\fi:`, followed by #1 characters from the input stream. Simply leaving this in the input stream closes the conditional properly and the `\or:` disappear.

```

5160 \cs_new:Npn \__str_collect_delimit_by_q_stop:w #1;

```

```

5161 { \__str_collect_loop:wn #1 ; { } }
5162 \cs_new:Npn \__str_collect_loop:wn #1 ;
5163 {
5164     \if_int_compare:w #1 > 7 \exp_stop_f:
5165         \exp_after:wN \__str_collect_loop:wnNNNNNNNN
5166     \else:
5167         \exp_after:wN \__str_collect_end:wn
5168     \fi:
5169     #1 ;
5170 }
5171 \cs_new:Npn \__str_collect_loop:wnNNNNNNNN #1; #2 #3#4#5#6#7#8#9
5172 {
5173     \exp_after:wN \__str_collect_loop:wn
5174     \int_value:w \int_eval:n { #1 - 7 } ;
5175     { #2 #3#4#5#6#7#8#9 }
5176 }
5177 \cs_new:Npn \__str_collect_end:wn #1 ;
5178 {
5179     \exp_after:wN \__str_collect_end:nnnnnnnnw
5180     \if_case:w \if_int_compare:w #1 > 0 \exp_stop_f:
5181         #1 \else: 0 \fi: \exp_stop_f:
5182         \or: \or: \or: \or: \or: \or: \or: \or: \fi:
5183 }
5184 \cs_new:Npn \__str_collect_end:nnnnnnnnw #1#2#3#4#5#6#7#8 #9 \s__str_stop
5185 { #1#2#3#4#5#6#7#8 }

```

(End definition for `__str_collect_delimit_by_q_stop:w` and others.)

9.7 Counting characters

`\str_count_spaces:N`
`\str_count_spaces:c`
`\str_count_spaces:n`

To speed up this function, we grab and discard 9 space-delimited arguments in each iteration of the loop. The loop stops when the last argument is one of the trailing $X\langle number\rangle$, and that $\langle number\rangle$ is added to the sum of 9 that precedes, to adjust the result.

```

5186 \cs_new:Npn \str_count_spaces:N
5187 { \exp_args:No \str_count_spaces:n }
5188 \cs_generate_variant:Nn \str_count_spaces:N { c }
5189 \cs_new:Npn \str_count_spaces:n #1
5190 {
5191     \int_eval:n
5192     {
5193         \exp_after:wN \__str_count_spaces_loop:w
5194         \tl_to_str:n {#1} ~
5195         X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
5196         \s__str_stop
5197     }
5198 }
5199 \cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
5200 {
5201     \if_meaning:w X #9
5202         \__str_use_i_delimit_by_s_stop:nw
5203     \fi:
5204     9 + \__str_count_spaces_loop:w
5205 }

```

(End definition for `\str_count_spaces:N`, `\str_count_spaces:n`, and `_str_count_spaces_loop:w`. These functions are documented on page 66.)

`\str_count:N` To count characters in a string we could first escape all spaces using `_kernel_str_to_other:n`, then pass the result to `\tl_count:n`. However, the escaping step would be quadratic in the number of characters in the string, and we can do better. Namely, sum the number of spaces (`\str_count_spaces:n`) and the result of `\tl_count:n`, which ignores spaces. Since strings tend to be longer than token lists, we use specialized functions to count characters ignoring spaces. Namely, loop, grabbing 9 non-space characters at each step, and end as soon as we reach one of the 9 trailing items. The internal function `_str_count:n`, used in `\str_item:nn` and `\str_range:nnn`, is similar to `\str_count_ignore_spaces:n` but expects its argument to already be a string or a string with spaces escaped.

```

5206 \cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
5207 \cs_generate_variant:Nn \str_count:N { c }
5208 \cs_new:Npn \str_count:n #1
5209 {
5210     \__str_count_aux:n
5211     {
5212         \str_count_spaces:n {#1}
5213         + \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
5214     }
5215 }
5216 \cs_new:Npn \__str_count:n #1
5217 {
5218     \__str_count_aux:n
5219     { \__str_count_loop:NNNNNNNNN #1 }
5220 }
5221 \cs_new:Npn \str_count_ignore_spaces:n #1
5222 {
5223     \__str_count_aux:n
5224     { \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1} }
5225 }
5226 \cs_new:Npn \__str_count_aux:n #1
5227 {
5228     \int_eval:n
5229     {
5230         #1
5231         { X 8 } { X 7 } { X 6 }
5232         { X 5 } { X 4 } { X 3 }
5233         { X 2 } { X 1 } { X 0 }
5234         \s__str_stop
5235     }
5236 }
5237 \cs_new:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
5238 {
5239     \if_meaning:w X #9
5240         \exp_after:wN \__str_use_none_delimit_by_s_stop:w
5241     \fi:
5242     9 + \__str_count_loop:NNNNNNNNN
5243 }
```

(End definition for `\str_count:N` and others. These functions are documented on page 66.)

9.8 The first character in a string

`\str_head:N` The `_ignore_spaces` variant applies `\tl_to_str:n` then grabs the first item, thus skipping spaces. As usual, `\str_head:N` expands its argument and hands it to `\str_head:n`. To circumvent the fact that TeX skips spaces when grabbing undelimited macro parameters, `_str_head:w` takes an argument delimited by a space. If #1 starts with a non-space character, `_str_use_i_delimit_by_s_stop:nw` leaves that in the input stream. On the other hand, if #1 starts with a space, the `_str_head:w` takes an empty argument, and the single (initially braced) space in the definition of `_str_head:w` makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of `\str_head:n` gives an empty result after passing through `_str_use_i_delimit_by_s_stop:nw`.

```

5244 \cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
5245 \cs_generate_variant:Nn \str_head:N { c }
5246 \cs_new:Npn \str_head:n #1
{
5247   {
5248     \exp_after:wN \_str_head:w
5249     \tl_to_str:n {#1}
5250     { { } } ~ \s__str_stop
5251   }
5252 \cs_new:Npn \_str_head:w #1 ~ %
5253 { \_str_use_i_delimit_by_s_stop:nw #1 { ~ } }
5254 \cs_new:Npn \str_head_ignore_spaces:n #1
{
5255   {
5256     \exp_after:wN \_str_use_i_delimit_by_s_stop:nw
5257     \tl_to_str:n {#1} { } \s__str_stop
5258   }
}

```

(End definition for `\str_head:N` and others. These functions are documented on page 67.)

`\str_tail:N` Getting the tail is a little bit more convoluted than the head of a string. We hit the front of the string with `\reverse_if:N \if_charcode:w \scan_stop:`. This removes the first character, and necessarily makes the test true, since the character cannot match `\scan_stop:`. The auxiliary function then inserts the required `\fi:` to close the conditional, and leaves the tail of the string in the input stream. The details are such that an empty string has an empty tail (this requires in particular that the end-marker X be unexpandable and not a control sequence). The `_ignore_spaces` is rather simpler: after converting the input to a string, `_str_tail_auxii:w` removes one undelimited argument and leaves everything else until an end-marker `\s__str_mark`. One can check that an empty (or blank) string yields an empty tail.

```

5259 \cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
5260 \cs_generate_variant:Nn \str_tail:N { c }
5261 \cs_new:Npn \str_tail:n #1
{
5262   {
5263     \exp_after:wN \_str_tail_auxii:w
5264     \reverse_if:N \if_charcode:w
5265       \scan_stop: \tl_to_str:n {#1} X X \s__str_stop
5266   }
5267 \cs_new:Npn \_str_tail_auxii:w #1 X #2 \s__str_stop { \fi: #1 }
5268 \cs_new:Npn \str_tail_ignore_spaces:n #1
{
5269   {
5270     \exp_after:wN \_str_tail_auxii:w
5271     \tl_to_str:n {#1} \s__str_mark \s__str_mark \s__str_stop
}

```

```

5272     }
5273 \cs_new:Npn \__str_tail_auxii:w #1 #2 \s__str_mark #3 \s__str_stop { #2 }
```

(End definition for `\str_tail:N` and others. These functions are documented on page 67.)

9.9 String manipulation

```

\str_foldcase:n
\str_foldcase:V
\str_lowercase:n
\str_lowercase:f
\str_lowercase:nn
\str_change_case_aux:nn
\str_change_case_result:nw
\str_change_case_output:nw
\str_change_case_output:fw
\str_change_case_end:nw
\str_change_case_loop:nw
\str_change_case_space:n
\str_change_case_char:nN

5274 \cs_new:Npn \str_foldcase:n #1 { \__str_change_case:nn {#1} { fold } }
5275 \cs_new:Npn \str_lowercase:n #1 { \__str_change_case:nn {#1} { lower } }
5276 \cs_new:Npn \str_uppercase:n #1 { \__str_change_case:nn {#1} { upper } }
5277 \cs_generate_variant:Nn \str_foldcase:n { V }
5278 \cs_generate_variant:Nn \str_lowercase:n { f }
5279 \cs_generate_variant:Nn \str_uppercase:n { f }
5280 \cs_new:Npn \__str_change_case:nn #1
5281 {
5282     \exp_after:wN \__str_change_case_aux:nn \exp_after:wN
5283         { \tl_to_str:n {#1} }
5284 }
5285 \cs_new:Npn \__str_change_case_aux:nn #1#2
5286 {
5287     \__str_change_case_loop:nw {#2} #1 \q__str_recursion_tail \q__str_recursion_stop
5288         \__str_change_case_result:n {}
5289 }
5290 \cs_new:Npn \__str_change_case_output:nw #1#2 \__str_change_case_result:n #3
5291     { #2 \__str_change_case_result:n { #3 #1 } }
5292 \cs_generate_variant:Nn \__str_change_case_output:nw { f }
5293 \cs_new:Npn \__str_change_case_end:wn #1 \__str_change_case_result:n #2
5294     { \tl_to_str:n {#2} }
5295 \cs_new:Npn \__str_change_case_loop:nw #1#2 \q__str_recursion_stop
5296 {
5297     \tl_if_head_is_space:nTF {#2}
5298         { \__str_change_case_space:n }
5299         { \__str_change_case_char:nN }
5300     {#1} #2 \q__str_recursion_stop
5301 }

5302 \exp_last_unbraced:NNNNo
5303 \cs_new:Npn \__str_change_case_space:n #1 \c_space_tl
5304 {
5305     \__str_change_case_output:nw { ~ }
5306     \__str_change_case_loop:nw {#1}
5307 }
5308 \cs_new:Npn \__str_change_case_char:nN #1#2
5309 {
5310     \__str_if_recursion_tail_stop_do:Nn #2
5311         { \__str_change_case_end:wn }
5312     \__str_change_case_output:fw
5313         { \use:c { char_str_ #1 case:N } #2 }
5314     \__str_change_case_loop:nw {#1}
5315 }
```

(End definition for `\str_foldcase:n` and others. These functions are documented on page 70.)

```

\c_ampersand_str      For all of those strings, use \cs_to_str:N to get characters with the correct category
\c_atsign_str
\c_backslash_str
\c_left_brace_str
\c_right_brace_str
\c_circumflex_str
\c_colon_str
\c_dollar_str
\c_hash_str
\c_percent_str
\c_tilde_str
\c_underscore_str

```

5316 \str_const:Nx \c_ampersand_str { \cs_to_str:N \& }
5317 \str_const:Nx \c_atsign_str { \cs_to_str:N \@ }
5318 \str_const:Nx \c_backslash_str { \cs_to_str:N \\ }
5319 \str_const:Nx \c_left_brace_str { \cs_to_str:N \{ }
5320 \str_const:Nx \c_right_brace_str { \cs_to_str:N \} }
5321 \str_const:Nx \c_circumflex_str { \cs_to_str:N \^{} }
5322 \str_const:Nx \c_colon_str { \cs_to_str:N \: }
5323 \str_const:Nx \c_dollar_str { \cs_to_str:N \\$ }
5324 \str_const:Nx \c_hash_str { \cs_to_str:N \# }
5325 \str_const:Nx \c_percent_str { \cs_to_str:N \% }
5326 \str_const:Nx \c_tilde_str { \cs_to_str:N \~{} }
5327 \str_const:Nx \c_underscore_str { \cs_to_str:N _ }

(End definition for \c_ampersand_str and others. These variables are documented on page 71.)

\l_tmpa_str Scratch strings.

```

\l_tmpb_str
\g_tmpa_str
\g_tmpb_str

```

5328 \str_new:N \l_tmpa_str
5329 \str_new:N \l_tmpb_str
5330 \str_new:N \g_tmpa_str
5331 \str_new:N \g_tmpb_str

(End definition for \l_tmpa_str and others. These variables are documented on page 71.)

9.10 Viewing strings

\str_show:n Displays a string on the terminal.

```

\str_show:N
\str_show:c
\str_log:n
\str_log:N
\str_log:c

```

5332 \cs_new_eq:NN \str_show:n \tl_show:n
5333 \cs_new_eq:NN \str_show:N \tl_show:N
5334 \cs_generate_variant:Nn \str_show:N { c }
5335 \cs_new_eq:NN \str_log:n \tl_log:n
5336 \cs_new_eq:NN \str_log:N \tl_log:N
5337 \cs_generate_variant:Nn \str_log:N { c }

(End definition for \str_show:n and others. These functions are documented on page 70.)

5338 ⟨/package⟩

10 I3str-convert implementation

```

5339 ⟨*package⟩
5340 ⟨@=str⟩

```

10.1 Helpers

10.1.1 Variables and constants

```

\__str_tmp:w
\l__str_internal_tl

```

Internal scratch space for some functions.

5341 \cs_new_protected:Npn __str_tmp:w { }
5342 \tl_new:N \l__str_internal_tl

(End definition for __str_tmp:w and \l__str_internal_tl.)

\g__str_result_tl The \g__str_result_tl variable is used to hold the result of various internal string operations (mostly conversions) which are typically performed in a group. The variable is global so that it remains defined outside the group, to be assigned to a user-provided variable.

```
5343 \tl_new:N \g__str_result_tl
```

(End definition for \g__str_result_tl.)

\c__str_replacement_char_int When converting, invalid bytes are replaced by the Unicode replacement character "FFFD".

```
5344 \int_const:Nn \c__str_replacement_char_int { "FFFD }
```

(End definition for \c__str_replacement_char_int.)

\c__str_max_byte_int The maximal byte number.

```
5345 \int_const:Nn \c__str_max_byte_int { 255 }
```

(End definition for \c__str_max_byte_int.)

\s__str Internal scan marks.

```
5346 \scan_new:N \s__str
```

(End definition for \s__str.)

\q__str_nil Internal quarks.

```
5347 \quark_new:N \q__str_nil
```

(End definition for \q__str_nil.)

\g__str_alias_prop To avoid needing one file per encoding/escaping alias, we keep track of those in a property list.

```
5348 \prop_new:N \g__str_alias_prop
5349 \prop_gput:Nnn \g__str_alias_prop { latin1 } { iso88591 }
5350 \prop_gput:Nnn \g__str_alias_prop { latin2 } { iso88592 }
5351 \prop_gput:Nnn \g__str_alias_prop { latin3 } { iso88593 }
5352 \prop_gput:Nnn \g__str_alias_prop { latin4 } { iso88594 }
5353 \prop_gput:Nnn \g__str_alias_prop { latin5 } { iso88599 }
5354 \prop_gput:Nnn \g__str_alias_prop { latin6 } { iso885910 }
5355 \prop_gput:Nnn \g__str_alias_prop { latin7 } { iso885913 }
5356 \prop_gput:Nnn \g__str_alias_prop { latin8 } { iso885914 }
5357 \prop_gput:Nnn \g__str_alias_prop { latin9 } { iso885915 }
5358 \prop_gput:Nnn \g__str_alias_prop { latin10 } { iso885916 }
5359 \prop_gput:Nnn \g__str_alias_prop { utf16le } { utf16 }
5360 \prop_gput:Nnn \g__str_alias_prop { utf16be } { utf16 }
5361 \prop_gput:Nnn \g__str_alias_prop { utf32le } { utf32 }
5362 \prop_gput:Nnn \g__str_alias_prop { utf32be } { utf32 }
5363 \prop_gput:Nnn \g__str_alias_prop { hexadecimal } { hex }
```

(End definition for \g__str_alias_prop.)

\g__str_error_bool In conversion functions with a built-in conditional, errors are not reported directly to the user, but the information is collected in this boolean, used at the end to decide on which branch of the conditional to take.

```
5364 \bool_new:N \g__str_error_bool
```

(End definition for `\g_str_error_bool`.)

`str_byte` Conversions from one $\langle encoding \rangle / \langle escaping \rangle$ pair to another are done within x-expanding assignments. Errors are signalled by raising the relevant flag.

```
5365 \flag_new:n { str_byte }
5366 \flag_new:n { str_error }
```

(End definition for `str_byte` and `str_error`. These variables are documented on page ??.)

10.2 String conditionals

`__str_if_contains_char:NnT`

Expects the $\langle token list \rangle$ to be an $\langle other string \rangle$: the caller is responsible for ensuring that no (too-)special catcodes remain. Loop over the characters of the string, comparing character codes. The loop is broken if character codes match. Otherwise we return “false”.

```
5367 \prg_new_conditional:Npnn \__str_if_contains_char:Nn #1#2 { T , TF }
5368 {
5369   \exp_after:wN \__str_if_contains_char_aux:nn \exp_after:wN {#1} {#2}
5370   { \prg_break:n { ? \fi: } }
5371   \prg_break_point:
5372   \prg_return_false:
5373 }
5374 \cs_new:Npn \__str_if_contains_char_aux:nn #1#2
5375 { \__str_if_contains_char_auxi:nN {#2} #1 }
5376 \prg_new_conditional:Npnn \__str_if_contains_char:nn #1#2 { TF }
5377 {
5378   \__str_if_contains_char_auxi:nN {#2} #1 { \prg_break:n { ? \fi: } }
5379   \prg_break_point:
5380   \prg_return_false:
5381 }
5382 \cs_new:Npn \__str_if_contains_char_auxi:nN #1#2
5383 {
5384   \if_charcode:w #1 #2
5385     \exp_after:wN \__str_if_contains_char_true:
5386     \fi:
5387     \__str_if_contains_char_auxi:nN {#1}
5388   }
5389 \cs_new:Npn \__str_if_contains_char_true:
5390 { \prg_break:n { \prg_return_true: \use_none:n } }
```

(End definition for `__str_if_contains_char:NnT` and others.)

`__str_octal_use:NTF`

`__str_octal_use:NTF` $\langle token \rangle$ $\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$

If the $\langle token \rangle$ is an octal digit, it is left in the input stream, followed by the $\langle true code \rangle$. Otherwise, the $\langle false code \rangle$ is left in the input stream.

TeXhackers note: This function will fail if the escape character is an octal digit. We are thus careful to set the escape character to a known value before using it. TeX dutifully detects octal digits for us: if #1 is an octal digit, then the right-hand side of the comparison is ’1#1, greater than 1. Otherwise, the right-hand side stops as ’1, and the conditional takes the `false` branch.

```

5391 \prg_new_if_int:NNN \__str_octal_use:N #1 { TF }
5392 {
5393     \if_int_compare:w 1 < '1 \token_to_str:N #1 \exp_stop_f:
5394         #1 \prg_return_true:
5395     \else:
5396         \prg_return_false:
5397     \fi:
5398 }

```

(End definition for `__str_octal_use:NTF`.)

`__str_hexadecimal_use:NTF` \TeX detects uppercase hexadecimal digits for us (see `__str_hexadecimal_use:NTF`), but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

5399 \prg_new_if_int:NNN \__str_hexadecimal_use:N #1 { TF }
5400 {
5401     \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
5402         #1 \prg_return_true:
5403     \else:
5404         \if_case:w \int_eval:n { \exp_after:wN ` \token_to_str:N #1 - `a }
5405             A
5406             \or: B
5407             \or: C
5408             \or: D
5409             \or: E
5410             \or: F
5411         \else:
5412             \prg_return_false:
5413             \exp_after:wN \use_none:n
5414         \fi:
5415         \prg_return_true:
5416     \fi:
5417 }

```

(End definition for `__str_hexadecimal_use:NTF`.)

10.3 Conversions

10.3.1 Producing one byte or character

`\c__str_byte_0_tl` `\c__str_byte_1_tl` `\c__str_byte_255_tl` `\c__str_byte_-1_tl` For each integer N in the range $[0, 255]$, we create a constant token list which holds three character tokens with category code other: the character with character code N , followed by the representation of N as two hexadecimal digits. The value -1 is given a default token list which ensures that later functions give an empty result for the input -1 .

```

5418 \group_begin:
5419   \__kernel_tl_set:Nx \l__str_internal_tl { \tl_to_str:n { 0123456789ABCDEF } }
5420   \tl_map_inline:Nn \l__str_internal_tl
5421   {
5422     \tl_map_inline:Nn \l__str_internal_tl
5423     {
5424       \tl_const:cx { c__str_byte_ \int_eval:n {"#1##1} _tl }
5425       { \char_generate:nn { "#1##1" } { 12 } #1 ##1 }
5426     }
5427   }
5428 \group_end:
5429 \tl_const:cn { c__str_byte_-1_tl } { { } \use_none:n { } }

```

(End definition for `\c__str_byte_0_t1` and others.)

```
\__str_output_byte:n  
\__str_output_byte:w  
\__str_output_hexadecimal:n  
  \__str_output_end:  
  
Those functions must be used carefully: feeding them a value outside the range [-1, 255] will attempt to use the undefined token list variable \c__str_byte_(number)_tl. Assuming that the argument is in the right range, we expand the corresponding token list, and pick either the byte (first token) or the hexadecimal representations (second and third tokens). The value -1 produces an empty result in both cases.
```

```
5430 \cs_new:Npn \__str_output_byte:n #1  
5431   { \__str_output_byte:w #1 \__str_output_end: }  
5432 \cs_new:Npn \__str_output_byte:w  
5433   {  
5434     \exp_after:wN \exp_after:wN  
5435     \exp_after:wN \use_i:nnn  
5436     \cs:w c__str_byte_ \int_eval:w  
5437   }  
5438 \cs_new:Npn \__str_output_hexadecimal:n #1  
5439   {  
5440     \exp_after:wN \exp_after:wN  
5441     \exp_after:wN \use_none:n  
5442     \cs:w c__str_byte_ \int_eval:n {#1} _tl \cs_end:  
5443   }  
5444 \cs_new:Npn \__str_output_end:  
5445   { \scan_stop: _tl \cs_end: }  
  
(End definition for \__str_output_byte:n and others.)
```

Convert a number in the range [0, 65535] to a pair of bytes, either big-endian or little-endian.

```
5446 \cs_new:Npn \__str_output_byte_pair_be:n #1  
5447   {  
5448     \exp_args:Nf \__str_output_byte_pair:nnN  
5449     { \int_div_truncate:nn { #1 } { "100" } {#1} \use:nn  
5450   }  
5451 \cs_new:Npn \__str_output_byte_pair_le:n #1  
5452   {  
5453     \exp_args:Nf \__str_output_byte_pair:nnN  
5454     { \int_div_truncate:nn { #1 } { "100" } {#1} \use_i:nn  
5455   }  
5456 \cs_new:Npn \__str_output_byte_pair:nnN #1#2#3  
5457   {  
5458     #3  
5459     { \__str_output_byte:n { #1 } }  
5460     { \__str_output_byte:n { #2 - #1 * "100" } }  
5461   }  
  
(End definition for \__str_output_byte_pair_be:n, \__str_output_byte_pair_le:n, and \__str_output_byte_pair:nnN.)
```

10.3.2 Mapping functions for conversions

This maps the function #1 over all characters in `\g__str_result_tl`, which should be a byte string in most cases, sometimes a native string.

```
5462 \cs_new_protected:Npn \__str_convert_gmap:N #1  
5463   {
```

```

5464     \__kernel_t1_gset:Nx \g__str_result_t1
5465     {
5466         \exp_after:wN \__str_convert_gmap_loop>NN
5467         \exp_after:wN #1
5468         \g__str_result_t1 { ? \prg_break: }
5469         \prg_break_point:
5470     }
5471 }
5472 \cs_new:Npn \__str_convert_gmap_loop>NN #1#2
5473 {
5474     \use_none:n #2
5475     #1#2
5476     \__str_convert_gmap_loop>NN #1
5477 }

```

(End definition for `__str_convert_gmap:N` and `__str_convert_gmap_loop>NN`.)

`__str_convert_gmap_internal:N` This maps the function #1 over all character codes in `\g__str_result_t1`, which must be in the internal representation.

```

5478 \cs_new_protected:Npn \__str_convert_gmap_internal:N #1
5479 {
5480     \__kernel_t1_gset:Nx \g__str_result_t1
5481     {
5482         \exp_after:wN \__str_convert_gmap_internal_loop:Nww
5483         \exp_after:wN #1
5484         \g__str_result_t1 \s__str \s__str_stop \prg_break: \s__str
5485         \prg_break_point:
5486     }
5487 }
5488 \cs_new:Npn \__str_convert_gmap_internal_loop:Nww #1 #2 \s__str #3 \s__str
5489 {
5490     \__str_use_none_delimit_by_s_stop:w #3 \s__str_stop
5491     #1 {#3}
5492     \__str_convert_gmap_internal_loop:Nww #1
5493 }

```

(End definition for `__str_convert_gmap_internal:N` and `__str_convert_gmap_internal_loop:Nw`.)

10.3.3 Error-reporting during conversion

When converting using the function `\str_set_convert:Nnnn`, errors should be reported to the user after each step in the conversion. Errors are signalled by raising some flag (typically `@@_error`), so here we test that flag: if it is raised, give the user an error, otherwise remove the arguments. On the other hand, in the conditional functions `\str_set_convert:NnnnTF`, errors should be suppressed. This is done by changing `__str_if_flag_error:nnx` into `__str_if_flag_no_error:nnx` locally.

```

5494 \cs_new_protected:Npn \__str_if_flag_error:nnx #1
5495 {
5496     \flag_if_raised:nTF {#1}
5497     { \__kernel_msg_error:nnx { str } }
5498     { \use_none:nn }
5499 }
5500 \cs_new_protected:Npn \__str_if_flag_no_error:nnx #1#2#3
5501 { \flag_if_raised:nT {#1} { \bool_gset_true:N \g__str_error_bool } }

```

(End definition for `__str_if_flag_error:n` and `__str_if_flag_no_error:n`.)

`__str_if_flag_times:nT` At the end of each conversion step, we raise all relevant errors as one error message, built on the fly. The height of each flag indicates how many times a given error was encountered. This function prints #2 followed by the number of occurrences of an error if it occurred, nothing otherwise.

```
5502 \cs_new:Npn \__str_if_flag_times:nT #1#2
  { \flag_if_raised:nT {#1} { #2~(x \flag_height:n {#1} ) } }
```

(End definition for `__str_if_flag_times:n`.)

10.3.4 Framework for conversions

Most functions in this module expect to be working with “native” strings. Strings can also be stored as bytes, in one of many encodings, for instance UTF8. The bytes themselves can be expressed in various ways in terms of TeX tokens, for instance as pairs of hexadecimal digits. The questions of going from arbitrary Unicode code points to bytes, and from bytes to tokens are mostly independent.

Conversions are done in four steps:

- “unescape” produces a string of bytes;
- “decode” takes in a string of bytes, and converts it to a list of Unicode characters in an internal representation, with items of the form

```
bytes \s__str Unicode code point \s__str
```

where we have collected the *bytes* which combined to form this particular Unicode character, and the *Unicode code point* is in the range [0, "10FFFF].

- “encode” encodes the internal list of code points as a byte string in the new encoding;
- “escape” escapes bytes as requested.

The process is modified in case one of the encoding is empty (or the conversion function has been set equal to the empty encoding because it was not found): then the unescape or escape step is ignored, and the decode or encode steps work on tokens instead of bytes. Otherwise, each step must ensure that it passes a correct byte string or internal string to the next step.

`\str_set_convert:Nnnn`
`\str_gset_convert:Nnnn`
`\str_set_convert:NnnnTF`
`\str_gset_convert:NnnnTF`

The input string is stored in `\g__str_result_t1`, then we: unescape and decode; encode and escape; exit the group and store the result in the user’s variable. The various conversion functions all act on `\g__str_result_t1`. Errors are silenced for the conditional functions by redefining `__str_if_flag_error:n` locally.

```
5504 \cs_new_protected:Npn \str_set_convert:Nnnn
  { \__str_convert:nNNnnn { } \tl_set_eq:NN }
5506 \cs_new_protected:Npn \str_gset_convert:Nnnn
  { \__str_convert:nNNnnn { } \tl_gset_eq:NN }
5508 \prg_new_protected_conditional:Npnn
  \str_set_convert:Nnnn #1#2#3#4 { T , F , TF }
5510 {
5511   \bool_gset_false:N \g__str_error_bool
  \__str_convert:nNNnnn
```

```

5513      { \cs_set_eq:NN \__str_if_flag_error:nx \__str_if_flag_no_error:nx }
5514      \tl_set_eq:NN #1 {#2} {#3} {#4}
5515      \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
5516    }
5517  \prg_new_protected_conditional:Npnn
5518    \str_gset_convert:Nnnn #1#2#3#4 { T , F , TF }
5519  {
5520    \bool_gset_false:N \g__str_error_bool
5521    \__str_convert:nNnnn
5522      { \cs_set_eq:NN \__str_if_flag_error:nx \__str_if_flag_no_error:nx }
5523      \tl_gset_eq:NN #1 {#2} {#3} {#4}
5524      \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
5525    }
5526  \cs_new_protected:Npn \__str_convert:nNnnn #1#2#3#4#5#6
5527  {
5528    \group_begin:
5529      #1
5530      \__kernel_tl_gset:Nx \g__str_result_tl { \__kernel_str_to_other_fast:n {#4} }
5531      \exp_after:wN \__str_convert:wwnn
5532        \tl_to_str:n {#5} /// \s__str_stop
5533        { decode } { unescape }
5534        \prg_do_nothing:
5535        \__str_convert_decode_:
5536        \exp_after:wN \__str_convert:wwnn
5537          \tl_to_str:n {#6} /// \s__str_stop
5538          { encode } { escape }
5539          \use_i:i:nn
5540            \__str_convert_encode_:
5541        \group_end:
5542        #2 #3 \g__str_result_tl
5543    }

```

(End definition for `\str_set_convert:Nnnn` and others. These functions are documented on page 72.)

`__str_convert:wwnn`
`__str_convert:NNNN`

The task of `__str_convert:wwnn` is to split `<encoding>/<escaping>` pairs into their components, #1 and #2. Calls to `__str_convert:nnn` ensure that the corresponding conversion functions are defined. The third auxiliary does the main work.

- #1 is the encoding conversion function;
- #2 is the escaping function;
- #3 is the escaping name for use in an error message;
- #4 is `\prg_do_nothing:` for unescaping/decoding, and `\use_i:i:nn` for encoding/escaping;
- #5 is the default encoding function (either “decode” or “encode”), for which there should be no escaping.

Let us ignore the native encoding for a second. In the unescaping/decoding phase, we want to do #2#1 in this order, and in the encoding/escaping phase, the order should be reversed: #4#2#1 does exactly that. If one of the encodings is the default (native), then the escaping should be ignored, with an error if any was given, and only the encoding, #1, should be performed.

```

5544 \cs_new_protected:Npn \__str_convert:wwwnn
5545     #1 / #2 // #3 \s__str_stop #4#5
5546 {
5547     \__str_convert:nnn {enc} {#4} {#1}
5548     \__str_convert:nnn {esc} {#5} {#2}
5549     \exp_args:Ncc \__str_convert>NNnNN
5550         { __str_convert_#4_#1: } { __str_convert_#5_#2: } {#2}
5551 }
5552 \cs_new_protected:Npn \__str_convert:NNnNN #1#2#3#4#5
5553 {
5554     \if_meaning:w #1 #5
5555         \tl_if_empty:nF {#3}
5556             { \__kernel_msg_error:nnx { str } { native-escaping } {#3} }
5557             #1
5558     \else:
5559         #4 #2 #1
5560     \fi:
5561 }

```

(End definition for `__str_convert:wwwnn` and `__str_convert:NNnNN`.)

`__str_convert:nnn`
`__str_convert:nnnn`

The arguments of `__str_convert:nnn` are: `enc` or `esc`, used to build filenames, the type of the conversion (unescape, decode, encode, escape), and the encoding or escaping name. If the function is already defined, no need to do anything. Otherwise, filter out all non-alphanumerics in the name, and lowercase it. Feed that, and the same three arguments, to `__str_convert:nnnn`. The task is then to make sure that the conversion function `#3_#1` corresponding to the type `#3` and filtered name `#1` is defined, then set our initial conversion function `#3_#4` equal to that.

How do we get the `#3_#1` conversion to be defined if it isn't? Two main cases.

First, if `#1` is a key in `\g__str_alias_prop`, then the value `\l__str_internal_tl` tells us what file to load. Loading is skipped if the file was already read, *i.e.*, if the conversion command based on `\l__str_internal_tl` already exists. Otherwise, try to load the file; if that fails, there is an error, use the default empty name instead.

Second, `#1` may be absent from the property list. The `\cs_if_exist:cF` test is automatically false, and we search for a file defining the encoding or escaping `#1` (this should allow third-party `.def` files). If the file is not found, there is an error, use the default empty name instead.

In all cases, the conversion based on `\l__str_internal_tl` is defined, so we can set the `#3_#1` function equal to that. In some cases (*e.g.*, `utf16be`), the `#3_#1` function is actually defined within the file we just loaded, and it is different from the `\l__str_internal_tl`-based function: we mustn't clobber that different definition.

```

5562 \cs_new_protected:Npn \__str_convert:nnn #1#2#3
5563 {
5564     \cs_if_exist:cF { __str_convert_#2_#3: }
5565     {
5566         \exp_args:Nx \__str_convert:nnnn
5567             { \__str_convert_lowercase_alphanum:n {#3} }
5568             {#1} {#2} {#3}
5569     }
5570 }
5571 \cs_new_protected:Npn \__str_convert:nnnn #1#2#3#4
5572 {
5573     \cs_if_exist:cF { __str_convert_#3_#1: }

```

```

5574 {
5575   \prop_get:NnNF \g__str_alias_prop {#1} \l__str_internal_tl
5576   { \tl_set:Nn \l__str_internal_tl {#1} }
5577   \cs_if_exist:cF { __str_convert_#3_ \l__str_internal_tl : }
5578   {
5579     \file_if_exist:nTF { 13str-#2- \l__str_internal_tl .def }
5580     {
5581       \group_begin:
5582         \__str_load_catcodes:
5583         \file_input:n { 13str-#2- \l__str_internal_tl .def }
5584       \group_end:
5585     }
5586     {
5587       \tl_clear:N \l__str_internal_tl
5588       \__kernel_msg_error:nxxx { str } { unknown-#2 } { #4 } { #1 }
5589     }
5590   }
5591   \cs_if_exist:cF { __str_convert_#3_#1: }
5592   {
5593     \cs_gset_eq:cc { __str_convert_#3_#1: }
5594     { __str_convert_#3_ \l__str_internal_tl : }
5595   }
5596 }
5597 \cs_gset_eq:cc { __str_convert_#3_#4: } { __str_convert_#3_#1: }
5598 }
```

(End definition for `__str_convert:nnn` and `__str_convert:nnnn`.)

This function keeps only letters and digits, with upper case letters converted to lower case.

```

5599 \cs_new:Npn \__str_convert_lowercase_alphanum:n #1
5600   {
5601     \exp_after:wN \__str_convert_lowercase_alphanum_loop:N
5602     \tl_to_str:n {#1} { ? \prg_break: }
5603     \prg_break_point:
5604   }
5605 \cs_new:Npn \__str_convert_lowercase_alphanum_loop:N #1
5606   {
5607     \use_none:n #1
5608     \if_int_compare:w '#1 > 'Z \exp_stop_f:
5609       \if_int_compare:w '#1 > 'z \exp_stop_f: \else:
5610         \if_int_compare:w '#1 < 'a \exp_stop_f: \else:
5611           #1
5612           \fi:
5613           \fi:
5614     \else:
5615       \if_int_compare:w '#1 < 'A \exp_stop_f:
5616         \if_int_compare:w 1 < 1#1 \exp_stop_f:
5617           #1
5618           \fi:
5619     \else:
5620       \__str_output_byte:n { '#1 + 'a - 'A }
5621     \fi:
5622   \fi:
```

```

5623     \__str_convert_lowercase_alphanum_loop:N
5624 }

```

(End definition for __str_convert_lowercase_alphanum:n and __str_convert_lowercase_alphanum-loop:N.)

__str_load_catcodes: Since encoding files may be loaded at arbitrary places in a T_EX document, including within verbatim mode, we set the catcodes of all characters appearing in any encoding definition file.

```

5625 \cs_new_protected:Npn \__str_load_catcodes:
5626 {
5627     \char_set_catcode_escape:N \\%
5628     \char_set_catcode_group_begin:N \{
5629     \char_set_catcode_group_end:N \}
5630     \char_set_catcode_math_toggle:N \$%
5631     \char_set_catcode_alignment:N \&%
5632     \char_set_catcode_parameter:N \#%
5633     \char_set_catcode_math_superscript:N \^%
5634     \char_set_catcode_ignore:N \ %
5635     \char_set_catcode_space:N \~%
5636     \tl_map_function:nN { abcdefghijklmnopqrstuvwxyz_ : ABCDEFILNPSTUX }%
5637         \char_set_catcode_letter:N%
5638     \tl_map_function:nN { 0123456789" ?*+-.( ) , !/<>[] ; = }%
5639         \char_set_catcode_other:N%
5640     \char_set_catcode_comment:N \%
5641     \int_set:Nn \tex_endlinechar:D {32}%
5642 }

```

(End definition for __str_load_catcodes:.)

10.3.5 Byte unescape and escape

Strings of bytes may need to be stored in auxiliary files in safe “escaping” formats. Each such escaping is only loaded as needed. By default, on input any non-byte is filtered out, while the output simply consists in letting bytes through.

__str_filter_bytes:n __str_filter_bytes_aux:N In the case of 8-bit engines, every character is a byte. For Unicode-aware engines, test the character code; non-bytes cause us to raise the flag **str_byte**. Spaces have already been given the correct category code when this function is called.

```

5643 \bool_lazy_any:nTF
5644 {
5645     \sys_if_engine_luatex_p:
5646     \sys_if_engine_xetex_p:
5647 }
5648 {
5649     \cs_new:Npn \__str_filter_bytes:n #1
5650     {
5651         \__str_filter_bytes_aux:N #1
5652         { ? \prg_break: }
5653         \prg_break_point:
5654     }
5655     \cs_new:Npn \__str_filter_bytes_aux:N #1
5656     {
5657         \use_none:n #1

```

```

5658     \if_int_compare:w '#1 < 256 \exp_stop_f:
5659         #1
5660     \else:
5661         \flag_raise:n { str_byte }
5662     \fi:
5663     \__str_filter_bytes_aux:N
5664 }
5665 }
5666 { \cs_new_eq:NN \__str_filter_bytes:n \use:n }

(End definition for \__str_filter_bytes:n and \__str_filter_bytes_aux:N.)

```

__str_convert_unescape_ : The simplest unescaping method removes non-bytes from \g__str_result_tl.

```

\__str_convert_unescape_bytes:
5667 \bool_lazy_any:nTF
5668 {
5669     \sys_if_engine_luatex_p:
5670     \sys_if_engine_xetex_p:
5671 }
5672 {
5673     \cs_new_protected:Npn \__str_convert_unescape_:
5674     {
5675         \flag_clear:n { str_byte }
5676         \__kernel_tl_gset:Nx \g__str_result_tl
5677             { \exp_args:No \__str_filter_bytes:n \g__str_result_tl }
5678             \__str_if_flag_error:n { str_byte } { non-byte } { bytes }
5679     }
5680 }
5681 { \cs_new_protected:Npn \__str_convert_unescape_:{ } }
5682 \cs_new_eq:NN \__str_convert_unescape_bytes: \__str_convert_unescape_:

(End definition for \__str_convert_unescape_ : and \__str_convert_unescape_bytes::)

```

__str_convert_escape_ : __str_convert_escape_bytes: The simplest form of escape leaves the bytes from the previous step of the conversion unchanged.

```

5683 \cs_new_protected:Npn \__str_convert_escape_:{ }
5684 \cs_new_eq:NN \__str_convert_escape_bytes: \__str_convert_escape_:

(End definition for \__str_convert_escape_ : and \__str_convert_escape_bytes::)

```

10.3.6 Native strings

__str_convert_decode_ : Convert each character to its character code, one at a time.

```

\__str_decode_native_char:N
5685 \cs_new_protected:Npn \__str_convert_decode_:
5686     { \__str_convert_gmap:N \__str_decode_native_char:N }
5687 \cs_new:Npn \__str_decode_native_char:N #1
5688     { #1 \s__str \int_value:w '#1 \s__str }

(End definition for \__str_convert_decode_ : and \__str_decode_native_char:N.)

```

__str_convert_encode_ : __str_encode_native_char:N The conversion from an internal string to native character tokens basically maps \char_generate:nn through the code-points, but in non-Unicode-aware engines we use a fall-back character ? rather than nothing when given a character code outside [0,255]. We detect the presence of bad characters using a flag and only produce a single error after the x-expanding assignment.

```

5689 \bool_lazy_any:nTF
5690   {
5691     \sys_if_engine_luatex_p:
5692     \sys_if_engine_xetex_p:
5693   }
5694   {
5695     \cs_new_protected:Npn \__str_convert_encode_:
5696       { \__str_convert_gmap_internal:N \__str_encode_native_char:n }
5697     \cs_new:Npn \__str_encode_native_char:n #1
5698       { \char_generate:nn {#1} {12} }
5699   }
5700   {
5701     \cs_new_protected:Npn \__str_convert_encode_:
5702       {
5703         \flag_clear:n { str_error }
5704         \__str_convert_gmap_internal:N \__str_encode_native_char:n
5705         \__str_if_flag_error:nnx { str_error }
5706           { native-overflow } { }
5707       }
5708     \cs_new:Npn \__str_encode_native_char:n #1
5709       {
5710         \if_int_compare:w #1 > \c__str_max_byte_int
5711           \flag_raise:n { str_error }
5712           ?
5713         \else:
5714           \char_generate:nn {#1} {12}
5715         \fi:
5716       }
5717     \__kernel_msg_new:nnnn { str } { native-overflow }
5718       { Character-code-too-large-for-this-engine. }
5719     {
5720       This-engine-only-support-8-bit-characters:-
5721       valid-character-codes-are-in-the-range-[0,255].-
5722       To-manipulate-arbitrary-Unicode,-use-LuaTeX-or-XeTeX.
5723     }
5724   }

```

(End definition for `__str_convert_encode_:` and `__str_encode_native_char:n.`)

10.3.7 clist

`__str_convert_decode_clist:` Convert each integer to the internal form. We first turn `\g__str_result_tl` into a `clist` variable, as this avoids problems with leading or trailing commas.

```

5725 \cs_new_protected:Npn \__str_convert_decode_clist:
5726   {
5727     \clist_gset:No \g__str_result_tl \g__str_result_tl
5728     \__kernel_tl_gset:Nx \g__str_result_tl
5729     {
5730       \exp_args:No \clist_map_function:nN
5731         \g__str_result_tl \__str_decode_clist_char:n
5732     }
5733   }
5734 \cs_new:Npn \__str_decode_clist_char:n #1
5735   { #1 \s__str \int_eval:n {#1} \s__str }

```

(End definition for `_str_convert_decode_clist`: and `_str_decode_clist_char:n`.)

```
\_str_convert_encode_clist:  
\_str_encode_clist_char:n
```

Convert the internal list of character codes to a comma-list of character codes. The first line produces a comma-list with a leading comma, removed in the next step (this also works in the empty case, since `\tl_tail:N` does not trigger an error in this case).

```
5736 \cs_new_protected:Npn \_str_convert_encode_clist:  
5737 {  
5738     \_str_convert_gmap_internal:N \_str_encode_clist_char:n  
5739     \_kernel_tl_gset:Nx \g__str_result_tl { \tl_tail:N \g__str_result_tl }  
5740 }  
5741 \cs_new:Npn \_str_encode_clist_char:n #1 { , #1 }
```

(End definition for `_str_convert_encode_clist`: and `_str_encode_clist_char:n`.)

10.3.8 8-bit encodings

It is not clear in what situations 8-bit encodings are used, hence it is not clear what should be optimized. The current approach is reasonably efficient to convert long strings, and it scales well when using many different encodings.

The data needed to support a given 8-bit encoding is stored in a file that consists of a single function call

```
\_str_declare_eight_bit_encoding:nnnn {\langle name\rangle} {\langle modulo\rangle} {\langle mapping\rangle}  
{\langle missing\rangle}
```

This declares the encoding `_str_declare_eight_bit_encoding` to map bytes to Unicode characters according to the `_str_mapping`, and map those bytes which are not mentioned in the `_str_mapping` either to the replacement character (if they appear in `_str_missing`), or to themselves. The `_str_mapping` argument is a token list of pairs `\{⟨byte⟩\} \{⟨Unicode⟩\}` expressed in uppercase hexadecimal notation. The `_str_missing` argument is a token list of `\{⟨byte⟩\}`. Every `\{⟨byte⟩\}` which does not appear in the `_str_mapping` nor the `_str_missing` lists maps to itself in Unicode, so for instance the `latin1` encoding has empty `_str_mapping` and `_str_missing` lists. The `_str_modulo` is a (decimal) integer between 256 and 558 inclusive, modulo which all Unicode code points supported by the encodings must be different.

We use two integer arrays per encoding. When decoding we only use the `_str_decode` integer array, with entry $n + 1$ (offset needed because integer array indices start at 1) equal to the Unicode code point that corresponds to the n -th byte in the encoding under consideration, or -1 if the given byte is invalid in this encoding. When encoding we use both arrays: upon seeing a code point n , we look up the entry (1 plus) n modulo some number M in the `_str_encode` array, which tells us the byte that might encode the given Unicode code point, then we check in the `_str_decode` array that indeed this byte encodes the Unicode code point we want. Here, M is an encoding-dependent integer between 256 and 558 (it turns out), chosen so that among the Unicode code points that can be validly represented in the given encoding, no pair of code points have the same value modulo M .

Loop through both lists of bytes to fill in the `_str_decode` integer array, then fill the `_str_encode` array accordingly. For bytes that are invalid in the given encoding, store -1 in the `_str_decode` array.

```
5742 \cs_new_protected:Npn \_str_declare_eight_bit_encoding:nnnn #1  
5743 {  
5744     \tl_set:Nn \l__str_internal_tl {#1}  
5745     \cs_new_protected:cpx { \_str_convert_decode_#1: }
```

```

5746      { \__str_convert_decode_eight_bit:n {#1} }
5747      \cs_new_protected:cpn { __str_convert_encode_#1: }
5748      { \__str_convert_encode_eight_bit:n {#1} }
5749      \exp_args:Ncc \__str_declare_eight_bit_aux:NNnnn
5750      { g__str_decode_#1_intarray } { g__str_encode_#1_intarray }
5751    }
5752 \cs_new_protected:Npn \__str_declare_eight_bit_aux:NNnnn #1#2#3#4#5
5753  {
5754    \intarray_new:Nn #1 { 256 }
5755    \int_step_inline:nnn { 0 } { 255 }
5756    { \intarray_gset:Nnn #1 { 1 + ##1 } {##1} }
5757    \__str_declare_eight_bit_loop:Nnn #1
5758    #4 { \s__str_stop \prg_break: } { }
5759    \prg_break_point:
5760    \__str_declare_eight_bit_loop:Nn #1
5761    #5 { \s__str_stop \prg_break: }
5762    \prg_break_point:
5763    \intarray_new:Nn #2 {#3}
5764    \int_step_inline:nnn { 0 } { 255 }
5765    {
5766      \int_compare:nNnf { \intarray_item:Nn #1 { 1 + ##1 } } = { -1 }
5767      {
5768        \intarray_gset:Nnn #2
5769        {
5770          1 +
5771          \int_mod:nn { \intarray_item:Nn #1 { 1 + ##1 } }
5772          { \intarray_count:N #2 }
5773        }
5774        {##1}
5775      }
5776    }
5777  }
5778 \cs_new_protected:Npn \__str_declare_eight_bit_loop:Nnn #1#2#3
5779  {
5780    \__str_use_none_delimit_by_s_stop:w #2 \s__str_stop
5781    \intarray_gset:Nnn #1 { 1 + "#2 } { "#3 }
5782    \__str_declare_eight_bit_loop:Nnn #1
5783  }
5784 \cs_new_protected:Npn \__str_declare_eight_bit_loop:Nn #1#2
5785  {
5786    \__str_use_none_delimit_by_s_stop:w #2 \s__str_stop
5787    \intarray_gset:Nnn #1 { 1 + "#2 } { -1 }
5788    \__str_declare_eight_bit_loop:Nn #1
5789  }

```

(End definition for `__str_declare_eight_bit_encoding:nnnn` and others.)

```
\__str_convert_decode_eight_bit:n
  \__str_decode_eight_bit_aux:n
  \__str_decode_eight_bit_aux:Nn
```

The map from bytes to Unicode code points is in the `decode` array corresponding to the given encoding. Define `__str_tmp:w` and pass it successively all bytes in the string. It produces an internal representation with suitable `\s__str` inserted, and the corresponding code point is obtained by looking it up in the integer array. If the entry is `-1` then issue a replacement character and raise the flag indicating that there was an error.

```
5790 \cs_new_protected:Npn \__str_convert_decode_eight_bit:n #1
5791  {
```

```

5792   \cs_set:Npx \__str_tmp:w
5793   {
5794     \exp_not:N \__str_decode_eight_bit_aux:Nn
5795     \exp_not:c { g__str_decode_#1_intarray }
5796   }
5797   \flag_clear:n { str_error }
5798   \__str_convert_gmap:N \__str_tmp:w
5799   \__str_if_flag_error:nnx { str_error } { decode-8-bit } {#1}
5800 }
5801 \cs_new:Npn \__str_decode_eight_bit_aux:Nn #1#2
5802 {
5803   #2 \s__str
5804   \exp_args:Nf \__str_decode_eight_bit_aux:n
5805   { \intarray_item:Nn #1 { 1 + '#2 } }
5806   \s__str
5807 }
5808 \cs_new:Npn \__str_decode_eight_bit_aux:n #1
5809 {
5810   \if_int_compare:w #1 < \c_zero_int
5811     \flag_raise:n { str_error }
5812     \int_value:w \c__str_replacement_char_int
5813   \else:
5814     #1
5815   \fi:
5816 }

```

(End definition for `__str_convert_decode_eight_bit:n`, `__str_decode_eight_bit_aux:n`, and `__str_decode_eight_bit_aux:Nn`.)

```
\__str_convert_encode_eight_bit:n
\__str_encode_eight_bit_aux:nnN
\__str_encode_eight_bit_aux:NNn
```

It is not practical to make an integer array with indices in the full Unicode range, so we work modulo some number, which is simply the size of the `encode` integer array for the given encoding. This gives us a candidate byte for representing a given Unicode code point. Of course taking the modulo leads to collisions so we check in the `decode` array that the byte we got is indeed correct. Otherwise the Unicode code point we started from is simply not representable in the given encoding.

```

5817 \int_new:N \l__str_modulo_int
5818 \cs_new_protected:Npn \__str_convert_encode_eight_bit:n #1
5819 {
5820   \cs_set:Npx \__str_tmp:w
5821   {
5822     \exp_not:N \__str_encode_eight_bit_aux:NNn
5823     \exp_not:c { g__str_encode_#1_intarray }
5824     \exp_not:c { g__str_decode_#1_intarray }
5825   }
5826   \flag_clear:n { str_error }
5827   \__str_convert_gmap_internal:N \__str_tmp:w
5828   \__str_if_flag_error:nnx { str_error } { encode-8-bit } {#1}
5829 }
5830 \cs_new:Npn \__str_encode_eight_bit_aux:NNn #1#2#3
5831 {
5832   \exp_args:Nf \__str_encode_eight_bit_aux:nnN
5833   {
5834     \intarray_item:Nn #1
5835     { 1 + \int_mod:nn {#3} { \intarray_count:N #1 } }

```

```

5836     }
5837     {#3}
5838     #2
5839   }
5840 \cs_new:Npn \__str_encode_eight_bit_aux:nnN #1#2#3
5841   {
5842     \int_compare:nNnTF { \intarray_item:Nn #3 { 1 + #1 } } = {#2}
5843     { \__str_output_byte:n {#1} }
5844     { \flag_raise:n { str_error } }
5845   }

(End definition for \__str_convert_encode_eight_bit:n, \__str_encode_eight_bit_aux:nnN, and \__str_encode_eight_bit_aux:NNn.)
```

10.4 Messages

General messages, and messages for the encodings and escapings loaded by default (“native”, and “bytes”).

```

5846 \__kernel_msg_new:nnn { str } { unknown-esc }
5847   { Escaping-scheme-'#1'~(filtered:-'#2')~unknown. }
5848 \__kernel_msg_new:nnn { str } { unknown-enc }
5849   { Encoding-scheme-'#1'~(filtered:-'#2')~unknown. }
5850 \__kernel_msg_new:nnnn { str } { native-escaping }
5851   { The-'native'-encoding-scheme-does-not-support-any-escaping. }
5852   {
5853     Since-native-strings-do-not-consist-in-bytes,~
5854     none-of-the-escaping-methods-make-sense.~
5855     The-specified-escaping,'#1',will be ignored.
5856   }
5857 \__kernel_msg_new:nnn { str } { file-not-found }
5858   { File-'l3str-#1.def'~not~found. }
```

Message used when the “bytes” unescaping fails because the string given to \str_set_convert:Nnnn contains a non-byte. This cannot happen for the -8-bit engines. Messages used for other escapings and encodings are defined in each definition file.

```

5859 \bool_lazy_any:nT
5860   {
5861     \sys_if_engine_luatex_p:
5862     \sys_if_engine_xetex_p:
5863   }
5864   {
5865     \__kernel_msg_new:nnnn { str } { non-byte }
5866     { String-invalid-in-escaping-'#1':~it-may-only-contain-bytes. }
5867     {
5868       Some-characters-in-the-string-you-asked-to-convert-are-not-
5869       8-bit-characters.~Perhaps-the-string-is-a-'native'~Unicode-string?~
5870       If-it-is,~try~using\\
5871       \\
5872       \iow_indent:n
5873         {
5874           \iow_char:N\str_set_convert:Nnnn \\
5875             \ \ <str-var>-{\~<string>~\}~\{-native~\}~\{-<target-encoding>~\}
5876         }
5877     }
5878 }
```

Those messages are used when converting to and from 8-bit encodings.

```

5879 \__kernel_msg_new:nnnn { str } { decode-8-bit }
5880   { Invalid~string~in~encoding~'#1'. }
5881   {
5882     LaTeX~came~across~a~byte~which~is~not~defined~to~represent~
5883     any~character~in~the~encoding~'#1'.
5884   }
5885 \__kernel_msg_new:nnnn { str } { encode-8-bit }
5886   { Unicode~string~cannot~be~converted~to~encoding~'#1'. }
5887   {
5888     The~encoding~'#1'~only~contains~a~subset~of~all~Unicode~characters.~
5889     LaTeX~was~asked~to~convert~a~string~to~that~encoding,~but~that~
5890     string~contains~a~character~that~'#1'~does~not~support.
5891 }
```

10.5 Escaping definitions

Several of those encodings are defined by the pdf file format. The following byte storage methods are defined:

- **bytes** (default), non-bytes are filtered out, and bytes are left untouched (this is defined by default);
- **hex** or **hexadecimal**, as per the pdftEX primitive `\pdfescapehex`
- **name**, as per the pdftEX primitive `\pdfescapename`
- **string**, as per the pdftEX primitive `\pdfescapestring`
- **url**, as per the percent encoding of urls.

10.5.1 Unescape methods

`__str_convert_unescape_hex:` Take chars two by two, and interpret each pair as the hexadecimal code for a byte. Anything else than hexadecimal digits is ignored, raising the flag. A string which contains an odd number of hexadecimal digits gets 0 appended to it: this is equivalent to appending a 0 in all cases, and dropping it if it is alone.

```

5892 \cs_new_protected:Npn \__str_convert_unescape_hex:
5893   {
5894     \group_begin:
5895       \flag_clear:n { str_error }
5896       \int_set:Nn \tex_escapechar:D { 92 }
5897       \__kernel_tl_gset:Nx \g__str_result_tl
5898       {
5899         \__str_output_byte:w "
5900           \exp_last_unbraced:Nf \__str_unescape_hex_auxi:N
5901             { \tl_to_str:N \g__str_result_tl }
5902             0 { ? 0 - 1 \prg_break: }
5903             \prg_break_point:
5904             \__str_output_end:
5905         }
5906         \__str_if_flag_error:nx { str_error } { unescape-hex } { }
5907       \group_end:
5908   }
```

```

5909 \cs_new:Npn \__str_unescape_hex_auxi:N #1
5910  {
5911      \use_none:n #1
5912      \__str_hexadecimal_use:NTF #1
5913      { \__str_unescape_hex_auxii:N }
5914      {
5915          \flag_raise:n { str_error }
5916          \__str_unescape_hex_auxi:N
5917      }
5918  }
5919 \cs_new:Npn \__str_unescape_hex_auxii:N #1
5920  {
5921      \use_none:n #1
5922      \__str_hexadecimal_use:NTF #1
5923      {
5924          \__str_output_end:
5925          \__str_output_byte:w " \__str_unescape_hex_auxi:N
5926      }
5927      {
5928          \flag_raise:n { str_error }
5929          \__str_unescape_hex_auxii:N
5930      }
5931  }
5932 \__kernel_msg_new:nnnn { str } { unescape-hex }
5933 { String~invalid~in~escaping~'hex':~only~hexadecimal~digits~allowed. }
5934 {
5935     Some~characters~in~the~string~you~asked~to~convert~are~not~
5936     hexadecimal~digits~(0-9,~A-F,~a-f)~nor~spaces.
5937 }

```

(End definition for `__str_convert_unescape_hex:`, `__str_unescape_hex_auxi:N`, and `__str_unescape_hex_auxii:N`.)

```

\__str_convert_unescape_name:
\__str_unescape_name_loop:wnn
\__str_convert_unescape_url:
\__str_unescape_url_loop:wnn

```

The `__str_convert_unescape_name:` function replaces each occurrence of `#` followed by two hexadecimal digits in `\g__str_result_tl` by the corresponding byte. The `url` function is identical, with escape character `%` instead of `#`. Thus we define the two together. The arguments of `__str_tmp:w` are the character code of `#` or `%` in hexadecimal, the name of the main function to define, and the name of the auxiliary which performs the loop.

The looping auxiliary `#3` finds the next escape character, reads the following two characters, and tests them. The test `__str_hexadecimal_use:NTF` leaves the upper-case digit in the input stream, hence we surround the test with `__str_output_byte:w` and `__str_output_end:`. If both characters are hexadecimal digits, they should be removed before looping: this is done by `\use_i:nnn`. If one of the characters is not a hexadecimal digit, then feed `"#1` to `__str_output_byte:w` to produce the escape character, raise the flag, and call the looping function followed by the two characters (remove `\use_i:nnn`).

```

5938 \cs_set_protected:Npn \__str_tmp:w #1#2#3
5939  {
5940      \cs_new_protected:cpx { \__str_convert_unescape_#2: }
5941      {
5942          \group_begin:
5943          \flag_clear:n { str_byte }

```

```

5944 \flag_clear:n { str_error }
5945 \int_set:Nn \tex_escapechar:D { 92 }
5946 \__kernel_tl_gset:Nx \g__str_result_tl
5947 {
5948   \exp_after:wN #3 \g__str_result_tl
5949   #1 ? { ? \prg_break: }
5950   \prg_break_point:
5951 }
5952 \__str_if_flag_error:nnx { str_byte } { non-byte } { #2 }
5953 \__str_if_flag_error:nnx { str_error } { unescape-#2 } { }
5954 \group_end:
5955 }
5956 \cs_new:Npn #3 ##1#1##2##3
5957 {
5958   \__str_filter_bytes:n {##1}
5959   \use_none:n ##3
5960   \__str_output_byte:w "
5961   \__str_hexadecimal_use:NTF ##2
5962   {
5963     \__str_hexadecimal_use:NTF ##3
5964     {
5965       \flag_raise:n { str_error }
5966       * 0 + '#1 \use_i:nn
5967     }
5968   }
5969   {
5970     \flag_raise:n { str_error }
5971     0 + '#1 \use_i:nn
5972   }
5973   \__str_output_end:
5974   \use_i:nnn #3 ##2##3
5975 }
5976 \__kernel_msg_new:nnnn { str } { unescape-#2 }
5977 { String-invalid-in-escaping-'#2'. }
5978 {
5979   LaTeX-came-across-the~escape-character-'#1'~not~followed~by~
5980   two~hexadecimal~digits.~This~is~invalid~in~the~escaping-'#2'.
5981 }
5982 }
5983 }
5984 \exp_after:wN \__str_tmp:w \c_hash_str { name }
5985   \__str_unescape_name_loop:wNN
5986 \exp_after:wN \__str_tmp:w \c_percent_str { url }
5987   \__str_unescape_url_loop:wNN

```

(End definition for `__str_convert_unescape_name:` and others.)

```

\__str_convert_unescape_string:
\__str_unescape_string_newlines:wN
\__str_unescape_string_loop:wNNN
\__str_unescape_string_repeat>NNNNNN

```

The `string` escaping is somewhat similar to the `name` and `url` escapings, with escape character `\`. The first step is to convert all three line endings, `^J`, `^M`, and `^M^J` to the common `^J`, as per the PDF specification. This step cannot raise the flag.

Then the following escape sequences are decoded.

`\n` Line feed (10)

`\r` Carriage return (13)

```

\t Horizontal tab (9)
\b Backspace (8)
\f Form feed (12)
\l Left parenthesis
\r Right parenthesis
\\ Backslash

\ddd (backslash followed by 1 to 3 octal digits) Byte ddd (octal), subtracting 256 in case
of overflow.

```

If followed by an end-of-line character, the backslash and the end-of-line are ignored. If followed by anything else, the backslash is ignored, raising the error flag.

```

5988 \group_begin:
5989   \char_set_catcode_other:N \^^J
5990   \char_set_catcode_other:N \^^M
5991   \cs_set_protected:Npn \__str_tmp:w #1
5992   {
5993     \cs_new_protected:Npn \__str_convert_unescape_string:
5994     {
5995       \group_begin:
5996         \flag_clear:n { str_byte }
5997         \flag_clear:n { str_error }
5998         \int_set:Nn \tex_escapechar:D { 92 }
5999         \__kernel_tl_gset:Nx \g__str_result_tl
6000         {
6001           \exp_after:wN \__str_unescape_string_newlines:wN
6002             \g__str_result_tl \prg_break: ^^M ?
6003             \prg_break_point:
6004           }
6005         \__kernel_tl_gset:Nx \g__str_result_tl
6006         {
6007           \exp_after:wN \__str_unescape_string_loop:wNNN
6008             \g__str_result_tl #1 ?? { ? \prg_break: }
6009             \prg_break_point:
6010           }
6011           \__str_if_flag_error:nnx { str_byte } { non-byte } { string }
6012           \__str_if_flag_error:nnx { str_error } { unescape-string } { }
6013         \group_end:
6014       }
6015     }
6016   \exp_args:No \__str_tmp:w { \c_backslash_str }
6017   \exp_last_unbraced:NNNNo
6018   \cs_new:Npn \__str_unescape_string_loop:wNNN #1 \c_backslash_str #2#3#4
6019   {
6020     \__str_filter_bytes:n {#1}
6021     \use_none:n #4
6022     \__str_output_byte:w ,
6023       \__str_octal_use:NTF #2
6024       {
6025         \__str_octal_use:NTF #3

```

```

6026   {
6027     \__str_octal_use:NTF #4
6028     {
6029       \if_int_compare:w #2 > 3 \exp_stop_f:
6030         - 256
6031       \fi:
6032       \__str_unescape_string_repeat:NNNNNN
6033     }
6034     { \__str_unescape_string_repeat:NNNNNN ? }
6035   }
6036   { \__str_unescape_string_repeat:NNNNNN ?? }
6037 }
6038 {
6039   \str_case_e:nnF {#2}
6040   {
6041     { \c_backslash_str } { 134 }
6042     { ( ) { 50 }
6043     { ) } { 51 }
6044     { r } { 15 }
6045     { f } { 14 }
6046     { n } { 12 }
6047     { t } { 11 }
6048     { b } { 10 }
6049     { ^^J } { 0 - 1 }
6050   }
6051   {
6052     \flag_raise:n { str_error }
6053     0 - 1 \use_i:nn
6054   }
6055 }
6056 \__str_output_end:
6057 \use_i:nn \__str_unescape_string_loop:wNNN #2#3#4
6058 }
6059 \cs_new:Npn \__str_unescape_string_repeat:NNNNNN #1#2#3#4#5#6
6060   { \__str_output_end: \__str_unescape_string_loop:wNNN }
6061 \cs_new:Npn \__str_unescape_string_newlines:wN #1 ^^M #2
6062   {
6063     #1
6064     \if_charcode:w ^^J #2 \else: ^^J \fi:
6065     \__str_unescape_string_newlines:wN #2
6066   }
6067 \__kernel_msg_new:nnnn { str } { unescape-string }
6068   { String~invalid~in~escaping~'string'. }
6069   {
6070     LaTeX~came~across~an~escape~character~'\c_backslash_str'~
6071     not~followed~by~any~of:~'n',~'r',~'t',~'b',~'f',~'(',~,~')',~,~'
6072     '\c_backslash_str',~one~to~three~octal~digits,~or~the~end~
6073     of~a~line.
6074   }
6075 \group_end:

```

(End definition for `__str_convert_unescape_string:` and others.)

10.5.2 Escape methods

Currently, none of the escape methods can lead to errors, assuming that their input is made out of bytes.

```
\__str_convert_escape_hex:  
  \__str_escape_hex_char:N  
  
 6076  \cs_new_protected:Npn \__str_convert_escape_hex:  
 6077    { \__str_convert_gmap:N \__str_escape_hex_char:N }  
 6078  \cs_new:Npn \__str_escape_hex_char:N #1  
 6079    { \__str_output_hexadecimal:n { '#1' } }  
  
(End definition for \__str_convert_escape_hex: and \__str_escape_hex_char:N.)  
  
\__str_convert_escape_name:  
  \__str_escape_name_char:n  
  \__str_if_escape_name:ntf  
    \c__str_escape_name_str  
  \c__str_escape_name_not_str  
  
 6080  \str_const:Nn \c__str_escape_name_not_str { ! " $ & ' } %$  
 6081  \str_const:Nn \c__str_escape_name_str { {} / > [ ] }  
 6082  \cs_new_protected:Npn \__str_convert_escape_name:  
 6083    { \__str_convert_gmap:N \__str_escape_name_char:n }  
 6084  \cs_new:Npn \__str_escape_name_char:n #1  
 6085    {  
 6086      \__str_if_escape_name:ntf { '#1' } { '#1' }  
 6087      { \c_hash_str \__str_output_hexadecimal:n { '#1' } }  
 6088    }  
 6089  \prg_new_conditional:Npnn \__str_if_escape_name:n #1 { TF }  
 6090    {  
 6091      \if_int_compare:w '#1 < "2A \exp_stop_f:  
 6092        \__str_if_contains_char:NnTF \c__str_escape_name_not_str { '#1' }  
 6093        \prg_return_true: \prg_return_false:  
 6094      \else:  
 6095        \if_int_compare:w '#1 > "7E \exp_stop_f:  
 6096          \prg_return_false:  
 6097        \else:  
 6098          \__str_if_contains_char:NnTF \c__str_escape_name_str { '#1' }  
 6099          \prg_return_false: \prg_return_true:  
 6100        \fi:  
 6101      \fi:  
 6102    }  
  
(End definition for \__str_convert_escape_name: and others.)
```

Any character below (and including) space, and any character above (and including) `del`, are converted to octal. One backslash is added before each parenthesis and backslash.

```
  \__str_convert_escape_string:  
  \__str_escape_string_char:N  
  \__str_if_escape_string:NTF  
    \c__str_escape_string_str  
  
 6103  \str_const:Nx \c__str_escape_string_str  
 6104    { \c_backslash_str ( ) }  
 6105  \cs_new_protected:Npn \__str_convert_escape_string:  
 6106    { \__str_convert_gmap:N \__str_escape_string_char:N }  
 6107  \cs_new:Npn \__str_escape_string_char:N #1  
 6108    {  
 6109      \__str_if_escape_string:NTF #1  
 6110      {  
 6111        \__str_if_contains_char:NnT
```

```

6112     \c__str_escape_string_str {#1}
6113     { \c_backslash_str }
6114     #1
6115   }
6116   {
6117     \c_backslash_str
6118     \int_div_truncate:nn {'#1} {64}
6119     \int_mod:nn { \int_div_truncate:nn {'#1} { 8 } } { 8 }
6120     \int_mod:nn {'#1} { 8 }
6121   }
6122 }
6123 \prg_new_conditional:Npnn \__str_if_escape_string:N #1 { TF }
6124 {
6125   \if_int_compare:w '#1 < "21 \exp_stop_f:
6126     \prg_return_false:
6127   \else:
6128     \if_int_compare:w '#1 > "7E \exp_stop_f:
6129       \prg_return_false:
6130     \else:
6131       \prg_return_true:
6132     \fi:
6133   \fi:
6134 }

```

(End definition for `__str_convert_escape_string:` and others.)

`__str_convert_escape_url:` This function is similar to `__str_convert_escape_name:`, escaping different characters.

```

6135 \cs_new_protected:Npn \__str_convert_escape_url:
6136   { \__str_convert_gmap:N \__str_escape_url_char:n }
6137 \cs_new:Npn \__str_escape_url_char:n #1
6138   {
6139     \__str_if_escape_url:nTF {#1} {#1}
6140     { \c_percent_str \__str_output_hexadecimal:n { '#1 } }
6141   }
6142 \prg_new_conditional:Npnn \__str_if_escape_url:n #1 { TF }
6143 {
6144   \if_int_compare:w '#1 < "41 \exp_stop_f:
6145     \__str_if_contains_char:nnTF { "-.<>" } {#1}
6146       \prg_return_true: \prg_return_false:
6147     \else:
6148       \if_int_compare:w '#1 > "7E \exp_stop_f:
6149         \prg_return_false:
6150       \else:
6151         \__str_if_contains_char:nnTF { [ ] } {#1}
6152           \prg_return_false: \prg_return_true:
6153         \fi:
6154       \fi:
6155 }

```

(End definition for `__str_convert_escape_url:`, `__str_escape_url_char:n`, and `__str_if_escape_url:nTF`.)

10.6 Encoding definitions

The `native` encoding is automatically defined. Other encodings are loaded as needed. The following encodings are supported:

- UTF-8;
- UTF-16, big-, little-endian, or with byte order mark;
- UTF-32, big-, little-endian, or with byte order mark;
- the ISO 8859 code pages, numbered from 1 to 16, skipping the nonexistent ISO 8859-12.

10.6.1 utf-8 support

Loop through the internal string, and convert each character to its UTF-8 representation. The representation is built from the right-most (least significant) byte to the left-most (most significant) byte. Continuation bytes are in the range [128, 191], taking 64 different values, hence we roughly want to express the character code in base 64, shifting the first digit in the representation by some number depending on how many continuation bytes there are. In the range [0, 127], output the corresponding byte directly. In the range [128, 2047], output the remainder modulo 64, plus 128 as a continuation byte, then output the quotient (which is in the range [0, 31]), shifted by 192. In the next range, [2048, 65535], split the character code into residue and quotient modulo 64, output the residue as a first continuation byte, then repeat; this leaves us with a quotient in the range [0, 15], which we output shifted by 224. The last range, [65536, 1114111], follows the same pattern: once we realize that dividing twice by 64 leaves us with a number larger than 15, we repeat, producing a last continuation byte, and offset the quotient by 240 for the leading byte.

How is that implemented? `_str_encode_utf_vii_loop:wwnnw` takes successive quotients as its first argument, the quotient from the previous step as its second argument (except in step 1), the bound for quotients that trigger one more step or not, and finally the offset used if this step should produce the leading byte. Leading bytes can be in the ranges [0, 127], [192, 223], [224, 239], and [240, 247] (really, that last limit should be 244 because Unicode stops at the code point 1114111). At each step, if the quotient #1 is less than the limit #3 for that range, output the leading byte (#1 shifted by #4) and stop. Otherwise, we need one more step: use the quotient of #1 by 64, and #1 as arguments for the looping auxiliary, and output the continuation byte corresponding to the remainder #2 - 64#1 + 128. The bizarre construction `- 1 + 0 * removes the spurious initial continuation byte` (better methods welcome).

```
6156 \cs_new_protected:cpx { __str_convert_encode_utf8: }
6157   { \__str_convert_gmap_internal:N \__str_encode_utf_viii_char:n }
6158 \cs_new:Npn \__str_encode_utf_viii_char:n #1
6159   {
6160     \__str_encode_utf_viii_loop:wwnnw #1 ; - 1 + 0 * ;
6161     { 128 } { 0 }
6162     { 32 } { 192 }
6163     { 16 } { 224 }
6164     { 8 } { 240 }
6165     \s__str_stop
6166   }
6167 \cs_new:Npn \__str_encode_utf_viii_loop:wwnnw #1; #2; #3#4 #5 \s__str_stop
```

```

6168  {
6169    \if_int_compare:w #1 < #3 \exp_stop_f:
6170      \__str_output_byte:n { #1 + #4 }
6171      \exp_after:wN \__str_use_none_delimit_by_s_stop:w
6172    \fi:
6173    \exp_after:wN \__str_encode_utf_viii_loop:wwnnw
6174      \int_value:w \int_div_truncate:nn {#1} {64} ; #1 ;
6175      #5 \s__str_stop
6176      \__str_output_byte:n { #2 - 64 * ( #1 - 2 ) }
6177  }

```

(End definition for `__str_convert_encode_utf8:`, `__str_encode_utf_viii_char:n`, and `__str_encode_utf_viii_loop:wwnnw`)

`\l__str_missing_flag`
`\l__str_extra_flag`
`\l__str_overlong_flag`
`\l__str_overflow_flag`

When decoding a string that is purportedly in the UTF-8 encoding, four different errors can occur, signalled by a specific flag for each (we define those flags using `\flag_clear_new:n` rather than `\flag_new:n`, because they are shared with other encoding definition files).

- “Missing continuation byte”: a leading byte is not followed by the right number of continuation bytes.
- “Extra continuation byte”: a continuation byte appears where it was not expected, *i.e.*, not after an appropriate leading byte.
- “Overlong”: a Unicode character is expressed using more bytes than necessary, for instance, "C0"80 for the code point 0, instead of a single null byte.
- “Overflow”: this occurs when decoding produces Unicode code points greater than 1114111.

We only raise one L^AT_EX3 error message, combining all the errors which occurred. In the short message, the leading comma must be removed to get a grammatically correct sentence. In the long text, first remind the user what a correct UTF-8 string should look like, then add error-specific information.

```

6178 \flag_clear_new:n { str_missing }
6179 \flag_clear_new:n { str_extra }
6180 \flag_clear_new:n { str_overlong }
6181 \flag_clear_new:n { str_overflow }
6182 \__kernel_msg_new:nnnn { str } { utf8-decode }
6183  {
6184    Invalid~UTF-8~string:
6185    \exp_last_unbraced:Nf \use_none:n
6186    {
6187      \__str_if_flag_times:nT { str_missing } { ,~missing~continuation~byte }
6188      \__str_if_flag_times:nT { str_extra } { ,~extra~continuation~byte }
6189      \__str_if_flag_times:nT { str_overlong } { ,~overlong~form }
6190      \__str_if_flag_times:nT { str_overflow } { ,~code~point~too~large }
6191    }
6192    .
6193  }
6194  {
6195    In~the~UTF-8~encoding,~each~Unicode~character~consists~in~
6196    1-to~4~bytes,~with~the~following~bit~pattern: \\~\iow_indent:n
6197

```

```

6198 {
6199   Code-point~\ \ \ \ <~128:~0xxxxxx \\ 
6200   Code-point~\ \ \ <~2048:~110xxxx~10xxxxxx \\ 
6201   Code-point~\ \ <~65536:~1110xxx~10xxxxxx~10xxxxxx \\ 
6202   Code-point~ <~1114112:~11110xx~10xxxxxx~10xxxxxx~10xxxxxx \\ 
6203 }
6204 Bytes-of-the-form-10xxxxxx-are-called-continuation-bytes.
6205 \flag_if_raised:nT { str_missing }
6206 {
6207   \\\\
6208   A-leading-byte-(in-the-range-[192,255])-was-not-followed-by-
6209   the-appropriate-number-of-continuation-bytes.
6210 }
6211 \flag_if_raised:nT { str_extra }
6212 {
6213   \\\\
6214   LaTeX-came-across-a-continuation-byte-when-it-was-not-expected.
6215 }
6216 \flag_if_raised:nT { str_overlong }
6217 {
6218   \\\\
6219   Every-Unicode-code-point-must-be-expressed-in-the-shortest-
6220   possible-form.-For-instance,-'0xC0'-'0x83'-is-not-a-valid-
6221   representation-for-the-code-point-3.
6222 }
6223 \flag_if_raised:nT { str_overflow }
6224 {
6225   \\\\
6226   Unicode-limits-code-points-to-the-range-[0,1114111].
6227 }
6228 }

```

(End definition for `\l_str_missing_flag` and others.)

```

__str_convert_decode_utf8:
  __str_decode_utf_viii_start:N
  __str_decode_utf_viii_continuation:wwN
    __str_decode_utf_viii_aux:wNnwN
    __str_decode_utf_viii_overflow:w
  __str_decode_utf_viii_end:

```

Decoding is significantly harder than encoding. As before, lower some flags, which are tested at the end (in bulk, to trigger at most one L^AT_EX3 error, as explained above). We expect successive multi-byte sequences of the form *<start byte> <continuation bytes>*. The `_start` auxiliary tests the first byte:

- [0, "7F]: the byte stands alone, and is converted to its own character code;
- ["80, "BF]: unexpected continuation byte, raise the appropriate flag, and convert that byte to the replacement character "FFFD";
- ["C0, "FF]: this byte should be followed by some continuation byte(s).

In the first two cases, `\use_none_delimit_by_q_stop:w` removes data that only the third case requires, namely the limits of ranges of Unicode characters which can be expressed with 1, 2, 3, or 4 bytes.

We can now concentrate on the multi-byte case and the `_continuation` auxiliary. We expect #3 to be in the range ["80, "BF]. The test for this goes as follows: if the character code is less than "80, we compare it to ="C0, yielding `false`; otherwise to "C0, yielding `true` in the range ["80, "BF] and `false` otherwise. If we find that the byte is not a continuation range, stop the current slew of bytes, output the replacement

character, and continue parsing with the `_start` auxiliary, starting at the byte we just tested. Once we know that the byte is a continuation byte, leave it behind us in the input stream, compute what code point the bytes read so far would produce, and feed that number to the `_aux` function.

The `_aux` function tests whether we should look for more continuation bytes or not. If the number it receives as #1 is less than the maximum #4 for the current range, then we are done: check for an overlong representation by comparing #1 with the maximum #3 for the previous range. Otherwise, we call the `_continuation` auxiliary again, after shifting the “current code point” by #4 (maximum from the range we just checked).

Two additional tests are needed: if we reach the end of the list of range maxima and we are still not done, then we are faced with an overflow. Clean up, and again insert the code point "FFFD for the replacement character. Also, every time we read a byte, we need to check whether we reached the end of the string. In a correct UTF-8 string, this happens automatically when the `_start` auxiliary leaves its first argument in the input stream: the end-marker begins with `\prg_break:`, which ends the loop. On the other hand, if the end is reached when looking for a continuation byte, the `\use_none:n` #3 construction removes the first token from the end-marker, and leaves the `_end` auxiliary, which raises the appropriate error flag before ending the mapping.

```

6229 \cs_new_protected:cpn { __str_convert_decode_utf8: }
6300 {
6301     \flag_clear:n { str_error }
6302     \flag_clear:n { str_missing }
6303     \flag_clear:n { str_extra }
6304     \flag_clear:n { str_overlong }
6305     \flag_clear:n { str_overflow }
6306     \__kernel_tl_gset:Nx \g__str_result_tl
6307     {
6308         \exp_after:wN \__str_decode_utf_viii_start:N \g__str_result_tl
6309         { \prg_break: \__str_decode_utf_viii_end: }
6310         \prg_break_point:
6311     }
6312     \__str_if_flag_error:nnx { str_error } { utf8-decode } { }
6313 }
6314 \cs_new:Npn \__str_decode_utf_viii_start:N #1
6315 {
6316     #1
6317     \if_int_compare:w '#1 < "C0 \exp_stop_f:
6318         \s__str
6319         \if_int_compare:w '#1 < "80 \exp_stop_f:
6320             \int_value:w '#1
6321         \else:
6322             \flag_raise:n { str_extra }
6323             \flag_raise:n { str_error }
6324             \int_use:N \c__str_replacement_char_int
6325         \fi:
6326     \else:
6327         \exp_after:wN \__str_decode_utf_viii_continuation:wwN
6328         \int_value:w \int_eval:n { '#1 - "C0 } \exp_after:wN
6329     \fi:
6330     \s__str
6331     \__str_use_none_delimit_by_s_stop:w {"80} {"800} {"10000} {"110000} \s__str_stop
6332     \__str_decode_utf_viii_start:N

```

```

6263     }
6264 \cs_new:Npn \__str_decode_utf_viii_continuation:wwN
6265     #1 \s__str #2 \__str_decode_utf_viii_start:N #3
6266 {
6267     \use_none:n #3
6268     \if_int_compare:w '#3 <
6269         \if_int_compare:w '#3 < "80 \exp_stop_f: - \fi:
6270             "C0 \exp_stop_f:
6271             #3
6272             \exp_after:wN \__str_decode_utf_viii_aux:wNnnwN
6273             \int_value:w \int_eval:n { #1 * "40 + '#3 - "80 } \exp_after:wN
6274 \else:
6275     \s__str
6276     \flag_raise:n { str_missing }
6277     \flag_raise:n { str_error }
6278     \int_use:N \c__str_replacement_char_int
6279 \fi:
6280 \s__str
6281 #2
6282     \__str_decode_utf_viii_start:N #3
6283 }
6284 \cs_new:Npn \__str_decode_utf_viii_aux:wNnnwN
6285     #1 \s__str #2#3#4 #5 \__str_decode_utf_viii_start:N #6
6286 {
6287     \if_int_compare:w #1 < #4 \exp_stop_f:
6288         \s__str
6289         \if_int_compare:w #1 < #3 \exp_stop_f:
6290             \flag_raise:n { str_overlong }
6291             \flag_raise:n { str_error }
6292             \int_use:N \c__str_replacement_char_int
6293 \else:
6294     #1
6295 \fi:
6296 \else:
6297     \if_meaning:w \s__str_stop #5
6298         \__str_decode_utf_viii_overflow:w #1
6299     \fi:
6300     \exp_after:wN \__str_decode_utf_viii_continuation:wwN
6301     \int_value:w \int_eval:n { #1 - #4 } \exp_after:wN
6302 \fi:
6303 \s__str
6304 #2 {#4} #5
6305     \__str_decode_utf_viii_start:N
6306 }
6307 \cs_new:Npn \__str_decode_utf_viii_overflow:w #1 \fi: #2 \fi:
6308 {
6309     \fi: \fi:
6310     \flag_raise:n { str_overflow }
6311     \flag_raise:n { str_error }
6312     \int_use:N \c__str_replacement_char_int
6313 }
6314 \cs_new:Npn \__str_decode_utf_viii_end:
6315 {
6316     \s__str

```

```

6317     \flag_raise:n { str_missing }
6318     \flag_raise:n { str_error }
6319     \int_use:N \c__str_replacement_char_int \s__str
6320     \prg_break:
6321 }

```

(End definition for `_str_convert_decode_utf8:` and others.)

10.6.2 utf-16 support

The definitions are done in a category code regime where the bytes 254 and 255 used by the byte order mark have catcode 12.

```

6322 \group_begin:
6323   \char_set_catcode_other:N \^fe
6324   \char_set_catcode_other:N \^ff

```

When the endianness is not specified, it is big-endian by default, and we add a byte-order mark. Convert characters one by one in a loop, with different behaviours depending on the character code.

- [0, "D7FF]: converted to two bytes;
- ["D800, "DFFF] are used as surrogates: they cannot be converted and are replaced by the replacement character;
- ["E000, "FFFF]: converted to two bytes;
- ["10000, "10FFFF]: converted to a pair of surrogates, each two bytes. The magic "D7C0 is "D800 - "10000/"400.

For the duration of this operation, `_str_tmp:w` is defined as a function to convert a number in the range [0, "FFFF] to a pair of bytes (either big endian or little endian), by feeding the quotient of the division of #1 by "100, followed by #1 to `_str_encode_utf_xvi_be:nn` or its `le` analog: those compute the remainder, and output two bytes for the quotient and remainder.

```

6325   \cs_new_protected:cpn { __str_convert_encode_utf16: }
6326   {
6327     \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n
6328     \tl_gput_left:Nx \g__str_result_tl { ^fe ^ff }
6329   }
6330   \cs_new_protected:cpn { __str_convert_encode_utf16be: }
6331   { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n }
6332   \cs_new_protected:cpn { __str_convert_encode_utf16le: }
6333   { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_le:n }
6334   \cs_new_protected:Npn \__str_encode_utf_xvi_aux:N #1
6335   {
6336     \flag_clear:n { str_error }
6337     \cs_set_eq:NN \__str_tmp:w #1
6338     \__str_convert_gmap_internal:N \__str_encode_utf_xvi_char:n
6339     \__str_if_flag_error:nnx { str_error } { utf16-encode } { }
6340   }
6341   \cs_new:Npn \__str_encode_utf_xvi_char:n #1
6342   {
6343     \if_int_compare:w #1 < "D800 \exp_stop_f:

```

```

6344     \__str_tmp:w {#1}
6345 \else:
6346     \if_int_compare:w #1 < "10000 \exp_stop_f:
6347     \if_int_compare:w #1 < "E000 \exp_stop_f:
6348         \flag_raise:n { str_error }
6349         \__str_tmp:w { \c__str_replacement_char_int }
6350     \else:
6351         \__str_tmp:w {#1}
6352     \fi:
6353 \else:
6354     \exp_args:Nf \__str_tmp:w { \int_div_truncate:nn {#1} {"400} + "D7C0 }
6355     \exp_args:Nf \__str_tmp:w { \int_mod:nn {#1} {"400} + "DC00 }
6356     \fi:
6357 \fi:
6358 }

```

(End definition for `__str_convert_encode_utf16`: and others.)

`\l__str_missing_flag`
`\l__str_extra_flag`
`\l__str_end_flag`

When encoding a Unicode string to UTF-16, only one error can occur: code points in the range ["D800, "DFFF], corresponding to surrogates, cannot be encoded. We use the all-purpose flag `@@_error` to signal that error.

When decoding a Unicode string which is purportedly in UTF-16, three errors can occur: a missing trail surrogate, an unexpected trail surrogate, and a string containing an odd number of bytes.

```

6359     \flag_clear_new:n { str_missing }
6360     \flag_clear_new:n { str_extra }
6361     \flag_clear_new:n { str_end }
6362     \__kernel_msg_new:nnnn { str } { utf16-encode }
6363     { Unicode~string~cannot~be~expressed~in~UTF-16:~surrogate. }
6364     {
6365         Surrogate~code~points~(in~the~range~[U+D800,~U+DFFF])~
6366         can~be~expressed~in~the~UTF-8~and~UTF-32~encodings,~
6367         but~not~in~the~UTF-16~encoding.
6368     }
6369     \__kernel_msg_new:nnnn { str } { utf16-decode }
6370     {
6371         Invalid~UTF-16~string:
6372         \exp_last_unbraced:Nf \use_none:n
6373         {
6374             \__str_if_flag_times:nT { str_missing } { ,~missing~trail~surrogate }
6375             \__str_if_flag_times:nT { str_extra } { ,~extra~trail~surrogate }
6376             \__str_if_flag_times:nT { str_end } { ,~odd~number~of~bytes }
6377         }
6378     .
6379     {
6380         In~the~UTF-16~encoding,~each~Unicode~character~is~encoded~as~
6381         2~or~4~bytes: \\%
6382         \iow_indent:n
6383         {
6384             Code~point~in~[U+0000,~U+D7FF]:~two~bytes \\
6385             Code~point~in~[U+D800,~U+DFFF]:~illegal \\
6386             Code~point~in~[U+E000,~U+FFFF]:~two~bytes \\
6387             Code~point~in~[U+10000,~U+10FFFF]:~illegal
6388     }

```

```

6389      a~lead~surrogate~and~a~trail~surrogate \\
6390      }
6391 Lead~surrogates~are~pairs~of~bytes~in~the~range~[0xD800,~0xDBFF],~
6392 and~trail~surrogates~are~in~the~range~[0xDC00,~0xFFFF].
6393 \flag_if_raised:nT { str_missing }
6394 {
6395   \\\\
6396   A~lead~surrogate~was~not~followed~by~a~trail~surrogate.
6397 }
6398 \flag_if_raised:nT { str_extra }
6399 {
6400   \\\\
6401   LaTeX~came~across~a~trail~surrogate~when~it~was~not~expected.
6402 }
6403 \flag_if_raised:nT { str_end }
6404 {
6405   \\\\
6406   The~string~contained~an~odd~number~of~bytes.~This~is~invalid:~
6407   the~basic~code~unit~for~UTF-16~is~16-bits~(2~bytes).
6408 }
6409 }

```

(End definition for `\l__str_missing_flag`, `\l__str_extra_flag`, and `\l__str_end_flag`.)

```

\__str_convert_decode_utf16:
  \__str_convert_decode_utf16be:
  \__str_convert_decode_utf16le:
\__str_decode_utf_xvi_bom>NN
\__str_decode_utf_xvi:Nw

```

As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are "FE" and "FF" in either order, remove them and use the corresponding endianness, otherwise assume big-endianness. The three endianness cases are based on a common auxiliary whose first argument is 1 for big-endian and 2 for little-endian, and whose second argument, delimited by the scan mark `\s__str_stop`, is expanded once (the string may be long; passing `\g__str_result_tl` as an argument before expansion is cheaper).

The `__str_decode_utf_xvi:Nw` function defines `__str_tmp:w` to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using `__str_decode_utf_xvi_pair>NN` described below.

```

6410 \cs_new_protected:cpn { __str_convert_decode_utf16be: }
6411   { \__str_decode_utf_xvi:Nw 1 \g__str_result_tl \s__str_stop }
6412 \cs_new_protected:cpn { __str_convert_decode_utf16le: }
6413   { \__str_decode_utf_xvi:Nw 2 \g__str_result_tl \s__str_stop }
6414 \cs_new_protected:cpn { __str_convert_decode_utf16: }
6415 {
6416   \exp_after:wN \__str_decode_utf_xvi_bom>NN
6417   \g__str_result_tl \s__str_stop \s__str_stop \s__str_stop
6418 }
6419 \cs_new_protected:Npn \__str_decode_utf_xvi_bom>NN #1#2
6420 {
6421   \str_if_eq:nnTF { #1#2 } { ^ff ^fe }
6422   { \__str_decode_utf_xvi:Nw 2 }
6423   {
6424     \str_if_eq:nnTF { #1#2 } { ^fe ^ff }
6425     { \__str_decode_utf_xvi:Nw 1 }
6426     { \__str_decode_utf_xvi:Nw 1 #1#2 }
6427 }

```

```

6428     }
6429 \cs_new_protected:Npn \__str_decode_utf_xvi:Nw #1#2 \s__str_stop
6430 {
6431     \flag_clear:n { str_error }
6432     \flag_clear:n { str_missing }
6433     \flag_clear:n { str_extra }
6434     \flag_clear:n { str_end }
6435     \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
6436     \kernel_tl_gset:Nx \g__str_result_tl
6437     {
6438         \exp_after:wn \__str_decode_utf_xvi_pair>NN
6439             #2 \q__str_nil \q__str_nil
6440         \prg_break_point:
6441     }
6442     \__str_if_flag_error:n { str_error } { utf16-decode } { }
6443 }

```

(End definition for `__str_convert_decode_utf16`: and others.)

```

\__str_decode_utf_xvi_pair:NN
\__str_decode_utf_xvi_quad>NNwNN
\__str_decode_utf_xvi_pair_end:Nw
\__str_decode_utf_xvi_error:nNN
\__str_decode_utf_xvi_extra>NNw

```

Bytes are read two at a time. At this stage, `\@@_tmp:w #1#2` expands to the character code of the most significant byte, and we distinguish cases depending on which range it lies in:

- ["D8, "DB] signals a lead surrogate, and the integer expression yields 1 (ε -T_EX rounds ties away from zero);
- ["DC, "DF] signals a trail surrogate, unexpected here, and the integer expression yields 2;
- any other value signals a code point in the Basic Multilingual Plane, which stands for itself, and the `\if_case:w` construction expands to nothing (cases other than 1 or 2), leaving the relevant material in the input stream, followed by another call to the `_pair` auxiliary.

The case of a lead surrogate is treated by the `_quad` auxiliary, whose arguments #1, #2, #4 and #5 are the four bytes. We expect the most significant byte of #4#5 to be in the range ["DC, "DF] (trail surrogate). The test is similar to the test used for continuation bytes in the UTF-8 decoding functions. In the case where #4#5 is indeed a trail surrogate, leave #1#2#4#5 `\s__str <code point> \s__str`, and remove the pair #4#5 before looping with `__str_decode_utf_xvi_pair>NN`. Otherwise, of course, complain about the missing surrogate.

The magic number "D7F7 is such that "D7F7*"400 = "D800*"400+"DC00—"10000.

Every time we read a pair of bytes, we test for the end-marker `\q__str_nil`. When reaching the end, we additionally check that the string had an even length. Also, if the end is reached when expecting a trail surrogate, we treat that as a missing surrogate.

```

6444 \cs_new:Npn \__str_decode_utf_xvi_pair:NN #1#2
6445 {
6446     \if_meaning:w \q__str_nil #2
6447         \__str_decode_utf_xvi_pair_end:Nw #1
6448     \fi:
6449     \if_case:w
6450         \int_eval:n { ( \__str_tmp:w #1#2 - "D6 ) / 4 } \scan_stop:
6451     \or: \exp_after:wn \__str_decode_utf_xvi_quad>NNwNN
6452     \or: \exp_after:wn \__str_decode_utf_xvi_extra>NNw

```

```

6453   \fi:
6454   #1#2 \s__str
6455   \int_eval:n { "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 } \s__str
6456   \__str_decode_utf_xvi_pair:NN
6457 }
6458 \cs_new:Npn \__str_decode_utf_xvi_quad:NNwNN
6459   #1#2 #3 \__str_decode_utf_xvi_pair:NN #4#5
6460 {
6461   \if_meaning:w \q__str_nil #5
6462     \__str_decode_utf_xvi_error:nNN { missing } #1#2
6463     \__str_decode_utf_xvi_pair_end:Nw #4
6464   \fi:
6465   \if_int_compare:w
6466     \if_int_compare:w \__str_tmp:w #4#5 < "DC \exp_stop_f:
6467       0 = 1
6468     \else:
6469       \__str_tmp:w #4#5 < "E0
6470     \fi:
6471     \exp_stop_f:
6472     #1 #2 #4 #5 \s__str
6473   \int_eval:n
6474   {
6475     ( "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 - "D7F7 ) * "400
6476     + "100 * \__str_tmp:w #4#5 + \__str_tmp:w #5#4
6477   }
6478   \s__str
6479   \exp_after:wN \use_i:nnn
6480 \else:
6481   \__str_decode_utf_xvi_error:nNN { missing } #1#2
6482 \fi:
6483   \__str_decode_utf_xvi_pair:NN #4#5
6484 }
6485 \cs_new:Npn \__str_decode_utf_xvi_pair_end:Nw #1 \fi:
6486 {
6487   \fi:
6488   \if_meaning:w \q__str_nil #1
6489   \else:
6490     \__str_decode_utf_xvi_error:nNN { end } #1 \prg_do_nothing:
6491   \fi:
6492   \prg_break:
6493 }
6494 \cs_new:Npn \__str_decode_utf_xvi_extra>NNw #1#2 \s__str #3 \s__str
6495   { \__str_decode_utf_xvi_error:nNN { extra } #1#2 }
6496 \cs_new:Npn \__str_decode_utf_xvi_error:nNN #1#2#3
6497 {
6498   \flag_raise:n { str_error }
6499   \flag_raise:n { str_#1 }
6500   #2 #3 \s__str
6501   \int_use:N \c__str_replacement_char_int \s__str
6502 }

```

(End definition for __str_decode_utf_xvi_pair:NN and others.)

Restore the original catcodes of bytes 254 and 255.

```
6503 \group_end:
```

10.6.3 utf-32 support

The definitions are done in a category code regime where the bytes 0, 254 and 255 used by the byte order mark have catcode “other”.

```

6504 \group_begin:
6505   \char_set_catcode_other:N \^00
6506   \char_set_catcode_other:N \^fe
6507   \char_set_catcode_other:N \^ff

\_str_convert_encode_utf32:
  \_str_convert_encode_utf32be:
    \_str_convert_encode_utf32le:
      \_str_encode_utf_xxxii_be:n
        \_str_encode_utf_xxxii_be_aux:nn
      \_str_encode_utf_xxxii_le:n
        \_str_encode_utf_xxxii_le_aux:nn

6508 \cs_new_protected:cpn { \_str_convert_encode_utf32: }
6509 {
  \_str_convert_gmap_internal:N \_str_encode_utf_xxxii_be:n
    \tl_gput_left:Nx \g_str_result_tl { ^00 ^00 ^fe ^ff }
}
6513 \cs_new_protected:cpn { \_str_convert_encode_utf32be: }
6514 { \_str_convert_gmap_internal:N \_str_encode_utf_xxxii_be:n }
6515 \cs_new_protected:cpn { \_str_convert_encode_utf32le: }
6516 { \_str_convert_gmap_internal:N \_str_encode_utf_xxxii_le:n }
6517 \cs_new:Npn \_str_encode_utf_xxxii_be:n #1
{
  \exp_args:Nf \_str_encode_utf_xxxii_be_aux:nn
  { \int_div_truncate:nn {#1} { "100 } } {#1}
}
6522 \cs_new:Npn \_str_encode_utf_xxxii_be_aux:nn #1#2
{
  ^00
  \_str_output_byte_pair_be:n {#1}
  \_str_output_byte:n { #2 - #1 * "100 }
}
6528 \cs_new:Npn \_str_encode_utf_xxxii_le:n #1
{
  \exp_args:Nf \_str_encode_utf_xxxii_le_aux:nn
  { \int_div_truncate:nn {#1} { "100 } } {#1}
}
6533 \cs_new:Npn \_str_encode_utf_xxxii_le_aux:nn #1#2
{
  \_str_output_byte:n { #2 - #1 * "100 }
  \_str_output_byte_pair_le:n {#1}
  ^00
}

(End definition for \_str_convert_encode_utf32: and others.)
```

```

str_overflow
str_end
```

There can be no error when encoding in UTF-32. When decoding, the string may not have length $4n$, or it may contain code points larger than "10FFFF". The latter case often happens if the encoding was in fact not UTF-32, because most arbitrary strings are not valid in UTF-32.

```

6539 \flag_clear_new:n { str_overflow }
6540 \flag_clear_new:n { str_end }
6541 \kernel_msg_new:nnnn { str } { utf32-decode }
6542 {
```

```

6543 Invalid~UTF-32~string:
6544 \exp_last_unbraced:Nf \use_none:n
6545 {
6546     \__str_if_flag_times:nT { str_overflow } { ,~code~point~too~large }
6547     \__str_if_flag_times:nT { str_end } { ,~truncated~string }
6548 }
6549 .
6550 }
6551 {
6552     In-the~UTF-32~encoding,~every~Unicode~character~
6553     (in~the~range~[U+0000,~U+10FFFF])~is~encoded~as~4~bytes.
6554     \flag_if_raised:nT { str_overflow }
6555     {
6556         \\\\
6557         LaTeX~came~across~a~code~point~larger~than~1114111,~
6558         the~maximum~code~point~defined~by~Unicode.~
6559         Perhaps~the~string~was~not~encoded~in~the~UTF-32~encoding?
6560     }
6561     \flag_if_raised:nT { str_end }
6562     {
6563         \\\\
6564         The~length~of~the~string~is~not~a~multiple~of~4.~
6565         Perhaps~the~string~was~truncated?
6566     }
6567 }

```

(End definition for `str_overflow` and `str_end`. These variables are documented on page ??.)

```

\__str_convert_decode_utf32:
    \_str_convert_decode_utf32be:
        \_str_convert_decode_utf32le:
    \__str_decode_utf_xxxii_bom:NNNN
\__str_decode_utf_xxxii:Nw
    \__str_decode_utf_xxxii_loop:NNNN
        \__str_decode_utf_xxxii_end:w

```

The structure is similar to UTF-16 decoding functions. If the endianness is not given, test the first 4 bytes of the string (possibly `\s__str_stop` if the string is too short) for the presence of a byte-order mark. If there is a byte-order mark, use that endianness, and remove the 4 bytes, otherwise default to big-endian, and leave the 4 bytes in place. The `__str_decode_utf_xxxii:Nw` auxiliary receives 1 or 2 as its first argument indicating endianness, and the string to convert as its second argument (expanded or not). It sets `__str_tmp:w` to expand to the character code of either of its two arguments depending on endianness, then triggers the `_loop` auxiliary inside an x-expanding assignment to `\g__str_result_tl`.

The `_loop` auxiliary first checks for the end-of-string marker `\s__str_stop`, calling the `_end` auxiliary if appropriate. Otherwise, leave the *(4 bytes)* `\s__str` behind, then check that the code point is not overflowing: the leading byte must be 0, and the following byte at most 16.

In the ending code, we check that there remains no byte: there should be nothing left until the first `\s__str_stop`. Break the map.

```

6568 \cs_new_protected:cpn { __str_convert_decode_utf32be: }
6569     { \__str_decode_utf_xxxii:Nw 1 \g__str_result_tl \s__str_stop }
6570 \cs_new_protected:cpn { __str_convert_decode_utf32le: }
6571     { \__str_decode_utf_xxxii:Nw 2 \g__str_result_tl \s__str_stop }
6572 \cs_new_protected:cpn { __str_convert_decode_utf32: }
6573     {
6574         \exp_after:wN \__str_decode_utf_xxxii_bom:NNNN \g__str_result_tl
6575             \s__str_stop \s__str_stop \s__str_stop \s__str_stop
6576     }
6577 \cs_new_protected:Npn \__str_decode_utf_xxxii_bom:NNNN #1#2#3#4

```

```

6578 {
6579   \str_if_eq:nnTF { #1#2#3#4 } { ^^ff ^~fe ^~00 ^~00 }
6580   { \__str_decode_utf_xxxii:Nw 2 }
6581   {
6582     \str_if_eq:nnTF { #1#2#3#4 } { ^~00 ^~00 ^~fe ^~ff }
6583     { \__str_decode_utf_xxxii:Nw 1 }
6584     { \__str_decode_utf_xxxii:Nw 1 #1#2#3#4 }
6585   }
6586 }
6587 \cs_new_protected:Npn \__str_decode_utf_xxxii:Nw #1#2 \s__str_stop
6588 {
6589   \flag_clear:n { str_overflow }
6590   \flag_clear:n { str_end }
6591   \flag_clear:n { str_error }
6592   \cs_set:Npn \__str_tmp:w ##1 ##2 { ` ## #1 }
6593   \kernel_tl_gset:Nx \g__str_result_tl
6594   {
6595     \exp_after:wN \__str_decode_utf_xxxii_loop:NNNN
6596     #2 \s__str_stop \s__str_stop \s__str_stop \s__str_stop
6597     \prg_break_point:
6598   }
6599   \__str_if_flag_error:nnx { str_error } { utf32-decode } { }
6600 }
6601 \cs_new:Npn \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4
6602 {
6603   \if_meaning:w \s__str_stop #4
6604     \exp_after:wN \__str_decode_utf_xxxii_end:w
6605   \fi:
6606   #1#2#3#4 \s__str
6607   \if_int_compare:w \__str_tmp:w #1#4 > 0 \exp_stop_f:
6608     \flag_raise:n { str_overflow }
6609     \flag_raise:n { str_error }
6610     \int_use:N \c__str_replacement_char_int
6611   \else:
6612     \if_int_compare:w \__str_tmp:w #2#3 > 16 \exp_stop_f:
6613       \flag_raise:n { str_overflow }
6614       \flag_raise:n { str_error }
6615       \int_use:N \c__str_replacement_char_int
6616     \else:
6617       \int_eval:n
6618         { \__str_tmp:w #2#3*"10000 + \__str_tmp:w #3#2*"100 + \__str_tmp:w #4#1 }
6619     \fi:
6620   \fi:
6621   \s__str
6622   \__str_decode_utf_xxxii_loop:NNNN
6623 }
6624 \cs_new:Npn \__str_decode_utf_xxxii_end:w #1 \s__str_stop
6625 {
6626   \tl_if_empty:nF {#1}
6627   {
6628     \flag_raise:n { str_end }
6629     \flag_raise:n { str_error }
6630     #1 \s__str
6631     \int_use:N \c__str_replacement_char_int \s__str

```

```

6632     }
6633   \prg_break:
6634 }

```

(End definition for `_str_convert_decode_utf32: and others.`)
 Restore the original catcodes of bytes 0, 254 and 255.
 6635 \group_end:

10.7 PDF names and strings by expansion

```

\str_convert_pdfname:n
\__str_convert_pdfname:n
  \str_convert_pdfname_bytes:n
  \str_convert_pdfname_bytes_aux:n
\__str_convert_pdfname_bytes_aux:mnn

```

To convert to PDF names by expansion, we work purely on UTF-8 input. The first step is to make a string with “other” spaces, after which we use a simple token-by-token approach. In Unicode engines, we break down everything before one-byte codepoints, but for 8-bit engines there is no need to worry. Actual escaping is covered by the same code as used in the non-expandable route.

```

6636 \cs_new:Npn \str_convert_pdfname:n #1
6637 {
6638   \exp_args:Ne \tl_to_str:n
6639   { \str_map_function:nN {#1} \__str_convert_pdfname:n }
6640 }
6641 \bool_lazy_or:nnTF
6642 { \sys_if_engine_luatex_p: }
6643 { \sys_if_engine_xetex_p: }
6644 {
6645   \cs_new:Npn \__str_convert_pdfname:n #1
6646   {
6647     \int_compare:nNnTF { '#1 } > { "7F" }
6648     { \__str_convert_pdfname_bytes:n {#1} }
6649     { \__str_escape_name_char:n {#1} }
6650   }
6651   \cs_new:Npn \__str_convert_pdfname_bytes:n #1
6652   {
6653     \exp_args:Ne \__str_convert_pdfname_bytes_aux:n
6654     { \char_to_utfviii_bytes:n {'#1} }
6655   }
6656   \cs_new:Npn \__str_convert_pdfname_bytes_aux:n #1
6657   { \__str_convert_pdfname_bytes_aux:nnnn #1 }
6658   \cs_new:Npx \__str_convert_pdfname_bytes_aux:nnnn #1#2#3#4
6659   {
6660     \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#1}
6661     \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#2}
6662     \exp_not:N \tl_if_blank:nF {#3}
6663     {
6664       \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#3}
6665       \exp_not:N \tl_if_blank:nF {#4}
6666       {
6667         \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#4}
6668       }
6669     }
6670   }
6671 }
6672 { \cs_new_eq:NN \__str_convert_pdfname:n \__str_escape_name_char:n }

```

(End definition for `\str_convert_pdfname:n` and others. This function is documented on page 74.)

6673 ⟨/package⟩

10.7.1 iso 8859 support

The ISO-8859-1 encoding exactly matches with the 256 first Unicode characters. For other 8-bit encodings of the ISO-8859 family, we keep track only of differences, and of unassigned bytes.

```
6674  {*iso88591}
6675  \__str_declare_eight_bit_encoding:nnnn { iso88591 } { 256 }
6676  {
6677  }
6678  {
6679  }
6680  /iso88591
6681  {*iso88592}
6682  \__str_declare_eight_bit_encoding:nnnn { iso88592 } { 399 }
6683  {
6684  { A1 } { 0104 }
6685  { A2 } { 02D8 }
6686  { A3 } { 0141 }
6687  { A5 } { 013D }
6688  { A6 } { 015A }
6689  { A9 } { 0160 }
6690  { AA } { 015E }
6691  { AB } { 0164 }
6692  { AC } { 0179 }
6693  { AE } { 017D }
6694  { AF } { 017B }
6695  { B1 } { 0105 }
6696  { B2 } { 02DB }
6697  { B3 } { 0142 }
6698  { B5 } { 013E }
6699  { B6 } { 015B }
6700  { B7 } { 02C7 }
6701  { B9 } { 0161 }
6702  { BA } { 015F }
6703  { BB } { 0165 }
6704  { BC } { 017A }
6705  { BD } { 02DD }
6706  { BE } { 017E }
6707  { BF } { 017C }
6708  { C0 } { 0154 }
6709  { C3 } { 0102 }
6710  { C5 } { 0139 }
6711  { C6 } { 0106 }
6712  { C8 } { 010C }
6713  { CA } { 0118 }
6714  { CC } { 011A }
6715  { CF } { 010E }
6716  { D0 } { 0110 }
6717  { D1 } { 0143 }
6718  { D2 } { 0147 }
```

```

6719      { D5 } { 0150 }
6720      { D8 } { 0158 }
6721      { D9 } { 016E }
6722      { DB } { 0170 }
6723      { DE } { 0162 }
6724      { EO } { 0155 }
6725      { E3 } { 0103 }
6726      { E5 } { 013A }
6727      { E6 } { 0107 }
6728      { E8 } { 010D }
6729      { EA } { 0119 }
6730      { EC } { 011B }
6731      { EF } { 010F }
6732      { FO } { 0111 }
6733      { F1 } { 0144 }
6734      { F2 } { 0148 }
6735      { F5 } { 0151 }
6736      { F8 } { 0159 }
6737      { F9 } { 016F }
6738      { FB } { 0171 }
6739      { FE } { 0163 }
6740      { FF } { 02D9 }
6741      }
6742      {
6743      }
6744 </iso88592>
6745 <*iso88593>
6746 \__str_declare_eight_bit_encoding:nnnn { iso88593 } { 384 }
6747 {
6748     { A1 } { 0126 }
6749     { A2 } { 02D8 }
6750     { A6 } { 0124 }
6751     { A9 } { 0130 }
6752     { AA } { 015E }
6753     { AB } { 011E }
6754     { AC } { 0134 }
6755     { AF } { 017B }
6756     { B1 } { 0127 }
6757     { B6 } { 0125 }
6758     { B9 } { 0131 }
6759     { BA } { 015F }
6760     { BB } { 011F }
6761     { BC } { 0135 }
6762     { BF } { 017C }
6763     { C5 } { 010A }
6764     { C6 } { 0108 }
6765     { D5 } { 0120 }
6766     { D8 } { 011C }
6767     { DD } { 016C }
6768     { DE } { 015C }
6769     { E5 } { 010B }
6770     { E6 } { 0109 }
6771     { F5 } { 0121 }
6772     { F8 } { 011D }

```

```

6773 { FD } { 016D }
6774 { FE } { 015D }
6775 { FF } { 02D9 }
6776 }
6777 {
6778 { A5 }
6779 { AE }
6780 { BE }
6781 { C3 }
6782 { DO }
6783 { E3 }
6784 { FO }
6785 }
6786 </iso88593>
6787 {*iso88594}
6788 \__str_declare_eight_bit_encoding:nnnn { iso88594 } { 383 }
6789 {
6790 { A1 } { 0104 }
6791 { A2 } { 0138 }
6792 { A3 } { 0156 }
6793 { A5 } { 0128 }
6794 { A6 } { 013B }
6795 { A9 } { 0160 }
6796 { AA } { 0112 }
6797 { AB } { 0122 }
6798 { AC } { 0166 }
6799 { AE } { 017D }
6800 { B1 } { 0105 }
6801 { B2 } { 02DB }
6802 { B3 } { 0157 }
6803 { B5 } { 0129 }
6804 { B6 } { 013C }
6805 { B7 } { 02C7 }
6806 { B9 } { 0161 }
6807 { BA } { 0113 }
6808 { BB } { 0123 }
6809 { BC } { 0167 }
6810 { BD } { 014A }
6811 { BE } { 017E }
6812 { BF } { 014B }
6813 { CO } { 0100 }
6814 { C7 } { 012E }
6815 { C8 } { 010C }
6816 { CA } { 0118 }
6817 { CC } { 0116 }
6818 { CF } { 012A }
6819 { DO } { 0110 }
6820 { D1 } { 0145 }
6821 { D2 } { 014C }
6822 { D3 } { 0136 }
6823 { D9 } { 0172 }
6824 { DD } { 0168 }
6825 { DE } { 016A }
6826 { EO } { 0101 }

```

```

6827      { E7 } { 012F }
6828      { E8 } { 010D }
6829      { EA } { 0119 }
6830      { EC } { 0117 }
6831      { EF } { 012B }
6832      { F0 } { 0111 }
6833      { F1 } { 0146 }
6834      { F2 } { 014D }
6835      { F3 } { 0137 }
6836      { F9 } { 0173 }
6837      { FD } { 0169 }
6838      { FE } { 016B }
6839      { FF } { 02D9 }
6840  }
6841  {
6842  }
6843 </iso88594>
6844 <*iso88595>
6845 \__str_declare_eight_bit_encoding:nnnn { iso88595 } { 374 }
6846  {
6847      { A1 } { 0401 }
6848      { A2 } { 0402 }
6849      { A3 } { 0403 }
6850      { A4 } { 0404 }
6851      { A5 } { 0405 }
6852      { A6 } { 0406 }
6853      { A7 } { 0407 }
6854      { A8 } { 0408 }
6855      { A9 } { 0409 }
6856      { AA } { 040A }
6857      { AB } { 040B }
6858      { AC } { 040C }
6859      { AE } { 040E }
6860      { AF } { 040F }
6861      { B0 } { 0410 }
6862      { B1 } { 0411 }
6863      { B2 } { 0412 }
6864      { B3 } { 0413 }
6865      { B4 } { 0414 }
6866      { B5 } { 0415 }
6867      { B6 } { 0416 }
6868      { B7 } { 0417 }
6869      { B8 } { 0418 }
6870      { B9 } { 0419 }
6871      { BA } { 041A }
6872      { BB } { 041B }
6873      { BC } { 041C }
6874      { BD } { 041D }
6875      { BE } { 041E }
6876      { BF } { 041F }
6877      { C0 } { 0420 }
6878      { C1 } { 0421 }
6879      { C2 } { 0422 }
6880      { C3 } { 0423 }

```

6881 { C4 } { 0424 }
6882 { C5 } { 0425 }
6883 { C6 } { 0426 }
6884 { C7 } { 0427 }
6885 { C8 } { 0428 }
6886 { C9 } { 0429 }
6887 { CA } { 042A }
6888 { CB } { 042B }
6889 { CC } { 042C }
6890 { CD } { 042D }
6891 { CE } { 042E }
6892 { CF } { 042F }
6893 { DO } { 0430 }
6894 { D1 } { 0431 }
6895 { D2 } { 0432 }
6896 { D3 } { 0433 }
6897 { D4 } { 0434 }
6898 { D5 } { 0435 }
6899 { D6 } { 0436 }
6900 { D7 } { 0437 }
6901 { D8 } { 0438 }
6902 { D9 } { 0439 }
6903 { DA } { 043A }
6904 { DB } { 043B }
6905 { DC } { 043C }
6906 { DD } { 043D }
6907 { DE } { 043E }
6908 { DF } { 043F }
6909 { EO } { 0440 }
6910 { E1 } { 0441 }
6911 { E2 } { 0442 }
6912 { E3 } { 0443 }
6913 { E4 } { 0444 }
6914 { E5 } { 0445 }
6915 { E6 } { 0446 }
6916 { E7 } { 0447 }
6917 { E8 } { 0448 }
6918 { E9 } { 0449 }
6919 { EA } { 044A }
6920 { EB } { 044B }
6921 { EC } { 044C }
6922 { ED } { 044D }
6923 { EE } { 044E }
6924 { EF } { 044F }
6925 { FO } { 2116 }
6926 { F1 } { 0451 }
6927 { F2 } { 0452 }
6928 { F3 } { 0453 }
6929 { F4 } { 0454 }
6930 { F5 } { 0455 }
6931 { F6 } { 0456 }
6932 { F7 } { 0457 }
6933 { F8 } { 0458 }
6934 { F9 } { 0459 }

```

6935      { FA } { 045A }
6936      { FB } { 045B }
6937      { FC } { 045C }
6938      { FD } { 00A7 }
6939      { FE } { 045E }
6940      { FF } { 045F }
6941      }
6942      {
6943      }
6944  </iso88595>
6945  <*iso88596>
6946  \__str_declare_eight_bit_encoding:nnnn { iso88596 } { 344 }
6947  {
6948      { AC } { 060C }
6949      { BB } { 061B }
6950      { BF } { 061F }
6951      { C1 } { 0621 }
6952      { C2 } { 0622 }
6953      { C3 } { 0623 }
6954      { C4 } { 0624 }
6955      { C5 } { 0625 }
6956      { C6 } { 0626 }
6957      { C7 } { 0627 }
6958      { C8 } { 0628 }
6959      { C9 } { 0629 }
6960      { CA } { 062A }
6961      { CB } { 062B }
6962      { CC } { 062C }
6963      { CD } { 062D }
6964      { CE } { 062E }
6965      { CF } { 062F }
6966      { DO } { 0630 }
6967      { D1 } { 0631 }
6968      { D2 } { 0632 }
6969      { D3 } { 0633 }
6970      { D4 } { 0634 }
6971      { D5 } { 0635 }
6972      { D6 } { 0636 }
6973      { D7 } { 0637 }
6974      { D8 } { 0638 }
6975      { D9 } { 0639 }
6976      { DA } { 063A }
6977      { EO } { 0640 }
6978      { E1 } { 0641 }
6979      { E2 } { 0642 }
6980      { E3 } { 0643 }
6981      { E4 } { 0644 }
6982      { E5 } { 0645 }
6983      { E6 } { 0646 }
6984      { E7 } { 0647 }
6985      { E8 } { 0648 }
6986      { E9 } { 0649 }
6987      { EA } { 064A }
6988      { EB } { 064B }

```

```

6989      { EC } { 064C }
6990      { ED } { 064D }
6991      { EE } { 064E }
6992      { EF } { 064F }
6993      { FO } { 0650 }
6994      { F1 } { 0651 }
6995      { F2 } { 0652 }
6996  }
6997  {
6998      { A1 }
6999      { A2 }
7000      { A3 }
7001      { A5 }
7002      { A6 }
7003      { A7 }
7004      { A8 }
7005      { A9 }
7006      { AA }
7007      { AB }
7008      { AE }
7009      { AF }
7010      { BO }
7011      { B1 }
7012      { B2 }
7013      { B3 }
7014      { B4 }
7015      { B5 }
7016      { B6 }
7017      { B7 }
7018      { B8 }
7019      { B9 }
7020      { BA }
7021      { BC }
7022      { BD }
7023      { BE }
7024      { CO }
7025      { DB }
7026      { DC }
7027      { DD }
7028      { DE }
7029      { DF }
7030  }
7031  </iso88596>
7032  /*iso88597
7033  \__str_declare_eight_bit_encoding:nmmm { iso88597 } { 498 }
7034  {
7035      { A1 } { 2018 }
7036      { A2 } { 2019 }
7037      { A4 } { 20AC }
7038      { A5 } { 20AF }
7039      { AA } { 037A }
7040      { AF } { 2015 }
7041      { B4 } { 0384 }
7042      { B5 } { 0385 }

```

7043 { B6 } { 0386 }
7044 { B8 } { 0388 }
7045 { B9 } { 0389 }
7046 { BA } { 038A }
7047 { BC } { 038C }
7048 { BE } { 038E }
7049 { BF } { 038F }
7050 { CO } { 0390 }
7051 { C1 } { 0391 }
7052 { C2 } { 0392 }
7053 { C3 } { 0393 }
7054 { C4 } { 0394 }
7055 { C5 } { 0395 }
7056 { C6 } { 0396 }
7057 { C7 } { 0397 }
7058 { C8 } { 0398 }
7059 { C9 } { 0399 }
7060 { CA } { 039A }
7061 { CB } { 039B }
7062 { CC } { 039C }
7063 { CD } { 039D }
7064 { CE } { 039E }
7065 { CF } { 039F }
7066 { DO } { 03A0 }
7067 { D1 } { 03A1 }
7068 { D3 } { 03A3 }
7069 { D4 } { 03A4 }
7070 { D5 } { 03A5 }
7071 { D6 } { 03A6 }
7072 { D7 } { 03A7 }
7073 { D8 } { 03A8 }
7074 { D9 } { 03A9 }
7075 { DA } { 03AA }
7076 { DB } { 03AB }
7077 { DC } { 03AC }
7078 { DD } { 03AD }
7079 { DE } { 03AE }
7080 { DF } { 03AF }
7081 { EO } { 03B0 }
7082 { E1 } { 03B1 }
7083 { E2 } { 03B2 }
7084 { E3 } { 03B3 }
7085 { E4 } { 03B4 }
7086 { E5 } { 03B5 }
7087 { E6 } { 03B6 }
7088 { E7 } { 03B7 }
7089 { E8 } { 03B8 }
7090 { E9 } { 03B9 }
7091 { EA } { 03BA }
7092 { EB } { 03BB }
7093 { EC } { 03BC }
7094 { ED } { 03BD }
7095 { EE } { 03BE }
7096 { EF } { 03BF }

```

7097      { F0 } { 03C0 }
7098      { F1 } { 03C1 }
7099      { F2 } { 03C2 }
7100      { F3 } { 03C3 }
7101      { F4 } { 03C4 }
7102      { F5 } { 03C5 }
7103      { F6 } { 03C6 }
7104      { F7 } { 03C7 }
7105      { F8 } { 03C8 }
7106      { F9 } { 03C9 }
7107      { FA } { 03CA }
7108      { FB } { 03CB }
7109      { FC } { 03CC }
7110      { FD } { 03CD }
7111      { FE } { 03CE }
7112  }
7113  {
7114      { AE }
7115      { D2 }
7116  }
7117 </iso88597>
7118 <*iso88598>
7119 \__str_declare_eight_bit_encoding:nnnn { iso88598 } { 308 }
7120  {
7121      { AA } { 00D7 }
7122      { BA } { 00F7 }
7123      { DF } { 2017 }
7124      { EO } { 05D0 }
7125      { E1 } { 05D1 }
7126      { E2 } { 05D2 }
7127      { E3 } { 05D3 }
7128      { E4 } { 05D4 }
7129      { E5 } { 05D5 }
7130      { E6 } { 05D6 }
7131      { E7 } { 05D7 }
7132      { E8 } { 05D8 }
7133      { E9 } { 05D9 }
7134      { EA } { 05DA }
7135      { EB } { 05DB }
7136      { EC } { 05DC }
7137      { ED } { 05DD }
7138      { EE } { 05DE }
7139      { EF } { 05DF }
7140      { FO } { 05E0 }
7141      { F1 } { 05E1 }
7142      { F2 } { 05E2 }
7143      { F3 } { 05E3 }
7144      { F4 } { 05E4 }
7145      { F5 } { 05E5 }
7146      { F6 } { 05E6 }
7147      { F7 } { 05E7 }
7148      { F8 } { 05E8 }
7149      { F9 } { 05E9 }
7150      { FA } { 05EA }

```

```

7151      { FD } { 200E }
7152      { FE } { 200F }
7153  }
7154  {
7155      { A1 }
7156      { BF }
7157      { CO }
7158      { C1 }
7159      { C2 }
7160      { C3 }
7161      { C4 }
7162      { C5 }
7163      { C6 }
7164      { C7 }
7165      { C8 }
7166      { C9 }
7167      { CA }
7168      { CB }
7169      { CC }
7170      { CD }
7171      { CE }
7172      { CF }
7173      { DO }
7174      { D1 }
7175      { D2 }
7176      { D3 }
7177      { D4 }
7178      { D5 }
7179      { D6 }
7180      { D7 }
7181      { D8 }
7182      { D9 }
7183      { DA }
7184      { DB }
7185      { DC }
7186      { DD }
7187      { DE }
7188      { FB }
7189      { FC }
7190  }
7191 </iso88598>
7192 <*iso88599>
7193 \__str_declare_eight_bit_encoding:nnnn { iso88599 } { 352 }
7194  {
7195      { DO } { 011E }
7196      { DD } { 0130 }
7197      { DE } { 015E }
7198      { FO } { 011F }
7199      { FD } { 0131 }
7200      { FE } { 015F }
7201  }
7202  {
7203  }
7204 </iso88599>

```

```

7205  <*iso885910>
7206  \__str_declare_eight_bit_encoding:nnnn { iso885910 } { 383 }
7207  {
7208      { A1 } { 0104 }
7209      { A2 } { 0112 }
7210      { A3 } { 0122 }
7211      { A4 } { 012A }
7212      { A5 } { 0128 }
7213      { A6 } { 0136 }
7214      { A8 } { 013B }
7215      { A9 } { 0110 }
7216      { AA } { 0160 }
7217      { AB } { 0166 }
7218      { AC } { 017D }
7219      { AE } { 016A }
7220      { AF } { 014A }
7221      { B1 } { 0105 }
7222      { B2 } { 0113 }
7223      { B3 } { 0123 }
7224      { B4 } { 012B }
7225      { B5 } { 0129 }
7226      { B6 } { 0137 }
7227      { B8 } { 013C }
7228      { B9 } { 0111 }
7229      { BA } { 0161 }
7230      { BB } { 0167 }
7231      { BC } { 017E }
7232      { BD } { 2015 }
7233      { BE } { 016B }
7234      { BF } { 014B }
7235      { CO } { 0100 }
7236      { C7 } { 012E }
7237      { C8 } { 010C }
7238      { CA } { 0118 }
7239      { CC } { 0116 }
7240      { D1 } { 0145 }
7241      { D2 } { 014C }
7242      { D7 } { 0168 }
7243      { D9 } { 0172 }
7244      { E0 } { 0101 }
7245      { E7 } { 012F }
7246      { E8 } { 010D }
7247      { EA } { 0119 }
7248      { EC } { 0117 }
7249      { F1 } { 0146 }
7250      { F2 } { 014D }
7251      { F7 } { 0169 }
7252      { F9 } { 0173 }
7253      { FF } { 0138 }
7254  }
7255  {
7256  }
7257  </iso885910>
7258  <*iso885911>

```

```
7259 \__str_declare_eight_bit_encoding:nnnn { iso885911 } { 369 }
7260 {
7261     { A1 } { OE01 }
7262     { A2 } { OE02 }
7263     { A3 } { OE03 }
7264     { A4 } { OE04 }
7265     { A5 } { OE05 }
7266     { A6 } { OE06 }
7267     { A7 } { OE07 }
7268     { A8 } { OE08 }
7269     { A9 } { OE09 }
7270     { AA } { OEOA }
7271     { AB } { OEOB }
7272     { AC } { OEOC }
7273     { AD } { OEOD }
7274     { AE } { OEOE }
7275     { AF } { OEOF }
7276     { BO } { OE10 }
7277     { B1 } { OE11 }
7278     { B2 } { OE12 }
7279     { B3 } { OE13 }
7280     { B4 } { OE14 }
7281     { B5 } { OE15 }
7282     { B6 } { OE16 }
7283     { B7 } { OE17 }
7284     { B8 } { OE18 }
7285     { B9 } { OE19 }
7286     { BA } { OE1A }
7287     { BB } { OE1B }
7288     { BC } { OE1C }
7289     { BD } { OE1D }
7290     { BE } { OE1E }
7291     { BF } { OE1F }
7292     { CO } { OE20 }
7293     { C1 } { OE21 }
7294     { C2 } { OE22 }
7295     { C3 } { OE23 }
7296     { C4 } { OE24 }
7297     { C5 } { OE25 }
7298     { C6 } { OE26 }
7299     { C7 } { OE27 }
7300     { C8 } { OE28 }
7301     { C9 } { OE29 }
7302     { CA } { OE2A }
7303     { CB } { OE2B }
7304     { CC } { OE2C }
7305     { CD } { OE2D }
7306     { CE } { OE2E }
7307     { CF } { OE2F }
7308     { DO } { OE30 }
7309     { D1 } { OE31 }
7310     { D2 } { OE32 }
7311     { D3 } { OE33 }
7312     { D4 } { OE34 }
```

```

7313 { D5 } { 0E35 }
7314 { D6 } { 0E36 }
7315 { D7 } { 0E37 }
7316 { D8 } { 0E38 }
7317 { D9 } { 0E39 }
7318 { DA } { 0E3A }
7319 { DF } { 0E3F }
7320 { EO } { 0E40 }
7321 { E1 } { 0E41 }
7322 { E2 } { 0E42 }
7323 { E3 } { 0E43 }
7324 { E4 } { 0E44 }
7325 { E5 } { 0E45 }
7326 { E6 } { 0E46 }
7327 { E7 } { 0E47 }
7328 { E8 } { 0E48 }
7329 { E9 } { 0E49 }
7330 { EA } { 0E4A }
7331 { EB } { 0E4B }
7332 { EC } { 0E4C }
7333 { ED } { 0E4D }
7334 { EE } { 0E4E }
7335 { EF } { 0E4F }
7336 { F0 } { 0E50 }
7337 { F1 } { 0E51 }
7338 { F2 } { 0E52 }
7339 { F3 } { 0E53 }
7340 { F4 } { 0E54 }
7341 { F5 } { 0E55 }
7342 { F6 } { 0E56 }
7343 { F7 } { 0E57 }
7344 { F8 } { 0E58 }
7345 { F9 } { 0E59 }
7346 { FA } { 0E5A }
7347 { FB } { 0E5B }
7348 }
7349 {
7350 { DB }
7351 { DC }
7352 { DD }
7353 { DE }
7354 }
7355 </iso885911>
7356 /*iso885913>
7357 \__str_declare_eight_bit_encoding:nnnn { iso885913 } { 399 }
7358 {
7359 { A1 } { 201D }
7360 { A5 } { 201E }
7361 { A8 } { 00D8 }
7362 { AA } { 0156 }
7363 { AF } { 00C6 }
7364 { B4 } { 201C }
7365 { B8 } { 00F8 }
7366 { BA } { 0157 }

```

```

7367 { BF } { 00E6 }
7368 { C0 } { 0104 }
7369 { C1 } { 012E }
7370 { C2 } { 0100 }
7371 { C3 } { 0106 }
7372 { C6 } { 0118 }
7373 { C7 } { 0112 }
7374 { C8 } { 010C }
7375 { CA } { 0179 }
7376 { CB } { 0116 }
7377 { CC } { 0122 }
7378 { CD } { 0136 }
7379 { CE } { 012A }
7380 { CF } { 013B }
7381 { DO } { 0160 }
7382 { D1 } { 0143 }
7383 { D2 } { 0145 }
7384 { D4 } { 014C }
7385 { D8 } { 0172 }
7386 { D9 } { 0141 }
7387 { DA } { 015A }
7388 { DB } { 016A }
7389 { DD } { 017B }
7390 { DE } { 017D }
7391 { EO } { 0105 }
7392 { E1 } { 012F }
7393 { E2 } { 0101 }
7394 { E3 } { 0107 }
7395 { E6 } { 0119 }
7396 { E7 } { 0113 }
7397 { E8 } { 010D }
7398 { EA } { 017A }
7399 { EB } { 0117 }
7400 { EC } { 0123 }
7401 { ED } { 0137 }
7402 { EE } { 012B }
7403 { EF } { 013C }
7404 { FO } { 0161 }
7405 { F1 } { 0144 }
7406 { F2 } { 0146 }
7407 { F4 } { 014D }
7408 { F8 } { 0173 }
7409 { F9 } { 0142 }
7410 { FA } { 015B }
7411 { FB } { 016B }
7412 { FD } { 017C }
7413 { FE } { 017E }
7414 { FF } { 2019 }

7415 }
7416 {
7417 }
7418 </iso885913>
7419 <*iso885914>
7420 \__str_declare_eight_bit_encoding:nnnn { iso885914 } { 529 }

```

```

7421    {
7422        { A1 } { 1E02 }
7423        { A2 } { 1E03 }
7424        { A4 } { 010A }
7425        { A5 } { 010B }
7426        { A6 } { 1EOA }
7427        { A8 } { 1E80 }
7428        { AA } { 1E82 }
7429        { AB } { 1EOB }
7430        { AC } { 1EF2 }
7431        { AF } { 0178 }
7432        { BO } { 1E1E }
7433        { B1 } { 1E1F }
7434        { B2 } { 0120 }
7435        { B3 } { 0121 }
7436        { B4 } { 1E40 }
7437        { B5 } { 1E41 }
7438        { B7 } { 1E56 }
7439        { B8 } { 1E81 }
7440        { B9 } { 1E57 }
7441        { BA } { 1E83 }
7442        { BB } { 1E60 }
7443        { BC } { 1EF3 }
7444        { BD } { 1E84 }
7445        { BE } { 1E85 }
7446        { BF } { 1E61 }
7447        { DO } { 0174 }
7448        { D7 } { 1E6A }
7449        { DE } { 0176 }
7450        { FO } { 0175 }
7451        { F7 } { 1E6B }
7452        { FE } { 0177 }
7453    }
7454    {
7455    }
7456 </iso885914>
7457 <*iso885915>
7458 \__str_declare_eight_bit_encoding:nnnn { iso885915 } { 383 }
7459 {
7460     { A4 } { 20AC }
7461     { A6 } { 0160 }
7462     { A8 } { 0161 }
7463     { B4 } { 017D }
7464     { B8 } { 017E }
7465     { BC } { 0152 }
7466     { BD } { 0153 }
7467     { BE } { 0178 }
7468 }
7469 {
7470 }
7471 </iso885915>
7472 <*iso885916>
7473 \__str_declare_eight_bit_encoding:nnnn { iso885916 } { 558 }

```

```

7474  {
7475  { A1 } { 0104 }
7476  { A2 } { 0105 }
7477  { A3 } { 0141 }
7478  { A4 } { 20AC }
7479  { A5 } { 201E }
7480  { A6 } { 0160 }
7481  { A8 } { 0161 }
7482  { AA } { 0218 }
7483  { AC } { 0179 }
7484  { AE } { 017A }
7485  { AF } { 017B }
7486  { B2 } { 010C }
7487  { B3 } { 0142 }
7488  { B4 } { 017D }
7489  { B5 } { 201D }
7490  { B8 } { 017E }
7491  { B9 } { 010D }
7492  { BA } { 0219 }
7493  { BC } { 0152 }
7494  { BD } { 0153 }
7495  { BE } { 0178 }
7496  { BF } { 017C }
7497  { C3 } { 0102 }
7498  { C5 } { 0106 }
7499  { D0 } { 0110 }
7500  { D1 } { 0143 }
7501  { D5 } { 0150 }
7502  { D7 } { 015A }
7503  { D8 } { 0170 }
7504  { DD } { 0118 }
7505  { DE } { 021A }
7506  { E3 } { 0103 }
7507  { E5 } { 0107 }
7508  { FO } { 0111 }
7509  { F1 } { 0144 }
7510  { F5 } { 0151 }
7511  { F7 } { 015B }
7512  { F8 } { 0171 }
7513  { FD } { 0119 }
7514  { FE } { 021B }
7515  }
7516  {
7517  }
7518 </iso885916>

```

11 I3seq implementation

The following test files are used for this code: m3seq002, m3seq003.

```

7519 <*package>
7520 <@@=seq>

```

A sequence is a control sequence whose top-level expansion is of the form “`\s__-seq __seq_item:n {⟨item1⟩} … __seq_item:n {⟨itemn⟩}`”, with a leading scan mark followed by n items of the same form. An earlier implementation used the structure “`\seq_elt:w ⟨item1⟩ \seq_elt_end: … \seq_elt:w ⟨itemn⟩ \seq_elt_end:`”. This allowed rapid searching using a delimited function, but was not suitable for items containing {, } and # tokens, and also lead to the loss of surrounding braces around items

`__seq_item:n *` `__seq_item:n {⟨item⟩}`

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

`__seq_push_item_def:n` `__seq_push_item_def:n {⟨code⟩}`

Saves the definition of `__seq_item:n` and redefines it to accept one parameter and expand to `⟨code⟩`. This function should always be balanced by use of `__seq_pop_item_def:`.

`__seq_pop_item_def:` `__seq_pop_item_def:`

Restores the definition of `__seq_item:n` most recently saved by `__seq_push_item_def:n`. This function should always be used in a balanced pair with `__seq_push_item_def:n`.

`\s__seq` This private scan mark.

7521 `\scan_new:N \s__seq`

(End definition for `\s__seq`.)

`\s__seq_mark` Private scan marks.

`\s__seq_stop` 7522 `\scan_new:N \s__seq_mark`
 7523 `\scan_new:N \s__seq_stop`

(End definition for `\s__seq_mark` and `\s__seq_stop`.)

`__seq_item:n` The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

7524 `\cs_new:Npn __seq_item:n`
7525 {
7526 `__kernel_msg_expandable_error:nn { kernel } { misused-sequence }`
7527 `\use_none:n`
7528 }

(End definition for `__seq_item:n`.)

`\l__seq_internal_a_t1` Scratch space for various internal uses.

`\l__seq_internal_b_t1` 7529 `\tl_new:N \l__seq_internal_a_t1`
 7530 `\tl_new:N \l__seq_internal_b_t1`

(End definition for `\l__seq_internal_a_t1` and `\l__seq_internal_b_t1`.)

`__seq_tmp:w` Scratch function for internal use.

7531 `\cs_new_eq:NN __seq_tmp:w ?`

(End definition for `__seq_tmp:w`.)

\c_empty_seq A sequence with no item, following the structure mentioned above.

```
7532 \tl_const:Nn \c_empty_seq { \s__seq }
```

(End definition for **\c_empty_seq**. This variable is documented on page 86.)

11.1 Allocation and initialisation

\seq_new:N Sequences are initialized to **\c_empty_seq**.

```
7533 \cs_new_protected:Npn \seq_new:N #1
7534 {
7535     __kernel_chk_if_free_cs:N #1
7536     \cs_gset_eq:NN #1 \c_empty_seq
7537 }
7538 \cs_generate_variant:Nn \seq_new:N { c }
```

(End definition for **\seq_new:N**. This function is documented on page 75.)

\seq_clear:N Clearing a sequence is similar to setting it equal to the empty one.

```
7539 \cs_new_protected:Npn \seq_clear:N #1
7540 {
7541     \seq_set_eq:NN #1 \c_empty_seq
7542     \cs_generate_variant:Nn \seq_clear:N { c }
7543     \cs_new_protected:Npn \seq_gclear:N #1
7544     {
7545         \seq_gset_eq:NN #1 \c_empty_seq
7546     }
7547     \cs_generate_variant:Nn \seq_gclear:N { c }
```

(End definition for **\seq_clear:N** and **\seq_gclear:N**. These functions are documented on page 75.)

\seq_clear_new:N Once again we copy code from the token list functions.

```
7545 \cs_new_protected:Npn \seq_clear_new:N #1
7546 {
7547     \seq_if_exist:NTF #1 {
7548         \seq_clear:N #1
7549         \seq_new:N #1
7550     }
7551     \cs_generate_variant:Nn \seq_clear_new:N { c }
7552     \cs_new_protected:Npn \seq_gclear_new:N #1
7553 {
7554     \seq_if_exist:NTF #1 {
7555         \seq_gclear:N #1
7556         \seq_new:N #1
7557     }
7558     \cs_generate_variant:Nn \seq_gclear_new:N { c }
```

(End definition for **\seq_clear_new:N** and **\seq_gclear_new:N**. These functions are documented on page 75.)

\seq_set_eq:NN Copying a sequence is the same as copying the underlying token list.

```
7551 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
7552 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
7553 \cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN
7554 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
7555 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
7556 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
7557 \cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
7558 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc
```

(End definition for **\seq_set_eq:NN** and **\seq_gset_eq:NN**. These functions are documented on page 75.)

```

\seq_set_from_clist:NN Setting a sequence from a comma-separated list is done using a simple mapping.
\seq_set_from_clist:cN
\seq_set_from_clist:Nc
\seq_set_from_clist:cc
\seq_set_from_clist:Nn
\seq_set_from_clist:cn
\seq_gset_from_clist:NN
\seq_gset_from_clist:cN
\seq_gset_from_clist:Nc
\seq_gset_from_clist:cc
\seq_gset_from_clist:Nn
\seq_gset_from_clist:cn

7559 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
7560 {
7561     \__kernel_tl_set:Nx #1
7562     { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
7563 }
7564 \cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
7565 {
7566     \__kernel_tl_set:Nx #1
7567     { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
7568 }
7569 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
7570 {
7571     \__kernel_tl_gset:Nx #1
7572     { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
7573 }
7574 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
7575 {
7576     \__kernel_tl_gset:Nx #1
7577     { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
7578 }
7579 \cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }
7580 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
7581 \cs_generate_variant:Nn \seq_set_from_clist:Nn { c }
7582 \cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }
7583 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
7584 \cs_generate_variant:Nn \seq_gset_from_clist:Nn { c }

```

(End definition for `\seq_set_from_clist:NN` and others. These functions are documented on page 75.)

`\seq_const_from_clist:Nn` Almost identical to `\seq_set_from_clist:Nn`.

```

\seq_const_from_clist:cN
7585 \cs_new_protected:Npn \seq_const_from_clist:Nn #1#2
7586 {
7587     \tl_const:Nx #1
7588     { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
7589 }
7590 \cs_generate_variant:Nn \seq_const_from_clist:Nn { c }

```

(End definition for `\seq_const_from_clist:Nn`. This function is documented on page 76.)

`\seq_set_split:Nnn` When the separator is empty, everything is very simple, just map `__seq_wrap_item:n` through the items of the last argument. For non-trivial separators, the goal is to split a given token list at the marker, strip spaces from each item, and remove one set of outer braces if after removing leading and trailing spaces the item is enclosed within braces. After `\tl_replace_all:Nnn`, the token list `\l__seq_internal_a_tl` is a repetition of the pattern `__seq_set_split_auxi:w \prg_do_nothing: <item with spaces>` `__seq_set_split_end:.` Then, x-expansion causes `__seq_set_split_auxi:w` to trim spaces, and leaves its result as `__seq_set_split_auxi:w <trimmed item> __seq_set_split_end:.` This is then converted to the l3seq internal structure by another x-expansion. In the first step, we insert `\prg_do_nothing:` to avoid losing braces too early: that would cause space trimming to act within those lost braces. The second step is solely there to strip braces which are outermost after space trimming.

```

7591 \cs_new_protected:Npn \seq_set_split:Nnn

```

```

7592 { \__seq_set_split:Nnn \__kernel_tl_set:Nx }
7593 \cs_new_protected:Npn \seq_gset_split:Nnn
7594 { \__seq_set_split:Nnn \__kernel_tl_gset:Nx }
7595 \cs_new_protected:Npn \__seq_set_split:Nnn #1#2#3#4
7596 {
7597     \tl_if_empty:nTF {#3}
7598     {
7599         \tl_set:Nn \l__seq_internal_a_tl
7600         { \tl_map_function:nN {#4} \__seq_wrap_item:n }
7601     }
7602     {
7603         \tl_set:Nn \l__seq_internal_a_tl
7604         {
7605             \__seq_set_split_auxi:w \prg_do_nothing:
7606             #4
7607             \__seq_set_split_end:
7608         }
7609         \tl_replace_all:Nnn \l__seq_internal_a_tl { #3 }
7610         {
7611             \__seq_set_split_end:
7612             \__seq_set_split_auxi:w \prg_do_nothing:
7613         }
7614         \__kernel_tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
7615     }
7616     #1 #2 { \s__seq \l__seq_internal_a_tl }
7617 }
7618 \cs_new:Npn \__seq_set_split_auxi:w #1 \__seq_set_split_end:
7619 {
7620     \exp_not:N \__seq_set_split_auxii:w
7621     \exp_args:No \tl_trim_spaces:n {#1}
7622     \exp_not:N \__seq_set_split_end:
7623 }
7624 \cs_new:Npn \__seq_set_split_auxii:w #1 \__seq_set_split_end:
7625 { \__seq_wrap_item:n {#1} }
7626 \cs_generate_variant:Nn \seq_set_split:Nnn { NnV }
7627 \cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }

```

(End definition for `\seq_set_split:Nnn` and others. These functions are documented on page 76.)

`\seq_concat:NNN`

`\seq_concat:ccc`

`\seq_gconcat:NNN`

`\seq_gconcat:ccc`

When concatenating sequences, one must remove the leading `\s__seq` of the second sequence. The result starts with `\s__seq` (of the first sequence), which stops f-expansion.

```

7628 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
7629 { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
7630 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
7631 { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
7632 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
7633 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for `\seq_concat:NNN` and `\seq_gconcat:NNN`. These functions are documented on page 76.)

`\seq_if_exist_p:N` Copies of the `cs` functions defined in `!3basics`.

`\seq_if_exist_p:c`

`\seq_if_exist:NTF`

`\seq_if_exist:cTF`

```

7634 \prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N
7635 { TF , T , F , p }
7636 \prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c

```

```
7637 { TF , T , F , p }
```

(End definition for `\seq_if_exist:NTF`. This function is documented on page 76.)

11.2 Appending data to either end

```
\seq_put_left:Nn
\seq_put_left:NV
\seq_put_left:Nv
\seq_put_left:Nv
\seq_put_left:No
\seq_put_left:Nx
\seq_put_left:cn
\seq_put_left:cV
\seq_put_left:cv
\seq_put_left:co
\seq_put_left:cX
\seq_gput_left:Nn
\seq_gput_left:NV
\seq_gput_left:Nv
\seq_gput_left:No
\seq_gput_left:Nx
\seq_gput_left:cn
\seq_gput_left:cV
\seq_gput_left:cV
\seq_gput_left:co
\seq_gput_left:cX
\_\_seq_put_left_aux:w
```

When adding to the left of a sequence, remove `\s__seq`. This is done by `__seq_put_left_aux:w`, which also stops f-expansion.

```
7638 \cs_new_protected:Npn \seq_put_left:Nn #1#2
7639 {
7640     \_\_kernel_tl_set:Nx #1
7641     {
7642         \exp_not:n { \s__seq \_\_seq_item:n {#2} }
7643         \exp_not:f { \exp_after:wN \_\_seq_put_left_aux:w #1 }
7644     }
7645 }
7646 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
7647 {
7648     \_\_kernel_tl_gset:Nx #1
7649     {
7650         \exp_not:n { \s__seq \_\_seq_item:n {#2} }
7651         \exp_not:f { \exp_after:wN \_\_seq_put_left_aux:w #1 }
7652     }
7653 }
7654 \cs_new:Npn \_\_seq_put_left_aux:w \s__seq { \exp_stop_f: }
7655 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
7656 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
7657 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
7658 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }
```

(End definition for `\seq_put_left:Nn`, `\seq_gput_left:Nn`, and `__seq_put_left_aux:w`. These functions are documented on page 76.)

```
\seq_put_right:Nn
\seq_put_right:NV
\seq_put_right:Nv
\seq_put_right:Nv
\seq_put_right:No
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:cv
\seq_put_right:co
\seq_put_right:cX
\seq_gput_right:Nn
\seq_gput_right:NV
\seq_gput_right:Nv
```

Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in `__seq_item:n`.

```
7659 \cs_new_protected:Npn \seq_put_right:Nn #1#2
7660     { \tl_put_right:Nn #1 { \_\_seq_item:n {#2} } }
7661 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
7662     { \tl_gput_right:Nn #1 { \_\_seq_item:n {#2} } }
7663 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
7664 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
7665 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
7666 \cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , co , cx }
```

(End definition for `\seq_put_right:Nn` and `\seq_gput_right:Nn`. These functions are documented on page 76.)

11.3 Modifying sequences

This function converts its argument to a proper sequence item in an x-expansion context.

```
7667 \cs_new:Npn \_\_seq_wrap_item:n #1 { \exp_not:n { \_\_seq_item:n {#1} } }
```

(End definition for `__seq_wrap_item:n`.)

\l__seq_remove_seq An internal sequence for the removal routines.

```
7668 \seq_new:N \l__seq_remove_seq
```

(End definition for \l__seq_remove_seq.)

\seq_remove_duplicates:N Removing duplicates means making a new list then copying it.

```
7669 \cs_new_protected:Npn \seq_remove_duplicates:N
7670   { \l__seq_remove_duplicates>NN \seq_set_eq:NN }
7671 \cs_new_protected:Npn \seq_gremove_duplicates:N
7672   { \l__seq_remove_duplicates>NN \seq_gset_eq:NN }
7673 \cs_new_protected:Npn \l__seq_remove_duplicates>NN #1#2
7674   {
7675     \seq_clear:N \l__seq_remove_seq
7676     \seq_map_inline:Nn #2
7677     {
7678       \seq_if_in:NnF \l__seq_remove_seq {##1}
7679         { \seq_put_right:Nn \l__seq_remove_seq {##1} }
7680     }
7681   #1 #2 \l__seq_remove_seq
7682 }
7683 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
7684 \cs_generate_variant:Nn \seq_gremove_duplicates:N { c }
```

(End definition for \seq_remove_duplicates:N, \seq_gremove_duplicates:N, and \l__seq_remove_duplicates>NN. These functions are documented on page 79.)

\seq_remove_all:Nn The idea of the code here is to avoid a relatively expensive addition of items one at a time

to an intermediate sequence. The approach taken is therefore similar to that in \l__seq_pop_right:NNN, using a “flexible” x-type expansion to do most of the work. As \tl_if_eq:nnT is not expandable, a two-part strategy is needed. First, the x-type expansion uses \str_if_eq:nnT to find potential matches. If one is found, the expansion is halted and the necessary set up takes place to use the \tl_if_eq:NNT test. The x-type is started again, including all of the items copied already. This happens repeatedly until the entire sequence has been scanned. The code is set up to avoid needing an intermediate scratch list: the lead-off x-type expansion (#1 #2 {#2}) ensures that nothing is lost.

```
7685 \cs_new_protected:Npn \seq_remove_all:Nn
7686   { \l__seq_remove_all_aux>NNn \__kernel_tl_set:Nx }
7687 \cs_new_protected:Npn \seq_gremove_all:Nn
7688   { \l__seq_remove_all_aux>NNn \__kernel_tl_gset:Nx }
7689 \cs_new_protected:Npn \l__seq_remove_all_aux>NNn #1#2#3
7690   {
7691     \__seq_push_item_def:n
7692     {
7693       \str_if_eq:nnT {##1} {#3}
7694       {
7695         \if_false: { \fi: }
7696         \tl_set:Nn \l__seq_internal_b_tl {##1}
7697       #1 #2
7698         { \if_false: } \fi:
7699         \exp_not:o {#2}
7700         \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
7701           { \use_none:nn }
7702     }
7703   \__seq_wrap_item:n {##1}
```

```

7704      }
7705      \tl_set:Nn \l__seq_internal_a_tl {#3}
7706      #1 #2 {#2}
7707      \__seq_pop_item_def:
7708  }
7709 \cs_generate_variant:Nn \seq_remove_all:Nn { c }
7710 \cs_generate_variant:Nn \seq_gremove_all:Nn { c }

(End definition for \seq_remove_all:Nn, \seq_gremove_all:Nn, and \__seq_remove_all_aux:Nnn.
These functions are documented on page 79.)

```

\seq_reverse:N Previously, \seq_reverse:N was coded by collecting the items in reverse order after an \exp_stop_f: marker.

```

\seq_greverse:N
\seq_greverse:c
\__seq_reverse>NN
\__seq_reverse_item:nwn
\__seq_reverse_item:nw

\cs_new_protected:Npn \seq_reverse:N #1
{
    \cs_set_eq:NN \@@_item:n \__reverse_item:nw
    \tl_set:Nf #2 { #2 \exp_stop_f: }
}
\cs_new:Npn \__reverse_item:nw #1 #2 \exp_stop_f:
{
    #2 \exp_stop_f:
    \@@_item:n {#1}
}

```

At first, this seems optimal, since we can forget about each item as soon as it is placed after \exp_stop_f:. Unfortunately, TeX's usual tail recursion does not take place in this case: since the following __seq_reverse_item:nw only reads tokens until \exp_stop_f:, and never reads the \@@_item:n {#1} left by the previous call, TeX cannot remove that previous call from the stack, and in particular must retain the various macro parameters in memory, until the end of the replacement text is reached. The stack is thus only flushed after all the __seq_reverse_item:nw are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. TeX can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of \exp_not:n. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

```

7711 \cs_new_protected:Npn \seq_reverse:N
7712  { \__seq_reverse>NN \__kernel_tl_set:Nx }
7713 \cs_new_protected:Npn \seq_greverse:N
7714  { \__seq_reverse>NN \__kernel_tl_gset:Nx }
7715 \cs_new_protected:Npn \__seq_reverse>NN #1 #2
7716  {
7717      \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
7718      \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
7719      #1 #2 { #2 \exp_not:n { } }
7720      \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
7721  }
7722 \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
7723  {
7724      #2
7725      \exp_not:n { \__seq_item:n {#1} #3 }
7726  }
7727 \cs_generate_variant:Nn \seq_reverse:N { c }
7728 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End definition for `\seq_reverse:N` and others. These functions are documented on page 79.)

`\seq_sort:Nn` Implemented in `l3sort`.

`\seq_sort:cn`

`\seq_gsort:Nn`

`\seq_gsort:cn`

11.4 Sequence conditionals

Similar to token lists, we compare with the empty sequence.

```

7729 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
7730   {
7731     \if_meaning:w #1 \c_empty_seq
7732       \prg_return_true:
7733     \else:
7734       \prg_return_false:
7735     \fi:
7736   }
7737 \prg_generate_conditional_variant:Nnn \seq_if_empty:N
7738   { c } { p , T , F , TF }

```

(End definition for `\seq_if_empty:NTF`. This function is documented on page 80.)

`\seq_shuffle:N`

`\seq_shuffle:c`

`\seq_gshuffle:N`

`\seq_gshuffle:c`

`__seq_shuffle:NN`

`__seq_shuffle_item:n`

`\g__seq_internal_seq`

We apply the Fisher–Yates shuffle, storing items in `\toks` registers. We use the primitive `\tex_uniformdeviate:D` for speed reasons. Its non-uniformity is of order its argument divided by 2^{28} , not too bad for small lists. For sequences with more than 13 elements there are more possible permutations than possible seeds ($13! > 2^{28}$) so the question of uniformity is somewhat moot. The integer variables are declared in `l3int`: load-order issues.

```

7739 \cs_if_exist:NTF \tex_uniformdeviate:D
7740   {
7741     \seq_new:N \g__seq_internal_seq
7742     \cs_new_protected:Npn \seq_shuffle:N { \__seq_shuffle:NN \seq_set_eq:NN }
7743     \cs_new_protected:Npn \seq_gshuffle:N { \__seq_shuffle:NN \seq_gset_eq:NN }
7744     \cs_new_protected:Npn \__seq_shuffle:NN #1#2
7745     {
7746       \int_compare:nNnTF { \seq_count:N #2 } > \c_max_register_int
7747         {
7748           \__kernel_msg_error:nnx { kernel } { shuffle-too-large }
7749             { \token_to_str:N #2 }
7750         }
7751     }
7752     \group_begin:
7753       \int_zero:N \l__seq_internal_a_int
7754       \__seq_push_item_def:
7755       \cs_gset_eq:NN \__seq_item:n \__seq_shuffle_item:n
7756         #2
7757       \__seq_pop_item_def:
7758       \seq_gset_from_inline_x:Nnn \g__seq_internal_seq
7759         { \int_step_function:nN { \l__seq_internal_a_int } }
7760         { \tex_the:D \tex_toks:D ##1 }
7761     \group_end:
7762     #1 #2 \g__seq_internal_seq
7763     \seq_gclear:N \g__seq_internal_seq
7764   }

```

```

7765   }
7766   \cs_new_protected:Npn \__seq_shuffle_item:n
7767   {
7768     \int_incr:N \l__seq_internal_a_int
7769     \int_set:Nn \l__seq_internal_b_int
7770       { 1 + \tex_uniformdeviate:D \l__seq_internal_a_int }
7771     \tex_toks:D \l__seq_internal_a_int
7772       = \tex_toks:D \l__seq_internal_b_int
7773     \tex_toks:D \l__seq_internal_b_int
7774   }
7775 }
7776 {
7777   \cs_new_protected:Npn \seq_shuffle:N #1
7778   {
7779     \__kernel_msg_error:nnn { kernel } { fp-no-random }
7780       { \seq_shuffle:N #1 }
7781   }
7782   \cs_new_eq:NN \seq_gshuffle:N \seq_shuffle:N
7783 }
7784 \cs_generate_variant:Nn \seq_shuffle:N { c }
7785 \cs_generate_variant:Nn \seq_gshuffle:N { c }

```

(End definition for `\seq_shuffle:N` and others. These functions are documented on page 80.)

\seq_if_in:NnTF The approach here is to define `__seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\group_end: \prg_return_true:` is inserted after skipping over the rest of the recursion. On the other hand, if there is no match then the loop breaks, returning `\prg_return_false:`. Everything is inside a group so that `__seq_item:n` is preserved in nested situations.

```

7786 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
7787   { T , F , TF }
7788   {
7789     \group_begin:
7790       \tl_set:Nn \l__seq_internal_a_tl {#2}
7791       \cs_set_protected:Npn \__seq_item:n ##1
7792         {
7793           \tl_set:Nn \l__seq_internal_b_tl {##1}
7794           \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
7795             \exp_after:wn \__seq_if_in:
7796           \fi:
7797         }
7798       #1
7799       \group_end:
7800       \prg_return_false:
7801       \prg_break_point:
7802     }
7803 \cs_new:Npn \__seq_if_in:
7804   { \prg_break:n { \group_end: \prg_return_true: } }
7805 \prg_generate_conditional_variant:Nnn \seq_if_in:Nn
7806   { NV , Nv , No , Nx , c , cV , cv , co , cx } { T , F , TF }

```

(End definition for `\seq_if_in:NnTF` and `__seq_if_in:`. This function is documented on page 80.)

11.5 Recovering data from sequences

__seq_pop:NNNN
__seq_pop_TF:NNNN

The two pop functions share their emptiness tests. We also use a common emptiness test for all branching get and pop functions.

```

7807 \cs_new_protected:Npn \_\_seq_pop:NNNN #1#2#3#4
7808 {
7809     \if_meaning:w #3 \c_empty_seq
7810         \tl_set:Nn #4 { \q_no_value }
7811     \else:
7812         #1#2#3#4
7813     \fi:
7814 }
7815 \cs_new_protected:Npn \_\_seq_pop_TF:NNNN #1#2#3#4
7816 {
7817     \if_meaning:w #3 \c_empty_seq
7818         % \tl_set:Nn #4 { \q_no_value }
7819         \prg_return_false:
7820     \else:
7821         #1#2#3#4
7822         \prg_return_true:
7823     \fi:
7824 }
```

(End definition for __seq_pop:NNNN and __seq_pop_TF:NNNN.)

\seq_get_left:NN
\seq_get_left:cN
__seq_get_left:wnw

Getting an item from the left of a sequence is pretty easy: just trim off the first item after __seq_item:n at the start. We append a \q_no_value item to cover the case of an empty sequence

```

7825 \cs_new_protected:Npn \seq_get_left:NN #1#2
7826 {
7827     \__kernel_tl_set:Nx #2
7828     {
7829         \exp_after:wN \_\_seq_get_left:wnw
7830         #1 \_\_seq_item:n { \q_no_value } \s_\_seq_stop
7831     }
7832 }
7833 \cs_new:Npn \_\_seq_get_left:wnw #1 \_\_seq_item:n #2#3 \s_\_seq_stop
7834 { \exp_not:n {#2} }
7835 \cs_generate_variant:Nn \seq_get_left:NN { c }
```

(End definition for \seq_get_left:NN and __seq_get_left:wnw. This function is documented on page 77.)

\seq_pop_left:NN
\seq_pop_left:cN
\seq_gpop_left:NN
\seq_gpop_left:cN
__seq_pop_left:NNN
__seq_pop_left:wnwNNN

The approach to popping an item is pretty similar to that to get an item, with the only difference being that the sequence itself has to be redefined. This makes it more sensible to use an auxiliary function for the local and global cases.

```

7836 \cs_new_protected:Npn \seq_pop_left:NN
7837 { \_\_seq_pop:NNNN \_\_seq_pop_left:NNN \tl_set:Nn }
7838 \cs_new_protected:Npn \seq_gpop_left:NN
7839 { \_\_seq_pop:NNNN \_\_seq_pop_left:NNN \tl_gset:Nn }
7840 \cs_new_protected:Npn \_\_seq_pop_left:NNN #1#2#3
7841 { \exp_after:wN \_\_seq_pop_left:wnwNNN #2 \s_\_seq_stop #1#2#3 }
7842 \cs_new_protected:Npn \_\_seq_pop_left:wnwNNN
7843 #1 \_\_seq_item:n #2#3 \s_\_seq_stop #4#5#6
```

```

7844  {
7845    #4 #5 { #1 #3 }
7846    \tl_set:Nn #6 {#2}
7847  }
7848 \cs_generate_variant:Nn \seq_pop_left:NN { c }
7849 \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End definition for `\seq_pop_left:NN` and others. These functions are documented on page 77.)

`\seq_get_right:NN`
`\seq_get_right:cN`
`__seq_get_right_loop:nw`
`__seq_get_right_end:NnN`

First remove `\s_seq` and prepend `\q_no_value`. The first argument of `__seq_get_right_loop:nw` is the last item found, and the second argument is empty until the end of the loop, where it is code that applies `\exp_not:n` to the last item and ends the loop.

```

7850 \cs_new_protected:Npn \seq_get_right:NN #1#2
7851  {
7852    \__kernel_tl_set:Nx #2
7853    {
7854      \exp_after:wN \use_i_i:nnn
7855      \exp_after:wN \__seq_get_right_loop:nw
7856      \exp_after:wN \q_no_value
7857      #1
7858      \__seq_get_right_end:NnN \__seq_item:n
7859    }
7860  }
7861 \cs_new:Npn \__seq_get_right_loop:nw #1#2 \__seq_item:n
7862  {
7863    #2 \use_none:n {#1}
7864    \__seq_get_right_loop:nw
7865  }
7866 \cs_new:Npn \__seq_get_right_end:NnN #1#2#3 { \exp_not:n {#2} }
7867 \cs_generate_variant:Nn \seq_get_right:NN { c }

```

(End definition for `\seq_get_right:NN`, `__seq_get_right_loop:nw`, and `__seq_get_right_end:NnN`. This function is documented on page 77.)

`\seq_pop_right:NN`
`\seq_pop_right:cN`
`\seq_gpop_right:NN`
`\seq_gpop_right:cN`
`__seq_pop_right:NNN`
`__seq_pop_right_loop:nn`

The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a “flexible length” way to set a token list variable. This is supplied by the `{ \if_false: } \fi: ... \if_false: { \fi: }` construct. Using an `x`-type expansion and a “non-expanding” definition for `__seq_item:n`, the left-most $n - 1$ entries in a sequence of n items are stored back in the sequence. That needs a loop of unknown length, hence using the strange `\if_false:` way of including braces. When the last item of the sequence is reached, the closing brace for the assignment is inserted, and `\tl_set:Nn #3` is inserted in front of the final entry. This therefore does the pop assignment. One more iteration is performed, with an empty argument and `\use_none:nn`, which finally stops the loop.

```

7868 \cs_new_protected:Npn \seq_pop_right:NN
7869  { \__seq_pop:NNNN \__seq_pop_right:NNN \__kernel_tl_set:Nx }
7870 \cs_new_protected:Npn \seq_gpop_right:NN
7871  { \__seq_pop:NNNN \__seq_pop_right:NNN \__kernel_tl_gset:Nx }
7872 \cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
7873  {
7874    \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
7875    \cs_set_eq:NN \__seq_item:n \scan_stop:
7876    #1 #2
7877    { \if_false: } \fi: \s_seq

```

```

7878     \exp_after:wN \use_i:nnn
7879     \exp_after:wN \__seq_pop_right_loop:nn
7880     #2
7881     {
7882         \if_false: { \fi: }
7883         \__kernel_tl_set:Nx #3
7884     }
7885     { } \use_none:nn
7886     \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
7887 }
7888 \cs_new:Npn \__seq_pop_right_loop:nn #1#2
7889 {
7890     #2 { \exp_not:n {#1} }
7891     \__seq_pop_right_loop:nn
7892 }
7893 \cs_generate_variant:Nn \seq_pop_right:NN { c }
7894 \cs_generate_variant:Nn \seq_gpop_right:NN { c }

```

(End definition for `\seq_pop_right:NN` and others. These functions are documented on page 77.)

`\seq_get_left:NNTF` Getting from the left or right with a check on the results. The first argument to `__-seq_pop_TF:NNNN` is left unused.

```

\seq_get_right:NNTF
\seq_get_right:cNTF
7895 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
7896 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
7897 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
7898 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
7899 \prg_generate_conditional_variant:Nnm \seq_get_left:NN
7900 { c } { T , F , TF }
7901 \prg_generate_conditional_variant:Nnn \seq_get_right:NN
7902 { c } { T , F , TF }

```

(End definition for `\seq_get_left:NNTF` and `\seq_get_right:NNTF`. These functions are documented on page 78.)

`\seq_pop_left:NNTF` More or less the same for popping.

```

\seq_pop_left:cNTF
\seq_gpop_left:NNTF
\seq_gpop_left:cNTF
7903 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2
7904 { T , F , TF }
7905 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
\seq_pop_right:NNTF
\seq_pop_right:cNTF
7906 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2
7907 { T , F , TF }
7908 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
\seq_gpop_right:NNTF
\seq_gpop_right:cNTF
7909 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2
7910 { T , F , TF }
7911 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \__kernel_tl_set:Nx #1 #2 }
7912 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2
7913 { T , F , TF }
7914 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \__kernel_tl_gset:Nx #1 #2 }
7915 \prg_generate_conditional_variant:Nnm \seq_pop_left:NN { c }
7916 { T , F , TF }
7917 \prg_generate_conditional_variant:Nnn \seq_gpop_left:NN { c }
7918 { T , F , TF }
7919 \prg_generate_conditional_variant:Nnn \seq_pop_right:NN { c }
7920 { T , F , TF }
7921 \prg_generate_conditional_variant:Nnn \seq_gpop_right:NN { c }
7922 { T , F , TF }

```

(End definition for `\seq_pop_left:NNTF` and others. These functions are documented on page 78.)

```
\seq_item:Nn
\seq_item:cn
\__seq_item:wNn
\__seq_item:nN
\__seq_item:nwn
7923 \cs_new:Npn \seq_item:Nn #1
7924   { \exp_after:wN \__seq_item:wNn #1 \s__seq_stop #1 }
7925 \cs_new:Npn \__seq_item:wNn \s__seq #1 \s__seq_stop #2#3
7926   {
7927     \exp_args:Nf \__seq_item:nwn
7928       { \exp_args:Nf \__seq_item:nN { \int_eval:n {#3} } #2 }
7929     #1
7930     \prg_break: \__seq_item:n { }
7931     \prg_break_point:
7932   }
7933 \cs_new:Npn \__seq_item:nN #1#2
7934   {
7935     \int_compare:nNnTF {#1} < 0
7936       { \int_eval:n { \seq_count:N #2 + 1 + #1 } }
7937     {#1}
7938   }
7939 \cs_new:Npn \__seq_item:nwn #1#2 \__seq_item:n #3
7940   {
7941     #2
7942     \int_compare:nNnTF {#1} = 1
7943       { \prg_break:n { \exp_not:n {#3} } }
7944       { \exp_args:Nf \__seq_item:nwn { \int_eval:n { #1 - 1 } } }
7945   }
7946 \cs_generate_variant:Nn \seq_item:Nn { c }
```

(End definition for `\seq_item:Nn` and others. This function is documented on page 77.)

`\seq_rand_item:N` Importantly, `\seq_item:Nn` only evaluates its argument once.

```
\seq_rand_item:c
7947 \cs_new:Npn \seq_rand_item:N #1
7948   {
7949     \seq_if_empty:NF #1
7950       { \seq_item:Nn #1 { \int_rand:nn { 1 } { \seq_count:N #1 } } }
7951   }
7952 \cs_generate_variant:Nn \seq_rand_item:N { c }
```

(End definition for `\seq_rand_item:N`. This function is documented on page 78.)

11.6 Mapping to sequences

`\seq_map_break:` To break a function, the special token `\prg_break_point:Nn` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted.

```
7953 \cs_new:Npn \seq_map_break:
7954   { \prg_map_break:Nn \seq_map_break: { } }
7955 \cs_new:Npn \seq_map_break:n
7956   { \prg_map_break:Nn \seq_map_break: }
```

(End definition for `\seq_map_break:` and `\seq_map_break:n`. These functions are documented on page 82.)

`\seq_map_function:NN` The idea here is to apply the code of #2 to each item in the sequence without altering the definition of `_seq_item:n`. The argument delimited by `_seq_item:n` is almost always empty, except at the end of the loop where it is `\prg_break:`. This allows to break the loop without needing to do a (relatively-expensive) quark test.

```
7957 \cs_new:Npn \seq_map_function:NN #1#2
7958 {
7959     \exp_after:wN \use_i_i:nnn
7960     \exp_after:wN \_seq_map_function:Nw
7961     \exp_after:wN #2
7962     #1
7963     \prg_break: \_seq_item:n { } \prg_break_point:
7964     \prg_break_point:Nn \seq_map_break: { }
7965 }
7966 \cs_new:Npn \_seq_map_function:Nw #1#2 \_seq_item:n #3
7967 {
7968     #2
7969     #1 {#3}
7970     \_seq_map_function:Nw #1
7971 }
7972 \cs_generate_variant:Nn \seq_map_function:NN { c }
```

(End definition for `\seq_map_function:NN` and `_seq_map_function>NNn`. This function is documented on page 80.)

`_seq_push_item_def:n` The definition of `_seq_item:n` needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

```
7973 \cs_new_protected:Npn \_seq_push_item_def:n
7974 {
7975     \_seq_push_item_def:
7976     \cs_gset:Npn \_seq_item:n ##1
7977 }
7978 \cs_new_protected:Npn \_seq_push_item_def:x
7979 {
7980     \_seq_push_item_def:
7981     \cs_gset:Npx \_seq_item:n ##1
7982 }
7983 \cs_new_protected:Npn \_seq_push_item_def:
7984 {
7985     \int_gincr:N \g_kernel_prg_map_int
7986     \cs_gset_eq:cN { \_seq_map_ \int_use:N \g_kernel_prg_map_int :w }
7987     \_seq_item:n
7988 }
7989 \cs_new_protected:Npn \_seq_pop_item_def:
7990 {
7991     \cs_gset_eq:Nc \_seq_item:n
7992     { \_seq_map_ \int_use:N \g_kernel_prg_map_int :w }
7993     \int_gdecr:N \g_kernel_prg_map_int
7994 }
```

(End definition for `_seq_push_item_def:n`, `_seq_push_item_def:`, and `_seq_pop_item_def:..`)

\seq_map_inline:Nn
\seq_map_inline:cn

The idea here is that `_seq_item:n` is already “applied” to each item in a sequence, and so an in-line mapping is just a case of redefining `_seq_item:n`.

```
7995 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
7996 {
7997     \_seq_push_item_def:n {#2}
7998     #1
7999     \prg_break_point:Nn \seq_map_break: { \_seq_pop_item_def: }
8000 }
8001 \cs_generate_variant:Nn \seq_map_inline:Nn { c }
```

(End definition for `\seq_map_inline:Nn`. This function is documented on page 80.)

\seq_map_tokens:Nn
\seq_map_tokens:cn
_seq_map_tokens:nw

This is based on the function mapping but using the same tricks as described for `\prop_map_tokens:Nn`. The idea is to remove the leading `\s_seq` and apply the tokens such that they are safe with the break points, hence the `\use:n`.

```
8002 \cs_new:Npn \seq_map_tokens:Nn #1#2
8003 {
8004     \exp_last_unbraced:Nno
8005     \use_i:nn { \_seq_map_tokens:nw {#2} } #1
8006     \prg_break: \_seq_item:n { } \prg_break_point:
8007     \prg_break_point:Nn \seq_map_break: { }
8008 }
8009 \cs_generate_variant:Nn \seq_map_tokens:Nn { c }
8010 \cs_new:Npn \_seq_map_tokens:nw #1#2 \_seq_item:n #3
8011 {
8012     #2
8013     \use:n {#1} {#3}
8014     \_seq_map_tokens:nw {#1}
8015 }
```

(End definition for `\seq_map_tokens:Nn` and `_seq_map_tokens:nw`. This function is documented on page 81.)

\seq_map_variable:NNn
\seq_map_variable:Ncn
\seq_map_variable:cNn
\seq_map_variable:ccn

This is just a specialised version of the in-line mapping function, using an `x`-type expansion for the code set up so that the number of `#` tokens required is as expected.

```
8016 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
8017 {
8018     \_seq_push_item_def:x
8019     {
8020         \tl_set:Nn \exp_not:N #2 {##1}
8021         \exp_not:n {#3}
8022     }
8023     #1
8024     \prg_break_point:Nn \seq_map_break: { \_seq_pop_item_def: }
8025 }
8026 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
8027 \cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }
```

(End definition for `\seq_map_variable:NNn`. This function is documented on page 81.)

\seq_map_indexed_function:NN
\seq_map_indexed_inline:Nn
_seq_map_indexed:nNn
_seq_map_indexed:Nw

Similar to `\seq_map_function:NN` but we keep track of the item index as a `,`-delimited argument of `_seq_map_indexed:Nw`.

```
8028 \cs_new:Npn \seq_map_indexed_function:NN #1#2
8029 {
```

```

8030     \_\_seq\_map\_indexed:NN #1#2
8031     \prg\_break\_point:Nn \seq\_map\_break: { }
8032   }
8033 \cs\_new\_protected:Npn \seq\_map\_indexed\_inline:Nn #1#2
8034   {
8035     \int\_gincr:N \g\_kernel\_prg\_map\_int
8036     \cs\_gset\_protected:cpn
8037       { \_\_seq\_map\_ \int\_use:N \g\_kernel\_prg\_map\_int :w } ##1##2 {#2}
8038     \exp\_args:NNc \_\_seq\_map\_indexed:NN #1
8039       { \_\_seq\_map\_ \int\_use:N \g\_kernel\_prg\_map\_int :w }
8040     \prg\_break\_point:Nn \seq\_map\_break:
8041       { \int\_gdecr:N \g\_kernel\_prg\_map\_int }
8042   }
8043 \cs\_new:Npn \_\_seq\_map\_indexed:NN #1#2
8044   {
8045     \exp\_after:wN \_\_seq\_map\_indexed:Nw
8046     \exp\_after:wN #2
8047     \int\_value:w 1
8048     \exp\_after:wN \use\_i:nn
8049     \exp\_after:wN ;
8050     #1
8051     \prg\_break: \_\_seq\_item:n { } \prg\_break\_point:
8052   }
8053 \cs\_new:Npn \_\_seq\_map\_indexed:Nw #1#2 ; #3 \_\_seq\_item:n #4
8054   {
8055     #3
8056     #1 {#2} {#4}
8057     \exp\_after:wN \_\_seq\_map\_indexed:Nw
8058     \exp\_after:wN #1
8059     \int\_value:w \int\_eval:w 1 + #2 ;
8060   }

```

(End definition for `\seq_map_indexed_function:NN` and others. These functions are documented on page 81.)

`\seq_set_map_x:NNn`
`\seq_gset_map_x:NNn`

`__seq_set_map_x:NNNn`

Very similar to `\seq_set_filter:NNn`. We could actually merge the two within a single function, but it would have weird semantics.

```

8061 \cs\_new\_protected:Npn \seq\_set\_map\_x:NNn
8062   { \_\_seq\_set\_map\_x:NNNn \_\_kernel\_tl\_set:Nx }
8063 \cs\_new\_protected:Npn \seq\_gset\_map\_x:NNn
8064   { \_\_seq\_set\_map\_x:NNNn \_\_kernel\_tl\_gset:Nx }
8065 \cs\_new\_protected:Npn \_\_seq\_set\_map\_x:NNNn #1#2#3#4
8066   {
8067     \_\_seq\_push\_item\_def:n { \exp\_not:N \_\_seq\_item:n {#4} }
8068     #1 #2 { #3 }
8069     \_\_seq\_pop\_item\_def:
8070   }

```

(End definition for `\seq_set_map_x:NNn`, `\seq_gset_map_x:NNn`, and `__seq_set_map_x:NNNn`. These functions are documented on page 83.)

`\seq_set_map:NNn`
`\seq_gset_map:NNn`

`__seq_set_map:NNNn`

Similar to `\seq_set_map_x:NNn`, but prevents expansion of the <inline function>.

```

8071 \cs\_new\_protected:Npn \seq\_set\_map:NNn
8072   { \_\_seq\_set\_map:NNNn \_\_kernel\_tl\_set:Nx }
8073 \cs\_new\_protected:Npn \seq\_gset\_map:NNn

```

```

8074 { \__seq_set_map:NNNn \__kernel_t1_gset:Nx }
8075 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
8076 {
8077     \__seq_push_item_def:n { \exp_not:n { \__seq_item:n {#4} } }
8078     #1 #2 { #3 }
8079     \__seq_pop_item_def:
8080 }

```

(End definition for `\seq_set_map:NNn`, `\seq_gset_map:NNn`, and `__seq_set_map:NNNn`. These functions are documented on page 82.)

`\seq_count:N` Since counting the items in a sequence is quite common, we optimize it by grabbing 8 items at a time and correspondingly adding 8 to an integer expression. At the end of the loop, #9 is `__seq_count_end:w` instead of being empty. It removes 8+ and instead places the number of `__seq_item:n` that `__seq_count:w` grabbed before reaching the end of the sequence.

```

8081 \cs_new:Npn \seq_count:N #1
8082 {
8083     \int_eval:n
8084     {
8085         \exp_after:wN \use_i:nn
8086         \exp_after:wN \__seq_count:w
8087         #1
8088         \__seq_count_end:w \__seq_item:n 7
8089         \__seq_count_end:w \__seq_item:n 6
8090         \__seq_count_end:w \__seq_item:n 5
8091         \__seq_count_end:w \__seq_item:n 4
8092         \__seq_count_end:w \__seq_item:n 3
8093         \__seq_count_end:w \__seq_item:n 2
8094         \__seq_count_end:w \__seq_item:n 1
8095         \__seq_count_end:w \__seq_item:n 0
8096         \prg_break_point:
8097     }
8098 }
8099 \cs_new:Npn \__seq_count:w
8100     #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4 \__seq_item:n
8101     #5 \__seq_item:n #6 \__seq_item:n #7 \__seq_item:n #8 #9 \__seq_item:n
8102     { #9 8 + \__seq_count:w }
8103 \cs_new:Npn \__seq_count_end:w 8 + \__seq_count:w #1#2 \prg_break_point: {#1}
8104 \cs_generate_variant:Nn \seq_count:N { c }

```

(End definition for `\seq_count:N`, `__seq_count:w`, and `__seq_count_end:w`. This function is documented on page 83.)

11.7 Using sequences

`\seq_use:Nnnn` See `\clist_use:Nnnn` for a general explanation. The main difference is that we use `__seq_item:n` as a delimiter rather than commas. We also need to add `__seq_item:n` at various places, and `\s_seq`.

```

8105 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
8106 {
8107     \seq_if_exist:NTF #1
8108     {
8109         \int_case:nnF { \seq_count:N #1 }

```

```

8110      {
8111          { 0 } { }
8112          { 1 } { \exp_after:wN \__seq_use>NNnNnn #1 ? { } { } }
8113          { 2 } { \exp_after:wN \__seq_use>NNnNnn #1 {#2} }
8114      }
8115      {
8116          \exp_after:wN \__seq_use_setup:w #1 \__seq_item:n
8117          \s__seq_mark { \__seq_use:nwwwnwn {#3} }
8118          \s__seq_mark { \__seq_use:nwwn {#4} }
8119          \s__seq_stop { }
8120      }
8121  }
8122  {
8123      \__kernel_msg_expandable_error:nnn
8124          { kernel } { bad-variable } {#1}
8125  }
8126  }
8127 \cs_generate_variant:Nn \seq_use:Nnnn { c }
8128 \cs_new:Npn \__seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
8129 \cs_new:Npn \__seq_use_setup:w \s__seq { \__seq_use:nwwwnwn { } }
8130 \cs_new:Npn \__seq_use:nwwwnwn
8131     #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
8132     \s__seq_mark #6#7 \s__seq_stop #8
8133  {
8134      #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
8135      \s__seq_mark {#6} #7 \s__seq_stop { #8 #1 #2 }
8136  }
8137 \cs_new:Npn \__seq_use:nwwn #1 \__seq_item:n #2 #3 \s__seq_stop #4
8138  { \exp_not:n { #4 #1 #2 } }
8139 \cs_new:Npn \seq_use:Nn #1#2
8140  { \seq_use:Nnnn #1 {#2} {#2} {#2} }
8141 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End definition for `\seq_use:Nnnn` and others. These functions are documented on page 83.)

11.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

<code>\seq_push:Nn</code>	Pushing to a sequence is the same as adding on the left.
\seq_push:NV	8142 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
\seq_push:Nv	8143 \cs_new_eq:NN \seq_push:NV \seq_put_left:NV
\seq_push:No	8144 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
\seq_push:Nx	8145 \cs_new_eq:NN \seq_push:No \seq_put_left:No
\seq_push:cn	8146 \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
\seq_push:cV	8147 \cs_new_eq:NN \seq_push:cn \seq_put_left:cN
\seq_push:cV	8148 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
\seq_push:co	8149 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
\seq_push:cX	8150 \cs_new_eq:NN \seq_push:co \seq_put_left:co
<code>\seq_gpush:Nn</code>	8151 \cs_new_eq:NN \seq_push:cX \seq_put_left:cX
\seq_gpush:NV	8152 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
\seq_gpush:Nv	8153 \cs_new_eq:NN \seq_gpush:NV \seq_gput_left:NV
\seq_gpush:No	8154 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
\seq_gpush:Nx	8155 \cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
\seq_gpush:cn	
\seq_gpush:cV	
\seq_gpush:cV	
\seq_gpush:co	
\seq_gpush:cX	

```

8156 \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
8157 \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
8158 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
8159 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
8160 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
8161 \cs_new_eq:NN \seq_gpush:cX \seq_gput_left:cX

```

(End definition for `\seq_push:Nn` and `\seq_gpush:Nn`. These functions are documented on page 85.)

`\seq_get:NN` In most cases, getting items from the stack does not need to specify that this is from the left. So alias are provided.

```

8162 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
8163 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
8164 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
8165 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
8166 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
8167 \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN

```

(End definition for `\seq_get:NN`, `\seq_pop:NN`, and `\seq_gpop:NN`. These functions are documented on page 84.)

`\seq_get:NNTF` More copies.

```

8168 \prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }
8169 \prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }
8170 \prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
8171 \prg_new_eq_conditional:NNn \seq_pop:cN \seq_pop_left:cN { T , F , TF }
8172 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
8173 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }

```

(End definition for `\seq_get:NNTF`, `\seq_pop:NNTF`, and `\seq_gpop:NNTF`. These functions are documented on page 84.)

11.9 Viewing sequences

`\seq_show:N` Apply the general `\msg_show:nnnnnn`.

```

8174 \cs_new_protected:Npn \seq_show:N { \__seq_show:NN \msg_show:nnxxxx }
8175 \cs_generate_variant:Nn \seq_show:N { c }
8176 \cs_new_protected:Npn \seq_log:N { \__seq_show:NN \msg_log:nnxxxx }
8177 \cs_generate_variant:Nn \seq_log:N { c }
8178 \cs_new_protected:Npn \__seq_show:NN #1#2
8179 {
8200     \__kernel_chk_defined:NT #2
8201     {
8202         #1 { LaTeX/kernel } { show-seq }
8203         { \token_to_str:N #2 }
8204         { \seq_map_function:NN #2 \msg_show_item:n }
8205         { } { }
8206     }
8207 }

```

(End definition for `\seq_show:N`, `\seq_log:N`, and `__seq_show:NN`. These functions are documented on page 87.)

11.10 Scratch sequences

\l_tmpa_seq Temporary comma list variables.
8188 \seq_new:N \l_tmpa_seq
8189 \seq_new:N \l_tmpb_seq
8190 \seq_new:N \g_tmpa_seq
8191 \seq_new:N \g_tmpb_seq

(End definition for `\l_tmpa_seq` and others. These variables are documented on page 87.)
8192 ⟨/package⟩

12 I3int implementation

8193 ⟨*package⟩
8194 ⟨@=int⟩

The following test files are used for this code: `m3int001,m3int002,m3int03`.

\c_max_register_int Done in I3basics.

(End definition for `\c_max_register_int`. This variable is documented on page 100.)

__int_to_roman:w Done in I3basics.

\if_int_compare:w (End definition for `__int_to_roman:w` and `\if_int_compare:w`. This function is documented on page 101.)

\or: Done in I3basics.

(End definition for `\or:`. This function is documented on page 101.)

\int_value:w Here are the remaining primitives for number comparisons and expressions.

8195 \cs_new_eq:NN \int_value:w \tex_number:D
8196 \cs_new_eq:NN __int_eval:w \tex_numexpr:D
8197 \cs_new_eq:NN __int_eval_end: \tex_relax:D
8198 \cs_new_eq:NN \if_int_odd:w \tex_ifodd:D
8199 \cs_new_eq:NN \if_case:w \tex_ifcase:D

(End definition for `\int_value:w` and others. These functions are documented on page 101.)

\s_int_mark Scan marks used throughout the module.

8200 \scan_new:N \s_int_mark
8201 \scan_new:N \s_int_stop

(End definition for `\s_int_mark` and `\s_int_stop`.)

__int_use_none_delimit_by_s_stop:w Function to gobble until a scan mark.

8202 \cs_new:Npn __int_use_none_delimit_by_s_stop:w #1 \s_int_stop { }

(End definition for `__int_use_none_delimit_by_s_stop:w`.)

\q_int_recursion_tail Quarks for recursion.

8203 \quark_new:N \q_int_recursion_tail
8204 \quark_new:N \q_int_recursion_stop

(End definition for `\q_int_recursion_tail` and `\q_int_recursion_stop`.)

```
\_\_int_if_recursion_tail_stop_do:Nn
\_\_int_if_recursion_tail_stop:N
 8205 \_\_kernel_quark_new_test:N \_\_int_if_recursion_tail_stop_do:Nn
 8206 \_\_kernel_quark_new_test:N \_\_int_if_recursion_tail_stop:N

(End definition for \_\_int_if_recursion_tail_stop_do:Nn and \_\_int_if_recursion_tail_stop:N.)
```

12.1 Integer expressions

\int_eval:n Wrapper for __int_eval:w: can be used in an integer expression or directly in the input stream. When debugging, use parentheses to catch early termination.

```
8207 \cs_new:Npn \int_eval:n #1
 8208   { \int_value:w \_\_int_eval:w #1 \_\_int_eval_end: }
 8209 \cs_new:Npn \int_eval:w { \int_value:w \_\_int_eval:w }
```

(End definition for \int_eval:n and \int_eval:w. These functions are documented on page 89.)

\int_sign:n See \int_abs:n. Evaluate the expression once (and when debugging is enabled, check that the expression is well-formed), then test the first character to determine the sign. This is wrapped in \int_value:w ... \exp_stop_f: to ensure a fixed number of expansions and to avoid dealing with closing the conditionals.

```
8210 \cs_new:Npn \int_sign:n #1
 8211   {
 8212     \int_value:w \exp_after:wN \_\_int_sign:Nw
 8213     \int_value:w \_\_int_eval:w #1 \_\_int_eval_end: ;
 8214     \exp_stop_f:
 8215   }
 8216 \cs_new:Npn \_\_int_sign:Nw #1#2 ;
 8217   {
 8218     \if_meaning:w 0 #1
 8219     0
 8220     \else:
 8221     \if_meaning:w - #1 - \fi: 1
 8222     \fi:
 8223   }
```

(End definition for \int_sign:n and __int_sign:Nw. This function is documented on page 90.)

\int_abs:n Functions for min, max, and absolute value with only one evaluation. The absolute value is obtained by removing a leading sign if any. All three functions expand in two steps.

```
8224 \cs_new:Npn \int_abs:n #1
 8225   {
 8226     \int_value:w \exp_after:wN \_\_int_abs:N
 8227     \int_value:w \_\_int_eval:w #1 \_\_int_eval_end:
 8228     \exp_stop_f:
 8229   }
 8230 \cs_new:Npn \_\_int_abs:N #1
 8231   { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
 8232 \cs_set:Npn \int_max:nn #1#2
 8233   {
 8234     \int_value:w \exp_after:wN \_\_int_maxmin:wwN
 8235     \int_value:w \_\_int_eval:w #1 \exp_after:wN ;
 8236     \int_value:w \_\_int_eval:w #2 ;
 8237     >
 8238     \exp_stop_f:
```

```

8239     }
8240 \cs_set:Npn \int_min:nn #1#2
8241 {
8242     \int_value:w \exp_after:wN \__int_maxmin:wwN
8243     \int_value:w \__int_eval:w #1 \exp_after:wN ;
8244     \int_value:w \__int_eval:w #2 ;
8245     <
8246     \exp_stop_f:
8247 }
8248 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
8249 {
8250     \if_int_compare:w #1 #3 #2 ~
8251     #1
8252     \else:
8253     #2
8254     \fi:
8255 }

```

(End definition for `\int_abs:n` and others. These functions are documented on page 90.)

`\int_div_truncate:nn`
`\int_div_round:nn`
`\int_mod:nn`

`__int_div_truncate:NwNw`
`__int_mod:ww`

As `__int_eval:w` rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expensive to evaluate (e.g., `\tl_count:n`). If the numerator `#1#2` is 0, then we divide 0 by the denominator (this ensures that $0/0$ is correctly reported as an error). Otherwise, shift the numerator `#1#2` towards 0 by $(|#3#4|-1)/2$, which we round away from zero. It turns out that this quantity exactly compensates the difference between ε -TeX's rounding and the truncating behaviour that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

```

8256 \cs_new:Npn \int_div_truncate:nn #1#2
8257 {
8258     \int_value:w \__int_eval:w
8259     \exp_after:wN \__int_div_truncate:NwNw
8260     \int_value:w \__int_eval:w #1 \exp_after:wN ;
8261     \int_value:w \__int_eval:w #2 ;
8262     \__int_eval_end:
8263 }
8264 \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
8265 {
8266     \if_meaning:w 0 #1
8267     0
8268     \else:
8269     (
8270         #1#2
8271         \if_meaning:w - #1 + \else: - \fi:
8272         ( \if_meaning:w - #3 - \fi: #3#4 - 1 ) / 2
8273     )
8274     \fi:
8275     / #3#4
8276 }

```

For the sake of completeness:

```

8277 \cs_new:Npn \int_div_round:nn #1#2
8278 { \int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }

```

Finally there's the modulus operation.

```

8279 \cs_new:Npn \int_mod:nn #1#2
8280 {
8281     \int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
8282     \int_value:w \__int_eval:w #1 \exp_after:wN ;
8283     \int_value:w \__int_eval:w #2 ;
8284     \__int_eval_end:
8285 }
8286 \cs_new:Npn \__int_mod:ww #1; #2;
8287 { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }
```

(End definition for `\int_div_truncate:nn` and others. These functions are documented on page 90.)

`__kernel_int_add:nnn`

Equivalent to `\int_eval:n {#1+#2+#3}` except that overflow only occurs if the final result overflows $[-2^{31} + 1, 2^{31} - 1]$. The idea is to choose the order in which the three numbers are added together. If #1 and #2 have opposite signs (one is in $[-2^{31} + 1, -1]$ and the other in $[0, 2^{31} - 1]$) then #1+#2 cannot overflow so we compute the result as #1+#2+#3. If they have the same sign, then either #3 has the same sign and the order does not matter, or #3 has the opposite sign and any order in which #3 is not last will work. We use #1+#3+#2.

```

8288 \cs_new:Npn \__kernel_int_add:nnn #1#2#3
8289 {
8290     \int_value:w \__int_eval:w #1
8291     \if_int_compare:w #2 < \c_zero_int \exp_after:wN \reverse_if:N \fi:
8292     \if_int_compare:w #1 < \c_zero_int + #2 + #3 \else: + #3 + #2 \fi:
8293     \__int_eval_end:
8294 }
```

(End definition for `__kernel_int_add:nnn`.)

12.2 Creating and initialising integers

`\int_new:N`

Two ways to do this: one for the format and one for the L^AT_EX 2 _{ε} package. In plain T_EX, `\newcount` (and other allocators) are `\outer`: to allow the code here to work in “generic” mode this is therefore accessed by name. (The same applies to `\newbox`, `\newdimen` and so on.)

```

8295 \cs_new_protected:Npn \int_new:N #1
8296 {
8297     \__kernel_chk_if_free_cs:N #1
8298     \cs:w newcount \cs_end: #1
8299 }
8300 \cs_generate_variant:Nn \int_new:N { c }
```

(End definition for `\int_new:N`. This function is documented on page 90.)

`\int_const:Nn`

`\int_const:cN`

`__int_constdef:Nw`

`\c_int_max_constdef_int`

As stated, most constants can be defined as `\chardef` or `\mathchardef` but that's engine dependent. As a result, there is some set up code to determine what can be done. No full engine testing just yet so everything is a little awkward. We cannot use `\int_gset:Nn` because (when `check-declarations` is enabled) this runs some checks that constants would fail.

```

8301 \cs_new_protected:Npn \int_const:Nn #1#2
8302 {
8303     \int_compare:nNnTF {#2} < \c_zero_int
```

```

8304 {
8305   \int_new:N #1
8306   \tex_global:D
8307 }
8308 {
8309   \int_compare:nNnTF {#2} > \c_int_max_constdef_int
8310   {
8311     \int_new:N #1
8312     \tex_global:D
8313   }
8314   {
8315     \__kernel_chk_if_free_cs:N #1
8316     \tex_global:D \__int_constdef:Nw
8317   }
8318 }
8319 #1 = \__int_eval:w #2 \__int_eval_end:
8320 }
8321 \cs_generate_variant:Nn \int_const:Nn { c }
8322 \if_int_odd:w 0
8323   \cs_if_exist:NT \tex_luatexversion:D { 1 }
8324   \cs_if_exist:NT \tex_omathchardef:D { 1 }
8325   \cs_if_exist:NT \tex_XeTeXversion:D { 1 } ~
8326   \cs_if_exist:NTF \tex_omathchardef:D
8327   { \cs_new_eq:NN \__int_constdef:Nw \tex_omathchardef:D }
8328   { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
8329   \__int_constdef:Nw \c_int_max_constdef_int 1114111 ~
8330 \else:
8331   \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
8332   \tex_mathchardef:D \c_int_max_constdef_int 32767 ~
8333 \fi:

```

(End definition for `\int_const:Nn`, `__int_constdef:Nw`, and `\c_int_max_constdef_int`. This function is documented on page 91.)

`\int_zero:N` Functions that reset an `<integer>` register to zero.

```

\int_zero:c 8334 \cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero_int }
\int_gzero:N 8335 \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero_int }
\int_gzero:c 8336 \cs_generate_variant:Nn \int_zero:N { c }
8337 \cs_generate_variant:Nn \int_gzero:N { c }

```

(End definition for `\int_zero:N` and `\int_gzero:N`. These functions are documented on page 91.)

`\int_zero_new:N` Create a register if needed, otherwise clear it.

```

\int_zero_new:c 8338 \cs_new_protected:Npn \int_zero_new:N #1
8339   { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
\int_gzero_new:N 8340 \cs_new_protected:Npn \int_gzero_new:N #1
8341   { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
8342 \cs_generate_variant:Nn \int_zero_new:N { c }
8343 \cs_generate_variant:Nn \int_gzero_new:N { c }

```

(End definition for `\int_zero_new:N` and `\int_gzero_new:N`. These functions are documented on page 91.)

\int_set_eq:NN Setting equal means using one integer inside the set function of another. Check that assigned integer is local/global. No need to check that the other one is defined as TeX does it for us.
\int_set_eq:cN
\int_set_eq:Nc
\int_set_eq:cc
\int_gset_eq:NN
\int_gset_eq:cN
\int_gset_eq:Nc
\int_gset_eq:cc

```

8344 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
8345 \cs_generate_variant:Nn \int_set_eq:NN { c , Nc , cc }
8346 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
8347 \cs_generate_variant:Nn \int_gset_eq:NN { c , Nc , cc }

```

(End definition for `\int_set_eq:NN` and `\int_gset_eq:NN`. These functions are documented on page 91.)

\int_if_exist_p:N Copies of the `cs` functions defined in `l3basics`.
\int_if_exist_p:c
\int_if_exist:NTF
\int_if_exist:cTF
\int_if_exist:pTF

```

8348 \prg_new_eq_conditional:Nnn \int_if_exist:N \cs_if_exist:N
8349   { TF , T , F , p }
8350 \prg_new_eq_conditional:Nnn \int_if_exist:c \cs_if_exist:c
8351   { TF , T , F , p }

```

(End definition for `\int_if_exist:NTF`. This function is documented on page 91.)

12.3 Setting and incrementing integers

\int_add:Nn Adding and subtracting to and from a counter.
\int_add:cN
\int_gadd:Nn
\int_gadd:cN
\int_sub:Nn
\int_sub:cN
\int_gsub:Nn
\int_gsub:cN

```

8352 \cs_new_protected:Npn \int_add:Nn #1#2
8353   { \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
8354 \cs_new_protected:Npn \int_sub:Nn #1#2
8355   { \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
8356 \cs_new_protected:Npn \int_gadd:Nn #1#2
8357   { \tex_global:D \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
8358 \cs_new_protected:Npn \int_gsub:Nn #1#2
8359   { \tex_global:D \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
8360 \cs_generate_variant:Nn \int_add:Nn { c }
8361 \cs_generate_variant:Nn \int_gadd:Nn { c }
8362 \cs_generate_variant:Nn \int_sub:Nn { c }
8363 \cs_generate_variant:Nn \int_gsub:Nn { c }

```

(End definition for `\int_add:Nn` and others. These functions are documented on page 91.)

\int_incr:N Incrementing and decrementing of integer registers is done with the following functions.
\int_incr:c
\int_gincr:N
\int_gincr:c
\int_decr:N
\int_decr:c
\int_gdecr:N
\int_gdecr:c

```

8364 \cs_new_protected:Npn \int_incr:N #1
8365   { \tex_advance:D #1 \c_one_int }
8366 \cs_new_protected:Npn \int_decr:N #1
8367   { \tex_advance:D #1 - \c_one_int }
8368 \cs_new_protected:Npn \int_gincr:N #1
8369   { \tex_global:D \tex_advance:D #1 \c_one_int }
8370 \cs_new_protected:Npn \int_gdecr:N #1
8371   { \tex_global:D \tex_advance:D #1 - \c_one_int }
8372 \cs_generate_variant:Nn \int_incr:N { c }
8373 \cs_generate_variant:Nn \int_decr:N { c }
8374 \cs_generate_variant:Nn \int_gincr:N { c }
8375 \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End definition for `\int_incr:N` and others. These functions are documented on page 91.)

```

\int_set:Nn As integers are register-based TEX issues an error if they are not defined.
\int_set:cn
\int_gset:Nn
\int_gset:cn
8376 \cs_new_protected:Npn \int_set:Nn #1#2
8377   { #1 ~ \_int_eval:w #2 \_int_eval_end: }
8378 \cs_new_protected:Npn \int_gset:Nn #1#2
8379   { \tex_global:D #1 ~ \_int_eval:w #2 \_int_eval_end: }
8380 \cs_generate_variant:Nn \int_set:Nn { c }
8381 \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End definition for `\int_set:Nn` and `\int_gset:Nn`. These functions are documented on page 92.)

12.4 Using integers

`\int_use:N` Here is how counters are accessed:

```
8382 \cs_new_eq:NN \int_use:N \tex_the:D
```

We hand-code this for some speed gain:

```
8383 \%cs_generate_variant:Nn \int_use:N { c }
8384 \cs_new:Npn \int_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(End definition for `\int_use:N`. This function is documented on page 92.)

12.5 Integer expression conditionals

```
\_int_compare_error:
\_int_compare_error:Nw
```

Those functions are used for comparison tests which use a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. The tests first evaluate their left-hand side, with a trailing `_int_compare_error:`. This marker is normally not expanded, but if the relation symbol is missing from the test's argument, then the marker inserts `=` (and itself) after triggering the relevant T_EX error. If the first token which appears after evaluating and removing the left-hand side is not a known relation symbol, then a judiciously placed `_int_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```
8385 \cs_new_protected:Npn \_int_compare_error:
8386   {
8387     \if_int_compare:w \c_zero_int \c_zero_int \fi:
8388     =
8389     \_int_compare_error:
8390   }
8391 \cs_new:Npn \_int_compare_error:Nw
8392   #1#2 \s_int_stop
8393   {
8394     {}
8395     \c_zero_int \fi:
8396     \kernel_msg_expandable_error:nnn
8397     { kernel } { unknown-comparison } {#1}
8398     \prg_return_false:
8399   }
```

(End definition for `_int_compare_error:` and `_int_compare_error:Nw`.)

```

\int_compare_p:n
\int_compare:nTF
\int_compare:w
\int_compare:Nw
\int_compare:NNw
\int_compare:nnN
\int_compare_end_=:NNw
\int_compare_=:NNw
\int_compare_<:NNw
\int_compare_>:NNw
\int_compare_==:NNw
\int_compare_!=:NNw
\int_compare_<=:NNw
\int_compare_>=:NNw

```

```

⟨operand⟩ \prg_return_false: \fi:
\reverse_if:N \if_int_compare:w ⟨operand⟩ ⟨comparison⟩
\__int_compare:Nw

```

in the input stream. Each call to this auxiliary provides the second operand of the last call's `\if_int_compare:w`. If one of the `⟨comparisons⟩` is `false`, the `true` branch of the TeX conditional is taken (because of `\reverse_if:N`), immediately returning `false` as the result of the test. There is no TeX conditional waiting the first operand, so we add an `\if_false:` and expand by hand with `\int_value:w`, thus skipping `\prg_return_false:` on the first iteration.

Before starting the loop, the first step is to make sure that there is at least one relation symbol. We first let TeX evaluate this left hand side of the (in)equality using `__int_eval:w`. Since the relation symbols `<`, `>`, `=` and `!` are not allowed in integer expressions, they would terminate the expression. If the argument contains no relation symbol, `__int_compare_error:` is expanded, inserting `=` and itself after an error. In all cases, `__int_compare:w` receives as its argument an integer, a relation symbol, and some more tokens. We then setup the loop, which is ended by the two odd-looking items `e` and `{=nd_}`, with a trailing `\s__int_stop` used to grab the entire argument when necessary.

```

8400 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
8401 {
8402   \exp_after:wN \__int_compare:w
8403   \int_value:w \__int_eval:w #1 \__int_compare_error:
8404 }
8405 \cs_new:Npn \__int_compare:w #1 \__int_compare_error:
8406 {
8407   \exp_after:wN \if_false: \int_value:w
8408   \__int_compare:Nw #1 e { = nd_ } \s__int_stop
8409 }

```

The goal here is to find an `⟨operand⟩` and a `⟨comparison⟩`. The `⟨operand⟩` is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying `__int_to_roman:w`, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with `__int_compare>NNw` to determine the relation symbol, building a control sequence from it (`\token_to_str:N` gives better errors if `#1` is not a character). All the extended forms have an extra `=` hence the test for that as a second token. If the relation symbol is unknown, then the control sequence is turned by TeX into `\scan_stop:`, ignored thanks to `\unexpanded`, and `__int_compare_error:Nw` raises an error.

```

8410 \cs_new:Npn \__int_compare:Nw #1#2 \s__int_stop
8411 {
8412   \exp_after:wN \__int_compare>NNw
8413   \__int_to_roman:w - 0 #2 \s__int_mark
8414   #1#2 \s__int_stop
8415 }
8416 \cs_new:Npn \__int_compare>NNw #1#2#3 \s__int_mark
8417 {
8418   \__kernel_exp_not:w
8419   \use:c
8420   {
8421     \__int_compare_ \token_to_str:N #1

```

```

8422     \if_meaning:w = #2 = \fi:
8423     :NNw
8424   }
8425   \_\_int_compare_error:Nw #1
8426 }

```

When the last $\langle operand \rangle$ is seen, `__int_compare:NNw` receives `e` and `=nd_` as arguments, hence calling `__int_compare_end_=:NNw` to end the loop: return the result of the last comparison (involving the operand that we just found). When a normal relation is found, the appropriate auxiliary calls `__int_compare:nnN` where `#1` is `\if_int_compare:w` or `\reverse_if:N \if_int_compare:w`, `#2` is the $\langle operand \rangle$, and `#3` is one of `<`, `=`, or `>`. As announced earlier, we leave the $\langle operand \rangle$ for the previous conditional. If this conditional is true the result of the test is known, so we remove all tokens and return `false`. Otherwise, we apply the conditional `#1` to the $\langle operand \rangle$ `#2` and the comparison `#3`, and call `__int_compare:Nw` to look for additional operands, after evaluating the following expression.

```

8427 \cs_new:cpn { \_\_int_compare_end_=:NNw } #1#2#3 e #4 \s_\_int_stop
8428 {
8429   {#3} \exp_stop_f:
8430   \prg_return_false: \else: \prg_return_true: \fi:
8431 }
8432 \cs_new:Npn \_\_int_compare:nnN #1#2#3
8433 {
8434   {#2} \exp_stop_f:
8435   \prg_return_false: \exp_after:wN \_\_int_use_none_delimit_by_s_stop:w
8436   \fi:
8437   #1 #2 #3 \exp_after:wN \_\_int_compare:Nw \int_value:w \_\_int_eval:w
8438 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `__int_compare_error:Nw` (*token*) responsible for error detection.

```

8439 \cs_new:cpn { \_\_int_compare_=:NNw } #1#2#3 =
8440   { \_\_int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
8441 \cs_new:cpn { \_\_int_compare_<:NNw } #1#2#3 <
8442   { \_\_int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
8443 \cs_new:cpn { \_\_int_compare_>:NNw } #1#2#3 >
8444   { \_\_int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
8445 \cs_new:cpn { \_\_int_compare_==:NNw } #1#2#3 ==
8446   { \_\_int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
8447 \cs_new:cpn { \_\_int_compare_!=:NNw } #1#2#3 !=
8448   { \_\_int_compare:nnN { \if_int_compare:w } {#3} = }
8449 \cs_new:cpn { \_\_int_compare_<=:NNw } #1#2#3 <=
8450   { \_\_int_compare:nnN { \if_int_compare:w } {#3} > }
8451 \cs_new:cpn { \_\_int_compare_>=:NNw } #1#2#3 >=
8452   { \_\_int_compare:nnN { \if_int_compare:w } {#3} < }

```

(End definition for `\int_compare:nTF` and others. This function is documented on page 93.)

`\int_compare_p:nNn` More efficient but less natural in typing.

```

\int_compare:nNnTF
8453 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
8454 {
8455   \if_int_compare:w \_\_int_eval:w #1 #2 \_\_int_eval:w #3 \_\_int_eval_end:
8456   \prg_return_true:

```

```

8457     \else:
8458         \prg_return_false:
8459     \fi:
8460 }

```

(End definition for `\int_compare:nNnTF`. This function is documented on page 92.)

```

\int_case:nn
\int_case:nnTF
\__int_case:nNTF
\__int_case:nw
\__int_case_end:nw
8461 \cs_new:Npn \int_case:nnTF #1
8462 {
8463     \exp:w
8464     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
8465 }
8466 \cs_new:Npn \int_case:nNT #1#2#3
8467 {
8468     \exp:w
8469     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
8470 }
8471 \cs_new:Npn \int_case:nnF #1#2
8472 {
8473     \exp:w
8474     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
8475 }
8476 \cs_new:Npn \int_case:nn #1#2
8477 {
8478     \exp:w
8479     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
8480 }
8481 \cs_new:Npn \__int_case:nnTF #1#2#3#4
8482 { \__int_case:nw {#1} #2 {#1} { } \s__int_mark {#3} \s__int_mark {#4} \s__int_stop }
8483 \cs_new:Npn \__int_case:nw #1#2#3
8484 {
8485     \int_compare:nNnTF {#1} = {#2}
8486     { \__int_case_end:nw {#3} }
8487     { \__int_case:nw {#1} }
8488 }
8489 \cs_new:Npn \__int_case_end:nw #1#2#3 \s__int_mark #4#5 \s__int_stop
8490 { \exp_end: #1 #4 }

```

(End definition for `\int_case:nnTF` and others. This function is documented on page 94.)

`\int_if_odd_p:n` A predicate function.

```

\int_if_odd:nTF
\int_if_even_p:n
\int_if_even:nTF
8491 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
8492 {
8493     \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
8494         \prg_return_true:
8495     \else:
8496         \prg_return_false:
8497     \fi:
8498 }
8499 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
8500 {
8501     \reverse_if:N \if_int_odd:w \__int_eval:w #1 \__int_eval_end:

```

```

8502     \prg_return_true:
8503     \else:
8504     \prg_return_false:
8505     \fi:
8506 }
```

(End definition for `\int_if_odd:nTF` and `\int_if_even:nTF`. These functions are documented on page 94.)

12.6 Integer expression loops

`\int_while_do:nn`
`\int_until_do:nn`

```

8507 \cs_new:Npn \int_while_do:nn #1#2
8508 {
8509     \int_compare:nT {#1}
8510     {
8511         #2
8512         \int_while_do:nn {#1} {#2}
8513     }
8514 }
8515 \cs_new:Npn \int_until_do:nn #1#2
8516 {
8517     \int_compare:nF {#1}
8518     {
8519         #2
8520         \int_until_do:nn {#1} {#2}
8521     }
8522 }
8523 \cs_new:Npn \int_do_while:nn #1#2
8524 {
8525     #2
8526     \int_compare:nT {#1}
8527     { \int_do_while:nn {#1} {#2} }
8528 }
8529 \cs_new:Npn \int_do_until:nn #1#2
8530 {
8531     #2
8532     \int_compare:nF {#1}
8533     { \int_do_until:nn {#1} {#2} }
8534 }
```

(End definition for `\int_while_do:nn` and others. These functions are documented on page 95.)

`\int_while_do:nNnn`
`\int_until_do:nNnn`
`\int_do_while:nNnn`
`\int_do_until:nNnn`

As above but not using the more natural syntax.

```

8535 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
8536 {
8537     \int_compare:nNnT {#1} #2 {#3}
8538     {
8539         #4
8540         \int_while_do:nNnn {#1} #2 {#3} {#4}
8541     }
8542 }
8543 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
```

```

8544    {
8545        \int_compare:nNnF {#1} #2 {#3}
8546        {
8547            #4
8548            \int_until_do:nNnn {#1} #2 {#3} {#4}
8549        }
8550    }
8551 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
8552    {
8553        #4
8554        \int_compare:nNnT {#1} #2 {#3}
8555        { \int_do_while:nNnn {#1} #2 {#3} {#4} }
8556    }
8557 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
8558    {
8559        #4
8560        \int_compare:nNnF {#1} #2 {#3}
8561        { \int_do_until:nNnn {#1} #2 {#3} {#4} }
8562    }

```

(End definition for `\int_while_do:nNnn` and others. These functions are documented on page 95.)

12.7 Integer step functions

```

\int_step_function:nnnN
  \__int_step:wwwN
  \__int_step:NwnnnN
\int_step_function:nN
\int_step_function:nnN

```

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

8563 \cs_new:Npn \int_step_function:nnnN #1#2#3
8564    {
8565        \exp_after:wN \__int_step:wwwN
8566        \int_value:w \__int_eval:w #1 \exp_after:wN ;
8567        \int_value:w \__int_eval:w #2 \exp_after:wN ;
8568        \int_value:w \__int_eval:w #3 ;
8569    }
8570 \cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
8571    {
8572        \int_compare:nNnTF {#2} > \c_zero_int
8573        { \__int_step:NwnnnN > }
8574        {
8575            \int_compare:nNnTF {#2} = \c_zero_int
8576            {
8577                \__kernel_msg_expandable_error:nnn
8578                { kernel } { zero-step } {#4}
8579                \prg_break:
8580            }
8581            { \__int_step:NwnnnN < }
8582        }
8583        #1 ; {#2} {#3} #4
8584        \prg_break_point:
8585    }
8586 \cs_new:Npn \__int_step:NwnnnN #1#2 ; #3#4#5
8587    {

```

```

8588     \if_int_compare:w #2 #1 #4 \exp_stop_f:
8589         \prg_break:n
8590     \fi:
8591     #5 {#2}
8592     \exp_after:wN \__int_step:NwnnN
8593     \exp_after:wN #1
8594     \int_value:w \__int_eval:w #2 + #3 ; {#3} {#4} #5
8595   }
8596   \cs_new:Npn \int_step_function:nN
8597   { \int_step_function:nnnN { 1 } { 1 } }
8598   \cs_new:Npn \int_step_function:nnN #1
8599   { \int_step_function:nnnN {#1} { 1 } }

```

(End definition for `\int_step_function:nnnN` and others. These functions are documented on page 96.)

```

\int_step_inline:nn
\int_step_inline:nnn
\int_step_inline:nnnn
\int_step_variable:nNn
\int_step_variable:nnNn
\int_step_variable:nnnNn
\int_step_variable:NNnnnn

```

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\int_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g_kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```

8600 \cs_new_protected:Npn \int_step_inline:nn
8601   { \int_step_inline:nnnn { 1 } { 1 } }
8602 \cs_new_protected:Npn \int_step_inline:nnn #1
8603   { \int_step_inline:nnnn {#1} { 1 } }
8604 \cs_new_protected:Npn \int_step_inline:nnnn
8605   {
8606     \int_gincr:N \g_kernel_prg_map_int
8607     \exp_args:NNc \__int_step:NNnnnn
8608     \cs_gset_protected:Npn
8609     { __int_map_ \int_use:N \g_kernel_prg_map_int :w }
8610   }
8611 \cs_new_protected:Npn \int_step_variable:nNn
8612   { \int_step_variable:nnnNn { 1 } { 1 } }
8613 \cs_new_protected:Npn \int_step_variable:nnNn #1
8614   { \int_step_variable:nnnNn {#1} { 1 } }
8615 \cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
8616   {
8617     \int_gincr:N \g_kernel_prg_map_int
8618     \exp_args:NNc \__int_step:NNnnnn
8619     \cs_gset_protected:Npx
8620     { __int_map_ \int_use:N \g_kernel_prg_map_int :w }
8621     {#1}{#2}{#3}
8622   {
8623     \tl_set:Nn \exp_not:N #4 {##1}
8624     \exp_not:n {#5}
8625   }
8626 }
8627 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
8628   {
8629     #1 #2 ##1 {#6}
8630     \int_step_function:nnnN {#3} {#4} {#5} #2
8631     \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g_kernel_prg_map_int }
8632   }

```

(End definition for `\int_step_inline:nn` and others. These functions are documented on page 96.)

12.8 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```
8633 \cs_new_eq:NN \int_to_arabic:n \int_eval:n
```

(End definition for `\int_to_arabic:n`. This function is documented on page 97.)

`\int_to_symbols:nnn` For conversion of integers to arbitrary symbols the method is in general as follows. The input number (#1) is compared to the total number of symbols available at each place (#2). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an f-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a ‘nice’ number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```
8634 \cs_new:Npn \int_to_symbols:nnn #1#2#3
8635 {
8636     \int_compare:nNnTF {#1} > {#2}
8637     {
8638         \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
8639         {
8640             \int_case:nn
8641                 { 1 + \int_mod:nn {#1 - 1} {#2} } {#3}
8642             {#3}
8643         }
8644         {#1} {#2} {#3}
8645     }
8646     { \int_case:nn {#1} {#3} }
8647 }
8648 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
8649 {
8650     \exp_args:Nf \int_to_symbols:nnn
8651     { \int_div_truncate:nn {#2 - 1} {#3} } {#3} {#4}
8652     #1
8653 }
```

(End definition for `\int_to_symbols:nnn` and `__int_to_symbols:nnnn`. This function is documented on page 97.)

`\int_to_alpha:n` These both use the above function with input functions that make sense for the alphabet
`\int_to_Alph:n` in English.

```
8654 \cs_new:Npn \int_to_alpha:n #1
8655 {
8656     \int_to_symbols:nnn {#1} { 26 }
8657     {
8658         { 1 } { a }
8659         { 2 } { b }
8660         { 3 } { c }
8661         { 4 } { d }
8662         { 5 } { e }
8663         { 6 } { f }
8664         { 7 } { g }
```

```

8665      { 8 } { h }
8666      { 9 } { i }
8667      { 10 } { j }
8668      { 11 } { k }
8669      { 12 } { l }
8670      { 13 } { m }
8671      { 14 } { n }
8672      { 15 } { o }
8673      { 16 } { p }
8674      { 17 } { q }
8675      { 18 } { r }
8676      { 19 } { s }
8677      { 20 } { t }
8678      { 21 } { u }
8679      { 22 } { v }
8680      { 23 } { w }
8681      { 24 } { x }
8682      { 25 } { y }
8683      { 26 } { z }
8684  }
8685  }
8686 \cs_new:Npn \int_to_Alph:n #1
8687  {
8688    \int_to_symbols:nnn {#1} { 26 }
8689  {
8690    { 1 } { A }
8691    { 2 } { B }
8692    { 3 } { C }
8693    { 4 } { D }
8694    { 5 } { E }
8695    { 6 } { F }
8696    { 7 } { G }
8697    { 8 } { H }
8698    { 9 } { I }
8699    { 10 } { J }
8700    { 11 } { K }
8701    { 12 } { L }
8702    { 13 } { M }
8703    { 14 } { N }
8704    { 15 } { O }
8705    { 16 } { P }
8706    { 17 } { Q }
8707    { 18 } { R }
8708    { 19 } { S }
8709    { 20 } { T }
8710    { 21 } { U }
8711    { 22 } { V }
8712    { 23 } { W }
8713    { 24 } { X }
8714    { 25 } { Y }
8715    { 26 } { Z }
8716  }
8717  }

```

(End definition for `\int_to_alpha:n` and `\int_to_Alph:n`. These functions are documented on page 97.)

```

\int_to_base:nn
\int_to_Base:nn
\_\_int_to_base:nn
\_\_int_to_Base:nnN
\_\_int_to_base:nnN
\_\_int_to_base:nnnN
\_\_int_to_Base:nnnN
\_\_int_to_letter:n
\_\_int_to_Letter:n

```

Converting from base ten (#1) to a second base (#2) starts with computing #1: if it is a complicated calculation, we shouldn't perform it twice. Then check the sign, store it, either - or \c_empty_t1, and feed the absolute value to the next auxiliary function.

```

8718 \cs_new:Npn \int_to_base:nn #1
8719   { \exp_args:Nf \_\_int_to_base:nn { \int_eval:n {#1} } }
8720 \cs_new:Npn \int_to_Base:nn #1
8721   { \exp_args:Nf \_\_int_to_Base:nn { \int_eval:n {#1} } }
8722 \cs_new:Npn \_\_int_to_base:nn #1#2
8723   {
8724     \int_compare:nNnTF {#1} < 0
8725       { \exp_args:No \_\_int_to_base:nnN { \use_none:n #1 } {#2} - }
8726       { \_\_int_to_base:nnN {#1} {#2} \c_empty_t1 }
8727   }
8728 \cs_new:Npn \_\_int_to_Base:nn #1#2
8729   {
8730     \int_compare:nNnTF {#1} < 0
8731       { \exp_args:No \_\_int_to_Base:nnN { \use_none:n #1 } {#2} - }
8732       { \_\_int_to_Base:nnN {#1} {#2} \c_empty_t1 }
8733   }

```

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in #1 is checked to see if it is less than the new base (#2). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.

```

8734 \cs_new:Npn \_\_int_to_base:nnN #1#2#3
8735   {
8736     \int_compare:nNnTF {#1} < {#2}
8737       { \exp_last_unbraced:Nf #3 { \_\_int_to_letter:n {#1} } }
8738   {
8739     \exp_args:Nf \_\_int_to_base:nnnN
8740       { \_\_int_to_letter:n { \int_mod:nn {#1} {#2} } }
8741       {#1}
8742       {#2}
8743       #3
8744   }
8745   }
8746 \cs_new:Npn \_\_int_to_base:nnnN #1#2#3#4
8747   {
8748     \exp_args:Nf \_\_int_to_base:nnN
8749       { \int_div_truncate:nn {#2} {#3} }
8750       {#3}
8751       #4
8752       #1
8753   }
8754 \cs_new:Npn \_\_int_to_Base:nnN #1#2#3
8755   {
8756     \int_compare:nNnTF {#1} < {#2}
8757       { \exp_last_unbraced:Nf #3 { \_\_int_to_Letter:n {#1} } }
8758   {
8759     \exp_args:Nf \_\_int_to_Base:nnnN
8760       { \_\_int_to_Letter:n { \int_mod:nn {#1} {#2} } }
8761       {#1}

```

```

8762          {#2}
8763          #3
8764      }
8765  }
8766 \cs_new:Npn \__int_to_Base:nnN #1#2#3#4
8767  {
8768      \exp_args:Nf \__int_to_Base:nnN
8769      { \int_div_truncate:nn {#2} {#3} }
8770      {#3}
8771      #4
8772      #1
8773  }

```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use `\int_case:nn`, but in our case, the cases are contiguous, so it is forty times faster to use the `\if_case:w` primitive. The first `\exp_after:wN` expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional. Since `#1` might be an expression, and not directly a single digit, we need to evaluate it properly, and expand the trailing `\fi:`.

```

8774 \cs_new:Npn \__int_to_letter:n #1
8775  {
8776      \exp_after:wN \exp_after:wN
8777      \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
8778          a
8779          \or: b
8780          \or: c
8781          \or: d
8782          \or: e
8783          \or: f
8784          \or: g
8785          \or: h
8786          \or: i
8787          \or: j
8788          \or: k
8789          \or: l
8790          \or: m
8791          \or: n
8792          \or: o
8793          \or: p
8794          \or: q
8795          \or: r
8796          \or: s
8797          \or: t
8798          \or: u
8799          \or: v
8800          \or: w
8801          \or: x
8802          \or: y
8803          \or: z
8804      \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
8805      \fi:
8806  }
8807 \cs_new:Npn \__int_to_Letter:n #1
8808  {

```

```

8809      \exp_after:wN \exp_after:wN
8810      \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
8811          A
8812          \or: B
8813          \or: C
8814          \or: D
8815          \or: E
8816          \or: F
8817          \or: G
8818          \or: H
8819          \or: I
8820          \or: J
8821          \or: K
8822          \or: L
8823          \or: M
8824          \or: N
8825          \or: O
8826          \or: P
8827          \or: Q
8828          \or: R
8829          \or: S
8830          \or: T
8831          \or: U
8832          \or: V
8833          \or: W
8834          \or: X
8835          \or: Y
8836          \or: Z
8837      \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
8838      \fi:
8839  }

```

(End definition for `\int_to_base:nn` and others. These functions are documented on page 98.)

`\int_to_bin:n` Wrappers around the generic function.

```

\int_to_hex:n 8840 \cs_new:Npn \int_to_bin:n #1
\int_to_Hex:n 8841  { \int_to_base:nn {#1} { 2 } }
\int_to_oct:n 8842 \cs_new:Npn \int_to_hex:n #1
8843  { \int_to_base:nn {#1} { 16 } }
8844 \cs_new:Npn \int_to_Hex:n #1
8845  { \int_to_Base:nn {#1} { 16 } }
8846 \cs_new:Npn \int_to_oct:n #1
8847  { \int_to_base:nn {#1} { 8 } }

```

(End definition for `\int_to_bin:n` and others. These functions are documented on page 98.)

`\int_to_roman:n` The `__int_to_roman:w` primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop is terminated by the conversion of the `Q`.

```

\__int_to_roman:N 8848 \cs_new:Npn \int_to_roman:n #1
\__int_to_roman:N 8849  {
\__int_to_roman_x:w 8850  \exp_after:wN \__int_to_roman:N
\__int_to_roman_l:w 8851    \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_roman_c:w
\__int_to_roman_d:w
\__int_to_roman_m:w
\__int_to_roman_Q:w
\__int_to_Roman_i:w
\__int_to_Roman_v:w
\__int_to_Roman_x:w
\__int_to_Roman_l:w
\__int_to_Roman_c:w
\__int_to_Roman_d:w

```

```

8852    }
8853 \cs_new:Npn \__int_to_roman:N #1
8854 {
8855     \use:c { __int_to_roman_ #1 :w }
8856     \__int_to_roman:N
8857 }
8858 \cs_new:Npn \int_to_Roman:n #1
8859 {
8860     \exp_after:wN \__int_to_Roman_aux:N
8861     \__int_to_roman:w \int_eval:n {#1} Q
8862 }
8863 \cs_new:Npn \__int_to_Roman_aux:N #1
8864 {
8865     \use:c { __int_to_Roman_ #1 :w }
8866     \__int_to_Roman_aux:N
8867 }
8868 \cs_new:Npn \__int_to_roman_i:w { i }
8869 \cs_new:Npn \__int_to_roman_v:w { v }
8870 \cs_new:Npn \__int_to_roman_x:w { x }
8871 \cs_new:Npn \__int_to_roman_l:w { l }
8872 \cs_new:Npn \__int_to_roman_c:w { c }
8873 \cs_new:Npn \__int_to_roman_d:w { d }
8874 \cs_new:Npn \__int_to_roman_m:w { m }
8875 \cs_new:Npn \__int_to_roman_Q:w #1 { }
8876 \cs_new:Npn \__int_to_Roman_i:w { I }
8877 \cs_new:Npn \__int_to_Roman_v:w { V }
8878 \cs_new:Npn \__int_to_Roman_x:w { X }
8879 \cs_new:Npn \__int_to_Roman_l:w { L }
8880 \cs_new:Npn \__int_to_Roman_c:w { C }
8881 \cs_new:Npn \__int_to_Roman_d:w { D }
8882 \cs_new:Npn \__int_to_Roman_m:w { M }
8883 \cs_new:Npn \__int_to_Roman_Q:w #1 { }

```

(End definition for `\int_to_Roman:n` and others. These functions are documented on page 98.)

12.9 Converting from other formats to integers

`__int_pass_signs:wn` Called as `__int_pass_signs:wn <signs and digits> \s__int_stop {<code>}`, this function leaves in the input stream any sign it finds, then inserts the `<code>` before the first non-sign token (and removes `\s__int_stop`). More precisely, it deletes any + and passes any - to the input stream, hence should be called in an integer expression.

```

8884 \cs_new:Npn \__int_pass_signs:wn #1
8885 {
8886     \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
8887     \exp_after:wN \__int_pass_signs:wn
8888     \else:
8889     \exp_after:wN \__int_pass_signs_end:wn
8890     \exp_after:wN #1
8891     \fi:
8892 }
8893 \cs_new:Npn \__int_pass_signs_end:wn #1 \s__int_stop #2 { #2 #1 }

```

(End definition for `__int_pass_signs:wn` and `__int_pass_signs_end:wn`.)

\int_from_alpha:n First take care of signs then loop through the input using the **recursion quarks**. The **__int_from_alpha:nN** auxiliary collects in its first argument the value obtained so far, and the auxiliary **__int_from_alpha:N** converts one letter to an expression which evaluates to the correct number.

```

8894 \cs_new:Npn \int_from_alpha:n #1
8895   {
8896     \int_eval:n
8897     {
8898       \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
8899       \s__int_stop { \__int_from_alpha:nN { 0 } }
8900       \q__int_recursion_tail \q__int_recursion_stop
8901     }
8902   }
8903 \cs_new:Npn \__int_from_alpha:nN #1#2
8904   {
8905     \__int_if_recursion_tail_stop_do:Nn #2 {#1}
8906     \exp_args:Nf \__int_from_alpha:nN
8907     { \int_eval:n { #1 * 26 + \__int_from_alpha:N #2 } }
8908   }
8909 \cs_new:Npn \__int_from_alpha:N #1
8910   { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }
```

(End definition for **\int_from_alpha:n**, **__int_from_alpha:nN**, and **__int_from_alpha:N**. This function is documented on page 98.)

\int_from_base:nn Leave the signs into the integer expression, then loop through characters, collecting the value found so far in the first argument of **__int_from_base:nnN**. To convert a single character, **__int_from_base:N** checks first for digits, then distinguishes lower from upper case letters, turning them into the appropriate number. Note that this auxiliary does not use **\int_eval:n**, hence is not safe for general use.

```

8911 \cs_new:Npn \int_from_base:nn #1#2
8912   {
8913     \int_eval:n
8914     {
8915       \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
8916       \s__int_stop { \__int_from_base:nnN { 0 } {#2} }
8917       \q__int_recursion_tail \q__int_recursion_stop
8918     }
8919   }
8920 \cs_new:Npn \__int_from_base:nnN #1#2#3
8921   {
8922     \__int_if_recursion_tail_stop_do:Nn #3 {#1}
8923     \exp_args:Nf \__int_from_base:nnN
8924     { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
8925     {#2}
8926   }
8927 \cs_new:Npn \__int_from_base:N #1
8928   {
8929     \int_compare:nNnTF { '#1 } < { 58 }
8930     { '#1 }
8931     { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
8932   }
```

(End definition for **\int_from_base:nn**, **__int_from_base:nnN**, and **__int_from_base:N**. This function is documented on page 99.)

```
\int_from_bin:n Wrappers around the generic function.
\int_from_hex:n
\int_from_oct:n
8933 \cs_new:Npn \int_from_bin:n #1
8934   { \int_from_base:nn {#1} { 2 } }
8935 \cs_new:Npn \int_from_hex:n #1
8936   { \int_from_base:nn {#1} { 16 } }
8937 \cs_new:Npn \int_from_oct:n #1
8938   { \int_from_base:nn {#1} { 8 } }
```

(End definition for `\int_from_bin:n`, `\int_from_hex:n`, and `\int_from_oct:n`. These functions are documented on page 99.)

`\c_int_from_roman_i_int` Constants used to convert from Roman numerals to integers.

```
\c_int_from_roman_v_int
\c_int_from_roman_x_int
\c_int_from_roman_l_int
\c_int_from_roman_c_int
\c_int_from_roman_d_int
\c_int_from_roman_m_int
\c_int_from_roman_I_int
\c_int_from_roman_V_int
\c_int_from_roman_X_int
\c_int_from_roman_L_int
\c_int_from_roman_C_int
\c_int_from_roman_D_int
\c_int_from_roman_M_int
8939 \int_const:cn { \c_int_from_roman_i_int } { 1 }
8940 \int_const:cn { \c_int_from_roman_v_int } { 5 }
8941 \int_const:cn { \c_int_from_roman_x_int } { 10 }
8942 \int_const:cn { \c_int_from_roman_l_int } { 50 }
8943 \int_const:cn { \c_int_from_roman_c_int } { 100 }
8944 \int_const:cn { \c_int_from_roman_d_int } { 500 }
8945 \int_const:cn { \c_int_from_roman_m_int } { 1000 }
8946 \int_const:cn { \c_int_from_roman_I_int } { 1 }
8947 \int_const:cn { \c_int_from_roman_V_int } { 5 }
8948 \int_const:cn { \c_int_from_roman_X_int } { 10 }
8949 \int_const:cn { \c_int_from_roman_L_int } { 50 }
8950 \int_const:cn { \c_int_from_roman_C_int } { 100 }
8951 \int_const:cn { \c_int_from_roman_D_int } { 500 }
8952 \int_const:cn { \c_int_from_roman_M_int } { 1000 }
```

(End definition for `\c_int_from_roman_i_int` and others.)

`\int_from_roman:n` The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by TeX. If any unknown letter is found, skip to the closing parenthesis and insert *0-1 afterwards, to replace the value by -1.

```
8953 \cs_new:Npn \int_from_roman:n #1
8954   {
8955     \int_eval:n
8956       {
8957         (
8958           0
8959           \exp_after:wN \__int_from_roman>NN \tl_to_str:n {#1}
8960           \q__int_recursion_tail \q__int_recursion_tail \q__int_recursion_stop
8961         )
8962       }
8963     }
8964 \cs_new:Npn \__int_from_roman>NN #1#
8965   {
8966     \__int_if_recursion_tail_stop:N #1
8967     \int_if_exist:cF { \c_int_from_roman_ #1 _int }
8968       { \__int_from_roman_error:w }
8969     \__int_if_recursion_tail_stop_do:Nn #2
8970       { + \use:c { \c_int_from_roman_ #1 _int } }
8971     \int_if_exist:cF { \c_int_from_roman_ #2 _int }
8972       { \__int_from_roman_error:w }
8973     \int_compare:nNnTF
```

```

8974 { \use:c { c__int_from_roman_ #1 _int } }
8975 <
8976 { \use:c { c__int_from_roman_ #2 _int } }
8977 {
8978     + \use:c { c__int_from_roman_ #2 _int }
8979     - \use:c { c__int_from_roman_ #1 _int }
8980     \_\_int_from_roman:NN
8981 }
8982 {
8983     + \use:c { c__int_from_roman_ #1 _int }
8984     \_\_int_from_roman:NN #2
8985 }
8986 }
8987 \cs_new:Npn \_\_int_from_roman_error:w #1 \q__int_recursion_stop #2
8988 { #2 * 0 - 1 }

```

(End definition for `\int_from_roman:n`, `__int_from_roman:NN`, and `__int_from_roman_error:w`. This function is documented on page 99.)

12.10 Viewing integer

\int_show:N Diagnostics.

```

8989 \cs_new_eq:NN \int_show:N \_\_kernel_register_show:N
8990 \cs_generate_variant:Nn \int_show:N { c }

```

(End definition for `\int_show:N` and `__int_show:nN`. This function is documented on page 100.)

\int_show:n We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output.

```

8991 \cs_new_protected:Npn \int_show:n
8992 { \msg_show_eval:Nn \int_eval:n }

```

(End definition for `\int_show:n`. This function is documented on page 100.)

\int_log:N Diagnostics.

```

8993 \cs_new_eq:NN \int_log:N \_\_kernel_register_log:N
8994 \cs_generate_variant:Nn \int_log:N { c }

```

(End definition for `\int_log:N`. This function is documented on page 100.)

\int_log:n Similar to `\int_show:n`.

```

8995 \cs_new_protected:Npn \int_log:n
8996 { \msg_log_eval:Nn \int_eval:n }

```

(End definition for `\int_log:n`. This function is documented on page 100.)

12.11 Random integers

\int_rand:nn Defined in `!3fp-random`.

(End definition for `\int_rand:nn`. This function is documented on page 99.)

12.12 Constant integers

`\c_zero_int` The zero is defined in l3basics.

`\c_one_int` 8997 `\int_const:Nn \c_one_int { 1 }`

(End definition for `\c_zero_int` and `\c_one_int`. These variables are documented on page 100.)

`\c_max_int` The largest number allowed is $2^{31} - 1$

8998 `\int_const:Nn \c_max_int { 2 147 483 647 }`

(End definition for `\c_max_int`. This variable is documented on page 100.)

`\c_max_char_int` The largest character code is 1114111 (hexadecimal 10FFFF) in X_ET_EX and LuaT_EX and 255 in other engines. In many places pT_EX and upT_EX support larger character codes but for instance the values of `\lccode` are restricted to [0, 255].

```
8999 \int_const:Nn \c_max_char_int
9000 {
9001   \if_int_odd:w 0
9002     \cs_if_exist:NT \tex_luatexversion:D { 1 }
9003     \cs_if_exist:NT \tex_XeTeXversion:D { 1 } ~
9004     "10FFFF
9005   \else:
9006     "FF
9007   \fi:
9008 }
```

(End definition for `\c_max_char_int`. This variable is documented on page 100.)

12.13 Scratch integers

`\l_tmpa_int` We provide two local and two global scratch counters, maybe we need more or less.

`\l_tmpb_int` 9009 `\int_new:N \l_tmpa_int`

`\g_tmpa_int` 9010 `\int_new:N \l_tmpb_int`

`\g_tmpb_int` 9011 `\int_new:N \g_tmpa_int`

9012 `\int_new:N \g_tmpb_int`

(End definition for `\l_tmpa_int` and others. These variables are documented on page 100.)

12.14 Integers for earlier modules

<@@=seq>

`\l__int_internal_a_int`

`\l__int_internal_b_int`

9013 `\int_new:N \l__int_internal_a_int`

9014 `\int_new:N \l__int_internal_b_int`

(End definition for `\l__int_internal_a_int` and `\l__int_internal_b_int`.)

9015 `</package>`

13 l3flag implementation

```
9016  {*package}  
9017  <@=flag>
```

The following test files are used for this code: *m3flag001*.

13.1 Non-expandable flag commands

The height h of a flag (initially zero) is stored by setting control sequences of the form $\text{\flag } \langle\text{name}\rangle \langle\text{integer}\rangle$ to \relax for $0 \leq \langle\text{integer}\rangle < h$. When a flag is raised, a “trap” function $\text{\flag } \langle\text{name}\rangle$ is called. The existence of this function is also used to test for the existence of a flag.

- \flag_new:n** For each flag, we define a “trap” function, which by default simply increases the flag by 1 by letting the appropriate control sequence to \relax . This can be done expandably!

```
9018  \cs_new_protected:Npn \flag_new:n #1  
9019  {  
9020  \cs_new:cpn { flag~#1 } ##1 ;  
9021  { \exp_after:wN \use_none:n \cs:w flag~#1~##1 \cs_end: }  
9022  }
```

(End definition for \flag_new:n . This function is documented on page 103.)

- \flag_clear:n** $__ \text{\flag_clear:wn}$ Undefine control sequences, starting from the 0 flag, upwards, until reaching an undefined control sequence. We don’t use \cs_undefine:c because that would act globally. When the option `check-declarations` is used, check for the function defined by \flag_new:n .

```
9023  \cs_new_protected:Npn \flag_clear:n #1 { \_\_ \text{\flag_clear:wn} 0 ; {#1} }  
9024  \cs_new_protected:Npn \_\_ \text{\flag_clear:wn} #1 ; #2  
9025  {  
9026  \if_cs_exist:w flag~#2~#1 \cs_end:  
9027  \cs_set_eq:cN { flag~#2~#1 } \tex_undefined:D  
9028  \exp_after:wN \_\_ \text{\flag_clear:wn}  
9029  \int_value:w \int_eval:w 1 + #1  
9030  \else:  
9031  \use_i:nnn  
9032  \fi:  
9033  ; {#2}  
9034  }
```

(End definition for \flag_clear:n and $__ \text{\flag_clear:wn}$. This function is documented on page 103.)

- \flag_clear_new:n** As for other datatypes, clear the $\langle\text{flag}\rangle$ or create a new one, as appropriate.

```
9035  \cs_new_protected:Npn \flag_clear_new:n #1  
9036  { \flag_if_exist:nTF {#1} { \flag_clear:n } { \flag_new:n } {#1} }
```

(End definition for \flag_clear_new:n . This function is documented on page 103.)

- \flag_show:n** Show the height (terminal or log file) using appropriate l3msg auxiliaries.
\flag_log:n
 $__ \text{\flag_show:Nn}$
- ```
9037 \cs_new_protected:Npn \flag_show:n { __ \text{\flag_show:Nn} \tl_show:n }
9038 \cs_new_protected:Npn \flag_log:n { __ \text{\flag_show:Nn} \tl_log:n }
9039 \cs_new_protected:Npn __ \text{\flag_show:Nn} #1#2
9040 {
9041 \exp_args:Nc __ kernel_chk_defined:NT { flag~#2 }
9042 {
```

```

9043 \exp_args:Nx #1
9044 { \tl_to_str:n { flag~#2~height } = \flag_height:n {#2} }
9045 }
9046 }

```

(End definition for `\flag_show:n`, `\flag_log:n`, and `\_flag_show:Nn`. These functions are documented on page 103.)

### 13.2 Expandable flag commands

`\flag_if_exist:p:n` A flag exist if the corresponding trap `\flag <flag name>:n` is defined.

```

9047 \prg_new_conditional:Npnn \flag_if_exist:n #1 { p , T , F , TF }
9048 {
9049 \cs_if_exist:cTF { flag~#1 }
9050 { \prg_return_true: } { \prg_return_false: }
9051 }

```

(End definition for `\flag_if_exist:nTF`. This function is documented on page 104.)

`\flag_if_raised:p:n` Test if the flag has a non-zero height, by checking the 0 control sequence.

```

9052 \prg_new_conditional:Npnn \flag_if_raised:n #1 { p , T , F , TF }
9053 {
9054 \if_cs_exist:w flag~#1~0 \cs_end:
9055 \prg_return_true:
9056 \else:
9057 \prg_return_false:
9058 \fi:
9059 }

```

(End definition for `\flag_if_raised:nTF`. This function is documented on page 104.)

`\flag_height:n` Extract the value of the flag by going through all of the control sequences starting from 0.

```

9060 \cs_new:Npn \flag_height:n #1 { _flag_height_loop:wn 0; {#1} }
9061 \cs_new:Npn _flag_height_loop:wn #1 ; #2
9062 {
9063 \if_cs_exist:w flag~#2~#1 \cs_end:
9064 \exp_after:wN _flag_height_loop:wn \int_value:w \int_eval:w 1 +
9065 \else:
9066 \exp_after:wN _flag_height_end:wn
9067 \fi:
9068 #1 ; {#2}
9069 }
9070 \cs_new:Npn _flag_height_end:wn #1 ; #2 {#1}

```

(End definition for `\flag_height:n`, `\_flag_height_loop:wn`, and `\_flag_height_end:wn`. This function is documented on page 104.)

`\flag_raise:n` Simply apply the trap to the height, after expanding the latter.

```

9071 \cs_new:Npn \flag_raise:n #1
9072 {
9073 \cs:w flag~#1 \exp_after:wN \cs_end:
9074 \int_value:w \flag_height:n {#1} ;
9075 }

```

(End definition for `\flag_raise:n`. This function is documented on page 104.)

9076

## 14 l3prg implementation

The following test files are used for this code: *m3prg001.lvt, m3prg002.lvt, m3prg003.lvt*.

9077 `(*package)`

### 14.1 Primitive conditionals

`\if_bool:N  
\if_predicate:w`

Those two primitive TeX conditionals are synonyms. `\if_bool:N` is defined in *l3basics*, as it's needed earlier to define quark test functions.

9078 `\cs_new_eq:NN \if_predicate:w \tex_ifodd:D`

(End definition for `\if_bool:N` and `\if_predicate:w`. These functions are documented on page 113.)

### 14.2 Defining a set of conditional functions

These are all defined in *l3basics*, as they are needed “early”. This is just a reminder!

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 105.)

### 14.3 The boolean data type

9079 `(@@=bool)`

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

9080 `\cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }`  
9081 `\cs_generate_variant:Nn \bool_new:N { c }`

(End definition for `\bool_new:N`. This function is documented on page 107.)

`\bool_const:Nn`

`\bool_const:cn`

A merger between `\tl_const:Nn` and `\bool_set:Nn`.

9082 `\cs_new_protected:Npn \bool_const:Nn #1#2`  
9083 `{`  
9084 `\__kernel_chk_if_free_cs:N #1`  
9085 `\tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}`  
9086 `}`  
9087 `\cs_generate_variant:Nn \bool_const:Nn { c }`

(End definition for `\bool_const:Nn`. This function is documented on page 108.)

`\bool_set_true:N`

`\bool_set_true:c`

`\bool_gset_true:N`

`\bool_gset_true:c`

`\bool_set_false:N`

`\bool_set_false:c`

`\bool_gset_false:N`

`\bool_gset_false:c`

Setting is already pretty easy. When `check-declarations` is active, the definitions are patched to make sure the boolean exists. This is needed because booleans are not based on token lists nor on TeX registers.

9088 `\cs_new_protected:Npn \bool_set_true:N #1`  
9089 `{ \cs_set_eq:NN #1 \c_true_bool }`  
9090 `\cs_new_protected:Npn \bool_set_false:N #1`  
9091 `{ \cs_set_eq:NN #1 \c_false_bool }`  
9092 `\cs_new_protected:Npn \bool_gset_true:N #1`  
9093 `{ \cs_gset_eq:NN #1 \c_true_bool }`  
9094 `\cs_new_protected:Npn \bool_gset_false:N #1`  
9095 `{ \cs_gset_eq:NN #1 \c_false_bool }`  
9096 `\cs_generate_variant:Nn \bool_set_true:N { c }`  
9097 `\cs_generate_variant:Nn \bool_set_false:N { c }`  
9098 `\cs_generate_variant:Nn \bool_gset_true:N { c }`  
9099 `\cs_generate_variant:Nn \bool_gset_false:N { c }`

(End definition for `\bool_set_true:N` and others. These functions are documented on page 108.)

`\bool_set_eq:NN` The usual copy code. While it would be cleaner semantically to copy the `\cs_set_eq:NN` family of functions, we copy `\tl_set_eq:NN` because that has the correct checking code.  
`\bool_set_eq:cN`  
`\bool_set_eq:Nc`  
`\bool_set_eq:cc`  
`\bool_gset_eq:NN`  
`\bool_gset_eq:cN`  
`\bool_gset_eq:Nc`  
`\bool_gset_eq:cc`

```
9100 \cs_new_eq:NN \bool_set_eq:NN \tl_set_eq:NN
9101 \cs_new_eq:NN \bool_gset_eq:NN \tl_gset_eq:NN
9102 \cs_generate_variant:Nn \bool_set_eq:NN { Nc, cN, cc }
9103 \cs_generate_variant:Nn \bool_gset_eq:NN { Nc, cN, cc }
```

(End definition for `\bool_set_eq:NN` and `\bool_gset_eq:NN`. These functions are documented on page 108.)

`\bool_set:Nn` This function evaluates a boolean expression and assigns the first argument the meaning `\c_true_bool` or `\c_false_bool`. Again, we include some checking code. It is important to evaluate the expression before applying the `\chardef` primitive, because that primitive sets the left-hand side to `\scan_stop:` before looking for the right-hand side.  
`\bool_set:cn`  
`\bool_gset:Nn`  
`\bool_gset:cn`

```
9104 \cs_new_protected:Npn \bool_set:Nn #1#2
9105 {
9106 \exp_last_unbraced:NNNf
9107 \tex_chardef:D #1 = { \bool_if_p:n {#2} }
9108 }
9109 \cs_new_protected:Npn \bool_gset:Nn #1#2
9110 {
9111 \exp_last_unbraced:NNNNf
9112 \tex_global:D \tex_chardef:D #1 = { \bool_if_p:n {#2} }
9113 }
9114 \cs_generate_variant:Nn \bool_set:Nn { c }
9115 \cs_generate_variant:Nn \bool_gset:Nn { c }
```

(End definition for `\bool_set:Nn` and `\bool_gset:Nn`. These functions are documented on page 108.)

#### 14.4 Internal auxiliaries

`\q_bool_recursion_tail` Internal recursion quarks.

```
9116 \quark_new:N \q_bool_recursion_tail
9117 \quark_new:N \q_bool_recursion_stop
```

(End definition for `\q_bool_recursion_tail` and `\q_bool_recursion_stop`.)

`\__bool_use_i_delimit_by_q_recursion_stop:nw` Functions to gobble up to a quark.

```
9118 \cs_new:Npn __bool_use_i_delimit_by_q_recursion_stop:nw
9119 #1 #2 \q_bool_recursion_stop {#1}
```

(End definition for `\__bool_use_i_delimit_by_q_recursion_stop:nw`.)

`\__bool_if_recursion_tail_stop_do:nn` Functions to query recursion quarks.

```
9120 __kernel_quark_new_test:N __bool_if_recursion_tail_stop_do:nn
```

(End definition for `\__bool_if_recursion_tail_stop_do:nn`.)

**\bool\_if\_p:N** Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```
9121 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
9122 {
9123 \if_bool:N #1
9124 \prg_return_true:
9125 \else:
9126 \prg_return_false:
9127 \fi:
9128 }
9129 \prg_generate_conditional_variant:Nnn \bool_if:N { c } { p , T , F , TF }
```

(End definition for **\bool\_if:NTF**. This function is documented on page 108.)

**\bool\_show:n** Show the truth value of the boolean, as **true** or **false**.

```
9130 \cs_new_protected:Npn \bool_show:n
9131 { \msg_show_eval:Nn __bool_to_str:n }
9132 \cs_new_protected:Npn \bool_log:n
9133 { \msg_log_eval:Nn __bool_to_str:n }
9134 \cs_new:Npn __bool_to_str:n #1
9135 { \bool_if:nTF {#1} { true } { false } }
```

(End definition for **\bool\_show:n**, **\bool\_log:n**, and **\\_\_bool\_to\_str:n**. These functions are documented on page 108.)

**\bool\_show:N** Show the truth value of the boolean, as **true** or **false**.

```
9136 \cs_new_protected:Npn \bool_show:N { __bool_show>NN \tl_show:n }
9137 \cs_generate_variant:Nn \bool_show:N { c }
9138 \cs_new_protected:Npn \bool_log:N { __bool_show>NN \tl_log:n }
9139 \cs_generate_variant:Nn \bool_log:N { c }
9140 \cs_new_protected:Npn __bool_show>NN #1#2
9141 {
9142 __kernel_chk_defined:NT #2
9143 { \exp_args:Nx #1 { \token_to_str:N #2 = __bool_to_str:n {#2} } }
9144 }
```

(End definition for **\bool\_show:N**, **\bool\_log:N**, and **\\_\_bool\_show>NN**. These functions are documented on page 108.)

**\l\_tmpa\_bool** A few booleans just if you need them.

```
9145 \bool_new:N \l_tmpa_bool
9146 \bool_new:N \l_tmpb_bool
9147 \bool_new:N \g_tmpa_bool
9148 \bool_new:N \g_tmpb_bool
```

(End definition for **\l\_tmpa\_bool** and others. These variables are documented on page 109.)

**\bool\_if\_exist\_p:N** Copies of the **cs** functions defined in l3basics.

```
9149 \prg_new_eq_conditional>NNn \bool_if_exist:N \cs_if_exist:N
9150 { TF , T , F , p }
9151 \prg_new_eq_conditional>NNn \bool_if_exist:c \cs_if_exist:c
9152 { TF , T , F , p }
```

(End definition for **\bool\_if\_exist:NTF**. This function is documented on page 109.)

## 14.5 Boolean expressions

`\bool_if_p:n` Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with ( and ) for grouping, ! for logical “Not”, && for logical “And” and || for logical “Or”. However, they perform eager evaluation. We shall use the terms Not, And, Or, Open and Close for these operations.

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a GetNext function:

- If an Open is seen, start evaluating a new expression using the Eval function and call GetNext again.
- If a Not is seen, remove the ! and call a GetNext function with the logic reversed.
- If none of the above, reinsert the token found (this is supposed to be a predicate function) in front of an Eval function, which evaluates it to the boolean value `<true>` or `<false>`.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

**`<true>And`** Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

**`<false>And`** Current truth value is false, logical And seen, stop using the values of predicates within this sub-expression until the next Close. Then return `<false>`.

**`<true>Or`** Current truth value is true, logical Or seen, stop using the values of predicates within this sub-expression until the nearest Close. Then return `<true>`.

**`<false>Or`** Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

**`<true>Close`** Current truth value is true, Close seen, return `<true>`.

**`<false>Close`** Current truth value is false, Close seen, return `<false>`.

```

9153 \prg_new_conditional:Npn \bool_if:n #1 { T , F , TF }
9154 {
9155 \if_predicate:w \bool_if_p:n {#1}
9156 \prg_return_true:
9157 \else:
9158 \prg_return_false:
9159 \fi:
9160 }
```

(End definition for `\bool_if:nTF`. This function is documented on page 110.)

**`\bool_if_p:n`** To speed up the case of a single predicate, f-expand and check whether the result is one token (possibly surrounded by spaces), which must be `\c_true_bool` or `\c_false_bool`. We use a version of `\tl_if_single:nTF` optimized for speed since we know that an empty #1 is an error. The auxiliary `\__bool_if_p_aux:w` removes the trailing parenthesis and gets rid of any space. For the general case, first issue a `\group_align_safe_begin:` as we are using && as syntax shorthand for the And operation and we need to hide it for

T<sub>E</sub>X. This group is closed after `\__bool_get_next:NN` returns `\c_true_bool` or `\c_false_bool`. That function requires the trailing parenthesis to know where the expression ends.

```

9161 \cs_new:Npn \bool_if_p:n { \exp_args:Nf __bool_if_p:n }
9162 \cs_new:Npn __bool_if_p:n #1
9163 {
9164 \tl_if_empty:oT { \use_none:nn #1 . } { __bool_if_p_aux:w }
9165 \group_align_safe_begin:
9166 \exp_after:wN
9167 \group_align_safe_end:
9168 \exp:w \exp_end_continue_f:w % (
9169 __bool_get_next:NN \use_i:nnnn #1)
9170 }
9171 \cs_new:Npn __bool_if_p_aux:w #1 \use_i:nnnn #2#3 {#2}

```

(End definition for `\bool_if_p:n`, `\__bool_if_p:n`, and `\__bool_if_p_aux:w`. This function is documented on page 110.)

`\__bool_get_next:NN` The GetNext operation. Its first argument is `\use_i:nnnn`, `\use_i:i:nnnn`, `\use_iii:nnnn`, or `\use_iv:nnnn` (we call these “states”). In the first state, this function eventually expand to the truth value `\c_true_bool` or `\c_false_bool` of the expression which follows until the next unmatched closing parenthesis. For instance “`\__bool_get_next:NN \use_i:nnnn \c_true_bool && \c_true_bool )`” (including the closing parenthesis) expands to `\c_true_bool`. In the second state (after a !) the logic is reversed. We call these two states “normal” and the next two “skipping”. In the third state (after `\c_true_bool||`) it always returns `\c_true_bool`. In the fourth state (after `\c_false_bool&&`) it always returns `\c_false_bool` and also stops when encountering ||, not only parentheses. This code itself is a switch: if what follows is neither ! nor |, we assume it is a predicate.

```

9172 \cs_new:Npn __bool_get_next:NN #1#2
9173 {
9174 \use:c
9175 {
9176 __bool_
9177 \if_meaning:w !#2 ! \else: \if_meaning:w (#2 (\else: p \fi: \fi:
9178 :Nw
9179 }
9180 #1 #2
9181 }

```

(End definition for `\__bool_get_next:NN`.)

`\__bool_!:_Nw` The Not operation reverses the logic: it discards the ! token and calls the GetNext operation with the appropriate first argument. Namely the first and second states are interchanged, but after `\c_true_bool||` or `\c_false_bool&&` the ! is ignored.

```

9182 \cs_new:cpn { __bool_!:_Nw } #1#2
9183 {
9184 \exp_after:wN __bool_get_next:NN
9185 #1 \use_i:i:nnnn \use_i:nnnn \use_ii:nnnn \use_iv:nnnn
9186 }

```

(End definition for `\__bool_!:_Nw`.)

\\_\\_bool\\_\\_(:Nw) The Open operation starts a sub-expression after discarding the open parenthesis. This is done by calling GetNext (which eventually discards the corresponding closing parenthesis), with a post-processing step which looks for And, Or or Close after the group.

```

9187 \cs_new:cpn { __bool_(:Nw) #1#2
9188 {
9189 \exp_after:wN __bool_choose:NNN \exp_after:wN #1
9190 \int_value:w __bool_get_next>NN \use_i:nnnn
9191 }

```

(End definition for \\_\\_bool\\_\\_(:Nw.)

\\_\\_bool\_p:Nw If what follows GetNext is neither ! nor (, evaluate the predicate using the primitive \int\_value:w. The canonical true and false values have numerical values 1 and 0 respectively. Look for And, Or or Close afterwards.

```

9192 \cs_new:cpn { __bool_p:Nw } #1
9193 { \exp_after:wN __bool_choose:NNN \exp_after:wN #1 \int_value:w }

```

(End definition for \\_\\_bool\_p:Nw.)

\\_\\_bool\_choose:NNN The arguments are #1: a function such as \use\_i:nnnn, #2: 0 or 1 encoding the current truth value, #3: the next operation, And, Or or Close. We distinguish three cases according to a combination of #1 and #2. Case 2 is when #1 is \use\_iii:nnnn (state 3), namely after \c\_true\_bool ||. Case 1 is when #1 is \use\_i:nnnn and #2 is true or when #1 is \use\_ii:nnnn and #2 is false, for instance for !\c\_false\_bool. Case 0 includes the same with true/false interchanged and the case where #1 is \use\_iv:nnnn namely after \c\_false\_bool &&.

When seeing ) the current subexpression is done, leave the appropriate boolean. When seeing & in case 0 go into state 4, equivalent to having seen \c\_false\_bool &&. In case 1, namely when the argument is true and we are in a normal state continue in the normal state 1. In case 2, namely when skipping alternatives in an Or, continue in the same state. When seeing | in case 0, continue in a normal state; in particular stop skipping for \c\_false\_bool && because that binds more tightly than ||. In the other two cases start skipping for \c\_true\_bool ||.

```

9194 \cs_new:Npn __bool_choose:NNN #1#2#3
9195 {
9196 \use:c
9197 {
9198 __bool_ \token_to_str:N #3 -
9199 #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: } 2 0 :
9200 }
9201 }
9202 \cs_new:cpn { __bool_0: } { \c_false_bool }
9203 \cs_new:cpn { __bool_1: } { \c_true_bool }
9204 \cs_new:cpn { __bool_2: } { \c_true_bool }
9205 \cs_new:cpn { __bool_&_0: } & { __bool_get_next>NN \use_iv:nnnn }
9206 \cs_new:cpn { __bool_&_1: } & { __bool_get_next>NN \use_i:nnnn }
9207 \cs_new:cpn { __bool_&_2: } & { __bool_get_next>NN \use_iii:nnnn }
9208 \cs_new:cpn { __bool_|_0: } | { __bool_get_next>NN \use_i:nnnn }
9209 \cs_new:cpn { __bool_|_1: } | { __bool_get_next>NN \use_iii:nnnn }
9210 \cs_new:cpn { __bool_|_2: } | { __bool_get_next>NN \use_iii:nnnn }

```

(End definition for \\_\\_bool\_choose:NNN and others.)

**\bool\_lazy\_all\_p:n** Go through the list of expressions, stopping whenever an expression is `false`. If the end is reached without finding any `false` expression, then the result is `true`.

```

__bool_lazy_all:n
9211 \cs_new:Npn \bool_lazy_all_p:n #1
9212 { __bool_lazy_all:n #1 \q__bool_recursion_tail \q__bool_recursion_stop }
9213 \prg_new_conditional:Npnn \bool_lazy_all:n #1 { T , F , TF }
9214 {
9215 \if_predicate:w \bool_lazy_all_p:n {#1}
9216 \prg_return_true:
9217 \else:
9218 \prg_return_false:
9219 \fi:
9220 }
9221 \cs_new:Npn __bool_lazy_all:n #1
9222 {
9223 __bool_if_recursion_tail_stop_do:nn {#1} { \c_true_bool }
9224 \bool_if:nF {#1}
9225 { __bool_use_i_delimit_by_q_recursion_stop:nw { \c_false_bool } }
9226 __bool_lazy_all:n
9227 }

```

(End definition for `\bool_lazy_all:nTF` and `\__bool_lazy_all:n`. This function is documented on page 110.)

**\bool\_lazy\_and\_p:nn** Only evaluate the second expression if the first is `true`. Note that `#2` must be removed as an argument, not just by skipping to the `\else:` branch of the conditional since `#2` may contain unbalanced TeX conditionals.

```

9228 \prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }
9229 {
9230 \if_predicate:w
9231 \bool_if:nTF {#1} { \bool_if_p:n {#2} } { \c_false_bool }
9232 \prg_return_true:
9233 \else:
9234 \prg_return_false:
9235 \fi:
9236 }

```

(End definition for `\bool_lazy_and:nnTF`. This function is documented on page 110.)

**\bool\_lazy\_any\_p:n** Go through the list of expressions, stopping whenever an expression is `true`. If the end is reached without finding any `true` expression, then the result is `false`.

```

__bool_lazy_any:n
9237 \cs_new:Npn \bool_lazy_any_p:n #1
9238 { __bool_lazy_any:n #1 \q__bool_recursion_tail \q__bool_recursion_stop }
9239 \prg_new_conditional:Npnn \bool_lazy_any:n #1 { T , F , TF }
9240 {
9241 \if_predicate:w \bool_lazy_any_p:n {#1}
9242 \prg_return_true:
9243 \else:
9244 \prg_return_false:
9245 \fi:
9246 }
9247 \cs_new:Npn __bool_lazy_any:n #1
9248 {
9249 __bool_if_recursion_tail_stop_do:nn {#1} { \c_false_bool }
9250 \bool_if:nT {#1}

```

```

9251 { __bool_use_i_delimit_by_q_recursion_stop:nw { \c_true_bool } }
9252 __bool_lazy_any:n
9253 }

```

(End definition for `\bool_lazy_any:nTF` and `\__bool_lazy_any:n`. This function is documented on page 111.)

`\bool_lazy_or_p:nn` Only evaluate the second expression if the first is `false`.

```

9254 \prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
9255 {
9256 \if_predicate:w
9257 \bool_if:nTF {#1} { \c_true_bool } { \bool_if_p:n {#2} }
9258 \prg_return_true:
9259 \else:
9260 \prg_return_false:
9261 \fi:
9262 }

```

(End definition for `\bool_lazy_or:nnTF`. This function is documented on page 111.)

`\bool_not_p:n` The Not variant just reverses the outcome of `\bool_if_p:n`. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```
9263 \cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! (#1) } }
```

(End definition for `\bool_not_p:n`. This function is documented on page 111.)

`\bool_xor_p:nn` Exclusive or. If the boolean expressions have same truth value, return `false`, otherwise return `true`.

```

9264 \prg_new_conditional:Npnn \bool_xor:nn #1#2 { p , T , F , TF }
9265 {
9266 \bool_if:nT {#1} \reverse_if:N
9267 \if_predicate:w \bool_if_p:n {#2}
9268 \prg_return_true:
9269 \else:
9270 \prg_return_false:
9271 \fi:
9272 }

```

(End definition for `\bool_xor:nnTF`. This function is documented on page 111.)

## 14.6 Logical loops

`\bool_while_do:Nn` A `while` loop where the boolean is tested before executing the statement. The “while” version executes the code as long as the boolean is true; the “until” version executes the code as long as the boolean is false.

```

9273 \cs_new:Npn \bool_while_do:Nn #1#2
9274 { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
9275 \cs_new:Npn \bool_until_do:Nn #1#2
9276 { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
9277 \cs_generate_variant:Nn \bool_while_do:Nn { c }
9278 \cs_generate_variant:Nn \bool_until_do:Nn { c }

```

(End definition for `\bool_while_do:Nn` and `\bool_until_do:Nn`. These functions are documented on page 111.)

```

\bool_do_while:Nn A do-while loop where the body is performed at least once and the boolean is tested
\bool_do_while:cn after executing the body. Otherwise identical to the above functions.

9279 \cs_new:Npn \bool_do_while:Nn #1#2
9280 { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
9281 \cs_new:Npn \bool_do_until:Nn #1#2
9282 { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
9283 \cs_generate_variant:Nn \bool_do_while:Nn { c }
9284 \cs_generate_variant:Nn \bool_do_until:Nn { c }

(End definition for \bool_do_while:Nn and \bool_do_until:Nn. These functions are documented on
page 111.)

```

\bool\_while\_do:nn Loop functions with the test either before or after the first body expansion.

```

\bool_do_while:nn
\bool_until_do:nn
\bool_do_until:nn

9285 \cs_new:Npn \bool_while_do:nn #1#2
9286 {
9287 \bool_if:nT {#1}
9288 {
9289 #2
9290 \bool_while_do:nn {#1} {#2}
9291 }
9292 }
9293 \cs_new:Npn \bool_do_while:nn #1#2
9294 {
9295 #2
9296 \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
9297 }
9298 \cs_new:Npn \bool_until_do:nn #1#2
9299 {
9300 \bool_if:nF {#1}
9301 {
9302 #2
9303 \bool_until_do:nn {#1} {#2}
9304 }
9305 }
9306 \cs_new:Npn \bool_do_until:nn #1#2
9307 {
9308 #2
9309 \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
9310 }

```

(End definition for \bool\_while\_do:nn and others. These functions are documented on page 112.)

## 14.7 Producing multiple copies

9311 ⟨@=prg⟩

\prg\_replicate:nn This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input 25 result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with 25 then the next function is the one that inserts two copies but it sees the ten copies handed

```

\prg_replicate:nn
__prg_replicate:N
_prg_replicate_first:N
 _prg_replicate_
 _prg_replicate_0:n
 _prg_replicate_1:n
 _prg_replicate_2:n
 _prg_replicate_3:n
 _prg_replicate_4:n
 _prg_replicate_5:n
 _prg_replicate_6:n
 _prg_replicate_7:n
 _prg_replicate_8:n
 _prg_replicate_9:n
_prg_replicate_first_-:n
__prg_replicate_first_0:n
__prg_replicate_first_1:n
__prg_replicate_first_2:n

```

down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of \exp:w, which ensures that \prg\_replicate:nn only requires two steps of expansion.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it severely affects the main memory once you start demanding hundreds of thousands of copies. Now I don't think this is a real limitation for any ordinary use, and if necessary, it is possible to write \prg\_replicate:nn {1000} { \prg\_replicate:nn {1000} {\langle code\rangle} }. An alternative approach is to create a string of m's with \exp:w which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```

9312 \cs_new:Npn \prg_replicate:nn #1
9313 {
9314 \exp:w
9315 \exp_after:wN __prg_replicate_first:N
9316 \int_value:w \int_eval:n {#1}
9317 \cs_end:
9318 }
9319 \cs_new:Npn __prg_replicate:N #1
9320 { \cs:w __prg_replicate_#1 :n __prg_replicate:N }
9321 \cs_new:Npn __prg_replicate_first:N #1
9322 { \cs:w __prg_replicate_first_ #1 :n __prg_replicate:N }

```

Then comes all the functions that do the hard work of inserting all the copies. The first function takes :n as a parameter.

```

9323 \cs_new:Npn __prg_replicate_ :n #1 { \cs_end: }
9324 \cs_new:cpn { __prg_replicate_0:n } #1
9325 { \cs_end: {#1#1#1#1#1#1#1#1#1} }
9326 \cs_new:cpn { __prg_replicate_1:n } #1
9327 { \cs_end: {#1#1#1#1#1#1#1#1#1} #1 }
9328 \cs_new:cpn { __prg_replicate_2:n } #1
9329 { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1 }
9330 \cs_new:cpn { __prg_replicate_3:n } #1
9331 { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1 }
9332 \cs_new:cpn { __prg_replicate_4:n } #1
9333 { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
9334 \cs_new:cpn { __prg_replicate_5:n } #1
9335 { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
9336 \cs_new:cpn { __prg_replicate_6:n } #1
9337 { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
9338 \cs_new:cpn { __prg_replicate_7:n } #1
9339 { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
9340 \cs_new:cpn { __prg_replicate_8:n } #1
9341 { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }
9342 \cs_new:cpn { __prg_replicate_9:n } #1
9343 { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }

```

Users shouldn't ask for something to be replicated once or even not at all but...

```

9344 \cs_new:cpn { __prg_replicate_first_-:n } #1
9345 {
9346 \exp_end:
9347 __kernel_msg_expandable_error:nn { kernel } { negative-replication }

```

```

9348 }
9349 \cs_new:cpn { __prg_replicate_first_0:n } #1 { \exp_end: }
9350 \cs_new:cpn { __prg_replicate_first_1:n } #1 { \exp_end: #1 }
9351 \cs_new:cpn { __prg_replicate_first_2:n } #1 { \exp_end: #1#1 }
9352 \cs_new:cpn { __prg_replicate_first_3:n } #1 { \exp_end: #1#1#1 }
9353 \cs_new:cpn { __prg_replicate_first_4:n } #1 { \exp_end: #1#1#1#1 }
9354 \cs_new:cpn { __prg_replicate_first_5:n } #1 { \exp_end: #1#1#1#1#1 }
9355 \cs_new:cpn { __prg_replicate_first_6:n } #1 { \exp_end: #1#1#1#1#1#1 }
9356 \cs_new:cpn { __prg_replicate_first_7:n } #1 { \exp_end: #1#1#1#1#1#1#1 }
9357 \cs_new:cpn { __prg_replicate_first_8:n } #1 { \exp_end: #1#1#1#1#1#1#1#1 }
9358 \cs_new:cpn { __prg_replicate_first_9:n } #1
9359 { \exp_end: #1#1#1#1#1#1#1#1#1#1#1 }

```

(End definition for `\prg_replicate:nn` and others. This function is documented on page 112.)

## 14.8 Detecting TeX's mode

`\mode_if_vertical:p`: For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\exp_end:` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```

9360 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
9361 { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_vertical:TF`. This function is documented on page 113.)

`\mode_if_horizontal:p`: For testing horizontal mode.

```

9362 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
9363 { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_horizontal:TF`. This function is documented on page 112.)

`\mode_if_inner:p`: For testing inner mode.

```

9364 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
9365 { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_inner:TF`. This function is documented on page 112.)

`\mode_if_math:p`: For testing math mode. At the beginning of an alignment cell, this should be used only inside a non-expandable function.

```

9366 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
9367 { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_math:TF`. This function is documented on page 112.)

## 14.9 Internal programming functions

\group\_align\_safe\_begin:

\group\_align\_safe\_end:

TeX's alignment structures present many problems. As Knuth says himself in *TeX: The Program*: "It's sort of a miracle whenever \halign or \valign work, [...]" One problem relates to commands that internally issues a \cr but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a & with category code 4 we get some sort of weird error message because the underlying \futurelet stores the token at the end of the alignment template. This could be a &<sub>4</sub> giving a message like ! Misplaced \cr. or even worse: it could be the \endtemplate token causing even more trouble! To solve this we have to open a special group so that TeX still thinks it's on safe ground but at the same time we don't want to introduce any brace group that may find its way to the output. The following functions help with this by using code documented only in Appendix D of *The TeXbook*... We place the \if\_false: { \fi: part at that place so that the successive expansions of \group\_align\_safe\_begin/end: are always brace balanced.

```
9368 \cs_new:Npn \group_align_safe_begin:
9369 { \if_int_compare:w \if_false: { \fi: ' } = \c_zero_int \fi: }
9370 \cs_new:Npn \group_align_safe_end:
9371 { \if_int_compare:w '{ = \c_zero_int } \fi: }
```

(End definition for \group\_align\_safe\_begin: and \group\_align\_safe\_end:. These functions are documented on page 114.)

\g\_kernel\_prg\_map\_int

A nesting counter for mapping.

```
9372 \int_new:N \g_kernel_prg_map_int
```

(End definition for \g\_kernel\_prg\_map\_int.)

\prg\_break\_point:Nn  
\prg\_map\_break:Nn

These are defined in l3basics, as they are needed "early". This is just a reminder that is the case!

(End definition for \prg\_break\_point:Nn and \prg\_map\_break:Nn. These functions are documented on page 113.)

\prg\_break\_point:  
\prg\_break:  
\prg\_break:n

Also done in l3basics.

(End definition for \prg\_break\_point:, \prg\_break:, and \prg\_break:n. These functions are documented on page 114.)

```
9373
```

## 15 I3sys implementation

```
9374
```

### 15.1 Kernel code

```
9375
```

```
9376
```

#### 15.1.1 Detecting the engine

\\_\_sys\_const:nn

Set the T, F, TF, p forms of #1 to be constants equal to the result of evaluating the boolean expression #2.

```
9377 \cs_new_protected:Npn __sys_const:nn #1#2
9378 {
```

```

9379 \bool_if:nTF {#2}
9380 {
9381 \cs_new_eq:cN { #1 :T } \use:n
9382 \cs_new_eq:cN { #1 :F } \use:none:n
9383 \cs_new_eq:cN { #1 :TF } \use_i:nn
9384 \cs_new_eq:cN { #1 _p: } \c_true_bool
9385 }
9386 {
9387 \cs_new_eq:cN { #1 :T } \use:none:n
9388 \cs_new_eq:cN { #1 :F } \use:n
9389 \cs_new_eq:cN { #1 :TF } \use_ii:nn
9390 \cs_new_eq:cN { #1 _p: } \c_false_bool
9391 }
9392 }

```

(End definition for `\_sys_const:nn`.)

`\sys_if_engine_luatex:p`: Set up the engine tests on the basis exactly one test should be true. Mainly a case of looking for the appropriate marker primitive.

```

\sys_if_engine_luatex:TF
\sys_if_engine_pdftex:p:
\sys_if_engine_pdftex:TF
\sys_if_engine_ptex:p:
\sys_if_engine_ptex:TF
\sys_if_engine_uptex:p:
\sys_if_engine_uptex:TF
\sys_if_engine_xetex:p:
\sys_if_engine_xetex:TF
 \c_sys_engine_str
9393 \str_const:Nx \c_sys_engine_str
9394 {
9395 \cs_if_exist:NT \tex_luatexversion:D { luatex }
9396 \cs_if_exist:NT \tex_pdftexversion:D { pdftex }
9397 \cs_if_exist:NT \tex_kanjiskip:D
9398 {
9399 \cs_if_exist:NTF \tex_enablecjktoken:D
9400 { uptex }
9401 { ptex }
9402 }
9403 \cs_if_exist:NT \tex_XeTeXversion:D { xetex }
9404 }
9405 \tl_map_inline:nn { { luatex } { pdftex } { ptex } { uptex } { xetex } }
9406 {
9407 _sys_const:nn { sys_if_engine_ #1 }
9408 { \str_if_eq_p:Vn \c_sys_engine_str {#1} }
9409 }

```

(End definition for `\sys_if_engine_luatex:TF` and others. These functions are documented on page 115.)

`\c_sys_engine_exec_str` and `\c_sys_engine_format_str` take the functions defined above, and set up the engine and format names. `\c_sys_engine_exec_str` differs from `\c_sys_engine_str` as it is the *actual* engine name, not a “filtered” version. It differs for `ptex` and `uptex`, which have a leading `e`, and for `luatex`, because L<sup>A</sup>T<sub>E</sub>X uses the LuaHBT<sub>E</sub>X engine.

`\c_sys_engine_format_str` is quite similar to `\c_sys_engine_str`, except that it differentiates `pdfflatex` from `latex` (which is pdfT<sub>E</sub>X in DVI mode). This differentiation, however, is reliable only if the user doesn’t change `\tex_pdfoutput:D` before loading this code.

```

9410 \group_begin:
9411 \cs_set_eq:NN \lua_now:e \tex_directlua:D
9412 \str_const:Nx \c_sys_engine_exec_str
9413 {
9414 \sys_if_engine_pdftex:T { pdf }
9415 \sys_if_engine_xetex:T { xe }

```

```

9416 \sys_if_engine_ptex:T { ep }
9417 \sys_if_engine_uptex:T { eup }
9418 \sys_if_engine_luatex:T
9419 {
9420 lua \lua_now:e
9421 {
9422 if (pcall(require, 'luaharfbuzz')) then ~
9423 tex.print("hb") ~
9424 end
9425 }
9426 }
9427 tex
9428 }
9429 \group_end:
9430 \str_const:Nx \c_sys_engine_format_str
9431 {
9432 \cs_if_exist:NTF \fmtname
9433 {
9434 \bool_lazy_or:nnTF
9435 { \str_if_eq_p:Vn \fmtname { plain } }
9436 { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
9437 {
9438 \sys_if_engine_pdftex:T
9439 { \int_compare:nNnT { \tex_pdfoutput:D } = { 1 } { pdf } }
9440 \sys_if_engine_xetex:T { xe }
9441 \sys_if_engine_ptex:T { p }
9442 \sys_if_engine_uptex:T { up }
9443 \sys_if_engine_luatex:T
9444 {
9445 \int_compare:nNnT { \tex_pdfoutput:D } = { 0 } { dvi }
9446 lua
9447 }
9448 \str_if_eq:VnTF \fmtname { LaTeX2e }
9449 { latex }
9450 {
9451 \bool_lazy_and:nnT
9452 { \sys_if_engine_pdftex_p: }
9453 { \int_compare_p:nNn { \tex_pdfoutput:D } = { 0 } }
9454 { e }
9455 tex
9456 }
9457 }
9458 { \fmtname }
9459 }
9460 { unknown }
9461 }

```

(End definition for `\c_sys_engine_exec_str` and `\c_sys_engine_format_str`. These variables are documented on page 116.)

### 15.1.2 Randomness

This candidate function is placed there because `\sys_if_rand_exist:TF` is used in `\l3fp-rand`.

`\sys_if_rand_exist_p:` Currently, randomness exists under pdfTeX, LuaTeX, pTeX and upTeX.

`\sys_if_rand_exist:TF`

```
9462 __sys_const:nn { sys_if_rand_exist }
9463 { \cs_if_exist_p:N \tex_uniformdeviate:D }
```

(End definition for `\sys_if_rand_exist:TF`. This function is documented on page 271.)

### 15.1.3 Platform

Setting these up requires the file module (file lookup), so is actually implemented there.

(End definition for `\sys_if_platform_unix:TF`, `\sys_if_platform_windows:TF`, and `\c_sys_platform_str`. These functions are documented on page 116.)

### 15.1.4 Configurations

Loading the backend code is pretty simply: check that the backend is valid, then load it up.

```
9464 \cs_new_protected:Npn \sys_load_backend:n #1
9465 {
9466 \sys_finalise:
9467 \str_if_exist:NTF \c_sys_backend_str
9468 {
9469 \str_if_eq:VnF \c_sys_backend_str {#1}
9470 { __kernel_msg_error:nn { sys } { backend-set } }
9471 }
9472 {
9473 \tl_if_blank:nF {#1}
9474 { \tl_set:Nn \g__sys_backend_tl {#1} }
9475 __sys_load_backend_check:N \g__sys_backend_tl
9476 \str_const:Nx \c_sys_backend_str { \g__sys_backend_tl }
9477 __kernel_sys_configuration_load:n
9478 { l3backend- \c_sys_backend_str }
9479 }
9480 }
9481 \cs_new_protected:Npn __sys_load_backend_check:N #1
9482 {
9483 \sys_if_engine_xetex:TF
9484 {
9485 \str_case:VnF #1
9486 {
9487 { dvisvgm } { }
9488 { xdvipdfmx } { \tl_gset:Nn #1 { xetex } }
9489 { xetex } { }
9490 }
9491 {
9492 __kernel_msg_error:nnxx { sys } { wrong-backend }
9493 #1 { xetex }
9494 \tl_gset:Nn #1 { xetex }
9495 }
9496 }
9497 }
9498 \sys_if_output_pdf:TF
9499 {
9500 \str_if_eq:VnTF #1 { pdfmode }
```

```

9501 {
9502 \sys_if_engine_luatex:TF
9503 { \tl_gset:Nn #1 { luatex } }
9504 { \tl_gset:Nn #1 { pdftex } }
9505 }
9506 {
9507 \bool_lazy_or:nnF
9508 { \str_if_eq_p:Vn #1 { luatex } }
9509 { \str_if_eq_p:Vn #1 { pdftex } }
9510 }
9511 {
9512 __kernel_msg_error:nxxx { sys } { wrong-backend }
9513 #1 { \sys_if_engine_luatex:TF { luatex } { pdftex } }
9514 \sys_if_engine_luatex:TF
9515 { \tl_gset:Nn #1 { luatex } }
9516 { \tl_gset:Nn #1 { pdftex } }
9517 }
9518 }
9519 {
9520 \str_case:VnF #1
9521 {
9522 { dvipdfmx } { }
9523 { dvips } { }
9524 { dvisvgm } { }
9525 }
9526 {
9527 __kernel_msg_error:nxxx { sys } { wrong-backend }
9528 #1 { dvips }
9529 \tl_gset:Nn #1 { dvips }
9530 }
9531 }
9532 }
9533 }

```

(End definition for `\sys_load_backend:n`, `\__sys_load_backend_check:N`, and `\c_sys_backend_str`. These functions are documented on page [118](#).)

```

\g__sys_debug_bool
\g__sys_deprecation_bool
9534 \bool_new:N \g__sys_debug_bool
9535 \bool_new:N \g__sys_deprecation_bool

```

(End definition for `\g__sys_debug_bool` and `\g__sys_deprecation_bool`.)

`\sys_load_debug:` Simple.

```

\sys_load_deprecation:
9536 \cs_new_protected:Npn \sys_load_debug:
9537 {
9538 \bool_if:NF \g__sys_debug_bool
9539 { __kernel_sys_configuration_load:n { 13debug } }
9540 \bool_gset_true:N \g__sys_debug_bool
9541 }
9542 \cs_new_protected:Npn \sys_load_deprecation:
9543 {
9544 \bool_if:NF \g__sys_deprecation_bool
9545 { __kernel_sys_configuration_load:n { 13depreciation } }
9546 \bool_gset_true:N \g__sys_deprecation_bool

```

9547 }

(End definition for `\sys_load_debug:` and `\sys_load_deprecation:`. These functions are documented on page 118.)

### 15.1.5 Access to the shell

```
\l__sys_internal_tl
9548 \tl_new:N \l__sys_internal_tl
```

(End definition for `\l__sys_internal_tl`.)

`\c__sys_marker_tl` The same idea as the marker for rescanning token lists.

```
9549 \tl_const:Nx \c__sys_marker_tl { : \token_to_str:N : }
```

(End definition for `\c__sys_marker_tl`.)

`\sys_get_shell:nnTF` Setting using a shell is at this level just a slightly specialised file operation, with an additional check for quotes, as these are not supported.

```
\sys_get_shell:nnN
9550 \cs_new_protected:Npn \sys_get_shell:nnN #1#2#3
9551 {
9552 \sys_get_shell:nnNF {#1} {#2} #3
9553 { \tl_set:Nn #3 { \q_no_value } }
9554 }
9555 \prg_new_protected_conditional:Npnn \sys_get_shell:nnN #1#2#3 { T , F , TF }
9556 {
9557 \sys_if_shell:TF
9558 { \exp_args:No __sys_get:nnN { \tl_to_str:n {#1} } {#2} #3 }
9559 { \prg_return_false: }
9560 }
9561 \cs_new_protected:Npn __sys_get:nnN #1#2#3
9562 {
9563 \tl_if_in:nnTF {#1} { " }
9564 {
9565 __kernel_msg_error:nnx
9566 { kernel } { quote-in-shell } {#1}
9567 \prg_return_false:
9568 }
9569 {
9570 \group_begin:
9571 \if_false: { \fi:
9572 \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
9573 \exp_args:No \tex_everyeof:D { \c__sys_marker_tl }
9574 #2 \scan_stop:
9575 \exp_after:wN __sys_get_do:Nw
9576 \exp_after:wN #3
9577 \exp_after:wN \prg_do_nothing:
9578 \tex_input:D / "#1" \scan_stop:
9579 \if_false: } \fi:
9580 \prg_return_true:
9581 }
9582 }
9583 \exp_args:Nno \use:nn
9584 { \cs_new_protected:Npn __sys_get_do:Nw #1#2 }
9585 { \c__sys_marker_tl }
```

```

9586 {
9587 \group_end:
9588 \tl_set:Nn #1 {\#2}
9589 }

```

(End definition for `\sys_get_shell:nnNTF` and others. These functions are documented on page 117.)

`\c_sys_shell_stream_int` This is not needed for LuaTeX: shell escape there isn't done using a TeX interface.

```

9590 \sys_if_engine_luatex:F
9591 { \int_const:Nn \c_sys_shell_stream_int { 18 } }

```

(End definition for `\c_sys_shell_stream_int`.)

`\sys_shell_now:n` Execute commands through shell escape immediately.

`\_sys_shell_now:e` For LuaTeX, we use a pseudo-primitive to do the actual work.

```

9592 </tex>
9593 <*lua>
9594 do
9595 local os_exec = os.execute
9596
9597 local function shellexcape(cmd)
9598 local status,msg = os_exec(cmd)
9599 if status == nil then
9600 write_nl("log","runsystem(\" .. cmd .. \")...(\" .. msg .. \")\n")
9601 elseif status == 0 then
9602 write_nl("log","runsystem(\" .. cmd .. \")...executed\n")
9603 else
9604 write_nl("log","runsystem(\" .. cmd .. \")...failed \" .. (msg or "") .. "\n")
9605 end
9606 end
9607 luacmd("_sys_shell_now:e", function()
9608 shellexcape(scan_string())
9609 end, "global", "protected")
9610 </lua>
9611 <*tex>
9612 \sys_if_engine_luatex:TF
9613 {
9614 \cs_new_protected:Npn \sys_shell_now:n #1
9615 { _sys_shell_now:e { \exp_not:n {#1} } }
9616 }
9617 {
9618 \cs_new_protected:Npn \sys_shell_now:n #1
9619 { \iow_now:Nn \c_sys_shell_stream_int {#1} }
9620 }
9621 \cs_generate_variant:Nn \sys_shell_now:n { x }
9622 </tex>

```

(End definition for `\sys_shell_now:n` and `\_sys_shell_now:e`. This function is documented on page 118.)

`\sys_shell_shipout:n` Execute commands through shell escape at shipout.

`\_sys_shell_shipout:e` For LuaTeX, we use the same helper as above but delayed to using a late\_lua whatsit.

```

9623 <*lua>
9624 local whatsit_id = node.id'whatsit'

```

```

9625 local latelua_sub = node.subtype'late_lua'
9626 local node_new = node.direct.new
9627 local setfield = node.direct.setwhatsitfield or node.direct.setfield
9628 local node_write = node.direct.write
9629
9630 luacmd("__sys_shell_shipout:e", function()
9631 local cmd = scan_string()
9632 local n = node_new(whatsit_id, latelua_sub)
9633 setfield(n, 'data', function() shellescape(cmd) end)
9634 node_write(n)
9635 end, "global", "protected")
9636 end
9637
```

{/lua}

{\*tex}

```

9638 \sys_if_engine_luatex:TF
9639 {
9640 \cs_new_protected:Npn \sys_shell_shipout:n #1
9641 { __sys_shell_shipout:e { \exp_not:n {#1} } }
9642 }
9643 {
9644 \cs_new_protected:Npn \sys_shell_shipout:n #1
9645 { \iow_shipout:Nn \c__sys_shell_stream_int {#1} }
9646 }
9647
9648 \cs_generate_variant:Nn \sys_shell_shipout:n { x }
```

(End definition for `\sys_shell_shipout:n` and `\__sys_shell_shipout:e`. This function is documented on page 118.)

## 15.2 Dynamic (every job) code

```

\sys_everyjob:
__sys_everyjob:n
\g__sys_everyjob_tl
9649 \cs_new_protected:Npn \sys_everyjob:
9650 {
9651 \tl_use:N \g__sys_everyjob_tl
9652 \tl_gclear:N \g__sys_everyjob_tl
9653 }
9654 \cs_new_protected:Npn __sys_everyjob:n #1
9655 { \tl_gput_right:Nn \g__sys_everyjob_tl {#1} }
9656 \tl_new:N \g__sys_everyjob_tl
```

(End definition for `\sys_everyjob:`, `\__sys_everyjob:n`, and `\g__sys_everyjob_tl`. This function is documented on page ??.)

### 15.2.1 The name of the job

`\c_sys_jobname_str`

Inherited from the L<sup>A</sup>T<sub>E</sub>X3 name for the primitive. This *has* to be the primitive as it's set in `\everyjob`. If the user does

```
pdflatex \input some-file-name
```

then `\everyjob` is inserted *before* `\jobname` is changed form `\texput`, and thus we would have the wrong result.

```

9657 __sys_everyjob:n
9658 { \cs_new_eq:NN \c_sys_jobname_str \tex_jobname:D }
```

(End definition for `\c_sys_jobname_str`. This variable is documented on page 115.)

### 15.2.2 Time and date

```
\c_sys_minute_int
\c_sys_hour_int
\c_sys_day_int
\c_sys_month_int
\c_sys_year_int
```

```
9659 __sys_everyjob:n
9660 {
9661 \group_begin:
9662 \cs_set:Npn __sys_tmp:w #1
9663 {
9664 \str_if_eq:eeTF { \cs_meaning:N #1 } { \token_to_str:N #1 }
9665 { #1 }
9666 {
9667 \cs_if_exist:NTF \tex_primitive:D
9668 {
9669 \bool_lazy_and:nnTF
9670 { \sys_if_engine_xetex_p: }
9671 {
9672 \int_compare_p:nNn
9673 { \exp_after:wN \use_none:n \tex_XeTeXrevision:D }
9674 < { 99999 }
9675 }
9676 { 0 }
9677 { \tex_primitive:D #1 }
9678 }
9679 { 0 }
9680 }
9681 }
9682 \int_const:Nn \c_sys_minute_int
9683 { \int_mod:nn { __sys_tmp:w \time } { 60 } }
9684 \int_const:Nn \c_sys_hour_int
9685 { \int_div_truncate:nn { __sys_tmp:w \time } { 60 } }
9686 \int_const:Nn \c_sys_day_int { __sys_tmp:w \day }
9687 \int_const:Nn \c_sys_month_int { __sys_tmp:w \month }
9688 \int_const:Nn \c_sys_year_int { __sys_tmp:w \year }
9689 \group_end:
9690 }
```

(End definition for `\c_sys_minute_int` and others. These variables are documented on page 115.)

### 15.2.3 Random numbers

`\sys_rand_seed:`

Unpack the primitive. When random numbers are not available, we return zero after an error (and incidentally make sure the number of expansions needed is the same as with random numbers available).

```
9691 __sys_everyjob:n
9692 {
9693 \sys_if_rand_exist:TF
9694 { \cs_new:Npn \sys_rand_seed: { \tex_the:D \tex_randomseed:D } }
9695 {
9696 \cs_new:Npn \sys_rand_seed:
9697 {
9698 \int_value:w
```

```

9699 __kernel_msg_expandable_error:nnn { kernel } { fp-no-random }
9700 { \sys_rand_seed: }
9701 \c_zero_int
9702 }
9703 }
9704 }
```

(End definition for `\sys_rand_seed:`. This function is documented on page 116.)

`\sys_gset_rand_seed:n` The primitive always assigns the seed globally.

```

9705 __sys_everyjob:n
9706 {
9707 \sys_if_rand_exist:TF
9708 {
9709 \cs_new_protected:Npn \sys_gset_rand_seed:n #1
9710 { \tex_setrandomseed:D \int_eval:n {#1} \exp_stop_f: }
9711 }
9712 {
9713 \cs_new_protected:Npn \sys_gset_rand_seed:n #1
9714 {
9715 __kernel_msg_error:nnn { kernel } { fp-no-random }
9716 { \sys_gset_rand_seed:n {#1} }
9717 }
9718 }
9719 }
```

(End definition for `\sys_gset_rand_seed:n`. This function is documented on page 117.)

#### 15.2.4 Access to the shell

`\c_sys_shell_escape_int` Expose the engine's shell escape status to the user.

```

9720 __sys_everyjob:n
9721 {
9722 \int_const:Nn \c_sys_shell_escape_int
9723 {
9724 \sys_if_engine_luatex:TF
9725 {
9726 \tex_directlua:D
9727 { \tex.sprint(status.shell_escape-or~os.execute()) }
9728 }
9729 { \tex_shellescape:D }
9730 }
9731 }
```

(End definition for `\c_sys_shell_escape_int`. This variable is documented on page 117.)

`\sys_if_shell_p:`

`\sys_if_shell:TF`

`\sys_if_shell_unrestricted_p:`

`\sys_if_shell_unrestricted:TF`

`\sys_if_shell_restricted_p:`

`\sys_if_shell_restricted:TF`

Performs a check for whether shell escape is enabled. The first set of functions returns true if either of restricted or unrestricted shell escape is enabled, while the other two sets of functions return true in only one of these two cases.

```

9732 __sys_everyjob:n
9733 {
9734 __sys_const:nn { sys_if_shell }
9735 { \int_compare_p:nNn \c_sys_shell_escape_int > 0 }
9736 __sys_const:nn { sys_if_shell_unrestricted }
```

```

9737 { \int_compare_p:nNn \c_sys_shell_escape_int = 1 }
9738 __sys_const:nn { sys_if_shell_restricted }
9739 { \int_compare_p:nNn \c_sys_shell_escape_int = 2 }
9740 }

```

(End definition for `\sys_if_shell:TF`, `\sys_if_shell_unrestricted:TF`, and `\sys_if_shell_restricted:TF`. These functions are documented on page 117.)

### 15.2.5 Held over from l3file

`\g_file_curr_name_str`

See comments about `\c_sys_jobname_str`: here, as soon as there is file input/output, things get “tided up”.

```

9741 __sys_everyjob:n
9742 { \cs_gset_eq:NN \g_file_curr_name_str \tex_jobname:D }

```

(End definition for `\g_file_curr_name_str`. This variable is documented on page 165.)

## 15.3 Last-minute code

`\sys_finalise:`

`\__sys_finalise:n`

`\g__sys_finalise_tl`

```

9743 \cs_new_protected:Npn \sys_finalise:
9744 {
9745 \sys_everyjob:
9746 \tl_use:N \g__sys_finalise_tl
9747 \tl_gclear:N \g__sys_finalise_tl
9748 }
9749 \cs_new_protected:Npn __sys_finalise:n #1
9750 { \tl_gput_right:Nn \g__sys_finalise_tl {#1} }
9751 \tl_new:N \g__sys_finalise_tl

```

(End definition for `\sys_finalise:`, `\__sys_finalise:n`, and `\g__sys_finalise_tl`. This function is documented on page 118.)

### 15.3.1 Detecting the output

`\sys_if_output_dvi:p`

`\sys_if_output_dvi:TF`

`\sys_if_output_pdf:p`

`\sys_if_output_pdf:TF`

`\c_sys_output_str`

```

9752 __sys_finalise:n
9753 {
9754 \str_const:Nx \c_sys_output_str
9755 {
9756 \int_compare:nNnTF
9757 { \cs_if_exist_use:NF \tex_pdfoutput:D { 0 } } > { 0 }
9758 { pdf }
9759 { dvi }
9760 }
9761 __sys_const:nn { sys_if_output_dvi }
9762 { \str_if_eq_p:Vn \c_sys_output_str { dvi } }
9763 __sys_const:nn { sys_if_output_pdf }
9764 { \str_if_eq_p:Vn \c_sys_output_str { pdf } }
9765 }

```

(End definition for `\sys_if_output_dvi:TF`, `\sys_if_output_pdf:TF`, and `\c_sys_output_str`. These functions are documented on page 116.)

### 15.3.2 Configurations

\g\_\_sys\_backend\_tl As the backend has to be checked and possibly adjusted, the approach here is to create a variable and use that in a one-shot to set a constant.

```

9766 \tl_new:N \g__sys_backend_tl
9767 __sys_finalise:n
9768 {
9769 __kernel_tl_gset:Nx \g__sys_backend_tl
9770 {
9771 \sys_if_engine_xetex:TF
9772 {
9773 \dvipdfmx
9774 }
9775 \sys_if_output_pdf:TF
9776 {
9777 \pdfmode
9778 }
9779 }
}

```

If there is a class option set, and recognised, we pick it up: these will over-ride anything set automatically but will themselves be over-written if there is a package option.

```

9780 __sys_finalise:n
9781 {
9782 \cs_if_exist:NT \c@classoptionslist
9783 {
9784 \cs_if_eq:NNF \c@classoptionslist \scan_stop:
9785 {
9786 \clist_map_inline:Nn \c@classoptionslist
9787 {
9788 \str_case:nnT {\#1}
9789 {
9790 \dvipdfmx
9791 {
9792 \t1_gset:Nn \g__sys_backend_tl { dvipdfmx } }
9793 \dvips
9794 {
9795 \t1_gset:Nn \g__sys_backend_tl { dvips } }
9796 \dvisvgm
9797 {
9798 \t1_gset:Nn \g__sys_backend_tl { dvisvgm } }
9799 \pdftex
9800 {
9801 \t1_gset:Nn \g__sys_backend_tl { pdfmode } }
9802 \xetex
9803 {
9804 \t1_gset:Nn \g__sys_backend_tl { xdvipdfmx } }
9805 }
9806 \clist_remove_all:Nn \c@unusedoptionlist {\#1} }
9807 }
9808 }
9809 }
9810 }
9811 }
9812 }
9813 }
9814 }
9815 }
```

(End definition for \g\_\_sys\_backend\_tl.)

```

9806 </tex>
9807 </package>
```

## 16 I3clist implementation

The following test files are used for this code: m3clist002.

```
9808 {*package}
9809 <@=clist>
```

**\c\_empty\_clist** An empty comma list is simply an empty token list.

```
9810 \cs_new_eq:NN \c_empty_clist \c_empty_tl
```

(End definition for **\c\_empty\_clist**. This variable is documented on page 128.)

**\l\_\_clist\_internal\_clist** Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before **\clist\_new:N**

```
9811 \tl_new:N \l__clist_internal_clist
```

(End definition for **\l\_\_clist\_internal\_clist**.)

**\s\_\_clist\_mark** Internal scan marks.

```
9812 \scan_new:N \s__clist_mark
9813 \scan_new:N \s__clist_stop
```

(End definition for **\s\_\_clist\_mark** and **\s\_\_clist\_stop**.)

**\\_clist\_use\_none\_delimit\_by\_s\_stop:w** Functions to gobble up to a scan mark.

```
9814 \cs_new:Npn _clist_use_none_delimit_by_s_stop:w #1 \s__clist_stop { }
9815 \cs_new:Npn _clist_use_i_delimit_by_s_stop:nw #1 #2 \s__clist_stop {#1}
```

(End definition for **\\_clist\_use\_none\_delimit\_by\_s\_stop:w** and **\\_clist\_use\_i\_delimit\_by\_s\_stop:nw**.)

**\q\_\_clist\_recursion\_tail** Internal recursion quarks.

```
9816 \quark_new:N \q__clist_recursion_tail
9817 \quark_new:N \q__clist_recursion_stop
```

(End definition for **\q\_\_clist\_recursion\_tail** and **\q\_\_clist\_recursion\_stop**.)

**\\_clist\_if\_recursion\_tail\_break:nN** Functions to query recursion quarks.

```
9818 __kernel_quark_new_test:N _clist_if_recursion_tail_break:nN
9819 __kernel_quark_new_test:N _clist_if_recursion_tail_stop:n
```

(End definition for **\\_clist\_if\_recursion\_tail\_break:nN** and **\\_clist\_if\_recursion\_tail\_stop:n**.)

**\\_clist\_tmp:w** A temporary function for various purposes.

```
9820 \cs_new_protected:Npn _clist_tmp:w { }
```

(End definition for **\\_clist\_tmp:w**.)

## 16.1 Removing spaces around items

`\__clist_trim_next:w`

Called as `\exp:w \__clist_trim_next:w \prg_do_nothing: <comma list> ...` it expands to `{<trimmed item>}` where the `<trimmed item>` is the first non-empty result from removing spaces from both ends of comma-delimited items in the `<comma list>`. The `\prg_do_nothing:` marker avoids losing braces. The test for blank items is a somewhat optimized `\tl_if_empty:oTF` construction; if blank, another item is sought, otherwise trim spaces.

```

9821 \cs_new:Npn __clist_trim_next:w #1 ,
9822 {
9823 \tl_if_empty:oTF { \use_none:nn #1 ? }
9824 { __clist_trim_next:w \prg_do_nothing: }
9825 { \tl_trim_spaces_apply:oN {#1} \exp_end: }
9826 }
```

(End definition for `\__clist_trim_next:w`.)

`\__clist_sanitize:n`

`\__clist_sanitize:Nn` The auxiliary `\__clist_sanitize:Nn` receives a delimiter (`\c_empty_tl` the first time, afterwards a comma) and that item as arguments. Unless we are done with the loop it calls `\__clist_wrap_item:w` to unbrace the item (using a comma delimiter is safe since `#2` came from removing spaces from an argument delimited by a comma) and possibly re-brace it if needed.

```

9827 \cs_new:Npn __clist_sanitize:n #1
9828 {
9829 \exp_after:wN __clist_sanitize:Nn \exp_after:wN \c_empty_tl
9830 \exp:w __clist_trim_next:w \prg_do_nothing:
9831 #1 , \q__clist_recursion_tail , \q__clist_recursion_stop
9832 }
9833 \cs_new:Npn __clist_sanitize:Nn #1#2
9834 {
9835 __clist_if_recursion_tail_stop:n {#2}
9836 #1 __clist_wrap_item:w #2 ,
9837 \exp_after:wN __clist_sanitize:Nn \exp_after:wN ,
9838 \exp:w __clist_trim_next:w \prg_do_nothing:
9839 }
```

(End definition for `\__clist_sanitize:n` and `\__clist_sanitize:Nn`.)

`\__clist_if_wrap:nTF`  
`\__clist_if_wrap:w`

True if the argument must be wrapped to avoid getting altered by some clist operations. That is the case whenever the argument

- starts or end with a space or contains a comma,
- is empty, or
- consists of a single braced group.

All `\clist` functions go through the same test when they need to determine whether to brace an item, so it is not a problem that this test has false positives such as “`\s__-clist_mark ?`”. If the argument starts or end with a space or contains a comma then one of the three arguments of `\__clist_if_wrap:w` will have its end delimiter (partly) in one of the three copies of `#1` in `\__clist_if_wrap:nTF`; this has a knock-on effect meaning that the result of the expansion is not empty; in that case, wrap. Otherwise,

the argument is safe unless it starts with a brace group (or is empty) and it is empty or consists of a single n-type argument.

```

9840 \prg_new_conditional:Npnn __clist_if_wrap:n #1 { TF }
9841 {
9842 \tl_if_empty:oTF
9843 {
9844 __clist_if_wrap:w
9845 \s__clist_mark ? #1 ~ \s__clist_mark ? ~ #1
9846 \s__clist_mark , ~ \s__clist_mark #1 ,
9847 }
9848 {
9849 \tl_if_head_is_group:nTF { #1 { } }
9850 {
9851 \tl_if_empty:nTF {#1}
9852 { \prg_return_true: }
9853 {
9854 \tl_if_empty:oTF { \use_none:n #1}
9855 { \prg_return_true: }
9856 { \prg_return_false: }
9857 }
9858 }
9859 { \prg_return_false: }
9860 }
9861 { \prg_return_true: }
9862 }
9863 \cs_new:Npn __clist_if_wrap:w #1 \s__clist_mark ? ~ #2 ~ \s__clist_mark #3 , { }

(End definition for __clist_if_wrap:nTF and __clist_if_wrap:w.)
```

\\_\_clist\_wrap\_item:w Safe items are put in \exp\_not:n, otherwise we put an extra set of braces.

```

9864 \cs_new:Npn __clist_wrap_item:w #1 ,
9865 { __clist_if_wrap:nTF {#1} { \exp_not:n { {#1} } } { \exp_not:n {#1} } }
```

(End definition for \\_\_clist\_wrap\_item:w.)

## 16.2 Allocation and initialisation

**\clist\_new:N** Internally, comma lists are just token lists.

```

\clist_new:c
9866 \cs_new_eq:NN \clist_new:N \tl_new:N
9867 \cs_new_eq:NN \clist_new:c \tl_new:c
```

(End definition for \clist\_new:N. This function is documented on page 119.)

**\clist\_const:Nn** Creating and initializing a constant comma list is done by sanitizing all items (stripping spaces and braces).

```

\clist_const:cn
9868 \cs_new_protected:Npn \clist_const:Nn #1#2
9869 { \tl_const:Nx #1 { __clist_sanitize:n {#2} } }
9870 \cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }
```

(End definition for \clist\_const:Nn. This function is documented on page 120.)

**\clist\_clear:N** Clearing comma lists is just the same as clearing token lists.

```
9871 \cs_new_eq:NN \clist_clear:N \tl_clear:N
9872 \cs_new_eq:NN \clist_clear:c \tl_clear:c
9873 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
9874 \cs_new_eq:NN \clist_gclear:c \tl_gclear:c
```

(End definition for **\clist\_clear:N** and **\clist\_gclear:N**. These functions are documented on page 120.)

**\clist\_clear\_new:N** Once again a copy from the token list functions.

```
9875 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
9876 \cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c
9877 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
9878 \cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c
```

(End definition for **\clist\_clear\_new:N** and **\clist\_gclear\_new:N**. These functions are documented on page 120.)

**\clist\_set\_eq:NN** Once again, these are simple copies from the token list functions.

```
9879 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
9880 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
9881 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
9882 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
9883 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
9884 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
9885 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
9886 \cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

(End definition for **\clist\_set\_eq:NN** and **\clist\_gset\_eq:NN**. These functions are documented on page 120.)

**\clist\_set\_from\_seq:NN** Setting a comma list from a comma-separated list is done using a simple mapping. Safe items are put in **\exp\_not:n**, otherwise we put an extra set of braces. The first comma must be removed, except in the case of an empty comma-list.

```
9887 \cs_new_protected:Npn \clist_set_from_seq:NN
9888 { __clist_set_from_seq>NNNN \clist_clear:N __kernel_tl_set:Nx }
9889 \cs_new_protected:Npn \clist_gset_from_seq:cN
9890 { __clist_set_from_seq>NNNN \clist_gclear:N __kernel_tl_gset:Nx }
9891 \cs_new_protected:Npn __clist_set_from_seq>NNNN #1#2#3#4
9892 {
9893 \seq_if_empty:NTF #4
9894 { #1 #3 }
9895 {
9896 #2 #3
9897 {
9898 \exp_after:wN \use_none:n \exp:w \exp_end_continue_f:w
9899 \seq_map_function:NN #4 __clist_set_from_seq:n
9900 }
9901 }
9902 }
9903 \cs_new:Npn __clist_set_from_seq:n #1
9904 {
9905 ,
9906 __clist_if_wrap:nTF {#1}
9907 { \exp_not:n { {#1} } }
```

```

9908 { \exp_not:n {#1} }
9909 }
9910 \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
9911 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
9912 \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
9913 \cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }

```

(End definition for `\clist_set_from_seq:NN` and others. These functions are documented on page [120](#).)

**\clist\_concat:NNN** Concatenating comma lists is not quite as easy as it seems, as there needs to be the correct addition of a comma to the output. So a little work to do.

```

9914 \cs_new_protected:Npn \clist_concat:NNN
9915 { __clist_concat:NNNN __kernel_tl_set:Nx }
9916 \cs_new_protected:Npn \clist_gconcat:NNN
9917 { __clist_concat:NNNN __kernel_tl_gset:Nx }
9918 \cs_new_protected:Npn __clist_concat:NNNN #1#2#3#4
9919 {
9920 #1 #2
9921 {
9922 \exp_not:o #3
9923 \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
9924 \exp_not:o #4
9925 }
9926 }
9927 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
9928 \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }

```

(End definition for `\clist_concat:NNN`, `\clist_gconcat:NNN`, and `\__clist_concat:NNNN`. These functions are documented on page [120](#).)

**\clist\_if\_exist\_p:N** Copies of the `cs` functions defined in `l3basics`.

```

9929 \prg_new_eq_conditional:Nnn \clist_if_exist:N \cs_if_exist:N
9930 { TF , T , F , p }
9931 \prg_new_eq_conditional:Nnn \clist_if_exist:c \cs_if_exist:c
9932 { TF , T , F , p }

```

(End definition for `\clist_if_exist:NTF`. This function is documented on page [120](#).)

### 16.3 Adding data to comma lists

**\clist\_set:Nn**

```

9933 \cs_new_protected:Npn \clist_set:Nn #1#2
9934 { __kernel_tl_set:Nx #1 { __clist_sanitize:n {#2} } }
9935 \cs_new_protected:Npn \clist_gset:Nn #1#2
9936 { __kernel_tl_gset:Nx #1 { __clist_sanitize:n {#2} } }
9937 \cs_generate_variant:Nn \clist_set:Nn { NV , No , Nx , c , cV , co , cx }
9938 \cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }

```

(End definition for `\clist_set:Nn` and `\clist_gset:Nn`. These functions are documented on page [121](#).)

**\clist\_lput\_gset:NV** Everything is based on concatenation after storing in `\l__clist_internal_clist`. This avoids having to worry here about space-trimming and so on.

```

9939 \cs_new_protected:Npn \clist_lput_left:Nn
9940 { __clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
9941 \cs_new_protected:Npn \clist_gput_left:Nn

```

**\clist\_lput\_gset:Nx**

**\clist\_lput\_gset:c**

**\clist\_lput\_gset:cw**

**\clist\_lput\_gset:cw**

**\clist\_lput\_left:cx**

**\clist\_gput\_left:Nn**

**\clist\_gput\_left:NV**

**\clist\_gput\_left:No**

**\clist\_gput\_left:Nx**

**\clist\_gput\_left:cn**

**\clist\_gput\_left:cV**

**\clist\_gput\_left:co**

```

9942 { __clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
9943 \cs_new_protected:Npn __clist_put_left:NNNn #1#2#3#4
9944 {
9945 #2 \l__clist_internal_clist {#4}
9946 #1 #3 \l__clist_internal_clist #3
9947 }
9948 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
9949 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
9950 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
9951 \cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }

```

(End definition for `\clist_put_left:Nn`, `\clist_gput_left:Nn`, and `\__clist_put_left:NNNn`. These functions are documented on page 121.)

```

\clist_put_right:Nn
\clist_put_right:NV
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cN
\clist_put_right:cV
\clist_put_right:co
\clist_put_right:cx
\clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cN
\clist_gput_right:cV
\clist_gput_right:co
\clist_gput_right:cx
__clist_put_right:NNNn

```

```

9952 \cs_new_protected:Npn \clist_put_right:Nn
9953 { __clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
9954 \cs_new_protected:Npn \clist_gput_right:Nn
9955 { __clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
9956 \cs_new_protected:Npn __clist_put_right:NNNn #1#2#3#4
9957 {
9958 #2 \l__clist_internal_clist {#4}
9959 #1 #3 \l__clist_internal_clist
9960 }
9961 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
9962 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
9963 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
9964 \cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }

```

(End definition for `\clist_put_right:Nn`, `\clist_gput_right:Nn`, and `\__clist_put_right:NNNn`. These functions are documented on page 121.)

## 16.4 Comma lists as stacks

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma. No need to trim spaces as comma-list *variables* are assumed to have “cleaned-up” items. (Note that grabbing a comma-delimited item removes an outer pair of braces if present, exactly as needed to uncover the underlying item.)

```

9965 \cs_new_protected:Npn \clist_get:NN #1#2
9966 {
9967 \if_meaning:w #1 \c_empty_clist
9968 \tl_set:Nn #2 { \q_no_value }
9969 \else:
9970 \exp_after:wN __clist_get:wN #1 , \s__clist_stop #2
9971 \fi:
9972 }
9973 \cs_new_protected:Npn __clist_get:wN #1 , #2 \s__clist_stop #3
9974 { \tl_set:Nn #3 {#1} }
9975 \cs_generate_variant:Nn \clist_get:NN { c }

```

(End definition for `\clist_get:NN` and `\__clist_get:wN`. This function is documented on page 126.)

```

\clist_pop:NN
\clist_pop:cN
\clist_gpop:NN
\clist_gpop:cN
__clist_pop:NNN
__clist_pop:wwNNN
__clist_pop:wN

```

An empty clist leads to `\q_no_value`, otherwise grab until the first comma and assign to the variable. The second argument of `\__clist_pop:wwNNN` is a comma list ending in a

comma and `\s__clist_mark`, unless the original clist contained exactly one item: then the argument is just `\s__clist_mark`. The next auxiliary picks either `\exp_not:n` or `\use_none:n` as #2, ensuring that the result can safely be an empty comma list.

```

9976 \cs_new_protected:Npn \clist_pop:NN
9977 { __clist_pop>NNN __kernel_tl_set:Nx }
9978 \cs_new_protected:Npn \clist_gpop:NN
9979 { __clist_pop>NNN __kernel_tl_gset:Nx }
9980 \cs_new_protected:Npn __clist_pop>NNN #1#2#3
9981 {
9982 \if_meaning:w #2 \c_empty_clist
9983 \tl_set:Nn #3 { \q_no_value }
9984 \else:
9985 \exp_after:wN __clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
9986 \fi:
9987 }
9988 \cs_new_protected:Npn __clist_pop:wwNNN #1 , #2 \s__clist_stop #3#4#5
9989 {
9990 \tl_set:Nn #5 {#1}
9991 #3 #4
9992 {
9993 __clist_pop:wN \prg_do_nothing:
9994 #2 \exp_not:o
9995 , \s__clist_mark \use_none:n
9996 \s__clist_stop
9997 }
9998 }
9999 \cs_new:Npn __clist_pop:wN #1 , \s__clist_mark #2 #3 \s__clist_stop { #2 {#1} }
10000 \cs_generate_variant:Nn \clist_pop:NN { c }
10001 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End definition for `\clist_pop:NN` and others. These functions are documented on page 126.)

**\clist\_get:NNTF** The same, as branching code: very similar to the above.

```

\clist_get:cNTF
10002 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
\clist_pop:NNTF
10003 {
10004 \if_meaning:w #1 \c_empty_clist
10005 \prg_return_false:
10006 \else:
10007 \exp_after:wN __clist_get:wN #1 , \s__clist_stop #2
10008 \prg_return_true:
10009 \fi:
10010 }
10011 \prg_generate_conditional_variant:Nnn \clist_get:NN { c } { T , F , TF }
10012 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
10013 { __clist_pop_TF>NNN __kernel_tl_set:Nx #1 #2 }
10014 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
10015 { __clist_pop_TF>NNN __kernel_tl_gset:Nx #1 #2 }
10016 \cs_new_protected:Npn __clist_pop_TF>NNN #1#2#3
10017 {
10018 \if_meaning:w #2 \c_empty_clist
10019 \prg_return_false:
10020 \else:
10021 \exp_after:wN __clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
10022 \prg_return_true:

```

```

10023 \fi:
10024 }
10025 \prg_generate_conditional_variant:Nnn \clist_pop:NN { c } { T , F , TF }
10026 \prg_generate_conditional_variant:Nnn \clist_gpop:NN { c } { T , F , TF }

```

(End definition for `\clist_get:NNTF` and others. These functions are documented on page 126.)

|                              |                                                                      |
|------------------------------|----------------------------------------------------------------------|
| <code>\clist_push:Nn</code>  | Pushing to a comma list is the same as adding on the left.           |
| <code>\clist_push:NV</code>  | 10027 <code>\cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn</code>   |
| <code>\clist_push:No</code>  | 10028 <code>\cs_new_eq:NN \clist_push:NV \clist_put_left:NV</code>   |
| <code>\clist_push:Nx</code>  | 10029 <code>\cs_new_eq:NN \clist_push:No \clist_put_left:No</code>   |
| <code>\clist_push:cn</code>  | 10030 <code>\cs_new_eq:NN \clist_push:Nx \clist_put_left:Nx</code>   |
| <code>\clist_push:cV</code>  | 10031 <code>\cs_new_eq:NN \clist_push:cn \clist_put_left:cn</code>   |
| <code>\clist_push:co</code>  | 10032 <code>\cs_new_eq:NN \clist_push:cV \clist_put_left:cV</code>   |
| <code>\clist_push:cx</code>  | 10033 <code>\cs_new_eq:NN \clist_push:co \clist_put_left:co</code>   |
| <code>\clist_gpush:Nn</code> | 10034 <code>\cs_new_eq:NN \clist_push:cx \clist_put_left:cx</code>   |
| <code>\clist_gpush:NV</code> | 10035 <code>\cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn</code> |
| <code>\clist_gpush:No</code> | 10036 <code>\cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV</code> |
| <code>\clist_gpush:Nx</code> | 10037 <code>\cs_new_eq:NN \clist_gpush:No \clist_gput_left:No</code> |
| <code>\clist_gpush:cn</code> | 10038 <code>\cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx</code> |
| <code>\clist_gpush:cV</code> | 10039 <code>\cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn</code> |
| <code>\clist_gpush:co</code> | 10040 <code>\cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV</code> |
| <code>\clist_gpush:cx</code> | 10041 <code>\cs_new_eq:NN \clist_gpush:co \clist_gput_left:co</code> |
|                              | 10042 <code>\cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx</code> |

(End definition for `\clist_push:Nn` and `\clist_gpush:Nn`. These functions are documented on page 127.)

## 16.5 Modifying comma lists

`\l__clist_internal_remove_clist`

```

\l__clist_internal_remove_seq
10043 \clist_new:N \l__clist_internal_remove_clist
10044 \seq_new:N \l__clist_internal_remove_seq

```

(End definition for `\l__clist_internal_remove_clist` and `\l__clist_internal_remove_seq`.)

|                                            |                                                                                          |
|--------------------------------------------|------------------------------------------------------------------------------------------|
| <code>\clist_remove_duplicates:N</code>    | Removing duplicates means making a new list then copying it.                             |
| <code>\clist_remove_duplicates:c</code>    | 10045 <code>\cs_new_protected:Npn \clist_remove_duplicates:N</code>                      |
| <code>\clist_gremove_duplicates:N</code>   | 10046 { <code>\__clist_remove_duplicates&gt;NN \clist_set_eq:NN</code> }                 |
| <code>\clist_gremove_duplicates:c</code>   | 10047 <code>\cs_new_protected:Npn \clist_gremove_duplicates:N</code>                     |
| <code>\__clist_remove_duplicates:NN</code> | 10048 { <code>\__clist_remove_duplicates&gt;NN \clist_gset_eq:NN</code> }                |
|                                            | 10049 <code>\cs_new_protected:Npn \__clist_remove_duplicates:NN #1#2</code>              |
|                                            | {                                                                                        |
|                                            | 10050 <code>\clist_clear:N \l__clist_internal_remove_clist</code>                        |
|                                            | 10051 <code>\clist_map_inline:Nn #2</code>                                               |
|                                            | {                                                                                        |
|                                            | 10052 <code>\clist_if_in:NnF \l__clist_internal_remove_clist {##1}</code>                |
|                                            | 10053         { <code>\clist_put_right:Nn \l__clist_internal_remove_clist {##1}</code> } |
|                                            | }                                                                                        |
|                                            | 10054     #1 #2 <code>\l__clist_internal_remove_clist</code>                             |
|                                            | }                                                                                        |
|                                            | 10055 <code>\cs_generate_variant:Nn \clist_remove_duplicates:N { c }</code>              |
|                                            | 10056 <code>\cs_generate_variant:Nn \clist_gremove_duplicates:N { c }</code>             |

(End definition for `\clist_remove_duplicates:N`, `\clist_gremove_duplicates:N`, and `\_clist_remove_duplicates>NN`. These functions are documented on page 122.)

```
\clist_remove_all:Nn
\clist_remove_all:cn
\clist_gremove_all:Nn
\clist_gremove_all:cn
_clist_remove_all:NNNn
__clist_remove_all:w
__clist_remove_all:
```

The method used here for safe items is very similar to `\tl_replace_all:Nnn`. However, if the item contains commas or leading/trailing spaces, or is empty, or consists of a single brace group, we know that it can only appear within braces so the code would fail; instead just convert to a sequence and do the removal with `\bseq` code (it involves somewhat elaborate code to do most of the work expandably but the final token list comparisons non-expandably).

For “safe” items, build a function delimited by the `<item>` that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the `<item>`. The loop is controlled by the argument grabbed by `\_\_clist_remove_all:w`: when the item was found, the `\s_\_clist_mark` delimiter used is the one inserted by `\_\_clist_tmp:w`, and `\_\_clist_use_none_delimit_by_s_stop:w` is deleted. At the end, the final `<item>` is grabbed, and the argument of `\_\_clist_tmp:w` contains `\s_\_clist_mark`: in that case, `\_\_clist_remove_all:w` removes the second `\s_\_clist_mark` (inserted by `\_\_clist_tmp:w`), and lets `\_\_clist_use_none_delimit_by_s_stop:w` act.

No brace is lost because items are always grabbed with a leading comma. The result of the first assignment has an extra leading comma, which we remove in a second assignment. Two exceptions: if the clist lost all of its elements, the result is empty, and we shouldn’t remove anything; if the clist started up empty, the first step happens to turn it into a single comma, and the second step removes it.

```
10061 \cs_new_protected:Npn \clist_remove_all:Nn
10062 { __clist_remove_all:NNNn \clist_set_from_seq:NN __kernel_tl_set:Nx }
10063 \cs_new_protected:Npn \clist_gremove_all:Nn
10064 { __clist_remove_all:NNNn \clist_gset_from_seq:NN __kernel_tl_gset:Nx }
10065 \cs_new_protected:Npn __clist_remove_all:NNNn #1#2#3#4
10066 {
10067 __clist_if_wrap:nTF {#4}
10068 {
10069 \seq_set_from_clist:NN \l__clist_internal_remove_seq #3
10070 \seq_remove_all:Nn \l__clist_internal_remove_seq {#4}
10071 #1 #3 \l__clist_internal_remove_seq
10072 }
10073 {
10074 \cs_set:Npn __clist_tmp:w ##1 , #4 ,
10075 {
10076 ##1
10077 , \s__clist_mark , __clist_use_none_delimit_by_s_stop:w ,
10078 __clist_remove_all:
10079 }
10080 #2 #3
10081 {
10082 \exp_after:wN __clist_remove_all:
10083 #3 , \s__clist_mark , #4 , \s__clist_stop
10084 }
10085 \clist_if_empty:NF #3
10086 {
10087 #2 #3
10088 {
10089 \exp_args:No \exp_not:o
```

```

10090 { \exp_after:wN \use_none:n #3 }
10091 }
10092 }
10093 }
10094 }
10095 \cs_new:Npn __clist_remove_all:
10096 { \exp_after:wN __clist_remove_all:w __clist_tmp:w , }
10097 \cs_new:Npn __clist_remove_all:w #1 , \s__clist_mark , #2 , { \exp_not:n {#1} }
10098 \cs_generate_variant:Nn \clist_remove_all:Nn { c }
10099 \cs_generate_variant:Nn \clist_gremove_all:Nn { c }

```

(End definition for `\clist_remove_all:Nn` and others. These functions are documented on page 122.)

**\clist\_reverse:N** Use `\clist_reverse:n` in an x-expanding assignment. The extra work that `\clist_reverse:n` does to preserve braces and spaces would not be needed for the well-controlled case of N-type comma lists, but the slow-down is not too bad.

**\clist\_greverse:c**

```

10100 \cs_new_protected:Npn \clist_reverse:N #1
10101 { __kernel_tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
10102 \cs_new_protected:Npn \clist_greverse:N #1
10103 { __kernel_tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
10104 \cs_generate_variant:Nn \clist_reverse:N { c }
10105 \cs_generate_variant:Nn \clist_greverse:N { c }

```

(End definition for `\clist_reverse:N` and `\clist_greverse:N`. These functions are documented on page 122.)

**\clist\_reverse:n** The reversed token list is built one item at a time, and stored between `\s__clist_stop` and `\s__clist_mark`, in the form of ? followed by zero or more instances of “`\item`”,. We start from a comma list “`\item_1`, ..., `\item_n`”. During the loop, the auxiliary `\__clist_reverse:wwNww` receives “?`\item_i`” as #1, “`\item_{i+1}`, ..., `\item_n`” as #2, `\__clist_reverse:wwNww` as #3, what remains until `\s__clist_stop` as #4, and “`\item_{i-1}`, ..., `\item_1`,” as #5. The auxiliary moves #1 just before #5, with a comma, and calls itself (#3). After the last item is moved, `\__clist_reverse:wwNww` receives “`\s__clist_mark \__clist_reverse:wwNww !`” as its argument #1, thus `\__clist_reverse_end:ww` as its argument #3. This second auxiliary cleans up until the marker !, removes the trailing comma (introduced when the first item was moved after `\s__clist_stop`), and leaves its argument #1 within `\exp_not:n`. There is also a need to remove a leading comma, hence `\exp_not:o` and `\use_none:n`.

```

10106 \cs_new:Npn \clist_reverse:n #1
10107 {
10108 __clist_reverse:wwNww ? #1 ,
10109 \s__clist_mark __clist_reverse:wwNww ! ,
10110 \s__clist_mark __clist_reverse_end:ww
10111 \s__clist_stop ? \s__clist_mark
10112 }
10113 \cs_new:Npn __clist_reverse:wwNww
10114 #1 , #2 \s__clist_mark #3 #4 \s__clist_stop ? #5 \s__clist_mark
10115 { #3 ? #2 \s__clist_mark #3 #4 \s__clist_stop #1 , #5 \s__clist_mark }
10116 \cs_new:Npn __clist_reverse_end:ww #1 ! #2 , \s__clist_mark
10117 { \exp_not:o { \use_none:n #2 } }

```

(End definition for `\clist_reverse:n`, `\__clist_reverse:wwNww`, and `\__clist_reverse_end:ww`. This function is documented on page 122.)

`\clist_sort:Nn` Implemented in `\l3sort`.  
`\clist_sort:cn`  
`\clist_gsort:Nn`  
`\clist_gsort:cn`  
*(End definition for `\clist_sort:Nn` and `\clist_gsort:Nn`. These functions are documented on page 122.)*

## 16.6 Comma list conditionals

`\clist_if_empty_p:N` Simple copies from the token list variable material.  
`\clist_if_empty_p:c`  
`\clist_if_empty:NTF`  
`\clist_if_empty:cTF`  
`\prg_new_eq_conditional:NNN \clist_if_empty:N \tl_if_empty:N`  
`{ p , T , F , TF }`  
`\prg_new_eq_conditional:NNN \clist_if_empty:c \tl_if_empty:c`  
`{ p , T , F , TF }`  
*(End definition for `\clist_if_empty:NTF`. This function is documented on page 123.)*

`\clist_if_empty_p:n`  
`\clist_if_empty:nTF`  
`\__clist_if_empty_n:w`  
`\__clist_if_empty_n:wNw`  
As usual, we insert a token (here `?`) before grabbing any argument: this avoids losing braces. The argument of `\tl_if_empty:oTF` is empty if `#1` is `?` followed by blank spaces (besides, this particular variant of the emptiness test is optimized). If the item of the comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second auxiliary grabs `\prg_return_false:` as `#2`, unless every item in the comma list was blank and the loop actually got broken by the trailing `\s__clist_mark \prg_return_false:` item.

```

10122 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
10123 {
10124 __clist_if_empty_n:w ? #1
10125 , \s__clist_mark \prg_return_false:
10126 , \s__clist_mark \prg_return_true:
10127 \s__clist_stop
10128 }
10129 \cs_new:Npn __clist_if_empty_n:w #1 ,
10130 {
10131 \tl_if_empty:oTF { \use_none:nn #1 ? }
10132 { __clist_if_empty_n:w ? }
10133 { __clist_if_empty_n:wNw }
10134 }
10135 \cs_new:Npn __clist_if_empty_n:wNw #1 \s__clist_mark #2#3 \s__clist_stop {#2}

```

*(End definition for `\clist_if_empty:nTF`, `\__clist_if_empty_n:w`, and `\__clist_if_empty_n:wNw`. This function is documented on page 123.)*

`\clist_if_in:NnTF` For “safe” items, we simply surround the comma list, and the item, with commas, then use the same code as for `\tl_if_in:Nn`. For “unsafe” items we follow the same route as `\seq_if_in:Nn`, mapping through the list a comparison function. If found, return `true` and remove `\prg_return_false:`.  
`\clist_if_in:NVT`  
`\clist_if_in:NVTF`  
`\clist_if_in:NoTF`  
`\clist_if_in:cNTF`  
`\clist_if_in:cVTF`  
`\clist_if_in:cOTF`  
`\clist_if_in:nnTF`  
`\clist_if_in:nVT`  
`\clist_if_in:notF`  
`\__clist_if_in_return:nnN`  
`\prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }`  
`{`  
`\exp_args:No \__clist_if_in_return:nnN #1 {#2} #1`  
`}`  
`\prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }`  
`{`  
`\clist_set:Nn \l__clist_internal_clist {#1}`  
`\exp_args:No \__clist_if_in_return:nnN \l__clist_internal_clist {#2}`  
`\l__clist_internal_clist`  
`}`

```

10146 \cs_new_protected:Npn __clist_if_in_return:nnN #1#2#3
10147 {
10148 __clist_if_wrap:nTF {#2}
10149 {
10150 \cs_set:Npx __clist_tmp:w ##1
10151 {
10152 \exp_not:N \tl_if_eq:nnT {##1}
10153 \exp_not:n
10154 {
10155 {#2}
10156 { \clist_map_break:n { \prg_return_true: \use_none:n } }
10157 }
10158 }
10159 \clist_map_function:NN #3 __clist_tmp:w
10160 \prg_return_false:
10161 }
10162 {
10163 \cs_set:Npn __clist_tmp:w ##1 ,#2, { }
10164 \tl_if_empty:oTF
10165 { __clist_tmp:w ,#1, {} {} ,#2, }
10166 { \prg_return_false: } { \prg_return_true: }
10167 }
10168 }
10169 \prg_generate_conditional_variant:Nnn \clist_if_in:Nn
10170 { NV , No , c , cV , co } { T , F , TF }
10171 \prg_generate_conditional_variant:Nnn \clist_if_in:nn
10172 { nV , no } { T , F , TF }

```

(End definition for `\clist_if_in:NnTF`, `\clist_if_in:nnTF`, and `\__clist_if_in_return:nnN`. These functions are documented on page 123.)

## 16.7 Mapping to comma lists

`\clist_map_function:NN`  
`\clist_map_function:cN`  
`\__clist_map_function:Nw`

If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing one comma-delimited item at a time. The end is marked by `\q_clist_recursion_tail`. The auxiliary function `\__clist_map_function:Nw` is also used in `\clist_map_inline:Nn`.

```

10173 \cs_new:Npn \clist_map_function:NN #1#
10174 {
10175 \clist_if_empty:NF #1
10176 {
10177 \exp_last_unbraced:NNo __clist_map_function:Nw #2 #1
10178 , \q_clist_recursion_tail ,
10179 \prg_break_point:Nn \clist_map_break: { }
10180 }
10181 }
10182 \cs_new:Npn __clist_map_function:Nw #1#2 ,
10183 {
10184 __clist_if_recursion_tail_break:nN {#2} \clist_map_break:
10185 #1 {#2}
10186 __clist_map_function:Nw #1
10187 }
10188 \cs_generate_variant:Nn \clist_map_function:NN { c }

```

(End definition for `\clist_map_function:N` and `\_clist_map_function:Nw`. This function is documented on page 123.)

```
\clist_map_function:nN
__clist_map_function_n:Nn
__clist_map_unbrace:Nw
10189 \cs_new:Npn \clist_map_function:nN #1#2
10190 {
10191 \exp_after:wN __clist_map_function_n:Nn \exp_after:wN #2
10192 \exp:w __clist_trim_next:w \prg_do_nothing: #1 , \q__clist_recursion_tail ,
10193 \prg_break_point:Nn \clist_map_break: { }
10194 }
10195 \cs_new:Npn __clist_map_function_n:Nn #1 #2
10196 {
10197 __clist_if_recursion_tail_break:nN {#2} \clist_map_break:
10198 __clist_map_unbrace:Nw #1 #2,
10199 \exp_after:wN __clist_map_function_n:Nn \exp_after:wN #1
10200 \exp:w __clist_trim_next:w \prg_do_nothing:
10201 }
10202 \cs_new:Npn __clist_map_unbrace:Nw #1 #2, { #1 {#2} }
```

(End definition for `\clist_map_function:nN`, `\__clist_map_function_n:Nn`, and `\__clist_map_unbrace:Nw`. This function is documented on page 123.)

`\clist_map_inline:Nn`  
`\clist_map_inline:cn`  
`\clist_map_inline:nn`

Inline mapping is done by creating a suitable function “on the fly”: this is done globally to avoid any issues with TeX’s groups. We use a different function for each level of nesting.

Since the mapping is non-expandable, we can perform the space-trimming needed by the `n` version simply by storing the comma-list in a variable. We don’t need a different comma-list for each nesting level: the comma-list is expanded before the mapping starts.

```
10203 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
10204 {
10205 \clist_if_empty:NF #1
10206 {
10207 \int_gincr:N \g__kernel_prg_map_int
10208 \cs_gset_protected:cpn
10209 { __clist_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
10210 \exp_last_unbraced:Nco __clist_map_function:Nw
10211 { __clist_map_ \int_use:N \g__kernel_prg_map_int :w }
10212 #1 , \q__clist_recursion_tail ,
10213 \prg_break_point:Nn \clist_map_break:
10214 { \int_gdecr:N \g__kernel_prg_map_int }
10215 }
10216 }
10217 \cs_new_protected:Npn \clist_map_inline:nn #1
10218 {
10219 \clist_set:Nn \l__clist_internal_clist {#1}
10220 \clist_map_inline:Nn \l__clist_internal_clist
10221 }
10222 \cs_generate_variant:Nn \clist_map_inline:Nn { c }
```

(End definition for `\clist_map_inline:Nn` and `\clist_map_inline:nn`. These functions are documented on page 124.)

**\clist\_map\_variable:Nn**  
**\clist\_map\_variable:cNn**  
**\clist\_map\_variable:nNn**  
**\\_clist\_map\_variable:Nnw**

As for other comma-list mappings, filter out the case of an empty list. Same approach as **\clist\_map\_function:Nn**, additionally we store each item in the given variable. As for inline mappings, space trimming for the **n** variant is done by storing the comma list in a variable. The quark test is done before assigning the item to the variable: this avoids storing a quark which the user wouldn't expect. The strange **\use:n** avoids unlikely problems when #2 would contain **\q\_clist\_recursion\_stop**.

```

10223 \cs_new_protected:Npn \clist_map_variable:Nn #1#2#3
10224 {
10225 \clist_if_empty:NF #1
10226 {
10227 \exp_args:Nno \use:nn
10228 { _clist_map_variable:Nnw #2 {#3} }
10229 #1
10230 , \q_clist_recursion_tail , \q_clist_recursion_stop
10231 \prg_break_point:Nn \clist_map_break: { }
10232 }
10233 }
10234 \cs_new_protected:Npn \clist_map_variable:nNn #1
10235 {
10236 \clist_set:Nn \l__clist_internal_clist {#1}
10237 \clist_map_variable:NNn \l__clist_internal_clist
10238 }
10239 \cs_new_protected:Npn _clist_map_variable:Nnw #1#2#3,
10240 {
10241 _clist_if_recursion_tail_stop:n {#3}
10242 \tl_set:Nn #1 {#3}
10243 \use:n {#2}
10244 _clist_map_variable:Nnw #1 {#2}
10245 }
10246 \cs_generate_variant:Nn \clist_map_variable:NNn { c }

(End definition for \clist_map_variable:Nn, \clist_map_variable:nNn, and _clist_map_variable:Nnw. These functions are documented on page 124.)

```

**\clist\_map\_break:**  
**\clist\_map\_break:n**

The break statements use the general **\prg\_map\_break:Nn** mechanism.

```

10247 \cs_new:Npn \clist_map_break:
10248 { \prg_map_break:Nn \clist_map_break: { } }
10249 \cs_new:Npn \clist_map_break:n
10250 { \prg_map_break:Nn \clist_map_break: }

(End definition for \clist_map_break: and \clist_map_break:n. These functions are documented on page 124.)

```

**\clist\_count:N**  
**\clist\_count:c**  
**\clist\_count:n**  
**\\_clist\_count:n**  
**\\_clist\_count:w**

Counting the items in a comma list is done using the same approach as for other token count functions: turn each entry into a **+1** then use integer evaluation to actually do the mathematics. In the case of an **n**-type comma-list, we could of course use **\clist\_map\_function:NN**, but that is very slow, because it carefully removes spaces. Instead, we loop manually, and skip blank items (but not {}, hence the extra spaces).

```

10251 \cs_new:Npn \clist_count:N #1
10252 {
10253 \int_eval:n
10254 {
10255 0
10256 \clist_map_function>NN #1 _clist_count:n

```

```

10257 }
10258 }
10259 \cs_generate_variant:Nn \clist_count:N { c }
10260 \cs_new:Npx \clist_count:n #1
10261 {
10262 \exp_not:N \int_eval:n
10263 {
10264 0
10265 \exp_not:N __clist_count:w \c_space_tl
10266 #1 \exp_not:n { , \q__clist_recursion_tail , \q__clist_recursion_stop }
10267 }
10268 }
10269 \cs_new:Npn __clist_count:n #1 { + 1 }
10270 \cs_new:Npx __clist_count:w #1 ,
10271 {
10272 \exp_not:n { \exp_args:Nf __clist_if_recursion_tail_stop:n } {#1}
10273 \exp_not:N \tl_if_blank:nF {#1} { + 1 }
10274 \exp_not:N __clist_count:w \c_space_tl
10275 }

```

(End definition for `\clist_count:N` and others. These functions are documented on page 125.)

## 16.8 Using comma lists

```

\clist_use:Nnnn
\clist_use:cnnn
__clist_use:wwn
__clist_use:nwwwnwn
__clist_use:nwn
\clist_use:Nn
\clist_use:cn

```

First check that the variable exists. Then count the items in the comma list. If it has none, output nothing. If it has one item, output that item, brace stripped (note that space-trimming has already been done when the comma list was assigned). If it has two, place the *<separator between two>* in the middle.

Otherwise, `\__clist_use:nwwwnwn` takes the following arguments; 1: a *<separator>*, 2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6: a *<continuation>* function (`use_ii` or `use_iii` with its *<separator>* argument), 7: junk, and 8: the temporary result, which is built in a brace group following `\q__clist_stop`. The *<separator>* and the first of the three items are placed in the result, then we use the *<continuation>*, placing the remaining two items after it. When we begin this loop, the three items really belong to the comma list, the first `\q__clist_mark` is taken as a delimiter to the `use_ii` function, and the continuation is `use_ii` itself. When we reach the last two items of the original token list, `\q__clist_mark` is taken as a third item, and now the second `\q__clist_mark` serves as a delimiter to `use_ii`, switching to the other *<continuation>*, `use_iii`, which uses the *<separator between final two>*.

```

10276 \cs_new:Npn \clist_use:Nnnn #1#2#3#4
10277 {
10278 \clist_if_exist:NTF #1
10279 {
10280 \int_case:nnF { \clist_count:N #1 }
10281 {
10282 { 0 } { }
10283 { 1 } { \exp_after:wN __clist_use:wwn #1 , , { } }
10284 { 2 } { \exp_after:wN __clist_use:wwn #1 , {#2} }
10285 }
10286 {
10287 \exp_after:wN __clist_use:nwwwnwn
10288 \exp_after:wN { \exp_after:wN } #1 ,
10289 \s__clist_mark , { __clist_use:nwwwnwn {#3} }

```

```

10290 \s__clist_mark , { __clist_use:nwnn {#4} }
10291 \s__clist_stop { }
10292 }
10293 }
10294 {
10295 __kernel_msg_expandable_error:nnn
10296 { kernel } { bad-variable } {#1}
10297 }
10298 }
10299 \cs_generate_variant:Nn \clist_use:Nnnn { c }
10300 \cs_new:Npn __clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
10301 \cs_new:Npn __clist_use:nwwwnwn
10302 #1#2 , #3 , #4 , #5 \s__clist_mark , #6#7 \s__clist_stop #8
10303 { #6 {#3} , {#4} , #5 \s__clist_mark , {#6} #7 \s__clist_stop { #8 #1 #2 } }
10304 \cs_new:Npn __clist_use:nwnn #1#2 , #3 \s__clist_stop #4
10305 { \exp_not:n { #4 #1 #2 } }
10306 \cs_new:Npn \clist_use:Nn #1#2
10307 { \clist_use:Nnnn #1 {#2} {#2} {#2} }
10308 \cs_generate_variant:Nn \clist_use:Nn { c }

(End definition for \clist_use:Nnnn and others. These functions are documented on page 125.)
```

## 16.9 Using a single item

\clist\_item:Nn

\clist\_item:cn

\\_\_clist\_item:nnnN

\\_\_clist\_item:ffoN

\\_\_clist\_item:ffnN

\\_\_clist\_item\_N\_loop:nw

To avoid needing to test the end of the list at each step, we first compute the  $\langle length \rangle$  of the list. If the item number is 0, less than  $-\langle length \rangle$ , or more than  $\langle length \rangle$ , the result is empty. If it is negative, but not less than  $-\langle length \rangle$ , add  $\langle length \rangle + 1$  to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached 1, otherwise, decrease the counter and repeat.

```

10309 \cs_new:Npn \clist_item:Nn #1#2
10310 {
10311 __clist_item:ffoN
10312 { \clist_count:N #1 }
10313 { \int_eval:n {#2} }
10314 #1
10315 __clist_item_N_loop:nw
10316 }
10317 \cs_new:Npn __clist_item:nnnN #1#2#3#4
10318 {
10319 \int_compare:nNnTF {#2} < 0
10320 {
10321 \int_compare:nNnTF {#2} < { - #1 }
10322 { __clist_use_none_delimit_by_s_stop:w }
10323 { \exp_args:Nf #4 { \int_eval:n { #2 + 1 + #1 } } }
10324 }
10325 {
10326 \int_compare:nNnTF {#2} > {#1}
10327 { __clist_use_none_delimit_by_s_stop:w }
10328 { #4 {#2} }
10329 }
10330 { } , #3 , \s__clist_stop
10331 }
10332 \cs_generate_variant:Nn __clist_item:nnnN { ffo, ff }
10333 \cs_new:Npn __clist_item_N_loop:nw #1 #2,
```

```

10334 {
10335 \int_compare:nNnTF {#1} = 0
10336 { __clist_use_i_delimit_by_s_stop:nw { \exp_not:n {#2} } }
10337 { \exp_args:Nf __clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
10338 }
10339 \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End definition for `\clist_item:Nn`, `\__clist_item:nnN`, and `\__clist_item_N_loop:nw`. This function is documented on page 127.)

```

\clist_item:nn
__clist_item_n:nw
__clist_item_n_loop:nw
__clist_item_n_end:n
__clist_item_n_strip:n
__clist_item_n_strip:w

```

This starts in the same way as `\clist_item:Nn` by counting the items of the comma list. The final item should be space-trimmed before being brace-stripped, hence we insert a couple of odd-looking `\prg_do_nothing:` to avoid losing braces. Blank items are ignored.

```

10340 \cs_new:Npn \clist_item:nn #1#2
10341 {
10342 __clist_item:ffnN
10343 { \clist_count:n {#1} }
10344 { \int_eval:n {#2} }
10345 {#1}
10346 __clist_item_n:nw
10347 }
10348 \cs_new:Npn __clist_item_n:nw #1
10349 { __clist_item_n_loop:nw {#1} \prg_do_nothing: }
10350 \cs_new:Npn __clist_item_n_loop:nw #1 #2,
10351 {
10352 \exp_args:No \tl_if_blank:nTF {#2}
10353 { __clist_item_n_loop:nw {#1} \prg_do_nothing: }
10354 {
10355 \int_compare:nNnTF {#1} = 0
10356 { \exp_args:No __clist_item_n_end:n {#2} }
10357 {
10358 \exp_args:Nf __clist_item_n_loop:nw
10359 { \int_eval:n { #1 - 1 } }
10360 \prg_do_nothing:
10361 }
10362 }
10363 }
10364 \cs_new:Npn __clist_item_n_end:n #1 #2 \s__clist_stop
10365 { \tl_trim_spaces_apply:nN {#1} __clist_item_n_strip:n }
10366 \cs_new:Npn __clist_item_n_strip:n #1 { __clist_item_n_strip:w #1 , }
10367 \cs_new:Npn __clist_item_n_strip:w #1 , { \exp_not:n {#1} }

```

(End definition for `\clist_item:nn` and others. This function is documented on page 127.)

```

\clist_rand_item:n
\clist_rand_item:N
\clist_rand_item:c
__clist_rand_item:nn

```

The N-type function is not implemented through the n-type function for efficiency: for instance comma-list variables do not require space-trimming of their items. Even testing for emptiness of an n-type comma-list is slow, so we count items first and use that both for the emptiness test and the pseudo-random integer. Importantly, `\clist_item:Nn` and `\clist_item:nn` only evaluate their argument once.

```

10368 \cs_new:Npn \clist_rand_item:n #1
10369 { \exp_args:Nf __clist_rand_item:nn { \clist_count:n {#1} } {#1} }
10370 \cs_new:Npn __clist_rand_item:nn #1#2
10371 {
10372 \int_compare:nNnF {#1} = 0

```

```

10373 { \clist_item:nn {#2} { \int_rand:nn { 1 } {#1} } }
10374 }
10375 \cs_new:Npn \clist_rand_item:N #1
10376 {
10377 \clist_if_empty:NF #1
10378 { \clist_item:Nn #1 { \int_rand:nn { 1 } { \clist_count:N #1 } } }
10379 }
10380 \cs_generate_variant:Nn \clist_rand_item:N { c }

(End definition for \clist_rand_item:n, \clist_rand_item:N, and __clist_rand_item:nn. These functions are documented on page 127.)

```

## 16.10 Viewing comma lists

\clist\_show:N Apply the general \\_\_kernel\_chk\_defined:NT and \msg\_show:nnnnnn.

```

\clist_show:c 10381 \cs_new_protected:Npn \clist_show:N { __clist_show:NN \msg_show:nnxxxx }
\clist_log:N 10382 \cs_generate_variant:Nn \clist_show:N { c }
\clist_log:c 10383 \cs_new_protected:Npn \clist_log:N { __clist_show:NN \msg_log:nnxxxx }
__clist_show:NN 10384 \cs_generate_variant:Nn \clist_log:N { c }
10385 \cs_new_protected:Npn __clist_show:NN #1#2
10386 {
10387 __kernel_chk_defined:NT #2
10388 {
10389 #1 { LaTeX/kernel } { show-clist }
10390 { \token_to_str:N #2 }
10391 { \clist_map_function:NN #2 \msg_show_item:n }
10392 { } { }
10393 }
10394 }

```

(End definition for \clist\_show:N, \clist\_log:N, and \\_\_clist\_show:NN. These functions are documented on page 127.)

\clist\_show:n A variant of the above: no existence check, empty first argument for the message.

```

\clist_log:n 10395 \cs_new_protected:Npn \clist_show:n { __clist_show:Nn \msg_show:nnxxxx }
__clist_show:Nn 10396 \cs_new_protected:Npn \clist_log:n { __clist_show:Nn \msg_log:nnxxxx }
10397 \cs_new_protected:Npn __clist_show:Nn #1#2
10398 {
10399 #1 { LaTeX/kernel } { show-clist }
10400 { } { \clist_map_function:nN {#2} \msg_show_item:n } { } { }
10401 }

```

(End definition for \clist\_show:n, \clist\_log:n, and \\_\_clist\_show:Nn. These functions are documented on page 128.)

## 16.11 Scratch comma lists

\l\_tmpa\_clist Temporary comma list variables.

```

\l_tmpb_clist 10402 \clist_new:N \l_tmpa_clist
\g_tmpa_clist 10403 \clist_new:N \l_tmpb_clist
\g_tmpb_clist 10404 \clist_new:N \g_tmpa_clist
10405 \clist_new:N \g_tmpb_clist

```

(End definition for \l\_tmpa\_clist and others. These variables are documented on page 128.)

```
10406 </package>
```

## 17 I3token implementation

```
10407 {*package}
10408 {*tex}
10409 <@=char>
```

### 17.1 Internal auxiliaries

\s\_\_char\_stop Internal scan mark.

```
10410 \scan_new:N \s__char_stop
```

(End definition for \s\_\_char\_stop.)

\q\_\_char\_no\_value Internal recursion quarks.

```
10411 \quark_new:N \q__char_no_value
```

(End definition for \q\_\_char\_no\_value.)

\\_char\_quark\_if\_no\_value\_p:N Functions to query recursion quarks.

```
10412 _kernel_quark_new_conditional:Nn _char_quark_if_no_value:N { TF }
```

(End definition for \\_char\_quark\_if\_no\_value:NTF.)

### 17.2 Manipulating and interrogating character tokens

Simple wrappers around the primitives.

```
10413 \cs_new_protected:Npn \char_set_catcode:nn #1#2
10414 { \tex_catcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
10415 \cs_new:Npn \char_value_catcode:n #1
10416 { \tex_the:D \tex_catcode:D \int_eval:n {#1} \exp_stop_f: }
10417 \cs_new_protected:Npn \char_show_value_catcode:n #1
10418 { \exp_args:Nf \tl_show:n { \char_value_catcode:n {#1} } }
```

(End definition for \char\_set\_catcode:nn, \char\_value\_catcode:n, and \char\_show\_value\_catcode:n.  
These functions are documented on page 132.)

\char\_set\_catcode\_escape:N

```
10419 \cs_new_protected:Npn \char_set_catcode_escape:N #1
```

\char\_set\_catcode\_group\_begin:N { \char\_set\_catcode:nn { '#1 } { 0 } }

\char\_set\_catcode\_group\_end:N

```
10420 { \char_set_catcode:nn { '#1 } { 0 } }
```

\char\_set\_catcode\_math\_toggle:N

```
10421 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
```

\char\_set\_catcode\_alignment:N

```
10422 { \char_set_catcode:nn { '#1 } { 1 } }
```

```
10423 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
```

\char\_set\_catcode\_end\_line:N

```
10424 { \char_set_catcode:nn { '#1 } { 2 } }
```

\char\_set\_catcode\_parameter:N

```
10425 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
```

\char\_set\_catcode\_math\_superscript:N

```
10426 { \char_set_catcode:nn { '#1 } { 3 } }
```

\char\_set\_catcode\_math\_subscript:N

```
10427 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
```

\char\_set\_catcode\_ignore:N

```
10428 { \char_set_catcode:nn { '#1 } { 4 } }
```

\char\_set\_catcode\_space:N

```
10429 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
```

\char\_set\_catcode\_letter:N

```
10430 { \char_set_catcode:nn { '#1 } { 5 } }
```

\char\_set\_catcode\_other:N

```
10431 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
```

\char\_set\_catcode\_active:N

```
10432 { \char_set_catcode:nn { '#1 } { 6 } }
```

\char\_set\_catcode\_comment:N

```
10433 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
```

\char\_set\_catcode\_invalid:N

```
10434 { \char_set_catcode:nn { '#1 } { 7 } }
```

```
10435 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
```

```

10436 { \char_set_catcode:nn { '#1 } { 8 } }
10437 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
10438 { \char_set_catcode:nn { '#1 } { 9 } }
10439 \cs_new_protected:Npn \char_set_catcode_space:N #1
10440 { \char_set_catcode:nn { '#1 } { 10 } }
10441 \cs_new_protected:Npn \char_set_catcode_letter:N #1
10442 { \char_set_catcode:nn { '#1 } { 11 } }
10443 \cs_new_protected:Npn \char_set_catcode_other:N #1
10444 { \char_set_catcode:nn { '#1 } { 12 } }
10445 \cs_new_protected:Npn \char_set_catcode_active:N #1
10446 { \char_set_catcode:nn { '#1 } { 13 } }
10447 \cs_new_protected:Npn \char_set_catcode_comment:N #1
10448 { \char_set_catcode:nn { '#1 } { 14 } }
10449 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
10450 { \char_set_catcode:nn { '#1 } { 15 } }

```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 131.)

```

\char_set_catcode_escape:n
\char_set_catcode_group_begin:n
\char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
\char_set_catcode_alignment:n
\char_set_catcode_end_line:n
\char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
\char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n

```

```

10451 \cs_new_protected:Npn \char_set_catcode_escape:n #1
10452 { \char_set_catcode:nn {#1} { 0 } }
10453 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
10454 { \char_set_catcode:nn {#1} { 1 } }
10455 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
10456 { \char_set_catcode:nn {#1} { 2 } }
10457 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
10458 { \char_set_catcode:nn {#1} { 3 } }
10459 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
10460 { \char_set_catcode:nn {#1} { 4 } }
10461 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
10462 { \char_set_catcode:nn {#1} { 5 } }
10463 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
10464 { \char_set_catcode:nn {#1} { 6 } }
10465 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
10466 { \char_set_catcode:nn {#1} { 7 } }
10467 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
10468 { \char_set_catcode:nn {#1} { 8 } }
10469 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
10470 { \char_set_catcode:nn {#1} { 9 } }
10471 \cs_new_protected:Npn \char_set_catcode_space:n #1
10472 { \char_set_catcode:nn {#1} { 10 } }
10473 \cs_new_protected:Npn \char_set_catcode_letter:n #1
10474 { \char_set_catcode:nn {#1} { 11 } }
10475 \cs_new_protected:Npn \char_set_catcode_other:n #1
10476 { \char_set_catcode:nn {#1} { 12 } }
10477 \cs_new_protected:Npn \char_set_catcode_active:n #1
10478 { \char_set_catcode:nn {#1} { 13 } }
10479 \cs_new_protected:Npn \char_set_catcode_comment:n #1
10480 { \char_set_catcode:nn {#1} { 14 } }
10481 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
10482 { \char_set_catcode:nn {#1} { 15 } }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 131.)

```

\char_set_mathcode:nn Pretty repetitive, but necessary!
\char_value_mathcode:n
\char_show_value_mathcode:n
\char_set_lccode:nn
\char_value_lccode:n
\char_show_value_lccode:n
\char_set_uccode:nn
\char_value_uccode:n
\char_show_value_uccode:n
\char_set_sfcode:nn
\char_value_sfcode:n
\char_show_value_sfcode:n

```

10483 \cs\_new\_protected:Npn \char\_set\_mathcode:nn #1#2  
10484 { \tex\_mathcode:D \int\_eval:n {#1} = \int\_eval:n {#2} \exp\_stop\_f: }  
10485 \cs\_new:Npn \char\_value\_mathcode:n #1  
10486 { \tex\_the:D \tex\_mathcode:D \int\_eval:n {#1} \exp\_stop\_f: }  
10487 \cs\_new\_protected:Npn \char\_show\_value\_mathcode:n #1  
10488 { \exp\_args:Nf \tl\_show:n { \char\_value\_mathcode:n {#1} } }  
10489 \cs\_new\_protected:Npn \char\_set\_lccode:nn #1#2  
10490 { \tex\_lccode:D \int\_eval:n {#1} = \int\_eval:n {#2} \exp\_stop\_f: }  
10491 \cs\_new:Npn \char\_value\_lccode:n #1  
10492 { \tex\_the:D \tex\_lccode:D \int\_eval:n {#1} \exp\_stop\_f: }  
10493 \cs\_new\_protected:Npn \char\_show\_value\_lccode:n #1  
10494 { \exp\_args:Nf \tl\_show:n { \char\_value\_lccode:n {#1} } }  
10495 \cs\_new\_protected:Npn \char\_set\_uccode:nn #1#2  
10496 { \tex\_uccode:D \int\_eval:n {#1} = \int\_eval:n {#2} \exp\_stop\_f: }  
10497 \cs\_new:Npn \char\_value\_uccode:n #1  
10498 { \tex\_the:D \tex\_uccode:D \int\_eval:n {#1} \exp\_stop\_f: }  
10499 \cs\_new\_protected:Npn \char\_show\_value\_uccode:n #1  
10500 { \exp\_args:Nf \tl\_show:n { \char\_value\_uccode:n {#1} } }  
10501 \cs\_new\_protected:Npn \char\_set\_sfcode:nn #1#2  
10502 { \tex\_sfcode:D \int\_eval:n {#1} = \int\_eval:n {#2} \exp\_stop\_f: }  
10503 \cs\_new:Npn \char\_value\_sfcode:n #1  
10504 { \tex\_the:D \tex\_sfcode:D \int\_eval:n {#1} \exp\_stop\_f: }  
10505 \cs\_new\_protected:Npn \char\_show\_value\_sfcode:n #1  
10506 { \exp\_args:Nf \tl\_show:n { \char\_value\_sfcode:n {#1} } }

(End definition for `\char_set_mathcode:nn` and others. These functions are documented on page 133.)

```

\l_char_active_seq
\l_char_special_seq

```

Two sequences for dealing with special characters. The first is characters which may be active, the second longer list is for “special” characters more generally. Both lists are escaped so that for example bulk code assignments can be carried out. In both cases, the order is by ASCII character code (as is done in for example `\ExplSyntaxOn`).

```

10507 \seq_new:N \l_char_special_seq
10508 \seq_set_split:Nnn \l_char_special_seq { }
10509 { \ " \# \$ % & \\ ^ _ { } ~ }
10510 \seq_new:N \l_char_active_seq
10511 \seq_set_split:Nnn \l_char_active_seq { }
10512 { " \$ & ^ _ ~ }

```

(End definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 133.)

### 17.3 Creating character tokens

```

\char_set_active_eq:NN
\char_set_active_eq:Nc
\char_gset_active_eq:NN
\char_gset_active_eq:Nc
\char_set_active_eq:nN
\char_gset_active_eq:nN
\char_gset_active_eq:nc

```

Four simple functions with very similar definitions, so set up using an auxiliary. These are similar to LuaTeX’s `\letcharcode` primitive.

```

10513 \group_begin:
10514 \char_set_catcode_active:N \^@%
10515 \cs_set_protected:Npn _char_tmp:nN #1#2
10516 {
10517 \cs_new_protected:cpn { #1 :nN } ##1
10518 {
10519 \group_begin:
10520 \char_set_lccode:nn { '\^@ } { ##1 }

```

```

10521 \tex_lowercase:D { \group_end: #2 ^^@ }
10522 }
10523 \cs_new_protected:cpx { #1 :NN } ##1
10524 { \exp_not:c { #1 : nN } { '##1 } }
10525 }
10526 _char_tmp:nN { char_set_active_eq } \cs_set_eq:NN
10527 _char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
10528 \group_end:
10529 \cs_generate_variant:Nn \char_set_active_eq:NN { Nc }
10530 \cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
10531 \cs_generate_variant:Nn \char_set_active_eq:nN { nc }
10532 \cs_generate_variant:Nn \char_gset_active_eq:nN { nc }

```

(End definition for \char\_set\_active\_eq:NN and others. These functions are documented on page 129.)

\\_char\_int\_to\_roman:w For efficiency in 8-bit engines, we use the faster primitive approach to making roman numerals.

```
10533 \cs_new_eq:NN _char_int_to_roman:w \tex_roman numeral:D
```

(End definition for \\_char\_int\_to\_roman:w.)

### \char\_generate:nn

```
_char_generate_aux:nn
_char_generate_aux:nnw
_char_generate_auxii:nnw
```

```

10534 \cs_new:Npn \char_generate:nn #1#2
10535 {
10536 \exp:w \exp_after:wN _char_generate_aux:w
10537 \int_value:w \int_eval:n {#1} \exp_after:wN ;
10538 \int_value:w \int_eval:n {#2} ;
10539 }
```

Before doing any actual conversion, first some special case filtering. Spaces are out here as LuaTeX emulation only makes normal (charcode 32 spaces). However,  $^{\wedge}\wedge@$  is filtered out separately as that can't be done with macro emulation either, so is flagged up separately. That done, hand off to the engine-dependent part.

```

10540 \cs_new:Npn _char_generate_aux:w #1 ; #2 ;
10541 {
10542 \if_int_compare:w #2 = 10 \exp_stop_f:
10543 \if_int_compare:w #1 = 0 \exp_stop_f:
10544 _kernel_msg_expandable_error:nn { kernel } { char-null-space }
10545 \else:
10546 _kernel_msg_expandable_error:nn { kernel } { char-space }
10547 \fi:
10548 \else:
10549 \if_int_odd:w 0
10550 \if_int_compare:w #2 < 1 \exp_stop_f: 1 \fi:
10551 \if_int_compare:w #2 = 5 \exp_stop_f: 1 \fi:
10552 \if_int_compare:w #2 = 9 \exp_stop_f: 1 \fi:
10553 \if_int_compare:w #2 > 13 \exp_stop_f: 1 \fi: \exp_stop_f:
10554 _kernel_msg_expandable_error:nn { kernel }
10555 { char-invalid-catcode }
10556 \else:
10557 \if_int_odd:w 0

```

```

10558 \if_int_compare:w #1 < 0 \exp_stop_f: 1 \fi:
10559 \if_int_compare:w #1 > \c_max_char_int 1 \fi: \exp_stop_f:
10560 _kernel_msg_expandable_error:nn { kernel }
10561 { char-out-of-range }
10562 \else:
10563 _char_generate_aux:nnw {#1} {#2}
10564 \fi:
10565 \fi:
10566 \fi:
10567 \exp_end:
10568 }
10569 \tl_new:N \l__char_tmp_tl

```

Engine-dependent definitions are now needed for the implementation. For LuaTeX and XeTeX there is engine-level support. They can do cases that macro emulation can't. All of those are filtered out here using a primitive-based boolean expression to avoid fixing the category code of the null character used in the false branch (for 8-bit engines). The final level is the basic definition at the engine level: the arguments here are integers so there is no need to worry about them too much. Older versions of XeTeX cannot generate active characters so we filter that: at some future stage that may change: the slightly odd ordering of auxiliaries reflects that.

```

10570 \group_begin:
10571 \char_set_catcode_active:N \^^L
10572 \cs_set:Npn \^L { }
10573 \char_set_catcode_other:n { 0 }
10574 \if_int_odd:w 0
10575 \sys_if_engine_luatex:T { 1 }
10576 \sys_if_engine_xetex:T { 1 } \exp_stop_f:
10577 \cs_new:Npn _char_generate_aux:nnw #1#2#3 \exp_end:
10578 {
10579 #3
10580 \exp_after:wN \exp_end:
10581 \tex_Ucharcat:D #1 \exp_stop_f: #2 \exp_stop_f:
10582 }
10583 \cs_if_exist:NF \tex_expanded:D
10584 {
10585 \cs_new_eq:NN _char_generate_auxii:nnw _char_generate_aux:nnw
10586 \cs_gset:Npn _char_generate_aux:nnw #1#2#3 \exp_end:
10587 {
10588 #3
10589 \if_int_compare:w #2 = 13 \exp_stop_f:
10590 _kernel_msg_expandable_error:nn { kernel } { char-active }
10591 \else:
10592 _char_generate_auxii:nnw {#1} {#2}
10593 \fi:
10594 \exp_end:
10595 }
10596 }
10597 \else:

```

For engines where \Ucharcat isn't available or emulated, we have to work in macros, and cover only the 8-bit range. The first stage is to build up a `tl` containing `^\0` with each category code that can be accessed in this way, with an error set up for the other cases. This is all done such that it can be quickly accessed using a \if\_case:w low-level

conditional. There are a few things to notice here. As  $\wedge\wedge L$  is `\outer` we need to locally set it to avoid a problem. To get open/close braces into the list, they are set up using `\if_false:` pairing and are then x-type expanded together into the desired form.

```

10598 \tl_set:Nn \l__char_tmp_tl { \exp_not:N \or: }
10599 \char_set_catcode_group_begin:n { 0 } % {
10600 \tl_put_right:Nn \l__char_tmp_tl { \wedge\wedge @ \if_false: } }
10601 \char_set_catcode_group_end:n { 0 }
10602 \tl_put_right:Nn \l__char_tmp_tl { { \fi: \exp_not:N \or: \wedge\wedge @ } % }
10603 \l_kernel_tl_set:Nx \l__char_tmp_tl { \l__char_tmp_tl }
10604 \char_set_catcode_math_toggle:n { 0 }
10605 \tl_put_right:Nn \l__char_tmp_tl { \or: \wedge\wedge @ }
10606 \char_set_catcode_alignment:n { 0 }
10607 \tl_put_right:Nn \l__char_tmp_tl { \or: \wedge\wedge @ }
10608 \tl_put_right:Nn \l__char_tmp_tl { \or: }
10609 \char_set_catcode_parameter:n { 0 }
10610 \tl_put_right:Nn \l__char_tmp_tl { \or: \wedge\wedge @ }
10611 \char_set_catcode_math_superscript:n { 0 }
10612 \tl_put_right:Nn \l__char_tmp_tl { \or: \wedge\wedge @ }
10613 \char_set_catcode_math_subscript:n { 0 }
10614 \tl_put_right:Nn \l__char_tmp_tl { \or: \wedge\wedge @ }
10615 \tl_put_right:Nn \l__char_tmp_tl { \or: }

```

For making spaces, there needs to be an o-type expansion of a `\use:n` (or some other tokenization) to avoid dropping the space.

```

10616 \char_set_catcode_space:n { 0 }
10617 \tl_put_right:No \l__char_tmp_tl { \use:n { \or: } \wedge\wedge @ }
10618 \char_set_catcode_letter:n { 0 }
10619 \tl_put_right:Nn \l__char_tmp_tl { \or: \wedge\wedge @ }
10620 \char_set_catcode_other:n { 0 }
10621 \tl_put_right:Nn \l__char_tmp_tl { \or: \wedge\wedge @ }
10622 \char_set_catcode_active:n { 0 }
10623 \tl_put_right:Nn \l__char_tmp_tl { \or: \wedge\wedge @ }

```

Convert the above temporary list into a series of constant token lists, one for each character code, using `\tex_lowercase:D` to convert  $\wedge\wedge @$  in each case. The x-type expansion ensures that `\tex_lowercase:D` receives the contents of the token list.  $\wedge\wedge L$  is awkward hence this is done in three parts: up to  $\wedge\wedge L$ ,  $\wedge\wedge L$  itself and above  $\wedge\wedge L$ . Notice that at this stage  $\wedge\wedge @$  is active.

```

10624 \cs_set_protected:Npn \l__char_tmp:n #
10625 {
10626 \char_set_lccode:nn { 0 } {#1}
10627 \char_set_lccode:nn { 32 } {#1}
10628 \exp_args:Nx \tex_lowercase:D
10629 {
10630 \tl_const:Nn
10631 \exp_not:c { c__char_ \l__char_int_to_roman:w #1 _tl }
10632 { \exp_not:o \l__char_tmp_tl }
10633 }
10634 }
10635 \int_step_function:nnN { 0 } { 11 } \l__char_tmp:n
10636 \group_begin:
10637 \tl_replace_once:Nnn \l__char_tmp_tl { \wedge\wedge @ } { \ERROR }
10638 \l__char_tmp:n { 12 }
10639 \group_end:

```

```
10640 \int_step_function:nnN { 13 } { 255 } _char_tmp:n
```

As T<sub>E</sub>X is very unhappy if it finds an alignment character inside a primitive `\halign` even when skipping false branches, some precautions are required. T<sub>E</sub>X is happy if the token is hidden between braces within `\if_false: ... \fi:`

```
10641 \cs_new:Npn _char_generate_aux:nnw #1#2#3 \exp_end:
10642 {
10643 #3
10644 \if_false: { \fi:
10645 \exp_after:wN \exp_after:wN
10646 \exp_after:wN \exp_end:
10647 \exp_after:wN \exp_after:wN
10648 \if_case:w #2
10649 \exp_last_unbraced:Nv \exp_stop_f:
10650 { c__char_ _char_int_to_roman:w #1 _tl }
10651 \or:
10652 \fi:
10653 }
10654 \fi:
10655 \group_end:
```

(End definition for `\char_generate:nn` and others. This function is documented on page 130.)

### `\char_to_utfviii_bytes:n`

This code converts a codepoint into the correct UTF-8 representation. In terms of the algorithm itself, see <https://en.wikipedia.org/wiki/UTF-8> for the octet pattern.

```
10656 \cs_new:Npn \char_to_utfviii_bytes:n #1
10657 {
10658 \exp_args:Nf _char_to_utfviii_bytes_auxi:n
10659 { \int_eval:n {#1} }
10660 }
10661 \cs_new:Npn _char_to_utfviii_bytes_auxi:n #1
10662 {
10663 \if_int_compare:w #1 > "80 \exp_stop_f:
10664 \if_int_compare:w #1 < "800 \exp_stop_f:
10665 _char_to_utfviii_bytes_outputi:nw
10666 { _char_to_utfviii_bytes_auxii:Nnn C {#1} { 64 } }
10667 _char_to_utfviii_bytes_outputii:nw
10668 { _char_to_utfviii_bytes_auxiii:n {#1} }
10669 \else:
10670 \if_int_compare:w #1 < "10000 \exp_stop_f:
10671 _char_to_utfviii_bytes_outputi:nw
10672 { _char_to_utfviii_bytes_auxii:Nnn E {#1} { 64 * 64 } }
10673 _char_to_utfviii_bytes_outputii:nw
10674 {
10675 _char_to_utfviii_bytes_auxiii:n
10676 { \int_div_truncate:nn {#1} { 64 } }
10677 }
10678 _char_to_utfviii_bytes_outputiii:nw
10679 { _char_to_utfviii_bytes_auxiii:n {#1} }
10680 \else:
10681 _char_to_utfviii_bytes_outputi:nw
10682 {
10683 _char_to_utfviii_bytes_auxii:Nnn F
10684 {#1} { 64 * 64 * 64 }
```

```

10685 }
10686 __char_to_utfviii_bytes_outputii:nw
10687 {
10688 __char_to_utfviii_bytes_auxiii:n
10689 { \int_div_truncate:nn {#1} { 64 * 64 } }
10690 }
10691 __char_to_utfviii_bytes_outputiii:nw
10692 {
10693 __char_to_utfviii_bytes_auxiii:n
10694 { \int_div_truncate:nn {#1} { 64 } }
10695 }
10696 __char_to_utfviii_bytes_outputiv:nw
10697 { __char_to_utfviii_bytes_auxiii:n {#1} }
10698 \fi:
10699 \fi:
10700 \else:
10701 __char_to_utfviii_bytes_outputi:nw {#1}
10702 \fi:
10703 __char_to_utfviii_bytes_end: { } { } { } { }
10704 }
10705 \cs_new:Npn __char_to_utfviii_bytes_auxii:Nnn #1#2#3
10706 { "#10 + \int_div_truncate:nn {#2} {#3} }
10707 \cs_new:Npn __char_to_utfviii_bytes_auxiii:n #1
10708 { \int_mod:nn {#1} { 64 } + 128 }
10709 \cs_new:Npn __char_to_utfviii_bytes_outputi:nw
10710 #1 #2 __char_to_utfviii_bytes_end: #3
10711 { __char_to_utfviii_bytes_output:fnn { \int_eval:n {#1} } { } {#2} }
10712 \cs_new:Npn __char_to_utfviii_bytes_outputii:nw
10713 #1 #2 __char_to_utfviii_bytes_end: #3#4
10714 { __char_to_utfviii_bytes_output:fnn { \int_eval:n {#1} } { {#3} } {#2} }
10715 \cs_new:Npn __char_to_utfviii_bytes_outputiii:nw
10716 #1 #2 __char_to_utfviii_bytes_end: #3#4#5
10717 {
10718 __char_to_utfviii_bytes_output:fnn
10719 { \int_eval:n {#1} } { {#3} {#4} } {#2}
10720 }
10721 \cs_new:Npn __char_to_utfviii_bytes_outputiv:nw
10722 #1 #2 __char_to_utfviii_bytes_end: #3#4#5#6
10723 {
10724 __char_to_utfviii_bytes_output:fnn
10725 { \int_eval:n {#1} } { {#3} {#4} {#5} } {#2}
10726 }
10727 \cs_new:Npn __char_to_utfviii_bytes_output:nnn #1#2#3
10728 {
10729 #3
10730 __char_to_utfviii_bytes_end: #2 {#1}
10731 }
10732 \cs_generate_variant:Nn __char_to_utfviii_bytes_output:nnn { f }
10733 \cs_new:Npn __char_to_utfviii_bytes_end: { }

```

(End definition for `\char_to_utfviii_bytes:n` and others. This function is documented on page 273.)

**\char\_to\_nfd:N** Look up any NFD and recursively produce the result.

```

__char_to_nfd:n
10734 \cs_new:Npn \char_to_nfd:N #1
__char_to_nfd:Nw

```

```

10735 {
10736 \cs_if_exist:cTF { c__char_nfd_ \token_to_str:N #1 _ tl }
10737 {
10738 \exp_after:wN \exp_after:wN \exp_after:wN __char_to_nfd:Nw
10739 \exp_after:wN \exp_after:wN \exp_after:wN #1
10740 \cs:w c__char_nfd_ \token_to_str:N #1 _ tl \cs_end:
10741 \s__char_stop
10742 }
10743 { \exp_not:n {#1} }
10744 }
10745 \cs_set_eq:NN __char_to_nfd:n \char_to_nfd:N
10746 \cs_new:Npn __char_to_nfd:Nw #1#2#3 \s__char_stop
10747 {
10748 \exp_args:Ne __char_to_nfd:n
10749 { \char_generate:nn { '#2 } { __char_change_case_catcode:N #1 } }
10750 \tl_if_blank:nF {#3}
10751 {
10752 \exp_args:Ne __char_to_nfd:n
10753 { \char_generate:nn { '#3 } { \char_value_catcode:n { '#3 } } }
10754 }
10755 }

```

(End definition for `\char_to_nfd:N`, `\__char_to_nfd:n`, and `\__char_to_nfd:Nw`. This function is documented on page 273.)

`\char_lowercase:N`  
`\char_uppercase:N`  
`\char_titlecase:N`  
`\char_foldcase:N`

To ensure that the category codes produced are predictable, every character is regenerated even if it is otherwise unchanged. This makes life a little interesting when we might have multiple output characters: we have to grab each of them and case change them in reverse order to maintain f-type expandability.

```

__char_change_case:nNN
__char_change_case:nN
__char_change_case_multi:nN
__char_change_case_multi:vN
 __char_change_case_multi:NNNNw
__char_change_case:NNN
__char_change_case:NNNN
__char_change_case:NN
 __char_change_case_catcode:N
\char_str_lowercase:N
\char_str_uppercase:N
\char_str_titlecase:N
 \char_str_foldcase:N
__char_str_change_case:nNN
__char_str_change_case:nN
10756 \cs_new:Npn \char_lowercase:N #1
10757 { __char_change_case:nNN { lower } \char_value_lccode:n #1 }
10758 \cs_new:Npn \char_uppercase:N #1
10759 { __char_change_case:nNN { upper } \char_value_uccode:n #1 }
10760 \cs_new:Npn \char_titlecase:N #1
10761 {
10762 \tl_if_exist:cTF { c__char_titlecase_ \token_to_str:N #1 _ tl }
10763 {
10764 __char_change_case_multi:vN
10765 { c__char_titlecase_ \token_to_str:N #1 _ tl } #1
10766 }
10767 { \char_uppercase:N #1 }
10768 }
10769 \cs_new:Npn \char_foldcase:N #1
10770 { __char_change_case:nNN { fold } \char_value_lccode:n #1 }
10771 \cs_new:Npn __char_change_case:nNN #1#2#3
10772 {
10773 \tl_if_exist:cTF { c__char_ #1 case_ \token_to_str:N #3 _ tl }
10774 {
10775 __char_change_case_multi:vN
10776 { c__char_ #1 case_ \token_to_str:N #3 _ tl } #3
10777 }
10778 { \exp_args:Nf __char_change_case:nN { #2 { '#3 } } #3 }
10779 }
10780 \cs_new:Npn __char_change_case:nN #1#2

```

```

10781 {
10782 \int_compare:nNnTF {#1} = 0
10783 { #2 }
10784 { \char_generate:nn {#1} { _char_change_case_catcode:N #2 } }
10785 }
10786 \cs_new:Npn _char_change_case_multi:nN #1#2
10787 { _char_change_case_multi:NNNw #2 #1 \q_char_no_value \q_char_no_value \s_char_stop
10788 \cs_generate_variant:Nn _char_change_case_multi:nN { v }
10789 \cs_new:Npn _char_change_case_multi:NNNw #1#2#3#4#5 \s_char_stop
10790 {
10791 _char_quark_if_no_value:NTF #4
10792 {
10793 _char_quark_if_no_value:NTF #3
10794 { _char_change_case:NN #1 #2 }
10795 { _char_change_case:NNN #1 #2#3 }
10796 }
10797 { _char_change_case:NNNN #1 #2#3#4 }
10798 }
10799 \cs_new:Npn _char_change_case:NNN #1#2#3
10800 {
10801 \exp_args:Nnf \use:nn
10802 { _char_change_case:NN #1 #2 }
10803 { _char_change_case:NN #1 #3 }
10804 }
10805 \cs_new:Npn _char_change_case:NNNN #1#2#3#4
10806 {
10807 \exp_args:Nnff \use:nnn
10808 { _char_change_case:NN #1 #2 }
10809 { _char_change_case:NN #1 #3 }
10810 { _char_change_case:NN #1 #4 }
10811 }
10812 \cs_new:Npn _char_change_case:NN #1#2
10813 { \char_generate:nn { '#2 } { _char_change_case_catcode:N #1 } }
10814 \cs_new:Npn _char_change_case_catcode:N #1
10815 {
10816 \if_catcode:w \exp_not:N #1 \c_math_toggle_token
10817 3
10818 \else:
10819 \if_catcode:w \exp_not:N #1 \c_alignment_token
10820 4
10821 \else:
10822 \if_catcode:w \exp_not:N #1 \c_math_superscript_token
10823 7
10824 \else:
10825 \if_catcode:w \exp_not:N #1 \c_math_subscript_token
10826 8
10827 \else:
10828 \if_catcode:w \exp_not:N #1 \c_space_token
10829 10
10830 \else:
10831 \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
10832 11
10833 \else:
10834 \if_catcode:w \exp_not:N #1 \c_catcode_other_token

```

```

10835 12
10836 \else:
10837 13
10838 \fi:
10839 \fi:
10840 \fi:
10841 \fi:
10842 \fi:
10843 \fi:
10844 \fi:
10845 }

```

Same story for the string version, except category code is easier to follow. This of course makes this version significantly faster.

```

10846 \cs_new:Npn \char_str_lowercase:N #1
10847 { __char_str_change_case:nNN { lower } \char_value_lccode:n #1 }
10848 \cs_new:Npn \char_str_uppercase:N #1
10849 { __char_str_change_case:nNN { upper } \char_value_uccode:n #1 }
10850 \cs_new:Npn \char_str_titlecase:N #1
10851 {
10852 \tl_if_exist:cTF { c__char_titlecase_ \token_to_str:N #1 _tl }
10853 { \tl_to_str:c { c__char_titlecase_ \token_to_str:N #1 _tl } }
10854 { \char_str_uppercase:N #1 }
10855 }
10856 \cs_new:Npn \char_str_foldcase:N #1
10857 { __char_str_change_case:nNN { fold } \char_value_lccode:n #1 }
10858 \cs_new:Npn __char_str_change_case:nNN #1#2#3
10859 {
10860 \tl_if_exist:cTF { c__char_ #1 case_ \token_to_str:N #3 _tl }
10861 { \tl_to_str:c { c__char_ #1 case_ \token_to_str:N #3 _tl } }
10862 { \exp_args:Nf __char_str_change_case:nN { #2 { '#3 } } #3 }
10863 }
10864 \cs_new:Npn __char_str_change_case:nN #1#2
10865 {
10866 \int_compare:nNnTF {#1} = 0
10867 { \tl_to_str:n {#2} }
10868 { \char_generate:nn {#1} { 12 } }
10869 }
10870 \bool_lazy_or:nnF
10871 { \cs_if_exist_p:N \tex_luatexversion:D }
10872 { \cs_if_exist_p:N \tex_XeTeXversion:D }
10873 {
10874 \cs_set:Npn __char_str_change_case:nN #1#2
10875 { \tl_to_str:n {#2} }
10876 }

```

(End definition for `\char_lowercase:N` and others. These functions are documented on page 130.)

`\c_catcode_other_space_tl` Create a space with category code 12: an “other” space.

```
10877 \tl_const:Nx \c_catcode_other_space_tl { \char_generate:nn { '\ } { 12 } }
```

(End definition for `\c_catcode_other_space_tl`. This function is documented on page 130.)

## 17.4 Generic tokens

*10878* `<@=token>`

`\s__token_stop` Internal scan marks.

*10879* `\scan_new:N \s__token_stop`

(*End definition for \s\_\_token\_stop.*)

`\token_to_meaning:N` These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!

`\token_to_meaning:c`

`\token_to_str:N`

`\token_to_str:c`

`\c_group_begin_token`

`\c_group_end_token`

`\c_math_toggle_token`

`\c_alignment_token`

`\c_parameter_token`

`\c_math_superscript_token`

`\c_math_subscript_token`

`\c_space_token`

`\c_catcode_letter_token`

`\c_catcode_other_token`

(*End definition for \token\_to\_meaning:N and \token\_to\_str:N. These functions are documented on page 134.*)

We define these useful tokens. For the brace and space tokens things have to be done by hand: the formal argument spec. for `\cs_new_eq:NN` does not cover them so we do things by hand. (As currently coded it would *work* with `\cs_new_eq:NN` but that’s not really a great idea to show off: we want people to stick to the defined interfaces and that includes us.) So that these few odd names go into the log when appropriate there is a need to hand-apply the `\_kernel_chk_if_free_cs:N` check.

*10880* `\group_begin:`

*10881*   `\_kernel_chk_if_free_cs:N \c_group_begin_token`

*10882*   `\tex_global:D \tex_let:D \c_group_begin_token {`

*10883*   `\_kernel_chk_if_free_cs:N \c_group_end_token`

*10884*   `\tex_global:D \tex_let:D \c_group_end_token }`

*10885*   `\char_set_catcode_math_toggle:N \*`

*10886*   `\cs_new_eq:NN \c_math_toggle_token *`

*10887*   `\char_set_catcode_alignment:N \*`

*10888*   `\cs_new_eq:NN \c_alignment_token *`

*10889*   `\cs_new_eq:NN \c_parameter_token #`

*10890*   `\cs_new_eq:NN \c_math_superscript_token ^`

*10891*   `\char_set_catcode_math_subscript:N \*`

*10892*   `\cs_new_eq:NN \c_math_subscript_token *`

*10893*   `\_kernel_chk_if_free_cs:N \c_space_token`

*10894*   `\use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~`

*10895*   `\cs_new_eq:NN \c_catcode_letter_token a`

*10896*   `\cs_new_eq:NN \c_catcode_other_token 1`

*10897* `\group_end:`

(*End definition for \c\_group\_begin\_token and others. These functions are documented on page 134.*)

`\c_catcode_active_tl` Not an implicit token!

*10898* `\group_begin:`

*10899*   `\char_set_catcode_active:N \*`

*10900*   `\tl_const:Nn \c_catcode_active_tl { \exp_not:N * }`

*10901* `\group_end:`

(*End definition for \c\_catcode\_active\_tl. This variable is documented on page 134.*)

## 17.5 Token conditionals

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.

```
10902 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
10903 {
10904 \if_catcode:w \exp_not:N #1 \c_group_begin_token
10905 \prg_return_true: \else: \prg_return_false: \fi:
10906 }
```

(End definition for `\token_if_group_begin:NTF`. This function is documented on page 135.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.

```
10907 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
10908 {
10909 \if_catcode:w \exp_not:N #1 \c_group_end_token
10910 \prg_return_true: \else: \prg_return_false: \fi:
10911 }
```

(End definition for `\token_if_group_end:NTF`. This function is documented on page 135.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.

```
10912 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
10913 {
10914 \if_catcode:w \exp_not:N #1 \c_math_toggle_token
10915 \prg_return_true: \else: \prg_return_false: \fi:
10916 }
```

(End definition for `\token_if_math_toggle:NTF`. This function is documented on page 135.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.

```
10917 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
10918 {
10919 \if_catcode:w \exp_not:N #1 \c_alignment_token
10920 \prg_return_true: \else: \prg_return_false: \fi:
10921 }
```

(End definition for `\token_if_alignment:NTF`. This function is documented on page 135.)

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this.

`\token_if_parameter:NTF` We have to trick TeX a bit to avoid an error message: within a group we prevent `\c_parameter_token` from behaving like a macro parameter character. The definitions of `\prg_new_conditional:Npnn` are global, so they remain after the group.

```
10922 \group_begin:
10923 \cs_set_eq:NN \c_parameter_token \scan_stop:
10924 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
10925 {
10926 \if_catcode:w \exp_not:N #1 \c_parameter_token
10927 \prg_return_true: \else: \prg_return_false: \fi:
10928 }
10929 \group_end:
```

(End definition for `\token_if_parameter:NTF`. This function is documented on page 135.)

`\token_if_math_superscript_p:N` `\token_if_math_superscript:NTF` Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.

```
10930 \prg_new_conditional:Npnn \token_if_math_superscript:N #1
10931 { p , T , F , TF }
10932 {
10933 \if_catcode:w \exp_not:N #1 \c_math_superscript_token
10934 \prg_return_true: \else: \prg_return_false: \fi:
10935 }
```

(End definition for `\token_if_math_superscript:NTF`. This function is documented on page 135.)

`\token_if_math_subscript_p:N` `\token_if_math_subscript:NTF` Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.

```
10936 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
10937 {
10938 \if_catcode:w \exp_not:N #1 \c_math_subscript_token
10939 \prg_return_true: \else: \prg_return_false: \fi:
10940 }
```

(End definition for `\token_if_math_subscript:NTF`. This function is documented on page 135.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.

`\token_if_space:NTF`

```
10941 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
10942 {
10943 \if_catcode:w \exp_not:N #1 \c_space_token
10944 \prg_return_true: \else: \prg_return_false: \fi:
10945 }
```

(End definition for `\token_if_space:NTF`. This function is documented on page 135.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.

`\token_if_letter:NTF`

```
10946 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
10947 {
10948 \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
10949 \prg_return_true: \else: \prg_return_false: \fi:
10950 }
```

(End definition for `\token_if_letter:NTF`. This function is documented on page 136.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_catcode_other_token` for this.

`\token_if_other:NTF`

```
10951 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
10952 {
10953 \if_catcode:w \exp_not:N #1 \c_catcode_other_token
10954 \prg_return_true: \else: \prg_return_false: \fi:
10955 }
```

(End definition for `\token_if_other:NTF`. This function is documented on page 136.)

`\token_if_active_p:N` Check if token is an active char token. We use the constant `\c_catcode_active_tl` for this. A technical point is that `\c_catcode_active_tl` is in fact a macro expanding to `\exp_not:N *`, where \* is active.

```
10956 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
10957 {
10958 \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
10959 \prg_return_true: \else: \prg_return_false: \fi:
10960 }
```

(End definition for `\token_if_active:NTF`. This function is documented on page 136.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.

```
10961 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
10962 {
10963 \if_meaning:w #1 #2
10964 \prg_return_true: \else: \prg_return_false: \fi:
10965 }
```

(End definition for `\token_if_eq_meaning:NNTF`. This function is documented on page 136.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.

```
10966 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
10967 {
10968 \if_catcode:w \exp_not:N #1 \exp_not:N #2
10969 \prg_return_true: \else: \prg_return_false: \fi:
10970 }
```

(End definition for `\token_if_eq_catcode:NNTF`. This function is documented on page 136.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.

```
10971 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
10972 {
10973 \if_charcode:w \exp_not:N #1 \exp_not:N #2
10974 \prg_return_true: \else: \prg_return_false: \fi:
10975 }
```

(End definition for `\token_if_eq_charcode:NNTF`. This function is documented on page 136.)

`\token_if_macro_p:N`

`\token_if_macro:NTF`

`\_\_token_if_macro_p:w`

When a token is a macro, `\token_to_meaning:N` always outputs something like `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`. However, this can fail the five `\...mark` primitives, whose meaning has the form `...mark:<user material>`. The problem is that the `<user material>` can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyse what is left. However, macros can have any combination of `\long`, `\protected` or `\outer` (not used in L<sup>A</sup>T<sub>E</sub>X3) before the string `macro::`. We thus only select the part of the meaning between the first `ma` and the first following `::`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `\m...`

Both `ma` and `:` must be of category code 12 (other), so are detokenized.

```
10976 \use:x
10977 {
10978 \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N ##1
10979 { p , T , F , TF }
10980 {
10981 \exp_not:N \exp_after:wN \exp_not:N __token_if_macro_p:w
10982 \exp_not:N \token_to_meaning:N ##1 \tl_to_str:n { ma : }
10983 \s__token_stop
10984 }
10985 \cs_new:Npn \exp_not:N __token_if_macro_p:w
```

```

10986 ##1 \tl_to_str:n { ma } ##2 \c_colon_str ##3 \s__token_stop
10987 }
10988 {
10989 \str_if_eq:nnTF { #2 } { cro }
10990 { \prg_return_true: }
10991 { \prg_return_false: }
10992 }

```

(End definition for `\token_if_macro:NTF` and `\_token_if_macro_p:w`. This function is documented on page 136.)

**\token\_if\_cs\_p:N** Check if token has same catcode as a control sequence. This follows the same pattern as **\token\_if\_cs:NTF** for `\token_if_letter:N` etc. We use `\scan_stop:` for this.

```

10993 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
10994 {
10995 \if_catcode:w \exp_not:N #1 \scan_stop:
10996 \prg_return_true: \else: \prg_return_false: \fi:
10997 }

```

(End definition for `\token_if_cs:NTF`. This function is documented on page 136.)

**\token\_if\_expandable\_p:N** **\token\_if\_expandable:NTF** Check if token is expandable. We use the fact that TeX temporarily converts `\exp_not:N` `\token` into `\scan_stop:` if `\token` is expandable. An undefined token is not considered as expandable. No problem nesting the conditionals, since the third #1 is only skipped if it is non-expandable (hence not part of TeX's conditional apparatus).

```

10998 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
10999 {
11000 \exp_after:wN \if_meaning:w \exp_not:N #1 #1
11001 \prg_return_false:
11002 \else:
11003 \if_cs_exist:N #1
11004 \prg_return_true:
11005 \else:
11006 \prg_return_false:
11007 \fi:
11008 \fi:
11009 }

```

(End definition for `\token_if_expandable:NTF`. This function is documented on page 136.)

`\_token_delimit_by_char":w` `\_token_delimit_by_count:w` `\_token_delimit_by_dimen:w` `\_token_delimit_by_\font:w` `\_token_delimit_by_macro:w` `\_token_delimit_by_muskip:w` `\_token_delimit_by_skip:w` `\_token_delimit_by_toks:w` These auxiliary functions are used below to define some conditionals which detect whether the `\meaning` of their argument begins with a particular string. Each auxiliary takes an argument delimited by a string, a second one delimited by `\s__token_stop`, and returns the first one and its delimiter. This result is eventually compared to another string. Note that the “font” auxiliary is delimited by a space followed by “font”. This avoids an unnecessary check for the `\font` primitive below.

```

11010 \group_begin:
11011 \cs_set_protected:Npn _token_tmp:w #1
11012 {
11013 \use:x
11014 {
11015 \cs_new:Npn \exp_not:c { _token_delimit_by_ #1 :w }
11016 #####1 \tl_to_str:n {#1} #####2 \s__token_stop
11017 { #####1 \tl_to_str:n {#1} }

```

```

11018 }
11019 }
11020 __token_tmp:w { char" }
11021 __token_tmp:w { count }
11022 __token_tmp:w { dimen }
11023 __token_tmp:w { ~ font }
11024 __token_tmp:w { macro }
11025 __token_tmp:w { muskip }
11026 __token_tmp:w { skip }
11027 __token_tmp:w { toks }
11028 \group_end:

```

(End definition for `\__token_delimit_by_char":w` and others.)

```

\token_if_chardef_p:N
\token_if_chardef:NTF
\token_if_mathchardef_p:N
\token_if_mathchardef:NTF
\token_if_long_macro_p:N
\token_if_long_macro:NTF
 \token_if_protected_macro_p:N
\token_if_protected_macro:NTF
 \token_if_protected_long_macro_p:N
 \token_if_protected_long_macro:NTF
\token_if_font_selection_p:N
\token_if_font_selection:NTF
\token_if_dim_register_p:N
\token_if_dim_register:NTF
\token_if_int_register_p:N
\token_if_int_register:NTF
 \token_if_muskip_register_p:N
\token_if_muskip_register:NTF
 \token_if_skip_register_p:N
\token_if_skip_register:NTF
\token_if_toks_register_p:N
\token_if_toks_register:NTF

```

Each of these conditionals tests whether its argument's `\meaning` starts with a given string. This is essentially done by having an auxiliary grab an argument delimited by the string and testing whether the argument was empty. Of course, a copy of this string must first be added to the end of the `\meaning` to avoid a runaway argument in case it does not contain the string. Two complications arise. First, the escape character is not fixed, and cannot be included in the delimiter of the auxiliary function (this function cannot be defined on the fly because tests must remain expandable): instead the first argument of the auxiliary (plus the delimiter to avoid complications with trailing spaces) is compared using `\str_if_eq:eeTF` to the result of applying `\token_to_str:N` to a control sequence. Second, the `\meaning` of primitives such as `\dimen` or `\dimendef` starts in the same way as registers such as `\dimen123`, so they must be tested for.

Characters used as delimiters must have catcode 12 and are obtained through `\tl_to_str:n`. This requires doing all definitions within x-expansion. The temporary function `\__token_tmp:w` used to define each conditional receives three arguments: the name of the conditional, the auxiliary's delimiter (also used to name the auxiliary), and the string to which one compares the auxiliary's result. Note that the `\meaning` of a protected long macro starts with `\protected\long macro`, with no space after `\protected` but a space after `\long`, hence the mixture of `\token_to_str:N` and `\tl_to_str:n`.

For the first six conditionals, `\cs_if_exist:cT` turns out to be `false` (thanks to the leading space for `font`), and the code boils down to a string comparison between the result of the auxiliary on the `\meaning` of the conditional's argument `####1`, and `#3`. Both are evaluated at run-time, as this is important to get the correct escape character.

The other five conditionals have additional code that compares the argument `####1` to two TeX primitives which would wrongly be recognized as registers otherwise. Despite using TeX's primitive conditional construction, this does not break when `####1` is itself a conditional, because branches of the conditionals are only skipped if `####1` is one of the two primitives that are tested for (which are not TeX conditionals).

```

11029 \group_begin:
11030 \cs_set_protected:Npn __token_tmp:w #1#2#3
11031 {
11032 \use:x
11033 {
11034 \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } ####1
11035 { p , T , F , TF }
11036 {
11037 \cs_if_exist:cT { tex_ #2 :D }
11038 {
11039 \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 :D }

```

```

11040 \exp_not:N \prg_return_false:
11041 \exp_not:N \else:
11042 \exp_not:N \if_meaning:w #####1 \exp_not:c { tex_ #2 def:D }
11043 \exp_not:N \prg_return_false:
11044 \exp_not:N \else:
11045 }
11046 \exp_not:N \str_if_eq:eeTF
11047 {
11048 \exp_not:N \exp_after:wN
11049 \exp_not:c { __token_delimit_by_ #2 :w }
11050 \exp_not:N \token_to_meaning:N #####1
11051 ? \tl_to_str:n {#2} \s__token_stop
11052 }
11053 { \exp_not:n {#3} }
11054 { \exp_not:N \prg_return_true: }
11055 { \exp_not:N \prg_return_false: }
11056 \cs_if_exist:cT { tex_ #2 :D }
11057 {
11058 \exp_not:N \fi:
11059 \exp_not:N \fi:
11060 }
11061 }
11062 }
11063 }
11064 __token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
11065 __token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
11066 __token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
11067 __token_tmp:w { protected_macro } { macro }
11068 { \tl_to_str:n { \protected } macro }
11069 __token_tmp:w { protected_long_macro } { macro }
11070 { \token_to_str:N \protected \tl_to_str:n { \long } macro }
11071 __token_tmp:w { font_selection } { ~ font } { select ~ font }
11072 __token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
11073 __token_tmp:w { int_register } { count } { \token_to_str:N \count }
11074 __token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
11075 __token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
11076 __token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
11077 \group_end:

```

(End definition for `\token_if_chardef:NTF` and others. These functions are documented on page 137.)

### `\token_if_primitive_p:N`

### `\token_if_primitive:NTF`

`\__token_if_primitive:NNw`

`\__token_if_primitive space:w`

`\__token_if_primitive_nullfont:N`

`\__token_if_primitive_loop:N`

`\__token_if_primitive:Nw`

`\__token_if_primitive_undefined:N`

`\__token_if_primitive_lua:N`

We filter out macros first, because they cause endless trouble later otherwise.

Primitives are almost distinguished by the fact that the result of `\token_to_meaning:N` is formed from letters only. Every other token has either a space (e.g., the letter A), a digit (e.g., `\count123`) or a double quote (e.g., `\char"A`).

Ten exceptions: on the one hand, `\tex_undefined:D` is not a primitive, but its meaning is `undefined`, only letters; on the other hand, `\space`, `\italiccorr`, `\hyphen`, `\firstmark`, `\topmark`, `\botmark`, `\splitfirstmark`, `\splitbotmark`, and `\nullfont` are primitives, but have non-letters in their meaning.

We start by removing the two first (non-space) characters from the meaning. This removes the escape character (which may be nonexistent depending on `\endlinechar`), and takes care of three of the exceptions: `\space`, `\italiccorr` and `\hyphen`, whose meaning is at most two characters. This leaves a string terminated by some `:`, and `\s__token_stop`.

The meaning of each one of the five `\...mark` primitives has the form  $\langle letters \rangle : \langle user material \rangle$ . In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either ", or a space, or a digit. Two exceptions remain: `\tex_undefined:D`, which is not a primitive, and `\nullfont`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\nullfont`. Otherwise, we go through characters one by one, and stop at the first character less than 'A' (this is not quite a test for "only letters", but is close enough to work in this context). If this first character is : then we have a primitive, or `\tex_undefined:D`, and if it is " or a digit, then the token is not a primitive.

For LuaTeX we use a different implementation which just looks at the command code for the token and compares it to a list of non-primitives. Again, `\nullfont` is a special case because it is the only primitive with the normally non-primitive `set_font` command code.

```

11078 \sys_if_engine_luatex:TF
11079 {
11080 </tex>
11081 <*lua>
11082 do
11083 local get_next = token.get_next
11084 local get_command = token.get_command
11085 local get_index = token.get_index
11086 local get_mode = token.get_mode or token.get_index
11087 local cmd = token.command_id
11088 local set_font = cmd'get_font'
11089 local biggest_char = token.biggest_char()
11090
11091 local mode_below_biggest_char = {}
11092 local index_not_nil = {}
11093 local mode_not_null = {}
11094 local non_primitive = {
11095 [cmd'left_brace'] = true,
11096 [cmd'right_brace'] = true,
11097 [cmd'math_shift'] = true,
11098 [cmd'mac_param'] = mode_below_biggest_char,
11099 [cmd'sup_mark'] = true,
11100 [cmd'sub_mark'] = true,
11101 [cmd'endv'] = true,
11102 [cmd'spacer'] = true,
11103 [cmd'letter'] = true,
11104 [cmd'other_char'] = true,
11105 [cmd'tab_mark'] = mode_below_biggest_char,
11106 [cmd'char_given'] = true,
11107 [cmd'math_given'] = true,
11108 [cmd'xmath_given'] = true,
11109 [cmd'set_font'] = mode_not_null,
11110 [cmd'undefined_cs'] = true,
11111 [cmd'call'] = true,
11112 [cmd'long_call'] = true,
11113 [cmd'outer_call'] = true,

```

```

11114 [cmd'long_outer_call'] = true,
11115 [cmd'assign_glue'] = index_not_nil,
11116 [cmd'assign_mu_glue'] = index_not_nil,
11117 [cmd'assign_toks'] = index_not_nil,
11118 [cmd'assign_int'] = index_not_nil,
11119 [cmd'assign_attr'] = true,
11120 [cmd'assign_dimen'] = index_not_nil,
11121 }
11122
11123 luacmd(" __token_if_primitive_lua:N", function()
11124 local tok = get_next()
11125 local is_non_primitive = non_primitive[get_command(tok)]
11126 return put_next(
11127 is_non_primitive == true
11128 and false_tok
11129 or is_non_primitive == nil
11130 and true_tok
11131 or is_non_primitive == mode_not_null
11132 and (get_mode(tok) == 0 and true_tok or false_tok)
11133 or is_non_primitive == index_not_nil
11134 and (get_index(tok) and false_tok or true_tok)
11135 or is_non_primitive == mode_below_biggest_char
11136 and (get_mode(tok) > biggest_char and true_tok or false_tok))
11137 end, "global")
11138 end
11139 </lua>
11140 <*tex>
11141 \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
11142 {
11143 __token_if_primitive_lua:N #1
11144 }
11145 }
11146 {
11147 \tex_chardef:D \c__token_A_int = 'A ~ %
11148 \use:x
11149 {
11150 \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N ##1
11151 { p , T , F , TF }
11152 {
11153 \exp_not:N \token_if_macro:NTF ##1
11154 \exp_not:N \prg_return_false:
11155 {
11156 \exp_not:N \exp_after:wN \exp_not:N __token_if_primitive>NNw
11157 \exp_not:N \token_to_meaning:N ##1
11158 \tl_to_str:n { : : : } \s__token_stop ##1
11159 }
11160 }
11161 \cs_new:Npn \exp_not:N __token_if_primitive>NNw
11162 ##1##2 ##3 \c_colon_str ##4 \s__token_stop
11163 {
11164 \exp_not:N \tl_if_empty:oTF
11165 { \exp_not:N __token_if_primitive_space:w ##3 ~ }
11166 {
11167 \exp_not:N __token_if_primitive_loop:N ##3

```

```

11168 \c_colon_str \s__token_stop
11169 }
11170 { \exp_not:N __token_if_primitive_nullfont:N }
11171 }
11172 }
11173 \cs_new:Npn __token_if_primitive_space:w #1 ~ { }
11174 \cs_new:Npn __token_if_primitive_nullfont:N #1
11175 {
11176 \if_meaning:w \tex_nullfont:D #1
11177 \prg_return_true:
11178 \else:
11179 \prg_return_false:
11180 \fi:
11181 }
11182 \cs_new:Npn __token_if_primitive_loop:N #1
11183 {
11184 \if_int_compare:w '#1 < \c__token_A_int %
11185 \exp_after:wN __token_if_primitive:Nw
11186 \exp_after:wN #1
11187 \else:
11188 \exp_after:wN __token_if_primitive_loop:N
11189 \fi:
11190 }
11191 \cs_new:Npn __token_if_primitive:Nw #1 #2 \s__token_stop
11192 {
11193 \if:w : #1
11194 \exp_after:wN __token_if_primitive_undefined:N
11195 \else:
11196 \prg_return_false:
11197 \exp_after:wN \use_none:n
11198 \fi:
11199 }
11200 \cs_new:Npn __token_if_primitive_undefined:N #1
11201 {
11202 \if_cs_exist:N #1
11203 \prg_return_true:
11204 \else:
11205 \prg_return_false:
11206 \fi:
11207 }
11208 }
```

(End definition for `\token_if_primitive:NTF` and others. This function is documented on page 138.)

## 17.6 Peeking ahead at the next token

`11209 ⟨@=peek⟩`

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;
2. peek at the next non-space token;

3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.

**\l\_peek\_token**  
**\g\_peek\_token**

Storage tokens which are publicly documented: the token peeked.

```
11210 \cs_new_eq:NN \l_peek_token ?
11211 \cs_new_eq:NN \g_peek_token ?
```

(End definition for **\l\_peek\_token** and **\g\_peek\_token**. These variables are documented on page 138.)

**\l\_\_peek\_search\_token**

The token to search for as an implicit token: cf. **\l\_\_peek\_search\_tl**.

```
11212 \cs_new_eq:NN \l__peek_search_token ?
```

(End definition for **\l\_\_peek\_search\_token**.)

**\l\_\_peek\_search\_tl**

The token to search for as an explicit token: cf. **\l\_\_peek\_search\_token**.

```
11213 \tl_new:N \l__peek_search_tl
```

(End definition for **\l\_\_peek\_search\_tl**.)

**\\_\\_peek\_true:w**

Functions used by the branching and space-stripping code.

```
11214 \cs_new:Npn __peek_true:w { }
11215 \cs_new:Npn __peek_true_aux:w { }
11216 \cs_new:Npn __peek_false:w { }
11217 \cs_new:Npn __peek_tmp:w { }
```

(End definition for **\\_\\_peek\_true:w** and others.)

**\s\_\_peek\_mark**

Internal scan marks.

**\s\_\_peek\_stop**

```
11218 \scan_new:N \s__peek_mark
11219 \scan_new:N \s__peek_stop
```

(End definition for **\s\_\_peek\_mark** and **\s\_\_peek\_stop**.)

**\\_\\_peek\_use\_none\_delimit\_by\_s\_stop:w**

Functions to gobble up to a scan mark.

```
11220 \cs_new:Npn __peek_use_none_delimit_by_s_stop:w #1 \s__peek_stop { }
```

(End definition for **\\_\\_peek\_use\_none\_delimit\_by\_s\_stop:w**.)

**\peek\_after:Nw**

Simple wrappers for **\futurelet**: no arguments absorbed here.

**\peek\_gafter:Nw**

```
11221 \cs_new_protected:Npn \peek_after:Nw
11222 { \tex_futurelet:D \l_peek_token }
11223 \cs_new_protected:Npn \peek_gafter:Nw
11224 { \tex_global:D \tex_futurelet:D \g_peek_token }
```

(End definition for **\peek\_after:Nw** and **\peek\_gafter:Nw**. These functions are documented on page 138.)

**\\_\\_peek\_true\_remove:w**

A function to remove the next token and then regain control.

```
11225 \cs_new_protected:Npn __peek_true_remove:w
11226 {
11227 \tex_afterassignment:D __peek_true_aux:w
11228 \cs_set_eq:NN __peek_tmp:w
11229 }
```

(End definition for **\\_\\_peek\_true\_remove:w**.)

**\\_peek\_remove\_spaces:n** Repeatedly use `\_peek_true_remove:w` to remove a space and call `\_peek_true_aux:w`.

```

11230 \cs_new_protected:Npn _peek_remove_spaces:n #1
11231 {
11232 \cs_set:Npx _peek_false:w { \exp_not:n {#1} }
11233 \group_align_safe_begin:
11234 \cs_set:Npn _peek_true_aux:w { \peek_after:Nw _peek_remove_spaces: }
11235 _peek_true_aux:w
11236 }
11237 \cs_new_protected:Npn _peek_remove_spaces:
11238 {
11239 \if_meaning:w \l_peek_token \c_space_token
11240 \exp_after:wN _peek_true_remove:w
11241 \else:
11242 \group_align_safe_end:
11243 \exp_after:wN _peek_false:w
11244 \fi:
11245 }

```

(End definition for `\_peek_remove_spaces:n` and `\_peek_remove_spaces:`. This function is documented on page 274.)

**\\_peek\_token\_generic\_aux:NNNTF** The generic functions store the test token in both implicit and explicit modes, and the **true** and **false** code as token lists, more or less. The two branches have to be absorbed here as the input stream needs to be cleared for the peek function itself. Here, #1 is `\_peek_true_remove:w` when removing the token and `\_peek_true_aux:w` otherwise.

```

11246 \cs_new_protected:Npn _peek_token_generic_aux:NNNTF #1#2#3#4#5
11247 {
11248 \group_align_safe_begin:
11249 \cs_set_eq:NN \l_peek_search_token #3
11250 \tl_set:Nn \l_peek_search_tl {#3}
11251 \cs_set:Npx _peek_true_aux:w
11252 {
11253 \exp_not:N \group_align_safe_end:
11254 \exp_not:n {#4}
11255 }
11256 \cs_set_eq:NN _peek_true:w #1
11257 \cs_set:Npx _peek_false:w
11258 {
11259 \exp_not:N \group_align_safe_end:
11260 \exp_not:n {#5}
11261 }
11262 \peek_after:Nw #2
11263 }

```

(End definition for `\_peek_token_generic_aux:NNNTF`.)

**\\_peek\_token\_generic:NNTF** For token removal there needs to be a call to the auxiliary function which does the work.

```

11264 \cs_new_protected:Npn _peek_token_generic:NNTF
11265 { _peek_token_generic_aux:NNNTF _peek_true_aux:w }
11266 \cs_new_protected:Npn _peek_token_generic:NNT #1#2#3
11267 { _peek_token_generic:NNTF #1 #2 {#3} { } }
11268 \cs_new_protected:Npn _peek_token_generic:NNF #1#2#3
11269 { _peek_token_generic:NNTF #1 #2 { } {#3} }

```

```

11270 \cs_new_protected:Npn __peek_token_remove_generic:NNTF
11271 { __peek_token_generic_aux:NNNTF __peek_true_remove:w }
11272 \cs_new_protected:Npn __peek_token_remove_generic:NNT #1#2#3
11273 { __peek_token_remove_generic:NNTF #1 #2 {#3} { } }
11274 \cs_new_protected:Npn __peek_token_remove_generic:NNF #1#2#3
11275 { __peek_token_remove_generic:NNTF #1 #2 { } {#3} }

```

(End definition for `\__peek_token_generic:NNTF` and `\__peek_token_remove_generic:NNTF`.)

`\__peek_execute_branches_meaning:`

```

11276 \cs_new:Npn __peek_execute_branches_meaning:
11277 {
11278 \if_meaning:w \l_peek_token \l__peek_search_token
11279 \exp_after:wN __peek_true:w
11280 \else:
11281 \exp_after:wN __peek_false:w
11282 \fi:
11283 }

```

(End definition for `\__peek_execute_branches_meaning`.)

```

__peek_execute_branches_catcode:
__peek_execute_branches_charcode:
__peek_execute_branches_catcode_aux:
__peek_execute_branches_catcode_auxii:N
__peek_execute_branches_catcode_auxiii:

```

The catcode and charcode tests are very similar, and in order to use the same auxiliaries we do something a little bit odd, firing `\if_catcode:w` and `\if_charcode:w` before finding the operands for those tests, which are only given in the `auxii:N` and `auxiii:` auxiliaries. For our purposes, three kinds of tokens may follow the peeking function:

- control sequences which are not equal to a non-active character token (*e.g.*, macro, primitive);
- active characters which are not equal to a non-active character token (*e.g.*, macro, primitive);
- explicit non-active character tokens, or control sequences or active characters set equal to a non-active character token.

The first two cases are not distinguishable simply using TeX's `\futurelet`, because we can only access the `\meaning` of tokens in that way. In those cases, detected thanks to a comparison with `\scan_stop`, we grab the following token, and compare it explicitly with the explicit search token stored in `\l__peek_search_t1`. The `\exp_not:N` prevents outer macros (coming from non-`\TeX3` code) from blowing up. In the third case, `\l_peek_token` is good enough for the test, and we compare it again with the explicit search token. Just like the peek token, the search token may be of any of the three types above, hence the need to use the explicit token that was given to the peek function.

```

11284 \cs_new:Npn __peek_execute_branches_catcode:
11285 { \if_catcode:w __peek_execute_branches_catcode_aux: }
11286 \cs_new:Npn __peek_execute_branches_charcode:
11287 { \if_charcode:w __peek_execute_branches_catcode_aux: }
11288 \cs_new:Npn __peek_execute_branches_catcode_aux:
11289 {
11290 \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
11291 \exp_after:wN \exp_after:wN
11292 \exp_after:wN __peek_execute_branches_catcode_auxii:N
11293 \exp_after:wN \exp_not:N
11294 \else:

```

```

11295 \exp_after:wN __peek_execute_branches_catcode_auxiii:
11296 \fi:
11297 }
11298 \cs_new:Npn __peek_execute_branches_catcode_auxii:N #1
11299 {
11300 \exp_not:N #1
11301 \exp_after:wN \exp_not:N \l__peek_search_tl
11302 \exp_after:wN __peek_true:w
11303 \else:
11304 \exp_after:wN __peek_false:w
11305 \fi:
11306 #1
11307 }
11308 \cs_new:Npn __peek_execute_branches_catcode_auxiii:
11309 {
11310 \exp_not:N \l__peek_token
11311 \exp_after:wN \exp_not:N \l__peek_search_tl
11312 \exp_after:wN __peek_true:w
11313 \else:
11314 \exp_after:wN __peek_false:w
11315 \fi:
11316 }

```

(End definition for `\__peek_execute_branches_catcode:` and others.)

`\peek_catcode:NTF` The public functions themselves cannot be defined using `\prg_new_conditional:Npnn`.  
`\peek_catcode_remove:NTF` Instead, the TF, T, F variants are defined in terms of corresponding variants of `\__-peek_token_generic:NNTF` or `\__-peek_token_remove_generic:NNTF`, with first argument one of `\__peek_execute_branches_catcode:`, `\__peek_execute_branches_-charcode:`, or `\__peek_execute_branches_meaning:`.

`\peek_meaning:NTF`

```

11317 \tl_map_inline:nn { { catcode } { charcode } { meaning } }
11318 {
11319 \tl_map_inline:nn { { } { _remove } }
11320 {
11321 \tl_map_inline:nn { { TF } { T } { F } }
11322 {
11323 \cs_new_protected:cpx { peek_ #1 ##1 :N #####1 }
11324 {
11325 \exp_not:c { __peek_token ##1 _generic:NN #####1 }
11326 \exp_not:c { __peek_execute_branches_ #1 : }
11327 }
11328 }
11329 }
11330 }

```

(End definition for `\peek_catcode:NTF` and others. These functions are documented on page 139.)

`\peek_catcode_ignore_spaces:NTF` To ignore spaces, remove them using `\peek_remove_spaces:n` before running the tests.  
`\peek_catcode_remove_ignore_spaces:NTF`

```

11331 \tl_map_inline:nn
11332 {
11333 { catcode } { catcode_remove }
11334 { charcode } { charcode_remove }
11335 { meaning } { meaning_remove }
11336 }

```

```

11337 {
11338 \cs_new_protected:cp{ peek_#1_ignore_spaces:NTF } ##1##2##3
11339 {
11340 \peek_remove_spaces:n
11341 { \exp_not:c { peek_#1:NTF } ##1 {##2} {##3} }
11342 }
11343 \cs_new_protected:cp{ peek_#1_ignore_spaces:NT } ##1##2
11344 {
11345 \peek_remove_spaces:n
11346 { \exp_not:c { peek_#1:NT } ##1 {##2} }
11347 }
11348 \cs_new_protected:cp{ peek_#1_ignore_spaces:NF } ##1##2
11349 {
11350 \peek_remove_spaces:n
11351 { \exp_not:c { peek_#1:NF } ##1 {##2} }
11352 }
11353 }

```

(End definition for `\peek_catcode_ignore_spaces:NTF` and others. These functions are documented on page 199.)

**\peek\_N\_type:TF** All tokens are N-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since `\l_peek_token` might be outer, we cannot use the convenient `\bool_if:nTF` function, and must resort to the old trick of using `\ifodd` to expand a set of tests. The `false` branch of this test is taken if the token is one of the first three kinds of non-N-type tokens (explicit or implicit), thus we call `\__peek_false:w`. In the `true` branch, we must detect outer tokens, without impacting performance too much for non-outer tokens. The first filter is to search for `outer` in the `\meaning` of `\l_peek_token`. If that is absent, `\__peek_use_none_delimit_by_s_stop:w` cleans up, and we call `\__peek_true:w`. Otherwise, the token can be a non-outer macro or a primitive mark whose parameter or replacement text contains `outer`, it can be the primitive `\outer`, or it can be an outer token. Macros and marks would have `ma` in the part before the first occurrence of `outer`; the meaning of `\outer` has nothing after `outer`, contrarily to outer macros; and that covers all cases, calling `\__peek_true:w` or `\__peek_false:w` as appropriate. Here, there is no `<search token>`, so we feed a dummy `\scan_stop:` to the `\__peek_token_generic:NNTF` function.

```

11354 \group_begin:
11355 \cs_set_protected:Npn __peek_tmp:w #1 \s__peek_stop
11356 {
11357 \cs_new_protected:Npn __peek_execute_branches_N_type:
11358 {
11359 \if_int_odd:w
11360 \if_catcode:w \exp_not:N \l_peek_token { 0 \exp_stop_f: \fi:
11361 \if_catcode:w \exp_not:N \l_peek_token } 0 \exp_stop_f: \fi:
11362 \if_meaning:w \l_peek_token \c_space_token 0 \exp_stop_f: \fi:
11363 1 \exp_stop_f:
11364 \exp_after:wn __peek_N_type:w
11365 \token_to_meaning:N \l_peek_token
11366 \s__peek_mark __peek_N_type_aux:nnw
11367 #1 \s__peek_mark __peek_use_none_delimit_by_s_stop:w
11368 \s__peek_stop
11369 \exp_after:wn __peek_true:w
11370 \else:

```

```

11371 \exp_after:wN _peek_false:w
11372 \fi:
11373 }
11374 \cs_new_protected:Npn _peek_N_type:w ##1 #1 ##2 \s_peek_mark ##3
11375 { ##3 {##1} {##2} }
11376 }
11377 \exp_after:wN _peek_tmp:w \tl_to_str:n { outer } \s_peek_stop
11378 \group_end:
11379 \cs_new_protected:Npn _peek_N_type_aux:nnw #1 #2 #3 \fi:
11380 {
11381 \fi:
11382 \tl_if_in:ntF {#1} { \tl_to_str:n {ma} }
11383 { _peek_true:w }
11384 { \tl_if_empty:ntF {#2} { _peek_true:w } { _peek_false:w } }
11385 }
11386 \cs_new_protected:Npn \peek_N_type:TF
11387 {
11388 _peek_token_generic:NNTF
11389 _peek_execute_branches_N_type: \scan_stop:
11390 }
11391 \cs_new_protected:Npn \peek_N_type:T
11392 { _peek_token_generic:NNT _peek_execute_branches_N_type: \scan_stop: }
11393 \cs_new_protected:Npn \peek_N_type:F
11394 { _peek_token_generic:NNF _peek_execute_branches_N_type: \scan_stop: }

(End definition for \peek_N_type:TF and others. This function is documented on page 141.)

11395 </tex>
11396 </package>

```

## 18 I3prop implementation

The following test files are used for this code: *m3prop001*, *m3prop002*, *m3prop003*, *m3prop004*, *m3show001*.

```

11397 <*package>
11398 <@=prop>

```

A property list is a macro whose top-level expansion is of the form

```

\s_prop _prop_pair:wn <key1> \s_prop {<value1>}
...
_prop_pair:wn <keyn> \s_prop {<valuen>}

```

where `\s_prop` is a scan mark (equal to `\scan_stop:`), and `\_prop_pair:wn` can be used to map through the property list.

`\s_prop` The internal token used at the beginning of property lists. This is also used after each `<key>` (see `\_prop_pair:wn`).

(End definition for `\s_prop`.)

|                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>\__prop_pair:wn</code>                | <code>\__prop_pair:wn &lt;key&gt; \s__prop {&lt;item&gt;}</code><br>The internal token used to begin each key–value pair in the property list. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.<br><i>(End definition for \__prop_pair:wn.)</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>\l__prop_internal_t1</code>           | Token list used to store new key–value pairs to be inserted by functions of the <code>\prop_put:Nnn</code> family.<br><i>(End definition for \l__prop_internal_t1.)</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <hr/> <code>\__prop_split:NnTF</code> <hr/> | <code>\__prop_split:NnTF &lt;property list&gt; {&lt;key&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <small>Updated: 2013-01-08</small>          | Splits the <code>&lt;property list&gt;</code> at the <code>&lt;key&gt;</code> , giving three token lists: the <code>&lt;extract&gt;</code> of <code>&lt;property list&gt;</code> before the <code>&lt;key&gt;</code> , the <code>&lt;value&gt;</code> associated with the <code>&lt;key&gt;</code> and the <code>&lt;extract&gt;</code> of the <code>&lt;property list&gt;</code> after the <code>&lt;value&gt;</code> . Both <code>&lt;extracts&gt;</code> retain the internal structure of a property list, and the concatenation of the two <code>&lt;extracts&gt;</code> is a property list. If the <code>&lt;key&gt;</code> is present in the <code>&lt;property list&gt;</code> then the <code>&lt;true code&gt;</code> is left in the input stream, with #1, #2, and #3 replaced by the first <code>&lt;extract&gt;</code> , the <code>&lt;value&gt;</code> , and the second extract. If the <code>&lt;key&gt;</code> is not present in the <code>&lt;property list&gt;</code> then the <code>&lt;false code&gt;</code> is left in the input stream, with no trailing material. Both <code>&lt;true code&gt;</code> and <code>&lt;false code&gt;</code> are used in the replacement text of a macro defined internally, hence macro parameter characters should be doubled, except #1, #2, and #3 which stand in the <code>&lt;true code&gt;</code> for the three extracts from the property list. The <code>&lt;key&gt;</code> comparison takes place as described for <code>\str_if_eq:nn</code> . |
| <code>\s__prop</code>                       | A private scan mark is used as a marker after each key, and at the very beginning of the property list.<br><code>11399 \scan_new:N \s__prop</code><br><i>(End definition for \s__prop.)</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>\__prop_pair:wn</code>                | The delimiter is always defined, but when misused simply triggers an error and removes its argument.<br><code>11400 \cs_new:Npn \__prop_pair:wn #1 \s__prop #2</code><br><code>11401 { \__kernel_msg_expandable_error:nn { kernel } { misused-prop } }</code><br><i>(End definition for \__prop_pair:wn.)</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>\l__prop_internal_t1</code>           | Token list used to store the new key–value pair inserted by <code>\prop_put:Nnn</code> and friends.<br><code>11402 \tl_new:N \l__prop_internal_t1</code><br><i>(End definition for \l__prop_internal_t1.)</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>\c_empty_prop</code>                  | An empty prop.<br><code>11403 \tl_const:Nn \c_empty_prop { \s__prop }</code><br><i>(End definition for \c_empty_prop. This variable is documented on page 150.)</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

## 18.1 Internal auxiliaries

\s\_\_prop\_mark Internal scan marks.

```
11404 \scan_new:N \s__prop_mark
11405 \scan_new:N \s__prop_stop
```

(End definition for \s\_\_prop\_mark and \s\_\_prop\_stop.)

\q\_\_prop\_recursion\_tail Internal recursion quarks.

```
11406 \quark_new:N \q__prop_recursion_tail
11407 \quark_new:N \q__prop_recursion_stop
```

(End definition for \q\_\_prop\_recursion\_tail and \q\_\_prop\_recursion\_stop.)

\\_prop\_if\_recursion\_tail\_stop:n Functions to query recursion quarks.

```
11408 __kernel_quark_new_test:N _prop_if_recursion_tail_stop:n
11409 \cs_generate_variant:Nn _prop_if_recursion_tail_stop:n { o }
```

(End definition for \\_prop\_if\_recursion\_tail\_stop:n and \\_prop\_if\_recursion\_tail\_stop:o.)

## 18.2 Allocation and initialisation

\prop\_new:N Property lists are initialized with the value \c\_empty\_prop.

```
11410 \cs_new_protected:Npn \prop_new:N #1
11411 {
11412 __kernel_chk_if_free_cs:N #1
11413 \cs_gset_eq:NN #1 \c_empty_prop
11414 }
11415 \cs_generate_variant:Nn \prop_new:N { c }
```

(End definition for \prop\_new:N. This function is documented on page 144.)

\prop\_clear:N The same idea for clearing.

```
11416 \cs_new_protected:Npn \prop_clear:N #1
11417 { \prop_set_eq:NN #1 \c_empty_prop }
11418 \cs_generate_variant:Nn \prop_clear:N { c }
11419 \cs_new_protected:Npn \prop_gclear:N #1
11420 { \prop_gset_eq:NN #1 \c_empty_prop }
11421 \cs_generate_variant:Nn \prop_gclear:N { c }
```

(End definition for \prop\_clear:N and \prop\_gclear:N. These functions are documented on page 144.)

\prop\_clear\_new:N Once again a simple variation of the token list functions.

```
11422 \cs_new_protected:Npn \prop_clear_new:N #1
11423 { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
11424 \cs_generate_variant:Nn \prop_clear_new:N { c }
11425 \cs_new_protected:Npn \prop_gclear_new:N #1
11426 { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
11427 \cs_generate_variant:Nn \prop_gclear_new:N { c }
```

(End definition for \prop\_clear\_new:N and \prop\_gclear\_new:N. These functions are documented on page 144.)

```

\prop_set_eq:NN These are simply copies from the token list functions.
\prop_set_eq:cN 11428 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
\prop_set_eq:Nc 11429 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
\prop_set_eq:cc 11430 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
\prop_gset_eq:NN 11431 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
\prop_gset_eq:cN 11432 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
\prop_gset_eq:Nc 11433 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
\prop_gset_eq:cc 11434 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
11435 \cs_new_eq:NN \prop_gset_eq:cc \tl_gset_eq:cc

```

(End definition for `\prop_set_eq:NN` and `\prop_gset_eq:NN`. These functions are documented on page 144.)

```

\l_tmpa_prop We can now initialize the scratch variables.
\l_tmpb_prop 11436 \prop_new:N \l_tmpa_prop
\g_tmpa_prop 11437 \prop_new:N \l_tmpb_prop
\g_tmpb_prop 11438 \prop_new:N \g_tmpa_prop
11439 \prop_new:N \g_tmpb_prop

```

(End definition for `\l_tmpa_prop` and others. These variables are documented on page 149.)

```
\l__prop_internal_prop Property list used by \prop_set_from_keyval:Nn and others.
```

```
11440 \prop_new:N \l__prop_internal_prop
```

(End definition for `\l__prop_internal_prop`.)

```

\prop_set_from_keyval:Nn To avoid tracking throughout the loop the variable name and whether the assignment
\prop_set_from_keyval:cn is local/global, do everything in a scratch variable and empty it afterwards to avoid
\prop_gset_from_keyval:Nn wasting memory. Loop through items separated by commas, with \prg_do_nothing:
\prop_gset_from_keyval:cn to avoid losing braces. After checking for termination, split the item at the first and
\prop_const_from_keyval:Nn then at the second = (which ought to be the first of the trailing = that we added). For
\prop_const_from_keyval:cn both splits trim spaces and call a function (first __prop_from_keyval_key:w then __-
__prop_from_keyval_value:w), followed by the trimmed material, \s__prop_mark, the
__prop_from_keyval_loop:w subsequent part of the item, and the trailing =’s and \s__prop_stop. After finding
__prop_from_keyval_split:Nw the \key just store it after \s__prop_stop. After finding the \value ignore completely
__prop_from_keyval_key:n empty items (both trailing = were used as delimiters and all parts are empty); if the
__prop_from_keyval_key:w remaining part #2 consists exactly of the second trailing = (namely there was exactly one
__prop_from_keyval_value:n = in the item) then output one key–value pair for the property list; otherwise complain
__prop_from_keyval_value:w about a missing or extra =.

```

```

11441 \cs_new_protected:Npn \prop_set_from_keyval:Nn #1#2
11442 {
11443 \prop_clear:N \l__prop_internal_prop
11444 __prop_from_keyval:n {#2}
11445 \prop_set_eq:NN #1 \l__prop_internal_prop
11446 \prop_clear:N \l__prop_internal_prop
11447 }
11448 \cs_generate_variant:Nn \prop_set_from_keyval:Nn { c }
11449 \cs_new_protected:Npn \prop_gset_from_keyval:Nn #1#2
11450 {
11451 \prop_clear:N \l__prop_internal_prop
11452 __prop_from_keyval:n {#2}
11453 \prop_gset_eq:NN #1 \l__prop_internal_prop
11454 \prop_clear:N \l__prop_internal_prop

```

```

11455 }
11456 \cs_generate_variant:Nn \prop_gset_from_keyval:Nn { c }
11457 \cs_new_protected:Npn \prop_const_from_keyval:Nn #1#2
11458 {
11459 \prop_clear:N \l__prop_internal_prop
11460 __prop_from_keyval:n {#2}
11461 \tl_const:Nx #1 { \exp_not:o \l__prop_internal_prop }
11462 \prop_clear:N \l__prop_internal_prop
11463 }
11464 \cs_generate_variant:Nn \prop_const_from_keyval:Nn { c }
11465 \cs_new_protected:Npn __prop_from_keyval:n #1
11466 {
11467 __prop_from_keyval_loop:w \prg_do_nothing: #1 ,
11468 \q__prop_recursion_tail , \q__prop_recursion_stop
11469 }
11470 \cs_new_protected:Npn __prop_from_keyval_loop:w #1 ,
11471 {
11472 __prop_if_recursion_tail_stop:o {#1}
11473 __prop_from_keyval_split:Nw __prop_from_keyval_key:n
11474 #1 = \s__prop_stop {#1}
11475 __prop_from_keyval_loop:w \prg_do_nothing:
11476 }
11477 \cs_new_protected:Npn __prop_from_keyval_split:Nw #1#2 =
11478 { \tl_trim_spaces_apply:oN {#2} #1 }
11479 \cs_new_protected:Npn __prop_from_keyval_key:n #1
11480 { __prop_from_keyval_key:w #1 \s__prop_mark }
11481 \cs_new_protected:Npn __prop_from_keyval_key:w #1 \s__prop_mark #2 \s__prop_stop
11482 {
11483 __prop_from_keyval_split:Nw __prop_from_keyval_value:n
11484 \prg_do_nothing: #2 \s__prop_stop {#1}
11485 }
11486 \cs_new_protected:Npn __prop_from_keyval_value:n #1
11487 { __prop_from_keyval_value:w #1 \s__prop_mark }
11488 \cs_new_protected:Npn __prop_from_keyval_value:w #1 \s__prop_mark #2 \s__prop_stop #3#4
11489 {
11490 \tl_if_empty:nF { #3 #1 #2 }
11491 {
11492 \str_if_eq:nnTF {#2} { = }
11493 { \prop_put:Nnn \l__prop_internal_prop {#3} {#1} }
11494 {
11495 __kernel_msg_error:nnx { kernel } { prop-keyval }
11496 { \exp_not:o {#4} }
11497 }
11498 }
11499 }

```

(End definition for `\prop_set_from_keyval:Nn` and others. These functions are documented on page [144](#).)

### 18.3 Accessing data in property lists

`\__prop_split:NnTF`, `\__prop_split_aux:NnTF`, `\__prop_split_aux:w`

This function is used by most of the module, and hence must be fast. It receives a *property list*, a *key*, a *true code* and a *false code*. The aim is to split the *property list* at the given *key* into the *extract<sub>i</sub>* before the key–value pair, the *value* associated

with the  $\langle key \rangle$  and the  $\langle extract_2 \rangle$  after the key–value pair. This is done using a delimited function, whose definition is as follows, where the  $\langle key \rangle$  is turned into a string.

```
\cs_set:Npn __prop_split_aux:w #1
 __prop_pair:wn <key> \s__prop #2
 #3 \s__prop_mark #4 #5 \s__prop_stop
 { #4 {<true code>} {<false code>} }
```

If the  $\langle key \rangle$  is present in the property list,  $\__prop_split_aux:w$ 's #1 is the part before the  $\langle key \rangle$ , #2 is the  $\langle value \rangle$ , #3 is the part after the  $\langle key \rangle$ , #4 is  $\use_i:nn$ , and #5 is additional tokens that we do not care about. The  $\langle true code \rangle$  is left in the input stream, and can use the parameters #1, #2, #3 for the three parts of the property list as desired. Namely, the original property list is in this case #1  $\__prop_pair:wn <key>$   $\s__prop$  {#2} #3.

If the  $\langle key \rangle$  is not there, then the  $\langle function \rangle$  is  $\use_{ii}:nn$ , which keeps the  $\langle false code \rangle$ .

```
11500 \cs_new_protected:Npn __prop_split:NnTF #1#2
11501 { \exp_args:NNo __prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
11502 \cs_new_protected:Npn __prop_split_aux:NnTF #1#2#3#4
11503 {
11504 \cs_set:Npn __prop_split_aux:w ##1
11505 __prop_pair:wn #2 \s__prop ##2 ##3 \s__prop_mark ##4 ##5 \s__prop_stop
11506 { ##4 {#3} {#4} }
11507 \exp_after:wN __prop_split_aux:w #1 \s__prop_mark \use_i:nn
11508 __prop_pair:wn #2 \s__prop { } \s__prop_mark \use_{ii}:nn \s__prop_stop
11509 }
11510 \cs_new:Npn __prop_split_aux:w { }
```

(End definition for  $\__prop_split:NnTF$ ,  $\__prop_split_aux:NnTF$ , and  $\__prop_split_aux:w$ .)

**\prop\_remove:Nn**  
**\prop\_remove:NV**  
**\prop\_remove:cn**  
**\prop\_remove:cV**

Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.

```
11511 \cs_new_protected:Npn \prop_remove:Nn #1#2
11512 {
11513 __prop_split:NnTF #1 {#2}
11514 { \tl_set:Nn #1 { ##1 ##3 } }
11515 { }
11516 }
11517 \cs_new_protected:Npn \prop_gremove:Nn #1#2
11518 {
11519 __prop_split:NnTF #1 {#2}
11520 { \tl_gset:Nn #1 { ##1 ##3 } }
11521 { }
11522 }
11523 \cs_generate_variant:Nn \prop_remove:Nn { NV }
11524 \cs_generate_variant:Nn \prop_remove:Nn { c , cV }
11525 \cs_generate_variant:Nn \prop_gremove:Nn { NV }
11526 \cs_generate_variant:Nn \prop_gremove:Nn { c , cV }
```

(End definition for  $\prop_remove:Nn$  and  $\prop_gremove:Nn$ . These functions are documented on page 146.)

**\prop\_get:NnN** Getting an item from a list is very easy: after splitting, if the key is in the property list, just set the token list variable to the return value, otherwise to `\q_no_value`.

```

\prop_get:NVN
\prop_get:NoN
\prop_get:cN
\prop_get:cVN
\prop_get:coN
11527 \cs_new_protected:Npn \prop_get:NnN #1#2#3
11528 {
11529 __prop_split:NnTF #1 {##2}
11530 { \tl_set:Nn #3 {##2} }
11531 { \tl_set:Nn #3 { \q_no_value } }
11532 }
11533 \cs_generate_variant:Nn \prop_get:NnN { NV , Nv , No }
11534 \cs_generate_variant:Nn \prop_get:NnN { c , cV , cv , co }
```

(End definition for `\prop_get:NnN`. This function is documented on page 145.)

**\prop\_pop:NnN** Popping a value also starts by doing the split. If the key is present, save the value in the token list and update the property list as when deleting. If the key is missing, save `\q_no_value` in the token list.

```

\prop_pop:NoN
\prop_pop:cN
\prop_pop:coN
\prop_gpop:NnN
\prop_gpop:NoN
\prop_gpop:cN
\prop_gpop:coN
11535 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
11536 {
11537 __prop_split:NnTF #1 {##2}
11538 {
11539 \tl_set:Nn #3 {##2}
11540 \tl_set:Nn #1 { ##1 ##3 }
11541 }
11542 { \tl_set:Nn #3 { \q_no_value } }
11543 }
11544 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
11545 {
11546 __prop_split:NnTF #1 {##2}
11547 {
11548 \tl_set:Nn #3 {##2}
11549 \tl_gset:Nn #1 { ##1 ##3 }
11550 }
11551 { \tl_set:Nn #3 { \q_no_value } }
11552 }
11553 \cs_generate_variant:Nn \prop_pop:NnN { No }
11554 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
11555 \cs_generate_variant:Nn \prop_gpop:NnN { No }
11556 \cs_generate_variant:Nn \prop_gpop:NnN { c , co }
```

(End definition for `\prop_pop:NnN` and `\prop_gpop:NnN`. These functions are documented on page 145.)

**\prop\_item:Nn** Getting the value corresponding to a key in a property list in an expandable fashion is similar to mapping some tokens. Go through the property list one  $\langle key \rangle$ - $\langle value \rangle$  pair at a time: the arguments of `\__prop_item_Nn:nwn` are the  $\langle key \rangle$  we are looking for, a  $\langle key \rangle$  of the property list, and its associated value. The  $\langle keys \rangle$  are compared (as strings). If they match, the  $\langle value \rangle$  is returned, within `\exp_not:n`. The loop terminates even if the  $\langle key \rangle$  is missing, and yields an empty value, because we have appended the appropriate  $\langle key \rangle$ - $\langle empty value \rangle$  pair to the property list.

```

\prop_item:cn
__prop_item_Nn:nwn
11557 \cs_new:Npn \prop_item:Nn #1#2
11558 {
11559 \exp_last_unbraced:Noo __prop_item_Nn:nwn { \tl_to_str:n {##2} } #1
11560 __prop_pair:wn \tl_to_str:n {##2} \s__prop { }
11561 \prg_break_point:
```

```

11562 }
11563 \cs_new:Npn __prop_item:Nn:nwwn #1#2 __prop_pair:wn #3 \s__prop #4
11564 {
11565 \str_if_eq:eeTF {#1} {#3}
11566 { \prg_break:n { \exp_not:n {#4} } }
11567 { __prop_item:Nn:nwwn {#1} }
11568 }
11569 \cs_generate_variant:Nn \prop_item:Nn { c }

```

(End definition for `\prop_item:Nn` and `\__prop_item:Nn:nwwn`. This function is documented on page 146.)

**\prop\_count:N** Counting the key–value pairs in a property list is done using the same approach as for other count functions: turn each entry into a `+1` then use integer evaluation to actually do the mathematics.

```

11570 \cs_new:Npn \prop_count:N #1
11571 {
11572 \int_eval:n
11573 {
11574 0
11575 \prop_map_function:NN #1 __prop_count:nn
11576 }
11577 }
11578 \cs_new:Npn __prop_count:nn #1#2 { + 1 }
11579 \cs_generate_variant:Nn \prop_count:N { c }

```

(End definition for `\prop_count:N` and `\__prop_count:nn`. This function is documented on page 146.)

**\prop\_pop:NnNTF** Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, `\prg_return_true:` is used after the assignments.

**\prop\_gpop:cnNTF**

```

11580 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
11581 {
11582 __prop_split:NnTF #1 {#2}
11583 {
11584 \tl_set:Nn #3 {##2}
11585 \tl_set:Nn #1 { ##1 ##3 }
11586 \prg_return_true:
11587 }
11588 { \prg_return_false: }
11589 }
11590 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
11591 {
11592 __prop_split:NnTF #1 {#2}
11593 {
11594 \tl_set:Nn #3 {##2}
11595 \tl_gset:Nn #1 { ##1 ##3 }
11596 \prg_return_true:
11597 }
11598 { \prg_return_false: }
11599 }
11600 \prg_generate_conditional_variant:Nnn \prop_pop:NnN { c } { T , F , TF }
11601 \prg_generate_conditional_variant:Nnn \prop_gpop:NnN { c } { T , F , TF }

```

(End definition for `\prop_pop:NnNTF` and `\prop_gpop:NnNTF`. These functions are documented on page 147.)

`\prop_put:Nnn` Since the branches of `\_prop_split:NnTF` are used as the replacement text of an internal macro, and since the `<key>` and new `<value>` may contain arbitrary tokens, it is not safe to include them in the argument of `\_prop_split:NnTF`. We thus start by storing in `\l\_prop_internal_tl` tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in `\_prop_split:NnTF`. If the `<key>` was absent, append the new key–value to the list. Otherwise concatenate the extracts `##1` and `##3` with the new key–value pair `\l\_prop_internal_tl`. The updated entry is placed at the same spot as the original `<key>` in the property list, preserving the order of entries.

```

11602 \cs_new_protected:Npn \prop_put:Nnn { _prop_put>NNnn _kernel_tl_set:Nx }
11603 \cs_new_protected:Npn \prop_gput:Nnn { _prop_put>NNnn _kernel_tl_gset:Nx }
11604 \cs_new_protected:Npn _prop_put>NNnn #1#2#3#4
11605 {
11606 \tl_set:Nn \l_prop_internal_tl
11607 {
11608 \exp_not:N _prop_pair:wn \tl_to_str:n {#3}
11609 \s__prop { \exp_not:n {#4} }
11610 }
11611 _prop_split:NnTF #2 {#3}
11612 { #1 #2 { \exp_not:n {##1} \l_prop_internal_tl \exp_not:n {##3} } }
11613 { #1 #2 { \exp_not:o {#2} \l_prop_internal_tl } }
11614 }
11615 \cs_generate_variant:Nn \prop_put:Nnn
11616 { NnV , Nno , Nnx , NV , NVV , NVx , Nvx , No , Noo , Nxx }
11617 \cs_generate_variant:Nn \prop_put:Nnn
11618 { c , cnV , cno , cnx , cV , cVV , cVx , cvx , co , coo , cxx }
11619 \cs_generate_variant:Nn \prop_gput:Nnn
11620 { NnV , Nno , Nnx , NV , NVV , NVx , Nvx , No , Noo , Nxx }
11621 \cs_generate_variant:Nn \prop_gput:Nnn
11622 { c , cnV , cno , cnx , cV , cVV , cVx , cvx , co , coo , cxx }

```

(End definition for `\prop_put:Nnn` and others. These functions are documented on page 145.)

Adding conditionally also splits. If the key is already present, the three brace groups given by `\_prop_split:NnTF` are removed. If the key is new, then the value is added, being careful to convert the key to a string using `\tl_to_str:n`.

```

11623 \cs_new_protected:Npn \prop_put_if_new:Nnn
11624 { _prop_put_if_new>NNnn _kernel_tl_set:Nx }
11625 \cs_new_protected:Npn \prop_gput_if_new:Nnn
11626 { _prop_put_if_new>NNnn _kernel_tl_gset:Nx }
11627 \cs_new_protected:Npn _prop_put_if_new>NNnn #1#2#3#4
11628 {
11629 \tl_set:Nn \l_prop_internal_tl
11630 {
11631 \exp_not:N _prop_pair:wn \tl_to_str:n {#3}
11632 \s__prop { \exp_not:n {#4} }
11633 }
11634 _prop_split:NnTF #2 {#3}
11635 { }
11636 { #1 #2 { \exp_not:o {#2} \l_prop_internal_tl } }
11637 }

```

```

11638 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
11639 \cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }

(End definition for \prop_put_if_new:Nnn, \prop_gput_if_new:Nnn, and __prop_put_if_new:Nnnn.
These functions are documented on page 145.)
```

## 18.4 Property list conditionals

\prop\_if\_exist\_p:N Copies of the `cs` functions defined in l3basics.

```

11640 \prg_new_eq_conditional:NNN \prop_if_exist:N \cs_if_exist:N
11641 { TF , T , F , p }
11642 \prg_new_eq_conditional:NNN \prop_if_exist:c \cs_if_exist:c
11643 { TF , T , F , p }
```

(End definition for \prop\_if\_exist:NTF. This function is documented on page 146.)

\prop\_if\_empty\_p:N Same test as for token lists.

```

11644 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
11645 {
11646 \tl_if_eq:NNTF #1 \c_empty_prop
11647 \prg_return_true: \prg_return_false:
11648 }
11649 \prg_generate_conditional_variant:Nnn \prop_if_empty:N
11650 { c } { p , T , F , TF }
```

(End definition for \prop\_if\_empty:NTF. This function is documented on page 146.)

\prop\_if\_in\_p:Nn Testing expandably if a key is in a property list requires to go through the key–value pairs one by one. This is rather slow, and a faster test would be

```

\prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
{
 __split:NnTF #1 {#2}
 { \prg_return_true: }
 { \prg_return_false: }
}
```

but `\__prop_split:NnTF` is non-expandable.

Instead, the key is compared to each key in turn using `\str_if_eq:ee`, which is expandable. To terminate the mapping, we append to the property list the key that is searched for. This second `\tl_to_str:n` is not expanded at the start, but only when included in the `\str_if_eq:ee`. It cannot make the breaking mechanism choke, because the arbitrary token list material is enclosed in braces. The second argument of `\__prop_if_in:nwwn` is most often empty. When the `\langle key` is found in the list, `\__prop_if_in:N` receives `\__prop_pair:wn`, and if it is found as the extra item, the function receives `\q__prop_recursion_tail`, easily recognizable.

Here, `\prop_map_function>NN` is not sufficient for the mapping, since it can only map a single token, and cannot carry the key that is searched for.

```

11651 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
11652 {
11653 \exp_last_unbraced:Noo __prop_if_in:nwwn { \tl_to_str:n {#2} } #1
11654 __prop_pair:wn \tl_to_str:n {#2} \s__prop { }
11655 \q__prop_recursion_tail
11656 \prg_break_point:
```

```

11657 }
11658 \cs_new:Npn __prop_if_in:nwwn #1#2 __prop_pair:wn #3 \s__prop #4
11659 {
11660 \str_if_eq:eeTF {#1} {#3}
11661 { __prop_if_in:N }
11662 { __prop_if_in:nwwn {#1} }
11663 }
11664 \cs_new:Npn __prop_if_in:N #1
11665 {
11666 \if_meaning:w \q__prop_recursion_tail #1
11667 \prg_return_false:
11668 \else:
11669 \prg_return_true:
11670 \fi:
11671 \prg_break:
11672 }
11673 \prg_generate_conditional_variant:Nnn \prop_if_in:Nn
11674 { NV , No , c , cV , co } { p , T , F , TF }

```

(End definition for `\prop_if_in:NnTF`, `\__prop_if_in:nwwn`, and `\__prop_if_in:N`. This function is documented on page 147.)

## 18.5 Recovering values from property lists with branching

`\prop_get:NnNTF` Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

```

11675 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
11676 {
11677 __prop_split:NnTF #1 {#2}
11678 {
11679 \tl_set:Nn #3 {##2}
11680 \prg_return_true:
11681 }
11682 { \prg_return_false: }
11683 }
11684 \prg_generate_conditional_variant:Nnn \prop_get:NnN
11685 { NV , Nv , No , c , cV , cv , co } { T , F , TF }

```

(End definition for `\prop_get:NnNTF`. This function is documented on page 147.)

## 18.6 Mapping to property lists

`\prop_map_function:NN` The argument delimited by `\__prop_pair:wn` is empty except at the end of the loop where it is `\prg_break:`. No need for any quark test.

```

11686 \cs_new:Npn \prop_map_function:NN #1#2
11687 {
11688 \exp_after:wN \use_i_i:nnn
11689 \exp_after:wN __prop_map_function:Nwwn
11690 \exp_after:wN #2
11691 #1
11692 \prg_break: __prop_pair:wn \s__prop { } \prg_break_point:
11693 \prg_break_point:Nn \prop_map_break: { }
11694 }

```

```

11695 \cs_new:Npn __prop_map_function:Nwwn #1#2 __prop_pair:wn #3 \s__prop #4
11696 {
11697 #2
11698 #1 {#3} {#4}
11699 __prop_map_function:Nwwn #1
11700 }
11701 \cs_generate_variant:Nn \prop_map_function:NN { Nc , c , cc }

```

(End definition for `\prop_map_function:NN` and `\__prop_map_function:Nwwn`. This function is documented on page 148.)

`\prop_map_inline:Nn`  
`\prop_map_inline:cn`

Mapping in line requires a nesting level counter. Store the current definition of `\__prop_pair:wn`, and define it anew. At the end of the loop, revert to the earlier definition. Note that besides pairs of the form `\__prop_pair:wn <key> \s__prop {<value>}`, there are a leading and a trailing tokens, but both are equal to `\scan_stop:`, hence have no effect in such inline mapping. Such `\scan_stop:` could have affected ligatures if they appeared during the mapping.

```

11702 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
11703 {
11704 \cs_gset_eq:cN
11705 { __prop_map_ \int_use:N \g_kernel_prg_map_int :wn } __prop_pair:wn
11706 \int_gincr:N \g_kernel_prg_map_int
11707 \cs_gset_protected:Npn __prop_pair:wn ##1 \s__prop ##2 {#2}
11708 #1
11709 \prg_break_point:Nn \prop_map_break:
11710 {
11711 \int_gdecr:N \g_kernel_prg_map_int
11712 \cs_gset_eq:Nc __prop_pair:wn
11713 { __prop_map_ \int_use:N \g_kernel_prg_map_int :wn }
11714 }
11715 }
11716 \cs_generate_variant:Nn \prop_map_inline:Nn { c }

```

(End definition for `\prop_map_inline:Nn`. This function is documented on page 148.)

`\prop_map_tokens:Nn`  
`\prop_map_tokens:cn`  
`\__prop_map_tokens:nwwn`

The mapping is very similar to `\prop_map_function:NN`. The `\use_i:nn` removes the leading `\s__prop`. The odd construction `\use:n {#1}` allows `#1` to contain any token without interfering with `\prop_map_break:`. The loop stops when the argument delimited by `\__prop_pair:wn` is `\prg_break:` instead of being empty.

```

11717 \cs_new:Npn \prop_map_tokens:Nn #1#2
11718 {
11719 \exp_last_unbraced:Nno
11720 \use_i:nn { __prop_map_tokens:nwwn {#2} } #1
11721 \prg_break: __prop_pair:wn \s__prop { } \prg_break_point:
11722 \prg_break_point:Nn \prop_map_break: { }
11723 }
11724 \cs_new:Npn __prop_map_tokens:nwwn #1#2 __prop_pair:wn #3 \s__prop #4
11725 {
11726 #2
11727 \use:n {#1} {#3} {#4}
11728 __prop_map_tokens:nwwn {#1}
11729 }
11730 \cs_generate_variant:Nn \prop_map_tokens:Nn { c }

```

(End definition for `\prop_map_tokens:Nn` and `\_prop_map_tokens:nwn`. This function is documented on page 148.)

`\prop_map_break:` The break statements are based on the general `\prg_map_break:Nn`.

```
11731 \cs_new:Npn \prop_map_break:
11732 { \prg_map_break:Nn \prop_map_break: { } }
11733 \cs_new:Npn \prop_map_break:n
11734 { \prg_map_break:Nn \prop_map_break: }
```

(End definition for `\prop_map_break:` and `\prop_map_break:n`. These functions are documented on page 148.)

## 18.7 Viewing property lists

`\prop_show:N` Apply the general `\_kernel_chk_defined:NT` and `\msg_show:nnnnnn`. Contrarily to sequences and comma lists, we use `\msg_show_item:nn` to format both the key and the value for each pair.

```
11735 \cs_new_protected:Npn \prop_show:N { _prop_show:NN \msg_show:nnxxxx }
11736 \cs_generate_variant:Nn \prop_show:N { c }
11737 \cs_new_protected:Npn \prop_log:N { _prop_show:NN \msg_log:nnxxxx }
11738 \cs_generate_variant:Nn \prop_log:N { c }
11739 \cs_new_protected:Npn _prop_show:NN #1#2
11740 {
11741 _kernel_chk_defined:NT #2
11742 {
11743 #1 { LaTeX/kernel } { show-prop }
11744 { \token_to_str:N #2 }
11745 { \prop_map_function:NN #2 \msg_show_item:nn }
11746 { } { }
11747 }
11748 }
```

(End definition for `\prop_show:N` and `\prop_log:N`. These functions are documented on page 149.)

```
11749 </package>
```

## 19 I3msg implementation

```
11750 <*package>
11751 <@=msg>
```

`\l__msg_internal_tl` A general scratch for the module.

```
11752 \tl_new:N \l__msg_internal_tl
11753 (End definition for \l__msg_internal_tl.)
```

`\l__msg_name_str` Used to save module info when creating messages.

```
11753 \str_new:N \l__msg_name_str
11754 \str_new:N \l__msg_text_str
```

(End definition for `\l__msg_name_str` and `\l__msg_text_str`.)

## 19.1 Internal auxiliaries

\s\_\_msg\_mark Internal scan marks.

```
11755 \scan_new:N \s__msg_mark
11756 \scan_new:N \s__msg_stop
```

(End definition for \s\_\_msg\_mark and \s\_\_msg\_stop.)

\\_\_msg\_use\_none\_delimit\_by\_s\_stop:w Functions to gobble up to a scan mark.

```
11757 \cs_new:Npn __msg_use_none_delimit_by_s_stop:w #1 \s__msg_stop { }
```

(End definition for \\_\_msg\_use\_none\_delimit\_by\_s\_stop:w.)

## 19.2 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

\c\_\_msg\_text\_prefix\_tl Locations for the text of messages.

```
11758 \tl_const:Nn \c__msg_text_prefix_tl { msg-text~>~ }
11759 \tl_const:Nn \c__msg_more_text_prefix_tl { msg-extra-text~>~ }
```

(End definition for \c\_\_msg\_text\_prefix\_tl and \c\_\_msg\_more\_text\_prefix\_tl.)

\msg\_if\_exist\_p:nn Test whether the control sequence containing the message text exists or not.

```
11760 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
11761 {
11762 \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
11763 { \prg_return_true: } { \prg_return_false: }
11764 }
```

(End definition for \msg\_if\_exist:nnTF. This function is documented on page 152.)

\\_\_msg\_chk\_if\_free:nn This auxiliary is similar to \\_\_kernel\_chk\_if\_free\_cs:N, and is used when defining messages with \msg\_new:nnnn.

```
11765 \cs_new_protected:Npn __msg_chk_free:nn #1#2
11766 {
11767 \msg_if_exist:nnT {#1} {#2}
11768 {
11769 __kernel_msg_error:nxxx { kernel } { message-already-defined }
11770 {#1} {#2}
11771 }
11772 }
```

(End definition for \\_\_msg\_chk\_if\_free:nn.)

\msg\_new:nnnn \msg\_new:nnn Setting a message simply means saving the appropriate text into two functions. A sanity check first.

```
11773 \cs_new_protected:Npn \msg_new:nnnn #1#2
11774 {
11775 __msg_chk_free:nn {#1} {#2}
11776 \msg_gset:nnnn {#1} {#2}
11777 }
11778 \cs_new_protected:Npn \msg_new:nnn #1#2#3
```

```

11779 { \msg_new:nnnn {#1} {#2} {#3} { } }
1180 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
1181 {
1182 \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
1183 ##1##2##3##4 {#3}
1184 \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
1185 ##1##2##3##4 {#4}
1186 }
1187 \cs_new_protected:Npn \msg_set:nnn #1#2#3
1188 { \msg_set:nnnn {#1} {#2} {#3} { } }
1189 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
1190 {
1191 \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
1192 ##1##2##3##4 {#3}
1193 \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
1194 ##1##2##3##4 {#4}
1195 }
1196 \cs_new_protected:Npn \msg_gset:nnn #1#2#3
1197 { \msg_gset:nnnn {#1} {#2} {#3} { } }

```

(End definition for `\msg_new:nnnn` and others. These functions are documented on page 151.)

### 19.3 Messages: support functions and text

Simple pieces of text for messages.

```

\c__msg_error_text_tl
\c__msg_continue_text_tl
\c__msg_critical_text_tl
\c__msg_fatal_text_tl
\c__msg_help_text_tl
\c__msg_no_info_text_tl
\c__msg_on_line_text_tl
\c__msg_return_text_tl
\c__msg_trouble_text_tl

11798 \tl_const:Nn \c__msg_error_text_tl
11799 {
1180 This-is-a-coding-error.
11801 \\ \\
11802 }
11803 \tl_const:Nn \c__msg_continue_text_tl
11804 { Type-<return>-to-continue }
11805 \tl_const:Nn \c__msg_critical_text_tl
11806 { Reading-the-current-file-'g_file_curr_name_str'-will-stop. }
11807 \tl_const:Nn \c__msg_fatal_text_tl
11808 { This-is-a-fatal-error:-LaTeX-will-abort. }
11809 \tl_const:Nn \c__msg_help_text_tl
11810 { For-immediate-help-type-H-<return> }
11811 \tl_const:Nn \c__msg_no_info_text_tl
11812 {
11813 LaTeX-does-not-know-anything-more-about-this-error,-sorry.
11814 \c__msg_return_text_tl
11815 }
11816 \tl_const:Nn \c__msg_on_line_text_tl { on-line }
11817 \tl_const:Nn \c__msg_return_text_tl
11818 {
11819 \\ \\
11820 Try-typing-<return>-to-proceed.
11821 \\
11822 If-that-doesn't-work,-type-X-<return>-to-quit.
11823 }
11824 \tl_const:Nn \c__msg_trouble_text_tl
11825 {
11826 \\ \\

```

```

11827 More~errors~will~almost~certainly~follow: \\
11828 the~LaTeX~run~should~be~aborted.
11829 }
```

(End definition for `\c_msg_coding_error_text_tl` and others.)

`\msg_line_number:` For writing the line number nicely. `\msg_line_context:` was set up earlier, so this is not new.

```

11830 \cs_new:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
11831 \cs_gset:Npn \msg_line_context:
11832 {
11833 \c_msg_on_line_text_tl
11834 \c_space_tl
11835 \msg_line_number:
11836 }
```

(End definition for `\msg_line_number:` and `\msg_line_context:`. These functions are documented on page 152.)

## 19.4 Showing messages: low level mechanism

`\__msg_interrupt:Nnnn`  
`\__msg_no_more_text:nnnn`

The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup TeX's `\errhelp` register before issuing an `\errmessage`. To deal with the various cases of critical or fatal errors with and without help text, there is a bit of argument-passing to do.

```

11837 \cs_new_protected:Npn __msg_interrupt:NnnnN #1#2#3#4#5
11838 {
11839 \str_set:Nx \l__msg_text_str { #1 {#2} }
11840 \str_set:Nx \l__msg_name_str { \msg_module_name:n {#2} }
11841 \cs_if_eq:cNTF
11842 { \c__msg_more_text_prefix_tl #2 / #3 }
11843 __msg_no_more_text:nnnn
11844 {
11845 __msg_interrupt_wrap:nnn
11846 { \use:c { \c__msg_text_prefix_tl #2 / #3 } #4 }
11847 { \c__msg_continue_text_tl }
11848 {
11849 \c__msg_no_info_text_tl
11850 \tl_if_empty:NF #5
11851 { \\ \\ #5 }
11852 }
11853 }
11854 {
11855 __msg_interrupt_wrap:nnn
11856 { \use:c { \c__msg_text_prefix_tl #2 / #3 } #4 }
11857 { \c__msg_help_text_tl }
11858 {
11859 \use:c { \c__msg_more_text_prefix_tl #2 / #3 } #4
11860 \tl_if_empty:NF #5
11861 { \\ \\ #5 }
11862 }
11863 }
```

```

11864 }
11865 \cs_new:Npn __msg_no_more_text:nnnn #1#2#3#4 { }

```

(End definition for `\__msg_interrupt:Nnnn` and `\__msg_no_more_text:nnnn`.)

```
__msg_interrupt_wrap:nnn
__msg_interrupt_text:n
__msg_interrupt_more_text:n
```

First setup TeX's `\errhelp` register with the extra help #1, then build a nice-looking error message with #2. Everything is done using x-type expansion as the new line markers are different for the two type of text and need to be correctly set up. The auxiliary `\__msg_interrupt_more_text:n` receives its argument as a line-wrapped string, which is thus unaffected by expansion. We have to split the main text into two parts as only the "message" itself is wrapped with a leader: the generic help is wrapped at full width. We also have to allow for the two characters used by `\errmessage` itself.

```

11866 \cs_new_protected:Npn __msg_interrupt_wrap:nnn #1#2#3
11867 {
11868 \iow_wrap:nnN { \\ #3 } { } { } __msg_interrupt_more_text:n
11869 \group_begin:
11870 \int_sub:Nn \l_iow_line_count_int { 2 }
11871 \iow_wrap:nxnN { \l__msg_text_str : ~ #1 }
11872 {
11873 (\l__msg_name_str)
11874 \prg_replicate:nn
11875 {
11876 \str_count:N \l__msg_text_str
11877 - \str_count:N \l__msg_name_str
11878 + 2
11879 }
11880 { ~ }
11881 }
11882 { } __msg_interrupt_text:n
11883 \iow_wrap:nnN { \l__msg_internal_tl \\ \\ #2 } { } { }
11884 __msg_interrupt:
11885 }
11886 \cs_new_protected:Npn __msg_interrupt_text:n #1
11887 {
11888 \group_end:
11889 \tl_set:Nn \l__msg_internal_tl {#1}
11890 }
11891 \cs_new_protected:Npn __msg_interrupt_more_text:n #1
11892 { \exp_args:Nx \tex_errhelp:D { #1 \iow_newline: } }
```

(End definition for `\__msg_interrupt_wrap:nnn`, `\__msg_interrupt_text:n`, and `\__msg_interrupt_more_text:n`.)

```
__msg_interrupt:n
```

The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made "invisible": TeX's own information involves the macro in which `\errmessage` is called, and the end of the argument of the `\errmessage`, including the closing brace. We use an active ! to call the `\errmessage` primitive, and end its argument with `\use:none:n {<spaces>}` which fills the output with spaces. Two trailing closing braces are turned into spaces to hide them as well. The group in which we alter the definition of the active ! is closed before producing the message: this ensures that tokens inserted by typing I in the command-line are inserted after the message is entirely cleaned up.

The `\__kernel_iow_with:Nnn` auxiliary, defined in l3file, expects an *<integer variable>*, an integer *<value>*, and some *<code>*. It runs the *<code>* after ensuring that the

*<integer variable>* takes the given *<value>*, then restores the former value of the *<integer variable>* if needed. We use it to ensure that the `\newlinechar` is 10, as needed for `\iow_newline:` to work, and that `\errorcontextlines` is  $-1$ , to avoid showing irrelevant context. Note that restoring the former value of these integers requires inserting tokens after the `\errmessage`, which go in the way of tokens which could be inserted by the user. This is unavoidable.

```

11893 \group_begin:
11894 \char_set_lccode:nn { 38 } { 32 } % &
11895 \char_set_lccode:nn { 46 } { 32 } % .
11896 \char_set_lccode:nn { 123 } { 32 } % {
11897 \char_set_lccode:nn { 125 } { 32 } % }
11898 \char_set_catcode_active:N \&
11899 \tex_lowercase:D
11900 {
11901 \group_end:
11902 \cs_new_protected:Npn __msg_interrupt:n #1
11903 {
11904 \iow_term:n { }
11905 __kernel_iow_with:Nnn \tex_newlinechar:D { '\^J }
11906 {
11907 __kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
11908 {
11909 \group_begin:
11910 \cs_set_protected:Npn &
11911 {
11912 \tex_errmessage:D
11913 {
11914 #1
11915 \use_none:n
11916 { }
11917 }
11918 }
11919 \exp_after:wN
11920 \group_end:
11921 &
11922 }
11923 }
11924 }
11925 }

```

(End definition for `\__msg_interrupt:n`.)

## 19.5 Displaying messages

L<sup>A</sup>T<sub>E</sub>X is handling error messages and so the T<sub>E</sub>X ones are disabled.

```
11926 \int_gset:Nn \tex_errorcontextlines:D { -1 }
```

`\msg_fatal_text:n` A function for issuing messages: both the text and order could in principle vary. The  
`\msg_critical_text:n` module name may be empty for kernel messages, hence the slightly contorted code path  
`\msg_error_text:n` for a space.

```

\msg_warning_text:n 11927 \cs_new:Npn \msg_fatal_text:n #1
\msg_info_text:n 11928 {
__msg_text:nn 11929 Fatal ~
__msg_text:n

```

```

11930 \msg_error_text:n {#1}
11931 }
11932 \cs_new:Npn \msg_critical_text:n #1
11933 {
11934 Critical ~
11935 \msg_error_text:n {#1}
11936 }
11937 \cs_new:Npn \msg_error_text:n #1
11938 { __msg_text:nn {#1} { Error } }
11939 \cs_new:Npn \msg_warning_text:n #1
11940 { __msg_text:nn {#1} { Warning } }
11941 \cs_new:Npn \msg_info_text:n #1
11942 { __msg_text:nn {#1} { Info } }
11943 \cs_new:Npn __msg_text:nn #1#2
11944 {
11945 \exp_args:Nf __msg_text:n { \msg_module_type:n {#1} }
11946 \msg_module_name:n {#1} ~
11947 #2
11948 }
11949 \cs_new:Npn __msg_text:n #1
11950 {
11951 \tl_if_blank:nF {#1}
11952 { #1 ~ }
11953 }

```

(End definition for `\msg_fatal_text:n` and others. These functions are documented on page 152.)

`\g_msg_module_name_prop`  
`\g_msg_module_type_prop`

For storing public module information: the kernel data is set up in advance.

```

11954 \prop_new:N \g_msg_module_name_prop
11955 \prop_gput:Nnn \g_msg_module_name_prop { LaTeX } { LaTeX3 }
11956 \prop_new:N \g_msg_module_type_prop
11957 \prop_gput:Nnn \g_msg_module_type_prop { LaTeX } { }

```

(End definition for `\g_msg_module_name_prop` and `\g_msg_module_type_prop`. These variables are documented on page 153.)

`\msg_module_type:n`

Contextual footer information, with the potential to give modules an alternative name.

```

11958 \cs_new:Npn \msg_module_type:n #1
11959 {
11960 \prop_if_in:NnTF \g_msg_module_type_prop {#1}
11961 { \prop_item:Nn \g_msg_module_type_prop {#1} }
11962 { Package }
11963 }

```

(End definition for `\msg_module_type:n`. This function is documented on page 153.)

`\msg_module_name:n`  
`\msg_see_documentation_text:n`

Contextual footer information, with the potential to give modules an alternative name.

```

11964 \cs_new:Npn \msg_module_name:n #1
11965 {
11966 \prop_if_in:NnTF \g_msg_module_name_prop {#1}
11967 { \prop_item:Nn \g_msg_module_name_prop {#1} }
11968 {#1}
11969 }
11970 \cs_new:Npn \msg_see_documentation_text:n #1
11971 {

```

```

11972 See~the~ \msg_module_name:n {#1} ~
11973 documentation~for~further~information.
11974 }

```

(End definition for \msg\_module\_name:n and \msg\_see\_documentation\_text:n. These functions are documented on page 153.)

```

__msg_class_new:nn
11975 \group_begin:
11976 \cs_set_protected:Npn __msg_class_new:nn #1#2
11977 {
11978 \prop_new:c { l__msg_redirect_ #1 _prop }
11979 \cs_new_protected:cpn { __msg_ #1 _code:nnnnnn }
11980 ##1##2##3##4##5##6 {#2}
11981 \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
11982 {
11983 \use:x
11984 {
11985 \exp_not:n { __msg_use:nnnnnnn {#1} {##1} {##2} }
11986 { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
11987 { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
11988 }
11989 }
11990 \cs_new_protected:cp { msg_ #1 :nnnn } ##1##2##3##4##5
11991 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
11992 \cs_new_protected:cp { msg_ #1 :nnnn } ##1##2##3##4
11993 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
11994 \cs_new_protected:cp { msg_ #1 :nnn } ##1##2##3
11995 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
11996 \cs_new_protected:cp { msg_ #1 :nn } ##1##2
11997 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
11998 \cs_new_protected:cp { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
11999 {
12000 \use:x
12001 {
12002 \exp_not:N \exp_not:n
12003 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
12004 {##3} {##4} {##5} {##6}
12005 }
12006 }
12007 \cs_new_protected:cp { msg_ #1 :nnxxx } ##1##2##3##4##5
12008 { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
12009 \cs_new_protected:cp { msg_ #1 :nnxx } ##1##2##3##4
12010 { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
12011 \cs_new_protected:cp { msg_ #1 :nnx } ##1##2##3
12012 { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
12013 }

```

(End definition for \\_\_msg\_class\_new:nn.)

\msg\_fatal:nnnn  
\msg\_fatal:nnxxxx  
\msg\_fatal:nnnn  
For fatal errors, after the error message T<sub>E</sub>X bails out. We force a bail out rather than using \end as this means it does not matter if we are in a context where normally the run cannot end.

```

\msg_fatal:nnxx
\msg_fatal:nnnn
\msg_fatal:nnxx
\msg_fatal:nnn
\msg_fatal:nnx
\msg_fatal:nn
__msg_fatal_exit:
12014 __msg_class_new:nn { fatal }

```

```

12015 {
12016 __msg_interrupt:NnnnN
12017 \msg_fatal_text:n {#1} {#2}
12018 { {#3} {#4} {#5} {#6} }
12019 \c__msg_fatal_text_tl
12020 __msg_fatal_exit:
12021 }
12022 \cs_new_protected:Npn __msg_fatal_exit:
12023 {
12024 \tex_batchmode:D
12025 \tex_read:D -1 to \l__msg_internal_tl
12026 }

```

(End definition for `\msg_fatal:nnnnnn` and others. These functions are documented on page 154.)

### `\msg_critical:nnnnnn`

```

\msg_critical:nnxxxx
\msg_critical:nnnnn
\msg_critical:nnxxx
\msg_critical:nnnn
\msg_critical:nnxx
\msg_critical:nnn
\msg_critical:nnx
\msg_critical:nn

```

```

12027 __msg_class_new:nn { critical }
12028 {
12029 __msg_interrupt:NnnnN
12030 \msg_critical_text:n {#1} {#2}
12031 { {#3} {#4} {#5} {#6} }
12032 \c__msg_critical_text_tl
12033 \tex_endinput:D
12034 }

```

(End definition for `\msg_critical:nnnnnn` and others. These functions are documented on page 154.)

### `\msg_error:nnnnnn`

```

\msg_error:nnxxxx
\msg_error:nnnnn
\msg_error:nnxxx
\msg_error:nnnn
\msg_error:nnxx
\msg_error:nnn
\msg_error:nnx
\msg_error:nn

```

For an error, the interrupt routine is called. We check if there is a “more text” by comparing that control sequence with a permanently empty text.

```

12035 __msg_class_new:nn { error }
12036 {
12037 __msg_interrupt:NnnnN
12038 \msg_error_text:n {#1} {#2}
12039 { {#3} {#4} {#5} {#6} }
12040 \c_empty_tl
12041 }

```

(End definition for `\msg_error:nnnnnn` and others. These functions are documented on page 154.)

### `\msg_warning:nnnnnn`

```

\msg_warning:nnxxxx
\msg_warning:nnnnn
\msg_warning:nnxxx
\msg_warning:nnnn
\msg_warning:nnxx
\msg_warning:nnn
\msg_warning:nnx
\msg_warning:nn

```

Warnings are printed to the terminal.

```

12042 __msg_class_new:nn { warning }
12043 {
12044 \str_set:Nx \l__msg_text_str { \msg_warning_text:n {#1} }
12045 \str_set:Nx \l__msg_name_str { \msg_module_name:n {#1} }
12046 \iow_term:n { }
12047 \iow_wrap:nxnN
12048 {
12049 \l__msg_text_str : ~
12050 \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
12051 }
12052 {
12053 (\l__msg_name_str)
12054 \prg_replicate:nn
12055 {
12056 \str_count:N \l__msg_text_str

```

```

12057 - \str_count:N \l__msg_name_str
12058 }
12059 { ~ }
12060 }
12061 { } \iow_term:n
12062 \iow_term:n { }
12063 }

```

(End definition for `\msg_warning:nnnnnn` and others. These functions are documented on page 154.)

`\msg_info:nnnnnn`

```

\msg_info:nnxxxx
\msg_info:nnnn
\msg_info:nnxxx
\msg_info:nnnn
\msg_info:nnxx
\msg_info:nnn
\msg_info:nnx
\msg_info:nn
12064 __msg_class_new:nn { info }
12065 {
12066 \str_set:Nx \l__msg_text_str { \msg_info_text:n {#1} }
12067 \str_set:Nx \l__msg_name_str { \msg_module_name:n {#1} }
12068 \iow_log:n { }
12069 \iow_wrap:nxnN
12070 {
12071 \l__msg_text_str : ~
12072 \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
12073 }
12074 {
12075 (\l__msg_name_str)
12076 \prg_replicate:nn
12077 {
12078 \str_count:N \l__msg_text_str
12079 - \str_count:N \l__msg_name_str
12080 }
12081 { ~ }
12082 }
12083 { } \iow_log:n
12084 \iow_log:n { }
12085 }

```

(End definition for `\msg_info:nnnnnn` and others. These functions are documented on page 155.)

`\msg_log:nnnnnn`

“Log” data is very similar to information, but with no extras added.

```

\msg_log:nnxxxx
\msg_log:nnnn
\msg_log:nnxxx
\msg_log:nnnn
\msg_log:nnxx
\msg_log:nnn
\msg_log:nnx
\msg_log:nn
12086 __msg_class_new:nn { log }
12087 {
12088 \iow_wrap:nxnN
12089 { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
12090 { } { } \iow_log:n
12091 }

```

(End definition for `\msg_log:nnnnnn` and others. These functions are documented on page 155.)

`\msg_term:nnnnnn`

`\msg_term:nnxxxx`

`\msg_term:nnnn`

`\msg_term:nnxx`

`\msg_term:nnn`

`\msg_term:nnx`

`\msg_term:nn`

“Term” is used for communicating with the user through the terminal, like diagnostic messages, and debugging. This is similar to “log” messages, but uses the terminal output.

```

\msg_term:nnnn
\msg_term:nnnn
\msg_term:nnnn
\msg_term:nnnn
\msg_term:nnnn
\msg_term:nnnn
\msg_term:nnnn
\msg_term:nnnn
12092 __msg_class_new:nn { term }
12093 {
12094 \iow_wrap:nxnN
12095 { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
12096 { } { } \iow_term:n
12097 }

```

(End definition for `\msg_term:nnnnnn` and others. These functions are documented on page 155.)

**\msg\_none:nnnnnn** The **none** message type is needed so that input can be gobbled.

\msg\_none:nnxxxxx 12098 \\_\_msg\_class\_new:nn { none } { }

(End definition for `\msg_none:nnnnnn` and others. These functions are documented on page 155.)

The `show` message type is used for `\seq_show:N` and similar complicated data structures. Wrap the given text with a trailing dot (important later) then pass it to `\_msg_show:n`. If there is `\>~` (or if the whole thing starts with `>~`) we split there, print the first part and show the second part using `\showtokens` (the `\exp_after:wN` ensure a nice display). Note that this primitive adds a leading `>~` and trailing dot. That is why we included a trailing dot before wrapping and removed it afterwards. If there is no `\>~` do the same but with an empty second part which adds a spurious but inevitable `>~`.

```

\msg_show:nnx 12099 __msg_class_new:nn { show }
\msg_show:nn 12100 {
 \iow_wrap:nnnN
 { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
 { } { } __msg_show:n
}
__msg_show_dot:w 12103 __msg_show:nn
__msg_show:nn 12104 }
__msg_show:nn 12105 \cs_new_protected:Npn __msg_show:n #1
__msg_show:nn 12106 {
 \tl_if_in:nnTF { ^^J #1 } { ^^J > ~ }
}
__msg_show:nn 12107 {
 \tl_if_in:nnTF { #1 \s__msg_mark } { . \s__msg_mark }
 { __msg_show_dot:w } { __msg_show:w }
 ^^J #1 \s__msg_stop
}
__msg_show:nn 12108 {
 { __msg_show:nn { ? #1 } { } }
}
__msg_show:nn 12109 \cs_new:Npn __msg_show_dot:w #1 ^^J > ~ #2 . \s__msg_stop
__msg_show:nn 12110 { __msg_show:nn {#1} {#2} }
__msg_show:nn 12111 \cs_new:Npn __msg_show:w #1 ^^J > ~ #2 \s__msg_stop
__msg_show:nn 12112 { __msg_show:nn {#1} {#2} }
__msg_show:nn 12113 \cs_new_protected:Npn __msg_show:nn #1#2
__msg_show:nn 12114 {
 \tl_if_empty:nF {#1}
 { \exp_args:No \iow_term:n { \use_none:n #1 } }
__msg_internal_tl 12115 \tl_set:Nn \l__msg_internal_tl {#2}
__kernel_iow_with:Nnn 12116 \tex_newlinechar:D { 10 }
{
 __kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
}
__msg_internal_tl 12117 {
 \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
 { \exp_after:wN \l__msg_internal_tl }
}
__msg_internal_tl 12118 }
__msg_internal_tl 12119 }
__msg_internal_tl 12120 }
__msg_internal_tl 12121 }
__msg_internal_tl 12122 }
__msg_internal_tl 12123 }
__msg_internal_tl 12124 }
__msg_internal_tl 12125 }
__msg_internal_tl 12126 }
__msg_internal_tl 12127 }
__msg_internal_tl 12128 }
__msg_internal_tl 12129 }
__msg_internal_tl 12130 }
__msg_internal_tl 12131 }
__msg_internal_tl 12132 }

```

(End definition for `\msg_show:nnnnnn` and others. These functions are documented on page 268.)

End the group to eliminate \\_\\_msg\\_class\\_new:nn.

12133 \group\_end:

\\_\\_msg\\_class\\_chk\\_exist:nT Checking that a message class exists. We build this from \cs\_if\_free:cTF rather than \cs\_if\_exist:cTF because that avoids reading the second argument earlier than necessary.

```

12134 \cs_new:Npn __msg_class_chk_exist:nT #1
12135 {
12136 \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
12137 { __kernel_msg_error:nx { kernel } { message-class-unknown } {#1} }
12138 }
```

(End definition for \\_\\_msg\\_class\\_chk\\_exist:nT.)

\l\\_\\_msg\\_class\\_tl Support variables needed for the redirection system.

```

12139 \tl_new:N \l__msg_class_tl
12140 \tl_new:N \l__msg_current_class_tl
```

(End definition for \l\\_\\_msg\\_class\\_tl and \l\\_\\_msg\\_current\\_class\\_tl.)

\l\\_\\_msg\\_redirect\\_prop For redirection of individually-named messages

```

12141 \prop_new:N \l__msg_redirect_prop
```

(End definition for \l\\_\\_msg\\_redirect\\_prop.)

\l\\_\\_msg\\_hierarchy\\_seq During redirection, split the message name into a sequence: {/module/submodule}, {/module}, and {}.

```

12142 \seq_new:N \l__msg_hierarchy_seq
```

(End definition for \l\\_\\_msg\\_hierarchy\\_seq.)

\l\\_\\_msg\\_class\\_loop\\_seq Classes encountered when following redirections to check for loops.

```

12143 \seq_new:N \l__msg_class_loop_seq
```

(End definition for \l\\_\\_msg\\_class\\_loop\\_seq.)

\\_\\_msg\\_use:nnnnnn  
\\_\\_msg\\_use\\_redirect\\_name:n  
\\_\\_msg\\_use\\_hierarchy:nwwN  
\\_\\_msg\\_use\\_redirect\\_module:n  
\\_\\_msg\\_use\\_code:  
\\_\\_msg\\_use\\_code: Actually using a message is a multi-step process. First, some safety checks on the message and class requested. The code and arguments are then stored to avoid passing them around. The assignment to \\_\\_msg\\_use\\_code: is similar to \tl\_set:Nn. The message is eventually produced with whatever \l\\_\\_msg\\_class\\_tl is when \\_\\_msg\\_use\\_code: is called. Here is also a good place to suppress tracing output if the trace package is loaded since all (non-expandable) messages go through this auxiliary.

```

12144 \cs_new_protected:Npn __msg_use:nnnnnnn #1#2#3#4#5#6#7
12145 {
12146 \cs_if_exist_use:N \conditionally@traceoff
12147 \msg_if_exist:nnTF {#2} {#3}
12148 {
12149 __msg_class_chk_exist:nT {#1}
12150 {
12151 \tl_set:Nn \l__msg_current_class_tl {#1}
12152 \cs_set_protected:Npx __msg_use_code:
12153 {
12154 \exp_not:n
12155 {
12156 \use:c { __msg_ \l__msg_class_tl _code:nnnnnn }
12157 {#2} {#3} {#4} {#5} {#6} {#7}
12158 }
12159 }
```

```

12159 }
12160 __msg_use_redirect_name:n { #2 / #3 }
12161 }
12162 }
12163 { __kernel_msg_error:nnxx { kernel } { message-unknown } {#2} {#3} }
12164 \cs_if_exist_use:N \conditionally@traceon
12165 }
12166 \cs_new_protected:Npn __msg_use_code: { }

```

The first check is for a individual message redirection. If this applies then no further redirection is attempted. Otherwise, split the message name into *<module>*, *<submodule>* and *<message>* (with an arbitrary number of slashes), and store *{/module/submodule}*, *{/module}* and *{}* into `\l__msg_hierarchy_seq`. We then map through this sequence, applying the most specific redirection.

```

12167 \cs_new_protected:Npn __msg_use_redirect_name:n #1
12168 {
12169 \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
12170 { __msg_use_code: }
12171 {
12172 \seq_clear:N \l__msg_hierarchy_seq
12173 __msg_use_hierarchy:nwwN { }
12174 #1 \s__msg_mark __msg_use_hierarchy:nwwN
12175 / \s__msg_mark __msg_use_none_delimit_by_s_stop:w
12176 \s__msg_stop
12177 __msg_use_redirect_module:n { }
12178 }
12179 }
12180 \cs_new_protected:Npn __msg_use_hierarchy:nwwN #1#2 / #3 \s__msg_mark #4
12181 {
12182 \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
12183 #4 { #1 / #2 } #3 \s__msg_mark #4
12184 }

```

At this point, the items of `\l__msg_hierarchy_seq` are the various levels at which we should look for a redirection. Redirections which are less specific than the argument of `\__msg_use_redirect_module:n` are not attempted. This argument is empty for a class redirection, `/module` for a module redirection, *etc.* Loop through the sequence to find the most specific redirection, with module `##1`. The loop is interrupted after testing for a redirection for `##1` equal to the argument `#1` (least specific redirection allowed). When a redirection is found, break the mapping, then if the redirection targets the same class, output the code with that class, and otherwise set the target as the new current class, and search for further redirections. Those redirections should be at least as specific as `##1`.

```

12185 \cs_new_protected:Npn __msg_use_redirect_module:n #1
12186 {
12187 \seq_map_inline:Nn \l__msg_hierarchy_seq
12188 {
12189 \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
12190 {##1} \l__msg_class_tl
12191 {
12192 \seq_map_break:n
12193 {
12194 \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
12195 { __msg_use_code: }

```

```

12196 {
12197 \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
12198 __msg_use_redirect_module:n {##1}
12199 }
12200 }
12201 }
12202 {
12203 \str_if_eq:nnT {##1} {#1}
12204 {
12205 \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
12206 \seq_map_break:n { __msg_use_code: }
12207 }
12208 }
12209 }
12210 }
```

(End definition for `\__msg_use:nnnnnnn` and others.)

### `\msg_redirect_name:nnn`

Named message always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```

12211 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
12212 {
12213 \tl_if_empty:nTF {#3}
12214 { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
12215 {
12216 __msg_class_chk_exist:nT {#3}
12217 { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
12218 }
12219 }
```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 157.)

### `\msg_redirect_class:nn`

### `\msg_redirect_module:nnn`

### `\__msg_redirect:nnn`

### `\__msg_redirect_loop_chk:nnn`

### `\__msg_redirect_loop_list:n`

If the target class is empty, eliminate the corresponding redirection. Otherwise, add the redirection. We must then check for a loop: as an initialization, we start by storing the initial class in `\l__msg_current_class_tl`.

```

12220 \cs_new_protected:Npn \msg_redirect_class:nn
12221 { __msg_redirect:nnn { } }
12222 \cs_new_protected:Npn \msg_redirect_module:nnn #1
12223 { __msg_redirect:nnn { / #1 } }
12224 \cs_new_protected:Npn __msg_redirect:nnn #1#2#3
12225 {
12226 __msg_class_chk_exist:nT {#2}
12227 {
12228 \tl_if_empty:nTF {#3}
12229 { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
12230 {
12231 __msg_class_chk_exist:nT {#3}
12232 {
12233 \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
12234 \tl_set:Nn \l__msg_current_class_tl {#2}
12235 \seq_clear:N \l__msg_class_loop_seq
12236 __msg_redirect_loop_chk:nnn {#2} {#3} {#1}
12237 }
12238 }
```

```

12239 }
1240 }
```

Since multiple redirections can only happen with increasing specificity, a loop requires that all steps are of the same specificity. The new redirection can thus only create a loop with other redirections for the exact same module, #1, and not submodules. After some initialization above, follow redirections with \l\_\_msg\_class\_t1, and keep track in \l\_\_msg\_class\_loop\_seq of the various classes encountered. A redirection from a class to itself, or the absence of redirection both mean that there is no loop. A redirection to the initial class marks a loop. To break it, we must decide which redirection to cancel. The user most likely wants the newly added redirection to hold with no further redirection. We thus remove the redirection starting from #2, target of the new redirection. Note that no message is emitted by any of the underlying functions: otherwise we may get an infinite loop because of a message from the message system itself.

```

12241 \cs_new_protected:Npn __msg_redirect_loop_chk:n {#1}{#2}{#3}
12242 {
12243 \seq_put_right:Nn \l__msg_class_loop_seq {#1}
12244 \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_t1
12245 {
12246 \str_if_eq:VnF \l__msg_class_t1 {#1}
12247 {
12248 \tl_if_eq:NNTF \l__msg_class_t1 \l__msg_current_class_t1
12249 {
12250 \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
12251 __kernel_msg_warning:nxxxxx
12252 { kernel } { message-redirect-loop }
12253 { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
12254 { \seq_item:Nn \l__msg_class_loop_seq { 2 } }
12255 {#3}
12256 {
12257 \seq_map_function:NN \l__msg_class_loop_seq
12258 __msg_redirect_loop_list:n
12259 { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
12260 }
12261 }
12262 { __msg_redirect_loop_chk:onn \l__msg_class_t1 {#2} {#3} }
12263 }
12264 }
12265 }
12266 \cs_generate_variant:Nn __msg_redirect_loop_chk:nnn { o }
12267 \cs_new:Npn __msg_redirect_loop_list:n {#1} { {#1} ~ => ~ }
```

(End definition for \msg\_redirect\_class:nn and others. These functions are documented on page 157.)

## 19.6 Kernel-specific functions

The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.

```

12268 \cs_new_protected:Npn __kernel_msg_new:nnnn #1#2
12269 { \msg_new:nnnn { LaTeX } { #1 / #2 } }
12270 \cs_new_protected:Npn __kernel_msg_new:nnn #1#2
12271 { \msg_new:nnn { LaTeX } { #1 / #2 } }
```

```

12272 \cs_new_protected:Npn __kernel_msg_set:nnnn #1#2
12273 { \msg_set:nnnn { LaTeX } { #1 / #2 } }
12274 \cs_new_protected:Npn __kernel_msg_set:nnn #1#2
12275 { \msg_set:nnn { LaTeX } { #1 / #2 } }

```

(End definition for `\__kernel_msg_new:nnnn` and others.)

`\__msg_kernel_class_new:nN`  
`\__msg_kernel_class_new_aux:nN`

All the functions for kernel messages come in variants ranging from 0 to 4 arguments. Those with less than 4 arguments are defined in terms of the 4-argument variant, in a way very similar to `\__msg_class_new:nn`. This auxiliary is destroyed at the end of the group.

```

12276 \group_begin:
12277 \cs_set_protected:Npn __msg_kernel_class_new:nN #1
12278 { __msg_kernel_class_new_aux:nN { __kernel_msg_ #1 } }
12279 \cs_set_protected:Npn __msg_kernel_class_new:nN #1#2
12280 {
12281 \cs_new_protected:cpx { #1 :nnnnnn } ##1##2##3##4##5##6
12282 {
12283 \use:x
12284 {
12285 \exp_not:n { #2 { LaTeX } { ##1 / ##2 } }
12286 { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
12287 { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
12288 }
12289 }
12290 \cs_new_protected:cpx { #1 :nnnnnn } ##1##2##3##4##5
12291 { \exp_not:c { #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
12292 \cs_new_protected:cpx { #1 :nnnn } ##1##2##3##4
12293 { \exp_not:c { #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
12294 \cs_new_protected:cpx { #1 :nnn } ##1##2##3
12295 { \exp_not:c { #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
12296 \cs_new_protected:cpx { #1 :nn } ##1##2
12297 { \exp_not:c { #1 :nnnnnn } {##1} {##2} { } { } { } { } }
12298 \cs_new_protected:cpx { #1 :nnxxxx } ##1##2##3##4##5##6
12299 {
12300 \use:x
12301 {
12302 \exp_not:N \exp_not:n
12303 { \exp_not:c { #1 :nnnnnn } {##1} {##2} }
12304 {##3} {##4} {##5} {##6}
12305 }
12306 }
12307 \cs_new_protected:cpx { #1 :nnxxxx } ##1##2##3##4##5
12308 { \exp_not:c { #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
12309 \cs_new_protected:cpx { #1 :nnxx } ##1##2##3##4
12310 { \exp_not:c { #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
12311 \cs_new_protected:cpx { #1 :nx } ##1##2##3
12312 { \exp_not:c { #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
12313 }

```

(End definition for `\__msg_kernel_class_new:nN` and `\__msg_kernel_class_new_aux:nN`.)

`\__kernel_msg_fatal:nnnnnn`  
`\__kernel_msg_fatal:nnxxxx`  
`\__kernel_msg_fatal:nnnn`  
`\__kernel_msg_fatal:nnx`  
`\__kernel_msg_fatal:nn`  
`\__kernel_msg_critical:nnnnnn`  
`\__kernel_msg_critical:nnxxxx`

Neither fatal kernel errors nor kernel errors can be redirected. We directly use the code for (non-kernel) fatal errors and errors, adding the “`LATEX`” module name. Three functions are already defined by `l3basics`; we need to undefine them to avoid errors.

```

12314 __msg_kernel_class_new:nN { fatal } __msg_fatal_code:nnnnnn
12315 __msg_kernel_class_new:nN { critical } __msg_critical_code:nnnnnn
12316 \cs_undefine:N __kernel_msg_error:nnxx
12317 \cs_undefine:N __kernel_msg_error:nnx
12318 \cs_undefine:N __kernel_msg_error:nn
12319 __msg_kernel_class_new:nN { error } __msg_error_code:nnnnnn

```

(End definition for `\_\_kernel_msg_fatal:nnnnnn` and others.)

Kernel messages which can be redirected simply use the machinery for normal messages, with the module name “`LATeX`”.

```

12320 __msg_kernel_class_new:nN { warning } \msg_warning:nnxxxx
12321 __msg_kernel_class_new:nN { info } \msg_info:nnxxxx

```

(End definition for `\_\_kernel_msg_warning:nnnnnn` and others.)

End the group to eliminate `\_\_msg_kernel_class_new:nN`.

```
12322 \group_end:
```

Error messages needed to actually implement the message system itself.

```

12323 __kernel_msg_new:nnn { kernel } { message-already-defined }
12324 { Message~'#2'~for~module~'#1'~already~defined. }
12325 {
12326 \c_msg_coding_error_text_tl
12327 LaTeX~was~asked~to~define~a~new~message~called~'#2'\\
12328 by~the~module~'#1':~this~message~already~exists.
12329 \c_msg_return_text_tl
12330 }
12331 __kernel_msg_new:nnn { kernel } { message-unknown }
12332 { Unknown~message~'#2'~for~module~'#1'. }
12333 {
12334 \c_msg_coding_error_text_tl
12335 LaTeX~was~asked~to~display~a~message~called~'#2'\\
12336 by~the~module~'#1':~this~message~does~not~exist.
12337 \c_msg_return_text_tl
12338 }
12339 __kernel_msg_new:nnnn { kernel } { message-class-unknown }
12340 { Unknown~message~class~'#1'. }
12341 {
12342 LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\\
12343 this~was~never~defined.
12344 \c_msg_return_text_tl
12345 }
12346 __kernel_msg_new:nnnn { kernel } { message-redirect-loop }
12347 {
12348 Message~redirection~loop~caused~by~ {#1} ~=>~ {#2}
12349 \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
12350 }
12351 {
12352 Adding~the~message~redirection~ {#1} ~=>~ {#2}
12353 \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
12354 created~an~infinite~loop\\\\
12355 \iow_indent:n { #4 \\\\ }
12356 }

```

Messages for earlier kernel modules plus a few for l3keys which cover coding errors.

```
12357 __kernel_msg_new:nnn { kernel } { bad-number-of-arguments }
12358 { Function~'#1'~cannot~be~defined~with~#2~arguments. }
12359 {
12360 \c__msg_coding_error_text_tl
12361 LaTeX~has~been~asked~to~define~a~function~'#1'~with~
12362 #2~arguments.~
12363 TeX~allows~between~0~and~9~arguments~for~a~single~function.
12364 }
12365 __kernel_msg_new:nnn { kernel } { char-active }
12366 { Cannot~generate~active~chars. }
12367 __kernel_msg_new:nnn { kernel } { char-invalid-catcode }
12368 { Invalid~catcode~for~char~generation. }
12369 __kernel_msg_new:nnn { kernel } { char-null-space }
12370 { Cannot~generate~null~char~as~a~space. }
12371 __kernel_msg_new:nnn { kernel } { char-out-of-range }
12372 { Charcode~requested~out~of~engine~range. }
12373 __kernel_msg_new:nnn { kernel } { char-space }
12374 { Cannot~generate~space~chars. }
12375 __kernel_msg_new:nnnn { kernel } { command-already-defined }
12376 { Control~sequence~#1~already~defined. }
12377 {
12378 \c__msg_coding_error_text_tl
12379 LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
12380 but~this~name~has~already~been~used~elsewhere. \\ \\
12381 The~current~meaning~is:\\
12382 \ #2
12383 }
12384 __kernel_msg_new:nnnn { kernel } { command-not-defined }
12385 { Control~sequence~#1~undefined. }
12386 {
12387 \c__msg_coding_error_text_tl
12388 LaTeX~has~been~asked~to~use~a~control~sequence~'#1':\\\
12389 this~has~not~been~defined~yet.
12390 }
12391 __kernel_msg_new:nnnn { kernel } { empty-search-pattern }
12392 { Empty~search~pattern. }
12393 {
12394 \c__msg_coding_error_text_tl
12395 LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'#1':~that~
12396 would~lead~to~an~infinite~loop!
12397 }
12398 __kernel_msg_new:nnnn { kernel } { out-of-registers }
12399 { No~room~for~a~new~#1. }
12400 {
12401 TeX~only~supports~\int_use:N \c_max_register_int \ %
12402 of~each~type.~All~the~#1~registers~have~been~used.~
12403 This~run~will~be~aborted~now.
12404 }
12405 __kernel_msg_new:nnnn { kernel } { non-base-function }
12406 { Function~'#1'~is~not~a~base~function }
12407 {
12408 \c__msg_coding_error_text_tl
12409 Functions~defined~through~\iow_char:N\cs_new:Nn~must~have~
```

```

12410 a-signature-consisting-of-only-normal-arguments~'N'~and~'n'.~
12411 To-define-variants-use~\iow_char:N\cs_generate_variant:Nn~
12412 and-to-define-other-functions-use~\iow_char:N\cs_new:Npn.
12413 }
12414 _kernel_msg_new:nnn { kernel } { missing-colon }
12415 { Function~'#1'~contains-no':'. }
12416 {
12417 \c_msg_error_text_tl
12418 Code-level-functions-must-contain':~to~separate~the~
12419 argument-specification~from~the~function~name.~This~is~
12420 needed~when~defining~conditionals~or~variants,~or~when~building~a~
12421 parameter~text~from~the~number~of~arguments~of~the~function.
12422 }
12423 _kernel_msg_new:nnn { kernel } { overflow }
12424 { Integers~larger~than~2^{30}-1~cannot~be~stored~in~arrays. }
12425 {
12426 An~attempt~was~made~to~store~#3~
12427 \tl_if_empty:nF {#2} { at~position~#2~ } in~the~array~'#1'.~
12428 The~largest~allowed~value~#4~will~be~used~instead.
12429 }
12430 _kernel_msg_new:nnn { kernel } { out-of-bounds }
12431 { Access~to~an~entry~beyond~an~array's~bounds. }
12432 {
12433 An~attempt~was~made~to~access~or~store~data~at~position~#2~of~the~
12434 array~'#1',~but~this~array~has~entries~at~positions~from~1~to~#3.
12435 }
12436 _kernel_msg_new:nnn { kernel } { protected-predicate }
12437 { Predicate~'#1'~must~be~expandable. }
12438 {
12439 \c_msg_error_text_tl
12440 LaTeX~has~been~asked~to~define~'#1'~as~a~protected~predicate.~
12441 Only~expandable~tests~can~have~a~predicate~version.
12442 }
12443 _kernel_msg_new:n { kernel } { randint-backward-range }
12444 { Bounds~ordered~backwards~in~\iow_char:N\int_rand:nn~{#1}~{#2}. }
12445 _kernel_msg_new:nnn { kernel } { conditional-form-unknown }
12446 { Conditional~form~'#1'~for~function~'#2'~unknown. }
12447 {
12448 \c_msg_error_text_tl
12449 LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
12450 the~function~'#2',~but~only~'TF',~'T',~'F',~and~'P'~forms~exist.
12451 }
12452 _kernel_msg_new:nnn { kernel } { key-no-property }
12453 { No~property~given~in~definition~of~key~'#1'. }
12454 {
12455 \c_msg_error_text_tl
12456 Inside~\keys_define:nn~each~key~name~
12457 needs~a~property:~\\~\\
12458 \iow_indent:n { #1 .<property> } \\~\\
12459 LaTeX~did~not~find~a~'.'~to~indicate~the~start~of~a~property.
12460 }
12461 _kernel_msg_new:nnn { kernel } { key-property-boolean-values-only }
12462 { The~property~'#1'~accepts~boolean~values~only. }
12463 {

```

```

12464 \c__msg_coding_error_text_tl
12465 The~property~'#1'~only~accepts~the~values~'true'~and~'false'.
12466 }
12467 __kernel_msg_new:nnnn { kernel } { key-property-requires-value }
12468 { The~property~'#1'~requires~a~value. }
12469 {
12470 \c__msg_coding_error_text_tl
12471 LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'. \\
12472 No~value~was~given~for~the~property,~and~one~is~required.
12473 }
12474 __kernel_msg_new:nnnn { kernel } { key-property-unknown }
12475 { The~key~property~'#1'~is~unknown. }
12476 {
12477 \c__msg_coding_error_text_tl
12478 LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
12479 this~property~is~not~defined.
12480 }
12481 __kernel_msg_new:nnnn { kernel } { quote-in-shell }
12482 { Quotes~in~shell~command~'#1'. }
12483 { Shell~commands~cannot~contain~quotes~()." . }
12484 __kernel_msg_new:nnnn { kernel } { invalid-quark-function }
12485 { Quark~test~function~'#1'~is~invalid. }
12486 {
12487 \c__msg_coding_error_text_tl
12488 LaTeX~has~been~asked~to~create~quark~test~function~'#1'~
12489 \tl_if_empty:nTF {#2}
12490 { but~that~name~ }
12491 { with~signature~'#2',~but~that~signature~ }
12492 is~not~valid.
12493 }
12494 __kernel_msg_new:nnn { kernel } { invalid-quark }
12495 { Invalid~quark~variable~'#1'. }
12496 __kernel_msg_new:nnnn { kernel } { scanmark-already-defined }
12497 { Scan~mark~#1~already~defined. }
12498 {
12499 \c__msg_coding_error_text_tl
12500 LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
12501 but~this~name~has~already~been~used~for~a~scan~mark.
12502 }
12503 __kernel_msg_new:nnnn { kernel } { shuffle-too-large }
12504 { The~sequence~#1~is~too~long~to~be~shuffled~by~TeX. }
12505 {
12506 TeX~has~ \int_eval:n { \c_max_register_int + 1 } ~
12507 toks~registers:~this~only~allows~to~shuffle~up~to~
12508 \int_use:N \c_max_register_int \ items.~
12509 The~list~will~not~be~shuffled.
12510 }
12511 __kernel_msg_new:nnnn { kernel } { variable-not-defined }
12512 { Variable~#1~undefined. }
12513 {
12514 \c__msg_coding_error_text_tl
12515 LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
12516 been~defined~yet.
12517 }

```

```

12518 __kernel_msg_new:nnnn { kernel } { variant-too-long }
12519 { Variant~form~'#1'~longer~than~base~signature~of~'#2'. }
12520 {
12521 \c__msg_coding_error_text_tl
12522 LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
12523 with~a~signature~starting~with~'#1',~but~that~is~longer~than~
12524 the~signature~(part~after~the~colon)~of~'#2'.
12525 }
12526 __kernel_msg_new:nnnn { kernel } { invalid-variant }
12527 { Variant~form~'#1'~invalid~for~base~form~'#2'. }
12528 {
12529 \c__msg_coding_error_text_tl
12530 LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
12531 with~a~signature~starting~with~'#1',~but~cannot~change~an~argument~
12532 from~type~'#3'~to~type~'#4'.
12533 }
12534 __kernel_msg_new:nnnn { kernel } { invalid-exp-args }
12535 { Invalid~variant~specifier~'#1'~in~'#2'. }
12536 {
12537 \c__msg_coding_error_text_tl
12538 LaTeX~has~been~asked~to~create~an~\iow_char:N\\exp_args:N...~
12539 function~with~signature~'N#2'~but~'#1'~is~not~a~valid~argument~
12540 specifier.
12541 }
12542 __kernel_msg_new:nnn { kernel } { deprecated-variant }
12543 {
12544 Variant~form~'#1'~deprecated~for~base~form~'#2'~.
12545 One~should~not~change~an~argument~from~type~'#3'~to~type~'#4',
12546 \str_case:nnF {#3}
12547 {
12548 { n } { :~use~a~'\token_if_eq_charcode:NNTF #4 c v V'~variant? }
12549 { N } { :~base~form~only~accepts~a~single~token~argument. }
12550 {#4} { :~base~form~is~already~a~variant. }
12551 } { . }
12552 }

```

Some errors are only needed in package mode if debugging is enabled by one of the options `enable-debug`, `check-declarations`, `log-functions`, or on the contrary if debugging is turned off. In format mode the error is somewhat different.

```

12553 __kernel_msg_new:nnnn { kernel } { enable-debug }
12554 { To~use~'#1'~set~the~'enable-debug'~option. }
12555 {
12556 The~function~'#1'~will~be~ignored~because~it~can~only~work~if~
12557 some~internal~functions~in~expl3~have~been~appropriately~
12558 defined.~This~only~happens~if~one~of~the~options~
12559 'enable-debug',~'check-declarations'~or~'log-functions'~was~
12560 given~as~an~option:~see~the~main~expl3~documentation.
12561 }

```

Some errors only appear in expandable settings, hence don't need a "more-text" argument.

```

12562 __kernel_msg_new:nnn { kernel } { bad-exp-end-f }
12563 { Misused~\exp_end_continue_f:w~or~:nw }
12564 __kernel_msg_new:nnn { kernel } { bad-variable }
12565 { Erroneous~variable~#1~used! }

```

```

12566 __kernel_msg_new:nnn { kernel } { misused-sequence }
12567 { A-sequence~was~misused. }
12568 __kernel_msg_new:nnn { kernel } { misused-prop }
12569 { A-property-list~was~misused. }
12570 __kernel_msg_new:nnn { kernel } { negative-replication }
12571 { Negative~argument~for~\iow_char:N\prg_replicate:nn. }
12572 __kernel_msg_new:nnn { kernel } { prop-keyval }
12573 { Missing/extraneous~in~#1~(in~..._keyval:Nn') }
12574 __kernel_msg_new:nnn { kernel } { unknown-comparison }
12575 { Relation~#1~unknown~use~=<,>,~=~,!=~,~<=,~>=. }
12576 __kernel_msg_new:nnn { kernel } { zero-step }
12577 { Zero~step~size~for~step~function~#1. }
12578 \cs_if_exist:NF \tex_expanded:D
12579 {
12580 __kernel_msg_new:nnn { kernel } { e-type }
12581 { #1 ~ in-e-type~argument }
12582 }

```

Messages used by the “show” functions.

```

12583 __kernel_msg_new:nnn { kernel } { show-clist }
12584 {
12585 The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
12586 \tl_if_empty:nTF {#2}
12587 { is~empty \\>~ . }
12588 { contains~the~items~(without~outer~braces): #2 . }
12589 }
12590 __kernel_msg_new:nnn { kernel } { show-intarray }
12591 { The~integer~array~#1~contains~#2~items: \\#3 . }
12592 __kernel_msg_new:nnn { kernel } { show-prop }
12593 {
12594 The~property~list~#1~
12595 \tl_if_empty:nTF {#2}
12596 { is~empty \\>~ . }
12597 { contains~the~pairs~(without~outer~braces): #2 . }
12598 }
12599 __kernel_msg_new:nnn { kernel } { show-seq }
12600 {
12601 The~sequence~#1~
12602 \tl_if_empty:nTF {#2}
12603 { is~empty \\>~ . }
12604 { contains~the~items~(without~outer~braces): #2 . }
12605 }
12606 __kernel_msg_new:nnn { kernel } { show-streams }
12607 {
12608 \tl_if_empty:nTF {#2} { No~ } { The~following~ }
12609 \str_case:nn {#1}
12610 {
12611 { ior } { input ~ }
12612 { iow } { output ~ }
12613 }
12614 streams~are~
12615 \tl_if_empty:nTF {#2} { open } { in~use: #2 . }
12616 }

```

System layer messages

```

12617 __kernel_msg_new:nnnn { sys } { backend-set }
12618 { Backend~configuration~already~set. }
12619 {
12620 Run-time~backend~selection~may~only~be~carried~out~once~during~a~run.~
12621 This~second~attempt~to~set~them~will~be~ignored.
12622 }
12623 __kernel_msg_new:nnnn { sys } { wrong-backend }
12624 { Backend~request~inconsistent~with~engine:~using~'#2'~backend. }
12625 {
12626 You~have~requested~backend~'#1',~but~this~is~not~suitable~for~use~with~the~
12627 active~engine.~LaTeX3~will~use~the~'#2'~backend~instead.
12628 }

```

## 19.7 Expandable errors

`\__msg_expandable_error:n`  
`\__msg_expandable_error:w`

In expansion only context, we cannot use the normal means of reporting errors. Instead, we feed T<sub>E</sub>X an undefined control sequence, `\LaTeX3 error:`. It is thus interrupted, and shows the context, which thanks to the odd-looking `\use:n` is

```

<argument> \LaTeX3 error:
The error message.

```

In other words, T<sub>E</sub>X is processing the argument of `\use:n`, which is `\LaTeX3 error: <error message>`. Then `\__msg_expandable_error:w` cleans up. In fact, there is an extra subtlety: if the user inserts tokens for error recovery, they should be kept. Thus we also use an odd space character (with category code 7) and keep tokens until that space character, dropping everything else until `\s__msg_stop`. The `\exp_end:` prevents losing braces around the user-inserted text if any, and stops the expansion of `\exp:w`. The group is used to prevent `\LaTeX3-error:` from being globally equal to `\scan_stop:`.

```

12629 \group_begin:
12630 \cs_set_protected:Npn __msg_tmp:w #1#2
12631 {
12632 \cs_new:Npn __msg_expandable_error:n ##1
12633 {
12634 \exp:w
12635 \exp_after:wN \exp_after:wN
12636 \exp_after:wN __msg_expandable_error:w
12637 \exp_after:wN \exp_after:wN
12638 \exp_after:wN \exp_end:
12639 \use:n { #1 #2 ##1 } #2
12640 }
12641 \cs_new:Npn __msg_expandable_error:w ##1 #2 ##2 #2 {##1}
12642 }
12643 \exp_args:Ncx __msg_tmp:w { LaTeX3-error: }
12644 { \char_generate:nn { '\ } { 7 } }
12645 \group_end:

```

(End definition for `\__msg_expandable_error:n` and `\__msg_expandable_error:w`.)

The command built from the csname `\c__msg_text_prefix_tl` LaTeX / #1 / #2 takes four arguments and builds the error text, which is fed to `\__msg_expandable_error:n` with appropriate expansion: just as for usual messages the arguments are first turned to strings, then the message is fully expanded.

```

__kernel_msg_expandable_error:nnnnn
__kernel_msg_expandable_error:nnffff
__kernel_msg_expandable_error:nnnnnn
__kernel_msg_expandable_error:nnnnf
__kernel_msg_expandable_error:nnnn
__kernel_msg_expandable_error:nnnf
__kernel_msg_expandable_error:nnf
__kernel_msg_expandable_error:nn

```

```

12646 \exp_args_generate:n { oooo }
12647 \cs_new:Npn __kernel_msg_expandable_error:nnnnnn #1#2#3#4#5#6
12648 {
12649 \exp_args:Ne __msg_expandable_error:n
12650 {
12651 \exp_args:Nc \exp_args:Noooo
12652 { \c__msg_text_prefix_tl LaTeX / #1 / #2 }
12653 { \tl_to_str:n {#3} }
12654 { \tl_to_str:n {#4} }
12655 { \tl_to_str:n {#5} }
12656 { \tl_to_str:n {#6} }
12657 }
12658 }
12659 \cs_new:Npn __kernel_msg_expandable_error:nnnnn #1#2#3#4#5
12660 {
12661 __kernel_msg_expandable_error:nnnnn
12662 {#1} {#2} {#3} {#4} {#5} { }
12663 }
12664 \cs_new:Npn __kernel_msg_expandable_error:nnnn #1#2#3#4
12665 {
12666 __kernel_msg_expandable_error:nnnnn
12667 {#1} {#2} {#3} {#4} { } { }
12668 }
12669 \cs_new:Npn __kernel_msg_expandable_error:nnn #1#2#3
12670 {
12671 __kernel_msg_expandable_error:nnnnn
12672 {#1} {#2} {#3} { } { } { }
12673 }
12674 \cs_new:Npn __kernel_msg_expandable_error:nn #1#2
12675 {
12676 __kernel_msg_expandable_error:nnnnn
12677 {#1} {#2} { } { } { } { }
12678 }
12679 \cs_generate_variant:Nn __kernel_msg_expandable_error:nnnnnn { nnffff }
12680 \cs_generate_variant:Nn __kernel_msg_expandable_error:nnnnn { nnfff }
12681 \cs_generate_variant:Nn __kernel_msg_expandable_error:nnnn { nnff }
12682 \cs_generate_variant:Nn __kernel_msg_expandable_error:nnn { nnf }

```

(End definition for `\__kernel_msg_expandable_error:nnnnnn` and others.)

```

\msg_expandable_error:nnnnnn
\msg_expandable_error:nnffff
\msg_expandable_error:nnnnn
\msg_expandable_error:nnfff
\msg_expandable_error:nnnn
\msg_expandable_error:nnff
\msg_expandable_error:nnn
\msg_expandable_error:nnf
\msg_expandable_error:nn
\msg_expandable_error:nnn
\msg_expandable_error_module:nn

```

Pass to an auxiliary the message to display and the module name

```

12683 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
12684 {
12685 \exp_args:Ne __msg_expandable_error_module:nn
12686 {
12687 \exp_args:Nc \exp_args:Noooo
12688 { \c__msg_text_prefix_tl #1 / #2 }
12689 { \tl_to_str:n {#3} }
12690 { \tl_to_str:n {#4} }
12691 { \tl_to_str:n {#5} }
12692 { \tl_to_str:n {#6} }
12693 }
12694 {#1}
12695 }

```

```

12696 \cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
12697 { \msg_expandable_error:nnnnn {#1} {#2} {#3} {#4} {#5} { } }
12698 \cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
12699 { \msg_expandable_error:nnnnn {#1} {#2} {#3} {#4} { } { } }
12700 \cs_new:Npn \msg_expandable_error:nnn #1#2#3
12701 { \msg_expandable_error:nnnnn {#1} {#2} {#3} {#4} { } { } { } }
12702 \cs_new:Npn \msg_expandable_error:nn #1#2
12703 { \msg_expandable_error:nnnnn {#1} {#2} { } { } { } { } { } }
12704 \cs_generate_variant:Nn \msg_expandable_error:nnnnn { nnffff }
12705 \cs_generate_variant:Nn \msg_expandable_error:nnnn { nnfff }
12706 \cs_generate_variant:Nn \msg_expandable_error:nnnn { nnff }
12707 \cs_generate_variant:Nn \msg_expandable_error:nnn { nnf }
12708 \cs_new:Npn __msg_expandable_error_module:nn #1#2
12709 {
12710 \exp_after:wN \exp_after:wN
12711 \exp_after:wN __msg_use_none_delimit_by_s_stop:w
12712 \use:n { \::error ! ~ #2 : ~ #1 } \s__msg_stop
12713 }

```

(End definition for `\msg_expandable_error:nnnnn` and others. These functions are documented on page [156](#).)

12714 `</package>`

## 20 I3file implementation

The following test files are used for this code: `m3file001`.

12715 `<*package>`

### 20.1 Input operations

12716 `<@@=ior>`

#### 20.1.1 Variables and constants

`\l__ior_internal_tl` Used as a short-term scratch variable.

12717 `\tl_new:N \l__ior_internal_tl`

(End definition for `\l__ior_internal_tl`.)

`\c__ior_term_ior` Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.

12718 `\int_const:Nn \c__ior_term_ior { 16 }`

(End definition for `\c__ior_term_ior`.)

`\g__ior_streams_seq` A list of the currently-available input streams to be used as a stack.

12719 `\seq_new:N \g__ior_streams_seq`

(End definition for `\g__ior_streams_seq`.)

`\l__ior_stream_tl` Used to recover the raw stream number from the stack.

12720 `\tl_new:N \l__ior_stream_tl`

(End definition for `\l__ior_stream_tl`.)

\g\_\_ior\_streams\_prop The name of the file attached to each stream is tracked in a property list. To get the correct number of reserved streams in package mode the underlying mechanism needs to be queried. For L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  and plain T<sub>E</sub>X this data is stored in \count16: with the etex package loaded we need to subtract 1 as the register holds the number of the next stream to use. In ConT<sub>E</sub>Xt, we need to look at \count38 but there is no subtraction: like the original plain T<sub>E</sub>X/L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  mechanism it holds the value of the *last* stream allocated.

```

12721 \prop_new:N \g__ior_streams_prop
12722 \int_step_inline:nnn
12723 { 0 }
12724 {
12725 \cs_if_exist:NTF \normalend
12726 { \tex_count:D 38 ~ }
12727 {
12728 \tex_count:D 16 ~ %
12729 \cs_if_exist:NT \locount { - 1 }
12730 }
12731 }
12732 {
12733 \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved-by~format }
12734 }

```

(End definition for \g\_\_ior\_streams\_prop.)

### 20.1.2 Stream management

\ior\_new:N Reserving a new stream is done by defining the name as equal to using the terminal.  
\ior\_new:c

```

12735 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c__ior_term_ior }
12736 \cs_generate_variant:Nn \ior_new:N { c }

```

(End definition for \ior\_new:N. This function is documented on page 158.)

\g\_tmpa\_ior The usual scratch space.

\g\_tmpb\_ior

```

12737 \ior_new:N \g_tmpa_ior
12738 \ior_new:N \g_tmpb_ior

```

(End definition for \g\_tmpa\_ior and \g\_tmpb\_ior. These variables are documented on page 165.)

\ior\_open:Nn Use the conditional version, with an error if the file is not found.

\ior\_open:cn

```

12739 \cs_new_protected:Npn \ior_open:Nn #1#2
12740 { \ior_open:NnF #1 {#2} { __kernel_file_missing:n {#2} } }
12741 \cs_generate_variant:Nn \ior_open:Nn { c }

```

(End definition for \ior\_open:Nn. This function is documented on page 158.)

\l\_\_ior\_file\_name\_tl Data storage.

```

12742 \tl_new:N \l__ior_file_name_tl

```

(End definition for \l\_\_ior\_file\_name\_tl.)

**\ior\_open:NnTF** An auxiliary searches for the file in the TeX, L<sup>A</sup>T<sub>E</sub>X 2<sub>E</sub> and L<sup>A</sup>T<sub>E</sub>X3 paths. Then pass the file found to the lower-level function which deals with streams. The `full_name` is empty when the file is not found.

```

12743 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
12744 {
12745 \file_get_full_name:nNTF {#2} \l__ior_file_name_tl
12746 {
12747 __kernel_ior_open:No #1 \l__ior_file_name_tl
12748 \prg_return_true:
12749 }
12750 { \prg_return_false: }
12751 }
12752 \prg_generate_conditional_variant:Nnn \ior_open:Nn { c } { T , F , TF }
```

(End definition for `\ior_open:NnTF`. This function is documented on page 159.)

**\\_\_ior\_new:N** Streams are reserved using `\newread` before they can be managed by `ior`. To prevent `ior` from being affected by redefinitions of `\newread` (such as done by the third-party package `morewrites`), this macro is saved here under a private name. The complicated code ensures that `\__ior_new:N` is not `\outer` despite plain TeX's `\newread` being `\outer`. For ConTeXt, we have to deal with the fact that `\newread` works like our own: it actually checks before altering definition.

```

12753 \exp_args:NNf \cs_new_protected:Npn __ior_new:N
12754 { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
12755 \cs_if_exist:NT \normalend
12756 {
12757 \cs_new_eq:NN __ior_new_aux:N __ior_new:N
12758 \cs_set_protected:Npn __ior_new:N #1
12759 {
12760 \cs_undefine:N #1
12761 __ior_new_aux:N #1
12762 }
12763 }
```

(End definition for `\__ior_new:N`.)

**\\_\_kernel\_ior\_open:Nn** The stream allocation itself uses the fact that there is a list of all of those available. Life gets more complex as it's important to keep things in sync. That is done using a two-part approach: any streams that have already been taken up by `ior` but are now free are tracked, so we first try those. If that fails, ask plain TeX or L<sup>A</sup>T<sub>E</sub>X 2<sub>E</sub> for a new stream and use that number (after a bit of conversion).

```

12764 \cs_new_protected:Npn __kernel_ior_open:Nn #1#2
12765 {
12766 \ior_close:N #1
12767 \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
12768 { __ior_open_stream:Nn #1 {#2} }
12769 {
12770 __ior_new:N #1
12771 __kernel_tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
12772 __ior_open_stream:Nn #1 {#2}
12773 }
12774 }
12775 \cs_generate_variant:Nn __kernel_ior_open:Nn { No }
```

Here, we act defensively in case LuaTeX is in use with an extensionless file name.

```

12776 \cs_new_protected:Npx __ior_open_stream:Nn #1#2
12777 {
12778 \tex_global:D \tex_chardef:D #1 = \exp_not:N \l__ior_stream_tl \scan_stop:
12779 \prop_gput:NVn \exp_not:N \g__ior_streams_prop #1 {#2}
12780 \tex_openin:D #1
12781 \sys_if_engine_luatex:TF
12782 { {#2} }
12783 { \exp_not:N __kernel_file_name_quote:n {#2} \scan_stop: }
12784 }

```

(End definition for `\__kernel_ior_open:Nn` and `\__ior_open_stream:Nn`.)

**\ior\_close:N** Closing a stream means getting rid of it at the TeX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range [0, 15]), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

```

12785 \cs_new_protected:Npn \ior_close:N #1
12786 {
12787 \int_compare:nT { -1 < #1 < \c__ior_term_ior }
12788 {
12789 \tex_closein:D #1
12790 \prop_gremove:NV \g__ior_streams_prop #1
12791 \seq_if_in:NVF \g__ior_streams_seq #1
12792 { \seq_gpush:NV \g__ior_streams_seq #1 }
12793 \cs_gset_eq:NN #1 \c__ior_term_ior
12794 }
12795 }
12796 \cs_generate_variant:Nn \ior_close:N { c }

```

(End definition for `\ior_close:N`. This function is documented on page 159.)

**\ior\_show\_list:** Show the property lists, but with some “pretty printing”. See the `l3msg` module. The first argument of the message is `ior` (as opposed to `iow`) and the second is empty if no read stream is open and non-empty (the list of streams formatted using `\msg_show_item_unbraced:nn`) otherwise. The code of the message `show-streams` takes care of translating `ior/iow` to English.

```

12797 \cs_new_protected:Npn \ior_show_list: { __ior_list:N \msg_show:nnxxxx }
12798 \cs_new_protected:Npn \ior_log_list: { __ior_list:N \msg_log:nnxxxx }
12799 \cs_new_protected:Npn __ior_list:N #1
12800 {
12801 #1 { LaTeX / kernel } { show-streams }
12802 { ior }
12803 {
12804 \prop_map_function:NN \g__ior_streams_prop
12805 \msg_show_item_unbraced:nn
12806 }
12807 { } { }
12808 }

```

(End definition for `\ior_show_list:`, `\ior_log_list:`, and `\__ior_list:N`. These functions are documented on page 159.)

### 20.1.3 Reading input

**\if\_eof:w** The primitive conditional

```
12809 \cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(End definition for **\if\_eof:w**. This function is documented on page 165.)

- \ior\_if\_eof\_p:N** To test if some particular input stream is exhausted the following conditional is provided.
- \ior\_if\_eof:NTF** The primitive test can only deal with numbers in the range [0, 15] so we catch outliers (they are exhausted).

```
12810 \prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
12811 {
12812 \cs_if_exist:NTF #1
12813 {
12814 \int_compare:nTF { -1 < #1 < \c_ior_term_ior }
12815 {
12816 \if_eof:w #1
12817 \prg_return_true:
12818 \else:
12819 \prg_return_false:
12820 \fi:
12821 }
12822 { \prg_return_true: }
12823 }
12824 { \prg_return_true: }
12825 }
```

(End definition for **\ior\_if\_eof:NTF**. This function is documented on page 162.)

**\ior\_get:NN** And here we read from files.

```
12826 \cs_new_protected:Npn \ior_get:NN #1#2
12827 { \ior_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
12828 \cs_new_protected:Npn __ior_get:NN #1#2
12829 { \tex_read:D #1 to #2 }
12830 \prg_new_protected_conditional:Npnn \ior_get:NN #1#2 { T , F , TF }
12831 {
12832 \ior_if_eof:NTF #1
12833 { \prg_return_false: }
12834 {
12835 __ior_get:NN #1 #2
12836 \prg_return_true:
12837 }
12838 }
```

(End definition for **\ior\_get:NN**, **\\_\_ior\_get:NN**, and **\ior\_get:NNTF**. These functions are documented on page 160.)

- \ior\_str\_get:NN** Reading as strings is a more complicated wrapper, as we wish to remove the endline character and restore it afterwards.
- \\_\_ior\_str\_get:NN**
- \ior\_str\_get:NNTF**

```
12839 \cs_new_protected:Npn \ior_str_get:NN #1#2
12840 { \ior_str_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
12841 \cs_new_protected:Npn __ior_str_get:NN #1#2
12842 {
12843 \exp_args:Nno \use:n
12844 {
```

```

12845 \int_set:Nn \tex_endlinechar:D { -1 }
12846 \tex_readline:D #1 to #2
12847 \int_set:Nn \tex_endlinechar:D
12848 } { \int_use:N \tex_endlinechar:D }
12849 }
12850 \prg_new_protected_conditional:Npnn \ior_str_get:NN #1#2 { T , F , TF }
12851 {
12852 \ior_if_eof:NTF #1
12853 { \prg_return_false: }
12854 {
12855 __ior_str_get:NN #1 #2
12856 \prg_return_true:
12857 }
12858 }

```

(End definition for `\ior_str_get:NN`, `\__ior_str_get:NN`, and `\ior_str_get:NNTF`. These functions are documented on page 160.)

`\c_ior_term_noprompt_ior` For reading without a prompt.

```
12859 \int_const:Nn \c_ior_term_noprompt_ior { -1 }
```

(End definition for `\c_ior_term_noprompt_ior`.)

`\ior_get_term:nN` Getting from the terminal is better with pretty-printing.

```

12860 \cs_new_protected:Npn \ior_get_term:nN #1#2
12861 { __ior_get_term:NnN __ior_get>NN {#1} #2 }
12862 \cs_new_protected:Npn \ior_str_get_term:nN #1#2
12863 { __ior_get_term:NnN __ior_str_get>NN {#1} #2 }
12864 \cs_new_protected:Npn __ior_get_term:NnN #1#2#3
12865 {
12866 \group_begin:
12867 \tex_escapechar:D = -1 \scan_stop:
12868 \tl_if_blank:nTF {#2}
12869 { \exp_args:NNc #1 \c_ior_term_noprompt_ior }
12870 { \exp_args:NNc #1 \c_ior_term_ior }
12871 {#2}
12872 \exp_args:NNNv \group_end:
12873 \tl_set:Nn #3 {#2}
12874 }

```

(End definition for `\ior_get_term:nN`, `\ior_str_get_term:nN`, and `\__ior_get_term:NnN`. These functions are documented on page 267.)

`\ior_map_break:` Usual map breaking functions.

```

12875 \cs_new:Npn \ior_map_break:
12876 { \prg_map_break:Nn \ior_map_break: { } }
12877 \cs_new:Npn \ior_map_break:n
12878 { \prg_map_break:Nn \ior_map_break: }

```

(End definition for `\ior_map_break:` and `\ior_map_break:n`. These functions are documented on page 161.)

```

\ior_map_inline:Nn
\ior_str_map_inline:Nn
__ior_map_inline:NNn
__ior_map_inline:NNNn
__ior_map_inline_loop:NNN

12879 \cs_new_protected:Npn \ior_map_inline:Nn
12880 { __ior_map_inline:NNn __ior_get:NN }
12881 \cs_new_protected:Npn \ior_str_map_inline:Nn
12882 { __ior_map_inline:NNn __ior_str_get:NN }
12883 \cs_new_protected:Npn __ior_map_inline:NNn
12884 {
12885 \int_gincr:N \g__kernel_prg_map_int
12886 \exp_args:Nc __ior_map_inline:NNNn
12887 { __ior_map_ \int_use:N \g__kernel_prg_map_int :n }
12888 }
12889 \cs_new_protected:Npn __ior_map_inline:NNNn #1#2#3#4
12890 {
12891 \cs_gset_protected:Npn #1 ##1 {#4}
12892 \ior_if_eof:NF #3 { __ior_map_inline_loop:NNN #1#2#3 }
12893 \prg_break_point:Nn \ior_map_break:
12894 { \int_gdecr:N \g__kernel_prg_map_int }
12895 }
12896 \cs_new_protected:Npn __ior_map_inline_loop:NNN #1#2#3
12897 {
12898 #2 #3 \l__ior_internal_tl
12899 \if_eof:w #3
12900 \exp_after:wN \ior_map_break:
12901 \fi:
12902 \exp_args:No #1 \l__ior_internal_tl
12903 __ior_map_inline_loop:NNN #1#2#3
12904 }

```

(End definition for `\ior_map_inline:Nn` and others. These functions are documented on page 161.)

Since the TeX primitive (`\read` or `\readline`) assigns the tokens read in the same way as a token list assignment, we simply call the appropriate primitive. The end-of-loop is checked using the primitive conditional for speed.

```

\ior_map_variable:NNn
\ior_str_map_variable:NNn
__ior_map_variable:NNNNn
__ior_map_variable_loop:NNNn

12905 \cs_new_protected:Npn \ior_map_variable:NNn
12906 { __ior_map_variable:NNNn \ior_get:NN }
12907 \cs_new_protected:Npn \ior_str_map_variable:NNn
12908 { __ior_map_variable:NNNn \ior_str_get:NN }
12909 \cs_new_protected:Npn __ior_map_variable:NNNn #1#2#3#4
12910 {
12911 \ior_if_eof:NF #2 { __ior_map_variable_loop:NNNn #1#2#3 {#4} }
12912 \prg_break_point:Nn \ior_map_break: { }
12913 }
12914 \cs_new_protected:Npn __ior_map_variable_loop:NNNn #1#2#3#4
12915 {
12916 #1 #2 #3
12917 \if_eof:w #2
12918 \exp_after:wN \ior_map_break:
12919 \fi:
12920 #4
12921 __ior_map_variable_loop:NNNn #1#2#3 {#4}
12922 }

```

(End definition for `\ior_map_variable:Nn` and others. These functions are documented on page 161.)

## 20.2 Output operations

12923 `<@@=iow>`

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

### 20.2.1 Variables and constants

`\c_log_iow`  
`\c_term_iow` Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`) and to both the terminal and transcript file (`\c_term_iow`). Recent LuaTeX provide 128 write streams; we also use `\c_term_iow` as the first non-allowed write stream so its value depends on the engine.

```
12924 \int_const:Nn \c_log_iow { -1 }
12925 \int_const:Nn \c_term_iow
12926 {
12927 \bool_lazy_and:nnTF
12928 { \sys_if_engine_luatex_p: }
12929 { \int_compare_p:nNn \tex_luatexversion:D > { 80 } }
12930 { 128 }
12931 { 16 }
12932 }
```

(End definition for `\c_log_iow` and `\c_term_iow`. These variables are documented on page 165.)

`\g__iow_streams_seq` A list of the currently-available output streams to be used as a stack.

12933 `\seq_new:N \g__iow_streams_seq`

(End definition for `\g__iow_streams_seq`.)

`\l__iow_stream_tl` Used to recover the raw stream number from the stack.

12934 `\tl_new:N \l__iow_stream_tl`

(End definition for `\l__iow_stream_tl`.)

`\g__iow_streams_prop` As for reads with the appropriate adjustment of the register numbers to check on.

```
12935 \prop_new:N \g__iow_streams_prop
12936 \int_step_inline:nnn
12937 { 0 }
12938 {
12939 \cs_if_exist:NTF \normalend
12940 { \tex_count:D 39 ~ }
12941 {
12942 \tex_count:D 17 ~
12943 \cs_if_exist:NT \loccount { - 1 }
12944 }
12945 }
12946 {
12947 \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved-by-format }
12948 }
```

(End definition for `\g__iow_streams_prop`.)

## 20.2.2 Internal auxiliaries

`\s__iow_mark` Internal scan marks.

```
12949 \scan_new:N \s__iow_mark
12950 \scan_new:N \s__iow_stop
```

(*End definition for `\s__iow_mark` and `\s__iow_stop`.*)

`\_iow_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

```
12951 \cs_new:Npn _iow_use_i_delimit_by_s_stop:nw #1 #2 \s__iow_stop {#1}
```

(*End definition for `\_iow_use_i_delimit_by_s_stop:nw`.*)

`\q__iow_nil` Internal quarks.

```
12952 \quark_new:N \q__iow_nil
```

(*End definition for `\q__iow_nil`.*)

## 20.3 Stream management

`\iow_new:N` Reserving a new stream is done by defining the name as equal to writing to the terminal:  
`\iow_new:c` odd but at least consistent.

```
12953 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
12954 \cs_generate_variant:Nn \iow_new:N { c }
```

(*End definition for `\iow_new:N`. This function is documented on page 158.*)

`\g_tmpa_iow` The usual scratch space.

```
12955 \iow_new:N \g_tmpa_iow
12956 \iow_new:N \g_tmpb_iow
```

(*End definition for `\g_tmpa_iow` and `\g_tmpb_iow`. These variables are documented on page 165.*)

`\_iow_new:N` As for read streams, copy `\newwrite`, making sure that it is not `\outer`.

```
12957 \exp_args:NNf \cs_new_protected:Npn _iow_new:N
12958 { \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }
```

(*End definition for `\_iow_new:N`.*)

`\l__iow_file_name_tl` Data storage.

```
12959 \tl_new:N \l__iow_file_name_tl
```

(*End definition for `\l__iow_file_name_tl`.*)

`\iow_open:Nn` The same idea as for reading, but without the path and without the need to allow for a  
`\iow_open:cn` conditional version.

```
12960 \cs_new_protected:Npn \iow_open:Nn #1#2
12961 {
12962 _kernel_tl_set:Nx \l__iow_file_name_tl
12963 { _kernel_file_name_SANITIZE:n {#2} }
12964 \iow_close:N #1
12965 \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
12966 { _iow_open_stream:NV #1 \l__iow_file_name_tl }
12967 {
12968 _iow_new:N #1
```

```

12969 __kernel_tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
12970 __iow_open_stream:NV #1 \l__iow_file_name_tl
12971 }
12972 }
12973 \cs_generate_variant:Nn \iow_open:Nn { c }
12974 \cs_new_protected:Npn __iow_open_stream:Nn #1#2
12975 {
12976 \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
12977 \prop_gput:NVn \g__iow_streams_prop #1 {#2}
12978 \tex_immediate:D \tex_openout:D
12979 #1 __kernel_file_name_quote:n {#2} \scan_stop:
12980 }
12981 \cs_generate_variant:Nn __iow_open_stream:Nn { NV }

(End definition for \iow_open:Nn and __iow_open_stream:Nn. This function is documented on page 159.)
```

**\iow\_close:N** Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

```

12982 \cs_new_protected:Npn \iow_close:N #1
12983 {
12984 \int_compare:nT { - \c_log_iow < #1 < \c_term_iow }
12985 {
12986 \tex_immediate:D \tex_closeout:D #1
12987 \prop_gremove:NV \g__iow_streams_prop #1
12988 \seq_if_in:NVF \g__iow_streams_seq #1
12989 { \seq_gpush:NV \g__iow_streams_seq #1 }
12990 \cs_gset_eq:NN #1 \c_term_iow
12991 }
12992 }
12993 \cs_generate_variant:Nn \iow_close:N { c }

(End definition for \iow_close:N. This function is documented on page 159.)
```

**\iow\_show\_list:** Done as for input, but with a copy of the auxiliary so the name is correct.

```

12994 \cs_new_protected:Npn \iow_show_list: { __iow_list:N \msg_show:nnxxxx }
12995 \cs_new_protected:Npn \iow_log_list: { __iow_list:N \msg_log:nnxxxx }
12996 \cs_new_protected:Npn __iow_list:N #1
12997 {
12998 #1 { LaTeX / kernel } { show-streams }
12999 { iow }
13000 {
13001 \prop_map_function:NN \g__iow_streams_prop
13002 \msg_show_item_unbraced:nn
13003 }
13004 { } { }
13005 }
```

(End definition for \iow\_show\_list:, \iow\_log\_list:, and \\_\_iow\_list:N. These functions are documented on page 159.)

### 20.3.1 Deferred writing

\iow\_shipout\_x:Nn

First the easy part, this is the primitive, which expects its argument to be braced.

```
13006 \cs_new_protected:Npn \iow_shipout_x:Nn #1#2
13007 { \tex_write:D #1 {#2} }
13008 \cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }
```

(End definition for \iow\_shipout\_x:Nn. This function is documented on page 163.)

\iow\_shipout:Nn

With  $\varepsilon$ - $\text{\TeX}$  available deferred writing without expansion is easy.

```
13009 \cs_new_protected:Npn \iow_shipout:Nn #1#2
13010 { \tex_write:D #1 { \exp_not:n {#2} } }
13011 \cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }
```

(End definition for \iow\_shipout:Nn. This function is documented on page 163.)

### 20.3.2 Immediate writing

\\_kernel\_iow\_with:Nnn

\\_iow\_with:nNnn

If the integer #1 is equal to #2, just leave #3 in the input stream. Otherwise, pass the old value to an auxiliary, which sets the integer to the new value, runs the code, and restores the integer.

```
13012 \cs_new_protected:Npn _kernel_iow_with:Nnn #1#2
13013 {
13014 \int_compare:nNnTF {#1} = {#2}
13015 { \use:n }
13016 { \exp_args:No _iow_with:nNnn { \int_use:N #1 } #1 {#2} }
13017 }
13018 \cs_new_protected:Npn _iow_with:nNnn #1#2#3#4
13019 {
13020 \int_set:Nn #2 {#3}
13021 #4
13022 \int_set:Nn #2 {#1}
13023 }
```

(End definition for \\_kernel\_iow\_with:Nnn and \\_iow\_with:nNnn.)

\iow\_now:Nn

\iow\_now:Nx

\iow\_now:cn

\iow\_now:cx

This routine writes the second argument onto the output stream without expansion. If this stream isn't open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error. We don't use the expansion done by \write to get the Nx variant, because it differs in subtle ways from x-expansion, namely, macro parameter characters would not need to be doubled. We set the \newlinechar to 10 using \\_kernel\_iow\_with:Nnn to support formats such as plain  $\text{\TeX}$ : otherwise, \iow\_newline: would not work. We do not do this for \iow\_shipout:Nn or \iow\_shipout\_x:Nn, as  $\text{\TeX}$  looks at the value of the \newlinechar at shipout time in those cases.

```
13024 \cs_new_protected:Npn \iow_now:Nn #1#2
13025 {
13026 _kernel_iow_with:Nnn \tex_newlinechar:D { '\^J' }
13027 { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
13028 }
13029 \cs_generate_variant:Nn \iow_now:Nn { c, Nx, cx }
```

(End definition for \iow\_now:Nn. This function is documented on page 162.)

```

\iow_log:n Writing to the log and the terminal directly are relatively easy.
\iow_log:x 13030 \cs_set_protected:Npn \iow_log:x { \iow_now:Nx \c_log_iow }
\iow_term:n 13031 \cs_new_protected:Npn \iow_log:n { \iow_now:Nn \c_log_iow }
\iow_term:x 13032 \cs_set_protected:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
\iow_term:n 13033 \cs_new_protected:Npn \iow_term:n { \iow_now:Nn \c_term_iow }

```

(End definition for *\iow\_log:n* and *\iow\_term:n*. These functions are documented on page 162.)

### 20.3.3 Special characters for writing

*\iow\_newline:* Global variable holding the character that forces a new line when something is written to an output stream.

```
13034 \cs_new:Npn \iow_newline: { ^J }
```

(End definition for *\iow\_newline:*. This function is documented on page 163.)

*\iow\_char:N* Function to write any escaped char to an output stream.

```
13035 \cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(End definition for *\iow\_char:N*. This function is documented on page 163.)

### 20.3.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

*\l\_iow\_line\_count\_int* This is the “raw” number of characters in a line which can be written to the terminal. The standard value is the line length typically used by TeXLive and MiK $\mathrm{\TeX}$ .

```

13036 \int_new:N \l_iow_line_count_int
13037 \int_set:Nn \l_iow_line_count_int { 78 }

```

(End definition for *\l\_iow\_line\_count\_int*. This variable is documented on page 164.)

*\l\_\_iow\_newline\_tl* The token list inserted to produce a new line, with the *<run-on text>*.

```
13038 \tl_new:N \l__iow_newline_tl
```

(End definition for *\l\_\_iow\_newline\_tl*.)

*\l\_\_iow\_line\_target\_int* This stores the target line count: the full number of characters in a line, minus any part for a leader at the start of each line.

```
13039 \int_new:N \l__iow_line_target_int
```

(End definition for *\l\_\_iow\_line\_target\_int*.)

*\\_\_iow\_set\_indent:n* The *one\_indent* variables hold one indentation marker and its length. The *\\_\_iow\_unindent:w* auxiliary removes one indentation. The function *\\_\_iow\_set\_indent:n* (that could possibly be public) sets the indentation in a consistent way. We set it to four spaces by default.

```

13040 \tl_new:N \l__iow_one_indent_tl
13041 \int_new:N \l__iow_one_indent_int
13042 \cs_new:Npn __iow_unindent:w { }
13043 \cs_new_protected:Npn __iow_set_indent:n #1
13044 {
 __kernel_tl_set:Nx \l__iow_one_indent_tl

```

```

13046 { \exp_args:N __kernel_str_to_other_fast:n { \tl_to_str:n {#1} } }
13047 \int_set:Nn \l__iow_one_indent_int
13048 { \str_count:N \l__iow_one_indent_tl }
13049 \exp_last_unbraced:NNo
13050 \cs_set:Npn __iow_unindent:w \l__iow_one_indent_tl { }
13051 }
13052 \exp_args:Nx __iow_set_indent:n { \prg_replicate:nn { 4 } { ~ } }

(End definition for __iow_set_indent:n and others.)

```

\l\_\_iow\_indent\_tl The current indentation (some copies of \l\_\_iow\_one\_indent\_tl) and its number of characters.

```

13053 \tl_new:N \l__iow_indent_tl
13054 \int_new:N \l__iow_indent_int

```

(End definition for \l\_\_iow\_indent\_tl and \l\_\_iow\_indent\_int.)

\l\_\_iow\_line\_tl These hold the current line of text and a partial line to be added to it, respectively.  
\l\_\_iow\_line\_part\_tl

```

13055 \tl_new:N \l__iow_line_tl
13056 \tl_new:N \l__iow_line_part_tl

```

(End definition for \l\_\_iow\_line\_tl and \l\_\_iow\_line\_part\_tl.)

\l\_\_iow\_line\_break\_bool Indicates whether the line was broken precisely at a chunk boundary.

```
13057 \bool_new:N \l__iow_line_break_bool
```

(End definition for \l\_\_iow\_line\_break\_bool.)

\l\_\_iow\_wrap\_tl Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.

```
13058 \tl_new:N \l__iow_wrap_tl
```

(End definition for \l\_\_iow\_wrap\_tl.)

\c\_\_iow\_wrap\_marker\_tl Every special action of the wrapping code starts with the same recognizable string, \c\_\_iow\_wrap\_end\_marker\_tl. Upon seeing that “word”, the wrapping code reads one space-delimited argument to know what operation to perform. The setting of \escapechar here is not very important, but makes \c\_\_iow\_wrap\_marker\_tl look marginally nicer.

```

13059 \group_begin:
13060 \int_set:Nn \tex_escapechar:D { -1 }
13061 \tl_const:Nx \c__iow_wrap_marker_tl
13062 { \tl_to_str:n { \^I \^O \^W \^_ \^W \^R \^A \^P } }
13063 \group_end:
13064 \tl_map_inline:nn
13065 { { end } { newline } { allow_break } { indent } { unindent } }
13066 {
13067 \tl_const:c { c__iow_wrap_ #1 _marker_tl }
13068 {
13069 \c__iow_wrap_marker_tl
13070 #1
13071 \c_catcode_other_space_tl
13072 }
13073 }

```

(End definition for \c\_\_iow\_wrap\_marker\_tl and others.)

**\iow\_allow\_break:** We set `\iow_allow_break:n` to produce an error when outside messages. Within wrapped message, it is set to `\__iow_allow_break:` when valid and otherwise to `\__-iow_allow_break_error:`. The second produces an error expandably.

```

13074 \cs_new_protected:Npn \iow_allow_break:
13075 {
13076 __kernel_msg_error:nnnn { kernel } { iow-indent }
13077 { \iow_wrap:nnN } { \iow_allow_break: }
13078 }
13079 \cs_new:Npx __iow_allow_break: { \c__iow_wrap_allow_break_marker_tl }
13080 \cs_new:Npn __iow_allow_break_error:
13081 {
13082 __kernel_msg_expandable_error:nnnn { kernel } { iow-indent }
13083 { \iow_wrap:nnN } { \iow_allow_break: }
13084 }

```

(End definition for `\iow_allow_break:`, `\__iow_allow_break:`, and `\__iow_allow_break_error:`. This function is documented on page 266.)

**\iow\_indent:n** We set `\iow_indent:n` to produce an error when outside messages. Within wrapped message, it is set to `\__iow_indent:n` when valid and otherwise to `\__iow_indent_error:n`. The first places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. The second produces an error expandably. Note that there are no forced line-break, so the indentation only changes when the next line is started.

```

13085 \cs_new_protected:Npn \iow_indent:n #1
13086 {
13087 __kernel_msg_error:nnnnn { kernel } { iow-indent }
13088 { \iow_wrap:nnN } { \iow_indent:n } {#1}
13089 #1
13090 }
13091 \cs_new:Npx __iow_indent:n #1
13092 {
13093 \c__iow_wrap_indent_marker_tl
13094 #1
13095 \c__iow_wrap_unindent_marker_tl
13096 }
13097 \cs_new:Npn __iow_indent_error:n #1
13098 {
13099 __kernel_msg_expandable_error:nnnnn { kernel } { iow-indent }
13100 { \iow_wrap:nnN } { \iow_indent:n } {#1}
13101 #1
13102 }

```

(End definition for `\iow_indent:n`, `\__iow_indent:n`, and `\__iow_indent_error:n`. This function is documented on page 164.)

**\iow\_wrap:nnnN** **\iow\_wrap:nxnN** The main wrapping function works as follows. First give `\\\`, `\_` and other formatting commands the correct definition for messages and perform the given setup #3. The definition of `\_` uses an “other” space rather than a normal space, because the latter might be absorbed by TeX to end a number or other f-type expansions. Use `\conditionally@traceoff` if defined; it is introduced by the trace package and suppresses uninteresting tracing of the wrapping code.

```

13103 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4

```

```

13104 {
13105 \group_begin:
13106 \cs_if_exist_use:N \conditionally@traceoff
13107 \int_set:Nn \tex_escapechar:D { -1 }
13108 \cs_set:Npx \{ { \token_to_str:N \{ }
13109 \cs_set:Npx \# { \token_to_str:N \# }
13110 \cs_set:Npx \} { \token_to_str:N \} }
13111 \cs_set:Npx \% { \token_to_str:N \% }
13112 \cs_set:Npx \~ { \token_to_str:N \~ }
13113 \int_set:Nn \tex_escapechar:D { 92 }
13114 \cs_set_eq:NN \\ \iow_newline:
13115 \cs_set_eq:NN \c_catcode_other_space_tl
13116 \cs_set_eq:NN \iow_allow_break: _iow_allow_break:
13117 \cs_set_eq:NN \iow_indent:n _iow_indent:n
13118 #3

```

Then fully-expand the input: in package mode, the expansion uses L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub> 's `\protect` mechanism in the same way as `\typeout`. In generic mode this setting is useless but harmless. As soon as the expansion is done, reset `\iow_indent:n` to its error definition: it only works in the first argument of `\iow_wrap:nnnN`.

```

13119 \cs_set_eq:NN \protect \token_to_str:N
13120 _kernel_tl_set:Nx \l_iow_wrap_tl {#1}
13121 \cs_set_eq:NN \iow_allow_break: _iow_allow_break_error:
13122 \cs_set_eq:NN \iow_indent:n _iow_indent_error:n

```

Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and initialize the target count for lines (the first line has target count `\l_iow_line_count_int` instead).

```

13123 _kernel_tl_set:Nx \l_iow_newline_tl { \iow_newline: #2 }
13124 _kernel_tl_set:Nx \l_iow_newline_tl { \tl_to_str:N \l_iow_newline_tl }
13125 \int_set:Nn \l_iow_line_target_int
13126 { \l_iow_line_count_int - \str_count:N \l_iow_newline_tl + 1 }

```

Sanity check.

```

13127 \int_compare:nNnT { \l_iow_line_target_int } < 0
13128 {
13129 \tl_set:Nn \l_iow_newline_tl { \iow_newline: }
13130 \int_set:Nn \l_iow_line_target_int
13131 { \l_iow_line_count_int + 1 }
13132 }

```

There is then a loop over the input, which stores the wrapped result in `\l_iow_wrap_tl`. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The `\tl_to_str:N` step converts the “other” spaces back to normal spaces. The f-expansion removes a leading space from `\l_iow_wrap_tl`.

```

13133 _iow_wrap_do:
13134 \exp_args:NNf \group_end:
13135 #4 { \tl_to_str:N \l_iow_wrap_tl }
13136 }
13137 \cs_generate_variant:Nn \iow_wrap:nnnN { nx }

```

(End definition for `\iow_wrap:nnnN`. This function is documented on page 164.)

`\_iow_wrap_do:` Escape spaces and change newlines to `\c_iow_wrap_newline_marker_tl`. Set up a few variables, in particular the initial value of `\l_iow_wrap_tl`: the space stops the

`\_iow_wrap_start:w`

f-expansion of the main wrapping function and \use\_none:n removes a newline marker inserted by later code. The main loop consists of repeatedly calling the chunk auxiliary to wrap chunks delimited by (newline or indentation) markers.

```

13138 \cs_new_protected:Npn __iow_wrap_do:
13139 {
13140 __kernel_tl_set:Nx \l__iow_wrap_tl
13141 {
13142 \exp_args:No __kernel_str_to_other_fast:n \l__iow_wrap_tl
13143 \c__iow_end_marker_tl
13144 }
13145 __kernel_tl_set:Nx \l__iow_wrap_tl
13146 {
13147 \exp_after:wN __iow_wrap_fix_newline:w \l__iow_wrap_tl
13148 ^^J \q__iow_nil ^^J \s__iow_stop
13149 }
13150 \exp_after:wN __iow_wrap_start:w \l__iow_wrap_tl
13151 }
13152 \cs_new:Npn __iow_wrap_fix_newline:w #1 ^^J #2 ^^J
13153 {
13154 #1
13155 \if_meaning:w \q__iow_nil #2
13156 __iow_use_i_delimit_by_s_stop:nw
13157 \fi:
13158 \c__iow_wrap_newline_marker_tl
13159 __iow_wrap_fix_newline:w #2 ^^J
13160 }
13161 \cs_new_protected:Npn __iow_wrap_start:w
13162 {
13163 \bool_set_false:N \l__iow_line_break_bool
13164 \tl_clear:N \l__iow_line_tl
13165 \tl_clear:N \l__iow_line_part_tl
13166 \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }
13167 \int_zero:N \l__iow_indent_int
13168 \tl_clear:N \l__iow_indent_tl
13169 __iow_wrap_chunk:nw { \l__iow_line_count_int }
13170 }
```

(End definition for \\_\_iow\_wrap\_do:, \\_\_iow\_wrap\_fix\_newline:w, and \\_\_iow\_wrap\_start:w.)

\\_\_iow\_wrap\_chunk:nw  
\\_\_iow\_wrap\_next:nw

The **chunk** and **next** auxiliaries are defined indirectly to obtain the expansions of \c\_catcode\_other\_space\_tl and \c\_\_iow\_wrap\_marker\_tl in their definition. The **next** auxiliary calls a function corresponding to the type of marker (its ##2), which can be **newline** or **indent** or **unindent** or **end**. The first argument of the **chunk** auxiliary is a target number of characters and the second is some string to wrap. If the chunk is empty simply call **next**. Otherwise, set up a call to \\_\_iow\_wrap\_line:nw, including the indentation if the current line is empty, and including a trailing space (#1) before the \\_\_iow\_wrap\_end\_chunk:w auxiliary.

```

13171 \cs_set_protected:Npn __iow_tmp:w #1#2
13172 {
13173 \cs_new_protected:Npn __iow_wrap_chunk:nw ##1##2 #2
13174 {
13175 \tl_if_empty:nTF {##2}
13176 }
```

```

13177 \tl_clear:N \l__iow_line_part_tl
13178 __iow_wrap_next:nw {##1}
13179 }
13180 {
13181 \tl_if_empty:NTF \l__iow_line_tl
13182 {
13183 __iow_wrap_line:nw
13184 { \l__iow_indent_tl }
13185 ##1 - \l__iow_indent_int ;
13186 }
13187 { __iow_wrap_line:nw { } ##1 ; }
13188 ##2 #1
13189 __iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \s__iow_stop
13190 }
13191 }
13192 \cs_new_protected:Npn __iow_wrap_next:nw ##1##2 #1
13193 { \use:c { __iow_wrap_##2:n } {##1} }
13194 }
13195 \exp_args:NVV __iow_tmp:w \c_catcode_other_space_tl \c__iow_wrap_marker_tl

```

(End definition for `\__iow_wrap_chunk:nw` and `\__iow_wrap_next:nw`.)

```

__iow_wrap_line:nw
__iow_wrap_line_loop:w
__iow_wrap_line_aux:Nw
__iow_wrap_line_seven:nnnnnnn
__iow_wrap_line_end:NNNNNNNN
__iow_wrap_line_end:nw
__iow_wrap_end_chunk:w

```

This is followed by `{<string>} <intexpr> ;`. It stores the `<string>` and up to `<intexpr>` characters from the current chunk into `\l__iow_line_part_tl`. Characters are grabbed 8 at a time and left in `\l__iow_line_part_tl` by the `line_loop` auxiliary. When  $k < 8$  remain to be found, the `line_aux` auxiliary calls the `line_end` auxiliary followed by (the single digit)  $k$ , then  $7 - k$  empty brace groups, then the chunk's remaining characters. The `line_end` auxiliary leaves  $k$  characters from the chunk in the line part, then ends the assignment. Ignore the `\use_none:nnnnn` line for now. If the next character is a space the line can be broken there: store what we found into the result and get the next line. Otherwise some work is needed to find a break-point. So far we have ignored what happens if the chunk is shorter than the requested number of characters: this is dealt with by the `end_chunk` auxiliary, which gets treated like a character by the rest of the code. It ends up being called either as one of the arguments `#2-#9` of the `line_loop` auxiliary or as one of the arguments `#2-#8` of the `line_end` auxiliary. In both cases stop the assignment and work out how many characters are still needed. Notice that when we have exactly seven arguments to clean up, a `\exp_stop_f:` has to be inserted to stop the `\exp:w`. The weird `\use_none:nnnnn` ensures that the required data is in the right place.

```

13196 \cs_new_protected:Npn __iow_wrap_line:nw #1
13197 {
13198 \tex_eodef:D \l__iow_line_part_tl { \if_false: } \fi:
13199 #1
13200 \exp_after:wN __iow_wrap_line_loop:w
13201 \int_value:w \int_eval:w
13202 }
13203 \cs_new:Npn __iow_wrap_line_loop:w #1 ; #2#3#4#5#6#7#8#9
13204 {
13205 \if_int_compare:w #1 < 8 \exp_stop_f:
13206 __iow_wrap_line_aux:Nw #1
13207 \fi:
13208 #2 #3 #4 #5 #6 #7 #8 #9
13209 \exp_after:wN __iow_wrap_line_loop:w

```

```

13210 \int_value:w \int_eval:w #1 - 8 ;
13211 }
13212 \cs_new:Npn __iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 ;
13213 {
13214 #2
13215 \exp_after:wN __iow_wrap_line_end:NnnnnnnN
13216 \exp_after:wN #1
13217 \exp:w \exp_end_continue_f:w
13218 \exp_after:wN \exp_after:wN
13219 \if_case:w #1 \exp_stop_f:
13220 \prg_do_nothing:
13221 \or: \use_none:n
13222 \or: \use_none:nn
13223 \or: \use_none:nnn
13224 \or: \use_none:nnnn
13225 \or: \use_none:nnnnn
13226 \or: \use_none:nnnnnn
13227 \or: __iow_wrap_line_seven:nnnnnnn
13228 \fi:
13229 { } { } { } { } { } { } { } #3
13230 }
13231 \cs_new:Npn __iow_wrap_line_seven:nnnnnnn #1#2#3#4#5#6#7 { \exp_stop_f: }
13232 \cs_new:Npn __iow_wrap_line_end:NnnnnnnnnN #1#2#3#4#5#6#7#8#9
13233 {
13234 #2 #3 #4 #5 #6 #7 #8
13235 \use_none:nnnnn \int_eval:w 8 - ; #9
13236 \token_if_eq_charcode:NNTF \c_space_token #9
13237 { __iow_wrap_line_end:nw { } }
13238 { \if_false: { \fi: } __iow_wrap_break:w #9 }
13239 }
13240 \cs_new:Npn __iow_wrap_line_end:nw #
13241 {
13242 \if_false: { \fi: }
13243 __iow_wrap_store_do:n {#1}
13244 __iow_wrap_next_line:w
13245 }
13246 \cs_new:Npn __iow_wrap_end_chunk:w
13247 #1 \int_eval:w #2 - #3 ; #4#5 \s__iow_stop
13248 {
13249 \if_false: { \fi: }
13250 \exp_args:Nf __iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
13251 }

```

(End definition for `\__iow_wrap_line:nw` and others.)

`\__iow_wrap_break:w` Functions here are defined indirectly: `\__iow_tmp:w` is eventually called with an “other”  
`\__iow_wrap_break_first:w` space as its argument. The goal is to remove from `\l__iow_line_part_tl` the part  
`\__iow_wrap_break_none:w` after the last space. In most cases this is done by repeatedly calling the `break_loop`  
`\__iow_wrap_break_loop:w` auxiliary, which leaves “words” (delimited by spaces) until it hits the trailing space: then  
`\__iow_wrap_break_end:w` its argument `##3` is ? `\__iow_wrap_break_end:w` instead of a single token, and that  
`break_end` auxiliary leaves in the assignment the line until the last space, then calls  
`\__iow_wrap_line_end:nw` to finish up the line and move on to the next. If there is  
`\__iow_wrap_line_end:nw` no space in `\l__iow_line_part_tl` then the `break_first` auxiliary calls the `break_-`  
`none` auxiliary. In that case, if the current line is empty, the complete word (including

##4, characters beyond what we had grabbed) is added to the line, making it over-long. Otherwise, the word is used for the following line (and the last space of the line so far is removed because it was inserted due to the presence of a marker).

```

13252 \cs_set_protected:Npn __iow_tmp:w #1
13253 {
13254 \cs_new:Npn __iow_wrap_break:w
13255 {
13256 \tex_eodef:D \l__iow_line_part_tl
13257 { \if_false: } \fi:
13258 \exp_after:wn __iow_wrap_break_first:w
13259 \l__iow_line_part_tl
13260 #1
13261 { ? __iow_wrap_break_end:w }
13262 \s__iow_mark
13263 }
13264 \cs_new:Npn __iow_wrap_break_first:w ##1 #1 ##2
13265 {
13266 \use_none:nn ##2 __iow_wrap_break_none:w
13267 __iow_wrap_break_loop:w ##1 #1 ##2
13268 }
13269 \cs_new:Npn __iow_wrap_break_none:w ##1##2 #1 ##3 \s__iow_mark ##4 #1
13270 {
13271 \tl_if_empty:NTF \l__iow_line_tl
13272 { ##2 ##4 __iow_wrap_line_end:nw { } }
13273 { __iow_wrap_line_end:nw { __iow_wrap_trim:N } ##2 ##4 #1 }
13274 }
13275 \cs_new:Npn __iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
13276 {
13277 \use_none:n ##3
13278 ##1 #1
13279 __iow_wrap_break_loop:w ##2 #1 ##3
13280 }
13281 \cs_new:Npn __iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \s__iow_mark
13282 { ##1 __iow_wrap_line_end:nw { } ##3 }
13283 }
13284 \exp_args:NV __iow_tmp:w \c_catcode_other_space_tl

```

(End definition for `\__iow_wrap_break:w` and others.)

`\__iow_wrap_next_line:w` The special case where the end of a line coincides with the end of a chunk is detected here, to avoid a spurious empty line. Otherwise, call `\__iow_wrap_line:nw` to find characters for the next line (remembering to account for the indentation).

```

13285 \cs_new_protected:Npn __iow_wrap_next_line:w #1#2 \s__iow_stop
13286 {
13287 \tl_clear:N \l__iow_line_tl
13288 \token_if_eq_meaning:NTF #1 __iow_wrap_end_chunk:w
13289 {
13290 \tl_clear:N \l__iow_line_part_tl
13291 \bool_set_true:N \l__iow_line_break_bool
13292 __iow_wrap_next:nw { \l__iow_line_target_int }
13293 }
13294 {
13295 __iow_wrap_line:nw
13296 { \l__iow_indent_tl }

```

```

13297 \l__iow_line_target_int - \l__iow_indent_int ;
13298 #1 #2 \s__iow_stop
13299 }
13300 }
```

(End definition for `\_\_iow_wrap_next_line:w`.)

`\_\_iow_wrap_allow_break:n` This is called after a chunk has been wrapped. The `\l__iow_line_part_tl` typically ends with a space (except at the beginning of a line?), which we remove since the `allow-break` marker should not insert a space. Then move on with the next chunk, making sure to adjust the target number of characters for the line in case we did remove a space.

```

13301 \cs_new_protected:Npn __iow_wrap_allow_break:n #1
13302 {
13303 __kernel_tl_set:Nx \l__iow_line_tl
13304 { \l__iow_line_tl __iow_wrap_trim:N \l__iow_line_part_tl }
13305 \bool_set_false:N \l__iow_line_break_bool
13306 \tl_if_empty:NTF \l__iow_line_part_tl
13307 { __iow_wrap_chunk:nw {#1} }
13308 { \exp_args:Nf __iow_wrap_chunk:nw { \int_eval:n { #1 + 1 } } }
13309 }
```

(End definition for `\_\_iow_wrap_allow_break:n`.)

`\_\_iow_wrap_indent:n` These functions are called after a chunk has been wrapped, when encountering `indent/unindent` markers. Add the line part (last line part of the previous chunk) to the line so far and reset a boolean denoting the presence of a line-break. Most importantly, add or remove one indent from the current indent (both the integer and the token list). Finally, continue wrapping.

```

13310 \cs_new_protected:Npn __iow_wrap_indent:n #1
13311 {
13312 \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
13313 \bool_set_false:N \l__iow_line_break_bool
13314 \int_add:Nn \l__iow_indent_int { \l__iow_one_indent_int }
13315 \tl_put_right:No \l__iow_indent_tl { \l__iow_one_indent_tl }
13316 __iow_wrap_chunk:nw {#1}
13317 }
13318 \cs_new_protected:Npn __iow_wrap_unindent:n #1
13319 {
13320 \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
13321 \bool_set_false:N \l__iow_line_break_bool
13322 \int_sub:Nn \l__iow_indent_int { \l__iow_one_indent_int }
13323 __kernel_tl_set:Nx \l__iow_indent_tl
13324 { \exp_after:wN __iow_unindent:w \l__iow_indent_tl }
13325 __iow_wrap_chunk:nw {#1}
13326 }
```

(End definition for `\_\_iow_wrap_indent:n` and `\_\_iow_wrap_unindent:n`.)

`\_\_iow_wrap_newline:n` These functions are called after a chunk has been line-wrapped, when encountering a `newline/end` marker. Unless we just took a line-break, store the line part and the line so far into the whole `\l__iow_wrap_tl`, trimming a trailing space. In the `newline` case look for a new line (of length `\l__iow_line_target_int`) in a new chunk.

```

13327 \cs_new_protected:Npn __iow_wrap_newline:n #1
13328 {
```

```

13329 \bool_if:NF \l_iow_line_break_bool
13330 { __iow_wrap_store_do:n { __iow_wrap_trim:N } }
13331 \bool_set_false:N \l_iow_line_break_bool
13332 __iow_wrap_chunk:nw { \l_iow_line_target_int }
13333 }
13334 \cs_new_protected:Npn __iow_wrap_end:n #1
13335 {
13336 \bool_if:NF \l_iow_line_break_bool
13337 { __iow_wrap_store_do:n { __iow_wrap_trim:N } }
13338 \bool_set_false:N \l_iow_line_break_bool
13339 }

```

(End definition for `\__iow_wrap_newline:n` and `\__iow_wrap_end:n`.)

`\__iow_wrap_store_do:n` First add the last line part to the line, then append it to `\l_iow_wrap_tl` with the appropriate new line (with “run-on” text), possibly with its last space removed (#1 is empty or `\__iow_wrap_trim:N`).

```

13340 \cs_new_protected:Npn __iow_wrap_store_do:n #1
13341 {
13342 __kernel_tl_set:Nx \l_iow_line_tl
13343 { \l_iow_line_tl \l_iow_line_part_tl }
13344 __kernel_tl_set:Nx \l_iow_wrap_tl
13345 {
13346 \l_iow_wrap_tl
13347 \l_iow_newline_tl
13348 #1 \l_iow_line_tl
13349 }
13350 \tl_clear:N \l_iow_line_tl
13351 }

```

(End definition for `\__iow_wrap_store_do:n`.)

`\__iow_wrap_trim:N` Remove one trailing “other” space from the argument if present.

```

13352 \cs_set_protected:Npn __iow_tmp:w #1
13353 {
13354 \cs_new:Npn __iow_wrap_trim:N ##1
13355 { \exp_after:wN __iow_wrap_trim:w ##1 \s_iow_mark #1 \s_iow_mark \s_iow_stop }
13356 \cs_new:Npn __iow_wrap_trim:w ##1 #1 \s_iow_mark
13357 { __iow_wrap_trim_aux:w ##1 \s_iow_mark }
13358 \cs_new:Npn __iow_wrap_trim_aux:w ##1 \s_iow_mark ##2 \s_iow_stop {##1}
13359 }
13360 \exp_args:NV __iow_tmp:w \c_catcode_other_space_tl

```

(End definition for `\__iow_wrap_trim:N`, `\__iow_wrap_trim:w`, and `\__iow_wrap_trim_aux:w`.)

13361 `<@=file>`

## 20.4 File operations

`\l_file_internal_tl` Used as a short-term scratch variable.

13362 `\tl_new:N \l_file_internal_tl`

(End definition for `\l_file_internal_tl`.)

`\g_file_curr_dir_str` The name of the current file should be available at all times: the name itself is set dynamically.

```
13363 \str_new:N \g_file_curr_dir_str
13364 \str_new:N \g_file_curr_ext_str
13365 \str_new:N \g_file_curr_name_str
```

(End definition for `\g_file_curr_dir_str`, `\g_file_curr_ext_str`, and `\g_file_curr_name_str`. These variables are documented on page 165.)

`\g__file_stack_seq` The input list of files is stored as a sequence stack. In package mode we can recover the information from the details held by L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  (we must be in the preamble and loaded using `\usepackage` or `\RequirePackage`). As L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  doesn't store directory and name separately, we stick to the same convention here. In pre-loading, `\@currnamestack` is empty so is skipped.

```
13366 \seq_new:N \g__file_stack_seq
13367 \group_begin:
13368 \cs_set_protected:Npn __file_tmp:w #1#2#3
13369 {
13370 \tl_if_blank:nTF {#1}
13371 {
13372 \cs_set:Npn __file_tmp:w ##1 " ##2 " ##3 \s__file_stop
13373 { { } {##2} { } }
13374 \seq_gput_right:Nx \g__file_stack_seq
13375 {
13376 \exp_after:wN __file_tmp:w \tex_jobname:D
13377 " \tex_jobname:D " \s__file_stop
13378 }
13379 }
13380 {
13381 \seq_gput_right:Nn \g__file_stack_seq { { } {#1} {#2} }
13382 __file_tmp:w
13383 }
13384 }
13385 \cs_if_exist:NT \@currnamestack
13386 {
13387 \tl_if_empty:NF \@currnamestack
13388 { \exp_after:wN __file_tmp:w \@currnamestack }
13389 }
13390 \group_end:
```

(End definition for `\g__file_stack_seq`.)

`\g__file_record_seq` The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list. The current file name should be included in the file list! We will eventually copy the contents of `\@filelist`.

```
13391 \seq_new:N \g__file_record_seq
```

(End definition for `\g__file_record_seq`.)

`\l__file_base_name_tl` For storing the basename and full path whilst passing data internally.

```
13392 \tl_new:N \l__file_base_name_tl
13393 \tl_new:N \l__file_full_name_tl
```

(End definition for `\l__file_base_name_tl` and `\l__file_full_name_tl`.)

\l\_file\_dir\_str    Used in parsing a path into parts: in contrast to the above, these are never used outside of the current module.

\l\_file\_ext\_str  
13394 \str\_new:N \l\_file\_dir\_str  
13395 \str\_new:N \l\_file\_ext\_str  
13396 \str\_new:N \l\_file\_name\_str

(End definition for \l\_file\_dir\_str, \l\_file\_ext\_str, and \l\_file\_name\_str.)

\l\_file\_search\_path\_seq    The current search path.

13397 \seq\_new:N \l\_file\_search\_path\_seq

(End definition for \l\_file\_search\_path\_seq. This variable is documented on page 166.)

\l\_file\_tmp\_seq    Scratch space for comma list conversion.

13398 \seq\_new:N \l\_file\_tmp\_seq

(End definition for \l\_file\_tmp\_seq.)

#### 20.4.1 Internal auxiliaries

\s\_file\_stop    Internal scan marks.

13399 \scan\_new:N \s\_file\_stop

(End definition for \s\_file\_stop.)

\q\_file\_nil    Internal quarks.

13400 \quark\_new:N \q\_file\_nil

(End definition for \q\_file\_nil.)

\\_\_file\_quark\_if\_nil\_p:n    Branching quark conditional.

\\_\_file\_quark\_if\_nil:nTF  
13401 \\_\_kernel\_quark\_new\_conditional:Nn \\_\_file\_quark\_if\_nil:n { TF }

(End definition for \\_\_file\_quark\_if\_nil:nTF.)

\q\_file\_recursion\_tail    Internal recursion quarks.

\q\_file\_recursion\_stop  
13402 \quark\_new:N \q\_file\_recursion\_tail  
13403 \quark\_new:N \q\_file\_recursion\_stop

(End definition for \q\_file\_recursion\_tail and \q\_file\_recursion\_stop.)

\\_\_file\_if\_recursion\_tail\_break:NN    Functions to query recursion quarks.

\\_\_file\_if\_recursion\_tail\_stop\_do:Nn  
13404 \\_\_kernel\_quark\_new\_test:N \\_\_file\_if\_recursion\_tail\_stop:N  
13405 \\_\_kernel\_quark\_new\_test:N \\_\_file\_if\_recursion\_tail\_stop\_do:nn

(End definition for \\_\_file\_if\_recursion\_tail\_break:NN and \\_\_file\_if\_recursion\_tail\_stop\_do:Nn.)

```

 __kernel_file_name_sanitize:n
 __kernel_file_name_expand_loop:w
__kernel_file_name_expand_N_type:Nw
__kernel_file_name_expand_group:nw
__kernel_file_name_expand_space:w
__kernel_file_name_strip_quotes:n
__kernel_file_name_strip_quotes:nnw
__kernel_file_name_strip_quotes:nnn
 __kernel_file_name_trim_spaces:n
 __kernel_file_name_trim_spaces:nw
__kernel_file_name_trim_spaces_aux:n
__kernel_file_name_trim_spaces_aux:w

```

Expanding the file name without expanding active characters is done using the same token-by-token approach as for example case changing. The finale outcome only need be e-type expandable, so there is no need for the shuffling that is seen in other locations.

```

13406 \cs_new:Npn __kernel_file_name_sanitize:n #1
13407 {
13408 \exp_args:Ne __kernel_file_name_trim_spaces:n
13409 {
13410 \exp_args:Ne __kernel_file_name_strip_quotes:n
13411 {
13412 __kernel_file_name_expand_loop:w #1
13413 \q_file_recursion_tail \q_file_recursion_stop
13414 }
13415 }
13416 }
13417 \cs_new:Npn __kernel_file_name_expand_loop:w #1 \q_file_recursion_stop
13418 {
13419 \tl_if_head_is_N_type:nTF {#1}
13420 {
13421 __kernel_file_name_expand_N_type:Nw
13422 }
13423 \tl_if_head_is_group:nTF {#1}
13424 {
13425 __kernel_file_name_expand_group:nw
13426 __kernel_file_name_expand_space:w
13427 }
13428 #1 \q_file_recursion_stop
13429 }
13430 \cs_new:Npn __kernel_file_name_expand_N_type:Nw #1
13431 {
13432 __file_if_recursion_tail_stop:N #1
13433 \bool_lazy_and:nnTF
13434 {
13435 \token_if_expandable_p:N #1
13436 }
13437 \bool_not_p:n
13438 \bool_lazy_any_p:n
13439 {
13440 \token_if_protected_macro_p:N #1
13441 \token_if_protected_long_macro_p:N #1
13442 \token_if_active_p:N #1
13443 }
13444 \exp_after:wN __kernel_file_name_expand_loop:w #1
13445 }
13446 \token_to_str:N #1
13447 __kernel_file_name_expand_loop:w
13448 }
13449 }
13450 \cs_new:Npx __kernel_file_name_expand_group:nw #1
13451 {
13452 \c_left_brace_str
13453 \exp_not:N __kernel_file_name_expand_loop:w
13454 #
13455 \c_right_brace_str
13456 }

```

```

13457 \exp_last_unbraced:NNo
13458 \cs_new:Npx __kernel_file_name_expand_space:w \c_space_tl
13459 {
13460 \c_space_tl
13461 \exp_not:N __kernel_file_name_expand_loop:w
13462 }

```

Quoting file name uses basically the same approach as for `luaquotejobname`: count the " tokens and remove them.

```

13463 \cs_new:Npn __kernel_file_name_strip_quotes:n #1
13464 {
13465 __kernel_file_name_strip_quotes:nnnw {#1} { 0 } { }
13466 #1 " \q_file_recursion_tail " \q_file_recursion_stop
13467 }
13468 \cs_new:Npn __kernel_file_name_strip_quotes:nnnw #1#2#3#4 "
13469 {
13470 __file_if_recursion_tail_stop_do:nn {#4}
13471 { __kernel_file_name_strip_quotes:nnn {#1} {#2} {#3} }
13472 __kernel_file_name_strip_quotes:nnnw {#1} { #2 + 1 } { #3#4 }
13473 }
13474 \cs_new:Npn __kernel_file_name_strip_quotes:nnn #1#2#3
13475 {
13476 \int_if_even:nT {#2}
13477 {
13478 __kernel_msg_expandable_error:nnn
13479 { kernel } { unbalanced-quote-in-filename } {#1}
13480 }
13481 #3
13482 }

```

Spaces need to be trimmed from the start of the name and from the end of any extension. However, the name we are passed might not have an extension: that means we have to look for one. If there is no extension, we still use the standard trimming function but deliberately prevent any spaces being removed at the end.

```

13483 \cs_new:Npn __kernel_file_name_trim_spaces:n #1
13484 { __kernel_file_name_trim_spaces:nw {#1} #1 . \q_file_nil . \s_file_stop }
13485 \cs_new:Npn __kernel_file_name_trim_spaces:nw #1#2 . #3 . #4 \s_file_stop
13486 {
13487 __file_quark_if_nil:nTF {#3}
13488 {
13489 \exp_args:Ne __kernel_file_name_trim_spaces_aux:n
13490 { \tl_trim_spaces:n {#1} \s_file_stop }
13491 }
13492 { \tl_trim_spaces:n {#1} }
13493 }
13494 \cs_new:Npn __kernel_file_name_trim_spaces_aux:n #1
13495 { __kernel_file_name_trim_spaces_aux:w #1 }
13496 \cs_new:Npn __kernel_file_name_trim_spaces_aux:w #1 \s_file_stop {#1}

```

(End definition for `\__kernel_file_name_sanitize:n` and others.)

```

__kernel_file_name_quote:n
__kernel_file_name_quote:nw
13497 \cs_new:Npn __kernel_file_name_quote:n #1
13498 { __kernel_file_name_quote:nw {#1} #1 ~ \q_file_nil \s_file_stop }
13499 \cs_new:Npn __kernel_file_name_quote:nw #1 #2 ~ #3 \s_file_stop

```

```

13500 {
13501 __file_quark_if_nil:nTF {#3}
13502 { #1 }
13503 { "#1" }
13504 }

(End definition for __kernel_file_name_quote:n and __kernel_file_name_quote:nw.)
```

\c\_\_file\_marker\_tl The same idea as the marker for rescanning token lists: this pair of tokens cannot appear in a file that is being input.

```

13505 \tl_const:Nx \c__file_marker_tl { : \token_to_str:N : }

(End definition for \c__file_marker_tl.)
```

\file\_get:nnNTF  
\file\_get:nnN  
\\_\_file\_get\_aux:nnN  
\\_\_file\_get\_do:Nw The approach here is similar to that for \tl\_set\_rescan:Nnn. The file contents are grabbed as an argument delimited by \c\_\_file\_marker\_tl. A few subtleties: braces in \if\_false: ... \fi: to deal with possible alignment tabs, \tex\_tracingnesting to avoid a warning about a group being closed inside the \scantokens, and \prg\_return\_true: is placed after the end-of-file marker.

```

13506 \cs_new_protected:Npn \file_get:nnN #1#2#3
13507 {
13508 \file_get:nnNF {#1} {#2} #3
13509 { \tl_set:Nn #3 { \q_no_value } }
13510 }
13511 \prg_new_protected_conditional:Npnn \file_get:nnN #1#2#3 { T , F , TF }
13512 {
13513 \file_get_full_name:nNTF {#1} \l__file_full_name_tl
13514 {
13515 \exp_args:NV __file_get_aux:nnN
13516 \l__file_full_name_tl
13517 {#2} #3
13518 \prg_return_true:
13519 }
13520 { \prg_return_false: }
13521 }
13522 \cs_new_protected:Npx __file_get_aux:nnN #1#2#3
13523 {
13524 \exp_not:N \if_false: { \exp_not:N \fi:
13525 \group_begin:
13526 \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
13527 \exp_not:N \exp_args:No \tex_everyeof:D
13528 { \exp_not:N \c__file_marker_tl }
13529 #2 \scan_stop:
13530 \exp_not:N \exp_after:wN \exp_not:N __file_get_do:Nw
13531 \exp_not:N \exp_after:wN #3
13532 \exp_not:N \exp_after:wN \exp_not:N \prg_do_nothing:
13533 \exp_not:N \tex_input:D
13534 \sys_if_engine_luatex:TF
13535 { {#1} }
13536 { \exp_not:N __kernel_file_name_quote:n {#1} \scan_stop: }
13537 \exp_not:N \if_false: } \exp_not:N \fi:
13538 }
13539 \exp_args:Nno \use:nn
13540 { \cs_new_protected:Npn __file_get_do:Nw #1#2 }
```

```

13541 { \c__file_marker_t1 }
13542 {
13543 \group_end:
13544 \tl_set:No #1 {#2}
13545 }
```

(End definition for `\file_get:nnNTF` and others. These functions are documented on page 166.)

`\__file_size:n` A copy of the primitive where it's available.

```
13546 \cs_new_eq:NN __file_size:n \tex filesize:D
```

(End definition for `\__file_size:n`.)

```

\file_full_name:n
__file_full_name:n
__file_full_name_aux:Nn
__file_full_name_aux:nN
__file_name_cleanup:w
 __file_name_end:
__file_name_ext_check:n
__file_name_ext_check:nw
__file_name_ext_check:nw
__file_name_ext_check:nn
__file_name_ext_check:nn
```

File searching can be carried out if the `\pdffilesize` primitive or an equivalent is available. That of course means we need to arrange for everything else to here to be done by expansion too. We start off by sanitizing the name and quoting if required: we may need to remove those quotes, so the raw name is passed too.

```

13547 \cs_new:Npn \file_full_name:n #1
13548 {
13549 \exp_args:Ne __file_full_name:n
13550 { __kernel_file_name_sanitize:n {#1} }
13551 }
```

First, we check if the file is just here: no mapping so we do not need the break part of the broader auxiliary. We are using the fact that the primitive here returns nothing if the file is entirely absent. For package mode, `\input@path` is a token list not a sequence.

```

13552 \cs_new:Npn __file_full_name:n #1
13553 {
13554 \tl_if_blank:nF {#1}
13555 {
13556 \tl_if_blank:eTF { __file_size:n {#1} }
13557 {
13558 \seq_map_tokens:Nn \l_file_search_path_seq
13559 { __file_full_name_aux:Nnn \seq_map_break:n {#1} }
13560 \cs_if_exist:NT \input@path
13561 {
13562 \tl_map_tokens:Nn \input@path
13563 { __file_full_name_aux:Nnn \tl_map_break:n {#1} }
13564 }
13565 __file_name_end:
13566 }
13567 { __file_ext_check:n {#1} }
13568 }
13569 }
```

Two pars to the auxiliary here so we can avoid doing quoting twice in the event we find the right file.

```

13570 \cs_new:Npn __file_full_name_aux:Nnn #1#2#3
13571 { \exp_args:Ne __file_full_name_aux:nN { \tl_to_str:n {#3} / #2 } #1 }
13572 \cs_new:Npn __file_full_name_aux:nN #1 #2
13573 {
13574 \tl_if_blank:eF { __file_size:n {#1} }
13575 {
13576 #2
13577 }
```

```

13578 __file_ext_check:n {#1}
13579 __file_name_cleanup:w
13580 }
13581 }
13582 }
13583 \cs_new:Npn __file_name_cleanup:w #1 __file_name_end: { }
13584 \cs_new:Npn __file_name_end: { }

```

As TeX automatically adds `.tex` if there is no extension, there is a little clean up to do here. First, make sure we are not in the directory part, saving that. Then check for an extension.

```

13585 \cs_new:Npn __file_ext_check:n #1
13586 { __file_ext_check:nw { / } #1 / \q_file_nil / \s_file_stop }
13587 \cs_new:Npn __file_ext_check:nw #1 #2 / #3 / #4 \s_file_stop
13588 {
13589 __file_quark_if_nil:nTF {#3}
13590 {
13591 \exp_args:No __file_ext_check:nnw
13592 { \use_none:n #1 } {#2} #2 . \q_file_nil . \s_file_stop
13593 }
13594 { __file_ext_check:nw { #1 #2 / } #3 / #4 \s_file_stop }
13595 }
13596 \cs_new:Npx __file_ext_check:nnw #1#2#3 . #4 . #5 \s_file_stop
13597 {
13598 \exp_not:N __file_quark_if_nil:nTF {#4}
13599 {
13600 \exp_not:N __file_ext_check:nn
13601 { #1 #2 } { #1 #2 \tl_to_str:n { .tex } }
13602 }
13603 { #1 #2 }
13604 }
13605 \cs_new:Npn __file_ext_check:nn #1#2
13606 {
13607 \tl_if_blank:eTF { __file_size:n {#2} }
13608 {#1}
13609 {
13610 \int_compare:nNnTF
13611 { __file_size:n {#1} } = { __file_size:n {#2} }
13612 {#2}
13613 {#1}
13614 }
13615 }

```

Deal with the fact that the primitive might not be available.

```

13616 \cs_if_exist:NF \tex filesize:D
13617 {
13618 \cs_gset:Npn \file_full_name:n #1
13619 {
13620 __kernel_msg_expandable_error:nnn
13621 { kernel } { primitive-not-available }
13622 { \pdffilesize }
13623 }
13624 }
13625 __kernel_msg_new:nnnn { kernel } { primitive-not-available }
13626 { Primitive~\token_to_str:N #1 not-available }

```

```

13627 {
13628 The~version~of~your~TeX~engine~does~not~provide~functionality~equivalent~to~
13629 the~#1~primitive.
13630 }

```

(End definition for `\file_full_name:n` and others. This function is documented on page 166.)

**\file\_get\_full\_name:nN** These functions pre-date using `\tex_filesize:D` for file searching, so are `get` functions with protection. To avoid having different search set ups, they are simply wrappers around the code above.

```

13631 \cs_new_protected:Npn \file_get_full_name:nN #1#2
13632 {
13633 \file_get_full_name:nNF {#1} #2
13634 { \tl_set:Nn #2 { \q_no_value } }
13635 }
13636 \cs_generate_variant:Nn \file_get_full_name:nN { V }
13637 \prg_new_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
13638 {
13639 __kernel_tl_set:Nx #2
13640 { \file_full_name:n {#1} }
13641 \tl_if_empty:NTF #2
13642 { \prg_return_false: }
13643 { \prg_return_true: }
13644 }
13645 \cs_generate_variant:Nn \file_get_full_name:nNT { V }
13646 \cs_generate_variant:Nn \file_get_full_name:nNF { V }
13647 \cs_generate_variant:Nn \file_get_full_name:nNTF { V }

```

If `\tex_filesize:D` is not available, the way to test if a file exists is to try to open it: if it does not exist then TeX reports end-of-file. A search is made looking at each potential path in turn (starting from the current directory). The first location is of course treated as the correct one: this is done by jumping to `\prg_break_point:`. If nothing is found, #2 is returned empty. A special case when there is no extension is that once the first location is found we test the existence of the file with `.tex` extension in that directory, and if it exists we include the `.tex` extension in the result.

```

13648 \cs_if_exist:NF \tex_filesize:D
13649 {
13650 \prg_set_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
13651 {
13652 __kernel_tl_set:Nx \l__file_base_name_tl
13653 { __kernel_file_name_sanitize:n {#1} }
13654 __file_get_full_name_search:nN { } \use:n
13655 \seq_map_inline:Nn \l_file_search_path_seq
13656 { __file_get_full_name_search:nN { ##1 / } \seq_map_break:n }
13657 \cs_if_exist:NT \input@path
13658 {
13659 \tl_map_inline:Nn \input@path
13660 { __file_get_full_name_search:nN { ##1 } \tl_map_break:n }
13661 }
13662 \tl_set:Nn \l__file_full_name_tl { \q_no_value }
13663 \prg_break_point:
13664 \quark_if_no_value:NTF \l__file_full_name_tl
13665 {
13666 \ior_close:N \g__file_internal_ior

```

```

13667 \prg_return_false:
13668 }
13669 {
13670 \file_parse_full_name:VNNN \l__file_full_name_tl
13671 \l__file_dir_str \l__file_name_str \l__file_ext_str
13672 \str_if_empty:NT \l__file_ext_str
13673 {
13674 __kernel_ior_open:No \g__file_internal_ior
13675 { \l__file_full_name_tl .tex }
13676 \ior_if_eof:NF \g__file_internal_ior
13677 { \tl_put_right:Nn \l__file_full_name_tl { .tex } }
13678 }
13679 \ior_close:N \g__file_internal_ior
13680 \tl_set_eq:NN #2 \l__file_full_name_tl
13681 \prg_return_true:
13682 }
13683 }
13684 }
13685 \cs_new_protected:Npn __file_get_full_name_search:nN #1#2
13686 {
13687 __kernel_tl_set:Nx \l__file_full_name_tl
13688 { \tl_to_str:n {#1} \l__file_base_name_tl }
13689 __kernel_ior_open:No \g__file_internal_ior \l__file_full_name_tl
13690 \ior_if_eof:NF \g__file_internal_ior { #2 { \prg_break: } }
13691 }

```

(End definition for `\file_get_full_name:nN`, `\file_get_full_name:nNTF`, and `\__file_get_full_name_search:nN`. These functions are documented on page 166.)

`\g__file_internal_ior` A reserved stream to test for file existence (if required), and for opening a shell.

```
13692 \ior_new:N \g__file_internal_ior
```

(End definition for `\g__file_internal_ior`.)

`\file_mdfive_hash:n` Getting file details by expansion is relatively easy if a bit repetitive. As the MD5 function has a slightly different syntax from the other commands, there is a little cleaning up to do.

```

13693 \cs_new:Npn \file_size:n #1
13694 { __file_details:nn {#1} { size } }
13695 \cs_new:Npn \file_timestamp:n #1
13696 { __file_details:nn {#1} { moddate } }
13697 \cs_new:Npn __file_details:nn #1#2
13698 {
13699 \exp_args:Ne __file_details_aux:nn
13700 { \file_full_name:n {#1} } {#2}
13701 }
13702 \cs_new:Npn __file_details_aux:nn #1#2
13703 {
13704 \tl_if_blank:nF {#1}
13705 { \use:c { tex_file #2 :D } {#1} }
13706 }
13707 \cs_new:Npn \file_mdfive_hash:n #1
13708 { \exp_args:Ne __file_mdfive_hash:n { \file_full_name:n {#1} } }
13709 \cs_new:Npn __file_mdfive_hash:n #1
13710 { \tex_mdfivesum:D file {#1} }

```

(End definition for `\file_mdfive_hash:n` and others. These functions are documented on page 168.)

These are separate as they need multiple arguments *or* the file size. For LuaTeX, the emulation does not need the file size so we save a little on expansion.

```
13711 \cs_new:Npn \file_hex_dump:n {#1} {#2} {#3}
13712 {
13713 \exp_args:Neee __file_hex_dump_auxi:nnn
13714 { \file_full_name:n {#1} }
13715 { \int_eval:n {#2} }
13716 { \int_eval:n {#3} }
13717 }
13718 \cs_new:Npn __file_hex_dump_auxi:nnn {#1} {#2} {#3}
13719 {
13720 \bool_lazy_any:nF
13721 {
13722 { \tl_if_blank_p:n {#1} }
13723 { \int_compare_p:nNn {#2} = 0 }
13724 { \int_compare_p:nNn {#3} = 0 }
13725 }
13726 {
13727 \exp_args:Ne __file_hex_dump_auxii:nnnn
13728 { __file_details_aux:nn {#1} { size } }
13729 {#1} {#2} {#3}
13730 }
13731 }
13732 \cs_new:Npn __file_hex_dump_auxii:nnnn {#1} {#2} {#3} {#4}
13733 {
13734 \int_compare:nNnTF {#3} > 0
13735 { __file_hex_dump_auxiii:nnnn {#3} }
13736 {
13737 \exp_args:Ne __file_hex_dump_auxiii:nnnn
13738 { \int_eval:n { #1 + #3 } }
13739 }
13740 {#1} {#2} {#4}
13741 }
13742 \cs_new:Npn __file_hex_dump_auxiii:nnnn {#1} {#2} {#3} {#4}
13743 {
13744 \int_compare:nNnTF {#4} > 0
13745 { __file_hex_dump_auxiv:nnn {#4} }
13746 {
13747 \exp_args:Ne __file_hex_dump_auxiv:nnn
13748 { \int_eval:n { #2 + #4 } }
13749 }
13750 {#1} {#3}
13751 }
13752 \cs_new:Npn __file_hex_dump_auxiv:nnn {#1} {#2} {#3}
13753 {
13754 \tex_filedump:D
13755 offset ~ \int_eval:n { #2 - 1 } ~
13756 length ~ \int_eval:n { #1 - #2 + 1 }
13757 {#3}
13758 }
13759 \cs_new:Npn \file_hex_dump:n {#1}
13760 { \exp_args:Ne __file_hex_dump:n { \file_full_name:n {#1} } }
```

```
13761 \sys_if_engine_luatex:TF
13762 {
13763 \cs_new:Npn __file_hex_dump:n #1
13764 {
13765 \tl_if_blank:nF {#1}
13766 { \tex_filedump:D whole {#1} {#1} }
13767 }
13768 }
13769 {
13770 \cs_new:Npn __file_hex_dump:n #1
13771 {
13772 \tl_if_blank:nF {#1}
13773 { \tex_filedump:D length \tex_filesize:D {#1} {#1} }
13774 }
13775 }
```

(End definition for `\file_hex_dump:nnn` and others. These functions are documented on page 167.)

Non-expandable wrappers around the above in the case where appropriate primitive support exists.

```
13776 \cs_new_protected:Npn \file_get_hex_dump:nN #1#2
13777 { \file_get_hex_dump:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
13778 \cs_new_protected:Npn \file_get_mdfive_hash:nN #1#2
13779 { \file_get_mdfive_hash:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
13780 \cs_new_protected:Npn \file_get_size:nN #1#2
13781 { \file_get_size:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
13782 \cs_new_protected:Npn \file_get_timestamp:nN #1#2
13783 { \file_get_timestamp:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
13784 \prg_new_protected_conditional:Npnn \file_get_hex_dump:nN #1#2 { T , F , TF }
13785 { __file_get_details:nnN {#1} { hex_dump } #2 }
13786 \prg_new_protected_conditional:Npnn \file_get_mdfive_hash:nN #1#2 { T , F , TF }
13787 { __file_get_details:nnN {#1} { mdfive_hash } #2 }
13788 \prg_new_protected_conditional:Npnn \file_get_size:nN #1#2 { T , F , TF }
13789 { __file_get_details:nnN {#1} { size } #2 }
13790 \prg_new_protected_conditional:Npnn \file_get_timestamp:nN #1#2 { T , F , TF }
13791 { __file_get_details:nnN {#1} { timestamp } #2 }
13792 \cs_new_protected:Npn __file_get_details:nnN #1#2#3
13793 {
13794 __kernel_tl_set:Nx #3
13795 { \use:c { file_ #2 :n } {#1} }
13796 \tl_if_empty:NTF #3
13797 { \prg_return_false: }
13798 { \prg_return_true: }
13799 }
```

Where the primitive is not available, issue an error: this is a little more conservative than absolutely needed, but does work.

```
13800 \cs_if_exist:NF \tex_filesize:D
13801 {
13802 \cs_set_protected:Npn __file_get_details:nnN #1#2#3
13803 {
13804 \tl_clear:N #3
13805 __kernel_msg_error:nnx
13806 { kernel } { primitive-not-available }
13807 {
```

```

13808 \token_to_str:N \pdf{file}
13809 \str_case:nn {#2}
13810 {
13811 { hex_dump } { dump }
13812 { mdfive_hash } { md5esum }
13813 { timestamp } { moddate }
13814 { size } { size }
13815 }
13816 }
13817 \prg_return_false:
13818 }
13819 }
```

(End definition for `\file_get_hex_dump:nNTF` and others. These functions are documented on page 167.)

`\file_get_hex_dump:nnN`

`\file_get_hex_dump:nnnNTF`

Custom code due to the additional arguments.

```

13820 \cs_new_protected:Npn \file_get_hex_dump:nnnN #1#2#3#4
13821 {
13822 \file_get_hex_dump:nnNF {#1} {#2} {#3} #4
13823 { \tl_set:Nn #4 { \q_no_value } }
13824 }
13825 \prg_new_protected_conditional:Npnn \file_get_hex_dump:nnN #1#2#3#4
13826 { T , F , TF }
13827 {
13828 __kernel_tl_set:Nx #4
13829 { \file_hex_dump:nnn {#1} {#2} {#3} }
13830 \tl_if_empty:NTF #4
13831 { \prg_return_false: }
13832 { \prg_return_true: }
13833 }
```

(End definition for `\file_get_hex_dump:nnnNTF`. This function is documented on page 167.)

`\__file_str_cmp:nn`

As we are doing a fixed-length “big” integer comparison, it is easiest to use the low-level behavior of string comparisons.

```
13834 \cs_new_eq:NN __file_str_cmp:nn \tex_strcmp:D
```

(End definition for `\__file_str_cmp:nn`.)

`\file_compare_timestamp:p:nNn`

`\file_compare_timestamp:nNnTF`

Comparison of file date can be done by using the low-level nature of the string comparison functions.

```

13835 \prg_new_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
13836 { p , T , F , TF }
13837 {
13838 \exp_args:Nee __file_compare_timestamp:nnN
13839 { \file_full_name:n {#1} }
13840 { \file_full_name:n {#3} }
13841 #2
13842 }
13843 \cs_new:Npn __file_compare_timestamp:nnN #1#2#3
13844 {
13845 \tl_if_blank:nTF {#1}
13846 {
13847 \if_charcode:w #3 <
```

```

13848 \prg_return_true:
13849
13850 \else:
13851 \prg_return_false:
13852 \fi:
13853 }
13854 {
13855 \tl_if_blank:nTF {#2}
13856 {
13857 \if_charcode:w #3 >
13858 \prg_return_true:
13859 \else:
13860 \prg_return_false:
13861 \fi:
13862 }
13863 \if_int_compare:w
13864 __file_str_cmp:nn
13865 { __file_timestamp:n {#1} }
13866 { __file_timestamp:n {#2} }
13867 #3 0 \exp_stop_f:
13868 \prg_return_true:
13869 \else:
13870 \prg_return_false:
13871 \fi:
13872 }
13873 }
13874 }
13875 \cs_new_eq:NN __file_timestamp:n \tex_filemoddate:D
13876 \cs_if_exist:NF __file_timestamp:n
13877 {
13878 \prg_set_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
13879 { p , T , F , TF }
13880 {
13881 __kernel_msg_expandable_error:nnn
13882 { kernel } { primitive-not-available }
13883 { \pdf_filemoddate }
13884 \prg_return_false:
13885 }
13886 }

```

(End definition for `\file_compare_timestamp:nNnTF`, `\__file_compare_timestamp:nnN`, and `\__file_timestamp:n`. This function is documented on page 169.)

**\file\_if\_exist:nTF** The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path contains something, whereas if the file was not located then the return value is empty.

```

13887 \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
13888 {
13889 \file_get_full_name:nNTF {#1} \l__file_full_name_tl
13890 { \prg_return_true: }
13891 { \prg_return_false: }
13892 }

```

(End definition for `\file_if_exist:nTF`. This function is documented on page 166.)

`\file_if_exist_input:n` Input of a file with a test for existence. We do not define the T or TF variants because the most useful place to place the *<true code>* would be inconsistent with other conditionals.

```

13893 \cs_new_protected:Npn \file_if_exist_input:n #1
13894 {
13895 \file_get_full_name:nNT {#1} \l__file_full_name_tl
13896 { __file_input:V \l__file_full_name_tl }
13897 }
13898 \cs_new_protected:Npn \file_if_exist_input:nF #1#2
13899 {
13900 \file_get_full_name:nNTF {#1} \l__file_full_name_tl
13901 { __file_input:V \l__file_full_name_tl }
13902 {#2}
13903 }
```

(End definition for `\file_if_exist_input:n` and `\file_if_exist_input:nF`. These functions are documented on page 169.)

`\file_input_stop:` A simple rename.

```
13904 \cs_new_protected:Npn \file_input_stop: { \tex_endinput:D }
```

(End definition for `\file_input_stop:`. This function is documented on page 169.)

`\__kernel_file_missing:n` An error message for a missing file, also used in `\ior_open:Nn`.

```

13905 \cs_new_protected:Npn __kernel_file_missing:n #1
13906 {
13907 __kernel_msg_error:nnx { kernel } { file-not-found }
13908 { __kernel_file_name_sanitize:n {#1} }
13909 }
```

(End definition for `\__kernel_file_missing:n`.)

`\file_input:n`  
`\__file_input:n`  
`\__file_input:V`  
`\__file_input_push:n`  
`\__kernel_file_input_push:n`  
`\__file_input_pop:`  
`\__kernel_file_input_pop:`  
`\__file_input_pop:nnn`

```

13910 \cs_new_protected:Npn \file_input:n #1
13911 {
13912 \file_get_full_name:nNTF {#1} \l__file_full_name_tl
13913 { __file_input:V \l__file_full_name_tl }
13914 { __kernel_file_missing:n {#1} }
13915 }
13916 \cs_new_protected:Npx __file_input:n #1
13917 {
13918 \exp_not:N \clist_if_exist:NTF \exp_not:N \@filelist
13919 { \exp_not:N \addtofilelist {#1} }
13920 { \seq_gput_right:Nn \exp_not:N \g__file_record_seq {#1} }
13921 \exp_not:N __file_input_push:n {#1}
13922 \exp_not:N \tex_input:D
13923 \sys_if_engine_luatex:TF
13924 { {#1} }
13925 { \exp_not:N __kernel_file_name_quote:n {#1} \scan_stop: }
13926 \exp_not:N __file_input_pop:
13927 }
13928 \cs_generate_variant:Nn __file_input:n { V }
```

Keeping a track of the file data is easy enough: we store the separated parts so we do not need to parse them twice.

```

13929 \cs_new_protected:Npn __file_input_push:n #1
13930 {
13931 \seq_gpush:Nx \g__file_stack_seq
13932 {
13933 { \g_file_curr_dir_str }
13934 { \g_file_curr_name_str }
13935 { \g_file_curr_ext_str }
13936 }
13937 \file_parse_full_name:nNNN {#1}
13938 \l__file_dir_str \l__file_name_str \l__file_ext_str
13939 \str_gset_eq:NN \g_file_curr_dir_str \l__file_dir_str
13940 \str_gset_eq:NN \g_file_curr_name_str \l__file_name_str
13941 \str_gset_eq:NN \g_file_curr_ext_str \l__file_ext_str
13942 }
13943 \cs_new_eq:NN __kernel_file_input_push:n __file_input_push:n
13944 \cs_new_protected:Npn __file_input_pop:
13945 {
13946 \seq_gpop>NN \g__file_stack_seq \l__file_internal_tl
13947 \exp_after:wN __file_input_pop:nnn \l__file_internal_tl
13948 }
13949 \cs_new_eq:NN __kernel_file_input_pop: __file_input_pop:
13950 \cs_new_protected:Npn __file_input_pop:nnn #1#2#3
13951 {
13952 \str_gset:Nn \g_file_curr_dir_str {#1}
13953 \str_gset:Nn \g_file_curr_name_str {#2}
13954 \str_gset:Nn \g_file_curr_ext_str {#3}
13955 }

```

(End definition for `\file_input:n` and others. This function is documented on page 169.)

`\file_parse_full_name:n`  
`\file_parse_full_name_apply:nN`

The main parsing macro `\file_parse_full_name_apply:nN` passes the file name #1 through `\__kernel_file_name_sanitize:n` so that we have a single normalised way to treat files internally. `\file_parse_full_name:n` uses the former, with `\prg_do_nothing:` to leave each part of the name within a pair of braces.

```

13956 \cs_new:Npn \file_parse_full_name:n #1
13957 {
13958 \file_parse_full_name_apply:nN {#1}
13959 \prg_do_nothing:
13960 }
13961 \cs_new:Npn \file_parse_full_name_apply:nN #1
13962 {
13963 \exp_args:Ne __file_parse_full_name_auxi:nN
13964 { __kernel_file_name_sanitize:n {#1} }
13965 }

```

`\__file_parse_full_name_auxi:nN`  
`\__file_parse_full_name_area:nw`

`\__file_parse_full_name_area:nw` splits the file name into chunks separated by /, until the last one is reached. The last chunk is the file name plus the extension, and everything before that is the path. When `\__file_parse_full_name_area:nw` is done, it leaves the path within braces after the scan mark `\s__file_stop` and proceeds parsing the actual file name.

```
13966 \cs_new:Npn __file_parse_full_name_auxi:nN #1
```

```

13967 {
13968 __file_parse_full_name_area:nw { } #1
13969 / \s__file_stop
13970 }
13971 \cs_new:Npn __file_parse_full_name_area:nw #1 #2 / #3 \s__file_stop
13972 {
13973 \tl_if_empty:nTF {#3}
13974 { __file_parse_full_name_base:nw { } #2 . \s__file_stop {#1} }
13975 { __file_parse_full_name_area:nw { #1 / #2 } #3 \s__file_stop }
13976 }

__file_parse_full_name_base:nw does roughly the same as above, but it separates
the chunks at each period. However here there's some extra complications: In case #1
is empty, it is assumed that the extension is actually empty, and the file name is #2.
Besides, an extra . has to be added to #2 because it is later removed in __file-
parse_full_name_tidy:nnnN. In any case, if there's an extension, it is returned with a
leading ..

13977 \cs_new:Npn __file_parse_full_name_base:nw #1 #2 . #3 \s__file_stop
13978 {
13979 \tl_if_empty:nTF {#3}
13980 {
13981 \tl_if_empty:nTF {#1}
13982 {
13983 \tl_if_empty:nTF {#2}
13984 { __file_parse_full_name_tidy:nnnN { } { } }
13985 { __file_parse_full_name_tidy:nnnN { .#2 } { } }
13986 }
13987 { __file_parse_full_name_tidy:nnnN {#1} { .#2 } }
13988 }
13989 { __file_parse_full_name_base:nw { #1 . #2 } #3 \s__file_stop }
13990 }

Now we just need to tidy some bits left loose before. The loop used in the two macros
above start with a leading / and . in the file path an name, so here we need to remove
them, except in the path, if it is a single /, in which case it's left as is. After all's done,
pass to #4.
```

```

13991 \cs_new:Npn __file_parse_full_name_tidy:nnnN #1 #2 #3 #4
13992 {
13993 \exp_args:Nee #4
13994 {
13995 \str_if_eq:nnF {#3} { / } { \use_none:n }
13996 #3 \prg_do_nothing:
13997 }
13998 { \use_none:n #1 \prg_do_nothing: }
13999 {#2}
14000 }

(End definition for \file_parse_full_name:n and others. These functions are documented on page
167.)
```

**\file\_parse\_full\_name:nNNN**  
**\file\_parse\_full\_name:VNNN**

```

14001 \cs_new_protected:Npn \file_parse_full_name:nNNN #1 #2 #3 #4
14002 {
```

```

14003 \file_parse_full_name_apply:nN {#1}
14004 _file_full_name_assign:nnnNNN #2 #3 #4
14005 }
14006 \cs_new_protected:Npn _file_full_name_assign:nnnNNN #1 #2 #3 #4 #5 #6
14007 {
14008 \str_set:Nn #4 {#1}
14009 \str_set:Nn #5 {#2}
14010 \str_set:Nn #6 {#3}
14011 }
14012 \cs_generate_variant:Nn \file_parse_full_name:nNNN { V }

```

(End definition for `\file_parse_full_name:nNNN`. This function is documented on page 167.)

`\file_show_list:` A function to list all files used to the log, without duplicates. In package mode, if `\@filelist` is still defined, we need to take this list of file names into account (we capture it `\AtBeginDocument` into `\g_file_record_seq`), turning it to a string (this does not affect the commas of this comma list).

```

14013 \cs_new_protected:Npn \file_show_list: { _file_list:N \msg_show:nnxxxx }
14014 \cs_new_protected:Npn \file_log_list: { _file_list:N \msg_log:nnxxxx }
14015 \cs_new_protected:Npn _file_list:N #1
14016 {
14017 \seq_clear:N \l_file_tmp_seq
14018 \clist_if_exist:NT \@filelist
14019 {
14020 \exp_args:NNx \seq_set_from_clist:Nn \l_file_tmp_seq
14021 { \tl_to_str:N \@filelist }
14022 }
14023 \seq_concat:NNN \l_file_tmp_seq \l_file_tmp_seq \g_file_record_seq
14024 \seq_remove_duplicates:N \l_file_tmp_seq
14025 #1 { LaTeX/kernel } { file-list }
14026 { \seq_map_function:NN \l_file_tmp_seq _file_list_aux:n }
14027 { } { } { }
14028 }
14029 \cs_new:Npn _file_list_aux:n #1 { \iow_newline: #1 }

```

(End definition for `\file_show_list:` and others. These functions are documented on page 169.)

When used as a package, there is a need to hold onto the standard file list as well as the new one here. File names recorded in `\@filelist` must be turned to strings before being added to `\g_file_record_seq`.

```

14030 \cs_if_exist:NT \@filelist
14031 {
14032 \AtBeginDocument
14033 {
14034 \exp_args:NNx \seq_set_from_clist:Nn \l_file_tmp_seq
14035 { \tl_to_str:N \@filelist }
14036 \seq_gconcat:NNN
14037 \g_file_record_seq
14038 \g_file_record_seq
14039 \l_file_tmp_seq
14040 }
14041 }

```

## 20.5 GetIdInfo

\GetIdInfo

```
__file_id_info_auxi:w
__file_id_info_auxii:w
__file_id_info_auxiii:w
```

As documented in `expl3.dtx` this function extracts file name etc from an SVN Id line. This used to be how we got version number and so on in all modules, so it had to be defined in `l3bootstrap`. Now it's more convenient to define it after we have set up quite a lot of tools, and `l3file` seems the least unreasonable place for it.

The idea here is to extract out the information needed from a standard SVN Id line, but to avoid a line that would get changed when the file is checked in. Hence the fact that none of the lines here include both a dollar sign and the `Id` keyword!

```
14042 \cs_new_protected:Npn \GetIdInfo
14043 {
14044 \tl_clear_new:N \ExplFileVersion
14045 \tl_clear_new:N \ExplFileName
14046 \tl_clear_new:N \ExplFileExtension
14047 \tl_clear_new:N \ExplFileDate
14048 \tl_clear_new:N \ExplFileVersion
14049 \group_begin:
14050 \char_set_catcode_space:n { 32 }
14051 \exp_after:wN
14052 \group_end:
14053 __file_id_info_auxi:w
14054 }
```

A first check for a completely empty SVN field. If that is not the case, there is a second case when a file created using `svn cp` but has not been checked in. That leaves a special marker `-1` version, which has no further data. Dealing correctly with that is the reason for the space in the line to use `\__file_id_info_auxi:w`.

```
14055 \cs_new_protected:Npn __file_id_info_auxi:w $ #1 $ #2
14056 {
14057 \tl_set:Nn \ExplFileVersion {#2}
14058 \str_if_eq:nnTF {#1} { Id }
14059 {
14060 \tl_set:Nn \ExplFileDate { 0000/00/00 }
14061 \tl_set:Nn \ExplFileName { [unknown] }
14062 \tl_set:Nn \ExplFileExtension { [unknown-extension] }
14063 \tl_set:Nn \ExplFileVersion {-1}
14064 }
14065 { __file_id_info_auxii:w #1 ~ \s_file_stop }
14066 }
```

Here, `#1` is `Id`, `#2` is the file name, `#3` is the extension, `#4` is the version, `#5` is the check in date and `#6` is the check in time and user, plus some trailing spaces. If `#4` is the marker `-1` value then `#5` and `#6` are empty.

```
14067 \cs_new_protected:Npn __file_id_info_auxii:w
14068 #1 ~ #2.#3 ~ #4 ~ #5 ~ #6 \s_file_stop
14069 {
14070 \tl_set:Nn \ExplFileName {#2}
14071 \tl_set:Nn \ExplFileExtension {#3}
14072 \tl_set:Nn \ExplFileVersion {#4}
14073 \str_if_eq:nnTF {#4} {-1}
14074 { \tl_set:Nn \ExplFileVersion { 0000/00/00 } }
14075 { __file_id_info_auxiii:w #5 - 0 - 0 - \s_file_stop }
14076 }
```

Convert an SVN-style date into a L<sup>A</sup>T<sub>E</sub>X-style one.

```
14077 \cs_new_protected:Npn __file_id_info_auxiii:w #1 - #2 - #3 - #4 \s__file_stop
14078 { \tl_set:Nn \ExplFileVersion { #1/#2/#3 } }
```

(End definition for `\GetIdInfo` and others. This function is documented on page 7.)

## 20.6 Checking the version of kernel dependencies

This function is responsible for checking if dependencies of the L<sup>A</sup>T<sub>E</sub>X3 kernel match the version preloaded in the L<sup>A</sup>T<sub>E</sub>X2<sub>ε</sub> kernel. If versions don't match, the function attempts to tell why by searching for a possible stray format file.

The function starts by checking that the kernel date is defined, and if not zero is used to force the error route. The kernel date is then compared with the argument requested date (usually the packaging date of the dependency). If the kernel date is less than the required date, it's an error and the loading should abort.

```
14079 \cs_new_protected:Npn __kernel_dependency_version_check:Nn #1
14080 { \exp_args:NV __kernel_dependency_version_check:nn #1 }
14081 \cs_new_protected:Npn __kernel_dependency_version_check:nn #1
14082 {
14083 \cs_if_exist:NTF \c_kernel_expl_date_tl
14084 {
14085 \exp_args:NV __file_kernel_dependency_compare:nnn
14086 \c_kernel_expl_date_tl {#1}
14087 }
14088 { __file_kernel_dependency_compare:nnn { 0000-00-00 } {#1} }
14089 }
14090 \cs_new_protected:Npn __file_kernel_dependency_compare:nnn #1 #2 #3
14091 {
14092 \int_compare:nNnT
14093 { __file_parse_version:w #1 \s__file_stop } <
14094 { __file_parse_version:w #2 \s__file_stop }
14095 { __file_mismatched_dependency_error:nn {#2} {#3} }
14096 }
14097 \cs_new:Npn __file_parse_version:w #1 - #2 - #3 \s__file_stop {#1#2#3}
```

If the versions differ, then we try to give the user some guidance. This function starts by taking the engine name `\c_sys_engine_str` and replacing `tex` by `latex`, then building a command of the form: `kpsewhich --all --engine=<engine> <format>[-dev].fmt` to query the format files available. A shell is opened and each line is read into a sequence.

```
14098 \cs_new_protected:Npn __file_mismatched_dependency_error:nn #1 #2
14099 {
14100 \exp_args:NNx \ior_shell_open:Nn \g__file_internal_ior
14101 {
14102 kpsewhich ~ --all ~
14103 --engine = \c_sys_engine_exec_str
14104 \c_space_tl \c_sys_engine_format_str
14105 \bool_lazy_and:nnF
14106 { \tl_if_exist_p:N \development@branch@name }
14107 { ! \tl_if_empty_p:N \development@branch@name }
14108 { -dev } .fmt
14109 }
14110 \seq_clear:N \l__file_tmp_seq
14111 \ior_map_inline:Nn \g__file_internal_ior
```

```

14112 { \seq_put_right:Nn \l_file_tmp_seq {##1} }
14113 \ior_close:N \g_file_internal_ior
14114 _kernel_msg_error:nnnn { kernel } { mismatched-support-file }
14115 {##1} {##2}

```

And finish by ending the current file.

```

14116 \tex_endinput:D
14117 }
14118 % \begin{macrocode}
14119 %
14120 % Now define the actual error message:
14121 % \begin{macrocode}
14122 _kernel_msg_new:nnnn { kernel } { mismatched-support-file }
14123 {
14124 Mismatched-LaTeX-support-files-detected. \\
14125 Loading #'#2' aborted!

```

\c\_kernel\_expl\_date\_tl may not exist, due to an older format, so only print the dates when the sentinel token list exists:

```

14126 \tl_if_exist:NT \c_kernel_expl_date_tl
14127 {
14128 \\
14129 The~L3~programming~layer~in~the~LaTeX~format \\
14130 is~dated~\c_kernel_expl_date_tl,~but~in~your~TeX~tree~the~files~require \\
14131 tree~at~least~#1.
14132 }
14133 }
14134 {

```

The sequence containing the format files should have exactly one item: the format file currently being run. If that's the case, the cause of the error is not that, so print a generic help with some possible causes. If more than one format file was found, then print the list to the user, with appropriate indications of what's in the system and what's in the user tree.

```

14135 \int_compare:nNnTF { \seq_count:N \l_file_tmp_seq } > 1
14136 {
14137 The~cause~seems~to~be~an~old~format~file~in~the~user~tree. \\
14138 LaTeX~found~these~files:
14139 \seq_map_tokens:Nn \l_file_tmp_seq { \\---\use:n } \\
14140 Try~deleting~the~file~in~the~user~tree~then~run~LaTeX~again.
14141 }
14142 {
14143 The~most~likely~causes~are:
14144 \\---A~recent~format~generation~failed;
14145 \\---A~stray~format~file~in~the~user~tree~which~needs~
14146 to~be~removed~or~rebuilt;
14147 \\---You~are~running~a~manually~installed~version~of~#2 \\
14148 \\ \\ which~is~incompatible~with~the~version~in~LaTeX. \\
14149 }
14150 \\
14151 LaTeX~will~abort~loading~the~incompatible~support~files~
14152 but~this~may~lead~to~\\ later~errors.~Please~ensure~that~
14153 your~LaTeX~format~is~correctly~regenerated.
14154 }

```

(End definition for \\_kernel\_dependency\_version\_check:Nn and others.)

## 20.7 Messages

```
14155 __kernel_msg_new:nnnn { kernel } { file-not-found }
14156 { File~'#1'~not~found. }
14157 {
14158 The~requested~file~could~not~be~found~in~the~current~directory,~
14159 in~the~TeX~search~path~or~in~the~LaTeX~search~path.
14160 }
14161 __kernel_msg_new:nnn { kernel } { file-list }
14162 {
14163 >File~List~<
14164 #1 \\
14165
14166 }
14167 __kernel_msg_new:nnnn { kernel } { unbalanced-quote-in-filename }
14168 { Unbalanced~quotes~in~file~name~'#1'. }
14169 {
14170 File~names~must~contain~balanced~numbers~of~quotes~(").
14171 }
14172 __kernel_msg_new:nnnn { kernel } { iow-indent }
14173 { Only~#1 (arg~1)~allows~#2 }
14174 {
14175 The~command~#2 can~only~be~used~in~messages~
14176 which~will~be~wrapped~using~#1.
14177 \tl_if_empty:nF {#3} { ~ It~was~called~with~argument~'#3'. }
14178 }
```

## 20.8 Functions delayed from earlier modules

<@@=sys>

**\c\_sys\_platform\_str** Detecting the platform on LuaTeX is easy: for other engines, we use the fact that the two common cases have special null files. It is possible to probe further (see package `platform`), but that requires shell escape and seems unlikely to be useful. This is set up here as it requires file searching.

```
14179 \sys_if_engine_luatex:TF
14180 {
14181 \str_const:Nx \c_sys_platform_str
14182 { \tex_directlua:D { tex.print(os.type) } }
14183 }
14184 {
14185 \file_if_exist:nTF { nul: }
14186 {
14187 \file_if_exist:nF { /dev/null }
14188 { \str_const:Nn \c_sys_platform_str { windows } }
14189 }
14190 {
14191 \file_if_exist:nT { /dev/null }
14192 { \str_const:Nn \c_sys_platform_str { unix } }
14193 }
14194 }
14195 \cs_if_exist:NF \c_sys_platform_str
14196 { \str_const:Nn \c_sys_platform_str { unknown } }
```

(End definition for `\c_sys_platform_str`. This variable is documented on page [116](#).)

```

\sys_if_platform_unix:p: We can now set up the tests.
\sys_if_platform_unix:TF
\sys_if_platform_windows:p:
\sys_if_platform_windows:TF
14197 \clist_map_inline:nn { unix , windows }
14198 {
14199 __file_const:nn { sys_if_platform_ #1 }
14200 { \str_if_eq_p:Vn \c_sys_platform_str { #1 } }
14201 }

(End definition for \sys_if_platform_unix:TF and \sys_if_platform_windows:TF. These functions are
documented on page 116.)
```

14202 ⟨/package⟩

## 21 l3skip implementation

```

14203 ⟨*package⟩
14204 ⟨@@=dim⟩
```

### 21.1 Length primitives renamed

Primitives renamed.

```

\if_dim:w
__dim_eval:w \tex_ifdim:D
__dim_eval_end: \tex_dimexpr:D
__dim_eval:w \tex_relax:D
```

(End definition for \if\_dim:w, \\_\_dim\_eval:w, and \\_\_dim\_eval\_end:. This function is documented on
page 184.)

### 21.2 Internal auxiliaries

\s\_\_dim\_mark Internal scan marks.

```

\s__dim_stop
14208 \scan_new:N \s__dim_mark
14209 \scan_new:N \s__dim_stop
```

(End definition for \s\_\_dim\_mark and \s\_\_dim\_stop.)

\\_\_dim\_use\_none\_delimit\_by\_s\_stop:w Functions to gobble up to a scan mark.

```

14210 \cs_new:Npn __dim_use_none_delimit_by_s_stop:w #1 \s__dim_stop { }
```

(End definition for \\_\_dim\_use\_none\_delimit\_by\_s\_stop:w.)

### 21.3 Creating and initialising dim variables

\dim\_new:N Allocating ⟨dim⟩ registers ...

```

\dim_new:c
14211 \cs_new_protected:Npn \dim_new:N #1
14212 {
14213 __kernel_chk_if_free_cs:N #1
14214 \cs:w newdimen \cs_end: #1
14215 }
14216 \cs_generate_variant:Nn \dim_new:N { c }
```

(End definition for \dim\_new:N. This function is documented on page 170.)

**\dim\_const:Nn** Contrarily to integer constants, we cannot avoid using a register, even for constants. We cannot use `\dim_gset:Nn` because debugging code would complain that the constant is not a global variable. Since `\dim_const:Nn` does not need to be fast, use `\dim_eval:n` to avoid needing a debugging patch that wraps the expression in checking code.

```

14217 \cs_new_protected:Npn \dim_const:Nn #1#2
14218 {
14219 \dim_new:N #1
14220 \tex_global:D #1 ~ \dim_eval:n {#2} \scan_stop:
14221 }
14222 \cs_generate_variant:Nn \dim_const:Nn { c }

```

(End definition for `\dim_const:Nn`. This function is documented on page 170.)

**\dim\_zero:N** Reset the register to zero. Using `\c_zero_skip` deals with the case where the variable passed is incorrectly a skip (for example a L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> length).

```

14223 \cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_skip }
14224 \cs_new_protected:Npn \dim_gzero:N #1
14225 { \tex_global:D #1 \c_zero_skip }
14226 \cs_generate_variant:Nn \dim_zero:N { c }
14227 \cs_generate_variant:Nn \dim_gzero:N { c }

```

(End definition for `\dim_zero:N` and `\dim_gzero:N`. These functions are documented on page 170.)

**\dim\_zero\_new:N** Create a register if needed, otherwise clear it.

```

14228 \cs_new_protected:Npn \dim_zero_new:N #1
14229 { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
14230 \cs_new_protected:Npn \dim_gzero_new:N #1
14231 { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
14232 \cs_generate_variant:Nn \dim_zero_new:N { c }
14233 \cs_generate_variant:Nn \dim_gzero_new:N { c }

```

(End definition for `\dim_zero_new:N` and `\dim_gzero_new:N`. These functions are documented on page 170.)

**\dim\_if\_exist\_p:N** Copies of the `cs` functions defined in l3basics.

```

14234 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N
14235 { TF , T , F , p }
14236 \prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c
14237 { TF , T , F , p }

```

(End definition for `\dim_if_exist:NTF`. This function is documented on page 170.)

## 21.4 Setting dim variables

**\dim\_set:Nn** Setting dimensions is easy enough but when debugging we want both to check that the variable is correctly local/global and to wrap the expression in some code. The `\scan_stop:` deals with the case where the variable passed is a skip (for example a L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> length).

```

14238 \cs_new_protected:Npn \dim_set:Nn #1#2
14239 { #1 ~ _dim_eval:w #2 _dim_eval_end: \scan_stop: }
14240 \cs_new_protected:Npn \dim_gset:Nn #1#2
14241 { \tex_global:D #1 ~ _dim_eval:w #2 _dim_eval_end: \scan_stop: }
14242 \cs_generate_variant:Nn \dim_set:Nn { c }
14243 \cs_generate_variant:Nn \dim_gset:Nn { c }

```

(End definition for `\dim_set:Nn` and `\dim_gset:Nn`. These functions are documented on page 171.)

`\dim_set_eq:NN` All straightforward, with a `\scan_stop:` to deal with the case where #1 is (incorrectly) a skip.  
`\dim_set_eq:cN`  
`\dim_set_eq:cc`  
`\dim_gset_eq:NN`  
`\dim_gset_eq:cN`  
`\dim_gset_eq:Nc`  
`\dim_gset_eq:cc`

```
14244 \cs_new_protected:Npn \dim_set_eq:NN #1#2
14245 { #1 = #2 \scan_stop: }
14246 \cs_generate_variant:Nn \dim_set_eq:NN { c , Nc , cc }
14247 \cs_new_protected:Npn \dim_gset_eq:NN #1#2
14248 { \tex_global:D #1 = #2 \scan_stop: }
14249 \cs_generate_variant:Nn \dim_gset_eq:NN { c , Nc , cc }
```

(End definition for `\dim_set_eq:NN` and `\dim_gset_eq:NN`. These functions are documented on page 171.)

`\dim_add:Nn` Using by here deals with the (incorrect) case `\dimen123`. Using `\scan_stop:` deals with skip variables. Since debugging checks that the variable is correctly local/global, the global versions cannot be defined as `\tex_global:D` followed by the local versions. The debugging code is inserted by `\_dim_tmp:w`.

`\dim_add:cn`  
`\dim_gadd:Nn`  
`\dim_gadd:cn`  
`\dim_sub:Nn`  
`\dim_sub:cn`  
`\dim_gsub:Nn`  
`\dim_gsub:cn`

```
14250 \cs_new_protected:Npn \dim_add:Nn #1#2
14251 { \tex_advance:D #1 by _dim_eval:w #2 _dim_eval_end: \scan_stop: }
14252 \cs_new_protected:Npn \dim_gadd:Nn #1#2
14253 {
14254 \tex_global:D \tex_advance:D #1 by
14255 _dim_eval:w #2 _dim_eval_end: \scan_stop:
14256 }
14257 \cs_generate_variant:Nn \dim_add:Nn { c }
14258 \cs_generate_variant:Nn \dim_gadd:Nn { c }
14259 \cs_new_protected:Npn \dim_sub:Nn #1#2
14260 { \tex_advance:D #1 by - _dim_eval:w #2 _dim_eval_end: \scan_stop: }
14261 \cs_new_protected:Npn \dim_gsub:Nn #1#2
14262 {
14263 \tex_global:D \tex_advance:D #1 by
14264 -_dim_eval:w #2 _dim_eval_end: \scan_stop:
14265 }
14266 \cs_generate_variant:Nn \dim_sub:Nn { c }
14267 \cs_generate_variant:Nn \dim_gsub:Nn { c }
```

(End definition for `\dim_add:Nn` and others. These functions are documented on page 171.)

## 21.5 Utilities for dimension calculations

`\dim_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value is evaluated by removing a leading - if present.

`\_dim_abs:N`  
`\dim_max:nn`  
`\dim_min:nn`  
`\_dim_maxmin:wwN`

```
14268 \cs_new:Npn \dim_abs:n #1
14269 {
14270 \exp_after:wn _dim_abs:N
14271 \dim_use:N _dim_eval:w #1 _dim_eval_end:
14272 }
14273 \cs_new:Npn _dim_abs:N #1
14274 { \if_meaning:w - #1 \else: \exp_after:wn #1 \fi: }
14275 \cs_new:Npn \dim_max:nn #1#2
14276 {
14277 \dim_use:N _dim_eval:w \exp_after:wn _dim_maxmin:wwN
14278 \dim_use:N _dim_eval:w #1 \exp_after:wn ;
```

```

14279 \dim_use:N __dim_eval:w #2 ;
14280 >
14281 __dim_eval_end:
14282 }
14283 \cs_new:Npn \dim_min:nn #1#2
14284 {
14285 \dim_use:N __dim_eval:w \exp_after:wN __dim_maxmin:wwN
14286 \dim_use:N __dim_eval:w #1 \exp_after:wN ;
14287 \dim_use:N __dim_eval:w #2 ;
14288 <
14289 __dim_eval_end:
14290 }
14291 \cs_new:Npn __dim_maxmin:wwN #1 ; #2 ; #3
14292 {
14293 \if_dim:w #1 #3 #2 ~
14294 #1
14295 \else:
14296 #2
14297 \fi:
14298 }

```

(End definition for `\dim_abs:n` and others. These functions are documented on page 171.)

**\dim\_ratio:nn** With dimension expressions, something like `10 pt * ( 5 pt / 10 pt )` does not work. Instead, the ratio part needs to be converted to an integer expression. Using `\int_value:w` forces everything into sp, avoiding any decimal parts.

```

14299 \cs_new:Npn \dim_ratio:nn #1#2
14300 { __dim_ratio:n {#1} / __dim_ratio:n {#2} }
14301 \cs_new:Npn __dim_ratio:n #1
14302 { \int_value:w __dim_eval:w (#1) __dim_eval_end: }

```

(End definition for `\dim_ratio:nn` and `\__dim_ratio:n`. This function is documented on page 172.)

## 21.6 Dimension expression conditionals

**\dim\_compare\_p:nNn** Simple comparison.

```

14303 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
14304 {
14305 \if_dim:w __dim_eval:w #1 #2 __dim_eval:w #3 __dim_eval_end:
14306 \prg_return_true: \else: \prg_return_false: \fi:
14307 }

```

(End definition for `\dim_compare:nNnTF`. This function is documented on page 172.)

**\dim\_compare\_p:n** **\dim\_compare:nNf** **\\_\_dim\_compare:w** **\\_\_dim\_compare:wnN** **\\_\_dim\_compare\_=:w** **\\_\_dim\_compare\_!=w** **\\_\_dim\_compare\_<:w** **\\_\_dim\_compare\_>:w** **\\_\_dim\_compare\_error:** This code is adapted from the `\int_compare:nTF` function. First make sure that there is at least one relation operator, by evaluating a dimension expression with a trailing `\__dim_compare_error:`. Just like for integers, the looping auxiliary `\__dim_compare:wNN` closes a primitive conditional and opens a new one. It is actually easier to grab a dimension operand than an integer one, because once evaluated, dimensions all end with `pt` (with category other). Thus we do not need specific auxiliaries for the three “simple” relations `<`, `=`, and `>`.

```

14308 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
14309 {
14310 \exp_after:wN __dim_compare:w

```

```

14311 \dim_use:N __dim_eval:w #1 __dim_compare_error:
14312 }
14313 \cs_new:Npn __dim_compare:w #1 __dim_compare_error:
14314 {
14315 \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
14316 __dim_compare:wNN #1 ? { = __dim_compare_end:w \else: } \s__dim_stop
14317 }
14318 \exp_args:Nno \use:nn
14319 { \cs_new:Npn __dim_compare:wNN #1 } { \tl_to_str:n {pt} #2#3 }
14320 {
14321 \if_meaning:w = #3
14322 \use:c { __dim_compare_#2:w }
14323 \fi:
14324 #1 pt \exp_stop_f:
14325 \prg_return_false:
14326 \exp_after:wN __dim_use_none_delimit_by_s_stop:w
14327 \fi:
14328 \reverse_if:N \if_dim:w #1 pt #2
14329 \exp_after:wN __dim_compare:wNN
14330 \dim_use:N __dim_eval:w #3
14331 }
14332 \cs_new:cpn { __dim_compare_ ! :w }
14333 #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
14334 \cs_new:cpn { __dim_compare_ = :w }
14335 #1 __dim_eval:w = { #1 __dim_eval:w }
14336 \cs_new:cpn { __dim_compare_ < :w }
14337 #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
14338 \cs_new:cpn { __dim_compare_ > :w }
14339 #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
14340 \cs_new:Npn __dim_compare_end:w #1 \prg_return_false: #2 \s__dim_stop
14341 { #1 \prg_return_false: \else: \prg_return_true: \fi: }
14342 \cs_new_protected:Npn __dim_compare_error:
14343 {
14344 \if_int_compare:w \c_zero_int \c_zero_int \fi:
14345 =
14346 __dim_compare_error:
14347 }

```

(End definition for `\dim_compare:nTF` and others. This function is documented on page [173](#).)

`\dim_case:nn` For dimension cases, the first task to fully expand the check condition. The over all idea is then much the same as for `\str_case:nn(TF)` as described in `I3basics`.

```

\dim_case:nnTF
\dim_case:nnTF
__dim_case:nnTF
__dim_case:nw
__dim_case_end:nw
14348 \cs_new:Npn \dim_case:nnTF #1
14349 {
14350 \exp:w
14351 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} }
14352 }
14353 \cs_new:Npn \dim_case:nnT #1#2#3
14354 {
14355 \exp:w
14356 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
14357 }
14358 \cs_new:Npn \dim_case:nnF #1#2
14359 {

```

```

14360 \exp:w
14361 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
14362 }
14363 \cs_new:Npn \dim_case:nn #1#2
14364 {
14365 \exp:w
14366 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
14367 }
14368 \cs_new:Npn __dim_case:nnTF #1#2#3#4
14369 { __dim_case:nw {#1} #2 {#1} { } \s_dim_mark {#3} \s_dim_mark {#4} \s_dim_stop }
14370 \cs_new:Npn __dim_case:nw #1#2#3
14371 {
14372 \dim_compare:nNnTF {#1} = {#2}
14373 { __dim_case_end:nw {#3} }
14374 { __dim_case:nw {#1} }
14375 }
14376 \cs_new:Npn __dim_case_end:nw #1#2#3 \s_dim_mark #4#5 \s_dim_stop
14377 { \exp_end: #1 #4 }

```

(End definition for `\dim_case:nnTF` and others. This function is documented on page 174.)

## 21.7 Dimension expression loops

`\dim_while_do:nn`  
`\dim_until_do:nn`

```

\dim_while_do:nn
\dim_until_do:nn
\dim_do_while:nn
\dim_do_until:nn
14378 \cs_new:Npn \dim_while_do:nn #1#2
14379 {
14380 \dim_compare:nT {#1}
14381 {
14382 #2
14383 \dim_while_do:nn {#1} {#2}
14384 }
14385 }
14386 \cs_new:Npn \dim_until_do:nn #1#2
14387 {
14388 \dim_compare:nF {#1}
14389 {
14390 #2
14391 \dim_until_do:nn {#1} {#2}
14392 }
14393 }
14394 \cs_new:Npn \dim_do_while:nn #1#2
14395 {
14396 #2
14397 \dim_compare:nT {#1}
14398 { \dim_do_while:nn {#1} {#2} }
14399 }
14400 \cs_new:Npn \dim_do_until:nn #1#2
14401 {
14402 #2
14403 \dim_compare:nF {#1}
14404 { \dim_do_until:nn {#1} {#2} }
14405 }

```

(End definition for `\dim_while_do:nNnn` and others. These functions are documented on page 175.)

```
\dim_while_do:nNnn while_do and do_while functions for dimensions. Same as for the int type only the
\dim_until_do:nNnn names have changed.

14406 \cs_new:Npn \dim_while_do:nNnn #1#2#3#4
14407 {
14408 \dim_compare:nNnT {#1} #2 {#3}
14409 {
14410 #4
14411 \dim_while_do:nNnn {#1} #2 {#3} {#4}
14412 }
14413 }
14414 \cs_new:Npn \dim_until_do:nNnn #1#2#3#4
14415 {
14416 \dim_compare:nNnF {#1} #2 {#3}
14417 {
14418 #4
14419 \dim_until_do:nNnn {#1} #2 {#3} {#4}
14420 }
14421 }
14422 \cs_new:Npn \dim_do_while:nNnn #1#2#3#4
14423 {
14424 #4
14425 \dim_compare:nNnT {#1} #2 {#3}
14426 { \dim_while:nNnn {#1} #2 {#3} {#4} }
14427 }
14428 \cs_new:Npn \dim_do_until:nNnn #1#2#3#4
14429 {
14430 #4
14431 \dim_compare:nNnF {#1} #2 {#3}
14432 { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
14433 }
```

(End definition for `\dim_while_do:nNnn` and others. These functions are documented on page 175.)

## 21.8 Dimension step functions

`\dim_step_function:nnnN`  
`\_dim_step:wwwN`  
`\_dim_step:NnnnN`

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```
14434 \cs_new:Npn \dim_step_function:nnnN #1#2#3
14435 {
14436 \exp_after:wN _dim_step:wwwN
14437 \tex_the:D _dim_eval:w #1 \exp_after:wN ;
14438 \tex_the:D _dim_eval:w #2 \exp_after:wN ;
14439 \tex_the:D _dim_eval:w #3 ;
14440 }
14441 \cs_new:Npn _dim_step:wwwN #1; #2; #3; #4
14442 {
14443 \dim_compare:nNnTF {#2} > \c_zero_dim
14444 { _dim_step:NnnnN > }
```

```

14445 {
14446 \dim_compare:nNnTF {#2} = \c_zero_dim
14447 {
14448 __kernel_msg_expandable_error:nnn { kernel } { zero-step } {#4}
14449 \use_none:nnnn
14450 }
14451 { __dim_step:NnnnN < }
14452 }
14453 {#1} {#2} {#3} #4
14454 }
14455 \cs_new:Npn __dim_step:NnnnN #1#2#3#4#5
14456 {
14457 \dim_compare:nNnF {#2} #1 {#4}
14458 {
14459 #5 {#2}
14460 \exp_args:NNf __dim_step:NnnnN
14461 #1 { \dim_eval:n { #2 + #3 } } {#3} {#4} #5
14462 }
14463 }

```

(End definition for `\dim_step_function:nnnN`, `\__dim_step:wwwN`, and `\__dim_step:NnnnN`. This function is documented on page 175.)

`\dim_step_inline:nnn`  
`\dim_step_variable:nnnNn`  
`\__dim_step:NNnnnn`

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\dim_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g_kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```

14464 \cs_new_protected:Npn \dim_step_inline:nnn
14465 {
14466 \int_gincr:N \g_kernel_prg_map_int
14467 \exp_args:NNc __dim_step:NNnnnn
14468 \cs_gset_protected:Npn
14469 { __dim_map_ \int_use:N \g_kernel_prg_map_int :w }
14470 }
14471 \cs_new_protected:Npn \dim_step_variable:nnnNn #1#2#3#4#5
14472 {
14473 \int_gincr:N \g_kernel_prg_map_int
14474 \exp_args:NNc __dim_step:NNnnnn
14475 \cs_gset_protected:Npx
14476 { __dim_map_ \int_use:N \g_kernel_prg_map_int :w }
14477 {#1}{#2}{#3}
14478 {
14479 \tl_set:Nn \exp_not:N #4 {##1}
14480 \exp_not:n {#5}
14481 }
14482 }
14483 \cs_new_protected:Npn __dim_step:NNnnnn #1#2#3#4#5#6
14484 {
14485 #1 #2 ##1 {#6}
14486 \dim_step_function:nnnN {#3} {#4} {#5} #2
14487 \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g_kernel_prg_map_int }
14488 }

```

(End definition for `\dim_step_inline:nnn`, `\dim_step_variable:nnNn`, and `\_dim_step>NNnnnn`.  
These functions are documented on page 175.)

## 21.9 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```
14489 \cs_new:Npn \dim_eval:n #1
14490 { \dim_use:N _dim_eval:w #1 _dim_eval_end: }
```

(End definition for `\dim_eval:n`. This function is documented on page 176.)

`\dim_sign:n` See `\dim_abs:n`. Contrarily to `\int_sign:n` the case of a zero dimension cannot be distinguished from a positive dimension by looking only at the first character, since `0.2pt` and `Opt` start the same way. We need explicit comparisons. We start by distinguishing the most common case of a positive dimension.

```
14491 \cs_new:Npn \dim_sign:n #1
14492 {
14493 \int_value:w \exp_after:wN _dim_sign:Nw
14494 \dim_use:N _dim_eval:w #1 _dim_eval_end: ;
14495 \exp_stop_f:
14496 }
14497 \cs_new:Npn _dim_sign:Nw #1#2 ;
14498 {
14499 \if_dim:w #1#2 > \c_zero_dim
14500 1
14501 \else:
14502 \if_meaning:w - #1
14503 -1
14504 \else:
14505 0
14506 \fi:
14507 \fi:
14508 }
```

(End definition for `\dim_sign:n` and `\_dim_sign:Nw`. This function is documented on page 176.)

`\dim_use:N` Accessing a  $\langle dim \rangle$ .

`\dim_use:c` `\cs_new_eq:NN \dim_use:N \tex_the:D`

We hand-code this for some speed gain:

```
14510 \%cs_generate_variant:Nn \dim_use:N { c }
14511 \cs_new:Npn \dim_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(End definition for `\dim_use:N`. This function is documented on page 176.)

`\dim_to_decimal:n` `\_dim_to_decimal:w` A function which comes up often enough to deserve a place in the kernel. Evaluate the dimension expression `#1` then remove the trailing `pt`. When debugging is enabled, the argument is put in parentheses as this prevents the dimension expression from terminating early and leaving extra tokens lying around. This is used a lot by low-level manipulations.

```
14512 \cs_new:Npn \dim_to_decimal:n #1
14513 {
14514 \exp_after:wN
14515 _dim_to_decimal:w \dim_use:N _dim_eval:w #1 _dim_eval_end:
14516 }
```

```

14517 \use:x
14518 {
14519 \cs_new:Npn \exp_not:N __dim_to_decimal:w
14520 ##1 . ##2 \tl_to_str:n { pt }
14521 }
14522 {
14523 \int_compare:nNnTF {#2} > { 0 }
14524 { #1 . #2 }
14525 { #1 }
14526 }

```

(End definition for `\dim_to_decimal:n` and `\__dim_to_decimal:w`. This function is documented on page 176.)

**\dim\_to\_decimal\_in\_bp:n** Conversion to big points is done using a scaling inside `\__dim_eval:w` as ε-TEx does that using 64-bit precision. Here, 800/803 is the integer fraction for 72/72.27. This is a common case so is hand-coded for accuracy (and speed).

```

14527 \cs_new:Npn \dim_to_decimal_in_bp:n #1
14528 { \dim_to_decimal:n { (#1) * 800 / 803 } }

```

(End definition for `\dim_to_decimal_in_bp:n`. This function is documented on page 177.)

**\dim\_to\_decimal\_in\_sp:n** Another hard-coded conversion: this one is necessary to avoid things going off-scale.

```

14529 \cs_new:Npn \dim_to_decimal_in_sp:n #1
14530 { \int_value:w __dim_eval:w #1 __dim_eval_end: }

```

(End definition for `\dim_to_decimal_in_sp:n`. This function is documented on page 177.)

**\dim\_to\_decimal\_in\_unit:nn** An analogue of `\dim_ratio:nn` that produces a decimal number as its result, rather than a rational fraction for use within dimension expressions.

```

14531 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
14532 {
14533 \dim_to_decimal:n
14534 {
14535 1pt *
14536 \dim_ratio:nn {#1} {#2}
14537 }
14538 }

```

(End definition for `\dim_to_decimal_in_unit:nn`. This function is documented on page 177.)

**\dim\_to\_fp:n** Defined in `l3fp-convert`, documented here.

(End definition for `\dim_to_fp:n`. This function is documented on page 177.)

## 21.10 Viewing dim variables

**\dim\_show:N** Diagnostics.

```

\dim_show:c
14539 \cs_new_eq:NN \dim_show:N __kernel_register_show:N
14540 \cs_generate_variant:Nn \dim_show:N { c }

```

(End definition for `\dim_show:N`. This function is documented on page 177.)

**\dim\_show:n** Diagnostics. We don't use the TeX primitive `\showthe` to show dimension expressions: this gives a more unified output.

```
14541 \cs_new_protected:Npn \dim_show:n
14542 { \msg_show_eval:Nn \dim_eval:n }
```

(End definition for `\dim_show:n`. This function is documented on page 178.)

**\dim\_log:N** Diagnostics. Redirect output of `\dim_show:n` to the log.

```
14543 \cs_new_eq:NN \dim_log:N __kernel_register_log:N
14544 \cs_new_eq:NN \dim_log:c __kernel_register_log:c
14545 \cs_new_protected:Npn \dim_log:n
14546 { \msg_log_eval:Nn \dim_eval:n }
```

(End definition for `\dim_log:N` and `\dim_log:n`. These functions are documented on page 178.)

## 21.11 Constant dimensions

**\c\_zero\_dim** Constant dimensions.

```
14547 \dim_const:Nn \c_zero_dim { 0 pt }
14548 \dim_const:Nn \c_max_dim { 16383.99999 pt }
```

(End definition for `\c_zero_dim` and `\c_max_dim`. These variables are documented on page 178.)

## 21.12 Scratch dimensions

**\l\_tmpa\_dim** We provide two local and two global scratch registers, maybe we need more or less.

```
14549 \dim_new:N \l_tmpa_dim
14550 \dim_new:N \l_tmpb_dim
14551 \dim_new:N \g_tmpa_dim
14552 \dim_new:N \g_tmpb_dim
```

(End definition for `\l_tmpa_dim` and others. These variables are documented on page 178.)

## 21.13 Creating and initialising skip variables

```
14553 <@=skip>
```

**\s\_\_skip\_stop** Internal scan marks.

```
14554 \scan_new:N \s__skip_stop
```

(End definition for `\s__skip_stop`.)

**\skip\_new:N** Allocation of a new internal registers.

```
14555 \cs_new_protected:Npn \skip_new:N #1
14556 {
14557 __kernel_chk_if_free_cs:N #1
14558 \cs:w newskip \cs_end: #1
14559 }
14560 \cs_generate_variant:Nn \skip_new:N { c }
```

(End definition for `\skip_new:N`. This function is documented on page 178.)

**\skip\_const:Nn** Contrarily to integer constants, we cannot avoid using a register, even for constants. See **\dim\_const:Nn** for why we cannot use **\skip\_gset:Nn**.

```

14561 \cs_new_protected:Npn \skip_const:Nn #1#2
14562 {
14563 \skip_new:N #1
14564 \tex_global:D #1 ~ \skip_eval:n {#2} \scan_stop:
14565 }
14566 \cs_generate_variant:Nn \skip_const:Nn { c }

```

(End definition for **\skip\_const:Nn**. This function is documented on page 179.)

**\skip\_zero:N** Reset the register to zero.

```

\skip_zero:c 14567 \cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
\skip_gzero:N 14568 \cs_new_protected:Npn \skip_gzero:N #1 { \tex_global:D #1 \c_zero_skip }
\skip_gzero:c 14569 \cs_generate_variant:Nn \skip_zero:N { c }
14570 \cs_generate_variant:Nn \skip_gzero:N { c }

```

(End definition for **\skip\_zero:N** and **\skip\_gzero:N**. These functions are documented on page 179.)

**\skip\_zero\_new:N** Create a register if needed, otherwise clear it.

```

\skip_zero_new:c 14571 \cs_new_protected:Npn \skip_zero_new:N #1
\skip_gzero_new:N 14572 { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\skip_gzero_new:c 14573 \cs_new_protected:Npn \skip_gzero_new:N #1
14574 { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
14575 \cs_generate_variant:Nn \skip_zero_new:N { c }
14576 \cs_generate_variant:Nn \skip_gzero_new:N { c }

```

(End definition for **\skip\_zero\_new:N** and **\skip\_gzero\_new:N**. These functions are documented on page 179.)

**\skip\_if\_exist\_p:N** Copies of the **cs** functions defined in **l3basics**.

```

\skip_if_exist_p:c 14577 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N
\skip_if_exist:NTF 14578 { TF , T , F , p }
\skip_if_exist:cTF 14579 \prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
14580 { TF , T , F , p }

```

(End definition for **\skip\_if\_exist:NTF**. This function is documented on page 179.)

## 21.14 Setting skip variables

**\skip\_set:Nn** Much the same as for dimensions.

```

\skip_set:cn 14581 \cs_new_protected:Npn \skip_set:Nn #1#2
14582 { #1 ~ \tex_glueexpr:D #2 \scan_stop: }
\skip_gset:Nn 14583 \cs_new_protected:Npn \skip_gset:Nn #1#2
14584 { \tex_global:D #1 ~ \tex_glueexpr:D #2 \scan_stop: }
14585 \cs_generate_variant:Nn \skip_set:Nn { c }
14586 \cs_generate_variant:Nn \skip_gset:Nn { c }

```

(End definition for **\skip\_set:Nn** and **\skip\_gset:Nn**. These functions are documented on page 179.)

**\skip\_set\_eq:NN** All straightforward.

```

\skip_set_eq:cN 14587 \cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
14588 \cs_generate_variant:Nn \skip_set_eq:NN { c , Nc , cc }
\skip_set_eq:cc 14589 \cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
14590 \cs_generate_variant:Nn \skip_gset_eq:NN { c , Nc , cc }
\skip_gset_eq:cN
\skip_gset_eq:Nc
\skip_gset_eq:cc

```

(End definition for `\skip_set_eq:NN` and `\skip_gset_eq:NN`. These functions are documented on page 179.)

`\skip_add:Nn` Using by here deals with the (incorrect) case `\skip123`.

```
14591 \cs_new_protected:Npn \skip_add:Nn #1#2
14592 { \tex_advance:D #1 by \tex_glueexpr:D #2 \scan_stop: }
14593 \cs_new_protected:Npn \skip_gadd:Nn #1#2
14594 { \tex_global:D \tex_advance:D #1 by \tex_glueexpr:D #2 \scan_stop: }
14595 \cs_generate_variant:Nn \skip_add:Nn { c }
14596 \cs_generate_variant:Nn \skip_gadd:Nn { c }
14597 \cs_new_protected:Npn \skip_sub:Nn #1#2
14598 { \tex_advance:D #1 by - \tex_glueexpr:D #2 \scan_stop: }
14599 \cs_new_protected:Npn \skip_gsub:Nn #1#2
14600 { \tex_global:D \tex_advance:D #1 by - \tex_glueexpr:D #2 \scan_stop: }
14601 \cs_generate_variant:Nn \skip_sub:Nn { c }
14602 \cs_generate_variant:Nn \skip_gsub:Nn { c }
```

(End definition for `\skip_add:Nn` and others. These functions are documented on page 179.)

## 21.15 Skip expression conditionals

`\skip_if_eq_p:nnTF` Comparing skips means doing two expansions to make strings, and then testing them. As a result, only equality is tested.

```
14603 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
14604 {
14605 \str_if_eq:eeTF { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
14606 { \prg_return_true: }
14607 { \prg_return_false: }
14608 }
```

(End definition for `\skip_if_eq:nnTF`. This function is documented on page 180.)

`\skip_if_finite_p:n` With  $\varepsilon$ - $\text{\TeX}$ , we have an easy access to the order of infinites of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string `fil` which characterizes infinite glue.

```
14609 \cs_set_protected:Npn __skip_tmp:w #1
14610 {
14611 \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
14612 {
14613 \exp_after:wN __skip_if_finite:wwNw
14614 \skip_use:N \tex_glueexpr:D ##1 ; \prg_return_false:
14615 #1 ; \prg_return_true: \s__skip_stop
14616 }
14617 \cs_new:Npn __skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \s__skip_stop {##3}
14618 }
14619 \exp_args:No __skip_tmp:w { \tl_to_str:n { fil } }
```

(End definition for `\skip_if_finite:nTF` and `\__skip_if_finite:wwNw`. This function is documented on page 180.)

## 21.16 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```
14620 \cs_new:Npn \skip_eval:n #1
14621 { \skip_use:N \tex_glueexpr:D #1 \scan_stop: }
```

(End definition for `\skip_eval:n`. This function is documented on page 180.)

`\skip_use:N` Accessing a  $\langle\text{skip}\rangle$ .

```
14622 \cs_new_eq:NN \skip_use:N \tex_the:D
14623 \%cs_generate_variant:Nn \skip_use:N { c }
14624 \cs_new:Npn \skip_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(End definition for `\skip_use:N`. This function is documented on page 180.)

## 21.17 Inserting skips into the output

`\skip_horizontal:N` Inserting skips.

```
14625 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
14626 \cs_new:Npn \skip_horizontal:n #1
14627 { \skip_horizontal:N \tex_glueexpr:D #1 \scan_stop: }
14628 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
14629 \cs_new:Npn \skip_vertical:n #1
14630 { \skip_vertical:N \tex_glueexpr:D #1 \scan_stop: }
14631 \%cs_generate_variant:Nn \skip_horizontal:N { c }
14632 \%cs_generate_variant:Nn \skip_vertical:N { c }
```

(End definition for `\skip_horizontal:N` and others. These functions are documented on page 181.)

## 21.18 Viewing skip variables

`\skip_show:N` Diagnostics.

```
14633 \cs_new_eq:NN \skip_show:N __kernel_register_show:N
14634 \%cs_generate_variant:Nn \skip_show:N { c }
```

(End definition for `\skip_show:N`. This function is documented on page 180.)

`\skip_show:c` Diagnostics. We don't use the TeX primitive `\showthe` to show skip expressions: this gives a more unified output.

```
14635 \cs_new_protected:Npn \skip_show:n
14636 { \msg_show_eval:Nn \skip_eval:n }
```

(End definition for `\skip_show:n`. This function is documented on page 180.)

`\skip_log:N` Diagnostics. Redirect output of `\skip_show:n` to the log.

```
14637 \cs_new_eq:NN \skip_log:N __kernel_register_log:N
14638 \cs_new_eq:NN \skip_log:c __kernel_register_log:c
14639 \cs_new_protected:Npn \skip_log:n
14640 { \msg_log_eval:Nn \skip_eval:n }
```

(End definition for `\skip_log:N` and `\skip_log:n`. These functions are documented on page 181.)

## 21.19 Constant skips

`\c_zero_skip` Skips with no rubber component are just dimensions but need to terminate correctly.  
`\c_max_skip`

```
14641 \skip_const:Nn \c_zero_skip { \c_zero_dim }
14642 \skip_const:Nn \c_max_skip { \c_max_dim }
```

(End definition for `\c_zero_skip` and `\c_max_skip`. These functions are documented on page 181.)

## 21.20 Scratch skips

`\l_tmpa_skip` We provide two local and two global scratch registers, maybe we need more or less.  
`\l_tmpb_skip`  
`\g_tmpa_skip`  
`\g_tmpb_skip`

```
14643 \skip_new:N \l_tmpa_skip
14644 \skip_new:N \l_tmpb_skip
14645 \skip_new:N \g_tmpa_skip
14646 \skip_new:N \g_tmpb_skip
```

(End definition for `\l_tmpa_skip` and others. These variables are documented on page 181.)

## 21.21 Creating and initialising `muskip` variables

`\muskip_new:N` And then we add muskips.

```
\muskip_new:c
14647 \cs_new_protected:Npn \muskip_new:N #1
14648 {
14649 __kernel_chk_if_free_cs:N #1
14650 \cs:w newmuskip \cs_end: #1
14651 }
14652 \cs_generate_variant:Nn \muskip_new:N { c }
```

(End definition for `\muskip_new:N`. This function is documented on page 182.)

`\muskip_const:Nn` See `\skip_const:Nn`.

```
\muskip_const:cn
14653 \cs_new_protected:Npn \muskip_const:Nn #1#2
14654 {
14655 \muskip_new:N #1
14656 \tex_global:D #1 ~ \muskip_eval:n {#2} \scan_stop:
14657 }
14658 \cs_generate_variant:Nn \muskip_const:Nn { c }
```

(End definition for `\muskip_const:Nn`. This function is documented on page 182.)

`\muskip_zero:N` Reset the register to zero.

```
\muskip_zero:c
14659 \cs_new_protected:Npn \muskip_zero:N #1
14660 { #1 \c_zero_muskip }
\muskip_gzero:N
14661 \cs_new_protected:Npn \muskip_gzero:N #1
14662 { \tex_global:D #1 \c_zero_muskip }
14663 \cs_generate_variant:Nn \muskip_zero:N { c }
14664 \cs_generate_variant:Nn \muskip_gzero:N { c }
```

(End definition for `\muskip_zero:N` and `\muskip_gzero:N`. These functions are documented on page 182.)

```

\muskip_zero_new:N Create a register if needed, otherwise clear it.
\muskip_zero_new:c 14665 \cs_new_protected:Npn \muskip_zero_new:N #1
14666 { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
\muskip_gzero_new:N 14667 \cs_new_protected:Npn \muskip_gzero_new:N #1
14668 { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
14669 \cs_generate_variant:Nn \muskip_zero_new:N { c }
14670 \cs_generate_variant:Nn \muskip_gzero_new:N { c }

(End definition for \muskip_zero_new:N and \muskip_gzero_new:N. These functions are documented on page 182.)

```

```

\muskip_if_exist_p:N Copies of the cs functions defined in l3basics.
\muskip_if_exist_p:c 14671 \prg_new_eq_conditional:NNN \muskip_if_exist:N \cs_if_exist:N
\muskip_if_exist:NTF 14672 { TF , T , F , p }
\muskip_if_exist:cTF 14673 \prg_new_eq_conditional:NNN \muskip_if_exist:c \cs_if_exist:c
14674 { TF , T , F , p }

(End definition for \muskip_if_exist:NTF. This function is documented on page 182.)

```

## 21.22 Setting muskip variables

```

\muskip_set:Nn This should be pretty familiar.
\muskip_set:cn 14675 \cs_new_protected:Npn \muskip_set:Nn #1#2
\muskip_gset:Nn 14676 { #1 ~ \tex_muexpr:D #2 \scan_stop: }
\muskip_gset:cn 14677 \cs_new_protected:Npn \muskip_gset:Nn #1#2
14678 { \tex_global:D #1 ~ \tex_muexpr:D #2 \scan_stop: }
14679 \cs_generate_variant:Nn \muskip_set:Nn { c }
14680 \cs_generate_variant:Nn \muskip_gset:Nn { c }

(End definition for \muskip_set:Nn and \muskip_gset:Nn. These functions are documented on page 183.)

```

```

\muskip_set_eq:NN All straightforward.
\muskip_set_eq:cN 14681 \cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
\muskip_set_eq:Nc 14682 \cs_generate_variant:Nn \muskip_set_eq:NN { c , Nc , cc }
\muskip_set_eq:cc 14683 \cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
14684 \cs_generate_variant:Nn \muskip_gset_eq:NN { c , Nc , cc }

(End definition for \muskip_set_eq:NN and \muskip_gset_eq:NN. These functions are documented on page 183.)

```

```

\muskip_gset_eq:cc Using by here deals with the (incorrect) case \muskip123.
\muskip_add:Nn 14685 \cs_new_protected:Npn \muskip_add:Nn #1#2
\muskip_add:cn 14686 { \tex_advance:D #1 by \tex_muexpr:D #2 \scan_stop: }
\muskip_gadd:Nn 14687 \cs_new_protected:Npn \muskip_gadd:Nn #1#2
14688 { \tex_global:D \tex_advance:D #1 by \tex_muexpr:D #2 \scan_stop: }
14689 \cs_generate_variant:Nn \muskip_add:Nn { c }
14690 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
14691 \cs_new_protected:Npn \muskip_sub:Nn #1#2
14692 { \tex_advance:D #1 by - \tex_muexpr:D #2 \scan_stop: }
14693 \cs_new_protected:Npn \muskip_gsub:Nn #1#2
14694 { \tex_global:D \tex_advance:D #1 by - \tex_muexpr:D #2 \scan_stop: }
14695 \cs_generate_variant:Nn \muskip_sub:Nn { c }
14696 \cs_generate_variant:Nn \muskip_gsub:Nn { c }

(End definition for \muskip_add:Nn and others. These functions are documented on page 182.)

```

## 21.23 Using muskip expressions and variables

`\muskip_eval:n` Evaluating a muskip expression expandably.

```
14697 \cs_new:Npn \muskip_eval:n #1
14698 { \muskip_use:N \tex_muexpr:D #1 \scan_stop: }
```

(End definition for `\muskip_eval:n`. This function is documented on page 183.)

`\muskip_use:N` Accessing a *(muskip)*.

```
14699 \cs_new_eq:NN \muskip_use:N \tex_the:D
14700 \cs_generate_variant:Nn \muskip_use:N { c }
```

(End definition for `\muskip_use:N`. This function is documented on page 183.)

## 21.24 Viewing muskip variables

`\muskip_show:N` Diagnostics.

```
14701 \cs_new_eq:NN \muskip_show:N __kernel_register_show:N
14702 \cs_generate_variant:Nn \muskip_show:N { c }
```

(End definition for `\muskip_show:N`. This function is documented on page 183.)

`\muskip_show:c` Diagnostics. We don't use the TeX primitive `\showthe` to show muskip expressions: this gives a more unified output.

```
14703 \cs_new_protected:Npn \muskip_show:n
14704 { \msg_show_eval:Nn \muskip_eval:n }
```

(End definition for `\muskip_show:n`. This function is documented on page 184.)

`\muskip_log:N` Diagnostics. Redirect output of `\muskip_show:n` to the log.

```
14705 \cs_new_eq:NN \muskip_log:N __kernel_register_log:N
14706 \cs_new_eq:NN \muskip_log:c __kernel_register_log:c
14707 \cs_new_protected:Npn \muskip_log:n
14708 { \msg_log_eval:Nn \muskip_eval:n }
```

(End definition for `\muskip_log:N` and `\muskip_log:n`. These functions are documented on page 184.)

## 21.25 Constant muskips

`\c_zero_muskip` Constant muskips given by their value.

```
14709 \muskip_const:Nn \c_zero_muskip { 0 mu }
14710 \muskip_const:Nn \c_max_muskip { 16383.99999 mu }
```

(End definition for `\c_zero_muskip` and `\c_max_muskip`. These functions are documented on page 184.)

## 21.26 Scratch muskips

We provide two local and two global scratch registers, maybe we need more or less.

```
14711 \muskip_new:N \l_tmpa_muskip
14712 \muskip_new:N \l_tmpb_muskip
14713 \muskip_new:N \g_tmpa_muskip
14714 \muskip_new:N \g_tmpb_muskip
```

(End definition for `\l_tmpa_muskip` and others. These variables are documented on page 184.)

```
14715
```

## 22 I3keys Implementation

14716 `(*package)`

### 22.1 Low-level interface

The low-level key parser's implementation is based heavily on `expkv`. Compared to `keyval` it adds a number of additional "safety" requirements and allows to process the parsed list of key-value pairs in a variety of ways. The net result is that this code needs around one and a half the amount of time as `keyval` to parse the same list of keys. To optimise speed as far as reasonably practical, a number of lower-level approaches are taken rather than using the higher-level `expl3` interfaces.

14717 `(@@=keyval)`

```
\s__keyval_nil
\s__keyval_mark
\s__keyval_stop
\s__keyval_tail
14718 \scan_new:N \s__keyval_nil
14719 \scan_new:N \s__keyval_mark
14720 \scan_new:N \s__keyval_stop
14721 \scan_new:N \s__keyval_tail
```

*(End definition for `\s__keyval_nil` and others.)*

This temporary macro will be used since some of the definitions will need an active comma or equals sign. Inside of this macro #1 will be the active comma and #2 will be the active equals sign.

```
14722 \group_begin:
14723 \cs_set_protected:Npn __keyval_tmp:NN #1#2
14724 }
```

`\keyval_parse:NNn`

The main function starts the first of two loops. The outer loop splits the key-value list at active commas, the inner loop will do so at other commas. The use of `\s__keyval_mark` here prevents loss of braces from the key argument.

```
14725 \cs_new:Npn \keyval_parse:NNn ##1 ##2 ##3
14726 { __keyval_loop_active:NNw ##1 ##2 \s__keyval_mark ##3 #1 \s__keyval_tail #1 }
```

*(End definition for `\keyval_parse:NNn`. This function is documented on page 197.)*

`\__keyval_loop_active:NNw`

First a fast test for the end of the loop is done, it'll gobble everything up to a `\s__keyval_tail`. The loop ending macro will gobble everything to the last `\s__keyval_mark` in this definition. If the end isn't reached yet, start the second loop splitting at other commas, and after that one iterate the current loop.

```
14727 \cs_new:Npn __keyval_loop_active:NNw ##1 ##2 ##3 #1
14728 {
14729 __keyval_if_recursion_tail:w ##3
14730 __keyval_end_loop_active:w \s__keyval_tail
14731 __keyval_loop_other:NNw ##1 ##2 ##3 , \s__keyval_tail ,
14732 __keyval_loop_active:NNw ##1 ##2 \s__keyval_mark
14733 }
```

*(End definition for `\__keyval_loop_active:NNw`.)*

```
__keyval_split_other:w
__keyval_split_active:w
```

These two macros allow to split at the first equals sign of category 12 or 13. At the same time they also execute branching by inserting the first token following `\s__keyval_mark` that followed the equals sign. Hence they also test for the presence of such an equals sign simultaneously.

```
14734 \cs_new:Npn __keyval_split_other:w ##1 = ##2 \s__keyval_mark ##3 ##4 \s__keyval_stop
14735 { ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }
14736 \cs_new:Npn __keyval_split_active:w ##1 #2 ##2 \s__keyval_mark ##3 ##4 \s__keyval_stop
14737 { ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }
```

(End definition for `\_\_keyval\_split\_other:w` and `\_\_keyval\_split\_active:w`.)

```
__keyval_loop_other>NNw
```

The second loop uses the same test for its end as the first loop, next it splits at the first active equals sign using `\_\_keyval\_split\_active:w`. The `\s__keyval_nil` prevents accidental brace stripping and acts as a delimiter in the next steps. First testing for an active equals sign will reduce the number of necessary expansion steps for the expected average use case of other equals signs and hence perform better on average.

```
14738 \cs_new:Npn __keyval_loop_other>NNw ##1 ##2 ##3 ,
14739 {
14740 __keyval_if_recursion_tail:w ##3
14741 __keyval_end_loop_other:w \s__keyval_tail
14742 __keyval_split_active:w ##3 \s__keyval_nil
14743 \s__keyval_mark __keyval_split_active_auxi:w
14744 #2 \s__keyval_mark __keyval_clean_up_active:w
14745 \s__keyval_stop
14746 ##1 ##2
14747 __keyval_loop_other>NNw ##1 ##2 \s__keyval_mark
14748 }
```

(End definition for `\_\_keyval\_loop\_other>NNw`.)

```
__keyval_split_active_auxi:w
__keyval_split_active_auxii:w
__keyval_split_active_auxiii:w
__keyval_split_active_auxiv:w
__keyval_split_active_auxv:w
```

After `\_\_keyval\_split\_active:w` the following will only be called if there was at least one active equals sign in the current key-value pair. Therefore this is the execution branch for a key-value pair with an active equals sign. `##1` will be everything up to the first active equals sign. First it tests for other equals signs in the key name, which will eventually throw an error via `\_\_keyval\_misplaced\_equal\_after\_active\_error:w`. If none was found we forward the key to `\_\_keyval\_split\_active\_auxii:w`.

```
14749 \cs_new:Npn __keyval_split_active_auxi:w ##1 \s__keyval_stop
14750 {
14751 __keyval_split_other:w ##1 \s__keyval_nil
14752 \s__keyval_mark __keyval_misplaced_equal_after_active_error:w
14753 = \s__keyval_mark __keyval_split_active_auxii:w
14754 \s__keyval_stop
14755 }
```

`\_\_keyval\_split\_active\_auxii:w` gets the correct key name with a leading `\s__keyval_mark` as `##1`. It has to sanitise the remainder of the previous test and trims the key name which will be forwarded to `\_\_keyval\_split\_active\_auxiii:w`.

```
14756 \cs_new:Npn __keyval_split_active_auxii:w
14757 ##1 \s__keyval_nil \s__keyval_mark __keyval_misplaced_equal_after_active_error:w
14758 \s__keyval_stop \s__keyval_mark
14759 { __keyval_trim:nN { ##1 } __keyval_split_active_auxiii:w }
```

Next we test for a misplaced active equals sign in the value, if none is found `\__keyval_split_active_auxiv:w` will be called.

```

14760 \cs_new:Npn __keyval_split_active_auxiii:w ##1 ##2 \s__keyval_nil
14761 {
14762 __keyval_split_active:w ##2 \s__keyval_nil
14763 \s__keyval_mark __keyval_misplaced_equal_in_split_error:w
14764 #2 \s__keyval_mark __keyval_split_active_auxiv:w
14765 \s__keyval_stop
14766 { ##1 }
14767 }

```

This runs the last test after sanitising the remainder of the previous one. This time test for a misplaced equals sign of category 12 in the value. Finally the last auxiliary macro will be called.

```

14768 \cs_new:Npn __keyval_split_active_auxiv:w
14769 ##1 \s__keyval_nil \s__keyval_mark __keyval_misplaced_equal_in_split_error:w
14770 \s__keyval_stop \s__keyval_mark
14771 {
14772 __keyval_split_other:w ##1 \s__keyval_nil
14773 \s__keyval_mark __keyval_misplaced_equal_in_split_error:w
14774 = \s__keyval_mark __keyval_split_active_auxv:w
14775 \s__keyval_stop
14776 }

```

This last macro in this execution branch sanitises the last test, trims the value and passes it to `\__keyval_pair:nnNN`.

```

14777 \cs_new:Npn __keyval_split_active_auxv:w
14778 ##1 \s__keyval_nil \s__keyval_mark __keyval_misplaced_equal_in_split_error:w
14779 \s__keyval_stop \s__keyval_mark
14780 { __keyval_trim:nN { ##1 } __keyval_pair:nnNN }

```

*(End definition for `\__keyval_split_active_auxi:w` and others.)*

`\__keyval_clean_up_active:w`

The following is the branch taken if the key-value pair doesn't contain an active equals sign. The remainder of that test will be cleaned up by `\__keyval_clean_up_active:w` which will then split at an equals sign of category other.

```

14781 \cs_new:Npn __keyval_clean_up_active:w
14782 ##1 \s__keyval_nil \s__keyval_mark __keyval_split_active_auxi:w \s__keyval_stop
14783 {
14784 __keyval_split_other:w ##1 \s__keyval_nil
14785 \s__keyval_mark __keyval_split_other_auxi:w
14786 = \s__keyval_mark __keyval_clean_up_other:w
14787 \s__keyval_stop
14788 }

```

*(End definition for `\__keyval_clean_up_active:w`.)*

`\__keyval_split_other_auxi:w`  
`\__keyval_split_other_auxii:w`  
`\__keyval_split_other_auxiii:w`

This is executed if the key-value pair doesn't contain an active equals sign but at least one other. `##1` of `\__keyval_split_other_auxi:w` will contain the complete key name, which is trimmed and forwarded to the next auxiliary macro.

```

14789 \cs_new:Npn __keyval_split_other_auxi:w ##1 \s__keyval_stop
14790 { __keyval_trim:nN { ##1 } __keyval_split_other_auxii:w }

```

We know that the value doesn't contain misplaced active equals signs but we have to test for others.

```

14791 \cs_new:Npn __keyval_split_other_auxii:w ##1 ##2 \s__keyval_nil
14792 {
14793 __keyval_split_other:w ##2 \s__keyval_nil
14794 \s__keyval_mark __keyval_misplaced_equal_in_split_error:w
14795 = \s__keyval_mark __keyval_split_other_auxii:w
14796 \s__keyval_stop
14797 { ##1 }
14798 }

__keyval_split_other_auxii:w sanitises the test for other equals signs, trims the value and forwards it to __keyval_pair:nnNN.

14799 \cs_new:Npn __keyval_split_other_auxii:w
14800 ##1 \s__keyval_nil \s__keyval_mark __keyval_misplaced_equal_in_split_error:w
14801 \s__keyval_stop \s__keyval_mark
14802 { __keyval_trim:nN { ##1 } __keyval_pair:nnNN }

(End definition for __keyval_split_other_auxi:w, __keyval_split_other_auxii:w, and __keyval-split_other_auxii:w.)
```

\\_\_keyval\_clean\_up\_other:w \\_\_keyval\_clean\_up\_other:w is the last branch that might exist. It is called if no equals sign was found, hence the only possibilities left are a blank list element, which is to be skipped, or a lonely key. If it's no empty list element this will trim the key name and forward it to \\_\_keyval\_key:nNN.

```

14803 \cs_new:Npn __keyval_clean_up_other:w
14804 ##1 \s__keyval_nil \s__keyval_mark __keyval_split_other_auxi:w \s__keyval_stop \
14805 {
14806 __keyval_if_blank:w ##1 \s__keyval_nil \s__keyval_stop __keyval_blank_true:w
14807 \s__keyval_mark \s__keyval_stop \use:n
14808 { __keyval_trim:nN { ##1 } __keyval_key:nNN }
14809 }

(End definition for __keyval_clean_up_other:w.)
```

All these two macros do is gobble the remainder of the current other loop execution and throw an error.

```

14810 \cs_new:Npn __keyval_misplaced_equal_after_active_error:w
14811 \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
14812 \s__keyval_mark ##3 \s__keyval_nil ##4 ##5
14813 {
14814 __kernel_msg_expandable_error:nn
14815 { kernel } { misplaced-equals-sign }
14816 }
14817 \cs_new:Npn __keyval_misplaced_equal_in_split_error:w
14818 \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
14819 ##3 ##4 ##5
14820 {
14821 __kernel_msg_expandable_error:nn
14822 { kernel } { misplaced-equals-sign }
14823 }

(End definition for __keyval_misplaced_equal_after_active_error:w and __keyval_misplaced-equal_in_split_error:w.)
```

```
__keyval_end_loop_other:w
__keyval_end_loop_active:w
```

All that's left for the parsing loops are the macros which end the recursion. Both just gobble the remaining tokens of the respective loop including the next recursion call.

```
14824 \cs_new:Npn __keyval_end_loop_other:w
14825 \s__keyval_tail
14826 __keyval_split_active:w ##1 \s__keyval_nil
14827 \s__keyval_mark __keyval_split_active_auxi:w
14828 #2 \s__keyval_mark __keyval_clean_up_active:w
14829 \s__keyval_stop
14830 ##2 ##3
14831 __keyval_loop_other:NNw ##4 \s__keyval_mark
14832 { }
14833 \cs_new:Npn __keyval_end_loop_active:w
14834 \s__keyval_tail
14835 __keyval_loop_other:NNw ##1 , \s__keyval_tail ,
14836 __keyval_loop_active:NNw ##2 \s__keyval_mark
14837 { }
```

(End definition for `\_\_keyval\_end\_loop\_other:w` and `\_\_keyval\_end\_loop\_active:w`.)

The parsing loops are done, so here ends the definition of `\_\_keyval\_tmp:NN`, which will finally set up the macros.

```
14838 }
14839 \char_set_catcode_active:n { '\, }
14840 \char_set_catcode_active:n { '\= }
14841 __keyval_tmp:NN , =
14842 \group_end:
```

```
__keyval_pair:nnNN
__keyval_key:nNN
```

These macros will be called on the parsed keys and values of the key-value list. All arguments are completely trimmed. They test for blank key names and call the functions passed to `\keyval_parse:NNn` inside of `\exp_not:n` with the correct arguments.

```
14843 \cs_new:Npn __keyval_pair:nnNN #1 #2 #3 #4
14844 {
14845 __keyval_if_blank:w \s__keyval_mark #2 \s__keyval_nil \s__keyval_stop __keyval_blank_
14846 \s__keyval_mark \s__keyval_stop
14847 \exp_not:n { #4 { #2 } { #1 } }
14848 }
14849 \cs_new:Npn __keyval_key:nNN #1 #2 #3
14850 {
14851 __keyval_if_blank:w \s__keyval_mark #1 \s__keyval_nil \s__keyval_stop __keyval_blank_
14852 \s__keyval_mark \s__keyval_stop
14853 \exp_not:n { #2 { #1 } }
14854 }
```

(End definition for `\_\_keyval\_pair:nnNN` and `\_\_keyval\_key:nNN`.)

```
__keyval_if_empty:w
__keyval_if_blank:w
__keyval_if_recursion_tail:w
```

All these tests work by gobbling tokens until a certain combination is met, which makes them pretty fast. The test for a blank argument should be called with an arbitrary token following the argument. Each of these utilize the fact that the argument will contain a leading `\s__keyval\_mark`.

```
14855 \cs_new:Npn __keyval_if_empty:w #1 \s__keyval_mark \s__keyval_stop { }
14856 \cs_new:Npn __keyval_if_blank:w \s__keyval_mark #1 { __keyval_if_empty:w \s__keyval_mark
14857 \cs_new:Npn __keyval_if_recursion_tail:w \s__keyval_mark #1 \s__keyval_tail { }
```

(End definition for `\_\_keyval_if_empty:w`, `\_\_keyval_if_blank:w`, and `\_\_keyval_if_recursion\_tail:w`.)

```
_keyval_blank_true:w
_keyval_blank_key_error:w
```

These macros will be called if the tests above didn't gobble them, they execute the branching.

```
14858 \cs_new:Npn _keyval_blank_true:w \s__keyval_mark \s__keyval_stop \use:n #1 #2 #3 { }
14859 \cs_new:Npn _keyval_blank_key_error:w \s__keyval_mark \s__keyval_stop \exp_not:n #1
14860 {
14861 _kernel_msg_expandable_error:nn
14862 { kernel } { blank-key-name }
14863 }
```

(End definition for \\_keyval\_blank\_true:w and \\_keyval\_blank\_key\_error:w.)

Two messages for the low level parsing system.

```
14864 _kernel_msg_new:nnn { kernel } { misplaced-equals-sign }
14865 { Misplaced-equals-sign-in-key-value~input~\msg_line_context: }
14866 _kernel_msg_new:nnn { kernel } { blank-key-name }
14867 { Blank~key~name~in~key~value~input~\msg_line_context: }
```

```
_keyval_trim:nN
_keyval_trim_auxi:w
_keyval_trim_auxii:w
_keyval_trim_auxiii:w
_keyval_trim_auxiv:w
```

And an adapted version of \\_tl\_trim\_spaces:nn which is a bit faster for our use case, as it can strip the braces at the end. This is pretty much the same concept, so I won't comment on it here. The speed gain by using this instead of \tl\_trim\_spaces\_apply:nN is about 10 % of the total time for \keyval\_parse>NNn with one key and one key-value pair, so I think it's worth it.

```
14868 \group_begin:
14869 \cs_set_protected:Npn _keyval_tmp:n #1
14870 {
14871 \cs_new:Npn _keyval_trim:nN ##1
14872 {
14873 _keyval_trim_auxi:w
14874 ##1
14875 \s__keyval_nil
14876 \s__keyval_mark #1 { }
14877 \s__keyval_mark _keyval_trim_auxii:w
14878 _keyval_trim_auxiii:w
14879 #1 \s__keyval_nil
14880 _keyval_trim_auxiv:w
14881 \s__keyval_stop
14882 }
14883 \cs_new:Npn _keyval_trim_auxi:w ##1 \s__keyval_mark #1 ##2 \s__keyval_mark ##3
14884 {
14885 ##3
14886 _keyval_trim_auxi:w
14887 \s__keyval_mark
14888 ##2
14889 \s__keyval_mark #1 {##1}
14890 }
14891 \cs_new:Npn _keyval_trim_auxii:w _keyval_trim_auxi:w \s__keyval_mark \s__keyval_m
14892 {
14893 _keyval_trim_auxiii:w
14894 ##1
14895 }
14896 \cs_new:Npn _keyval_trim_auxiii:w ##1 #1 \s__keyval_nil ##2
14897 {
14898 ##2
14899 ##1 \s__keyval_nil
```

```

14900 __keyval_trim_auxiii:w
14901 }

```

This is the one macro which differs from the original definition.

```

14902 \cs_new:Npn __keyval_trim_auxiv:w \s_keyval_mark ##1 \s_keyval_nil ##2 \s_keyval_nil
14903 { ##3 { ##1 } }
14904 }
14905 __keyval_tmp:n { ~ }
14906 \group_end:

```

(End definition for `\_\_keyval\_trim:nN` and others.)

## 22.2 Constants and variables

`\c\_keys_code_root_str`

Various storage areas for the different data which make up keys.

```

14908 \str_const:Nn \c_keys_code_root_str { key-code~>~ }
14909 \str_const:Nn \c_keys_default_root_str { key-default~>~ }
14910 \str_const:Nn \c_keys_groups_root_str { key-groups~>~ }
14911 \str_const:Nn \c_keys_inherit_root_str { key-inherit~>~ }
14912 \str_const:Nn \c_keys_type_root_str { key-type~>~ }
14913 \str_const:Nn \c_keys_validate_root_str { key-validate~>~ }

```

(End definition for `\c\_keys_code_root_str` and others.)

`\c\_keys_props_root_str`

The prefix for storing properties.

```
14914 \str_const:Nn \c_keys_props_root_str { key-prop~>~ }
```

(End definition for `\c\_keys_props_root_str`.)

`\l\_keys_choice_int`  
`\l\_keys_choice_tl`

Publicly accessible data on which choice is being used when several are generated as a set.

```

14915 \int_new:N \l_keys_choice_int
14916 \tl_new:N \l_keys_choice_tl

```

(End definition for `\l\_keys_choice_int` and `\l\_keys_choice_tl`. These variables are documented on page 191.)

`\l\_keys_groups_clist`

Used for storing and recovering the list of groups which apply to a key: set as a comma list but at one point we have to use this for a token list recovery.

```
14917 \clist_new:N \l_keys_groups_clist
```

(End definition for `\l\_keys_groups_clist`.)

`\l\_keys_key_str`  
`\l\_keys_key_tl`

The name of a key itself: needed when setting keys. The `tl` version is deprecated but has to be handled manually.

```

14918 \str_new:N \l_keys_key_str
14919 \tl_new:N \l_keys_key_tl

```

(End definition for `\l\_keys_key_str` and `\l\_keys_key_tl`. These variables are documented on page 193.)

`\l\_keys_module_str`

The module for an entire set of keys.

```
14920 \str_new:N \l_keys_module_str
```

(End definition for `\l_keys_module_str`.)

`\l_keys_no_value_bool` A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.

14921 `\bool_new:N \l_keys_no_value_bool`

(End definition for `\l_keys_no_value_bool`.)

`\l_keys_only_known_bool` Used to track if only “known” keys are being set.

14922 `\bool_new:N \l_keys_only_known_bool`

(End definition for `\l_keys_only_known_bool`.)

`\l_keys_path_str` The “path” of the current key is stored here: this is available to the programmer and so is public. The older version is deprecated but has to be handled manually.

14923 `\str_new:N \l_keys_path_str`

14924 `\tl_new:N \l_keys_path_tl`

(End definition for `\l_keys_path_str` and `\l_keys_path_tl`. These variables are documented on page 193.)

`\l_keys_inherit_str`

14925 `\str_new:N \l_keys_inherit_str`

(End definition for `\l_keys_inherit_str`.)

`\l_keys_relative_tl` The relative path for passing keys back to the user. As this can be explicitly no-value, it must be a token list.

14926 `\tl_new:N \l_keys_relative_tl`

14927 `\tl_set:Nn \l_keys_relative_tl { \q_keys_no_value }`

(End definition for `\l_keys_relative_tl`.)

`\l_keys_property_str` The “property” begin set for a key at definition time is stored here.

14928 `\str_new:N \l_keys_property_str`

(End definition for `\l_keys_property_str`.)

`\l_keys_selective_bool` Two flags for using key groups: one to indicate that “selective” setting is active, a second to specify which type (“opt-in” or “opt-out”).

14929 `\bool_new:N \l_keys_selective_bool`

14930 `\bool_new:N \l_keys_filtered_bool`

(End definition for `\l_keys_selective_bool` and `\l_keys_filtered_bool`.)

`\l_keys_selective_seq` The list of key groups being filtered in or out during selective setting.

14931 `\seq_new:N \l_keys_selective_seq`

(End definition for `\l_keys_selective_seq`.)

`\l_keys_unused_clist` Used when setting only some keys to store those left over.

14932 `\tl_new:N \l_keys_unused_clist`

(End definition for `\l_keys_unused_clist`.)

**\l\_keys\_value\_tl** The value given for a key: may be empty if no value was given.

14933 \tl\_new:N \l\_keys\_value\_tl

(End definition for **\l\_keys\_value\_tl**. This variable is documented on page 193.)

**\l\_keys\_tmp\_bool** Scratch space.

14934 \bool\_new:N \l\_keys\_tmp\_bool  
14935 \tl\_new:N \l\_keys\_tmpa\_tl  
14936 \tl\_new:N \l\_keys\_tmpb\_tl

(End definition for **\l\_keys\_tmp\_bool**, **\l\_keys\_tmpa\_tl**, and **\l\_keys\_tmpb\_tl**.)

### 22.2.1 Internal auxiliaries

**\s\_keys\_nil** Internal scan marks.

14937 \scan\_new:N \s\_keys\_nil  
14938 \scan\_new:N \s\_keys\_mark  
14939 \scan\_new:N \s\_keys\_stop

(End definition for **\s\_keys\_nil**, **\s\_keys\_mark**, and **\s\_keys\_stop**.)

**\q\_keys\_no\_value** Internal quarks.

14940 \quark\_new:N \q\_keys\_no\_value

(End definition for **\q\_keys\_no\_value**.)

**\\_keys\_quark\_if\_no\_value\_p:N** Branching quark conditional.

14941 \\_kernel\_quark\_new\_conditional:Nn \\_keys\_quark\_if\_no\_value:N { TF }

(End definition for **\\_keys\_quark\_if\_no\_value:NTF**.)

## 22.3 The key defining mechanism

**\keys\_define:nn**

**\\_keys\_define:nnn**

**\\_keys\_define:onn**

The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

14942 \cs\_new\_protected:Npn \keys\_define:nn  
14943 { \\_keys\_define:onn \l\_keys\_module\_str }  
14944 \cs\_new\_protected:Npn \\_keys\_define:nnn #1#2#3  
14945 {  
14946 \str\_set:Nx \l\_keys\_module\_str { \\_keys\_trim\_spaces:n {#2} }  
14947 \keyval\_parse:NNn \\_keys\_define:n \\_keys\_define:nn {#3}  
14948 \str\_set:Nn \l\_keys\_module\_str {#1}  
14949 }  
14950 \cs\_generate\_variant:Nn \\_keys\_define:nnn { o }

(End definition for **\keys\_define:nn** and **\\_keys\_define:nnn**. This function is documented on page 186.)

```
__keys_define:n
__keys_define:nn
__keys_define_aux:nn
```

The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

```
14951 \cs_new_protected:Npn __keys_define:n #1
14952 {
14953 \bool_set_true:N \l__keys_no_value_bool
14954 __keys_define_aux:nn {#1} { }
14955 }
14956 \cs_new_protected:Npn __keys_define:nn #1#2
14957 {
14958 \bool_set_false:N \l__keys_no_value_bool
14959 __keys_define_aux:nn {#1} {#2}
14960 }
14961 \cs_new_protected:Npn __keys_define_aux:nn #1#2
14962 {
14963 __keys_property_find:n {#1}
14964 \cs_if_exist:cTF { \c__keys_props_root_str \l__keys_property_str }
14965 { __keys_define_code:n {#2} }
14966 {
14967 \str_if_empty:NF \l__keys_property_str
14968 {
14969 __kernel_msg_error:nnxx { kernel } { key-property-unknown }
14970 \l__keys_property_str \l__keys_path_str
14971 }
14972 }
14973 }
```

(End definition for `\__keys_define:n`, `\__keys_define:nn`, and `\__keys_define_aux:nn`.)

```
__keys_property_find:n
__keys_property_find_auxi:w
__keys_property_find_auxii:w
__keys_property_find_auxiii:w
__keys_property_find_auxiv:w
```

Searching for a property means finding the last . in the input, and storing the text before and after it. Everything is turned into strings, so there is no problem using an x-type expansion. Since `\__keys_trim_spaces:n` will turn its argument into a string anyway, this function uses `\cs_set_nopar:Npx` instead of `\tl_set:Nx` to gain some speed.

```
14974 \cs_new_protected:Npn __keys_property_find:n #1
14975 {
14976 \cs_set_nopar:Npx \l__keys_property_str { __keys_trim_spaces:n { #1 } }
14977 \exp_after:wN __keys_property_find_auxi:w \l__keys_property_str
14978 \s__keys_nil __keys_property_find_auxii:w
14979 . \s__keys_nil __keys_property_find_err:w
14980 }
14981 \cs_new_protected:Npn __keys_property_find_auxi:w #1 . #2 \s__keys_nil #3
14982 {
14983 #3 #1 \s__keys_mark #2 \s__keys_nil #3
14984 }
14985 \cs_new_protected:Npn __keys_property_find_auxii:w
14986 #1 \s__keys_mark #2 \s__keys_nil __keys_property_find_auxii:w . \s__keys_nil
14987 __keys_property_find_err:w
14988 {
14989 \cs_set_nopar:Npx \l__keys_path_str
14990 { \str_if_empty:NF \l__keys_module_str { \l__keys_module_str / } #1 }
14991 __keys_property_find_auxi:w #2 \s__keys_nil __keys_property_find_auxiii:w . \s__keys_nil
14992 __keys_property_find_auxiv:w
14993 }
14994 \cs_new_protected:Npn __keys_property_find_auxiii:w #1 \s__keys_mark
```

```

14995 {
14996 \cs_set_nopar:Npx \l_keys_path_str { \l_keys_path_str . #1 }
14997 __keys_property_find_auxi:w
14998 }
14999 \cs_new_protected:Npn __keys_property_find_auxiv:w
15000 #1 \s__keys_nil __keys_property_find_auxiii:w
15001 \s__keys_mark \s__keys_nil __keys_property_find_auxiv:w
15002 {
15003 \cs_set_nopar:Npx \l__keys_property_str { . #1 }
15004 \cs_set_nopar:Npx \l_keys_path_str
15005 { \exp_after:wN __keys_trim_spaces:n \exp_after:wN { \l_keys_path_str } }
15006 \tl_set_eq:NN \l_keys_path_tl \l_keys_path_str
15007 }
15008 \cs_new_protected:Npn __keys_property_find_err:w
15009 #1 \s__keys_nil #2 __keys_property_find_err:w
15010 {
15011 \str_clear:N \l__keys_property_str
15012 __kernel_msg_error:nnn { kernel } { key-no-property } {#1}
15013 }

```

(End definition for `\__keys_property_find:n` and others.)

`\__keys_define_code:n` Two possible cases. If there is a value for the key, then just use the function. If not, then a check to make sure there is no need for a value with the property. If there should be one then complain, otherwise execute it. There is no need to check for a : as if it was missing the earlier tests would have failed.

```

15014 \cs_new_protected:Npn __keys_define_code:n #
15015 {
15016 \bool_if:NTF \l__keys_no_value_bool
15017 {
15018 \exp_after:wN __keys_define_code:w
15019 \l__keys_property_str \s__keys_stop
15020 { \use:c { \c__keys_props_root_str \l__keys_property_str } }
15021 {
15022 __kernel_msg_error:nnxx { kernel } { key-property-requires-value }
15023 \l__keys_property_str \l_keys_path_str
15024 }
15025 }
15026 { \use:c { \c__keys_props_root_str \l__keys_property_str } {#1} }
15027 }
15028 \exp_last_unbraced:NNNN
15029 \cs_new:Npn __keys_define_code:w #1 \c_colon_str #2 \s__keys_stop
15030 { \tl_if_empty:nTF {#2} }

```

(End definition for `\__keys_define_code:n` and `\__keys_define_code:w`.)

## 22.4 Turning properties into actions

`\__keys_bool_set:Nn` Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```

15031 \cs_new_protected:Npn __keys_bool_set:Nn #1#2
15032 {
15033 \bool_if_exist:NF #1 { \bool_new:N #1 }
15034 __keys_choice_make:

```

```

15035 __keys_cmd_set:nx { \l_keys_path_str / true }
15036 { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
15037 __keys_cmd_set:nx { \l_keys_path_str / false }
15038 { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
15039 __keys_cmd_set:nn { \l_keys_path_str / unknown }
15040 {
15041 __kernel_msg_error:nnx { kernel } { boolean-values-only }
15042 \l_keys_key_str
15043 }
15044 __keys_default_set:n { true }
15045 }
15046 \cs_generate_variant:Nn __keys_bool_set:Nn { c }

(End definition for __keys_bool_set:Nn.)

```

\\_\_keys\_bool\_set\_inverse:Nn  
\\_\_keys\_bool\_set\_inverse:cn Inverse boolean setting is much the same.

```

15047 \cs_new_protected:Npn __keys_bool_set_inverse:Nn #1#2
15048 {
15049 \bool_if_exist:NF #1 { \bool_new:N #1 }
15050 __keys_choice_make:
15051 __keys_cmd_set:nx { \l_keys_path_str / true }
15052 { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
15053 __keys_cmd_set:nx { \l_keys_path_str / false }
15054 { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
15055 __keys_cmd_set:nn { \l_keys_path_str / unknown }
15056 {
15057 __kernel_msg_error:nnx { kernel } { boolean-values-only }
15058 \l_keys_key_str
15059 }
15060 __keys_default_set:n { true }
15061 }
15062 \cs_generate_variant:Nn __keys_bool_set_inverse:Nn { c }

(End definition for __keys_bool_set_inverse:Nn.)

```

\\_\_keys\_choice\_make:  
\\_\_keys\_multichoice\_make:  
\\_\_keys\_choice\_make:N  
\\_\_keys\_choice\_make\_aux:N To make a choice from a key, two steps: set the code, and set the unknown key. As multichoice and choices are essentially the same bar one function, the code is given together.

```

15063 \cs_new_protected:Npn __keys_choice_make:
15064 { __keys_choice_make:N __keys_choice_find:n }
15065 \cs_new_protected:Npn __keys_multichoice_make:
15066 { __keys_choice_make:N __keys_multichoice_find:n }
15067 \cs_new_protected:Npn __keys_choice_make:N #
15068 {
15069 \cs_if_exist:cTF
15070 { \c_keys_type_root_str __keys_parent:o \l_keys_path_str }
15071 {
15072 \str_if_eq:vnTF
15073 { \c_keys_type_root_str __keys_parent:o \l_keys_path_str }
15074 { choice }
15075 {
15076 __kernel_msg_error:nnxx { kernel } { nested-choice-key }
15077 \l_keys_path_tl { __keys_parent:o \l_keys_path_str }
15078 }

```

```

15079 { __keys_choice_make_aux:N #1 }
15080 }
15081 { __keys_choice_make_aux:N #1 }
15082 }
15083 \cs_new_protected:Npn __keys_choice_make_aux:N #1
15084 {
15085 \cs_set_nopar:cpxn { \c__keys_type_root_str \l_keys_path_str }
15086 { choice }
15087 __keys_cmd_set:nn \l_keys_path_str { #1 {##1} }
15088 __keys_cmd_set:nn { \l_keys_path_str / unknown }
15089 {
15090 __kernel_msg_error:nxxx { kernel } { key-choice-unknown }
15091 \l_keys_path_str {##1}
15092 }
15093 }

```

(End definition for `\__keys_choice_make:` and others.)

`\__keys_choices_make:nn`  
`\__keys_multichoices_make:nn`  
`\__keys_choices_make:Nnn`

Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```

15094 \cs_new_protected:Npn __keys_choices_make:nn
15095 { __keys_choices_make:Nnn __keys_choice_make: }
15096 \cs_new_protected:Npn __keys_multichoices_make:nn
15097 { __keys_choices_make:Nnn __keys_multichoices_make: }
15098 \cs_new_protected:Npn __keys_choices_make:Nnn #1#2#3
15099 {
15100 #1
15101 \int_zero:N \l_keys_choice_int
15102 \clist_map_inline:nn {#2}
15103 {
15104 \int_incr:N \l_keys_choice_int
15105 __keys_cmd_set:nx
15106 { \l_keys_path_str / __keys_trim_spaces:n {##1} }
15107 {
15108 \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
15109 \int_set:Nn \exp_not:N \l_keys_choice_int
15110 { \int_use:N \l_keys_choice_int }
15111 \exp_not:n {#3}
15112 }
15113 }
15114 }

```

(End definition for `\__keys_choices_make:nn`, `\__keys_multichoices_make:nn`, and `\__keys_choices_make:Nnn`.)

`\__keys_cmd_set:nn`  
`\__keys_cmd_set:nx`  
`\__keys_cmd_set:Vn`  
`\__keys_cmd_set:Vo`

Setting the code for a key first logs if appropriate that we are defining a new key, then saves the code.

```

15115 \cs_new_protected:Npn __keys_cmd_set:nn #1#2
15116 { \cs_set_protected:cpxn { \c__keys_code_root_str #1 } ##1 {#2} }
15117 \cs_generate_variant:Nn __keys_cmd_set:nn { nx , Vn , Vo }

```

(End definition for `\__keys_cmd_set:nn`.)

\\_\\_keys\\_cs\\_set:NNpn  
\\_\\_keys\\_cs\\_set:Ncpn

Creating control sequences is a bit more tricky than other cases as we need to pick up the p argument. To make the internals look clearer, the trailing n argument here is just for appearance.

```

15118 \cs_new_protected:Npn __keys_cs_set:NNpn #1#2#3#
15119 {
15120 \cs_set_protected:cpx { \c__keys_code_root_str \l_keys_path_str } ##1
15121 { #1 \exp_not:N #2 \exp_not:n {#3} {##1} }
15122 \use_none:n
15123 }
15124 \cs_generate_variant:Nn __keys_cs_set:NNpn { Nc }
```

(End definition for \\_\\_keys\\_cs\\_set:NNpn.)

\\_\\_keys\_default\_set:n

Setting a default value is easy. These are stored using \cs\_set:cpx as this avoids any worries about whether a token list exists.

```

15125 \cs_new_protected:Npn __keys_default_set:n #1
15126 {
15127 \tl_if_empty:nTF {#1}
15128 {
15129 \cs_set_eq:cN
15130 { \c__keys_default_root_str \l_keys_path_str }
15131 \tex_undefined:D
15132 }
15133 {
15134 \cs_set_nopar:cpx
15135 { \c__keys_default_root_str \l_keys_path_str }
15136 { \exp_not:n {#1} }
15137 __keys_value_requirement:nn { required } { false }
15138 }
15139 }
```

(End definition for \\_\\_keys\\_default\\_set:n.)

\\_\\_keys\_groups\_set:n

Assigning a key to one or more groups uses comma lists. As the list of groups only exists if there is anything to do, the setting is done using a scratch list. For the usual grouping reasons we use the low-level approach to undefining a list. We also use the low-level approach for the other case to avoid tripping up the check-declarations code.

```

15140 \cs_new_protected:Npn __keys_groups_set:n #1
15141 {
15142 \clist_set:Nn \l__keys_groups_clist {#1}
15143 \clist_if_empty:NTF \l__keys_groups_clist
15144 {
15145 \cs_set_eq:cN { \c__keys_groups_root_str \l_keys_path_str }
15146 \tex_undefined:D
15147 }
15148 {
15149 \cs_set_eq:cN { \c__keys_groups_root_str \l_keys_path_str }
15150 \l__keys_groups_clist
15151 }
15152 }
```

(End definition for \\_\\_keys\\_groups\\_set:n.)

\\_\\_keys\\_inherit:n Inheritance means ignoring anything already said about the key: zap the lot and set up.

```

15153 \cs_new_protected:Npn __keys_inherit:n #1
15154 {
15155 __keys_undefine:
15156 \cs_set_nopar:cpn { \c__keys_inherit_root_str \l__keys_path_str } {#1}
15157 }
```

(End definition for \\_\\_keys\\_inherit:n.)

\\_\\_keys\\_initialise:n A set up for initialisation: just run the code if it exists.

```

15158 \cs_new_protected:Npn __keys_initialise:n #1
15159 {
15160 \cs_if_exist:cT
15161 { \c__keys_inherit_root_str __keys_parent:o \l__keys_path_str }
15162 { __keys_execute_inherit: }
15163 {
15164 \str_clear:N \l__keys_inherit_str
15165 \cs_if_exist:cT { \c__keys_code_root_str \l__keys_path_str }
15166 { __keys_execute:nn \l__keys_path_str {#1} }
15167 }
15168 }
```

(End definition for \\_\\_keys\\_initialise:n.)

\\_\\_keys\\_meta\\_make:n To create a meta-key, simply set up to pass data through.

```

15169 \cs_new_protected:Npn __keys_meta_make:n #1
15170 {
15171 __keys_cmd_set:Vo \l__keys_path_str
15172 {
15173 \exp_after:wN \keys_set:nn \exp_after:wN { \l__keys_module_str } {#1}
15174 }
15175 }
15176 \cs_new_protected:Npn __keys_meta_make:nn #1#2
15177 { __keys_cmd_set:Vn \l__keys_path_str { \keys_set:nn {#1} {#2} } }
```

(End definition for \\_\\_keys\\_meta\\_make:n and \\_\\_keys\\_meta\\_make:nn.)

\\_\\_keys\\_prop\\_put:Nn \\_\\_keys\\_prop\\_put:cn Much the same as other variables, but needs a dedicated auxiliary.

```

15178 \cs_new_protected:Npn __keys_prop_put:Nn #1#2
15179 {
15180 \prop_if_exist:NF #1 { \prop_new:N #1 }
15181 \exp_after:wN __keys_find_key_module:wNN \l__keys_path_str \s__keys_stop
15182 \l__keys_tma_t1 \l__keys_tmpb_t1
15183 __keys_cmd_set:nx \l__keys_path_str
15184 {
15185 \exp_not:c { prop_ #2 put:Nnn }
15186 \exp_not:N #1
15187 { \l__keys_tmpb_t1 }
15188 \exp_not:n { {##1} }
15189 }
15190 }
15191 \cs_generate_variant:Nn __keys_prop_put:Nn { c }
```

(End definition for \\_\\_keys\\_prop\\_put:Nn.)

\\_\_keys\_undefine: Undefining a key has to be done without \cs\_undefine:c as that function acts globally.

```
15192 \cs_new_protected:Npn __keys_undefine:
15193 {
15194 \clist_map_inline:nn
15195 { code , default , groups , inherit , type , validate }
15196 {
15197 \cs_set_eq:cN
15198 { \tl_use:c { c__keys_ ##1 _root_str } \l_keys_path_str }
15199 \tex_undefined:D
15200 }
15201 }
```

(End definition for \\_\_keys\_undefine:.)

\\_\_keys\_value\_requirement:nn  
\\_\_keys\_validate\_forbidden:  
\\_\_keys\_validate\_required:

Validating key input is done using a second function which runs before the main key code. Setting that up means setting it equal to a generic stub which does the check. This approach makes the lookup very fast at the cost of one additional csname per key that needs it. The cleanup here has to know the structure of the following code.

```
15202 \cs_new_protected:Npn __keys_value_requirement:nn #1#
15203 {
15204 \str_case:nnF {#2}
15205 {
15206 { true }
15207 {
15208 \cs_set_eq:cc
15209 { \c__keys_validate_root_str \l_keys_path_str }
15210 { __keys_validate_ #1 : }
15211 }
15212 { false }
15213 {
15214 \cs_if_eq:ccT
15215 { \c__keys_validate_root_str \l_keys_path_str }
15216 { __keys_validate_ #1 : }
15217 {
15218 \cs_set_eq:cN
15219 { \c__keys_validate_root_str \l_keys_path_str }
15220 \tex_undefined:D
15221 }
15222 }
15223 }
15224 {
15225 __kernel_msg_error:nnx { kernel }
15226 { key-property-boolean-values-only }
15227 { .value_ #1 :n }
15228 }
15229 }
15230 \cs_new_protected:Npn __keys_validate_forbidden:
15231 {
15232 \bool_if:NF \l__keys_no_value_bool
15233 {
15234 __kernel_msg_error:nxxx { kernel } { value-forbidden }
15235 \l_keys_path_str \l_keys_value_tl
15236 \use_none:nnn
15237 }
```

```

15238 }
15239 \cs_new_protected:Npn __keys_validate_required:
15240 {
15241 \bool_if:NT \l__keys_no_value_bool
15242 {
15243 __kernel_msg_error:n { kernel } { value-required }
15244 \l_keys_path_str
15245 \use_none:n
15246 }
15247 }

```

(End definition for `\__keys_value_requirement:nn`, `\__keys_validate_forbidden:`, and `\__keys_validate_required:..`)

`\__keys_variable_set:NnnN` `\__keys_variable_set:cnnN`  
`\__keys_variable_set_required:NnnN` `\__keys_variable_set_required:cnnN`

```

15248 \cs_new_protected:Npn __keys_variable_set:NnnN #1#2#3#4
15249 {
15250 \use:c { #2_if_exist:NF } #1 { \use:c { #2 _new:N } #1 }
15251 __keys_cmd_set:nx \l_keys_path_str
15252 {
15253 \exp_not:c { #2 _ #3 set:N #4 }
15254 \exp_not:N #1
15255 \exp_not:n { {##1} }
15256 }
15257 }
15258 \cs_generate_variant:Nn __keys_variable_set:NnnN { c }
15259 \cs_new_protected:Npn __keys_variable_set_required:NnnN #1#2#3#4
15260 {
15261 __keys_variable_set:NnnN #1 {#2} {#3} #4
15262 __keys_value_requirement:nn { required } { true }
15263 }
15264 \cs_generate_variant:Nn __keys_variable_set_required:NnnN { c }

```

(End definition for `\__keys_variable_set:NnnN` and `\__keys_variable_set_required:NnnN`.)

## 22.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have “normal” argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

`.bool_set:N` One function for this.

```

15265 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set:N } #1
15266 { __keys_bool_set:Nn #1 { } }
15267 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set:c } #1
15268 { __keys_bool_set:cn {#1} { } }
15269 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:N } #1
15270 { __keys_bool_set:Nn #1 { g } }
15271 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:c } #1
15272 { __keys_bool_set:cn {#1} { g } }

```

(End definition for `.bool_set:N` and `.bool_gset:N`. These functions are documented on page 187.)

`.bool_set_inverse:N` One function for this.

```
15273 \cs_new_protected:cpn { \c_keys_props_root_str .bool_set_inverse:N } #1
15274 { _keys_bool_set_inverse:Nn #1 { } }
15275 \cs_new_protected:cpn { \c_keys_props_root_str .bool_set_inverse:c } #1
15276 { _keys_bool_set_inverse:cn {#1} { } }
15277 \cs_new_protected:cpn { \c_keys_props_root_str .bool_gset_inverse:N } #1
15278 { _keys_bool_set_inverse:Nn #1 { g } }
15279 \cs_new_protected:cpn { \c_keys_props_root_str .bool_gset_inverse:c } #1
15280 { _keys_bool_set_inverse:cn {#1} { g } }
```

(End definition for `.bool_set_inverse:N` and `.bool_gset_inverse:N`. These functions are documented on page 187.)

`.choice:` Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```
15281 \cs_new_protected:cpn { \c_keys_props_root_str .choice: }
15282 { _keys_choice_make: }
```

(End definition for `.choice:`. This function is documented on page 187.)

`.choices:nn` For auto-generation of a series of mutually-exclusive choices. Here, #1 consists of two separate arguments, hence the slightly odd-looking implementation.

```
15283 \cs_new_protected:cpn { \c_keys_props_root_str .choices:nn } #1
15284 { _keys_choices_make:nn #1 }
15285 \cs_new_protected:cpn { \c_keys_props_root_str .choices:Vn } #1
15286 { \exp_args:NV _keys_choices_make:nn #1 }
15287 \cs_new_protected:cpn { \c_keys_props_root_str .choices:on } #1
15288 { \exp_args:No _keys_choices_make:nn #1 }
15289 \cs_new_protected:cpn { \c_keys_props_root_str .choices:xn } #1
15290 { \exp_args:Nx _keys_choices_make:nn #1 }
```

(End definition for `.choices:nn`. This function is documented on page 187.)

`.code:n` Creating code is simply a case of passing through to the underlying `set` function.

```
15291 \cs_new_protected:cpn { \c_keys_props_root_str .code:n } #1
15292 { _keys_cmd_set:nn \l_keys_path_str {#1} }
```

(End definition for `.code:n`. This function is documented on page 187.)

`.clist_set:N`

```
15293 \cs_new_protected:cpn { \c_keys_props_root_str .clist_set:N } #1
15294 { _keys_variable_set:NnnN #1 { clist } { } n }
.clist_gset:N
```

`.clist_gset:c`

```
15295 \cs_new_protected:cpn { \c_keys_props_root_str .clist_set:c } #1
15296 { _keys_variable_set:cnnN {#1} { clist } { } n }
15297 \cs_new_protected:cpn { \c_keys_props_root_str .clist_gset:N } #1
15298 { _keys_variable_set:NnnN #1 { clist } { g } n }
15299 \cs_new_protected:cpn { \c_keys_props_root_str .clist_gset:c } #1
15300 { _keys_variable_set:cnnN {#1} { clist } { g } n }
```

(End definition for `.clist_set:N` and `.clist_gset:N`. These functions are documented on page 187.)

```

.cs_set:Np
.cs_set:cp
.cs_set_protected:Np
.cs_set_protected:cp
.cs_gset:Np
.cs_gset:cp
.cs_gset_protected:Np
.cs_gset_protected:cp

15301 \cs_new_protected:cpn { \c_keys_props_root_str .cs_set:Np } #1
15302 { _keys_cs_set:NNpn \cs_set:Npn #1 { } }
15303 \cs_new_protected:cpn { \c_keys_props_root_str .cs_set:cp } #1
15304 { _keys_cs_set:Ncpn \cs_set:Npn #1 { } }
15305 \cs_new_protected:cpn { \c_keys_props_root_str .cs_set_protected:Np } #1
15306 { _keys_cs_set:NNpn \cs_set_protected:Npn #1 { } }
15307 \cs_new_protected:cpn { \c_keys_props_root_str .cs_set_protected:cp } #1
15308 { _keys_cs_set:Ncpn \cs_set_protected:Npn #1 { } }
15309 \cs_new_protected:cpn { \c_keys_props_root_str .cs_gset:Np } #1
15310 { _keys_cs_set:NNpn \cs_gset:Npn #1 { } }
15311 \cs_new_protected:cpn { \c_keys_props_root_str .cs_gset:cp } #1
15312 { _keys_cs_set:Ncpn \cs_gset:Npn #1 { } }
15313 \cs_new_protected:cpn { \c_keys_props_root_str .cs_gset_protected:Np } #1
15314 { _keys_cs_set:NNpn \cs_gset_protected:Npn #1 { } }
15315 \cs_new_protected:cpn { \c_keys_props_root_str .cs_gset_protected:cp } #1
15316 { _keys_cs_set:Ncpn \cs_gset_protected:Npn #1 { } }

```

(End definition for `.cs_set:Np` and others. These functions are documented on page 187.)

`.default:n` Expansion is left to the internal functions.

```

.default:V
15317 \cs_new_protected:cpn { \c_keys_props_root_str .default:n } #1
.default:o
15318 { _keys_default_set:n {#1} }
.default:x
15319 \cs_new_protected:cpn { \c_keys_props_root_str .default:V } #1
15320 { \exp_args:NV _keys_default_set:n #1 }
15321 \cs_new_protected:cpn { \c_keys_props_root_str .default:o } #1
15322 { \exp_args:No _keys_default_set:n {#1} }
15323 \cs_new_protected:cpn { \c_keys_props_root_str .default:x } #1
15324 { \exp_args:Nx _keys_default_set:n {#1} }

```

(End definition for `.default:n`. This function is documented on page 188.)

`.dim_set:N` Setting a variable is very easy: just pass the data along.

```

.dim_set:c
15325 \cs_new_protected:cpn { \c_keys_props_root_str .dim_set:N } #1
.dim_gset:N
15326 { _keys_variable_set_required:NnnN #1 { dim } { } n }
.dim_gset:c
15327 \cs_new_protected:cpn { \c_keys_props_root_str .dim_set:c } #1
15328 { _keys_variable_set_required:cnnN {#1} { dim } { } n }
15329 \cs_new_protected:cpn { \c_keys_props_root_str .dim_gset:N } #1
15330 { _keys_variable_set_required:NnnN #1 { dim } { g } n }
15331 \cs_new_protected:cpn { \c_keys_props_root_str .dim_gset:c } #1
15332 { _keys_variable_set_required:cnnN {#1} { dim } { g } n }

```

(End definition for `.dim_set:N` and `.dim_gset:N`. These functions are documented on page 188.)

`.fp_set:N` Setting a variable is very easy: just pass the data along.

```

.fp_set:c
15333 \cs_new_protected:cpn { \c_keys_props_root_str .fp_set:N } #1
.fp_gset:N
15334 { _keys_variable_set_required:NnnN #1 { fp } { } n }
.fp_gset:c
15335 \cs_new_protected:cpn { \c_keys_props_root_str .fp_set:c } #1
15336 { _keys_variable_set_required:cnnN {#1} { fp } { } n }
15337 \cs_new_protected:cpn { \c_keys_props_root_str .fp_gset:N } #1
15338 { _keys_variable_set_required:NnnN #1 { fp } { g } n }
15339 \cs_new_protected:cpn { \c_keys_props_root_str .fp_gset:c } #1
15340 { _keys_variable_set_required:cnnN {#1} { fp } { g } n }

```

(End definition for `.fp_set:N` and `.fp_gset:N`. These functions are documented on page 188.)

**.groups:n** A single property to create groups of keys.

```
15341 \cs_new_protected:cpn { \c_keys_props_root_str .groups:n } #1
15342 { _keys_groups_set:n {#1} }
```

(End definition for **.groups:n**. This function is documented on page 188.)

**.inherit:n** Nothing complex: only one variant at the moment!

```
15343 \cs_new_protected:cpn { \c_keys_props_root_str .inherit:n } #1
15344 { _keys_inherit:n {#1} }
```

(End definition for **.inherit:n**. This function is documented on page 188.)

**.initial:n** The standard hand-off approach.

```
.initial:V 15345 \cs_new_protected:cpn { \c_keys_props_root_str .initial:n } #1
.initial:o 15346 { _keys_initialise:n {#1} }
.initial:x 15347 \cs_new_protected:cpn { \c_keys_props_root_str .initial:V } #1
15348 { \exp_args:NV _keys_initialise:n #1 }
15349 \cs_new_protected:cpn { \c_keys_props_root_str .initial:o } #1
15350 { \exp_args:No _keys_initialise:n {#1} }
15351 \cs_new_protected:cpn { \c_keys_props_root_str .initial:x } #1
15352 { \exp_args:Nx _keys_initialise:n {#1} }
```

(End definition for **.initial:n**. This function is documented on page 189.)

**.int\_set:N** Setting a variable is very easy: just pass the data along.

```
.int_set:c 15353 \cs_new_protected:cpn { \c_keys_props_root_str .int_set:N } #1
15354 { _keys_variable_set_required:NnnN #1 { int } { } n }
.int_gset:c 15355 \cs_new_protected:cpn { \c_keys_props_root_str .int_set:c } #1
15356 { _keys_variable_set_required:cnnN {#1} { int } { } n }
15357 \cs_new_protected:cpn { \c_keys_props_root_str .int_gset:N } #1
15358 { _keys_variable_set_required:NnnN #1 { int } { g } n }
15359 \cs_new_protected:cpn { \c_keys_props_root_str .int_gset:c } #1
15360 { _keys_variable_set_required:cnnN {#1} { int } { g } n }
```

(End definition for **.int\_set:N** and **.int\_gset:N**. These functions are documented on page 189.)

**.meta:n** Making a meta is handled internally.

```
15361 \cs_new_protected:cpn { \c_keys_props_root_str .meta:n } #1
15362 { _keys_meta_make:n {#1} }
```

(End definition for **.meta:n**. This function is documented on page 189.)

**.meta:nn** Meta with path: potentially lots of variants, but for the moment no so many defined.

```
15363 \cs_new_protected:cpn { \c_keys_props_root_str .meta:nn } #1
15364 { _keys_meta_make:nn #1 }
```

(End definition for **.meta:nn**. This function is documented on page 189.)

**.multichoice:** The same idea as **.choice:** and **.choices:nn**, but where more than one choice is allowed.

```
.multichoices:nn 15365 \cs_new_protected:cpn { \c_keys_props_root_str .multichoice: }
15366 { _keys_multichoice_make: }
.multichoices:Vn 15367 \cs_new_protected:cpn { \c_keys_props_root_str .multichoices:nn } #1
15368 { _keys_multichoices_make:nn #1 }
15369 \cs_new_protected:cpn { \c_keys_props_root_str .multichoices:Vn } #1
15370 { \exp_args:NV _keys_multichoices_make:nn #1 }
```

```

15371 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:on } #1
15372 { \exp_args:No __keys_multichoices_make:nn #1 }
15373 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:xn } #1
15374 { \exp_args:Nx __keys_multichoices_make:nn #1 }

```

(End definition for `.multichoice:` and `.multichoices:nn`. These functions are documented on page [189](#).)

**.muskip\_set:N** Setting a variable is very easy: just pass the data along.

```

15375 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:N } #1
15376 { __keys_variable_set_required:NnnN #1 { muskip } { } n }
15377 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:c } #1
15378 { __keys_variable_set_required:cnnN {#1} { muskip } { } n }
15379 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:N } #1
15380 { __keys_variable_set_required:NnnN #1 { muskip } { g } n }
15381 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:c } #1
15382 { __keys_variable_set_required:cnnN {#1} { muskip } { g } n }

```

(End definition for `.muskip_set:N` and `.muskip_gset:N`. These functions are documented on page [189](#).)

**.prop\_put:N** Setting a variable is very easy: just pass the data along.

```

15383 \cs_new_protected:cpn { \c__keys_props_root_str .prop_put:N } #1
15384 { __keys_prop_put:Nn #1 { } }
15385 \cs_new_protected:cpn { \c__keys_props_root_str .prop_put:c } #1
15386 { __keys_prop_put:cn {#1} { } }
15387 \cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:N } #1
15388 { __keys_prop_put:Nn #1 { g } }
15389 \cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:c } #1
15390 { __keys_prop_put:cn {#1} { g } }

```

(End definition for `.prop_put:N` and `.prop_gput:N`. These functions are documented on page [189](#).)

**.skip\_set:N** Setting a variable is very easy: just pass the data along.

```

15391 \cs_new_protected:cpn { \c__keys_props_root_str .skip_set:N } #1
15392 { __keys_variable_set_required:NnnN #1 { skip } { } n }
15393 \cs_new_protected:cpn { \c__keys_props_root_str .skip_set:c } #1
15394 { __keys_variable_set_required:cnnN {#1} { skip } { } n }
15395 \cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:N } #1
15396 { __keys_variable_set_required:NnnN #1 { skip } { g } n }
15397 \cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:c } #1
15398 { __keys_variable_set_required:cnnN {#1} { skip } { g } n }

```

(End definition for `.skip_set:N` and `.skip_gset:N`. These functions are documented on page [190](#).)

**.tl\_set:N** Setting a variable is very easy: just pass the data along.

```

15399 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set:N } #1
15400 { __keys_variable_set:NnnN #1 { tl } { } n }
15401 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set:c } #1
15402 { __keys_variable_set:cnnN {#1} { tl } { } n }
15403 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:N } #1
15404 { __keys_variable_set:NnnN #1 { tl } { } x }
15405 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:c } #1
15406 { __keys_variable_set:cnnN {#1} { tl } { } x }
15407 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:N } #1
15408 { __keys_variable_set:NnnN #1 { tl } { g } n }

```

```

15409 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:c } #1
15410 { __keys_variable_set:cnnN {#1} { tl } { g } n }
15411 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:N } #1
15412 { __keys_variable_set:NnnN #1 { tl } { g } x }
15413 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:c } #1
15414 { __keys_variable_set:cnnN {#1} { tl } { g } x }

```

(End definition for `.tl_set:N` and others. These functions are documented on page 190.)

**.undefine:** Another simple wrapper.

```

15415 \cs_new_protected:cpn { \c__keys_props_root_str .undefine: }
15416 { __keys_undefine: }

```

(End definition for `.undefine:`. This function is documented on page 190.)

**.value\_forbidden:n** These are very similar, so both call the same function.

```

.bvalue_forbidden:n 15417 \cs_new_protected:cpn { \c__keys_props_root_str .value_forbidden:n } #1
.bvalue_required:n 15418 { __keys_value_requirement:nn { forbidden } {#1} }
15419 \cs_new_protected:cpn { \c__keys_props_root_str .value_required:n } #1
15420 { __keys_value_requirement:nn { required } {#1} }

```

(End definition for `.value_forbidden:n` and `.value_required:n`. These functions are documented on page 190.)

## 22.6 Setting keys

**\keys\_set:nn** A simple wrapper allowing for nesting.

```

\keys_set:nV 15421 \cs_new_protected:Npn \keys_set:nn #1#2
\keys_set:nv 15422 {
\keys_set:no 15423 \use:x
__keys_set:nn 15424 {
__keys_set:nnn 15425 \bool_set_false:N \exp_not:N \l__keys_only_known_bool
__keys_set:nnn 15426 \bool_set_false:N \exp_not:N \l__keys_filtered_bool
__keys_set:nnn 15427 \bool_set_false:N \exp_not:N \l__keys_selective_bool
__keys_set:nnn 15428 \tl_set:Nn \exp_not:N \l__keys_relative_tl
__keys_set:nnn 15429 { \exp_not:N \q__keys_no_value }
__keys_set:nnn 15430 __keys_set:nn \exp_not:n { {#1} {#2} }
__keys_set:nnn 15431 \bool_if:NT \l__keys_only_known_bool
__keys_set:nnn 15432 { \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
__keys_set:nnn 15433 \bool_if:NT \l__keys_filtered_bool
__keys_set:nnn 15434 { \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
__keys_set:nnn 15435 \bool_if:NT \l__keys_selective_bool
__keys_set:nnn 15436 { \bool_set_true:N \exp_not:N \l__keys_selective_bool }
__keys_set:nnn 15437 \tl_set:Mn \exp_not:N \l__keys_relative_tl
__keys_set:nnn 15438 { \exp_not:o \l__keys_relative_tl }
__keys_set:nnn 15439 }
__keys_set:nnn 15440 }
__keys_set:nnn 15441 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
__keys_set:nnn 15442 \cs_new_protected:Npn __keys_set:nn #1#2
__keys_set:nnn 15443 { \exp_args:No __keys_set:nnn \l__keys_module_str {#1} {#2} }
__keys_set:nnn 15444 \cs_new_protected:Npn __keys_set:nnn #1#2#3
__keys_set:nnn 15445 {
__keys_set:nnn 15446 \str_set:Nx \l__keys_module_str { __keys_trim_spaces:n {#2} }
__keys_set:nnn 15447 \keyval_parse:NNn __keys_set_keyval:n __keys_set_keyval:nn {#3}
__keys_set:nnn 15448 \str_set:Nn \l__keys_module_str {#1}
__keys_set:nnn 15449 }

```

(End definition for `\keys_set:nn`, `\_keys_set:nn`, and `\__keys_set:nnn`. This function is documented on page [193](#).)

`\keys_set_known:nnN` Setting known keys simply means setting the appropriate flag, then running the standard code. To allow for nested setting, any existing value of `\l_keys_unused_clist` is saved on the stack and reset afterwards. Note that for speed/simplicity reasons we use a `tl` operation to set the `clist` here!

```

\keys_set_known:nnN
\keys_set_known:nVnN
\keys_set_known:nvnN
\keys_set_known:nonN
_keys_set_known:nnnnN
 \keys_set_known:nn
 \keys_set_known:nV
 \keys_set_known:nv
 \keys_set_known:no
_keys_set_known:nnn

15450 \cs_new_protected:Npn \keys_set_known:nnN #1#2#3
15451 {
15452 \exp_args:No __keys_set_known:nnnnN
15453 \l__keys_unused_clist \q__keys_no_value {#1} {#2} #3
15454 }
15455 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
15456 \cs_new_protected:Npn \keys_set_known:nnnnN #1#2#3#4
15457 {
15458 \exp_args:No __keys_set_known:nnnnN
15459 \l__keys_unused_clist {#3} {#1} {#2} #4
15460 }
15461 \cs_generate_variant:Nn \keys_set_known:nnnnN { nV , nv , no }
15462 \cs_new_protected:Npn _keys_set_known:nnnnN #1#2#3#4#5
15463 {
15464 \clist_clear:N \l__keys_unused_clist
15465 __keys_set_known:nnn {#2} {#3} {#4}
15466 __kernel_tl_set:Nx #5 { \exp_not:o \l__keys_unused_clist }
15467 \tl_set:Nn \l__keys_unused_clist {#1}
15468 }
15469 \cs_new_protected:Npn \keys_set_known:nn #1#2
15470 { __keys_set_known:nnn \q__keys_no_value {#1} {#2} }
15471 \cs_generate_variant:Nn \keys_set_known:nn { nV , nv , no }
15472 \cs_new_protected:Npn _keys_set_known:nnn #1#2#3
15473 {
15474 \use:x
15475 {
15476 \bool_set_true:N \exp_not:N \l__keys_only_known_bool
15477 \bool_set_false:N \exp_not:N \l__keys_filtered_bool
15478 \bool_set_false:N \exp_not:N \l__keys_selective_bool
15479 \tl_set:Nn \exp_not:N \l__keys_relative_tl { \exp_not:n {#1} }
15480 __keys_set:nn \exp_not:n { {#2} {#3} }
15481 \bool_if:NF \l__keys_only_known_bool
15482 { \bool_set_false:N \exp_not:N \l__keys_only_known_bool }
15483 \bool_if:NT \l__keys_filtered_bool
15484 { \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
15485 \bool_if:NT \l__keys_selective_bool
15486 { \bool_set_true:N \exp_not:N \l__keys_selective_bool }
15487 \tl_set:Nn \exp_not:N \l__keys_relative_tl
15488 { \exp_not:o \l__keys_relative_tl }
15489 }
15490 }

```

(End definition for `\keys_set_known:nnN` and others. These functions are documented on page [194](#).)

The idea of setting keys in a selective manner again uses flags wrapped around the basic code. The comments on `\keys_set_known:nnN` also apply here. We have a bit more shuffling to do to keep everything nestable.

```
\keys_set_filter:nnnN
\keys_set_filter:nnVN
\keys_set_filter:nnvN
\keys_set_filter:nnoN
\keys_set_filter:nnnnN
\keys_set_filter:nnVnN
\keys_set_filter:nnvnN
\keys_set_filter:nnonN
_keys_set_filter:nnnnnnN
 \keys_set_filter:nnN
 \keys_set_filter:nnV
 \keys_set_filter:nnv
 \keys_set_filter:nnn
```

```

15491 \cs_new_protected:Npn \keys_set_filter:nnnN #1#2#3#4
15492 {
15493 \exp_args:No __keys_set_filter:nnnnN
15494 \l__keys_unused_clist
15495 \q__keys_no_value {#1} {#2} {#3} #4
15496 }
15497 \cs_generate_variant:Nn \keys_set_filter:nnnn { nnV , nnv , nno }
15498 \cs_new_protected:Npn \keys_set_filter:nnnnN #1#2#3#4#5
15499 {
15500 \exp_args:No __keys_set_filter:nnnnN
15501 \l__keys_unused_clist {#4} {#1} {#2} {#3} #5
15502 }
15503 \cs_generate_variant:Nn \keys_set_filter:nnnn { nnV , nnv , nno }
15504 \cs_new_protected:Npn __keys_set_filter:nnnnN #1#2#3#4#5#6
15505 {
15506 \clist_clear:N \l__keys_unused_clist
15507 __keys_set_filter:nnnn {#2} {#3} {#4} {#5}
15508 __kernel_tl_set:Nx #6 { \exp_not:o \l__keys_unused_clist }
15509 \tl_set:Nn \l__keys_unused_clist {#1}
15510 }
15511 \cs_new_protected:Npn \keys_set_filter:nnn #1#2#3
15512 {__keys_set_filter:nnnn \q__keys_no_value {#1} {#2} {#3} }
15513 \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
15514 \cs_new_protected:Npn __keys_set_filter:nnnn #1#2#3#4
15515 {
15516 \use:x
15517 {
15518 \bool_set_false:N \exp_not:N \l__keys_only_known_bool
15519 \bool_set_true:N \exp_not:N \l__keys_filtered_bool
15520 \bool_set_true:N \exp_not:N \l__keys_selective_bool
15521 \tl_set:Nn \exp_not:N \l__keys_relative_tl { \exp_not:n {#1} }
15522 __keys_set_selective:nnn \exp_not:n { {#2} {#3} {#4} }
15523 \bool_if:NT \l__keys_only_known_bool
15524 { \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
15525 \bool_if:NF \l__keys_filtered_bool
15526 { \bool_set_false:N \exp_not:N \l__keys_filtered_bool }
15527 \bool_if:NF \l__keys_selective_bool
15528 { \bool_set_false:N \exp_not:N \l__keys_selective_bool }
15529 \tl_set:Nn \exp_not:N \l__keys_relative_tl
15530 { \exp_not:o \l__keys_relative_tl }
15531 }
15532 }
15533 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
15534 {
15535 \use:x
15536 {
15537 \bool_set_false:N \exp_not:N \l__keys_only_known_bool
15538 \bool_set_false:N \exp_not:N \l__keys_filtered_bool
15539 \bool_set_true:N \exp_not:N \l__keys_selective_bool
15540 \tl_set:Nn \exp_not:N \l__keys_relative_tl
15541 { \exp_not:N \q__keys_no_value }
15542 __keys_set_selective:nnn \exp_not:n { {#1} {#2} {#3} }
15543 \bool_if:NT \l__keys_only_known_bool
15544 { \bool_set_true:N \exp_not:N \l__keys_only_known_bool }

```

```

15545 \bool_if:NF \l__keys_filtered_bool
15546 { \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
15547 \bool_if:NF \l__keys_selective_bool
15548 { \bool_set_false:N \exp_not:N \l__keys_selective_bool }
15549 \tl_set:Nn \exp_not:N \l__keys_relative_tl
15550 { \exp_not:o \l__keys_relative_tl }
15551 }
15552 }
15553 \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }
15554 \cs_new_protected:Npn __keys_set_selective:nnn
15555 { \exp_args:No __keys_set_selective:nnnn \l__keys_selective_seq }
15556 \cs_new_protected:Npn __keys_set_selective:nnnn #1#2#3#4
15557 {
15558 \seq_set_from_clist:Nn \l__keys_selective_seq {#3}
15559 __keys_set:nn {#2} {#4}
15560 \tl_set:Nn \l__keys_selective_seq {#1}
15561 }

```

(End definition for `\keys_set_filter:nnN` and others. These functions are documented on page 195.)

A shared system once again. First, set the current path and add a default if needed. There are then checks to see if the a value is required or forbidden. If everything passes, move on to execute the code.

```

15562 \cs_new_protected:Npn __keys_set_keyval:n #1
15563 {
15564 \bool_set_true:N \l__keys_no_value_bool
15565 __keys_set_keyval:onn \l__keys_module_str {#1} { }
15566 }
15567 \cs_new_protected:Npn __keys_set_keyval:nn #1#2
15568 {
15569 \bool_set_false:N \l__keys_no_value_bool
15570 __keys_set_keyval:onn \l__keys_module_str {#1} {#2}
15571 }

```

The key path here can be fully defined, after which there is a search for the key and module names: the user may have passed them with part of what is actually the module (for our purposes) in the key name. As that happens on a per-key basis, we use the stack approach to restore the module name without a group.

```

15572 \cs_new_protected:Npn __keys_set_keyval:nnn #1#2#3
15573 {
15574 __kernel_tl_set:Nx \l_keys_path_str
15575 {
15576 \tl_if_blank:nF {#1}
15577 { #1 / }
15578 __keys_trim_spaces:n {#2}
15579 }
15580 \str_clear:N \l__keys_module_str
15581 \str_clear:N \l__keys_inherit_str
15582 \exp_after:wN __keys_find_key_module:wNN \l_keys_path_str \s__keys_stop
15583 \l__keys_module_str \l_keys_key_str
15584 \tl_set_eq:NN \l_keys_key_tl \l_keys_key_str
15585 __keys_value_or_default:n {#3}
15586 \bool_if:NTF \l__keys_selective_bool
15587 __keys_set_selective:

```

```

15588 _keys_execute:
15589 \str_set:Nn \l__keys_module_str {#1}
15590 }
15591 \cs_generate_variant:Nn _keys_set_keyval:nnn { o }

This function uses \cs_set_nopar:Npx internally for performance reasons, the argument #1 is already a string in every usage, so turning it into a string again seems unnecessary.

15592 \cs_new_protected:Npn _keys_find_key_module:wNN #1 \s__keys_stop #2 #3
15593 {
15594 _keys_find_key_module_auxi:Nw #2 #1 \s__keys_nil _keys_find_key_module_auxii:Nw
15595 / \s__keys_nil _keys_find_key_module_auxiv:Nw #3
15596 }
15597 \cs_new_protected:Npn _keys_find_key_module_auxi:Nw #1 #2 / #3 \s__keys_nil #4
15598 {
15599 #4 #1 #2 \s__keys_mark #3 \s__keys_nil #4
15600 }
15601 \cs_new_protected:Npn _keys_find_key_module_auxii:Nw
15602 #1 #2 \s__keys_mark #3 \s__keys_nil _keys_find_key_module_auxii:Nw
15603 {
15604 \cs_set_nopar:Npx #1 { \tl_if_empty:NF #1 { #1 / } #2 }
15605 _keys_find_key_module_auxi:Nw #1 #3 \s__keys_nil _keys_find_key_module_auxiii:Nw
15606 }
15607 \cs_new_protected:Npn _keys_find_key_module_auxiii:Nw #1 #2 \s__keys_mark
15608 {
15609 \cs_set_nopar:Npx #1 { \tl_if_empty:NF #1 { #1 / } #2 }
15610 _keys_find_key_module_auxi:Nw #1
15611 }
15612 \cs_new_protected:Npn _keys_find_key_module_auxiv:Nw
15613 #1 #2 \s__keys_nil #3 \s__keys_mark
15614 \s__keys_nil _keys_find_key_module_auxiv:Nw #4
15615 {
15616 \cs_set_nopar:Npn #4 { #2 }
15617 }

```

If selective setting is active, there are a number of possible sub-cases to consider. The key name may not be known at all or if it is, it may not have any groups assigned. There is then the question of whether the selection is opt-in or opt-out.

```

15618 \cs_new_protected:Npn _keys_set_selective:
15619 {
15620 \cs_if_exist:cTF { \c__keys_groups_root_str \l_keys_path_str }
15621 {
15622 \clist_set_eq:Nc \l__keys_groups_clist
15623 { \c__keys_groups_root_str \l_keys_path_str }
15624 _keys_check_groups:
15625 }
15626 {
15627 \bool_if:NTF \l__keys_filtered_bool
15628 _keys_execute:
15629 _keys_store_unused:
15630 }
15631 }

```

In the case where selective setting requires a comparison of the list of groups which apply to a key with the list of those which have been set active. That requires two mappings, and again a different outcome depending on whether opt-in or opt-out is set.

```

15632 \cs_new_protected:Npn __keys_check_groups:
15633 {
15634 \bool_set_false:N \l__keys_tmp_bool
15635 \seq_map_inline:Nn \l__keys_selective_seq
15636 {
15637 \clist_map_inline:Nn \l__keys_groups_clist
15638 {
15639 \str_if_eq:nnT {##1} {####1}
15640 {
15641 \bool_set_true:N \l__keys_tmp_bool
15642 \clist_map_break:n \seq_map_break:
15643 }
15644 }
15645 }
15646 \bool_if:NTF \l__keys_tmp_bool
15647 {
15648 \bool_if:NTF \l__keys_filtered_bool
15649 __keys_store_unused:
15650 __keys_execute:
15651 }
15652 {
15653 \bool_if:NTF \l__keys_filtered_bool
15654 __keys_execute:
15655 __keys_store_unused:
15656 }
15657 }

```

(End definition for `\__keys_set_keyval:n` and others.)

`\__keys_value_or_default:n` If a value is given, return it as #1, otherwise send a default if available.

```

__keys_default_inherit:
15658 \cs_new_protected:Npn __keys_value_or_default:n #1
15659 {
15660 \bool_if:NTF \l__keys_no_value_bool
15661 {
15662 \cs_if_exist:cTF { \c__keys_default_root_str \l_keys_path_str }
15663 {
15664 \tl_set_eq:Nc
15665 \l_keys_value_tl
15666 { \c__keys_default_root_str \l_keys_path_str }
15667 }
15668 {
15669 \tl_clear:N \l_keys_value_tl
15670 \cs_if_exist:cT
15671 { \c__keys_inherit_root_str __keys_parent:o \l_keys_path_str }
15672 { __keys_default_inherit: }
15673 }
15674 }
15675 { \tl_set:Nn \l_keys_value_tl {#1} }
15676 }
15677 \cs_new_protected:Npn __keys_default_inherit:
15678 {
15679 \clist_map_inline:cn
15680 { \c__keys_inherit_root_str __keys_parent:o \l_keys_path_str }
15681 {

```

```

15682 \cs_if_exist:cT
15683 { \c_keys_default_root_str ##1 / \l_keys_key_str }
15684 {
15685 \tl_set_eq:Nc
15686 \l_keys_value_tl
15687 { \c_keys_default_root_str ##1 / \l_keys_key_str }
15688 \clist_map_break:
15689 }
15690 }
15691 }

```

(End definition for `\_keys_value_or_default:n` and `\_keys_default_inherit:..`)

`\_keys_execute:` Actually executing a key is done in two parts. First, look for the key itself, then look for the `unknown` key with the same path. If both of these fail, complain. What exactly happens if a key is unknown depends on whether unknown keys are being skipped or if an error should be raised.

```

15692 \cs_new_protected:Npn _keys_execute:
15693 {
15694 \cs_if_exist:cTF { \c_keys_code_root_str \l_keys_path_str }
15695 {
15696 \cs_if_exist_use:c { \c_keys_validate_root_str \l_keys_path_str }
15697 _keys_execute:no \l_keys_path_str \l_keys_value_tl
15698 }
15699 {
15700 \cs_if_exist:cTF
15701 { \c_keys_inherit_root_str _keys_parent:o \l_keys_path_str }
15702 { _keys_execute_inherit: }
15703 { _keys_execute_unknown: }
15704 }
15705 }

```

To deal with the case where there is no hit, we leave `\_keys_execute_unknown:` in the input stream and clean it up using the `break` function: that avoids needing a boolean.

```

15706 \cs_new_protected:Npn _keys_execute_inherit:
15707 {
15708 \clist_map_inline:cn
15709 { \c_keys_inherit_root_str _keys_parent:o \l_keys_path_str }
15710 {
15711 \cs_if_exist:cT
15712 { \c_keys_code_root_str ##1 / \l_keys_key_str }
15713 {
15714 \str_set:Nn \l_keys_inherit_str {##1}
15715 \cs_if_exist_use:c { \c_keys_validate_root_str ##1 / \l_keys_key_str }
15716 _keys_execute:no { ##1 / \l_keys_key_str } \l_keys_value_tl
15717 \clist_map_break:n \use_none:n
15718 }
15719 }
15720 _keys_execute_unknown:
15721 }
15722 \cs_new_protected:Npn _keys_execute_unknown:
15723 {
15724 \bool_if:NTF \l_keys_only_known_bool
15725 { _keys_store_unused: }

```

```

15726 {
15727 \cs_if_exist:cTF
15728 { \c_keys_code_root_str \l_keys_module_str / unknown }
15729 { __keys_execute:no { \l_keys_module_str / unknown } \l_keys_value_tl }
15730 {
15731 __kernel_msg_error:nxxx { kernel } { key-unknown }
15732 \l_keys_path_str \l_keys_module_str
15733 }
15734 }
15735 }
```

A key's code is in the control sequence with csname `\c_keys_code_root_str` #1. We expand it once to get the replacement text (with argument #2) and call `\use:n` with this replacement as its argument. This ensures that any undefined control sequence error in the key's code will lead to an error message of the form `<argument>...<control sequence>` in which one can read the (undefined) `<control sequence>` in full, rather than an error message that starts with the potentially very long key name, which would make the (undefined) `<control sequence>` be truncated or sometimes completely hidden. See <https://github.com/latex3/latex2e/issues/351>.

```

15736 \cs_new:Npn __keys_execute:nn #1#2
15737 { __keys_execute:no {#1} { \prg_do_nothing: #2 } }
15738 \cs_new:Npn __keys_execute:no #1#2
15739 {
15740 \exp_args:NNo \exp_args:No \use:n
15741 {
15742 \cs:w \c_keys_code_root_str #1 \exp_after:wN \cs_end:
15743 \exp_after:wN {#2}
15744 }
15745 }
```

When there is no relative path, things here are easy: just save the key name and value. When we are working with a relative path, first we need to turn it into a string: that can't happen earlier as we need to store `\q_keys_no_value`. Then, use a standard delimited approach to fish out the partial path.

```

15746 \cs_new_protected:Npn __keys_store_unused:
15747 {
15748 __keys_quark_if_no_value:NTF \l_keys_relative_tl
15749 {
15750 \clist_put_right:Nx \l_keys_unused_clist
15751 {
15752 \l_keys_key_str
15753 \bool_if:NF \l_keys_no_value_bool
15754 { = { \exp_not:o \l_keys_value_tl } }
15755 }
15756 }
15757 {
15758 \tl_if_empty:NTF \l_keys_relative_tl
15759 {
15760 \clist_put_right:Nx \l_keys_unused_clist
15761 {
15762 \l_keys_path_str
15763 \bool_if:NF \l_keys_no_value_bool
15764 { = { \exp_not:o \l_keys_value_tl } }
15765 }
15766 }
```

```

15766 }
15767 { __keys_store_unused_aux: }
15768 }
15769 }
15770 \cs_new_protected:Npn __keys_store_unused_aux:
15771 {
15772 __kernel_tl_set:Nx \l__keys_relative_tl
15773 { \exp_args:No __keys_trim_spaces:n \l__keys_relative_tl }
15774 \use:x
15775 {
15776 \cs_set_protected:Npn __keys_store_unused:w
15777 #####1 \l__keys_relative_tl /
15778 #####2 \l__keys_relative_tl /
15779 #####3 \s__keys_stop
15780 }
15781 {
15782 \tl_if_blank:nF {##1}
15783 {
15784 __kernel_msg_error:nxxx { kernel } { bad-relative-key-path }
15785 \l_keys_path_str
15786 \l__keys_relative_tl
15787 }
15788 \clist_put_right:Nx \l__keys_unused_clist
15789 {
15790 \exp_not:n {##2}
15791 \bool_if:NF \l__keys_no_value_bool
15792 { = { \exp_not:o \l_keys_value_tl } }
15793 }
15794 }
15795 \use:x
15796 {
15797 __keys_store_unused:w \l_keys_path_str
15798 \l__keys_relative_tl / \l__keys_relative_tl /
15799 \s__keys_stop
15800 }
15801 }
15802 \cs_new_protected:Npn __keys_store_unused:w { }

```

(End definition for `\__keys_execute:` and others.)

```

__keys_choice_find:n
__keys_choice_find:nn
__keys_multichoice_find:n

```

Executing a choice has two parts. First, try the choice given, then if that fails call the unknown key. That always exists, as it is created when a choice is first made. So there is no need for any escape code. For multiple choices, the same code ends up used in a mapping.

```

15803 \cs_new:Npn __keys_choice_find:n #1
15804 {
15805 \str_if_empty:NTF \l__keys_inherit_str
15806 { __keys_choice_find:nn \l_keys_path_str {#1} }
15807 {
15808 __keys_choice_find:nn
15809 { \l__keys_inherit_str / \l_keys_key_str } {#1}
15810 }
15811 }
15812 \cs_new:Npn __keys_choice_find:nn #1#2

```

```

15813 {
15814 \cs_if_exist:cTF { \c_keys_code_root_str #1 / \keys_trim_spaces:n {#2} }
15815 { \keys_execute:nn { #1 / \keys_trim_spaces:n {#2} } {#2} }
15816 { \keys_execute:nn { #1 / unknown } {#2} }
15817 }
15818 \cs_new:Npn \keys_multichoice_find:n #1
15819 { \clist_map_function:nN {#1} \keys_choice_find:n }

(End definition for \keys_choice_find:n, \keys_choice_find:nn, and \keys_multichoice-
find:n.)

```

## 22.7 Utilities

\keys\_parent:o Used to strip off the ending part of the key path after the last /.

```

15820 \cs_new:Npn \keys_parent:o #1
15821 {
15822 \exp_after:wN \keys_parent_auxi:w #1 \q_nil \keys_parent_auxii:w
15823 / \q_nil \keys_parent_auxiv:w
15824 }
15825 \cs_new:Npn \keys_parent_auxi:w #1 / #2 \q_nil #3
15826 {
15827 #3 { #1 } #2 \q_nil #3
15828 }
15829 \cs_new:Npn \keys_parent_auxii:w #1 #2 \q_nil \keys_parent_auxii:w
15830 {
15831 #1 \keys_parent_auxi:w #2 \q_nil \keys_parent_auxiii:n
15832 }
15833 \cs_new:Npn \keys_parent_auxiii:n #1
15834 {
15835 / #1 \keys_parent_auxi:w
15836 }
15837 \cs_new:Npn \keys_parent_auxiv:w #1 \q_nil \keys_parent_auxiv:w
15838 {
15839 }

```

(End definition for \keys\_parent:o and others.)

\keys\_trim\_spaces:n Space stripping has to allow for the fact that the key here might have several parts, and spaces need to be stripped from each part. Since the key name is turned into a string groups can't be stripped accidentally and the precautions of \tl\_trim\_spaces:n aren't necessary, in this case it is much faster to just directly strip spaces around /.

```

15840 \group_begin:
15841 \cs_set:Npn \keys_tmp:n #1
15842 {
15843 \cs_new:Npn \keys_trim_spaces:n ##1
15844 {
15845 \exp_after:wN \keys_trim_spaces_auxi:w \tl_to_str:n { / ##1 } /
15846 \s_keys_nil \keys_trim_spaces_auxi:w
15847 \s_keys_mark \keys_trim_spaces_auxii:w
15848 #1 / #1
15849 \s_keys_nil \keys_trim_spaces_auxii:w
15850 \s_keys_mark \keys_trim_spaces_auxiii:w
15851 }
15852 }

```

```

15853 __keys_tmp:n { ~ }
15854 \group_end:
15855 \cs_new:Npn __keys_trim_spaces_auxi:w #1 ~ / #2 \s__keys_nil #3
15856 {
15857 #3 #1 / #2 \s__keys_nil #3
15858 }
15859 \cs_new:Npn __keys_trim_spaces_auxii:w #1 / ~ #2 \s__keys_mark #3
15860 {
15861 #3 #1 / #2 \s__keys_mark #3
15862 }
15863 \cs_new:Npn __keys_trim_spaces_auxiii:w
15864 / #1 /
15865 \s__keys_nil __keys_trim_spaces_auxi:w
15866 \s__keys_mark __keys_trim_spaces_auxii:w
15867 /
15868 \s__keys_nil __keys_trim_spaces_auxii:w
15869 \s__keys_mark __keys_trim_spaces_auxiii:w
15870 {
15871 #1
15872 }

```

(End definition for `\__keys_trim_spaces:n` and others.)

`\keys_if_exist_p:nn` A utility for others to see if a key exists.

`\keys_if_exist:nnTF`

```

15873 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
15874 {
15875 \cs_if_exist:cTF
15876 { \c__keys_code_root_str __keys_trim_spaces:n { #1 / #2 } }
15877 { \prg_return_true: }
15878 { \prg_return_false: }
15879 }

```

(End definition for `\keys_if_exist:nnTF`. This function is documented on page 195.)

`\keys_if_choice_exist_p:nnn` Just an alternative view on `\keys_if_exist:nnTF`.

`\keys_if_choice_exist:nnnTF`

```

15880 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
15881 { p , T , F , TF }
15882 {
15883 \cs_if_exist:cTF
15884 { \c__keys_code_root_str __keys_trim_spaces:n { #1 / #2 / #3 } }
15885 { \prg_return_true: }
15886 { \prg_return_false: }
15887 }

```

(End definition for `\keys_if_choice_exist:nnnTF`. This function is documented on page 196.)

`\keys_show:nn` To show a key, show its code using a message.

`\keys_log:nn`

`\__keys_show:Nnn`

```

15888 \cs_new_protected:Npn \keys_show:nn
15889 { __keys_show:Nnn \msg_show:nnxxxx }
15890 \cs_new_protected:Npn \keys_log:nn
15891 { __keys_show:Nnn \msg_log:nnxxxx }
15892 \cs_new_protected:Npn __keys_show:Nnn #1#2#3
15893 {
15894 #1 { LaTeX / kernel } { show-key }
15895 { __keys_trim_spaces:n { #2 / #3 } }

```

```

15896 {
15897 \keys_if_exist:nnt {#2} {#3}
15898 {
15899 \exp_args:Nnf \msg_show_item_unbraced:nn { code }
15900 {
15901 \exp_args:Nc \cs_replacement_spec:N
15902 {
15903 \c_keys_code_root_str
15904 __keys_trim_spaces:n { #2 / #3 }
15905 }
15906 }
15907 }
15908 }
15909 { } { }
15910 }

```

(End definition for `\keys_show:nn`, `\keys_log:nn`, and `\__keys_show:Nnn`. These functions are documented on page 196.)

## 22.8 Messages

For when there is a need to complain.

```

15911 __kernel_msg_new:nnnn { kernel } { bad-relative-key-path }
15912 { The~key~'#1'~is~not~inside~the~'#2'~path. }
15913 { The~key~'#1'~cannot~be~expressed~relative~to~path~'#2'. }
15914 __kernel_msg_new:nnnn { kernel } { boolean-values-only }
15915 { Key~'#1'~accepts~boolean~values~only. }
15916 { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
15917 __kernel_msg_new:nnnn { kernel } { key-choice-unknown }
15918 { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
15919 {
15920 The~key~'#1'~only~accepts~predefined~values,~
15921 and~'#2'~is~not~one~of~these.
15922 }
15923 __kernel_msg_new:nnnn { kernel } { key-unknown }
15924 { The~key~'#1'~is~unknown~and~is~being~ignored. }
15925 {
15926 The~module~'#2'~does~not~have~a~key~called~'#1'.\\
15927 Check~that~you~have~spelled~the~key~name~correctly.
15928 }
15929 __kernel_msg_new:nnnn { kernel } { nested-choice-key }
15930 { Attempt~to~define~'#1'~as~a~nested~choice~key. }
15931 {
15932 The~key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
15933 itself~a~choice.
15934 }
15935 __kernel_msg_new:nnnn { kernel } { value-forbidden }
15936 { The~key~'#1'~does~not~take~a~value. }
15937 {
15938 The~key~'#1'~should~be~given~without~a~value.\\
15939 The~value~'#2'~was~present:~the~key~will~be~ignored.
15940 }
15941 __kernel_msg_new:nnnn { kernel } { value-required }
15942 { The~key~'#1'~requires~a~value. }

```

```

15943 {
15944 The~key~'#1'~must~have~a~value.\\
15945 No~value~was~present:~the~key~will~be~ignored.
15946 }
15947 __kernel_msg_new:nnn { kernel } { show-key }
15948 {
15949 The~key~'#1~'
15950 \tl_if_empty:nTF {#2}
15951 { is~undefined. }
15952 { has~the~properties: #2 . }
15953 }
15954
```

## 23 I3intarray implementation

```

15955 (*package)
15956 @=intarray

```

### 23.1 Allocating arrays

`\__intarray_entry:w`  
`\__intarray_count:w`

```

15957 \cs_new_eq:NN __intarray_entry:w \tex_fontdimen:D
15958 \cs_new_eq:NN __intarray_count:w \tex_hyphenchar:D

```

(End definition for `\__intarray_entry:w` and `\__intarray_count:w`.)

`\l__intarray_loop_int` A loop index.

```

15959 \int_new:N \l__intarray_loop_int

```

(End definition for `\l__intarray_loop_int`.)

`\c__intarray_sp_dim` Used to convert integers to dimensions fast.

```

15960 \dim_const:Nn \c__intarray_sp_dim { 1 sp }

```

(End definition for `\c__intarray_sp_dim`.)

`\g__intarray_font_int` Used to assign one font per array.

```

15961 \int_new:N \g__intarray_font_int

```

(End definition for `\g__intarray_font_int`.)

```

15962 __kernel_msg_new:nnn { kernel } { negative-array-size }
15963 { Size~of~array~may~not~be~negative:~#1 }

```

`\intarray_new:Nn`  
`\intarray_new:cN`  
`\__intarray_new:N`

Declare #1 to be a font (arbitrarily `cmr10` at a never-used size). Store the array's size as the `\hyphenchar` of that font and make sure enough `\fontdimen` are allocated, by setting the last one. Then clear any `\fontdimen` that `cmr10` starts with. It seems LuaTeX's `cmr10` has an extra `\fontdimen` parameter number 8 compared to other engines (for a math font we would replace 8 by 22 or some such). Every `intarray` must be global; it's enough to run this check in `\intarray_new:Nn`.

```

15964 \cs_new_protected:Npn __intarray_new:N #1
15965 {
15966 __kernel_chk_if_free_cs:N #1
15967 \int_gincr:N \g__intarray_font_int

```

```

15968 \tex_global:D \tex_font:D #1
15969 = cmr10-at~ \g__intarray_font_int \c__intarray_sp_dim \scan_stop:
15970 \int_step_inline:nn { 8 }
15971 { __kernel_intarray_gset:Nnn #1 {##1} \c_zero_int }
15972 }
15973 \cs_new_protected:Npn \intarray_new:Nn #1#2
15974 {
15975 __intarray_new:N #1
15976 __intarray_count:w #1 = \int_eval:n {#2} \scan_stop:
15977 \int_compare:nNnT { \intarray_count:N #1 } < 0
15978 {
15979 __kernel_msg_error:nnx { kernel } { negative-array-size }
15980 { \intarray_count:N #1 }
15981 }
15982 \int_compare:nNnT { \intarray_count:N #1 } > 0
15983 { __kernel_intarray_gset:Nnn #1 { \intarray_count:N #1 } { 0 } }
15984 }
15985 \cs_generate_variant:Nn \intarray_new:Nn { c }

(End definition for \intarray_new:Nn and __intarray_new:N. This function is documented on page 198.)
```

\intarray\_count:N Size of an array.

```

\intarray_count:c
15986 \cs_new:Npn \intarray_count:N #1 { \int_value:w __intarray_count:w #1 }
15987 \cs_generate_variant:Nn \intarray_count:N { c }
```

(End definition for \intarray\_count:N. This function is documented on page 198.)

## 23.2 Array items

\\_\_intarray\_signed\_max\_dim:n Used when an item to be stored is larger than \c\_max\_dim in absolute value; it is replaced by  $\pm \c_max_dim$ .

```

15988 \cs_new:Npn __intarray_signed_max_dim:n #1
15989 { \int_value:w \int_compare:nNnT {#1} < 0 { - } \c_max_dim }
```

(End definition for \\_\_intarray\_signed\_max\_dim:n.)

\\_\_intarray\_bounds>NNnTF \\_\_intarray\_bounds\_error>NNnw The functions \intarray\_gset:Nnn and \intarray\_item:Nn share bounds checking. The T branch is used if #3 is within bounds of the array #2.

```

15990 \cs_new:Npn __intarray_bounds>NNnTF #1#2#3
15991 {
15992 \if_int_compare:w 1 > #3 \exp_stop_f:
15993 __intarray_bounds_error>NNnw #1 #2 {#3}
15994 \else:
15995 \if_int_compare:w #3 > \intarray_count:N #2 \exp_stop_f:
15996 __intarray_bounds_error>NNnw #1 #2 {#3}
15997 \fi:
15998 \fi:
15999 \use_i:nn
16000 }
16001 \cs_new:Npn __intarray_bounds_error>NNnw #1#2#3#4 \use_i:nn #5#6
16002 {
16003 #4
16004 #1 { kernel } { out-of-bounds }
```

```

16005 { \token_to_str:N #2 } {#3} { \intarray_count:N #2 }
16006 #6
16007 }

```

(End definition for `\_intarray_bounds:NNnTF` and `\_intarray_bounds_error:NNnw`.)

`\intarray_gset:Nnn`  
`\intarray_gset:cnn`

Set the appropriate `\fontdimen`. The `\_kernel_intarray_gset:Nnn` function does not use `\int_eval:n`, namely its arguments must be suitable for `\int_value:w`. The user version checks the position and value are within bounds.

```

16008 \cs_new_protected:Npn _kernel_intarray_gset:Nnn #1#2#3
16009 { _intarray_entry:w #2 #1 #3 \c_intarray_sp_dim }
16010 \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
16011 {
16012 \exp_after:wN _intarray_gset:Nww
16013 \exp_after:wN #1
16014 \int_value:w \int_eval:n {#2} \exp_after:wN ;
16015 \int_value:w \int_eval:n {#3} ;
16016 }
16017 \cs_generate_variant:Nn \intarray_gset:Nnn { c }
16018 \cs_new_protected:Npn _intarray_gset:Nww #1#2 ; #3 ;
16019 {
16020 _intarray_bounds:NNnTF _kernel_msg_error:nnxxxx #1 {#2}
16021 {
16022 _intarray_gset_overflow_test:nw {#3}
16023 _kernel_intarray_gset:Nnn #1 {#2} {#3}
16024 }
16025 {
16026 }
16027 \cs_if_exist:NTF \tex_ifabsnum:D
16028 {
16029 \cs_new_protected:Npn _intarray_gset_overflow_test:nw #1
16030 {
16031 \tex_ifabsnum:D #1 > \c_max_dim
16032 \exp_after:wN _intarray_gset_overflow:NNnn
16033 \fi:
16034 }
16035 {
16036 {
16037 \cs_new_protected:Npn _intarray_gset_overflow_test:nw #1
16038 {
16039 \if_int_compare:w \int_abs:n {#1} > \c_max_dim
16040 \exp_after:wN _intarray_gset_overflow:NNnn
16041 \fi:
16042 }
16043 }
16044 \cs_new_protected:Npn _intarray_gset_overflow:NNnn #1#2#3#4
16045 {
16046 _kernel_msg_error:nnxxxx { kernel } { overflow }
16047 { \token_to_str:N #2 } {#3} {#4} { _intarray_signed_max_dim:n {#4} }
16048 #1 #2 {#3} { _intarray_signed_max_dim:n {#4} }
16049 }

```

(End definition for `\intarray_gset:Nnn` and others. This function is documented on page 198.)

`\intarray_gzero:N` Set the appropriate `\fontdimen` to zero. No bound checking needed. The `\prg_replicate:nn` possibly uses quite a lot of memory, but this is somewhat comparable to the size of the array, and it is much faster than an `\int_step_inline:nn` loop.

```

16050 \cs_new_protected:Npn \intarray_gzero:N #1
16051 {
16052 \int_zero:N \l__intarray_loop_int
16053 \prg_replicate:nn { \intarray_count:N #1 }
16054 {
16055 \int_incr:N \l__intarray_loop_int
16056 __intarray_entry:w \l__intarray_loop_int #1 \c_zero_dim
16057 }
16058 }
16059 \cs_generate_variant:Nn \intarray_gzero:N { c }
```

(End definition for `\intarray_gzero:N`. This function is documented on page 198.)

`\intarray_item:Nn` Get the appropriate `\fontdimen` and perform bound checks. The `\__kernel_intarray_item:Nn` function omits bound checks and omits `\int_eval:n`, namely its argument must be a TeX integer suitable for `\int_value:w`.

```

16060 \cs_new:Npn __kernel_intarray_item:Nn #1#2
16061 { \int_value:w __intarray_entry:w #2 #1 }
16062 \cs_new:Npn \intarray_item:Nn #1#2
16063 {
16064 \exp_after:wN __intarray_item:Nw
16065 \exp_after:wN #1
16066 \int_value:w \int_eval:n {#2} ;
16067 }
16068 \cs_generate_variant:Nn \intarray_item:Nn { c }
16069 \cs_new:Npn __intarray_item:Nw #1#2 ;
16070 {
16071 __intarray_bounds>NNnTF __kernel_msg_expandable_error:nnfff #1 {#2}
16072 { __kernel_intarray_item:Nn #1 {#2} }
16073 { 0 }
16074 }
```

(End definition for `\intarray_item:Nn`, `\__kernel_intarray_item:Nn`, and `\__intarray_item:Nn`. This function is documented on page 199.)

`\intarray_rand_item:N` Importantly, `\intarray_item:Nn` only evaluates its argument once.

```

16075 \cs_new:Npn \intarray_rand_item:N #1
16076 { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
16077 \cs_generate_variant:Nn \intarray_rand_item:N { c }
```

(End definition for `\intarray_rand_item:N`. This function is documented on page 199.)

### 23.3 Working with contents of integer arrays

`\intarray_const_from_clist:Nn` Similar to `\intarray_new:Nn` (which we don't use because when debugging is enabled that function checks the variable name starts with `g_`). We make use of the fact that TeX allows allocation of successive `\fontdimen` as long as no other font has been declared: no need to count the comma list items first. We need the code in `\intarray_gset:Nnn` that checks the item value is not too big, namely `\__intarray_gset_overflow_test:nw`, but not the code that checks bounds. At the end, set the size of the intarray.

```

16078 \cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
16079 {
16080 __intarray_new:N #1
16081 \int_zero:N \l__intarray_loop_int
16082 \clist_map_inline:nn {#2}
16083 { \exp_args:Nf __intarray_const_from_clist:nN { \int_eval:n {##1} } #1 }
16084 __intarray_count:w #1 \l__intarray_loop_int
16085 }
16086 \cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }
16087 \cs_new_protected:Npn __intarray_const_from_clist:nN #1#2
16088 {
16089 \int_incr:N \l__intarray_loop_int
16090 __intarray_gset_overflow_test:nw {#1}
16091 __kernel_intarray_gset:Nnn #2 \l__intarray_loop_int {#1}
16092 }

```

(End definition for `\intarray_const_from_clist:Nn` and `\__intarray_const_from_clist:nN`. This function is documented on page 198.)

**\intarray\_to\_clist:N** Loop through the array, putting a comma before each item. Remove the leading comma with f-expansion. We also use the auxiliary in `\intarray_show:N` with argument comma, space.

```

\intarray_to_clist:c
\intarray_to_clist:Nn
\intarray_to_clist:w
16093 \cs_new:Npn \intarray_to_clist:N #1 { __intarray_to_clist:Nn #1 { , } }
16094 \cs_generate_variant:Nn \intarray_to_clist:N { c }
16095 \cs_new:Npn __intarray_to_clist:Nn #1#2
16096 {
16097 \int_compare:nNnF { \intarray_count:N #1 } = \c_zero_int
16098 {
16099 \exp_last_unbraced:Nf \use_none:n
16100 { __intarray_to_clist:w 1 ; #1 {#2} \prg_break_point: }
16101 }
16102 }
16103 \cs_new:Npn __intarray_to_clist:w #1 ; #2#3
16104 {
16105 \if_int_compare:w #1 > __intarray_count:w #2
16106 \prg_break:n
16107 \fi:
16108 #3 __kernel_intarray_item:Nn #2 {#1}
16109 \exp_after:wN __intarray_to_clist:w
16110 \int_value:w \int_eval:w #1 + \c_one_int ; #2 {#3}
16111 }

```

(End definition for `\intarray_to_clist:N`, `\__intarray_to_clist:Nn`, and `\__intarray_to_clist:w`. This function is documented on page 267.)

**\intarray\_show:N** Convert the list to a comma list (with spaces after each comma)

```

\intarray_show:c
\intarray_log:N
\intarray_log:c
16112 \cs_new_protected:Npn \intarray_show:N { __intarray_show>NN \msg_show:nxxxxx }
16113 \cs_generate_variant:Nn \intarray_show:N { c }
16114 \cs_new_protected:Npn \intarray_log:N { __intarray_show>NN \msg_log:nxxxxx }
16115 \cs_generate_variant:Nn \intarray_log:N { c }
16116 \cs_new_protected:Npn __intarray_show>NN #1#2
16117 {
16118 __kernel_chk_defined:NT #2
16119 {
16120 #1 { LaTeX/kernel } { show-intarray }

```

```

16121 { \token_to_str:N #2 }
16122 { \intarray_count:N #2 }
16123 { >~ __intarray_to_clist:Nn #2 { , ~ } }
16124 { }
16125 }
16126 }
```

(End definition for `\intarray_show:N` and `\intarray_log:N`. These functions are documented on page 199.)

## 23.4 Random arrays

We only perform the bounds checks once. This is done by two `\__intarray_gset_overflow_test:nw`, with an appropriate empty argument to avoid a spurious “at position #1” part in the error message. Then calculate the number of choices: this is at most  $(2^{30}-1)-(-(2^{30}-1))+1 = 2^{31}-1$ , which just barely does not overflow. For small ranges use `\__kernel_randint:n` (making sure to subtract 1 *before* adding the random number to the `<min>`, to avoid overflow when `<min>` or `<max>` are  $\pm\text{\c_max\_int}$ ), otherwise `\__kernel_randint:nn`. Finally, if there are no random numbers do not define any of the auxiliaries.

```

16127 \cs_new_protected:Npn \intarray_gset_rand:Nn #1
16128 { \intarray_gset_rand:Nnn #1 { 1 } }
16129 \cs_generate_variant:Nn \intarray_gset_rand:Nn { c }
16130 \sys_if_rand_exist:TF
16131 {
16132 \cs_new_protected:Npn \intarray_gset_rand:Nnn #1#2#3
16133 {
16134 __intarray_gset_rand:Nff #1
16135 { \int_eval:n {#2} } { \int_eval:n {#3} }
16136 }
16137 \cs_new_protected:Npn __intarray_gset_rand:Nnn #1#2#3
16138 {
16139 \int_compare:nNnTF {#2} > {#3}
16140 {
16141 __kernel_msg_expandable_error:nnnn
16142 { kernel } { randint-backward-range } {#2} {#3}
16143 __intarray_gset_rand:Nnn #1 {#3} {#2}
16144 }
16145 {
16146 __intarray_gset_overflow_test:nw {#2}
16147 __intarray_gset_rand_auxi:Nnnn #1 { } {#2} {#3}
16148 }
16149 }
16150 \cs_generate_variant:Nn __intarray_gset_rand:Nnn { Nff }
16151 \cs_new_protected:Npn __intarray_gset_rand_auxi:Nnnn #1#2#3#4
16152 {
16153 __intarray_gset_overflow_test:nw {#4}
16154 __intarray_gset_rand_auxi:Nnnn #1 { } {#4} {#3}
16155 }
16156 \cs_new_protected:Npn __intarray_gset_rand_auxii:Nnnn #1#2#3#4
16157 {
16158 \exp_args:NNf __intarray_gset_rand_auxiii:Nnnn #1
16159 { \int_eval:n { #3 - #4 + 1 } } {#4} {#3}
```

```

16160 }
16161 \cs_new_protected:Npn __intarray_gset_rand_auxiii:Nnnn #1#2#3#4
16162 {
16163 \exp_args:NNf __intarray_gset_all_same:Nn #1
16164 {
16165 \int_compare:nNnTF {#2} > \c_kernel randint_max_int
16166 {
16167 \exp_stop_f:
16168 \int_eval:n { __kernel_randint:nn {#3} {#4} }
16169 }
16170 {
16171 \exp_stop_f:
16172 \int_eval:n { __kernel_randint:n {#2} - 1 + #3 }
16173 }
16174 }
16175 }
16176 \cs_new_protected:Npn __intarray_gset_all_same:Nn #1#2
16177 {
16178 \int_zero:N \l__intarray_loop_int
16179 \prg_replicate:nn { \intarray_count:N #1 }
16180 {
16181 \int_incr:N \l__intarray_loop_int
16182 __kernel_intarray_gset:Nnn #1 \l__intarray_loop_int {#2}
16183 }
16184 }
16185 }
16186 {
16187 \cs_new_protected:Npn \intarray_gset_rand:Nnn #1#2#3
16188 {
16189 __kernel_msg_error:nnn { kernel } { fp-no-random }
16190 { \intarray_gset_rand:Nnn #1 {#2} {#3} }
16191 }
16192 }
16193 \cs_generate_variant:Nn \intarray_gset_rand:Nnn { c }

(End definition for \intarray_gset_rand:Nn and others. These functions are documented on page 267.)
```

16194

## 24 I3fp implementation

Nothing to see here: everything is in the subfiles!

## 25 I3fp-aux implementation

```

16195 (*package)
16196 @@=fp
```

### 25.1 Access to primitives

\\_fp\\_int\\_eval:w      Largely for performance reasons, we need to directly access primitives rather than use \\_fp\\_int\\_eval\_end: . This happens *a lot*, so we use private names. The same is true for \\_fp\\_int\\_to\\_roman:w \romannumeral, although it is used much less widely.

```

16197 \cs_new_eq:NN __fp_int_eval:w \tex_numexpr:D
16198 \cs_new_eq:NN __fp_int_eval_end: \scan_stop:
16199 \cs_new_eq:NN __fp_int_to_roman:w \tex_roman numeral:D

(End definition for __fp_int_eval:w, __fp_int_eval_end:, and __fp_int_to_roman:w.)

```

## 25.2 Internal representation

Internally, a floating point number  $\langle X \rangle$  is a token list containing

```
\s_fp __fp_chk:w ⟨case⟩ ⟨sign⟩ ⟨body⟩ ;
```

Let us explain each piece separately.

Internal floating point numbers are used in expressions, and in this context are subject to f-expansion. They must leave a recognizable mark after f-expansion, to prevent the floating point number from being re-parsed. Thus, `\s_fp` is simply another name for `\relax`.

When used directly without an accessor function, floating points should produce an error: this is the role of `\__fp_chk:w`. We could make floating point variables be protected to prevent them from expanding under x-expansion, but it seems more convenient to treat them as a subcase of token list variables.

The (decimal part of the) IEEE-754-2008 standard requires the format to be able to represent special floating point numbers besides the usual positive and negative cases. We distinguish the various possibilities by their `⟨case⟩`, which is a single digit:

- 0 zeros: +0 and -0,
- 1 “normal” numbers (positive and negative),
- 2 infinities: `+inf` and `-inf`,
- 3 quiet and signalling `nan`.

The `⟨sign⟩` is 0 (positive) or 2 (negative), except in the case of `nan`, which have `⟨sign⟩` = 1. This ensures that changing the `⟨sign⟩` digit to 2 – `⟨sign⟩` is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

```
\s_fp __fp_chk:w ⟨case⟩ ⟨sign⟩ \s_fp_... ;
```

where `\s_fp_...` is a scan mark carrying information about how the number was formed (useful for debugging).

Normal floating point numbers (`⟨case⟩` = 1) have the form

```
\s_fp __fp_chk:w 1 ⟨sign⟩ {⟨exponent⟩} {⟨X1⟩} {⟨X2⟩} {⟨X3⟩} {⟨X4⟩} ;
```

Here, the `⟨exponent⟩` is an integer, between -10000 and 10000. The body consists in four blocks of exactly 4 digits,  $0000 \leq \langle X_i \rangle \leq 9999$ , and the floating point is

$$(-1)^{\langle sign \rangle / 2} \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle \cdot 10^{\langle exponent \rangle - 16}$$

where we have concatenated the 16 digits. Currently, floating point numbers are normalized such that the `⟨exponent⟩` is minimal, in other words,  $1000 \leq \langle X_1 \rangle \leq 9999$ .

Calculations are done in base 10000, i.e. one myriad.

Table 3: Internal representation of floating point numbers.

| Representation                                                                                                      | Meaning                  |
|---------------------------------------------------------------------------------------------------------------------|--------------------------|
| 0 0 \s_fp_... ;                                                                                                     | Positive zero.           |
| 0 2 \s_fp_... ;                                                                                                     | Negative zero.           |
| 1 0 {\langle exponent\rangle} {\langle X_1\rangle} {\langle X_2\rangle} {\langle X_3\rangle} {\langle X_4\rangle} ; | Positive floating point. |
| 1 2 {\langle exponent\rangle} {\langle X_1\rangle} {\langle X_2\rangle} {\langle X_3\rangle} {\langle X_4\rangle} ; | Negative floating point. |
| 2 0 \s_fp_... ;                                                                                                     | Positive infinity.       |
| 2 2 \s_fp_... ;                                                                                                     | Negative infinity.       |
| 3 1 \s_fp_... ;                                                                                                     | Quiet nan.               |
| 3 1 \s_fp_... ;                                                                                                     | Signalling nan.          |

### 25.3 Using arguments and semicolons

\\_fp\_use\_none\_stop\_f:n

This function removes an argument (typically a digit) and replaces it by \exp\_stop\_f:, a marker which stops f-type expansion.

```
16200 \cs_new:Npn _fp_use_none_stop_f:n #1 { \exp_stop_f: }
```

(End definition for \\_fp\_use\_none\_stop\_f:n.)

\\_fp\_use\_s:n

Those functions place a semicolon after one or two arguments (typically digits).

```
16201 \cs_new:Npn _fp_use_s:n #1 { #1; }
```

```
16202 \cs_new:Npn _fp_use_s:nn #1#2 { #1#2; }
```

(End definition for \\_fp\_use\_s:n and \\_fp\_use\_s:nn.)

\\_fp\_use\_none\_until\_s:w  
\\_fp\_use\_i\_until\_s:nw

Those functions select specific arguments among a set of arguments delimited by a semi-colon.

```
16203 \cs_new:Npn _fp_use_none_until_s:w #1; { }
```

```
16204 \cs_new:Npn _fp_use_i_until_s:nw #1#2; {#1}
```

```
16205 \cs_new:Npn _fp_use_i_until_s:nnw #1#2#3; {#2}
```

(End definition for \\_fp\_use\_none\_until\_s:w, \\_fp\_use\_i\_until\_s:nw, and \\_fp\_use\_i\_until\_s:nnw.)

\\_fp\_reverse\_args:Nww

Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.

```
16206 \cs_new:Npn _fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }
```

(End definition for \\_fp\_reverse\_args:Nww.)

\\_fp\_rrrot:www

Rotate three arguments delimited by semicolons. This is the inverse (or the square) of the Forth primitive ROT, hence the name.

```
16207 \cs_new:Npn _fp_rrrot:www #1; #2; #3; { #2; #3; #1; }
```

(End definition for \\_fp\_rrrot:www.)

\\_fp\_use\_i:ww  
\\_fp\_use\_i:www

Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.

```
16208 \cs_new:Npn _fp_use_i:ww #1; #2; { #1; }
```

```
16209 \cs_new:Npn _fp_use_i:www #1; #2; #3; { #1; }
```

(End definition for \\_fp\_use\_i:ww and \\_fp\_use\_i:www.)

## 25.4 Constants, and structure of floating points

`\__fp_misused:n` This receives a floating point object (floating point number or tuple) and generates an error stating that it was misused. This is called when for instance an `fp` variable is left in the input stream and its contents reach TeX's stomach.

```
16210 \cs_new_protected:Npn __fp_misused:n #1
```

```
16211 { __kernel_msg_error:nnx { kernel } { misused-fp } { \fp_to_tl:n {#1} } }
```

(End definition for `\__fp_misused:n`.)

`\s_fp` `\__fp_chk:w` Floating points numbers all start with `\s_fp \__fp_chk:w`, where `\s_fp` is equal to the TeX primitive `\relax`, and `\__fp_chk:w` is protected. The rest of the floating point number is made of characters (or `\relax`). This ensures that nothing expands under `f`-expansion, nor under `x`-expansion. However, when typeset, `\s_fp` does nothing, and `\__fp_chk:w` is expanded. We define `\__fp_chk:w` to produce an error.

```
16212 \scan_new:N \s_fp
```

```
16213 \cs_new_protected:Npn __fp_chk:w #1 ;
```

```
16214 { __fp_misused:n { \s_fp __fp_chk:w #1 ; } }
```

(End definition for `\s_fp` and `\__fp_chk:w`.)

`\s_fp_expr_mark` Aliases of `\tex_relax:D`, used to terminate expressions.

`\s_fp_expr_stop`

```
16215 \scan_new:N \s_fp_expr_mark
```

```
16216 \scan_new:N \s_fp_expr_stop
```

(End definition for `\s_fp_expr_mark` and `\s_fp_expr_stop`.)

`\s_fp_mark` Generic scan marks used throughout the module.

`\s_fp_stop`

```
16217 \scan_new:N \s_fp_mark
```

```
16218 \scan_new:N \s_fp_stop
```

(End definition for `\s_fp_mark` and `\s_fp_stop`.)

`\__fp_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

```
16219 \cs_new:Npn __fp_use_i_delimit_by_s_stop:nw #1 #2 \s_fp_stop {#1}
```

(End definition for `\__fp_use_i_delimit_by_s_stop:nw`.)

`\s_fp_invalid` A couple of scan marks used to indicate where special floating point numbers come from.

`\s_fp_underflow`

```
16220 \scan_new:N \s_fp_invalid
```

`\s_fp_overflow`

```
16221 \scan_new:N \s_fp_underflow
```

`\s_fp_division`

```
16222 \scan_new:N \s_fp_overflow
```

`\s_fp_exact`

```
16223 \scan_new:N \s_fp_division
```

```
16224 \scan_new:N \s_fp_exact
```

(End definition for `\s_fp_invalid` and others.)

`\c_zero_fp` The special floating points. We define the floating points here as “exact”.

`\c_minus_zero_fp`

```
16225 \tl_const:Nn \c_zero_fp { \s_fp __fp_chk:w 0 0 \s_fp_exact ; }
```

`\c_inf_fp`

```
16226 \tl_const:Nn \c_minus_zero_fp { \s_fp __fp_chk:w 0 2 \s_fp_exact ; }
```

`\c_minus_inf_fp`

```
16227 \tl_const:Nn \c_inf_fp { \s_fp __fp_chk:w 2 0 \s_fp_exact ; }
```

`\c_nan_fp`

```
16228 \tl_const:Nn \c_minus_inf_fp { \s_fp __fp_chk:w 2 2 \s_fp_exact ; }
```

```
16229 \tl_const:Nn \c_nan_fp { \s_fp __fp_chk:w 3 1 \s_fp_exact ; }
```

(End definition for `\c_zero_fp` and others. These variables are documented on page 207.)

\c\_fp\_prec\_int The number of digits of floating points.  
16230 \int\_const:Nn \c\_fp\_prec\_int { 16 }  
16231 \int\_const:Nn \c\_fp\_half\_prec\_int { 8 }  
16232 \int\_const:Nn \c\_fp\_block\_int { 4 }  
(End definition for \c\_fp\_prec\_int, \c\_fp\_half\_prec\_int, and \c\_fp\_block\_int.)

\c\_fp\_myriad\_int Blocks have 4 digits so this integer is useful.  
16233 \int\_const:Nn \c\_fp\_myriad\_int { 10000 }  
(End definition for \c\_fp\_myriad\_int.)

\c\_fp\_minus\_min\_exponent\_int  
\c\_fp\_max\_exponent\_int Normal floating point numbers have an exponent between \c\_fp\_minus\_min\_exponent and \c\_fp\_max\_exponent inclusive. Larger numbers are rounded to  $\pm\infty$ . Smaller numbers are rounded to  $\pm 0$ . It would be more natural to define a \c\_fp\_min\_exponent with the opposite sign but that would waste one TeX count.  
16234 \int\_const:Nn \c\_fp\_minus\_min\_exponent\_int { 10000 }  
16235 \int\_const:Nn \c\_fp\_max\_exponent\_int { 10000 }  
(End definition for \c\_fp\_minus\_min\_exponent\_int and \c\_fp\_max\_exponent\_int.)

\c\_fp\_max\_exp\_exponent\_int If a number's exponent is larger than that, its exponential overflows/underflows.  
16236 \int\_const:Nn \c\_fp\_max\_exp\_exponent\_int { 5 }  
(End definition for \c\_fp\_max\_exp\_exponent\_int.)

\c\_fp\_overflowing\_fp A floating point number that is bigger than all normal floating point numbers. This replaces infinities when converting to formats that do not support infinities.  
16237 \tl\_const:Nx \c\_fp\_overflowing\_fp  
16238 {  
16239 \s\_fp \c\_fp\_chk:w 1 0  
16240 { \int\_eval:n { \c\_fp\_max\_exponent\_int + 1 } }  
16241 {1000} {0000} {0000} {0000} ;  
16242 }  
(End definition for \c\_fp\_overflowing\_fp.)

\\_fp\_zero\_fp:N In case of overflow or underflow, we have to output a zero or infinity with a given sign.  
16243 \cs\_new:Npn \\_fp\_zero\_fp:N #1  
16244 { \s\_fp \\_fp\_chk:w 0 #1 \s\_fp\_underflow ; }  
16245 \cs\_new:Npn \\_fp\_inf\_fp:N #1  
16246 { \s\_fp \\_fp\_chk:w 2 #1 \s\_fp\_overflow ; }  
(End definition for \\_fp\_zero\_fp:N and \\_fp\_inf\_fp:N.)

\\_fp\_exponent:w For normal numbers, the function expands to the exponent, otherwise to 0. This is used in l3str-format.  
16247 \cs\_new:Npn \\_fp\_exponent:w \s\_fp \\_fp\_chk:w #1  
16248 {  
16249 \if\_meaning:w 1 #1  
16250 \exp\_after:wN \\_fp\_use\_ii\_until\_s:nnw  
16251 \else:  
16252 \exp\_after:wN \\_fp\_use\_i\_until\_s:nw  
16253 \exp\_after:wN 0  
16254 \fi:  
16255 }

(End definition for `\_fp_exponent:w`.)

`\_fp_neg_sign:N` When appearing in an integer expression or after `\int_value:w`, this expands to the sign opposite to #1, namely 0 (positive) is turned to 2 (negative), 1 (`nan`) to 1, and 2 to 0.

```
16256 \cs_new:Npn _fp_neg_sign:N #1
16257 { _fp_int_eval:w 2 - #1 _fp_int_eval_end: }
```

(End definition for `\_fp_neg_sign:N`.)

`\_fp_kind:w` Expands to 0 for zeros, 1 for normal floating point numbers, 2 for infinities, 3 for `Nan`, 4 for tuples.

```
16258 \cs_new:Npn _fp_kind:w #1
16259 {
16260 _fp_if_type_fp:NTwFw
16261 #1 _fp_use_i_until_s:nnw
16262 \s_fp { _fp_use_i_until_s:nw 4 }
16263 \s_fp_stop
16264 }
```

(End definition for `\_fp_kind:w`.)

## 25.5 Overflow, underflow, and exact zero

`\_fp_sanitize:Nw` `\_fp_sanitize:wN` `\_fp_sanitize_zero:w` Expects the sign and the exponent in some order, then the significand (which we don't touch). Outputs the corresponding floating point number, possibly underflowed to  $\pm 0$  or overflowed to  $\pm\infty$ . The functions `\_fp_underflow:w` and `\_fp_overflow:w` are defined in `l3fp-traps`.

```
16265 \cs_new:Npn _fp_sanitize:Nw #1 #2;
16266 {
16267 \if_case:w
16268 \if_int_compare:w #2 > \c_fp_max_exponent_int 1 ~ \else:
16269 \if_int_compare:w #2 < - \c_fp_minus_min_exponent_int 2 ~ \else:
16270 \if_meaning:w 1 #1 3 ~ \fi: \fi: \fi: 0 ~
16271 \or: \exp_after:wN _fp_overflow:w
16272 \or: \exp_after:wN _fp_underflow:w
16273 \or: \exp_after:wN _fp_sanitize_zero:w
16274 \fi:
16275 \s_fp _fp_chk:w 1 #1 {#2}
16276 }
16277 \cs_new:Npn _fp_sanitize:wN #1; #2 { _fp_sanitize:Nw #2 #1; }
16278 \cs_new:Npn _fp_sanitize_zero:w \s_fp _fp_chk:w #1 #2 #3;
16279 { \c_zero_fp }
```

(End definition for `\_fp_sanitize:Nw`, `\_fp_sanitize:wN`, and `\_fp_sanitize_zero:w`.)

## 25.6 Expanding after a floating point number

`\_fp_exp_after_o:w` *<floating point>*  
`\_fp_exp_after_f:nw` {*<tokens>*} *<floating point>*

Places *<tokens>* (empty in the case of `\_fp_exp_after_o:w`) between the *<floating point>* and the following tokens, then hits those tokens with o or f-expansion, and leaves the floating point number unchanged.

We first distinguish normal floating points, which have a significand, from the much simpler special floating points.

```

16280 \cs_new:Npn __fp_exp_after_o:w \s__fp __fp_chk:w #1
16281 {
16282 \if_meaning:w 1 #1
16283 \exp_after:wN __fp_exp_after_normal:nNnw
16284 \else:
16285 \exp_after:wN __fp_exp_after_special:nNnw
16286 \fi:
16287 { }
16288 #1
16289 }
16290 \cs_new:Npn __fp_exp_after_f:nw #1 \s__fp __fp_chk:w #2
16291 {
16292 \if_meaning:w 1 #2
16293 \exp_after:wN __fp_exp_after_normal:nNnw
16294 \else:
16295 \exp_after:wN __fp_exp_after_special:nNnw
16296 \fi:
16297 { \exp:w \exp_end_continue_f:w #1 }
16298 #2
16299 }
```

(End definition for `\__fp_exp_after_o:w` and `\__fp_exp_after_f:nw`.)

```

__fp_exp_after_special:nNnw __fp_exp_after_special:nNnw {(after)} {case} {sign} {scan mark} ;
Special floating point numbers are easy to jump over since they contain few tokens.
16300 \cs_new:Npn __fp_exp_after_special:nNnw #1#2#3#4;
16301 {
16302 \exp_after:wN \s__fp
16303 \exp_after:wN __fp_chk:w
16304 \exp_after:wN #2
16305 \exp_after:wN #3
16306 \exp_after:wN #4
16307 \exp_after:wN ;
16308 #1
16309 }
```

(End definition for `\__fp_exp_after_special:nNnw`.)

For normal floating point numbers, life is slightly harder, since we have many tokens to jump over. Here it would be slightly better if the digits were not braced but instead were delimited arguments (for instance delimited by ,). That may be changed some day.

```

16310 \cs_new:Npn __fp_exp_after_normal:nNnw #1 1 #2 #3 #4#5#6#7;
16311 {
16312 \exp_after:wN __fp_exp_after_normal:Nwwwww
16313 \exp_after:wN #2
16314 \int_value:w #3 \exp_after:wN ;
16315 \int_value:w 1 #4 \exp_after:wN ;
16316 \int_value:w 1 #5 \exp_after:wN ;
16317 \int_value:w 1 #6 \exp_after:wN ;
16318 \int_value:w 1 #7 \exp_after:wN ; #1
16319 }
16320 \cs_new:Npn __fp_exp_after_normal:Nwwwww
```

```

16321 #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
16322 { \s_fp _fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }

```

(End definition for `\_fp_exp_after_normal:nNnw.`)

## 25.7 Other floating point types

Floating point tuples take the form `\s_fp_tuple \_fp_tuple_chk:w { <fp 1> <fp 2> ... } ;` where each `<fp>` is a floating point number or tuple, hence ends with ; itself. When a tuple is typeset, `\_fp_tuple_chk:w` produces an error, just like usual floating point numbers. Tuples may have zero or one element.

```

16323 \scan_new:N \s_fp_tuple
16324 \cs_new_protected:Npn _fp_tuple_chk:w #1 ;
16325 { _fp_misused:n { \s_fp_tuple _fp_tuple_chk:w #1 ; } }
16326 \tl_const:Nn \c_fp_empty_tuple_fp
16327 { \s_fp_tuple _fp_tuple_chk:w { } ; }

```

(End definition for `\s_fp_tuple`, `\_fp_tuple_chk:w`, and `\c_fp_empty_tuple_fp`.)

Count the number of items in a tuple of floating points by counting semicolons. The technique is very similar to `\tl_count:n`, but with the loop built-in. Checking for the end of the loop is done with the `\use_none:n #1` construction.

```

16328 \cs_new:Npn _fp_array_count:n #1
16329 { _fp_tuple_count:w \s_fp_tuple _fp_tuple_chk:w {#1} ; }
16330 \cs_new:Npn _fp_tuple_count:w \s_fp_tuple _fp_tuple_chk:w #1 ;
16331 {
16332 \int_value:w _fp_int_eval:w 0
16333 _fp_tuple_count_loop:Nw #1 { ? \prg_break: } ;
16334 \prg_break_point:
16335 _fp_int_eval_end:
16336 }
16337 \cs_new:Npn _fp_tuple_count_loop:Nw #1#2;
16338 { \use_none:n #1 + 1 _fp_tuple_count_loop:Nw }

```

(End definition for `\_fp_tuple_count:w`, `\_fp_array_count:n`, and `\_fp_tuple_count_loop:Nw`.)

Used as `\_fp_if_type_fp:NTwFw <marker> {<true code>} \s_fp {<false code>} \s_fp_stop`, this test whether the `<marker>` is `\s_fp` or not and runs the appropriate `<code>`. The very unusual syntax is for optimization purposes as that function is used for all floating point operations.

```
16339 \cs_new:Npn _fp_if_type_fp:NTwFw #1 \s_fp #2 #3 \s_fp_stop {#2}
```

(End definition for `\_fp_if_type_fp:NTwFw`.)

True if all items are floating point numbers. Used for `min`.

```

16340 \cs_new:Npn _fp_array_if_all_fp:nTF #1
16341 {
16342 _fp_array_if_all_fp_loop:w #1 { \s_fp \prg_break: } ;
16343 \prg_break_point: \use_i:nn
16344 }
16345 \cs_new:Npn _fp_array_if_all_fp_loop:w #1#2 ;
16346 {
16347 _fp_if_type_fp:NTwFw
16348 #1 _fp_array_if_all_fp_loop:w

```

```

16349 \s_fp { \prg_break:n \use_iii:nnn }
16350 \s_fp_stop
16351 }

(End definition for \fp_array_if_all_fp:nTF and \fp_array_if_all_fp_loop:w.)
```

\fp\_type\_from\_scan:N Used as \fp\_type\_from\_scan:N *token*. Grabs the pieces of the stringified *token* which lies after the first `s_fp`. If the *token* does not contain that string, the result is `_?`.

```

16352 \cs_new:Npn \fp_type_from_scan:N #1
16353 {
16354 \fp_if_type_fp:NTwFw
16355 #1 { }
16356 \s_fp { \fp_type_from_scan_other:N #1 }
16357 \s_fp_stop
16358 }
16359 \cs_new:Npx \fp_type_from_scan_other:N #1
16360 {
16361 \exp_not:N \exp_after:wN \exp_not:N \fp_type_from_scan:w
16362 \exp_not:N \token_to_str:N #1 \s_fp_mark
16363 \tl_to_str:n { s_fp _? } \s_fp_mark \s_fp_stop
16364 }
16365 \exp_last_unbraced:NNNNo
16366 \cs_new:Npn \fp_type_from_scan:w #1
16367 { \tl_to_str:n { s_fp } } #2 \s_fp_mark #3 \s_fp_stop {#2}
```

(End definition for \fp\_type\_from\_scan:N, \fp\_type\_from\_scan\_other:N, and \fp\_type\_from\_scan:w.)

\fp\_change\_func\_type:NNN \fp\_change\_func\_type\_aux:w \fp\_change\_func\_type\_chk:NNN Arguments are *type marker* *function* *recovery*. This gives the function obtained by placing the type after `@@`. If the function is not defined then *recovery* *function* is used instead; however that test is not run when the *type marker* is `s_fp`.

```

16368 \cs_new:Npn \fp_change_func_type:NNN #1#2#3
16369 {
16370 \fp_if_type_fp:NTwFw
16371 #1 #2
16372 \s_fp
16373 {
16374 \exp_after:wN \fp_change_func_type_chk:NNN
16375 \cs:w
16376 __fp \fp_type_from_scan_other:N #1
16377 \exp_after:wN \fp_change_func_type_aux:w \token_to_str:N #2
16378 \cs_end:
16379 #2 #3
16380 }
16381 \s_fp_stop
16382 }
16383 \exp_last_unbraced:NNNNo
16384 \cs_new:Npn \fp_change_func_type_aux:w #1 { \tl_to_str:n { __fp } } { }
16385 \cs_new:Npn \fp_change_func_type_chk:NNN #1#2#3
16386 {
16387 \if_meaning:w \scan_stop: #1
16388 \exp_after:wN #3 \exp_after:wN #2
16389 \else:
```

```

16390 \exp_after:wN #1
16391 \fi:
16392 }

```

(End definition for `\_fp_change_func_type:NNN`, `\_fp_change_func_type_aux:w`, and `\_fp_change_func_type_chk:NNN`.)

The `Nnw` function simply dispatches to the appropriate `\_fp_exp_after..._f:nw` with “`...`” (either empty or `_<type>`) extracted from `#1`, which should start with `\s_fp`. If it doesn’t start with `\s_fp` the function `\_fp_exp_after_?_f:nw` defined in `l3fp-parse` gives an error; another special `<type>` is `stop`, useful for loops, see below. The `nw` function has an important optimization for floating points numbers; it also fetches its type marker `#2` from the floating point.

```

16393 \cs_new:Npn _fp_exp_after_any_f:Nnw #1
16394 { \cs:w _fp_exp_after _fp_type_from_scan_other:N #1 _f:nw \cs_end: }
16395 \cs_new:Npn _fp_exp_after_any_f:nw #1#2
16396 {
16397 _fp_if_type_fp:NTwFw
16398 #2 _fp_exp_after_f:nw
16399 \s_fp { _fp_exp_after_any_f:Nnw #2 }
16400 \s_fp_stop
16401 {#1} #2
16402 }
16403 \cs_new_eq:NN _fp_exp_after_expr_stop_f:nw \use_none:nn

```

(End definition for `\_fp_exp_after_any_f:Nnw`, `\_fp_exp_after_any_f:nw`, and `\_fp_exp_after_expr_stop_f:nw`.)

The loop works by using the `n` argument of `\_fp_exp_after_any_f:nw` to place the loop macro after the next item in the tuple and expand it.

```

_fp_exp_after_array_f:w
⟨fp1⟩ ;
...
⟨fpn⟩ ;
\s_fp_expr_stop

16404 \cs_new:Npn _fp_exp_after_tuple_o:w
16405 { _fp_exp_after_tuple_f:nw { \exp_after:wN \exp_stop_f: } }
16406 \cs_new:Npn _fp_exp_after_tuple_f:nw
16407 #1 \s_fp_tuple _fp_tuple_chk:w #2 ;
16408 {
16409 \exp_after:wN \s_fp_tuple
16410 \exp_after:wN _fp_tuple_chk:w
16411 \exp_after:wN {
16412 \exp:w \exp_end_continue_f:w
16413 _fp_exp_after_array_f:w #2 \s_fp_expr_stop
16414 \exp_after:wN }
16415 \exp_after:wN ;
16416 \exp:w \exp_end_continue_f:w #1
16417 }
16418 \cs_new:Npn _fp_exp_after_array_f:w
16419 { _fp_exp_after_any_f:nw { _fp_exp_after_array_f:w } }

```

(End definition for `\_fp_exp_after_tuple_o:w`, `\_fp_exp_after_tuple_f:nw`, and `\_fp_exp_after_array_f:w`.)

## 25.8 Packing digits

When a positive integer #1 is known to be less than  $10^8$ , the following trick splits it into two blocks of 4 digits, padding with zeros on the left.

```
\cs_new:Npn \pack:NNNNw #1 #2#3#4#5 #6; { {#2#3#4#5} {#6} }
\exp_after:wN \pack:NNNNw
 \fp_int_value:w \fp_int_eval:w 1 0000 0000 + #1 ;
```

The idea is that adding  $10^8$  to the number ensures that it has exactly 9 digits, and can then easily find which digits correspond to what position in the number. Of course, this can be modified for any number of digits less or equal to 9 (we are limited by TeX's integers). This method is very heavily relied upon in `l3fp-basics`.

More specifically, the auxiliary inserts `+ #1#2#3#4#5 ; {#6}`, which allows us to compute several blocks of 4 digits in a nested manner, performing carries on the fly. Say we want to compute  $12345 \times 66778899$ . With simplified names, we would do

```
\exp_after:wN \post_processing:w
\fp_int_value:w \fp_int_eval:w - 5 0000
\exp_after:wN \pack:NNNNw
\fp_int_value:w \fp_int_eval:w 4 9995 0000
+ 12345 * 6677
\exp_after:wN \pack:NNNNw
\fp_int_value:w \fp_int_eval:w 5 0000 0000
+ 12345 * 8899 ;
```

The `\exp_after:wN` triggers `\int_value:w \fp_int_eval:w`, which starts a first computation, whose initial value is  $-50000$  (the “leading shift”). In that computation appears an `\exp_after:wN`, which triggers the nested computation `\int_value:w \fp_int_eval:w` with starting value  $499950000$  (the “middle shift”). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation's value is  $500000000 + 12345 \times 8899$ , which has 9 digits. Adding  $5 \cdot 10^8$  to the product allowed us to know how many digits to expect as long as the numbers to multiply are not too big; it also works to some extent with negative results. The `pack` function puts the last 4 of those 9 digits into a brace group, moves the semi-colon delimiter, and inserts a `+`, which combines the carry with the previous computation. The shifts nicely combine into  $500000000/10^4 + 499950000 = 500000000$ . As long as the operands are in some range, the result of this second computation has 9 digits. The corresponding `pack` function, expanded after the result is computed, braces the last 4 digits, and leaves `+ <5 digits>` for the initial computation. The “leading shift” cancels the combination of the other shifts, and the `\post_processing:w` takes care of packing the last few digits.

Admittedly, this is quite intricate. It is probably the key in making l3fp as fast as other pure TeX floating point units despite its increased precision. In fact, this is used so much that we provide different sets of packing functions and shifts, depending on ranges of input.

This set of shifts allows for computations involving results in the range  $[-4 \cdot 10^8, 5 \cdot 10^8 - 1]$ . Shifted values all have exactly 9 digits.

```
16420 \int_const:Nn \c_fp_leading_shift_int { - 5 0000 }
16421 \int_const:Nn \c_fp_middle_shift_int { 5 0000 * 9999 }
16422 \int_const:Nn \c_fp_trailing_shift_int { 5 0000 * 10000 }
16423 \cs_new:Npn \fp_pack:NNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }
```

(End definition for `\_fp_pack:NNNNNw` and others.)

```
_fp_pack_big:NNNNNw
\c_fp_big_trailing_shift_int
\c_fp_big_middle_shift_int
\c_fp_big_leading_shift_int
```

This set of shifts allows for computations involving results in the range  $[-5 \cdot 10^8, 6 \cdot 10^8 - 1]$  (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper bound is due to TeX's limit of  $2^{31} - 1$  on integers. The shifts are chosen to be roughly the mid-point of  $10^9$  and  $2^{31}$ , the two bounds on 10-digit integers in TeX.

```
16424 \int_const:Nn \c_fp_big_leading_shift_int { - 15 2374 }
16425 \int_const:Nn \c_fp_big_middle_shift_int { 15 2374 * 9999 }
16426 \int_const:Nn \c_fp_big_trailing_shift_int { 15 2374 * 10000 }
16427 \cs_new:Npn _fp_pack_big:NNNNNw #1#2 #3#4#5#6 #7;
16428 { + #1#2#3#4#5#6 ; {#7} }
```

(End definition for `\_fp_pack_big:NNNNNw` and others.)

```
_fp_pack_Bigg:NNNNNNw
\c_fp_Bigg_trailing_shift_int
\c_fp_Bigg_middle_shift_int
\c_fp_Bigg_leading_shift_int
```

This set of shifts allows for computations with results in the range  $[-1 \cdot 10^9, 147483647]$ ; the end-point is  $2^{31} - 1 - 2 \cdot 10^9 \simeq 1.47 \cdot 10^8$ . Shifted values all have exactly 10 digits.

```
16429 \int_const:Nn \c_fp_Bigg_leading_shift_int { - 20 0000 }
16430 \int_const:Nn \c_fp_Bigg_middle_shift_int { 20 0000 * 9999 }
16431 \int_const:Nn \c_fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
16432 \cs_new:Npn _fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
16433 { + #1#2#3#4#5#6 ; {#7} }
```

(End definition for `\_fp_pack_Bigg:NNNNNNw` and others.)

```
_fp_pack_twice_four:wNNNNNNNNN
\c_fp_pack_twice_four:wNNNNNNNNN
```

Grabs two sets of 4 digits and places them before the semi-colon delimiter. Putting several copies of this function before a semicolon packs more digits since each takes the digits packed by the others in its first argument.

```
16434 \cs_new:Npn _fp_pack_twice_four:wNNNNNNNNN #1; #2#3#4#5 #6#7#8#9
16435 { #1 {#2#3#4#5} {#6#7#8#9} ; }
```

(End definition for `\_fp_pack_twice_four:wNNNNNNNNN`.)

```
_fp_pack_eight:wNNNNNNNNN
```

Grabs one set of 8 digits and places them before the semi-colon delimiter as a single group. Putting several copies of this function before a semicolon packs more digits since each takes the digits packed by the others in its first argument.

```
16436 \cs_new:Npn _fp_pack_eight:wNNNNNNNNN #1; #2#3#4#5 #6#7#8#9
16437 { #1 {#2#3#4#5#6#7#8#9} ; }
```

(End definition for `\_fp_pack_eight:wNNNNNNNNN`.)

```
_fp_basics_pack_low:NNNNNw
\c_fp_basics_pack_high:NNNNNw
\c_fp_basics_pack_high_carry:w
```

Addition and multiplication of significands are done in two steps: first compute a (more or less) exact result, then round and pack digits in the final (braced) form. These functions take care of the packing, with special attention given to the case where rounding has caused a carry. Since rounding can only shift the final digit by 1, a carry always produces an exact power of 10. Thus, `\_fp_basics_pack_high_carry:w` is always followed by four times `{0000}`.

This is used in l3fp-basics and l3fp-extended.

```
16438 \cs_new:Npn _fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6;
16439 { + #1 - 1 ; {#2#3#4#5} {#6} ; }
16440 \cs_new:Npn _fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6;
16441 {
```

```

16442 \if_meaning:w 2 #1
16443 _fp_basics_pack_high_carry:w
16444 \fi:
16445 ; {#2#3#4#5} {#6}
16446 }
16447 \cs_new:Npn _fp_basics_pack_high_carry:w \fi: ; #1
16448 { \fi: + 1 ; {1000} }

(End definition for _fp_basics_pack_low:NNNNw, _fp_basics_pack_high:NNNNw, and _fp-
basics_pack_high_carry:w.)

```

\\_fp\_basics\_pack\_weird\_low:NNNNw  
\\_fp\_basics\_pack\_weird\_high:NNNNNNNNw

This is used in l3fp-basics for additions and divisions. Their syntax is confusing, hence the name.

```

16449 \cs_new:Npn _fp_basics_pack_weird_low:NNNNw #1 #2#3#4 #5;
16450 {
16451 \if_meaning:w 2 #1
16452 + 1
16453 \fi:
16454 _fp_int_eval_end:
16455 #2#3#4; {#5} ;
16456 }
16457 \cs_new:Npn _fp_basics_pack_weird_high:NNNNNNNNw
16458 1 #1#2#3#4 #5#6#7#8 #9; { ; {#1#2#3#4} {#5#6#7#8} {#9} }

(End definition for _fp_basics_pack_weird_low:NNNNw and _fp_basics_pack_weird_high:NNNNNNNNw.)

```

## 25.9 Decimate (dividing by a power of 10)

\\_fp\_decimate:nNnnn {⟨shift⟩} ⟨f<sub>1</sub>⟩  
{⟨X<sub>1</sub>⟩} {⟨X<sub>2</sub>⟩} {⟨X<sub>3</sub>⟩} {⟨X<sub>4</sub>⟩}

Each ⟨X<sub>i</sub>⟩ consists in 4 digits exactly, and 1000 ≤ ⟨X<sub>1</sub>⟩ < 9999. The first argument determines by how much we shift the digits. ⟨f<sub>1</sub>⟩ is called as follows:

⟨f<sub>1</sub>⟩ ⟨rounding⟩ {⟨X'<sub>1</sub>⟩} {⟨X'<sub>2</sub>⟩} ⟨extra-digits⟩ ;

where 0 ≤ ⟨X'<sub>i</sub>⟩ < 10<sup>8</sup> − 1 are 8 digit integers, forming the truncation of our number. In other words,

$$\left( \sum_{i=1}^4 \langle X_i \rangle \cdot 10^{-4i} \cdot 10^{-\langle shift \rangle} \right) - (\langle X'_1 \rangle \cdot 10^{-8} + \langle X'_2 \rangle \cdot 10^{-16}) = 0. \langle extra-digits \rangle \cdot 10^{-16} \in [0, 10^{-16}).$$

To round properly later, we need to remember some information about the difference. The ⟨rounding⟩ digit is 0 if and only if the difference is exactly 0, and 5 if and only if the difference is exactly  $0.5 \cdot 10^{-16}$ . Otherwise, it is the (non-0, non-5) digit closest to  $10^{17}$  times the difference. In particular, if the shift is 17 or more, all the digits are dropped, ⟨rounding⟩ is 1 (not 0), and ⟨X'<sub>1</sub>⟩ and ⟨X'<sub>2</sub>⟩ are both zero.

If the shift is 1, the ⟨rounding⟩ digit is simply the only digit that was pushed out of the brace groups (this is important for subtraction). It would be more natural for the ⟨rounding⟩ digit to be placed after the ⟨X'<sub>i</sub>⟩, but the choice we make involves less reshuffling.

Note that this function treats negative ⟨shift⟩ as 0.

```

16459 \cs_new:Npn _fp_decimate:nNnnn #1
16460 {

```

```

16461 \cs:w
16462 __fp_decimate_
16463 \if_int_compare:w __fp_int_eval:w #1 > \c__fp_prec_int
16464 tiny
16465 \else:
16466 __fp_int_to_roman:w __fp_int_eval:w #1
16467 \fi:
16468 :Nnnnn
16469 \cs_end:
16470 }

```

Each of the auxiliaries see the function  $\langle f_1 \rangle$ , followed by 4 blocks of 4 digits.

(End definition for  $\_\_fp\_decimate:nNnnnn$ .)

If the  $\langle shift \rangle$  is zero, or too big, life is very easy.

```

16471 \cs_new:Npn __fp_decimate_:Nnnnn #1 #2#3#4#5
16472 { #1 0 {#2#3} {#4#5} ; }
16473 \cs_new:Npn __fp_decimate_tiny:Nnnnn #1 #2#3#4#5
16474 { #1 1 { 0000 0000 } { 0000 0000 } 0 #2#3#4#5 ; }

```

(End definition for  $\_\_fp\_decimate_:Nnnnn$  and  $\_\_fp\_decimate_tiny:Nnnnn$ .)

$\_\_fp\_decimate\_auxi:Nnnnn$

Shifting happens in two steps: compute the  $\langle rounding \rangle$  digit, and repack digits into two blocks of 8. The sixteen functions are very similar, and defined through  $\_\_fp\_tmp:w$ . The arguments are as follows: #1 indicates which function is being defined; after one step of expansion, #2 yields the “extra digits” which are then converted by  $\_\_fp\_round\_digit:Nw$  to the  $\langle rounding \rangle$  digit (note the + separating blocks of digits to avoid overflowing TeX’s integers). This triggers the f-expansion of  $\_\_fp\_decimate\_pack:nnnnnnnnnw$ ,<sup>8</sup> responsible for building two blocks of 8 digits, and removing the rest. For this to work, #3 alternates between braced and unbraced blocks of 4 digits, in such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.

```

16475 \cs_new:Npn __fp_tmp:w #1 #2 #3
16476 {
16477 \cs_new:cpn { __fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
16478 {
16479 \exp_after:wN ##1
16480 \int_value:w
16481 \exp_after:wN __fp_round_digit:Nw #2 ;
16482 __fp_decimate_pack:nnnnnnnnnw #3 ;
16483 }
16484 }
16485 __fp_tmp:w {i} {\use_none:nnn #50}{ 0{#2}#3{#4}#5 }}
16486 __fp_tmp:w {ii} {\use_none:nn #5 }{ 00{#2}#3{#4}#5 }}
16487 __fp_tmp:w {iii} {\use_none:n #5 }{ 000{#2}#3{#4}#5 }}
16488 __fp_tmp:w {iv} { #5 }{ {0000}#2{#3}#4 #5 }}
16489 __fp_tmp:w {v} {\use_none:nnn #4#5 }{ 0{0000}#2{#3}#4 #5 }}
16490 __fp_tmp:w {vi} {\use_none:nn #4#5 }{ 00{0000}#2{#3}#4 #5 }}
16491 __fp_tmp:w {vii} {\use_none:n #4#5 }{ 000{0000}#2{#3}#4 #5 }}
16492 __fp_tmp:w {viii}{ #4#5 }{ {0000}0000{#2}#3 #4 #5 }}
16493 __fp_tmp:w {ix} {\use_none:nnn #3#4+#5}{ 0{0000}0000{#2}#3 #4 #5 }}

```

---

<sup>8</sup>No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

```

16494 __fp_tmp:w {x} {\use_none:nn #3#4+#5}{ 00{0000}0000{#2}#3 #4 #5 }
16495 __fp_tmp:w {xi} {\use_none:n #3#4+#5}{ 000{0000}0000{#2}#3 #4 #5 }
16496 __fp_tmp:w {xii} { #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5 }
16497 __fp_tmp:w {xiii}{\use_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5 }
16498 __fp_tmp:w {xiv} {\use_none:nn #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5 }
16499 __fp_tmp:w {xv} {\use_none:n #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5 }
16500 __fp_tmp:w {xvi} { #2#3+#4#5}{ {0000}0000{0000}0000 #2 #3 #4 #5}

```

(End definition for `\__fp_decimate_auxi:Nnnnn` and others.)

`\__fp_decimate_pack:nnnnnnnnnw`

The computation of the `\langle rounding` digit leaves an unfinished `\int_value:w`, which expands the following functions. This allows us to repack nicely the digits we keep. Those digits come as an alternation of unbraced and braced blocks of 4 digits, such that the first 5 groups of token consist in 4 single digits, and one brace group (in some order), and the next 5 have the same structure. This is followed by some digits and a semicolon.

```

16501 \cs_new:Npn __fp_decimate_pack:nnnnnnnnnw #1#2#3#4#5
16502 { __fp_decimate_pack:nnnnnnnw { #1#2#3#4#5 } }
16503 \cs_new:Npn __fp_decimate_pack:nnnnnnw #1 #2#3#4#5#6
16504 { {#1} {#2#3#4#5#6} }

```

(End definition for `\__fp_decimate_pack:nnnnnnnnnnw`.)

## 25.10 Functions for use within primitive conditional branches

The functions described in this section are not pretty and can easily be misused. When correctly used, each of them removes one `\fi:` as part of its parameter text, and puts one back as part of its replacement text.

Many computation functions in l3fp must perform tests on the type of floating points that they receive. This is often done in an `\if_case:w` statement or another conditional statement, and only a few cases lead to actual computations: most of the special cases are treated using a few standard functions which we define now. A typical use context for those functions would be

```

\if_case:w <integer> \exp_stop_f:
 __fp_case_return_o:Nw <fp var>
\or: __fp_case_use:nw <some computation>
\or: __fp_case_return_same_o:w
\or: __fp_case_return:nw <something>
\fi:
<junk>
<floating point>

```

In this example, the case 0 returns the floating point `<fp var>`, expanding once after that floating point. Case 1 does `<some computation>` using the `<floating point>` (presumably compute the operation requested by the user in that non-trivial case). Case 2 returns the `<floating point>` without modifying it, removing the `<junk>` and expanding once after. Case 3 closes the conditional, removes the `<junk>` and the `<floating point>`, and expands `<something>` next. In other cases, the “`<junk>`” is expanded, performing some other operation on the `<floating point>`. We provide similar functions with two trailing `<floating points>`.

\\_fp\_case\_use:nw This function ends a TeX conditional, removes junk until the next floating point, and places its first argument before that floating point, to perform some operation on the floating point.

```
16505 \cs_new:Npn _fp_case_use:nw #1#2 \fi: #3 \s_fp { \fi: #1 \s_fp }
```

(End definition for \\_fp\_case\_use:nw.)

\\_fp\_case\_return:nw This function ends a TeX conditional, removes junk and a floating point, and places its first argument in the input stream. A quirk is that we don't define this function requiring a floating point to follow, simply anything ending in a semicolon. This, in turn, means that the *junk* may not contain semicolons.

```
16506 \cs_new:Npn _fp_case_return:nw #1#2 \fi: #3 ; { \fi: #1 }
```

(End definition for \\_fp\_case\_return:nw.)

\\_fp\_case\_return\_o:Nw This function ends a TeX conditional, removes junk and a floating point, and returns its first argument (an *fp var*) then expands once after it.

```
16507 \cs_new:Npn _fp_case_return_o:Nw #1#2 \fi: #3 \s_fp #4 ;
16508 { \fi: \exp_after:wN #1 }
```

(End definition for \\_fp\_case\_return\_o:Nw.)

\\_fp\_case\_return\_same\_o:w This function ends a TeX conditional, removes junk, and returns the following floating point, expanding once after it.

```
16509 \cs_new:Npn _fp_case_return_same_o:w #1 \fi: #2 \s_fp
16510 { \fi: \fp_exp_after_o:w \s_fp }
```

(End definition for \\_fp\_case\_return\_same\_o:w.)

\\_fp\_case\_return\_o:Nww Same as \\_fp\_case\_return\_o:Nw but with two trailing floating points.

```
16511 \cs_new:Npn _fp_case_return_o:Nww #1#2 \fi: #3 \s_fp #4 ; #5 ;
16512 { \fi: \exp_after:wN #1 }
```

(End definition for \\_fp\_case\_return\_o:Nww.)

\\_fp\_case\_return\_i\_o:ww \\_fp\_case\_return\_ii\_o:ww Similar to \\_fp\_case\_return\_same\_o:w, but this returns the first or second of two trailing floating point numbers, expanding once after the result.

```
16513 \cs_new:Npn _fp_case_return_i_o:ww #1 \fi: #2 \s_fp #3 ; \s_fp #4 ;
16514 { \fi: \fp_exp_after_o:w \s_fp #3 ; }
16515 \cs_new:Npn _fp_case_return_ii_o:ww #1 \fi: #2 \s_fp #3 ;
16516 { \fi: \fp_exp_after_o:w }
```

(End definition for \\_fp\_case\_return\_i\_o:ww and \\_fp\_case\_return\_ii\_o:ww.)

## 25.11 Integer floating points

`\_fp_int_p:w` Tests if the floating point argument is an integer. For normal floating point numbers, this holds if the rounding digit resulting from `\_fp_decimate:nNnnn` is 0.

```

16517 \prg_new_conditional:Npnn _fp_int:w \s_fp _fp_chk:w #1 #2 #3 #4;
16518 { TF , T , F , p }
16519 {
16520 \if_case:w #1 \exp_stop_f:
16521 \prg_return_true:
16522 \or:
16523 \if_charcode:w 0
16524 _fp_decimate:nNnnn { \c_fp_prec_int - #3 }
16525 _fp_use_i_until_s:nw #4
16526 \prg_return_true:
16527 \else:
16528 \prg_return_false:
16529 \fi:
16530 \else: \prg_return_false:
16531 \fi:
16532 }
```

(End definition for `\_fp_int:wTF`.)

## 25.12 Small integer floating points

`\_fp_small_int:wTF` `\_fp_small_int_true:wTF` `\_fp_small_int_normal:NnwTF` `\_fp_small_int_test:NnnwNTF` Tests if the floating point argument is an integer or  $\pm\infty$ . If so, it is clipped to an integer in the range  $[-10^8, 10^8]$  and fed as a braced argument to the *true code*. Otherwise, the *false code* is performed.

First filter special cases: zeros and infinities are integers, `nan` is not. For normal numbers, decimate. If the rounding digit is not 0 run the *false code*. If it is, then the integer is #2 #3; use #3 if #2 vanishes and otherwise  $10^8$ .

```

16533 \cs_new:Npn _fp_small_int:wTF \s_fp _fp_chk:w #1#2
16534 {
16535 \if_case:w #1 \exp_stop_f:
16536 _fp_case_return:nw { _fp_small_int_true:wTF 0 ; }
16537 \or: \exp_after:wN _fp_small_int_normal:NnwTF
16538 \or:
16539 _fp_case_return:nw
16540 {
16541 \exp_after:wN _fp_small_int_true:wTF \int_value:w
16542 \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
16543 }
16544 \else: _fp_case_return:nw \use_i:nn
16545 \fi:
16546 #2
16547 }
16548 \cs_new:Npn _fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
16549 \cs_new:Npn _fp_small_int_normal:NnwTF #1#2#3;
16550 {
16551 _fp_decimate:nNnnn { \c_fp_prec_int - #2 }
16552 _fp_small_int_test:NnnwNw
16553 #3 #1
16554 }
```

```

16555 \cs_new:Npn __fp_small_int_test:NnnwNw #1#2#3#4; #5
16556 {
16557 \if_meaning:w 0 #1
16558 \exp_after:wN __fp_small_int_true:wTF
16559 \int_value:w \if_meaning:w 2 #5 - \fi:
16560 \if_int_compare:w #2 > 0 \exp_stop_f:
16561 1 0000 0000
16562 \else:
16563 #3
16564 \fi:
16565 \exp_after:wN ;
16566 \else:
16567 \exp_after:wN \use_i:nn
16568 \fi:
16569 }

```

(End definition for `\__fp_small_int:wTF` and others.)

### 25.13 Fast string comparison

`\__fp_str_if_eq:nn` A private version of the low-level string comparison function.

```
16570 \cs_new_eq:NN __fp_str_if_eq:nn \tex_strcmp:D
```

(End definition for `\__fp_str_if_eq:nn`.)

### 25.14 Name of a function from its **I3fp-parse** name

The goal is to convert for instance `\__fp_sin_o:w` to `sin`. This is used in error messages hence does not need to be fast.

```

16571 \cs_new:Npn __fp_func_to_name:N #1
16572 {
16573 \exp_last_unbraced:Nf
16574 __fp_func_to_name_aux:w { \cs_to_str:N #1 } X
16575 }
16576 \cs_set_protected:Npn __fp_tmp:w #1 #
16577 { \cs_new:Npn __fp_func_to_name_aux:w ##1 #1 ##2 #2 ##3 X {##2} }
16578 \exp_args:Nff __fp_tmp:w { \tl_to_str:n { __fp_ } }
16579 { \tl_to_str:n { _o: } }

```

(End definition for `\__fp_func_to_name:N` and `\__fp_func_to_name_aux:w`.)

### 25.15 Messages

Using a floating point directly is an error.

```

16580 __kernel_msg_new:nnnn { kernel } { misused-fp }
16581 { A-floating-point-with-value-'#1'~was~misused. }
16582 {
16583 To~obtain~the~value~of~a~floating~point~variable,~use~
16584 '\token_to_str:N \fp_to_decimal:N',~
16585 '\token_to_str:N \fp_to_tl:N',~or~other~
16586 conversion~functions.
16587 }
16588
```

/package)

## 26 I3fp-traps Implementation

```
16589 /*package)
```

```
16590 @@=fp)
```

Exceptions should be accessed by an n-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

### 26.1 Flags

Flags to denote exceptions.

```
16591 \flag_new:n { fp_invalid_operation }
16592 \flag_new:n { fp_division_by_zero }
16593 \flag_new:n { fp_overflow }
16594 \flag_new:n { fp_underflow }
```

(End definition for flag `fp_invalid_operation` and others. These variables are documented on page 209.)

### 26.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an N-type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives  $+\infty$ . Currently, the inexact exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions

- `\_fp_invalid_operation:nnw,`
- `\_fp_invalid_operation_o:Nww,`
- `\_fp_invalid_operation_tl_o:ff,`
- `\_fp_division_by_zero_o:Nnw,`
- `\_fp_division_by_zero_o:NNww,`
- `\_fp_overflow:w,`
- `\_fp_underflow:w.`

Rather than changing them directly, we provide a user interface as `\fp_trap:nn {exception} {way of trapping}`, where the `way of trapping` is one of `error`, `flag`, or `none`.

We also provide `\_fp_invalid_operation_o:nw`, defined in terms of `\_fp_invalid_operation:nnw`.

```

\fp_trap:nn
 16595 \cs_new_protected:Npn \fp_trap:nn #1#2
 16596 {
 16597 \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
 16598 {
 16599 \clist_if_in:nTF
 16600 { invalid_operation , division_by_zero , overflow , underflow }
 16601 {#1}
 16602 {
 16603 __kernel_msg_error:nnxx { kernel }
 16604 { unknown-fpu-trap-type } {#1} {#2}
 16605 }
 16606 {
 16607 __kernel_msg_error:nnx
 16608 { kernel } { unknown-fpu-exception } {#1}
 16609 }
 16610 }
 16611 }

```

(End definition for `\fp_trap:nn`. This function is documented on page 209.)

We provide three types of trapping for invalid operations: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases, the function produces as a result its first argument, possibly with post-expansion.

```

\fp_trap_invalid_operation_set_error:
\fp_trap_invalid_operation_set_flag:
\fp_trap_invalid_operation_set_none:
\fp_trap_invalid_operation_set:N
 16612 \cs_new_protected:Npn \fp_trap_invalid_operation_set_error:
 16613 { \fp_trap_invalid_operation_set:N \prg_do_nothing: }
 16614 \cs_new_protected:Npn \fp_trap_invalid_operation_set_flag:
 16615 { \fp_trap_invalid_operation_set:N \use_none:nnnnn }
 16616 \cs_new_protected:Npn \fp_trap_invalid_operation_set_none:
 16617 { \fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
 16618 \cs_new_protected:Npn \fp_trap_invalid_operation_set:N #1
 16619 {
 16620 \exp_args:Nno \use:n
 16621 { \cs_set:Npn \fp_invalid_operation:nw ##1##2##3; }
 16622 {
 16623 #1
 16624 \fp_error:nfn { fp-invalid } {##2} { \fp_to_tl:n {##3; } } { }
 16625 \flag_raise_if_clear:n { fp_invalid_operation }
 16626 ##1
 16627 }
 16628 \exp_args:Nno \use:n
 16629 { \cs_set:Npn \fp_invalid_operation_o:Nww ##1##2; ##3; }
 16630 {
 16631 #1
 16632 \fp_error:nffn { fp-invalid-ii }
 16633 { \fp_to_tl:n {##2; } } { \fp_to_tl:n {##3; } } {##1}
 16634 \flag_raise_if_clear:n { fp_invalid_operation }
 16635 \exp_after:wN \c_nan_fp
 16636 }
 16637 \exp_args:Nno \use:n
 16638 { \cs_set:Npn \fp_invalid_operation_tl_o:ff ##1##2 }
 16639 {
 16640 #1
 16641 \fp_error:nffn { fp-invalid } {##1} {##2} { }

```

```

16642 \flag_raise_if_clear:n { fp_invalid_operation }
16643 \exp_after:wN \c_nan_fp
16644 }
16645 }

```

(End definition for `\_fp_trap_invalid_operation_set_error: and others.`)

```

\fp_trap_division_by_zero_set_error:
\fp_trap_division_by_zero_set_flag:
\fp_trap_division_by_zero_set_none:
\fp_trap_division_by_zero_set:N

```

We provide three types of trapping for invalid operations and division by zero: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In all cases, the function must produce a result, namely its first argument,  $\pm\infty$  or NaN.

```

16646 \cs_new_protected:Npn _fp_trap_division_by_zero_set_error:
16647 { _fp_trap_division_by_zero_set:N \prg_do_nothing: }
16648 \cs_new_protected:Npn _fp_trap_division_by_zero_set_flag:
16649 { _fp_trap_division_by_zero_set:N \use_none:nnnnn }
16650 \cs_new_protected:Npn _fp_trap_division_by_zero_set_none:
16651 { _fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
16652 \cs_new_protected:Npn _fp_trap_division_by_zero_set:N #1
16653 {
16654 \exp_args:Nno \use:n
16655 { \cs_set:Npn _fp_division_by_zero_o:Nnw ##1##2##3; }
16656 {
16657 #1
16658 _fp_error:nfn { fp-zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
16659 \flag_raise_if_clear:n { fp_division_by_zero }
16660 \exp_after:wN ##1
16661 }
16662 \exp_args:Nno \use:n
16663 { \cs_set:Npn _fp_division_by_zero_o>NNww ##1##2##3; ##4; }
16664 {
16665 #1
16666 _fp_error:nffn { fp-zero-div-ii }
16667 { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
16668 \flag_raise_if_clear:n { fp_division_by_zero }
16669 \exp_after:wN ##1
16670 }
16671 }

```

(End definition for `\_fp_trap_division_by_zero_set_error: and others.`)

```

\fp_trap_overflow_set_error:
\fp_trap_overflow_set_flag:
\fp_trap_overflow_set_none:
\fp_trap_overflow_set:N
\fp_trap_underflow_set_error:
\fp_trap_underflow_set_flag:
\fp_trap_underflow_set_none:
\fp_trap_underflow_set:N
\fp_trap_overflow_set:NnNn

```

Just as for invalid operations and division by zero, the three different behaviours are obtained by feeding `\prg_do_nothing:`, `\use_none:nnnnn` or `\use_none:nnnnnnn` to an auxiliary, with a further auxiliary common to overflow and underflow functions. In most cases, the argument of the `\_fp_overflow:w` and `\_fp_underflow:w` functions will be an (almost) normal number (with an exponent outside the allowed range), and the error message thus displays that number together with the result to which it overflowed or underflowed. For extreme cases such as  $10^{**} 1e9999$ , the exponent would be too large for TeX, and `\_fp_overflow:w` receives  $\pm\infty$  (`\_fp_underflow:w` would receive  $\pm 0$ ); then we cannot do better than simply say an overflow or underflow occurred.

```

16672 \cs_new_protected:Npn _fp_trap_overflow_set_error:
16673 { _fp_trap_overflow_set:N \prg_do_nothing: }
16674 \cs_new_protected:Npn _fp_trap_overflow_set_flag:
16675 { _fp_trap_overflow_set:N \use_none:nnnnn }
16676 \cs_new_protected:Npn _fp_trap_overflow_set_none:

```

```

16677 { __fp_trap_overflow_set:N \use_none:nnnnnnn }
16678 \cs_new_protected:Npn __fp_trap_overflow_set:N #1
16679 { __fp_trap_overflow_set:NnNn #1 { overflow } __fp_inf_fp:N { inf } }
16680 \cs_new_protected:Npn __fp_trap_underflow_set_error:
16681 { __fp_trap_underflow_set:N \prg_do_nothing: }
16682 \cs_new_protected:Npn __fp_trap_underflow_set_flag:
16683 { __fp_trap_underflow_set:N \use_none:nnnnnn }
16684 \cs_new_protected:Npn __fp_trap_underflow_set_none:
16685 { __fp_trap_underflow_set:N \use_none:nnnnnnn }
16686 \cs_new_protected:Npn __fp_trap_underflow_set:N #1
16687 { __fp_trap_overflow_set:NnNn #1 { underflow } __fp_zero_fp:N { 0 } }
16688 \cs_new_protected:Npn __fp_trap_overflow_set:NnNn #1#2#3#4
16689 {
16690 \exp_args:Nno \use:n
16691 { \cs_set:cpn { __fp_#2 :w } \s_fp __fp_chk:w ##1##2##3; }
16692 {
16693 #1
16694 __fp_error:nffn
16695 { fp-flow \if_meaning:w 1 ##1 -to \fi: }
16696 { \fp_to_tl:n { \s_fp __fp_chk:w ##1##2##3; } }
16697 { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
16698 {#2}
16699 \flag_raise_if_clear:n { fp_#2 }
16700 #3 ##2
16701 }
16702 }

```

(End definition for `\__fp_trap_overflow_set_error:` and others.)

```

__fp_invalid_operation:nw
 __fp_invalid_operation_o:Nww
 __fp_invalid_operation_tl_o:ff
__fp_division_by_zero_o:Nnw
 __fp_division_by_zero_o:NNww
 __fp_overflow:w
 __fp_underflow:w

```

Initialize the control sequences (to log properly their existence). Then set invalid operations to trigger an error, and division by zero, overflow, and underflow to act silently on their flag.

```

16703 \cs_new:Npn __fp_invalid_operation:nw #1#2#3; { }
16704 \cs_new:Npn __fp_invalid_operation_o:Nww #1#2; #3; { }
16705 \cs_new:Npn __fp_invalid_operation_tl_o:ff #1 #2 { }
16706 \cs_new:Npn __fp_division_by_zero_o:Nnw #1#2#3; { }
16707 \cs_new:Npn __fp_division_by_zero_o:NNww #1#2#3; #4; { }
16708 \cs_new:Npn __fp_overflow:w { }
16709 \cs_new:Npn __fp_underflow:w { }
16710 \fp_trap:nn { invalid_operation } { error }
16711 \fp_trap:nn { division_by_zero } { flag }
16712 \fp_trap:nn { overflow } { flag }
16713 \fp_trap:nn { underflow } { flag }

```

(End definition for `\__fp_invalid_operation:nw` and others.)

```

__fp_invalid_operation_o:nw
__fp_invalid_operation_o:fw

```

Convenient short-hands for returning `\c_nan_fp` for a unary or binary operation, and expanding after.

```

16714 \cs_new:Npn __fp_invalid_operation_o:nw
16715 { __fp_invalid_operation:nw { \exp_after:wN \c_nan_fp } }
16716 \cs_generate_variant:Nn __fp_invalid_operation_o:nw { f }

```

(End definition for `\__fp_invalid_operation_o:nw`.)

### 26.3 Errors

```
__fp_error:nnnn
__fp_error:nnfn
__fp_error:nfnf
__fp_error:nfff
16717 \cs_new:Npn __fp_error:nnnn
16718 { __kernel_msg_expandable_error:nnnnn { kernel } }
16719 \cs_generate_variant:Nn __fp_error:nnnn { nnf, nff , nfff }

(End definition for __fp_error:nnnn.)
```

### 26.4 Messages

Some messages.

```
16720 __kernel_msg_new:nnnn { kernel } { unknown-fpu-exception }
16721 {
16722 The~FPU~exception~'#1'~is~not~known:~
16723 that~trap~will~never~be~triggered.
16724 }
16725 {
16726 The~only~exceptions~to~which~traps~can~be~attached~are \\
16727 \iow_indent:n
16728 {
16729 * ~ invalid_operation \\
16730 * ~ division_by_zero \\
16731 * ~ overflow \\
16732 * ~ underflow
16733 }
16734 }
16735 __kernel_msg_new:nnnn { kernel } { unknown-fpu-trap-type }
16736 { The~FPU~trap-type~'#2'~is~not~known. }
16737 {
16738 The~trap~type~must~be~one~of \\
16739 \iow_indent:n
16740 {
16741 * ~ error \\
16742 * ~ flag \\
16743 * ~ none
16744 }
16745 }
16746 __kernel_msg_new:nnn { kernel } { fp-flow }
16747 { An ~ #3 ~ occurred. }
16748 __kernel_msg_new:nnn { kernel } { fp-flow-to }
16749 { #1 ~ #3 ed ~ to ~ #2 . }
16750 __kernel_msg_new:nnn { kernel } { fp-zero-div }
16751 { Division~by~zero~in~ #1 (#2) }
16752 __kernel_msg_new:nnn { kernel } { fp-zero-div-ii }
16753 { Division~by~zero~in~ (#1) #3 (#2) }
16754 __kernel_msg_new:nnn { kernel } { fp-invalid }
16755 { Invalid~operation~ #1 (#2) }
16756 __kernel_msg_new:nnn { kernel } { fp-invalid-ii }
16757 { Invalid~operation~ (#1) #3 (#2) }
16758 __kernel_msg_new:nnn { kernel } { fp-unknown-type }
16759 { Unknown~type~for~'#1' }

16760 </package>
```

## 27 I3fp-round implementation

```
16761 /*package)
16762 <@=fp>

__fp_parse_word_trunc:N
__fp_parse_word_floor:N
__fp_parse_word.ceil:N
16763 \cs_new:Npn __fp_parse_word_trunc:N
16764 { __fp_parse_function>NNN __fp_round_o:Nw __fp_round_to_zero>NNN }
16765 \cs_new:Npn __fp_parse_word_floor:N
16766 { __fp_parse_function>NNN __fp_round_o:Nw __fp_round_to_ninf>NNN }
16767 \cs_new:Npn __fp_parse_word.ceil:N
16768 { __fp_parse_function>NNN __fp_round_o:Nw __fp_round_to_pinf>NNN }

(End definition for __fp_parse_word_trunc:N, __fp_parse_word_floor:N, and __fp_parse_word.ceil:N.)
```

```
__fp_parse_word_round:N
__fp_parse_round:Nw
16769 \cs_new:Npn __fp_parse_word_round:N #1#2
16770 {
16771 __fp_parse_function>NNN
16772 __fp_round_o:Nw __fp_round_to_nearest>NNN #1
16773 #2
16774 }
16775 \cs_new:Npn __fp_parse_round:Nw #1 #2 __fp_round_to_nearest>NNN #3#4
16776 { #2 #1 #3 }
16777
```

(End definition for \\_\_fp\_parse\_word\_round:N and \\_\_fp\_parse\_round:Nw.)

### 27.1 Rounding tools

\c\_fp\_five\_int

This is used as the half-point for which numbers are rounded up/down.

```
16778 \int_const:Nn \c_fp_five_int { 5 }
```

(End definition for \c\_fp\_five\_int.)

Floating point operations often yield a result that cannot be exactly represented in a significand with 16 digits. In that case, we need to round the exact result to a representable number. The IEEE standard defines four rounding modes:

- Round to nearest: round to the representable floating point number whose absolute difference with the exact result is the smallest. If the exact result lies exactly at the mid-point between two consecutive representable floating point numbers, round to the floating point number whose last digit is even.
- Round towards negative infinity: round to the greatest floating point number not larger than the exact result.
- Round towards zero: round to a floating point number with the same sign as the exact result, with the largest absolute value not larger than the absolute value of the exact result.
- Round towards positive infinity: round to the least floating point number not smaller than the exact result.

This is not fully implemented in l3fp yet, and transcendental functions fall back on the “round to nearest” mode. All rounding for basic algebra is done through the functions defined in this module, which can be redefined to change their rounding behaviour (but there is not interface for that yet).

The rounding tools available in this module are many variations on a base function `\_fp_round:NNN`, which expands to `0\exp_stop_f:` or `1\exp_stop_f:` depending on whether the final result should be rounded up or down.

- `\_fp_round:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f::`.
- `\_fp_round_s:NNNw <sign> <digit1> <digit2> <more digits>;` can expand to `0\exp_stop_f::`; or `1\exp_stop_f::..`.
- `\_fp_round_neg:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f::..`

See implementation comments for details on the syntax.

```
_fp_round:NNN
\fp_round_to_nearest:NNN
\fp_round_to_nearest_ninf:NNN
\fp_round_to_nearest_zero:NNN
\fp_round_to_nearest_pinf:NNN
\fp_round_to_ninf:NNN
\fp_round_to_zero:NNN
\fp_round_to_pinf:NNN
```

If rounding the number `<final sign><digit1>.⟨digit2⟩` to an integer rounds it towards zero (truncates it), this function expands to `0\exp_stop_f::`, and otherwise to `1\exp_stop_f::`. Typically used within the scope of an `\_fp_int_eval:w`, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.

It is very important that `<final sign>` be the final sign of the result. Otherwise, the result would be incorrect in the case of rounding towards  $-\infty$  or towards  $+\infty$ . Also recall that `<final sign>` is 0 for positive, and 2 for negative.

By default, the functions below return `0\exp_stop_f::`, but this is superseded by `\_fp_round_return_one::`, which instead returns `1\exp_stop_f::`, expanding everything and removing `0\exp_stop_f::` in the process. In the case of rounding towards  $\pm\infty$  or towards 0, this is not really useful, but it prepares us for the “round to nearest, ties to even” mode.

The “round to nearest” mode is the default. If the `<digit2>` is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that `<digit1>` plus the result is even. In other words, round up if `<digit1>` is odd.

The “round to nearest” mode has three variants, which differ in how ties are rounded: down towards  $-\infty$ , truncated towards 0, or up towards  $+\infty$ .

```
16779 \cs_new:Npn \fp_round_return_one:
16800 { \exp_after:wN 1 \exp_after:wN \exp_stop_f: \exp:w }
16801 \cs_new:Npn \fp_round_to_ninf:NNN #1 #2 #3
16802 {
16803 \if_meaning:w 2 #1
16804 \if_int_compare:w #3 > 0 \exp_stop_f:
16805 \fp_round_return_one:
16806 \fi:
16807 \fi:
16808 0 \exp_stop_f:
16809 }
16810 \cs_new:Npn \fp_round_to_zero:NNN #1 #2 #3 { 0 \exp_stop_f: }
16811 \cs_new:Npn \fp_round_to_pinf:NNN #1 #2 #3
16812 {
16813 \if_meaning:w 0 #1
16814 \if_int_compare:w #3 > 0 \exp_stop_f:
```

```

16795 __fp_round_return_one:
16796 \fifi:
16797 \fifi:
16798 0 \exp_stop_f:
16799 }
16800 \cs_new:Npn __fp_round_to_nearest:NNN #1 #2 #3
16801 {
16802 \if_int_compare:w #3 > \c__fp_five_int
16803 __fp_round_return_one:
16804 \else:
16805 \if_meaning:w 5 #3
16806 \if_int_odd:w #2 \exp_stop_f:
16807 __fp_round_return_one:
16808 \fifi:
16809 \fifi:
16810 \fifi:
16811 0 \exp_stop_f:
16812 }
16813 \cs_new:Npn __fp_round_to_nearest_ninf:NNN #1 #2 #3
16814 {
16815 \if_int_compare:w #3 > \c__fp_five_int
16816 __fp_round_return_one:
16817 \else:
16818 \if_meaning:w 5 #3
16819 \if_meaning:w 2 #1
16820 __fp_round_return_one:
16821 \fifi:
16822 \fifi:
16823 \fifi:
16824 0 \exp_stop_f:
16825 }
16826 \cs_new:Npn __fp_round_to_nearest_zero:NNN #1 #2 #3
16827 {
16828 \if_int_compare:w #3 > \c__fp_five_int
16829 __fp_round_return_one:
16830 \fifi:
16831 0 \exp_stop_f:
16832 }
16833 \cs_new:Npn __fp_round_to_nearest_pinf:NNN #1 #2 #3
16834 {
16835 \if_int_compare:w #3 > \c__fp_five_int
16836 __fp_round_return_one:
16837 \else:
16838 \if_meaning:w 5 #3
16839 \if_meaning:w 0 #1
16840 __fp_round_return_one:
16841 \fifi:
16842 \fifi:
16843 \fifi:
16844 0 \exp_stop_f:
16845 }
16846 \cs_new_eq:NN __fp_round:NNN __fp_round_to_nearest:NNN

```

(End definition for `\_\_fp_round:NNN` and others.)

\\_\\_fp\\_round\\_s:NNNw      \\_\\_fp\\_round\\_s:NNNw *(final sign)* *(digit)* *(more digits)* ;  
   Similar to \\_\\_fp\\_round:NNN, but with an extra semicolon, this function expands  
 to 0\exp\_stop\_f:; if rounding *(final sign)**(digit)*.*(more digits)* to an integer truncates,  
 and to 1\exp\_stop\_f:; otherwise. The *(more digits)* part must be a digit, followed by  
 something that does not overflow a \int\_use:N \\_\\_fp\\_int\\_eval:w construction. The  
 only relevant information about this piece is whether it is zero or not.

```

16847 \cs_new:Npn __fp_round_s:NNNw #1 #2 #3 #4;
16848 {
16849 \exp_after:wN __fp_round:NNN
16850 \exp_after:wN #1
16851 \exp_after:wN #2
16852 \int_value:w __fp_int_eval:w
16853 \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
16854 \if_meaning:w 5 #3 1 \fi:
16855 \exp_stop_f:
16856 \if_int_compare:w __fp_int_eval:w #4 > 0 \exp_stop_f:
16857 1 +
16858 \fi:
16859 \fi:
16860 #3
16861 ;
16862 }
```

(End definition for \\_\\_fp\\_round\\_s:NNNw.)

\\_\\_fp\\_round\\_digit:Nw      \int\_value:w \\_\\_fp\\_round\\_digit:Nw *(digit)* *(intexpr)* ;  
   This function should always be called within an \int\_value:w or \\_\\_fp\\_int\\_eval:w  
 expansion; it may add an extra \\_\\_fp\\_int\\_eval:w, which means that the integer or  
 integer expression should not be ended with a synonym of \relax, but with a semi-colon  
 for instance.

```

16863 \cs_new:Npn __fp_round_digit:Nw #1 #2;
16864 {
16865 \if_int_odd:w \if_meaning:w 0 #1 1 \else:
16866 \if_meaning:w 5 #1 1 \else:
16867 0 \fi: \fi: \exp_stop_f:
16868 \if_int_compare:w __fp_int_eval:w #2 > 0 \exp_stop_f:
16869 __fp_int_eval:w 1 +
16870 \fi:
16871 \fi:
16872 #1
16873 }
```

(End definition for \\_\\_fp\\_round\\_digit:Nw.)

\\_\\_fp\\_round\\_neg:NNN      \\_\\_fp\\_round\\_neg:NNN *(final sign)* *(digit<sub>1</sub>)* *(digit<sub>2</sub>)*  
   \\_\\_fp\\_round\\_to\\_nearest\\_neg:NNN  
   \\_\\_fp\\_round\\_to\\_nearest\\_ninf\\_neg:NNN  
   \\_\\_fp\\_round\\_to\\_nearest\\_zero\\_neg:NNN  
   \\_\\_fp\\_round\\_to\\_nearest\\_pinf\\_neg:NNN  
   \\_\\_fp\\_round\\_to\\_ninf\\_neg:NNN  
   \\_\\_fp\\_round\\_to\\_zero\\_neg:NNN  
   \\_\\_fp\\_round\\_to\\_pinf\\_neg:NNN  
   This expands to 0\exp\_stop\_f: or 1\exp\_stop\_f: after doing the following test.  
 Starting from a number of the form *(final sign)*0.*(15 digits)**(digit<sub>1</sub>)* with exactly 15 (non-all-zero)  
 digits before *(digit<sub>1</sub>)*, subtract from it *(final sign)*0.0...0*(digit<sub>2</sub>)*, where there  
 are 16 zeros. If in the current rounding mode the result should be rounded down, then  
 this function returns 1\exp\_stop\_f:. Otherwise, i.e., if the result is rounded back to  
 the first operand, then this function returns 0\exp\_stop\_f:.

It turns out that this negative “round to nearest” is identical to the positive one.  
 And this is the default mode.

```

16874 \cs_new_eq:NN __fp_round_to_ninf_neg:NNN __fp_round_to_pinf:NNN
16875 \cs_new:Npn __fp_round_to_zero_neg:NNN #1 #2 #3
16876 {
16877 \if_int_compare:w #3 > 0 \exp_stop_f:
16878 __fp_round_return_one:
16879 \fi:
16880 0 \exp_stop_f:
16881 }
16882 \cs_new_eq:NN __fp_round_to_pinf_neg:NNN __fp_round_to_ninf:NNN
16883 \cs_new_eq:NN __fp_round_to_nearest_neg:NNN __fp_round_to_nearest:NNN
16884 \cs_new_eq:NN __fp_round_to_nearest_ninf_neg:NNN
16885 __fp_round_to_nearest_pinf:NNN
16886 \cs_new:Npn __fp_round_to_nearest_zero_neg:NNN #1 #2 #3
16887 {
16888 \if_int_compare:w #3 < \c_fp_five_int \else:
16889 __fp_round_return_one:
16890 \fi:
16891 0 \exp_stop_f:
16892 }
16893 \cs_new_eq:NN __fp_round_to_nearest_pinf_neg:NNN
16894 __fp_round_to_nearest_ninf:NNN
16895 \cs_new_eq:NN __fp_round_neg:NNN __fp_round_to_nearest_neg:NNN

```

(End definition for `\__fp_round_neg:NNN` and others.)

## 27.2 The round function

```

__fp_round_o:Nw
__fp_round_aux_o:Nw

```

First check that all arguments are floating point numbers. The `trunc`, `ceil` and `floor` functions expect one or two arguments (the second is 0 by default), and the `round` function also accepts a third argument (`nan` by default), which changes #1 from `\__fp_round_to_nearest:NNN` to one of its analogues.

```

16896 \cs_new:Npn __fp_round_o:Nw #1
16897 {
16898 __fp_parse_function_all_fp_o:fnw
16899 { __fp_round_name_from_cs:N #1 }
16900 { __fp_round_aux_o:Nw #1 }
16901 }
16902 \cs_new:Npn __fp_round_aux_o:Nw #1#2 @
16903 {
16904 \if_case:w
16905 __fp_int_eval:w __fp_array_count:n {#2} __fp_int_eval_end:
16906 __fp_round_no_arg_o:Nw #1 \exp:w
16907 \or: __fp_round:Nwn #1 #2 {0} \exp:w
16908 \or: __fp_round:Nww #1 #2 \exp:w
16909 \else: __fp_round:Nwww #1 #2 @ \exp:w
16910 \fi:
16911 \exp_after:wN \exp_end:
16912 }

```

(End definition for `\__fp_round_o:Nw` and `\__fp_round_aux_o:Nw`.)

```

__fp_round_no_arg_o:Nw

```

```

16913 \cs_new:Npn __fp_round_no_arg_o:Nw #1
16914 {

```

```

16915 \cs_if_eq:NNTF #1 __fp_round_to_nearest:NNN
16916 { __fp_error:nnnn { fp-num-args } { round () } { 1 } { 3 } }
16917 {
16918 __fp_error:nffn { fp-num-args }
16919 { __fp_round_name_from_cs:N #1 () } { 1 } { 2 }
16920 }
16921 \exp_after:wN \c_nan_fp
16922 }

```

(End definition for `\__fp_round_no_arg_o:Nw`.)

`\__fp_round:Nwww` Having three arguments is only allowed for `round`, not `trunc`, `ceil`, `floor`, so check for that case. If all is well, construct one of `\__fp_round_to_nearest:NNN`, `\__fp_round_to_nearest_zero:NNN`, `\__fp_round_to_nearest_ninf:NNN`, `\__fp_round_to_nearest_pinf:NNN` and act accordingly.

```

16923 \cs_new:Npn __fp_round:Nwww #1#2 ; #3 ; \s__fp __fp_chk:w #4#5#6 ; #7 @
16924 {
16925 \cs_if_eq:NNTF #1 __fp_round_to_nearest:NNN
16926 {
16927 \tl_if_empty:nTF {#7}
16928 {
16929 \exp_args:Nc __fp_round:Nww
16930 {
16931 __fp_round_to_nearest
16932 \if_meaning:w 0 #4 _zero \else:
16933 \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
16934 :NNN
16935 }
16936 #2 ; #3 ;
16937 }
16938 {
16939 __fp_error:nnnn { fp-num-args } { round () } { 1 } { 3 }
16940 \exp_after:wN \c_nan_fp
16941 }
16942 }
16943 {
16944 __fp_error:nffn { fp-num-args }
16945 { __fp_round_name_from_cs:N #1 () } { 1 } { 2 }
16946 \exp_after:wN \c_nan_fp
16947 }
16948 }

```

(End definition for `\__fp_round:Nwww`.)

`\__fp_round_name_from_cs:N`

```

16949 \cs_new:Npn __fp_round_name_from_cs:N #1
16950 {
16951 \cs_if_eq:NNTF #1 __fp_round_to_zero:NNN { trunc }
16952 {
16953 \cs_if_eq:NNTF #1 __fp_round_to_ninf:NNN { floor }
16954 {
16955 \cs_if_eq:NNTF #1 __fp_round_to_pinf:NNN { ceil }
16956 { round }
16957 }
16958 }

```

```

16958 }
16959 }
```

(End definition for `\_fp_round_name_from_cs:N.`)

```
_fp_round:Nww
\fp_round:Nwn
```

```
\fp_round_normal:NwNNnw
```

```
\fp_round_normal:NnnwNNnn
```

```
\fp_round_pack:Nw
```

```
\fp_round_normal:NNwNnn
```

```
\fp_round_normal_end:wwNnn
```

```
\fp_round_special:NwwNnn
```

```
\fp_round_special_aux:Nw
```

If the number of digits to round to is an integer or infinity all is good; if it is `nan` then just produce a `nan`; otherwise invalid as we have something like `round(1,3.14)` where the number of digits is not an integer.

```

16960 \cs_new:Npn \fp_round:Nww #1#2 ; #3 ;
16961 {
16962 \fp_small_int:wTF #3; { \fp_round:Nwn #1#2; }
16963 {
16964 \if:w 3 \fp_kind:w #3 ;
16965 \exp_after:wN \use_i:nn
16966 \else:
16967 \exp_after:wN \use_ii:nn
16968 \fi:
16969 { \exp_after:wN \c_nan_fp }
16970 {
16971 \fp_invalid_operation_tl_o:ff
16972 { \fp_round_name_from_cs:N #1 }
16973 { \fp_array_to_clist:n { #2; #3; } }
16974 }
16975 }
16976 }
16977 \cs_new:Npn \fp_round:Nwn #1 \s_fp \fp_chk:w #2#3#4; #5
16978 {
16979 \if_meaning:w 1 #2
16980 \exp_after:wN \fp_round_normal:NwNNnw
16981 \exp_after:wN #1
16982 \int_value:w #5
16983 \else:
16984 \exp_after:wN \fp_exp_after_o:w
16985 \fi:
16986 \s_fp \fp_chk:w #2#3#4;
16987 }
16988 \cs_new:Npn \fp_round_normal:NwNNnw #1#2 \s_fp \fp_chk:w 1#3#4#5;
16989 {
16990 \fp_decimate:nNnnnn { \c_fp_prec_int - #4 - #2 }
16991 \fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
16992 }
16993 \cs_new:Npn \fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
16994 {
16995 \exp_after:wN \fp_round_normal>NNwNnn
16996 \int_value:w \fp_int_eval:w
16997 \if_int_compare:w #2 > 0 \exp_stop_f:
16998 1 \int_value:w #2
16999 \exp_after:wN \fp_round_pack:Nw
17000 \int_value:w \fp_int_eval:w 1#3 +
17001 \else:
17002 \if_int_compare:w #3 > 0 \exp_stop_f:
17003 1 \int_value:w #3 +
17004 \fi:
17005 \fi:
```

```

17006 \exp_after:wN #5
17007 \exp_after:wN #6
17008 \use_none:nnnnnnn #3
17009 #1
17010 _fp_int_eval_end:
17011 0000 0000 0000 0000 ; #6
17012 }
17013 \cs_new:Npn _fp_round_pack:Nw #1
17014 { \if_meaning:w 2 #1 + 1 \fi: _fp_int_eval_end: }
17015 \cs_new:Npn _fp_round_normal:NNwNnn #1 #2
17016 {
17017 \if_meaning:w 0 #2
17018 \exp_after:wN _fp_round_special:NwwNnn
17019 \exp_after:wN #1
17020 \fi:
17021 _fp_pack_twice_four:wNNNNNNNN
17022 _fp_pack_twice_four:wNNNNNNNN
17023 _fp_round_normal_end:wwNnn
17024 ; #2
17025 }
17026 \cs_new:Npn _fp_round_normal_end:wwNnn #1;#2;#3#4#5
17027 {
17028 \exp_after:wN _fp_exp_after_o:w \exp:w \exp_end_continue_f:w
17029 _fp_SANITIZE:Nw #3 #4 ; #1 ;
17030 }
17031 \cs_new:Npn _fp_round_special:NwwNnn #1#2;#3;#4#5#6
17032 {
17033 \if_meaning:w 0 #1
17034 _fp_case_return:nw
17035 { \exp_after:wN _fp_zero_fp:N \exp_after:wN #4 }
17036 \else:
17037 \exp_after:wN _fp_round_special_aux:Nw
17038 \exp_after:wN #4
17039 \int_value:w _fp_int_eval:w 1
17040 \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
17041 \fi:
17042 ;
17043 }
17044 \cs_new:Npn _fp_round_special_aux:Nw #1#2;
17045 {
17046 \exp_after:wN _fp_exp_after_o:w \exp:w \exp_end_continue_f:w
17047 _fp_SANITIZE:Nw #1#2; {1000}{0000}{0000}{0000};
17048 }

```

(End definition for `\_fp_round:Nww` and others.)

17049 ⟨/package⟩

## 28 l3fp-parse implementation

```

17050 (*package)
17051 <@@=fp>

```

## 28.1 Work plan

The task at hand is non-trivial, and some previous failed attempts show that the code leads to unreadable logs, so we had better get it (almost) right the first time. Let us first describe our goal, then discuss the design precisely before writing any code.

In this file at least, a  $\langle\text{floating point object}\rangle$  is a floating point number or tuple. This can be extended to anything that starts with  $\backslash\text{s\_fp}$  or  $\backslash\text{s\_fp}_{\langle\text{type}\rangle}$  and ends with ; with some internal structure that depends on the  $\langle\text{type}\rangle$ .

$\backslash\text{--fp\_parse:n}$        $\backslash\text{--fp\_parse:n} \{\langle\text{fpexpr}\rangle\}$

Evaluates the  $\langle\text{floating point expression}\rangle$  and leaves the result in the input stream as a floating point object. This function forms the basis of almost all public l3fp functions. During evaluation, each token is fully f-expanded.

$\backslash\text{--fp\_parse:o:n}$  does the same but expands once after its result.

**TEXhackers note:** Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as  $\backslash\text{int\_use:N}$ . Invalid tokens remaining after f-expansion lead to unrecoverable low-level TeX errors.

(End definition for  $\backslash\text{--fp\_parse:n}$ .)

$\backslash\text{c\_fp\_prec\_func\_int}$   
 $\backslash\text{c\_fp\_prec\_hatii\_int}$   
 $\backslash\text{c\_fp\_prec\_hat\_int}$   
 $\backslash\text{c\_fp\_prec\_not\_int}$   
 $\backslash\text{c\_fp\_prec\_juxt\_int}$   
 $\backslash\text{c\_fp\_prec\_times\_int}$   
 $\backslash\text{c\_fp\_prec\_plus\_int}$   
 $\backslash\text{c\_fp\_prec\_comp\_int}$   
 $\backslash\text{c\_fp\_prec\_and\_int}$   
 $\backslash\text{c\_fp\_prec\_or\_int}$   
 $\backslash\text{c\_fp\_prec\_quest\_int}$   
 $\backslash\text{c\_fp\_prec\_colon\_int}$   
 $\backslash\text{c\_fp\_prec\_comma\_int}$   
 $\backslash\text{c\_fp\_prec\_tuple\_int}$   
 $\backslash\text{c\_fp\_prec\_end\_int}$

Floating point expressions are composed of numbers, given in various forms, infix operators, such as +, \*\*, or , (which joins two numbers into a list), and prefix operators, such as the unary -, functions, or opening parentheses. Here is a list of precedences which control the order of evaluation (some distinctions are irrelevant for the order of evaluation, but serve as signals), from the tightest binding to the loosest binding.

16 Function calls.

13/14 Binary \*\* and ^ (right to left).

12 Unary +, -, ! (right to left).

11 Juxtaposition (implicit \*) with no parenthesis.

10 Binary \* and /.

9 Binary + and -.

7 Comparisons.

6 Logical and, denoted by &&.

5 Logical or, denoted by ||.

4 Ternary operator ?:, piece ?.

3 Ternary operator ?:, piece :.

2 Commas.

1 Place where a comma is allowed and generates a tuple.

0 Start and end of the expression.

```

17052 \int_const:Nn \c__fp_prec_func_int { 16 }
17053 \int_const:Nn \c__fp_prec_hatii_int { 14 }
17054 \int_const:Nn \c__fp_prec_hat_int { 13 }
17055 \int_const:Nn \c__fp_prec_not_int { 12 }
17056 \int_const:Nn \c__fp_prec_juxtap_int { 11 }
17057 \int_const:Nn \c__fp_prec_times_int { 10 }
17058 \int_const:Nn \c__fp_prec_plus_int { 9 }
17059 \int_const:Nn \c__fp_prec_comp_int { 7 }
17060 \int_const:Nn \c__fp_prec_and_int { 6 }
17061 \int_const:Nn \c__fp_prec_or_int { 5 }
17062 \int_const:Nn \c__fp_prec_quest_int { 4 }
17063 \int_const:Nn \c__fp_prec_colon_int { 3 }
17064 \int_const:Nn \c__fp_prec_comma_int { 2 }
17065 \int_const:Nn \c__fp_prec_tuple_int { 1 }
17066 \int_const:Nn \c__fp_prec_end_int { 0 }

```

(End definition for `\c__fp_prec_func_int` and others.)

### 28.1.1 Storing results

The main question in parsing expressions expandably is to decide where to put the intermediate results computed for various subexpressions.

One option is to store the values at the start of the expression, and carry them together as the first argument of each macro. However, we want to f-expand tokens one by one in the expression (as `\int_eval:n` does), and with this approach, expanding the next unread token forces us to jump with `\exp_after:wN` over every value computed earlier in the expression. With this approach, the run-time grows at least quadratically in the length of the expression, if not as its cube (inserting the `\exp_after:wN` is tricky and slow).

A second option is to place those values at the end of the expression. Then expanding the next unread token is straightforward, but this still hits a performance issue: for long expressions we would be reaching all the way to the end of the expression at every step of the calculation. The run-time is again quadratic.

A variation of the above attempts to place the intermediate results which appear when computing a parenthesized expression near the closing parenthesis. This still lets us expand tokens as we go, and avoids performance problems as long as there are enough parentheses. However, it would be better to avoid requiring the closing parenthesis to be present as soon as the corresponding opening parenthesis is read: the closing parenthesis may still be hidden in a macro yet to be expanded.

Hence, we need to go for some fine expansion control: the result is stored *before* the start!

Let us illustrate this idea in a simple model: adding positive integers which may be resulting from the expansion of macros, or may be values of registers. Assume that one number, say, 12345, has already been found, and that we want to parse the next number. The current status of the code may look as follows.

```

\exp_after:wN \add:ww \int_value:w 12345 \exp_after:wN ;
\exp:w \operand:w <stuff>

```

One step of expansion expands `\exp_after:wN`, which triggers the primitive `\int_value:w`, which reads the five digits we have already found, 12345. This integer is unfinished, causing the second `\exp_after:wN` to expand, and to trigger the construction

`\exp:w`, which expands `\operand:w`, defined to read what follows and make a number out of it, then leave `\exp_end:`, the number, and a semicolon in the input stream. Once `\operand:w` is done expanding, we obtain essentially

```
\exp_after:wN \add:ww \int_value:w 12345 ;
\exp:w \exp_end: 333444 ;
```

where in fact `\exp_after:wN` has already been expanded, `\int_value:w` has already seen 12345, and `\exp:w` is still looking for a number. It finds `\exp_end:`, hence expands to nothing. Now, `\int_value:w` sees the `;`, which cannot be part of a number. The expansion stops, and we are left with

```
\add:ww 12345 ; 333444 ;
```

which can safely perform the addition by grabbing two arguments delimited by `;`.

If we were to continue parsing the expression, then the following number should also be cleaned up before the next use of a binary operation such as `\add:ww`. Just like `\int_value:w 12345 \exp_after:wN ;` expanded what follows once, we need `\add:ww` to do the calculation, and in the process to expand the following once. This is also true in our real application: all the functions of the form `\_fp_\dots_o:ww` expand what follows once. This comes at the cost of leaving tokens in the input stack, and we need to be careful not to waste this memory. All of our discussion above is nice but simplistic, as operations should not simply be performed in the order they appear.

### 28.1.2 Precedence and infix operators

The various operators we will encounter have different precedences, which influence the order of calculations:  $1 + 2 \times 3 = 1 + (2 \times 3)$  because  $\times$  has a higher precedence than  $+$ . The true analog of our macro `\operand:w` must thus take care of that. When looking for an operand, it needs to perform calculations until reaching an operator which has lower precedence than the one which called `\operand:w`. This means that `\operand:w` must know what the previous binary operator is, or rather, its precedence: we thus rename it `\operand:Nw`. Let us describe as an example how we plan to do the calculation  $41 - 2^3 * 4 + 5$ . More precisely we describe how to perform the first operation in this expression. Here, we abuse notations: the first argument of `\operand:Nw` should be an integer constant (`\c_fp_prec_plus_int, ...`) equal to the precedence of the given operator, not directly the operator itself.

- Clean up 41 and find `-`. We call `\operand:Nw -` to find the second operand.
- Clean up 2 and find `^`.
- Compare the precedences of `-` and `^`. Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw ^`.
- Clean up 3 and find `*`.
- Compare the precedences of `^` and `*`. Since the former is higher, `\operand:Nw ^` has found the second operand of the exponentiation, which is computed:  $2^3 = 8$ .
- We now have  $41 - 8 * 4 + 5$ , and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 8?

- Compare the precedences of `-` and `*`. Since the latter is higher, we are not done with 8. Call `\operand:Nw *` to find the second operand of the multiplication.
- Clean up 4, and find `+`.
- Compare the precedences of `*` and `+`. Since the former is higher, `\operand:Nw *` has found the second operand of the multiplication, which is computed:  $8 * 4 = 32$ .
- We now have  $41 - 32 + 5$ , and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 32?
- Compare the precedences of `-` and `+`. Since they are equal, `\operand:Nw -` has found the second operand for the subtraction, which is computed:  $41 - 32 = 9$ .
- We now have  $9 + 5$ .

The procedure above stops short of performing all computations, but adding a surrounding call to `\operand:Nw` with a very low precedence ensures that all computations are performed before `\operand:Nw` is done. Adding a trailing marker with the same very low precedence prevents the surrounding `\operand:Nw` from going beyond the marker.

The pattern above to find an operand for a given operator, is to find one number and the next operator, then compare precedences to know if the next computation should be done. If it should, then perform it after finding its second operand, and look at the next operator, then compare precedences to know if the next computation should be done. This continues until we find that the next computation should not be done. Then, we stop.

We are now ready to get a bit more technical and describe which of the `l3fp-parse` functions correspond to each step above.

First, `\_fp_parse_operand:Nw` is the `\operand:Nw` function above, with small modifications due to expansion issues discussed later. We denote by `precedence` the argument of `\_fp_parse_operand:Nw`, that is, the precedence of the binary operator whose operand we are trying to find. The basic action is to read numbers from the input stream. This is done by `\_fp_parse_one:Nw`. A first approximation of this function is that it reads one `number`, performing no computation, and finds the following binary `operator`. Then it expands to

```
number
\fp_parse_infix_<operator>:N <precedence>
```

expanding the `infix` auxiliary before leaving the above in the input stream.

We now explain the `infix` auxiliaries. We need some flexibility in how we treat the case of equal precedences: most often, the first operation encountered should be performed, such as  $1-2-3$  being computed as  $(1-2)-3$ , but  $2^3^4$  should be evaluated as  $2^{(3^4)}$  instead. For this reason, and to support the equivalence between `**` and `^` more easily, each binary operator is converted to a control sequence `\_fp_parse_infix_<operator>:N` when it is encountered for the first time. Instead of passing both precedences to a test function to do the comparison steps above, we pass the `precedence` (of the earlier operator) to the `infix` auxiliary for the following `<operator>`, to know whether to perform the computation of the `<operator>`. If it should not be performed, the `infix` auxiliary expands to

```
@ \use_none:n \fp_parse_infix_<operator>:N
```

and otherwise it calls `\_fp_parse_operand:Nw` with the precedence of the `<operator>` to find its second operand `<number2>` and the next `<operator2>`, and expands to

```
@ _fp_parse_apply_binary:NwNwN
 <operator> <number2>
@ _fp_parse_infix_<operator2>:N
```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand `<number>` is before the `infix` function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of `\_fp_parse_operand:Nw <precedence>` with some of the expansion control removed is

```
\exp_after:wN _fp_parse_continue:NwN
\exp_after:wN <precedence>
\exp:w \exp_end_continue_f:w
 _fp_parse_one:Nw <precedence>
```

This expands `\_fp_parse_one:Nw <precedence>` completely, which finds a number, wraps the next `<operator>` into an `infix` function, feeds this function the `<precedence>`, and expands it, yielding either

```
_fp_parse_continue:NwN <precedence>
<number> @
\use_none:n _fp_parse_infix_<operator>:N
```

or

```
_fp_parse_continue:NwN <precedence>
<number> @
_fp_parse_apply_binary:NwNwN
 <operator> <number2>
@ _fp_parse_infix_<operator2>:N
```

The definition of `\_fp_parse_continue:NwN` is then very simple:

```
\cs_new:Npn _fp_parse_continue:NwN #1#2#3 { #3 #1 #2 @ }
```

In the first case, #3 is `\use_none:n`, yielding

```
\use_none:n <precedence> <number> @
_fp_parse_infix_<operator>:N
```

then `<number> @ \_fp_parse_infix_<operator>:N`. In the second case, #3 is `\_fp_parse_apply_binary:NwNwN`, whose role is to compute `<number> <operator> <number2>` and to prepare for the next comparison of precedences: first we get

```
_fp_parse_apply_binary:NwNwN
 <precedence> <number> @
 <operator> <number2>
@ _fp_parse_infix_<operator2>:N
```

then

```

\exp_after:wN _fp_parse_continue:NwN
\exp_after:wN ⟨precedence⟩
\exp:w \exp_end_continue_f:w
fp⟨operator⟩_o:ww ⟨number⟩ ⟨number₂⟩
\exp:w \exp_end_continue_f:w
_fp_parse_infix_⟨operator₂⟩:N ⟨precedence⟩

```

where  $\_fp_⟨operator⟩_o:ww$  computes  $⟨number⟩ ⟨operator⟩ ⟨number₂⟩$  and expands after the result, thus triggers the comparison of the precedence of the  $⟨operator₂⟩$  and the  $⟨precedence⟩$ , continuing the loop.

We have introduced the most important functions here, and the next few paragraphs we describe various subtleties.

### 28.1.3 Prefix operators, parentheses, and functions

Prefix operators (unary  $-$ ,  $+$ ,  $!$ ) and parentheses are taken care of by the same mechanism, and functions ( $\sin$ ,  $\exp$ , etc.) as well. Finding the argument of the unary  $-$ , for instance, is very similar to grabbing the second operand of a binary infix operator, with a subtle precedence explained below. Once that operand is found, the operator can be applied to it (for the unary  $-$ , this simply flips the sign). A left parenthesis is just a prefix operator with a very low precedence equal to that of the closing parenthesis (which is treated as an infix operator, since it normally appears just after numbers), so that all computations are performed until the closing parenthesis. The prefix operator associated to the left parenthesis does not alter its argument, but it removes the closing parenthesis (with some checks).

Prefix operators are the reason why we only summarily described the function  $\_fp_parse_one:Nw$  earlier. This function is responsible for reading in the input stream the first possible  $⟨number⟩$  and the next infix  $⟨operator⟩$ . If what follows  $\_fp_parse_one:Nw ⟨precedence⟩$  is a prefix operator, then we must find the operand of this prefix operator through a nested call to  $\_fp_parse_operand:Nw$  with the appropriate precedence, then apply the operator to the operand found to yield the result of  $\_fp_parse_one:Nw$ . So far, all is simple.

The unary operators  $+$ ,  $-$ ,  $!$  complicate things a little bit:  $-3^{**}2$  should be  $-(3^2) = -9$ , and not  $(-3)^2 = 9$ . This would easily be done by giving  $-$  a lower precedence, equal to that of the infix  $+$  and  $-$ . Unfortunately, this fails in cases such as  $3^{**}-2^{*}4$ , yielding  $3^{-2\times 4}$  instead of the correct  $3^{-2} \times 4$ . A second attempt would be to call  $\_fp_parse_operand:Nw$  with the  $⟨precedence⟩$  of the previous operator, but  $0>-2+3$  is then parsed as  $0>-(2+3)$ : the addition is performed because it binds more tightly than the comparison which precedes  $-$ . The correct approach is for a unary  $-$  to perform operations whose precedence is greater than both that of the previous operation, and that of the unary  $-$  itself. The unary  $-$  is given a precedence higher than multiplication and division. This does not lead to any surprising result, since  $-(x/y) = (-x)/y$  and similarly for multiplication, and it reduces the number of nested calls to  $\_fp_parse_operand:Nw$ .

Functions are implemented as prefix operators with very high precedence, so that their argument is the first number that can possibly be built.

Note that contrarily to the `infix` functions discussed earlier, the `prefix` functions do perform tests on the previous  $⟨precedence⟩$  to decide whether to find an argument or not, since we know that we need a number, and must never stop there.

### 28.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a “number”? A number is typically given in the form  $\langle\text{significand}\rangle e \langle\text{exponent}\rangle$ , where the  $\langle\text{significand}\rangle$  is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent “ $e \langle\text{exponent}\rangle$ ” is optional and is composed of an exponent mark `e` followed by a possibly empty string of signs `+` or `-` and a non-empty string of decimal digits. The  $\langle\text{significand}\rangle$  can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the  $\langle\text{exponent}\rangle$  can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as `nan`, `inf` or `pi`. We may add more types in the future.

When `\_fp_parse_one:Nw` is looking for a “number”, here is what happens.

- If the next token is a control sequence with the meaning of `\scan_stop:`, it can be: `\s_fp`, in which case our job is done, as what follows is an internal floating point number, or `\s_fp_expr_mark`, in which case the expression has come to an early end, as we are still looking for a number here, or something else, in which case we consider the control sequence to be a bad variable resulting from c-expansion.
- If the next token is a control sequence with a different meaning, we assume that it is a register, unpack it with `\tex_the:D`, and use its value (in `pt` for dimensions and skips, `mu` for muskips) as the  $\langle\text{significand}\rangle$  of a number: we look for an exponent.
- If the next token is a digit, we remove any leading zeros, then read a significand larger than 1 if the next character is a digit, read a significand smaller than 1 if the next character is a period, or we have found a significand equal to 0 otherwise, and look for an exponent.
- If the next token is a letter, we collect more letters until the first non-letter: the resulting word may denote a function such as `asin`, a constant such as `pi` or be unknown. In the first case, we call `\_fp_parse_operand:Nw` to find the argument of the function, then apply the function, before declaring that we are done. Otherwise, we are done, either with the value of the constant, or with the value `nan` for unknown words.
- If the next token is anything else, we check whether it is a known prefix operator, in which case `\_fp_parse_operand:Nw` finds its operand. If it is not known, then either a number is missing (if the token is a known infix operator) or the token is simply invalid in floating point expressions.

Once a number is found, `\_fp_parse_one:Nw` also finds an infix operator. This goes as follows.

- If the next token is a control sequence, it could be the special marker `\s_fp_expr_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_zero_int`, with an implied multiplication.
- If the next token is a letter, it is also a case of juxtaposition, as letters cannot be proper infix operators.
- Otherwise (including in the case of digits), if the token is a known infix operator, the appropriate `\_fp_infix_{operator}:N` function is built, and if it does not exist, we complain. In particular, the juxtaposition `\c_zero_int 2` is disallowed.

In the above, we need to test whether a character token #1 is a digit:

```
\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
 is a digit
\else:
 not a digit
\fi:
```

To exclude 0, replace 9 by 10. The use of `\token_to_str:N` ensures that a digit with any catcode is detected. To test if a character token is a letter, we need to work with its character code, testing if ‘#1 lies in [65, 90] (uppercase letters) or [97, 112] (lowercase letters)

```
\if_int_compare:w _fp_int_eval:w
 ('#1 \if_int_compare:w '#1 > 'Z - 32 \fi:) / 26 = 3 \exp_stop_f:
 is a letter
\else:
 not a letter
\fi:
```

At all steps, we try to accept all category codes: when #1 is kept to be used later, it is almost always converted to category code other through `\token_to_str:N`. More precisely, catcodes {3, 6, 7, 8, 11, 12} should work without trouble, but not {1, 2, 4, 10, 13}, and of course {0, 5, 9} cannot become tokens.

Floating point expressions should behave as much as possible like  $\varepsilon$ - $\text{\TeX}$ -based integer expressions and dimension expressions. In particular, f-expansion should be performed as the expression is read, token by token, forcing the expansion of protected macros, and ignoring spaces. One advantage of expanding at every step is that restricted expandable functions can then be used in floating point expressions just as they can be in other kinds of expressions. Problematically, spaces stop f-expansion: for instance, the macro `\X` below would not be expanded if we simply performed f-expansion.

```
\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }
```

Of course, spaces typically do not appear in a code setting, but may very easily come in document-level input, from which some expressions may come. To avoid this problem, at every step, we do essentially what `\use:f` would do: take an argument, put it back in the input stream, then f-expand it. This is not a complete solution, since a macro’s expansion could contain leading spaces which would stop the f-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers are correctly expanded to the underlying `\s_fp ...` structure. The f-expansion is performed by `\_fp_parse_expand:w`.

## 28.2 Main auxiliary functions

```
_fp_parse_operand:Nw \exp:w _fp_parse_operand:Nw <precedence> _fp_parse_expand:w
Reads the "...", performing every computation with a precedence higher than
<precedence>, then expands to
```

```
<result> @ _fp_parse_infix_<operation>:N ...
```

where the  $\langle operation \rangle$  is the first operation with a lower precedence, possibly `end`, and the “...” start just after the  $\langle operation \rangle$ .

(End definition for `\_\_fp\_parse\_operand:Nw.`)

`\_\_fp\_parse\_infix_+:N`

`\_\_fp\_parse\_infix_+:N \langle precedence \rangle ...`

If `+` has a precedence higher than the  $\langle precedence \rangle$ , cleans up a second  $\langle operand \rangle$  and finds the  $\langle operation_2 \rangle$  which follows, and expands to

`@ \_\_fp\_parse\_apply\_binary:NwNwN + \langle operand \rangle @ \_\_fp\_parse\_infix\_<operation_2>:N  
...`

Otherwise expands to

`@ \use\_none:n \_\_fp\_parse\_infix_+:N ...`

A similar function exists for each infix operator.

(End definition for `\_\_fp\_parse\_infix_+:N.`)

`\_\_fp\_parse\_one:Nw`

`\_\_fp\_parse\_one:Nw \langle precedence \rangle ...`

Cleans up one or two operands depending on how the precedence of the next operation compares to the  $\langle precedence \rangle$ . If the following  $\langle operation \rangle$  has a precedence higher than  $\langle precedence \rangle$ , expands to

`\langle operand_1 \rangle @ \_\_fp\_parse\_apply\_binary:NwNwN \langle operation \rangle \langle operand_2 \rangle @\n\_\_fp\_parse\_infix\_<operation_2>:N ...`

and otherwise expands to

`\langle operand \rangle @ \use\_none:n \_\_fp\_parse\_infix\_<operation>:N ...`

(End definition for `\_\_fp\_parse\_one:Nw.`)

### 28.3 Helpers

`\_\_fp\_parse\_expand:w`

`\exp:w \_\_fp\_parse\_expand:w \langle tokens \rangle`

This function must always come within a `\exp:w` expansion. The  $\langle tokens \rangle$  should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

`17067 \cs_new:Npn \_\_fp\_parse\_expand:w #1 { \exp_end_continue_f:w #1 }`

(End definition for `\_\_fp\_parse\_expand:w.`)

`\_\_fp\_parse\_return\_semicolon:w`

This very odd function swaps its position with the following `\fi:` and removes `\_\_fp\_parse\_expand:w` normally responsible for expansion. That turns out to be useful.

`17068 \cs_new:Npn \_\_fp\_parse\_return\_semicolon:w  
17069 #1 \fi: \_\_fp\_parse\_expand:w { \fi: ; #1 }`

(End definition for `\_\_fp\_parse\_return\_semicolon:w.`)

`\_\_fp\_parse\_digits_vii:N`

These functions must be called within an `\int_value:w` or `\_\_fp\_int\_eval:w` construction. The first token which follows must be f-expanded prior to calling those functions. The functions read tokens one by one, and output digits into the input stream, until meeting a non-digit, or up to a number of digits equal to their index. The full expansion is

`\_\_fp\_parse\_digits_vi:N`

`\_\_fp\_parse\_digits_v:N`

`\_\_fp\_parse\_digits_iv:N`

`\_\_fp\_parse\_digits_iii:N`

`\_\_fp\_parse\_digits_ii:N`

`\_\_fp\_parse\_digits_i:N`

`\_\_fp\_parse\_digits_:_N`

$\langle digits \rangle ; \langle filling \ 0 \rangle ; \langle length \rangle$

where  $\langle filling \ 0 \rangle$  is a string of zeros such that  $\langle digits \rangle \langle filling \ 0 \rangle$  has the length given by the index of the function, and  $\langle length \rangle$  is the number of zeros in the  $\langle filling \ 0 \rangle$  string. Each function puts a digit into the input stream and calls the next function, until we find a non-digit. We are careful to pass the tested tokens through `\token_to_str:N` to normalize their category code.

```

17070 \cs_set_protected:Npn __fp_tmp:w #1 #2 #
17071 {
17072 \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
17073 {
17074 \if_int_compare:w 9 < 1 \token_to_str:N ##1 \exp_stop_f:
17075 \token_to_str:N ##1 \exp_after:wN #2 \exp:w
17076 \else:
17077 __fp_parse_return_semicolon:w #3 ##1
17078 \fi:
17079 __fp_parse_expand:w
17080 }
17081 }
17082 __fp_tmp:w {vii} __fp_parse_digits_vii:N { 0000000 ; 7 }
17083 __fp_tmp:w {vi} __fp_parse_digits_vi:N { 000000 ; 6 }
17084 __fp_tmp:w {v} __fp_parse_digits_iv:N { 00000 ; 5 }
17085 __fp_tmp:w {iv} __fp_parse_digits_iii:N { 00000 ; 4 }
17086 __fp_tmp:w {iii} __fp_parse_digits_ii:N { 0000 ; 3 }
17087 __fp_tmp:w {ii} __fp_parse_digits_i:N { 000 ; 2 }
17088 __fp_tmp:w {i} __fp_parse_digits_:N { 00 ; 1 }
17089 \cs_new:Npn __fp_parse_digits_:N { ; ; 0 }

(End definition for __fp_parse_digits_vii:N and others.)

```

## 28.4 Parsing one number

This function finds one number, and packs the symbol which follows in an `\__fp_parse_infix_...` csname. #1 is the previous  $\langle precedence \rangle$ , and #2 the first token of the operand. We distinguish four cases: #2 is equal to `\scan_stop:` in meaning, #2 is a different control sequence, #2 is a digit, and #2 is something else (this last case is split further later). Despite the earlier f-expansion, #2 may still be expandable if it was protected by `\exp_not:N`, as may happen with the L<sup>A</sup>T<sub>E</sub>X 2<sub>&</sub> command `\protect`. Using a well placed `\reverse_if:N`, this case is sent to `\__fp_parse_one_fp>NN` which deals with it robustly.

```

17090 \cs_new:Npn __fp_parse_one:Nw #1 #2
17091 {
17092 \if_catcode:w \scan_stop: \exp_not:N #2
17093 \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
17094 \exp_after:wN \reverse_if:N
17095 \fi:
17096 \if_meaning:w \scan_stop: #2
17097 \exp_after:wN \exp_after:wN
17098 \exp_after:wN __fp_parse_one_fp:NN
17099 \else:
17100 \exp_after:wN \exp_after:wN
17101 \exp_after:wN __fp_parse_one_register:NN
17102 \fi:

```

```

17103 \else:
17104 \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
17105 \exp_after:wN \exp_after:wN
17106 \exp_after:wN __fp_parse_one_digit:NN
17107 \else:
17108 \exp_after:wN \exp_after:wN
17109 \exp_after:wN __fp_parse_one_other:NN
17110 \fi:
17111 \fi:
17112 #1 #2
17113 }

```

(End definition for `\__fp_parse_one:Nw`.)

`\__fp_parse_one_fp:NN`  
`\__fp_exp_after_expr_mark_f:nw`  
`\__fp_exp_after_?_f:nw`

This function receives a *precedence* and a control sequence equal to `\scan_stop:` in meaning. There are three cases.

- `\s_fp` starts a floating point number, and we call `\__fp_exp_after_f:nw`, which f-expands after the floating point.
- `\s_fp_expr_mark` is a premature end, we call `\__fp_exp_after_expr_mark_-f:nw`, which triggers an `fp-early-end` error.
- For a control sequence not containing `\s_fp`, we call `\__fp_exp_after_?_f:nw`, causing a `bad-variable` error.

This scheme is extensible: additional types can be added by starting the variables with a scan mark of the form `\s_fp_<type>` and defining `\__fp_exp_after_<type>_f:nw`. In all cases, we make sure that the second argument of `\__fp_parse_infix:NN` is correctly expanded. A special case only enabled in L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> is that if `\protect` is encountered then the error message mentions the control sequence which follows it rather than `\protect` itself. The test for L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> uses `@unexpandable@protect` rather than `\protect` because `\protect` is often `\scan_stop:` hence “does not exist”.

```

17114 \cs_new:Npn __fp_parse_one_fp:NN #1
17115 {
17116 __fp_exp_after_any_f:nw
17117 {
17118 \exp_after:wN __fp_parse_infix:NN
17119 \exp_after:wN #1 \exp:w __fp_parse_expand:w
17120 }
17121 }
17122 \cs_new:Npn __fp_exp_after_expr_mark_f:nw #1
17123 {
17124 \int_case:nnF { \exp_after:wN \use_i:nnn \use_none:nnn #1 }
17125 {
17126 \c_fp_prec_comma_int { }
17127 \c_fp_prec_tuple_int { }
17128 \c_fp_prec_end_int
17129 {
17130 \exp_after:wN \c_fp_empty_tuple_fp
17131 \exp:w \exp_end_continue_f:w
17132 }
17133 }
17134 }

```

```

17135 __kernel_msg_expandable_error:nn { kernel } { fp-early-end }
17136 \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
17137 }
17138 #1
17139 }
17140 \cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
17141 {
17142 __kernel_msg_expandable_error:nnn { kernel } { bad-variable }
17143 {#2}
17144 \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
17145 }
17146 \cs_set_protected:Npn __fp_tmp:w #1
17147 {
17148 \cs_if_exist:NT #1
17149 {
17150 \cs_gset:cpn { __fp_exp_after_?_f:nw } ##1##2
17151 {
17152 \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w ##1
17153 \str_if_eq:nnTF {##2} { \protect }
17154 {
17155 \cs_if_eq:NNTF ##2 #1 { \use_i:nn } { \use:n }
17156 {
17157 __kernel_msg_expandable_error:nnn { kernel }
17158 { fp-robust-cmd }
17159 }
17160 }
17161 {
17162 __kernel_msg_expandable_error:nnn { kernel }
17163 { bad-variable } {##2}
17164 }
17165 }
17166 }
17167 }
17168 \exp_args:Nc __fp_tmp:w { @unexpandable@protect }

(End definition for __fp_parse_one_fp:NN, __fp_exp_after_expr_mark_f:nw, and __fp_exp_after_?-f:nw.)

```

\\_\_fp\_parse\_one\_register:NN  
\\_\_fp\_parse\_one\_register\_aux:Nw  
\\_\_fp\_parse\_one\_register\_auxii:wwwNw  
\\_\_fp\_parse\_one\_register\_int:www  
\\_\_fp\_parse\_one\_register\_mu:www  
\\_\_fp\_parse\_one\_register\_dim:ww

This is called whenever #2 is a control sequence other than \scan\_stop: in meaning. We special-case \wd, \ht, \dp (see later) and otherwise assume that it is a register, but carefully unpack it with \tex\_the:D within braces. First, we find the exponent following #2. Then we unpack #2 with \tex\_the:D, and the auxii auxiliary distinguishes integer registers from dimensions/skips from muskips, according to the presence of a period and/or of pt. For integers, simply convert  $\langle value \rangle e \langle exponent \rangle$  to a floating point number with \\_\_fp\_parse:n (this is somewhat wasteful). For other registers, the decimal rounding provided by TeX does not accurately represent the binary value that it manipulates, so we extract this binary value as a number of scaled points with \int\_value:w \dim\_to\_decimal\_in\_sp:n {  $\langle decimal\ value \rangle$  pt }, and use an auxiliary of \dim\_to\_fp:n, which performs the multiplication by  $2^{-16}$ , correctly rounded.

```

17169 \cs_new:Npn __fp_parse_one_register:NN #1#2
17170 {
17171 \exp_after:wN __fp_parse_infix_after_operand:NwN
17172 \exp_after:wN #1
17173 \exp:w \exp_end_continue_f:w

```

```

17174 _fp_parse_one_register_special:N #2
17175 \exp_after:wN _fp_parse_one_register_aux:Nw
17176 \exp_after:wN #2
17177 \int_value:w
17178 \exp_after:wN _fp_parse_exponent:N
17179 \exp:w _fp_parse_expand:w
17180 }
17181 \cs_new:Npx _fp_parse_one_register_aux:Nw #1
17182 {
17183 \exp_not:n
17184 {
17185 \exp_after:wN \use:nn
17186 \exp_after:wN _fp_parse_one_register_auxii:wwwNw
17187 }
17188 \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
17189 ; \exp_not:N _fp_parse_one_register_dim:ww
17190 \tl_to_str:n { pt } ; \exp_not:N _fp_parse_one_register_mu:www
17191 . \tl_to_str:n { pt } ; \exp_not:N _fp_parse_one_register_int:www
17192 \s_fp_stop
17193 }
17194 \exp_args:Nno \use:nn
17195 { \cs_new:Npn _fp_parse_one_register_auxii:wwwNw #1 . #2 }
17196 { \tl_to_str:n { pt } #3 ; #4#5 \s_fp_stop }
17197 { #4 #1.#2; }
17198 \exp_args:Nno \use:nn
17199 { \cs_new:Npn _fp_parse_one_register_mu:www #1 }
17200 { \tl_to_str:n { mu } ; #2 ; }
17201 { _fp_parse_one_register_dim:ww #1 ; }
17202 \cs_new:Npn _fp_parse_one_register_int:www #1; #2.; #3;
17203 { _fp_parse:n { #1 e #3 } }
17204 \cs_new:Npn _fp_parse_one_register_dim:ww #1; #2;
17205 {
17206 \exp_after:wN _fp_from_dim_test:ww
17207 \int_value:w #2 \exp_after:wN ,
17208 \int_value:w \dim_to_decimal_in_sp:n { #1 pt } ;
17209 }

```

(End definition for `\_fp_parse_one_register>NN` and others.)

The `\wd`, `\dp`, `\ht` primitives expect an integer argument. We abuse the exponent parser to find the integer argument: simply include the exponent marker `e`. Once that “exponent” is found, use `\tex_the:D` to find the box dimension and then copy what we did for dimensions.

```

17210 \cs_new:Npn _fp_parse_one_register_special:N #1
17211 {
17212 \if_meaning:w \box_wd:N #1 _fp_parse_one_register_wd:w \fi:
17213 \if_meaning:w \box_ht:N #1 _fp_parse_one_register_wd:w \fi:
17214 \if_meaning:w \box_dp:N #1 _fp_parse_one_register_wd:w \fi:
17215 \if_meaning:w \infty #1
17216 _fp_parse_one_register_math>NNw \infty #1
17217 \fi:
17218 \if_meaning:w \pi #1
17219 _fp_parse_one_register_math>NNw \pi #1
17220 \fi:

```

```

17221 }
17222 \cs_new:Npn __fp_parse_one_register_math:NNw
17223 #1#2#3#4 __fp_parse_expand:w
17224 {
17225 #3
17226 \str_if_eq:nnTF {#1} {#2}
17227 {
17228 __kernel_msg_expandable_error:nnn
17229 { kernel } { fp-infnty-pi } {#1}
17230 \c_nan_fp
17231 }
17232 { #4 __fp_parse_expand:w }
17233 }
17234 \cs_new:Npn __fp_parse_one_register_wd:w
17235 #1#2 \exp_after:wN #3#4 __fp_parse_expand:w
17236 {
17237 #1
17238 \exp_after:wN __fp_parse_one_register_wd:Nw
17239 #4 __fp_parse_expand:w e
17240 }
17241 \cs_new:Npn __fp_parse_one_register_wd:Nw #1#2 ;
17242 {
17243 \exp_after:wN __fp_from_dim_test:ww
17244 \exp_after:wN 0 \exp_after:wN ,
17245 \int_value:w \dim_to_decimal_in_sp:n { #1 #2 } ;
17246 }

```

(End definition for `\__fp_parse_one_register_special:N` and others.)

#### `\__fp_parse_one_digit>NN`

A digit marks the beginning of an explicit floating point number. Once the number is found, we catch the case of overflow and underflow with `\__fp_sanitize:wN`, then `\__fp_parse_infix_after_operand:NwN` expands `\__fp_parse_infix>NN` after the number we find, to wrap the following infix operator as required. Finding the number itself begins by removing leading zeros: further steps are described later.

```

17247 \cs_new:Npn __fp_parse_one_digit:NN #1
17248 {
17249 \exp_after:wN __fp_parse_infix_after_operand:NwN
17250 \exp_after:wN #1
17251 \exp:w \exp_end_continue_f:w
17252 \exp_after:wN __fp_sanitize:wN
17253 \int_value:w __fp_int_eval:w 0 __fp_parse_trim_zeros:N
17254 }

```

(End definition for `\__fp_parse_one_digit>NN`.)

#### `\__fp_parse_one_other>NN`

For this function, #2 is a character token which is not a digit. If it is an ASCII letter, `\__fp_parse_letters:N` beyond this one and give the result to `\__fp_parse_word:Nw`. Otherwise, the character is assumed to be a prefix operator, and we build `\__fp_parse_prefix_<operator>:Nw`.

```

17255 \cs_new:Npn __fp_parse_one_other>NN #1 #2
17256 {
17257 \if_int_compare:w
17258 __fp_int_eval:w
17259 ('#2 \if_int_compare:w '#2 > 'Z - 32 \fi:) / 26

```

```

17260 = 3 \exp_stop_f:
17261 \exp_after:wN _fp_parse_word:Nw
17262 \exp_after:wN #1
17263 \exp_after:wN #2
17264 \exp:w \exp_after:wN _fp_parse_letters:N
17265 \exp:w
17266 \else:
17267 \exp_after:wN _fp_parse_prefix>NNN
17268 \exp_after:wN #1
17269 \exp_after:wN #2
17270 \cs:w
17271 _fp_parse_prefix_ \token_to_str:N #2 :Nw
17272 \exp_after:wN
17273 \cs_end:
17274 \exp:w
17275 \fi:
17276 _fp_parse_expand:w
17277 }

```

(End definition for `\_fp_parse_one_other>NN.`)

`\_fp_parse_word:Nw`  
`\_fp_parse_letters:N`

Finding letters is a simple recursion. Once `\_fp_parse_letters:N` has done its job, we try to build a control sequence from the word `#2`. If it is a known word, then the corresponding action is taken, and otherwise, we complain about an unknown word, yield `\c_nan_fp`, and look for the following infix operator. Note that the unknown word could be a mistyped function as well as a mistyped constant, so there is no way to tell whether to look for arguments; we do not. The standard requires “inf” and “infinity” and “nan” to be recognized regardless of case, but we probably don’t want to allow every `l3fp` word to have an arbitrary mixture of lower and upper case, so we test and use a differently-named control sequence.

```

17278 \cs_new:Npn _fp_parse_word:Nw #1#2;
17279 {
17280 \cs_if_exist_use:cF { _fp_parse_word:#2:N }
17281 {
17282 \cs_if_exist_use:cF
17283 { _fp_parse_caseless_ \str_foldcase:n {#2} :N }
17284 {
17285 _kernel_msg_expandable_error:nnn
17286 { kernel } { unknown-fp-word } {#2}
17287 \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
17288 _fp_parse_infix:NN
17289 }
17290 }
17291 #1
17292 }
17293 \cs_new:Npn _fp_parse_letters:N #
17294 {
17295 \exp_end_continue_f:w
17296 \if_int_compare:w
17297 \if_catcode:w \scan_stop: \exp_not:N #1
17298 0
17299 \else:
17300 _fp_int_eval:w
17301 ('#1 \if_int_compare:w '#1 > 'Z - 32 \fi:) / 26

```

```

17302 \fi:
17303 = 3 \exp_stop_f:
17304 \exp_after:wN #1
17305 \exp:w \exp_after:wN _fp_parse_letters:N
17306 \exp:w
17307 \else:
17308 _fp_parse_return_semicolon:w #1
17309 \fi:
17310 _fp_parse_expand:w
17311 }

```

(End definition for `\_fp_parse_word:Nw` and `\_fp_parse_letters:N`.)

```
_fp_parse_prefix:NNN
\fp_parse_prefix_unknown:NNN
```

For this function, #1 is the previous *precedence*, #2 is the operator just seen, and #3 is a control sequence which implements the operator if it is a known operator. If this control sequence is `\scan_stop:`, then the operator is in fact unknown. Either the expression is missing a number there (if the operator is valid as an infix operator), and we put `nan`, wrapping the infix operator in a csname as appropriate, or the character is simply invalid in floating point expressions, and we continue looking for a number, starting again from `\_fp_parse_one:Nw`.

```

17312 \cs_new:Npn _fp_parse_prefix:NNN #1#2#3
17313 {
17314 \if_meaning:w \scan_stop: #3
17315 \exp_after:wN _fp_parse_prefix_unknown:NNN
17316 \exp_after:wN #2
17317 \fi:
17318 #3 #1
17319 }
17320 \cs_new:Npn _fp_parse_prefix_unknown:NNN #1#2#3
17321 {
17322 \cs_if_exist:cTF { _fp_parse_infix_ \token_to_str:N #1 :N }
17323 {
17324 _kernel_msg_expandable_error:nnn
17325 { kernel } { fp-missing-number } {#1}
17326 \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
17327 _fp_parse_infix:NN #3 #1
17328 }
17329 {
17330 _kernel_msg_expandable_error:nnn
17331 { kernel } { fp-unknown-symbol } {#1}
17332 _fp_parse_one:Nw #3
17333 }
17334 }
```

(End definition for `\_fp_parse_prefix:NNN` and `\_fp_parse_prefix_unknown:NNN`.)

#### 28.4.1 Numbers: trimming leading zeros

Numbers are parsed as follows: first we trim leading zeros, then if the next character is a digit, start reading a significand  $\geq 1$  with the set of functions `\_fp_parse_large...`; if it is a period, the significand is  $< 1$ ; and otherwise it is zero. In the second case, trim additional zeros after the period, counting them for an exponent shift  $\langle exp_1 \rangle < 0$ , then read the significand with the set of functions `\_fp_parse_small...`. Once the significand is read, read the exponent if `e` is present.

```
__fp_parse_trim_zeros:N
 __fp_parse_trim_end:w
```

This function expects an already expanded token. It removes any leading zero, then distinguishes three cases: if the first non-zero token is a digit, then call `\_\_fp\_parse\_large:N` (the significand is  $\geq 1$ ); if it is `.`, then continue trimming zeros with `\_\_fp\_parse\_strim\_zeros:N`; otherwise, our number is exactly zero, and we call `\_\_fp\_parse\_zero:` to take care of that case.

```
17335 \cs_new:Npn __fp_parse_trim_zeros:N #1
17336 {
17337 \if:w 0 \exp_not:N #1
17338 \exp_after:wN __fp_parse_trim_zeros:N
17339 \exp:w
17340 \else:
17341 \if:w . \exp_not:N #1
17342 \exp_after:wN __fp_parse_strim_zeros:N
17343 \exp:w
17344 \else:
17345 __fp_parse_trim_end:w #1
17346 \fi:
17347 \fi:
17348 __fp_parse_expand:w
17349 }
17350 \cs_new:Npn __fp_parse_trim_end:w #1 \fi: \fi: __fp_parse_expand:w
17351 {
17352 \fi:
17353 \fi:
17354 \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
17355 \exp_after:wN __fp_parse_large:N
17356 \else:
17357 \exp_after:wN __fp_parse_zero:
17358 \fi:
17359 #1
17360 }
```

(End definition for `\_\_fp\_parse\_trim\_zeros:N` and `\_\_fp\_parse\_trim\_end:w`.)

```
__fp_parse_strim_zeros:N
 __fp_parse_strim_end:w
```

If we have removed all digits until a period (or if the body started with a period), then enter the “`small_trim`” loop which outputs  $-1$  for each removed 0. Those  $-1$  are added to an integer expression waiting for the exponent. If the first non-zero token is a digit, call `\_\_fp\_parse\_small:N` (our significand is smaller than 1), and otherwise, the number is an exact zero. The name `strim` stands for “small trim”.

```
17361 \cs_new:Npn __fp_parse_strim_zeros:N #1
17362 {
17363 \if:w 0 \exp_not:N #1
17364 - 1
17365 \exp_after:wN __fp_parse_strim_zeros:N \exp:w
17366 \else:
17367 __fp_parse_strim_end:w #1
17368 \fi:
17369 __fp_parse_expand:w
17370 }
17371 \cs_new:Npn __fp_parse_strim_end:w #1 \fi: __fp_parse_expand:w
17372 {
17373 \fi:
17374 \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
```

```

17375 \exp_after:wN __fp_parse_small:N
17376 \else:
17377 \exp_after:wN __fp_parse_zero:
17378 \fi:
17379 #1
17380 }

```

(End definition for `\__fp_parse_strim_zeros:N` and `\__fp_parse_strim_end:w`.)

`\__fp_parse_zero:` After reading a significand of 0, find any exponent, then put a sign of 1 for `\__fp-sanitize:wN`, which removes everything and leaves an exact zero.

```

17381 \cs_new:Npn __fp_parse_zero:
17382 {
17383 \exp_after:wN ; \exp_after:wN 1
17384 \int_value:w __fp_parse_exponent:N
17385 }

```

(End definition for `\__fp_parse_zero:..`)

#### 28.4.2 Number: small significand

`\__fp_parse_small:N` This function is called after we have passed the decimal separator and removed all leading zeros from the significand. It is followed by a non-zero digit (with any catcode). The goal is to read up to 16 digits. But we can't do that all at once, because `\int_value:w` (which allows us to collect digits and continue expanding) can only go up to 9 digits. Hence we grab digits in two steps of 8 digits. Since `#1` is a digit, read seven more digits using `\__fp_parse_digits_vii:N`. The `small_leading` auxiliary leaves those digits in the `\int_value:w`, and grabs some more, or stops if there are no more digits. Then the `pack_leading` auxiliary puts the various parts in the appropriate order for the processing further up.

```

17386 \cs_new:Npn __fp_parse_small:N #1
17387 {
17388 \exp_after:wN __fp_parse_pack_leading:NNNNNww
17389 \int_value:w __fp_int_eval:w 1 \token_to_str:N #1
17390 \exp_after:wN __fp_parse_small_leading:wwNN
17391 \int_value:w 1
17392 \exp_after:wN __fp_parse_digits_vii:N
17393 \exp:w __fp_parse_expand:w
17394 }

```

(End definition for `\__fp_parse_small:N`.)

`\__fp_parse_small_leading:wwNN`

`\__fp_parse_small_leading:wwNN 1 <digits> ; <zeros> ; <number of zeros>`

We leave `<digits>` `<zeros>` in the input stream: the functions used to grab digits are such that this constitutes digits 1 through 8 of the significand. Then prepare to pack 8 more digits, with an exponent shift of zero (this shift is used in the case of a large significand). If `#4` is a digit, leave it behind for the packing function, and read 6 more digits to reach a total of 15 digits: further digits are involved in the rounding. Otherwise put 8 zeros in to complete the significand, then look for an exponent.

```

17395 \cs_new:Npn __fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
17396 {
17397 #1 #2
17398 \exp_after:wN __fp_parse_pack_trailing:NNNNNNww

```

```

17399 \exp_after:wN 0
17400 \int_value:w _fp_int_eval:w 1
17401 \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
17402 \token_to_str:N #4
17403 \exp_after:wN _fp_parse_small_trailing:wwNN
17404 \int_value:w 1
17405 \exp_after:wN _fp_parse_digits_vi:N
17406 \exp:w
17407 \else:
17408 0000 0000 _fp_parse_exponent:Nw #4
17409 \fi:
17410 _fp_parse_expand:w
17411 }

```

(End definition for `\_fp_parse_small_leading:wwNN`.)

```

_fp_parse_small_trailing:wwNN 1 <digits> ; <zeros> ; <number of zeros>
<next token>

```

Leave digits 10 to 15 (arguments #1 and #2) in the input stream. If the *<next token>* is a digit, it is the 16th digit, we keep it, then the `small_round` auxiliary considers this digit and all further digits to perform the rounding: the function expands to nothing, to +0 or to +1. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```

17412 \cs_new:Npn _fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
17413 {
17414 #1 #2
17415 \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
17416 \token_to_str:N #4
17417 \exp_after:wN _fp_parse_small_round:NN
17418 \exp_after:wN #4
17419 \exp:w
17420 \else:
17421 0 _fp_parse_exponent:Nw #4
17422 \fi:
17423 _fp_parse_expand:w
17424 }

```

(End definition for `\_fp_parse_small_trailing:wwNN`.)

```

_fp_parse_pack_trailing:NNNNNNNww
_fp_parse_pack_leading:NNNNNNww
_fp_parse_pack_carry:w

```

Those functions are expanded after all the digits are found, we took care of the rounding, as well as the exponent. The last argument is the exponent. The previous five arguments are 8 digits which we pack in groups of 4, and the argument before that is 1, except in the rare case where rounding lead to a carry, in which case the argument is 2. The `trailing` function has an exponent shift as its first argument, which we add to the exponent found in the `e...` syntax. If the trailing digits cause a carry, the integer expression for the leading digits is incremented (+1 in the code below). If the leading digits propagate this carry all the way up, the function `\_fp_parse_pack_carry:w` increments the exponent, and changes the significand from 0000... to 1000...: this is simple because such a carry can only occur to give rise to a power of 10.

```

17425 \cs_new:Npn _fp_parse_pack_trailing:NNNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
17426 {
17427 \if_meaning:w 2 #2 + 1 \fi:
17428 ; #8 + #1 ; {#3#4#5#6} {#7};
17429 }

```

```

17430 \cs_new:Npn __fp_parse_pack_leading:NNNNNww #1 #2#3#4#5 #6; #7;
17431 {
17432 + #7
17433 \if_meaning:w 2 #1 __fp_parse_pack_carry:w \fi:
17434 ; 0 {#2#3#4#5} {#6}
17435 }
17436 \cs_new:Npn __fp_parse_pack_carry:w \fi: ; 0 #1
17437 { \fi: + 1 ; 0 {1000} }

(End definition for __fp_parse_pack_trailing:NNNNNNww, __fp_parse_pack_leading:NNNNNww, and
__fp_parse_pack_carry:w.)

```

### 28.4.3 Number: large significand

Parsing a significand larger than 1 is a little bit more difficult than parsing small significands. We need to count the number of digits before the decimal separator, and add that to the final exponent. We also need to test for the presence of a dot each time we run out of digits, and branch to the appropriate `parse_small` function in those cases.

`\__fp_parse_large:N` This function is followed by the first non-zero digit of a “large” significand ( $\geq 1$ ). It is called within an integer expression for the exponent. Grab up to 7 more digits, for a total of 8 digits.

```

17438 \cs_new:Npn __fp_parse_large:N #1
17439 {
17440 \exp_after:wN __fp_parse_large_leading:wwNN
17441 \int_value:w 1 \token_to_str:N #1
17442 \exp_after:wN __fp_parse_digits_vii:N
17443 \exp:w __fp_parse_expand:w
17444 }

```

(End definition for \\_\_fp\_parse\_large:N.)

`\__fp_parse_large_leading:wwNN`  
 $\langle \text{next token} \rangle$

We shift the exponent by the number of digits in `#1`, namely the target number, 8, minus the  $\langle \text{number of zeros} \rangle$  (number of digits missing). Then prepare to pack the 8 first digits. If the  $\langle \text{next token} \rangle$  is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try to grab the end of our 8 first digits, branching to the `small` functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the  $\langle \text{zeros} \rangle$  to complete the 8 first digits, insert 8 more, and look for an exponent.

```

17445 \cs_new:Npn __fp_parse_large_leading:wwNN #1 ; #2; #3 #4
17446 {
17447 + \c_fp_half_prec_int - #3
17448 \exp_after:wN __fp_parse_pack_leading:NNNNNww
17449 \int_value:w __fp_int_eval:w 1 #1
17450 \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
17451 \exp_after:wN __fp_parse_large_trailing:wwNN
17452 \int_value:w 1 \token_to_str:N #4
17453 \exp_after:wN __fp_parse_digits_vi:N
17454 \exp:w
17455 \else:
17456 \if:w . \exp_not:N #4

```

```

17457 \exp_after:wN __fp_parse_small_leading:wwNN
17458 \int_value:w 1
17459 \cs:w
17460 __fp_parse_digits_
17461 __fp_int_to_roman:w #3
17462 :N \exp_after:wN
17463 \cs_end:
17464 \exp:w
17465 \else:
17466 #2
17467 \exp_after:wN __fp_parse_pack_trailing:NNNNNNww
17468 \exp_after:wN 0
17469 \int_value:w 1 0000 0000
17470 __fp_parse_exponent:Nw #4
17471 \fi:
17472 \fi:
17473 __fp_parse_expand:w
17474 }

```

(End definition for `\__fp_parse_large_leading:wwNN`.)

`\__fp_parse_large_trailing:wwNN 1 <digits> ; <zeros> ; <number of zeros>`  
`<next token>`

We have just read 15 digits. If the `<next token>` is a digit, then the exponent shift caused by this block of 8 digits is 8, first argument to the `pack_trailing` function. We keep the `<digits>` and this 16-th digit, and find how this should be rounded using `\__fp_parse_large_round:NN`. Otherwise, the exponent shift is the number of `<digits>`, 7 minus the `<number of zeros>`, and we test for a decimal point. This case happens in 123451234512345.67 with exactly 15 digits before the decimal separator. Then branch to the appropriate `small` auxiliary, grabbing a few more digits to complement the digits we already grabbed. Finally, if this is truly the end of the significand, look for an exponent after using the `<zeros>` and providing a 16-th digit of 0.

```

17475 \cs_new:Npn __fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
17476 {
17477 \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
17478 \exp_after:wN __fp_parse_pack_trailing:NNNNNNww
17479 \exp_after:wN \c__fp_half_prec_int
17480 \int_value:w __fp_int_eval:w 1 #1 \token_to_str:N #4
17481 \exp_after:wN __fp_parse_large_round:NN
17482 \exp_after:wN #4
17483 \exp:w
17484 \else:
17485 \exp_after:wN __fp_parse_pack_trailing:NNNNNNww
17486 \int_value:w __fp_int_eval:w 7 - #3 \exp_stop_f:
17487 \int_value:w __fp_int_eval:w 1 #1
17488 \if:w . \exp_not:N #4
17489 \exp_after:wN __fp_parse_small_trailing:wwNN
17490 \int_value:w 1
17491 \cs:w
17492 __fp_parse_digits_
17493 __fp_int_to_roman:w #3
17494 :N \exp_after:wN
17495 \cs_end:

```

```

17496 \exp:w
17497 \else:
17498 #2 0 __fp_parse_exponent:Nw #4
17499 \fi:
17500 \fi:
17501 __fp_parse_expand:w
17502 }

```

(End definition for `\__fp_parse_large_trailing:wwNN`.)

#### 28.4.4 Number: beyond 16 digits, rounding

`\__fp_parse_round_loop:N`  
`\__fp_parse_round_up:N`

This loop is called when rounding a number (whether the mantissa is small or large). It should appear in an integer expression. This function reads digits one by one, until reaching a non-digit, and adds 1 to the integer expression for each digit. If all digits found are 0, the function ends the expression by ;0, otherwise by ;1. This is done by switching the loop to `round_up` at the first non-zero digit, thus we avoid to test whether digits are 0 or not once we see a first non-zero digit.

```

17503 \cs_new:Npn __fp_parse_round_loop:N #1
17504 {
17505 \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
17506 + 1
17507 \if:w 0 \token_to_str:N #1
17508 \exp_after:wN __fp_parse_round_loop:N
17509 \exp:w
17510 \else:
17511 \exp_after:wN __fp_parse_round_up:N
17512 \exp:w
17513 \fi:
17514 \else:
17515 __fp_parse_return_semicolon:w 0 #1
17516 \fi:
17517 __fp_parse_expand:w
17518 }
17519 \cs_new:Npn __fp_parse_round_up:N #1
17520 {
17521 \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
17522 + 1
17523 \exp_after:wN __fp_parse_round_up:N
17524 \exp:w
17525 \else:
17526 __fp_parse_return_semicolon:w 1 #1
17527 \fi:
17528 __fp_parse_expand:w
17529 }

```

(End definition for `\__fp_parse_round_loop:N` and `\__fp_parse_round_up:N`.)

`\__fp_parse_round_after:wn`

After the loop `\__fp_parse_round_loop:N`, this function fetches an exponent with `\__fp_parse_exponent:N`, and combines it with the number of digits counted by `\__fp_parse_round_loop:N`. At the same time, the result 0 or 1 is added to the surrounding integer expression.

```

17530 \cs_new:Npn __fp_parse_round_after:wn #1; #2
17531 {

```

```

17532 + #2 \exp_after:wN ;
17533 \int_value:w \fp_int_eval:w #1 + \fp_parse_exponent:N
17534 }

```

(End definition for `\fp_parse_round_after:wN`.)

`\fp_parse_small_round:NN`  
`\fp_parse_round_after:wN`

Here, `#1` is the digit that we are currently rounding (we only care whether it is even or odd). If `#2` is not a digit, then fetch an exponent and expand to ; $\langle\text{exponent}\rangle$  only. Otherwise, we expand to `+0` or `+1`, then ; $\langle\text{exponent}\rangle$ . To decide which, call `\fp_round_s>NNNw` to know whether to round up, giving it as arguments a sign 0 (all explicit numbers are positive), the digit `#1` to round, the first following digit `#2`, and either `+0` or `+1` depending on whether the following digits are all zero or not. This last argument is obtained by `\fp_parse_round_loop:N`, whose number of digits we discard by multiplying it by 0. The exponent which follows the number is also fetched by `\fp_parse_round_after:wN`.

```

17535 \cs_new:Npn \fp_parse_small_round:NN #1#2
17536 {
17537 \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
17538 +
17539 \exp_after:wN \fp_round_s:NNNw
17540 \exp_after:wN 0
17541 \exp_after:wN #1
17542 \exp_after:wN #2
17543 \int_value:w \fp_int_eval:w
17544 \exp_after:wN \fp_parse_round_after:wN
17545 \int_value:w \fp_int_eval:w 0 * \fp_int_eval:w 0
17546 \exp_after:wN \fp_parse_round_loop:N
17547 \exp:w
17548 \else:
17549 \fp_parse_exponent:Nw #2
17550 \fi:
17551 \fp_parse_expand:w
17552 }

```

(End definition for `\fp_parse_small_round:NN` and `\fp_parse_round_after:wN`.)

`\fp_parse_large_round:NN`  
`\fp_parse_large_round_test:NN`  
`\fp_parse_large_round_aux:wNN`

Large numbers are harder to round, as there may be a period in the way. Again, `#1` is the digit that we are currently rounding (we only care whether it is even or odd). If there are no more digits (`#2` is not a digit), then we must test for a period: if there is one, then switch to the rounding function for small significands, otherwise fetch an exponent. If there are more digits (`#2` is a digit), then round, checking with `\fp_parse_round_loop:N` if all further digits vanish, or some are non-zero. This loop is not enough, as it is stopped by a period. After the loop, the aux function tests for a period: if it is present, then we must continue looking for digits, this time discarding the number of digits we find.

```

17553 \cs_new:Npn \fp_parse_large_round:NN #1#2
17554 {
17555 \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
17556 +
17557 \exp_after:wN \fp_round_s:NNNw
17558 \exp_after:wN 0
17559 \exp_after:wN #1
17560 \exp_after:wN #2

```

```

17561 \int_value:w _fp_int_eval:w
17562 \exp_after:wN _fp_parse_large_round_aux:wNN
17563 \int_value:w _fp_int_eval:w 1
17564 \exp_after:wN _fp_parse_round_loop:N
17565 \else: %^^A could be dot, or e, or other
17566 \exp_after:wN _fp_parse_large_round_test>NN
17567 \exp_after:wN #1
17568 \exp_after:wN #2
17569 \fi:
17570 }
17571 \cs_new:Npn _fp_parse_large_round_test>NN #1#2
17572 {
17573 \if:w . \exp_not:N #2
17574 \exp_after:wN _fp_parse_small_round>NN
17575 \exp_after:wN #1
17576 \exp:w
17577 \else:
17578 _fp_parse_exponent:Nw #2
17579 \fi:
17580 _fp_parse_expand:w
17581 }
17582 \cs_new:Npn _fp_parse_large_round_aux:wNN #1 ; #2 #3
17583 {
17584 + #2
17585 \exp_after:wN _fp_parse_round_after:wN
17586 \int_value:w _fp_int_eval:w #1
17587 \if:w . \exp_not:N #3
17588 + 0 * _fp_int_eval:w 0
17589 \exp_after:wN _fp_parse_round_loop:N
17590 \exp:w \exp_after:wN _fp_parse_expand:w
17591 \else:
17592 \exp_after:wN ;
17593 \exp_after:wN 0
17594 \exp_after:wN #3
17595 \fi:
17596 }

```

(End definition for `\_fp_parse_large_round>NN`, `\_fp_parse_large_round_test>NN`, and `\_fp_parse_large_round_aux:wNN`.)

#### 28.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication is implicit.

```

\fp_parse:n { 3.2 erf(0.1) }
\fp_parse:n { 3.2 e\l_my_int }
\fp_parse:n { 3.2 \c_pi_fp }

```

The first case indicates that just looking one character ahead for an “e” is not enough, since we would mistake the function `erf` for an exponent of “rf”. An alternative would be to look two tokens ahead and check if what follows is a sign or a digit, considering in that case that we must be finding an exponent. But taking care of the second case requires that we unpack registers after `e`. However, blindly expanding the two tokens ahead completely would break the third example (unpacking is even worse). Indeed, in

the course of reading 3.2, `\c_pi_fp` is expanded to `\s_fp \_fp_chk:w 1 0 {-1} {3141}` ... ; and `\s_fp` stops the expansion. Expanding two tokens ahead would then force the expansion of `\_fp_chk:w` (despite it being protected), and that function tries to produce an error.

What can we do? Really, the reason why this last case breaks is that just as TeX does, we should read ahead as little as possible. Here, the only case where there may be an exponent is if the first token ahead is `e`. Then we expand (and possibly unpack) the second token.

`\_fp_parse_exponent:Nw`

This auxiliary is convenient to smuggle some material through `\fi:` ending conditional processing. We place those `\fi:` (argument #2) at a very odd place because this allows us to insert `\_fp_int_eval:w ...` there if needed.

```

17597 \cs_new:Npn _fp_parse_exponent:Nw #1 #2 _fp_parse_expand:w
17598 {
17599 \exp_after:wN ;
17600 \int_value:w #2 _fp_parse_exponent:N #1
17601 }
```

(End definition for `\_fp_parse_exponent:Nw`.)

`\_fp_parse_exponent:N`

This function should be called within an `\int_value:w` expansion (or within an integer expression). It leaves digits of the exponent behind it in the input stream, and terminates the expansion with a semicolon. If there is no `e` (or `E`), leave an exponent of 0. If there is an `e` or `E`, expand the next token to run some tests on it. The first rough test is that if the character code of #1 is greater than that of 9 (largest code valid for an exponent, less than any code valid for an identifier), there was in fact no exponent; otherwise, we search for the sign of the exponent.

```

17602 \cs_new:Npn _fp_parse_exponent:N #1
17603 {
17604 \if:w e \if:w E \exp_not:N #1 e \else: \exp_not:N #1 \fi:
17605 \exp_after:wN _fp_parse_exponent_aux:NN
17606 \exp_after:wN #1
17607 \exp:w
17608 \else:
17609 0 _fp_parse_return_semicolon:w #1
17610 \fi:
17611 _fp_parse_expand:w
17612 }
17613 \cs_new:Npn _fp_parse_exponent_aux:NN #1#2
17614 {
17615 \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #2
17616 0 \else: '#2 \fi: > '9 \exp_stop_f:
17617 0 \exp_after:wN ; \exp_after:wN #1
17618 \else:
17619 \exp_after:wN _fp_parse_exponent_sign:N
17620 \fi:
17621 #2
17622 }
```

(End definition for `\_fp_parse_exponent:N` and `\_fp_parse_exponent_aux:NN`.)

`\_fp_parse_exponent_sign:N`

Read signs one by one (if there is any).

```
17623 \cs_new:Npn _fp_parse_exponent_sign:N #1
```

```

17624 {
17625 \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
17626 \exp_after:wN __fp_parse_exponent_sign:N
17627 \exp:w \exp_after:wN __fp_parse_expand:w
17628 \else:
17629 \exp_after:wN __fp_parse_exponent_body:N
17630 \exp_after:wN #1
17631 \fi:
17632 }

```

(End definition for `\__fp_parse_exponent_sign:N`.)

`\__fp_parse_exponent_body:N` An exponent can be an explicit integer (most common case), or various other things (most of which are invalid).

```

17633 \cs_new:Npn __fp_parse_exponent_body:N #1
17634 {
17635 \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
17636 \token_to_str:N #1
17637 \exp_after:wN __fp_parse_exponent_digits:N
17638 \exp:w
17639 \else:
17640 __fp_parse_exponent_keep:NTF #1
17641 { __fp_parse_return_semicolon:w #1 }
17642 {
17643 \exp_after:wN ;
17644 \exp:w
17645 }
17646 \fi:
17647 __fp_parse_expand:w
17648 }

```

(End definition for `\__fp_parse_exponent_body:N`.)

`\__fp_parse_exponent_digits:N` Read digits one by one, and leave them behind in the input stream. When finding a non-digit, stop, and insert a semicolon. Note that we do not check for overflow of the exponent, hence there can be a TeX error. It is mostly harmless, except when parsing `0e9876543210`, which should be a valid representation of 0, but is not.

```

17649 \cs_new:Npn __fp_parse_exponent_digits:N #1
17650 {
17651 \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
17652 \token_to_str:N #1
17653 \exp_after:wN __fp_parse_exponent_digits:N
17654 \exp:w
17655 \else:
17656 __fp_parse_return_semicolon:w #1
17657 \fi:
17658 __fp_parse_expand:w
17659 }

```

(End definition for `\__fp_parse_exponent_digits:N`.)

`\__fp_parse_exponent_keep:NTF` This is the last building block for parsing exponents. The argument `#1` is already fully expanded, and neither `+` nor `-` nor a digit. It can be:

- `\s__fp`, marking the start of an internal floating point, invalid here;

- another control sequence equal to `\relax`, probably a bad variable;
- a register: in this case we make sure that it is an integer register, not a dimension;
- a character other than +, - or digits, again, an error.

```

17660 \prg_new_conditional:Npn __fp_parse_exponent_keep:N #1 { TF }
17661 {
17662 \if_catcode:w \scan_stop: \exp_not:N #1
17663 \if_meaning:w \scan_stop: #1
17664 \if_int_compare:w
17665 __fp_str_if_eq:nn { \s_fp } { \exp_not:N #1 }
17666 = 0 \exp_stop_f:
17667 0
17668 __kernel_msg_expandable_error:nnn
17669 { kernel } { fp-after-e } { floating-point~ }
17670 \prg_return_true:
17671 \else:
17672 0
17673 __kernel_msg_expandable_error:nnn
17674 { kernel } { bad-variable } {#1}
17675 \prg_return_false:
17676 \fi:
17677 \else:
17678 \if_int_compare:w
17679 __fp_str_if_eq:nn { \int_value:w #1 } { \tex_the:D #1 }
17680 = 0 \exp_stop_f:
17681 \int_value:w #1
17682 \else:
17683 0
17684 __kernel_msg_expandable_error:nnn
17685 { kernel } { fp-after-e } { dimension~#1 }
17686 \fi:
17687 \prg_return_false:
17688 \fi:
17689 \else:
17690 0
17691 __kernel_msg_expandable_error:nnn
17692 { kernel } { fp-missing } { exponent }
17693 \prg_return_true:
17694 \fi:
17695 }
```

(End definition for `\__fp_parse_exponent_keep:NTF.`)

## 28.5 Constants, functions and prefix operators

### 28.5.1 Prefix operators

`\__fp_parse_prefix_+:Nw` A unary + does nothing: we should continue looking for a number.

```

17696 \cs_new_eq:cN { __fp_parse_prefix_+:Nw } __fp_parse_one:Nw
```

(End definition for `\__fp_parse_prefix_+:Nw.`)

\\_fp\_parse\_apply\_function:NNNwN Here, #1 is a precedence, #2 is some extra data used by some functions, #3 is e.g., \\_\\_fp\_sin\_o:w, and expands once after the calculation, #4 is the operand, and #5 is a \\_\\_fp\_parse\_infix\_...:N function. We feed the data #2, and the argument #4, to the function #3, which expands \exp:w thus the infix function #5.

```

17697 \cs_new:Npn __fp_parse_apply_function:NNNwN #1#2#3#4@#5
17698 {
17699 #3 #2 #4 @
17700 \exp:w \exp_end_continue_f:w #5 #1
17701 }
```

(End definition for \\_\\_fp\_parse\_apply\_function:NNNwN.)

\\_fp\_parse\_apply\_unary:NNNwN  
\\_\\_fp\_parse\_apply\_unary\_chk:NwNw  
\\_\\_fp\_parse\_apply\_unary\_chk:nNNNw  
\\_\\_fp\_parse\_apply\_unary\_type:NNN  
\\_\\_fp\_parse\_apply\_unary\_error:NNw

In contrast to \\_\\_fp\_parse\_apply\_function:NNNwN, this checks that the operand #4 is a single argument (namely there is a single ;). We use the fact that any floating point starts with a “safe” token like \s\\_fp. If there is no argument produce the fp-no-arg error; if there are at least two produce fp-multi-arg. For the error message extract the mathematical function name (such as sin) from the expl3 function that computes it, such as \\_\\_fp\_sin\_o:w.

In addition, since there is a single argument we can dispatch on type and check that the resulting function exists. This catches things like sin((1,2)) where it does not make sense to take the sine of a tuple.

```

17702 \cs_new:Npn __fp_parse_apply_unary:NNNwN #1#2#3#4@#5
17703 {
17704 __fp_parse_apply_unary_chk:NwNw #4 @ . \s_fp_stop
17705 __fp_parse_apply_unary_type:NNN
17706 #3 #2 #4 @
17707 \exp:w \exp_end_continue_f:w #5 #1
17708 }
17709 \cs_new:Npn __fp_parse_apply_unary_chk:NwNw #1#2 ; #3#4 \s_fp_stop
17710 {
17711 \if_meaning:w @ #3 \else:
17712 \token_if_eq_meaning:NNTF . #3
17713 { __fp_parse_apply_unary_chk:nNNNNw { no } }
17714 { __fp_parse_apply_unary_chk:nNNNNw { multi } }
17715 \fi:
17716 }
17717 \cs_new:Npn __fp_parse_apply_unary_chk:nNNNNw #1#2#3#4#5#6 @
17718 {
17719 #2
17720 __fp_error:nffn { fp-#1-arg } { __fp_func_to_name:N #4 } { } { }
17721 \exp_after:wN #4 \exp_after:wN #5 \c_nan_fp @
17722 }
17723 \cs_new:Npn __fp_parse_apply_unary_type:NNN #1#2#3
17724 {
17725 __fp_change_func_type:NNN #3 #1 __fp_parse_apply_unary_error:NNw
17726 #2 #3
17727 }
17728 \cs_new:Npn __fp_parse_apply_unary_error:NNw #1#2#3 @
17729 { __fp_invalid_operation_o:fw { __fp_func_to_name:N #1 } #3 }
```

(End definition for \\_\\_fp\_parse\_apply\_unary:NNNwN and others.)

```
__fp_parse_prefix_:-:Nw
__fp_parse_prefix_!:Nw
```

The unary - and boolean not are harder: we parse the operand using a precedence equal to the maximum of the previous precedence ##1 and the precedence \c\_fp\_prec\_not\_int of the unary operator, then call the appropriate \\_\_fp\_{operation}\_o:w function, where the *{operation}* is set\_sign or not.

```
17730 \cs_set_protected:Npn __fp_tmp:w #1#2#3#4
17731 {
17732 \cs_new:cpn { __fp_parse_prefix_ #1 :Nw } ##1
17733 {
17734 \exp_after:wN __fp_parse_apply_unary:NNNwN
17735 \exp_after:wN ##1
17736 \exp_after:wN #4
17737 \exp_after:wN #3
17738 \exp:w
17739 \if_int_compare:w #2 < ##1
17740 __fp_parse_operand:Nw ##1
17741 \else:
17742 __fp_parse_operand:Nw #2
17743 \fi:
17744 __fp_parse_expand:w
17745 }
17746 }
17747 __fp_tmp:w - \c_fp_not_int __fp_set_sign_o:w 2
17748 __fp_tmp:w ! \c_fp_not_int __fp_not_o:w ?
```

(End definition for \\_\_fp\_parse\_prefix\_:-:Nw and \\_\_fp\_parse\_prefix\_!:Nw.)

```
__fp_parse_prefix_.:Nw
```

Numbers which start with a decimal separator (a period) end up here. Of course, we do not look for an operand, but for the rest of the number. This function is very similar to \\_\_fp\_parse\_one\_digit:NN but calls \\_\_fp\_parse\_strim\_zeros:N to trim zeros after the decimal point, rather than the trim\_zeros function for zeros before the decimal point.

```
17749 \cs_new:cpn { __fp_parse_prefix_.:Nw } #1
17750 {
17751 \exp_after:wN __fp_parse_infix_after_operand:NwN
17752 \exp_after:wN #1
17753 \exp:w \exp_end_continue_f:w
17754 \exp_after:wN __fp_SANITIZE:wN
17755 \int_value:w __fp_int_eval:w 0 __fp_parse_strim_zeros:N
17756 }
```

(End definition for \\_\_fp\_parse\_prefix\_.:Nw.)

```
__fp_parse_prefix_(:Nw
__fp_parse_lparen_after:NwN
```

The left parenthesis is treated as a unary prefix operator because it appears in exactly the same settings. If the previous precedence is \c\_fp\_prec\_func\_int we are parsing arguments of a function and commas should not build tuples; otherwise commas should build tuples. We distinguish these cases by precedence: \c\_fp\_prec\_comma\_int for the case of arguments, \c\_fp\_prec\_tuple\_int for the case of tuples. Once the operand is found, the lparen\_after auxiliary makes sure that there was a closing parenthesis (otherwise it complains), and leaves in the input stream an operand, fetching the following infix operator.

```
17757 \cs_new:cpn { __fp_parse_prefix_(:Nw } #1
17758 {
17759 \exp_after:wN __fp_parse_lparen_after:NwN
```

```

17760 \exp_after:wN #1
17761 \exp:w
17762 \if_int_compare:w #1 = \c_fp_prec_func_int
17763 _fp_parse_operand:Nw \c_fp_prec_comma_int
17764 \else:
17765 _fp_parse_operand:Nw \c_fp_prec_tuple_int
17766 \fi:
17767 _fp_parse_expand:w
17768 }
17769 \cs_new:Npx _fp_parse_lparen_after:NwN #1#2 @ #3
17770 {
17771 \exp_not:N \token_if_eq_meaning:NNTF #3
17772 \exp_not:c { _fp_parse_infix_ }:N }
17773 {
17774 \exp_not:N _fp_exp_after_array_f:w #2 \s_fp_expr_stop
17775 \exp_not:N \exp_after:wN
17776 \exp_not:N _fp_parse_infix_after_paren:NN
17777 \exp_not:N \exp_after:wN #1
17778 \exp_not:N \exp:w
17779 \exp_not:N _fp_parse_expand:w
17780 }
17781 {
17782 \exp_not:N _kernel_msg_expandable_error:nnn
17783 { kernel } { fp-missing } {) }
17784 \exp_not:N \tl_if_empty:nT {#2} \exp_not:N \c_fp_empty_tuple_fp
17785 #2 @
17786 \exp_not:N \use_none:n #3
17787 }
17788 }

```

(End definition for `\_fp_parse_prefix_(:Nw` and `\_fp_parse_lparen_after:NwN`.)

`\_fp_parse_prefix_):Nw`

The right parenthesis can appear as a prefix in two similar cases: in an empty tuple or tuple ending with a comma, or in an empty argument list or argument list ending with a comma, such as in `max(1,2,)` or in `rand()`.

```

17789 \cs_new:cpn { _fp_parse_prefix_):Nw } #1
17790 {
17791 \if_int_compare:w #1 = \c_fp_prec_comma_int
17792 \else:
17793 \if_int_compare:w #1 = \c_fp_prec_tuple_int
17794 \exp_after:wN \c_fp_empty_tuple_fp \exp:w
17795 \else:
17796 _kernel_msg_expandable_error:nnn
17797 { kernel } { fp-missing-number } {) }
17798 \exp_after:wN \c_nan_fp \exp:w
17799 \fi:
17800 \exp_end_continue_f:w
17801 \fi:
17802 _fp_parse_infix_after_paren:NN #1)
17803 }

```

(End definition for `\_fp_parse_prefix_):Nw`.)

### 28.5.2 Constants

`\_fp_parse_word_inf:N` Some words correspond to constant floating points. The floating point constant is left as a result of `\_fp_parse_one:Nw` after expanding `\_fp_parse_infix>NN`.

```

17804 \cs_set_protected:Npn _fp_tmp:w #1 #2
17805 {
17806 \cs_new:cpn { _fp_parse_word_#1:N }
17807 { \exp_after:wN #2 \exp:w \exp_end_continue_f:w _fp_parse_infix>NN }
17808 }
17809 _fp_tmp:w { inf } \c_inf_fp
17810 _fp_tmp:w { nan } \c_nan_fp
17811 _fp_tmp:w { pi } \c_pi_fp
17812 _fp_tmp:w { deg } \c_one_degree_fp
17813 _fp_tmp:w { true } \c_one_fp
17814 _fp_tmp:w { false } \c_zero_fp

```

(End definition for `\_fp_parse_word_inf:N` and others.)

`\_fp_parse_caseless_inf:N` Copies of `\_fp_parse_word_...:N` commands, to allow arbitrary case as mandated by the standard.

```

17815 \cs_new_eq:NN _fp_parse_caseless_inf:N _fp_parse_word_inf:N
17816 \cs_new_eq:NN _fp_parse_caseless_infinity:N _fp_parse_word_inf:N
17817 \cs_new_eq:NN _fp_parse_caseless_nan:N _fp_parse_word_nan:N

```

(End definition for `\_fp_parse_caseless_inf:N`, `\_fp_parse_caseless_infinity:N`, and `\_fp_parse_caseless_nan:N`.)

`\_fp_parse_word_pt:N` Dimension units are also floating point constants but their value is not stored as a floating point constant. We give the values explicitly here.

```

17818 \cs_set_protected:Npn _fp_tmp:w #1 #2
17819 {
17820 \cs_new:cpn { _fp_parse_word_#1:N }
17821 {
17822 _fp_exp_after_f:nw { _fp_parse_infix>NN }
17823 \s_fp _fp_chk:w 10 #2 ;
17824 }
17825 }
17826 _fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
17827 _fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
17828 _fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
17829 _fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
17830 _fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
17831 _fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
17832 _fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
17833 _fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
17834 _fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
17835 _fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
17836 _fp_tmp:w {sp} { {-4} {1525} {8789} {0625} {0000} }

```

(End definition for `\_fp_parse_word_pt:N` and others.)

`\_fp_parse_word_em:N` The font-dependent units `em` and `ex` must be evaluated on the fly. We reuse an auxiliary of `\dim_to_fp:n`.

```

17837 \tl_map_inline:nn { {em} {ex} }
17838 {

```

```

17839 \cs_new:cpn { __fp_parse_word_#1:N }
17840 {
17841 \exp_after:wN __fp_from_dim_test:ww
17842 \exp_after:wN 0 \exp_after:wN ,
17843 \int_value:w \dim_to_decimal_in_sp:n { 1 #1 } \exp_after:wN ;
17844 \exp:w \exp_end_continue_f:w __fp_parse_infix>NN
17845 }
17846 }

```

(End definition for `\__fp_parse_word_em:N` and `\__fp_parse_word_ex:N`.)

### 28.5.3 Functions

```

__fp_parse_unary_function:NNN
__fp_parse_function:NNN
17847 \cs_new:Npn __fp_parse_unary_function:NNN #1#2#3
17848 {
17849 \exp_after:wN __fp_parse_apply_unary:NNNwN
17850 \exp_after:wN #3
17851 \exp_after:wN #2
17852 \exp_after:wN #1
17853 \exp:w
17854 __fp_parse_operand:Nw \c__fp_prec_func_int __fp_parse_expand:w
17855 }
17856 \cs_new:Npn __fp_parse_function:NNN #1#2#3
17857 {
17858 \exp_after:wN __fp_parse_apply_function:NNNwN
17859 \exp_after:wN #3
17860 \exp_after:wN #2
17861 \exp_after:wN #1
17862 \exp:w
17863 __fp_parse_operand:Nw \c__fp_prec_func_int __fp_parse_expand:w
17864 }

```

(End definition for `\__fp_parse_unary_function:NNN` and `\__fp_parse_function:NNN`.)

## 28.6 Main functions

`\__fp_parse:n`  
`\__fp_parse_o:n`  
`\__fp_parse_after:ww`

Start an `\exp:w` expansion so that `\__fp_parse:n` expands in two steps. The `\__fp_parse_operand:Nw` function performs computations until reaching an operation with precedence `\c__fp_prec_end_int` or less, namely, the end of the expression. The marker `\s__fp_expr_mark` indicates that the next token is an already parsed version of an infix operator, and `\__fp_parse_infix_end:N` has infinitely negative precedence. Finally, clean up a (well-defined) set of extra tokens and stop the initial expansion with `\exp_end:..`.

```

17865 \cs_new:Npn __fp_parse:n #1
17866 {
17867 \exp:w
17868 \exp_after:wN __fp_parse_after:ww
17869 \exp:w
17870 __fp_parse_operand:Nw \c__fp_prec_end_int
17871 __fp_parse_expand:w #1
17872 \s__fp_expr_mark __fp_parse_infix_end:N
17873 \s__fp_expr_stop

```

```

17874 \exp_end:
17875 }
17876 \cs_new:Npn _fp_parse_after:ww
17877 #1@ _fp_parse_infix_end:N \s_fp_expr_stop #2 { #2 #1 }
17878 \cs_new:Npn _fp_parse_o:n #1
17879 {
17880 \exp:w
17881 \exp_after:wN _fp_parse_after:ww
17882 \exp:w
17883 _fp_parse_operand:Nw \c_fp_prec_end_int
17884 _fp_parse_expand:w #1
17885 \s_fp_expr_mark _fp_parse_infix_end:N
17886 \s_fp_expr_stop
17887 {
17888 \exp_end_continue_f:w
17889 _fp_exp_after_any_f:nw { \exp_after:wN \exp_stop_f: }
17890 }
17891 }

```

(End definition for `\_fp_parse:n`, `\_fp_parse_o:n`, and `\_fp_parse_after:ww`.)

`\_fp_parse_operand:Nw`  
`\_fp_parse_continue:NwN`

```

17892 \cs_new:Npn _fp_parse_operand:Nw #1
17893 {
17894 \exp_end_continue_f:w
17895 \exp_after:wN _fp_parse_continue:NwN
17896 \exp_after:wN #1
17897 \exp:w \exp_end_continue_f:w
17898 \exp_after:wN _fp_parse_one:Nw
17899 \exp_after:wN #1
17900 \exp:w
17901 }
17902 \cs_new:Npn _fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }

```

(End definition for `\_fp_parse_operand:Nw` and `\_fp_parse_continue:NwN`.)

`\_fp_parse_apply_binary:NwNwN`  
`\_fp_parse_apply_binary_chk>NN`  
`\_fp_parse_apply_binary_error:NNN`

Receives  $\langle\text{precedence}\rangle \langle\text{operand}_1\rangle @ \langle\text{operation}\rangle \langle\text{operand}_2\rangle @ \langle\text{infix command}\rangle$ . Builds the appropriate call to the  $\langle\text{operation}\rangle$  #3, dispatching on both types. If the resulting control sequence does not exist, the operation is not allowed.

This is redefined in `l3fp-extras`.

```

17903 \cs_new:Npn _fp_parse_apply_binary:NwNwN #1 #2#3@ #4 #5#6@ #7
17904 {
17905 \exp_after:wN _fp_parse_continue:NwN
17906 \exp_after:wN #1
17907 \exp:w \exp_end_continue_f:w
17908 \exp_after:wN _fp_parse_apply_binary_chk>NN
17909 \cs:w
17910 __fp
17911 _fp_type_from_scan:N #2
17912 _#4
17913 _fp_type_from_scan:N #5
17914 _o:ww
17915 \cs_end:

```

```

17916 #4
17917 #2#3 #5#6
17918 \exp:w \exp_end_continue_f:w #7 #1
17919 }
17920 \cs_new:Npn _fp_parse_apply_binary_chk:NN #1#2
17921 {
17922 \if_meaning:w \scan_stop: #1
17923 _fp_parse_apply_binary_error:NNN #2
17924 \fi:
17925 #1
17926 }
17927 \cs_new:Npn _fp_parse_apply_binary_error:NNN #1#2#3
17928 {
17929 #2
17930 _fp_invalid_operation_o:Nww #1
17931 }

```

(End definition for `\_fp_parse_apply_binary:NwNwN`, `\_fp_parse_apply_binary_chk:NN`, and `\_fp_parse_apply_binary_error:NNN`.)

Applies the operator #1 to its two arguments, dispatching according to their types, and expands once after the result. The `rev` version swaps its arguments before doing this.

```

17932 \cs_new:Npn _fp_binary_type_o:Nww #1 #2#3 ; #4
17933 {
17934 \exp_after:wN _fp_parse_apply_binary_chk:NN
17935 \cs:w
17936 __fp
17937 _fp_type_from_scan:N #2
17938 _ #1
17939 _fp_type_from_scan:N #4
17940 _o:ww
17941 \cs_end:
17942 #1
17943 #2 #3 ; #4
17944 }
17945 \cs_new:Npn _fp_binary_rev_type_o:Nww #1 #2#3 ; #4#5 ;
17946 {
17947 \exp_after:wN _fp_parse_apply_binary_chk:NN
17948 \cs:w
17949 __fp
17950 _fp_type_from_scan:N #4
17951 _ #1
17952 _fp_type_from_scan:N #2
17953 _o:ww
17954 \cs_end:
17955 #1
17956 #4 #5 ; #2 #3 ;
17957 }

```

(End definition for `\_fp_binary_type_o:Nww` and `\_fp_binary_rev_type_o:Nww`.)

## 28.7 Infix operators

`\_fp_parse_infix_after_operand:NwN`

```

17958 \cs_new:Npn __fp_parse_infix_after_operand:NwN #1 #2;
17959 {
17960 __fp_exp_after_f:nw { __fp_parse_infix>NN #1 }
17961 #2;
17962 }
17963 \cs_new:Npn __fp_parse_infix>NN #1 #2
17964 {
17965 \if_catcode:w \scan_stop: \exp_not:N #2
17966 \if_int_compare:w
17967 __fp_str_if_eq:nn { \s__fp_expr_mark } { \exp_not:N #2 }
17968 = 0 \exp_stop_f:
17969 \exp_after:wN \exp_after:wN
17970 \exp_after:wN __fp_parse_infix_mark:NNN
17971 \else:
17972 \exp_after:wN \exp_after:wN
17973 \exp_after:wN __fp_parse_infix_juxt:N
17974 \fi:
17975 \else:
17976 \if_int_compare:w
17977 __fp_int_eval:w
17978 ('#2 \if_int_compare:w '#2 > 'Z - 32 \fi:) / 26
17979 = 3 \exp_stop_f:
17980 \exp_after:wN \exp_after:wN
17981 \exp_after:wN __fp_parse_infix_juxt:N
17982 \else:
17983 \exp_after:wN __fp_parse_infix_check:NNN
17984 \cs:w
17985 __fp_parse_infix_ \token_to_str:N #2 :N
17986 \exp_after:wN \exp_after:wN \exp_after:wN
17987 \cs_end:
17988 \fi:
17989 \fi:
17990 #1
17991 #2
17992 }
17993 \cs_new:Npn __fp_parse_infix_check:NNN #1#2#3
17994 {
17995 \if_meaning:w \scan_stop: #1
17996 __kernel_msg_expandable_error:nnn
17997 { kernel } { fp-missing } { * }
17998 \exp_after:wN __fp_parse_infix_mul:N
17999 \exp_after:wN #2
18000 \exp_after:wN #3
18001 \else:
18002 \exp_after:wN #1
18003 \exp_after:wN #2
18004 \exp:w \exp_after:wN __fp_parse_expand:w
18005 \fi:
18006 }

```

(End definition for `\__fp_parse_infix_after_operand:NwN`.)

`\__fp_parse_infix_after_paren>NN` Variant of `\__fp_parse_infix>NN` for use after a closing parenthesis. The only difference is that `\__fp_parse_infix_juxt:N` is replaced by `\__fp_parse_infix_mul:N`.

```

18007 \cs_new:Npn __fp_parse_infix_after_paren:NN #1 #2
18008 {
18009 \if_catcode:w \scan_stop: \exp_not:N #2
18010 \if_int_compare:w
18011 __fp_str_if_eq:nn { \s__fp_expr_mark } { \exp_not:N #2 }
18012 = 0 \exp_stop_f:
18013 \exp_after:wN \exp_after:wN
18014 \exp_after:wN __fp_parse_infix_mark:NNN
18015 \else:
18016 \exp_after:wN \exp_after:wN
18017 \exp_after:wN __fp_parse_infix_mul:N
18018 \fi:
18019 \else:
18020 \if_int_compare:w
18021 __fp_int_eval:w
18022 ('#2 \if_int_compare:w '#2 > 'Z - 32 \fi:) / 26
18023 = 3 \exp_stop_f:
18024 \exp_after:wN \exp_after:wN
18025 \exp_after:wN __fp_parse_infix_mul:N
18026 \else:
18027 \exp_after:wN __fp_parse_infix_check:NNN
18028 \cs:w
18029 __fp_parse_infix_ \token_to_str:N #2 :N
18030 \exp_after:wN \exp_after:wN \exp_after:wN
18031 \cs_end:
18032 \fi:
18033 \fi:
18034 #1
18035 #2
18036 }

```

(End definition for `\__fp_parse_infix_after_paren:NN`.)

### 28.7.1 Closing parentheses and commas

`\__fp_parse_infix_mark:NNN` As an infix operator, `\s__fp_expr_mark` means that the next token (#3) has already gone through `\__fp_parse_infix:NN` and should be provided the precedence #1. The scan mark #2 is discarded.

```
18037 \cs_new:Npn __fp_parse_infix_mark:NNN #1#2#3 { #3 #1 }
```

(End definition for `\__fp_parse_infix_mark:NNN`.)

`\__fp_parse_infix_end:N` This one is a little bit odd: force every previous operator to end, regardless of the precedence.

```
18038 \cs_new:Npn __fp_parse_infix_end:N #1
18039 { @ \use_none:n __fp_parse_infix_end:N }
```

(End definition for `\__fp_parse_infix_end:N`.)

`\__fp_parse_infix_)N` This is very similar to `\__fp_parse_infix_end:N`, complaining about an extra closing parenthesis if the previous operator was the beginning of the expression, with precedence `\c__fp_prec_end_int`.

```
18040 \cs_set_protected:Npn __fp_tmp:w #1
18041 {
```

```

18042 \cs_new:Npn #1 ##1
18043 {
18044 \if_int_compare:w ##1 > \c_fp_prec_end_int
18045 \exp_after:wN @
18046 \exp_after:wN \use_none:n
18047 \exp_after:wN #1
18048 \else:
18049 _kernel_msg_expandable_error:nnn { kernel } { fp-extra } { }
18050 \exp_after:wN _fp_parse_infix:NN
18051 \exp_after:wN ##1
18052 \exp:w \exp_after:wN _fp_parse_expand:w
18053 \fi:
18054 }
18055 }
18056 \exp_args:Nc _fp_tmp:w { _fp_parse_infix_):N }

(End definition for _fp_parse_infix_):N.)

```

As for other infix operations, if the previous operations has higher precedence the comma waits. Otherwise we call `\_fp_parse_operand:Nw` to read more comma-delimited arguments that `\_fp_parse_infix_comma:w` simply concatenates into a @-delimited array. The first comma in a tuple that is not a function argument is distinguished: in that case call `\_fp_parse_apply_comma:NwNwN` whose job is to convert the first item of the tuple and an array of the remaining items into a tuple. In contrast to `\_fp_parse_apply_binary:NwNwN` this function's operands are not single-object arrays.

```

18057 \cs_set_protected:Npn _fp_tmp:w #1
18058 {
18059 \cs_new:Npn #1 ##1
18060 {
18061 \if_int_compare:w ##1 > \c_fp_prec_comma_int
18062 \exp_after:wN @
18063 \exp_after:wN \use_none:n
18064 \exp_after:wN #1
18065 \else:
18066 \if_int_compare:w ##1 < \c_fp_prec_comma_int
18067 \exp_after:wN @
18068 \exp_after:wN _fp_parse_apply_comma:NwNwN
18069 \exp_after:wN ,
18070 \exp:w
18071 \else:
18072 \exp_after:wN _fp_parse_infix_comma:w
18073 \exp:w
18074 \fi:
18075 _fp_parse_operand:Nw \c_fp_prec_comma_int
18076 \exp_after:wN _fp_parse_expand:w
18077 \fi:
18078 }
18079 }
18080 \exp_args:Nc _fp_tmp:w { _fp_parse_infix_,:N }
18081 \cs_new:Npn _fp_parse_infix_comma:w #1 @
18082 { #1 @ \use_none:n }
18083 \cs_new:Npn _fp_parse_apply_comma:NwNwN #1 #2@ #3 #4@ #5
18084 {
18085 \exp_after:wN _fp_parse_continue:NwN

```

```

18086 \exp_after:wN #1
18087 \exp:w \exp_end_continue_f:w
18088 _fp_exp_after_tuple_f:nw { }
18089 \s_fp_tuple _fp_tuple_chk:w { #2 #4 } ;
18090 #5 #1
18091 }

```

(End definition for `\_fp_parse_infix_::N`, `\_fp_parse_infix_comma:w`, and `\_fp_parse_apply_-comma:NwNwN`.)

### 28.7.2 Usual infix operators

`\_fp_parse_infix_+:N`    `\_fp_parse_infix_-=N`    `\_fp_parse_infix_juxt:N`    `\_fp_parse_infix_/:N`    `\_fp_parse_infix_mul:N`    `\_fp_parse_infix_and:N`    `\_fp_parse_infix_or:N`    `\_fp_parse_infix_^:N`

As described in the “work plan”, each infix operator has an associated `\..._infix_...` function, a computing function, and precedence, given as arguments to `\_fp_tmp:w`. Using the general mechanism for arithmetic operations. The power operation must be associative in the opposite order from all others. For this, we use two distinct precedences.

```

18092 \cs_set_protected:Npn _fp_tmp:w #1#2#3#4
18093 {
18094 \cs_new:Npn #1 ##1
18095 {
18096 \if_int_compare:w ##1 < #3
18097 \exp_after:wN @
18098 \exp_after:wN _fp_parse_apply_binary:NwNwN
18099 \exp_after:wN #2
18100 \exp:w
18101 _fp_parse_operand:Nw #4
18102 \exp_after:wN _fp_parse_expand:w
18103 \else:
18104 \exp_after:wN @
18105 \exp_after:wN \use_none:n
18106 \exp_after:wN #1
18107 \fi:
18108 }
18109 }
18110 \exp_args:Nc _fp_tmp:w { _fp_parse_infix_^:N } ^ ~
18111 \c_fp_prec_hat_i_int \c_fp_prec_hat_int
18112 \exp_args:Nc _fp_tmp:w { _fp_parse_infix_juxt:N } *
18113 \c_fp_prec_juxt_int \c_fp_prec_juxt_int
18114 \exp_args:Nc _fp_tmp:w { _fp_parse_infix_/:N } /
18115 \c_fp_prec_times_int \c_fp_prec_times_int
18116 \exp_args:Nc _fp_tmp:w { _fp_parse_infix_mul:N } *
18117 \c_fp_prec_times_int \c_fp_prec_times_int
18118 \exp_args:Nc _fp_tmp:w { _fp_parse_infix_-=N } -
18119 \c_fp_prec_plus_int \c_fp_prec_plus_int
18120 \exp_args:Nc _fp_tmp:w { _fp_parse_infix_+:N } +
18121 \c_fp_prec_plus_int \c_fp_prec_plus_int
18122 \exp_args:Nc _fp_tmp:w { _fp_parse_infix_and:N } &
18123 \c_fp_prec_and_int \c_fp_prec_and_int
18124 \exp_args:Nc _fp_tmp:w { _fp_parse_infix_or:N } |
18125 \c_fp_prec_or_int \c_fp_prec_or_int

```

(End definition for `\_fp_parse_infix_+:N` and others.)

### 28.7.3 Juxtaposition

\\_\\_fp\\_parse\\_infix\\_(\N) When an opening parenthesis appears where we expect an infix operator, we compute the product of the previous operand and the contents of the parentheses using \\_\\_fp\\_parse\\_infix\\_mul\N.

```
18126 \cs_new:cpn { __fp_parse_infix_(:N) #1
18127 { __fp_parse_infix_mul:N #1 (} }
```

(End definition for \\_\\_fp\\_parse\\_infix\\_(\N).)

### 28.7.4 Multi-character cases

\\_\\_fp\\_parse\\_infix\\_\*:\N

```
18128 \cs_set_protected:Npn __fp_tmp:w #1
18129 {
18130 \cs_new:cpn { __fp_parse_infix_*:N } ##1##2
18131 {
18132 \if:w * \exp_not:N ##2
18133 \exp_after:wN #1
18134 \exp_after:wN ##1
18135 \else:
18136 \exp_after:wN __fp_parse_infix_mul:N
18137 \exp_after:wN ##1
18138 \exp_after:wN ##2
18139 \fi:
18140 }
18141 }
18142 \exp_args:Nc __fp_tmp:w { __fp_parse_infix_^\N }
```

(End definition for \\_\\_fp\\_parse\\_infix\\_\*:\N.)

\\_\\_fp\\_parse\\_infix\\_|:\Nw

\\_\\_fp\\_parse\\_infix\&:\Nw

```
18143 \cs_set_protected:Npn __fp_tmp:w #1#2#3
18144 {
18145 \cs_new:Npn #1 ##1##2
18146 {
18147 \if:w #2 \exp_not:N ##2
18148 \exp_after:wN #1
18149 \exp_after:wN ##1
18150 \exp:w \exp_after:wN __fp_parse_expand:w
18151 \else:
18152 \exp_after:wN #3
18153 \exp_after:wN ##1
18154 \exp_after:wN ##2
18155 \fi:
18156 }
18157 }
18158 \exp_args:Nc __fp_tmp:w { __fp_parse_infix_|:N } | __fp_parse_infix_or:N
18159 \exp_args:Nc __fp_tmp:w { __fp_parse_infix_&:N } & __fp_parse_infix_and:N
```

(End definition for \\_\\_fp\\_parse\\_infix\\_|:\Nw and \\_\\_fp\\_parse\\_infix\&:\Nw.)

### 28.7.5 Ternary operator

```

__fp_parse_infix_?:N
__fp_parse_infix_::N
18160 \cs_set_protected:Npn __fp_tmp:w #1#2#3#4
18161 {
18162 \cs_new:Npn #1 ##1
18163 {
18164 \if_int_compare:w ##1 < \c_fp_prec_quest_int
18165 #4
18166 \exp_after:wN @
18167 \exp_after:wN #2
18168 \exp:w
18169 __fp_parse_operand:Nw #3
18170 \exp_after:wN __fp_parse_expand:w
18171 \else:
18172 \exp_after:wN @
18173 \exp_after:wN \use_none:n
18174 \exp_after:wN #1
18175 \fi:
18176 }
18177 }
18178 \exp_args:Nc __fp_tmp:w { __fp_parse_infix_?:N }
18179 __fp_ternary:NwwN \c_fp_prec_quest_int { }
18180 \exp_args:Nc __fp_tmp:w { __fp_parse_infix_::N }
18181 __fp_ternary_auxii:NwwN \c_fp_prec_colon_int
18182 {
18183 __kernel_msg_expandable_error:nnnn
18184 { kernel } { fp-missing } { ? } { ~for~?: }
18185 }

```

(End definition for `\__fp_parse_infix_?:N` and `\__fp_parse_infix_::N`.)

### 28.7.6 Comparisons

```

__fp_parse_infix_<:N
__fp_parse_infix_=:N
__fp_parse_infix_>:N
__fp_parse_infix_!:N
__fp_parse_excl_error:
__fp_parse_compare:NNNNNNN
__fp_parse_compare_auxi:NNNNNN
__fp_parse_compare_auxii:NNNNNN
__fp_parse_compare_end:NNNNw
__fp_compare:wNNNNw
18186 \cs_new:cpn { __fp_parse_infix_<:N } #1
18187 { __fp_parse_compare:NNNNNNN #1 1 0 0 0 0 < }
18188 \cs_new:cpn { __fp_parse_infix_=:N } #1
18189 { __fp_parse_compare:NNNNNNN #1 1 0 0 0 0 = }
18190 \cs_new:cpn { __fp_parse_infix_>:N } #1
18191 { __fp_parse_compare:NNNNNNN #1 1 0 0 0 0 > }
18192 \cs_new:cpn { __fp_parse_infix_!:N } #1
18193 {
18194 \exp_after:wN __fp_parse_compare:NNNNNNN
18195 \exp_after:wN #1
18196 \exp_after:wN 0
18197 \exp_after:wN 1
18198 \exp_after:wN 1
18199 \exp_after:wN 1
18200 \exp_after:wN 1
18201 }
18202 \cs_new:Npn __fp_parse_excl_error:
18203 {
18204 __kernel_msg_expandable_error:nnnn

```

```

18205 { kernel } { fp-missing } { = } { ~after~!. }
18206 }
18207 \cs_new:Npn __fp_parse_compare:NNNNNNN #1
18208 {
18209 \if_int_compare:w #1 < \c__fp_prec_comp_int
18210 \exp_after:wN __fp_parse_compare_auxi:NNNNNNN
18211 \exp_after:wN __fp_parse_excl_error:
18212 \else:
18213 \exp_after:wN @
18214 \exp_after:wN \use_none:n
18215 \exp_after:wN __fp_parse_compare:NNNNNNN
18216 \fi:
18217 }
18218 \cs_new:Npn __fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
18219 {
18220 \if_case:w
18221 __fp_int_eval:w \exp_after:wN ` \token_to_str:N #7 - '<
18222 __fp_int_eval_end:
18223 __fp_parse_compare_auxii:NNNN #2#2#4#5#6
18224 \or: __fp_parse_compare_auxii:NNNN #2#3#2#5#6
18225 \or: __fp_parse_compare_auxii:NNNN #2#3#4#2#6
18226 \or: __fp_parse_compare_auxii:NNNN #2#3#4#5#2
18227 \else: #1 __fp_parse_compare_end:NNNNw #3#4#5#6#7
18228 \fi:
18229 }
18230 \cs_new:Npn __fp_parse_compare_auxii:NNNN #1#2#3#4#5
18231 {
18232 \exp_after:wN __fp_parse_compare_auxi:NNNNNNN
18233 \exp_after:wN \prg_do_nothing:
18234 \exp_after:wN #1
18235 \exp_after:wN #2
18236 \exp_after:wN #3
18237 \exp_after:wN #4
18238 \exp_after:wN #5
18239 \exp:w \exp_after:wN __fp_parse_expand:w
18240 }
18241 \cs_new:Npn __fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
18242 {
18243 \fi:
18244 \exp_after:wN @
18245 \exp_after:wN __fp_parse_apply_compare:NwNNNNNwN
18246 \exp_after:wN \c_one_fp
18247 \exp_after:wN #1
18248 \exp_after:wN #2
18249 \exp_after:wN #3
18250 \exp_after:wN #4
18251 \exp:w
18252 __fp_parse_operand:Nw \c__fp_prec_comp_int __fp_parse_expand:w #5
18253 }
18254 \cs_new:Npn __fp_parse_apply_compare:NwNNNNNwN
18255 #1 #2@ #3 #4#5#6#7 #8@ #9
18256 {
18257 \if_int_odd:w
18258 \if_meaning:w \c_zero_fp #3

```

```

18259 0
18260 \else:
18261 \if_case:w __fp_compare_back_any:ww #8 #2 \exp_stop_f:
18262 #5 \or: #6 \or: #7 \else: #4
18263 \fi:
18264 \fi:
18265 \exp_stop_f:
18266 \exp_after:wN __fp_parse_apply_compare_aux>NNwN
18267 \exp_after:wN \c_one_fp
18268 \else:
18269 \exp_after:wN __fp_parse_apply_compare_aux>NNwN
18270 \exp_after:wN \c_zero_fp
18271 \fi:
18272 #1 #8 #9
18273 }
18274 \cs_new:Npn __fp_parse_apply_compare_aux>NNwN #1 #2 #3; #4
18275 {
18276 \if_meaning:w __fp_parse_compare>NNNNNNN #4
18277 \exp_after:wN __fp_parse_continue_compare>NNwNN
18278 \exp_after:wN #1
18279 \exp_after:wN #2
18280 \exp:w \exp_end_continue_f:w
18281 __fp_exp_after_o:w #3;
18282 \exp:w \exp_end_continue_f:w
18283 \else:
18284 \exp_after:wN __fp_parse_continue:NwN
18285 \exp_after:wN #2
18286 \exp:w \exp_end_continue_f:w
18287 \exp_after:wN #1
18288 \exp:w \exp_end_continue_f:w
18289 \fi:
18290 #4 #2
18291 }
18292 \cs_new:Npn __fp_parse_continue_compare>NNwNN #1#2 #3@ #4#5
18293 { #4 #2 #3@ #1 }

```

(End definition for `\__fp_parse_infix_<:N` and others.)

## 28.8 Tools for functions

Followed by `{<function name>}` `{<code>}` `<float array>` @ this checks all floats are floating point numbers (no tuples).

```

18294 \cs_new:Npn __fp_parse_function_all_fp_o:fnw #1#2#3 @
18295 {
18296 __fp_array_if_all_fp:nTF {#3}
18297 { #2 #3 @ }
18298 {
18299 __fp_error:nffn { fp-bad-args }
18300 {#1}
18301 { \fp_to_tl:n { \s__fp_tuple __fp_tuple_chk:w {#3} ; } }
18302 { }
18303 \exp_after:wN \c_nan_fp
18304 }
18305 }

```

(End definition for `\_fp_parse_function_all_fp_o:fnw`.)

```
_fp_parse_function_one_two:nnw
\fp_parse_function_one_two_error_o:w
\fp_parse_function_one_two_aux:nnw
\fp_parse_function_one_two_auxii:nnw
```

This is followed by `{<function name>} {<code>} <float array>` @. It checks that the `<float array>` consists of one or two floating point numbers (not tuples), then leaves the `<code>` (if there is one float) or its tail (if there are two floats) followed by the `<float array>`. The `<code>` should start with a single token such as `\_fp_atan_default:w` that deals with the single-float case.

The first `\_fp_if_type_fp:NTwFw` test catches the case of no argument and the case of a tuple argument. The next one distinguishes the case of a single argument (no error, just add `\c_one_fp`) from a tuple second argument. Finally check there is no further argument.

```
18306 \cs_new:Npn _fp_parse_function_one_two:nnw #1#2#3
18307 {
18308 _fp_if_type_fp:NTwFw
18309 #3 { } \s_fp _fp_parse_function_one_two_error_o:w \s_fp_stop
18310 _fp_parse_function_one_two_aux:nnw {#1} {#2} #3
18311 }
18312 \cs_new:Npn _fp_parse_function_one_two_error_o:w #1#2#3#4 @
18313 {
18314 _fp_error:nffn { fp-bad-args }
18315 {#2}
18316 { \fp_to_tl:n { \s_fp_tuple _fp_tuple_chk:w {#4} ; } }
18317 { }
18318 \exp_after:wN \c_nan_fp
18319 }
18320 \cs_new:Npn _fp_parse_function_one_two_aux:nnw #1#2 #3; #4
18321 {
18322 _fp_if_type_fp:NTwFw
18323 #4 { }
18324 \s_fp
18325 {
18326 \if_meaning:w @ #4
18327 \exp_after:wN \use_iv:nnnn
18328 \fi:
18329 _fp_parse_function_one_two_error_o:w
18330 }
18331 \s_fp_stop
18332 _fp_parse_function_one_two_auxii:nnw {#1} {#2} #3; #4
18333 }
18334 \cs_new:Npn _fp_parse_function_one_two_auxii:nnw #1#2#3; #4; #5
18335 {
18336 \if_meaning:w @ #5 \else:
18337 \exp_after:wN _fp_parse_function_one_two_error_o:w
18338 \fi:
18339 \use_i:nn {#1} { \use_none:n #2 } #3; #4; #5
18340 }
```

(End definition for `\_fp_parse_function_one_two:nnw` and others.)

```
_fp_tuple_map_o:nw
\fp_tuple_map_loop_o:nnw
```

Apply #1 to all items in the following tuple and expand once afterwards. The code #1 should itself expand once after its result.

```
18341 \cs_new:Npn _fp_tuple_map_o:nw #1 \s_fp_tuple _fp_tuple_chk:w #2 ;
18342 {
```

```

18343 \exp_after:wN \s__fp_tuple
18344 \exp_after:wN _fp_tuple_chk:w
18345 \exp_after:wN {
18346 \exp:w \exp_end_continue_f:w
18347 _fp_tuple_map_loop_o:nw {#1} #2
18348 { \s__fp \prg_break: } ;
18349 \prg_break_point:
18350 \exp_after:wN } \exp_after:wN ;
18351 }
18352 \cs_new:Npn _fp_tuple_map_loop_o:nw #1#2#3 ;
18353 {
18354 \use_none:n #2
18355 #1 #2 #3 ;
18356 \exp:w \exp_end_continue_f:w
18357 _fp_tuple_map_loop_o:nw {#1}
18358 }

```

(End definition for `\_fp_tuple_map_o:nw` and `\_fp_tuple_map_loop_o:nw`.)

`\_fp_tuple_mapthread_o:nww` Apply #1 to pairs of items in the two following tuples and expand once afterwards.

```

18359 \cs_new:Npn _fp_tuple_mapthread_o:nww #1
18360 \s__fp_tuple _fp_tuple_chk:w #2 ;
18361 \s__fp_tuple _fp_tuple_chk:w #3 ;
18362 {
18363 \exp_after:wN \s__fp_tuple
18364 \exp_after:wN _fp_tuple_chk:w
18365 \exp_after:wN {
18366 \exp:w \exp_end_continue_f:w
18367 _fp_tuple_mapthread_loop_o:nw {#1}
18368 #2 { \s__fp \prg_break: } ; @
18369 #3 { \s__fp \prg_break: } ;
18370 \prg_break_point:
18371 \exp_after:wN } \exp_after:wN ;
18372 }
18373 \cs_new:Npn _fp_tuple_mapthread_loop_o:nw #1#2#3 ; #4 @ #5#6 ;
18374 {
18375 \use_none:n #2
18376 \use_none:n #5
18377 #1 #2 #3 ; #5 #6 ;
18378 \exp:w \exp_end_continue_f:w
18379 _fp_tuple_mapthread_loop_o:nw {#1} #4 @
18380 }

```

(End definition for `\_fp_tuple_mapthread_o:nww` and `\_fp_tuple_mapthread_loop_o:nw`.)

## 28.9 Messages

```

18381 _kernel_msg_new:nnn { kernel } { fp-deprecated }
18382 { '#1'~deprecated;~use~'#2' }
18383 _kernel_msg_new:nnn { kernel } { unknown-fp-word }
18384 { Unknown~fp~word~#1. }
18385 _kernel_msg_new:nnn { kernel } { fp-missing }
18386 { Missing~#1~inserted #2. }
18387 _kernel_msg_new:nnn { kernel } { fp-extra }
18388 { Extra~#1~ignored. }

```

```

18389 __kernel_msg_new:nnn { kernel } { fp-early-end }
18390 { Premature~end~in~fp~expression. }
18391 __kernel_msg_new:nnn { kernel } { fp-after-e }
18392 { Cannot~use~#1 after~'e'. }
18393 __kernel_msg_new:nnn { kernel } { fp-missing-number }
18394 { Missing~number~before~'#1'. }
18395 __kernel_msg_new:nnn { kernel } { fp-unknown-symbol }
18396 { Unknown~symbol~#1~ignored. }
18397 __kernel_msg_new:nnn { kernel } { fp-extra-comma }
18398 { Unexpected~comma~turned~to~nan~result. }
18399 __kernel_msg_new:nnn { kernel } { fp-no-arg }
18400 { #1~got~no~argument;~used~nan. }
18401 __kernel_msg_new:nnn { kernel } { fp-multi-arg }
18402 { #1~got~more~than~one~argument;~used~nan. }
18403 __kernel_msg_new:nnn { kernel } { fp-num-args }
18404 { #1~expects~between~#2~and~#3~arguments. }
18405 __kernel_msg_new:nnn { kernel } { fp-bad-args }
18406 { Arguments~in~#1#2~are~invalid. }
18407 __kernel_msg_new:nnn { kernel } { fp-infty-pi }
18408 { Math~command~#1~is~not~an~fp~}
18409 \cs_if_exist:cT { @unexpandable@protect }
18410 {
18411 __kernel_msg_new:nnn { kernel } { fp-robust-cmd }
18412 { Robust~command~#1~invalid~in~fp~expression! }
18413 }
18414
```

## 29 I3fp-assign implementation

```

18415 (*package)
18416 <@=fp>

```

### 29.1 Assigning values

**\fp\_new:N** Floating point variables are initialized to be +0.

```

18417 \cs_new_protected:Npn \fp_new:N #1
18418 { \cs_new_eq:NN #1 \c_zero_fp }
18419 \cs_generate_variant:Nn \fp_new:N {c}

```

(End definition for \fp\_new:N. This function is documented on page 201.)

**\fp\_set:Nn** Simply use \\_\\_fp\_parse:n within various f-expanding assignments.

```

\fp_set:cn 18420 \cs_new_protected:Npn \fp_set:Nn #1#2
\fp_gset:Nn 18421 { __kernel_tl_set:Nx #1 { \exp_not:f { __fp_parse:n {#2} } } }
\fp_gset:cn 18422 \cs_new_protected:Npn \fp_gset:Nn #1#2
\fp_const:Nn 18423 { __kernel_tl_gset:Nx #1 { \exp_not:f { __fp_parse:n {#2} } } }
\fp_const:cn 18424 \cs_new_protected:Npn \fp_const:Nn #1#2
18425 { \tl_const:Nx #1 { \exp_not:f { __fp_parse:n {#2} } } }
18426 \cs_generate_variant:Nn \fp_set:Nn {c}
18427 \cs_generate_variant:Nn \fp_gset:Nn {c}
18428 \cs_generate_variant:Nn \fp_const:Nn {c}

```

(End definition for \fp\_set:Nn, \fp\_gset:Nn, and \fp\_const:Nn. These functions are documented on page 202.)

**\fp\_set\_eq:NN** Copying a floating point is the same as copying the underlying token list.

```

\fp_set_eq:cN 18429 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
\fp_set_eq:Nc 18430 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\fp_set_eq:cc 18431 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }
\fp_gset_eq:NN 18432 \cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }

```

(End definition for `\fp_set_eq:NN` and `\fp_gset_eq:NN`. These functions are documented on page 202.)

**\fp\_gset\_eq:cN** Setting a floating point to zero: copy `\c_zero_fp`.

```

\fp_zero:c 18433 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\fp_gzero:N 18434 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\fp_gzero:c 18435 \cs_generate_variant:Nn \fp_zero:N { c }
18436 \cs_generate_variant:Nn \fp_gzero:N { c }

```

(End definition for `\fp_zero:N` and `\fp_gzero:N`. These functions are documented on page 201.)

**\fp\_zero\_new:N** Set the floating point to zero, or define it if needed.

```

\fp_zero_new:c 18437 \cs_new_protected:Npn \fp_zero_new:N #1
\fp_gzero_new:N 18438 { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\fp_gzero_new:c 18439 \cs_new_protected:Npn \fp_gzero_new:N #1
18440 { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
18441 \cs_generate_variant:Nn \fp_zero_new:N { c }
18442 \cs_generate_variant:Nn \fp_gzero_new:N { c }

```

(End definition for `\fp_zero_new:N` and `\fp_gzero_new:N`. These functions are documented on page 202.)

## 29.2 Updating values

These match the equivalent functions in `\int` and `\skip`.

For the sake of error recovery we should not simply set #1 to  $\#1 \pm (\#2)$ : for instance, if #2 is 0) + 2, the parsing error would be raised at the last closing parenthesis rather than at the closing parenthesis in the user argument. Thus we evaluate #2 instead of just putting parentheses. As an optimization we use `\_fp_parse:n` rather than `\fp_eval:n`, which would convert the result away from the internal representation and back.

```

\fp_add:Nn 18443 \cs_new_protected:Npn \fp_add:Nn { _fp_add:NNNn \fp_set:Nn + }
\fp_gadd:Nn 18444 \cs_new_protected:Npn \fp_gadd:Nn { _fp_add:NNNn \fp_gset:Nn + }
\fp_gadd:cn 18445 \cs_new_protected:Npn \fp_sub:Nn { _fp_add:NNNn \fp_set:Nn - }
\fp_gsub:cn 18446 \cs_new_protected:Npn \fp_gsub:Nn { _fp_add:NNNn \fp_gset:Nn - }
\fp_gsub:Nn 18447 \cs_new_protected:Npn _fp_add:NNNn #1#2#3#4
18448 { #1 #3 { #3 #2 _fp_parse:n {#4} } }
18449 \cs_generate_variant:Nn \fp_add:Nn { c }
18450 \cs_generate_variant:Nn \fp_gadd:Nn { c }
18451 \cs_generate_variant:Nn \fp_sub:Nn { c }
18452 \cs_generate_variant:Nn \fp_gsub:Nn { c }

```

(End definition for `\fp_add:Nn` and others. These functions are documented on page 202.)

### 29.3 Showing values

**\fp\_show:N** This shows the result of computing its argument by passing the right data to \tl\_show:n or \tl\_log:n.

```
18453 \cs_new_protected:Npn \fp_show:N { _fp_show:NN \tl_show:n }
18454 \cs_generate_variant:Nn \fp_show:N { c }
18455 \cs_new_protected:Npn \fp_log:N { _fp_show:NN \tl_log:n }
18456 \cs_generate_variant:Nn \fp_log:N { c }
18457 \cs_new_protected:Npn _fp_show:NN #1#2
18458 {
18459 _kernel_chk_defined:NT #2
18460 { \exp_args:Nx #1 { \token_to_str:N #2 = \fp_to_tl:N #2 } }
18461 }
```

(End definition for \fp\_show:N, \fp\_log:N, and \\_fp\_show:NN. These functions are documented on page 209.)

**\fp\_show:n** Use general tools.

```
18462 \cs_new_protected:Npn \fp_show:n
18463 { \msg_show_eval:Nn \fp_to_tl:n }
18464 \cs_new_protected:Npn \fp_log:n
18465 { \msg_log_eval:Nn \fp_to_tl:n }
```

(End definition for \fp\_show:n and \fp\_log:n. These functions are documented on page 209.)

### 29.4 Some useful constants and scratch variables

**\c\_one\_fp** Some constants.

```
18466 \fp_const:Nn \c_e_fp { 2.718 2818 2845 9045 }
18467 \fp_const:Nn \c_one_fp { 1 }
```

(End definition for \c\_one\_fp and \c\_e\_fp. These variables are documented on page 207.)

**\c\_pi\_fp** We simply round  $\pi$  to and  $\pi/180$  to 16 significant digits.

```
18468 \fp_const:Nn \c_pi_fp { 3.141 5926 5358 9793 }
18469 \fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }
```

(End definition for \c\_pi\_fp and \c\_one\_degree\_fp. These variables are documented on page 208.)

**\l\_tmpa\_fp** Scratch variables are simply initialized there.

```
18470 \fp_new:N \l_tmpa_fp
18471 \fp_new:N \l_tmpb_fp
18472 \fp_new:N \g_tmpa_fp
18473 \fp_new:N \g_tmpb_fp
```

(End definition for \l\_tmpa\_fp and others. These variables are documented on page 208.)

18474

## 30 I3fp-logic Implementation

```
18475 (*package)
18476 (@@=fp)
```

\\_\\_fp\\_parse\\_word\\_max:N Those functions may receive a variable number of arguments.

```
18477 \cs_new:Npn __fp_parse_word_max:N
18478 { __fp_parse_function>NNN __fp_minmax_o:Nw 2 }
18479 \cs_new:Npn __fp_parse_word_min:N
18480 { __fp_parse_function>NNN __fp_minmax_o:Nw 0 }
```

(End definition for \\_\\_fp\\_parse\\_word\\_max:N and \\_\\_fp\\_parse\\_word\\_min:N.)

### 30.1 Syntax of internal functions

- \\_\\_fp\\_compare\\_npos:nwnw { $\langle expo_1 \rangle$ }  $\langle body_1 \rangle$  ; { $\langle expo_2 \rangle$ }  $\langle body_2 \rangle$  ;
- \\_\\_fp\\_minmax\_o:Nw  $\langle sign \rangle$   $\langle floating\ point\ array \rangle$
- \\_\\_fp\\_not\_o:w ?  $\langle floating\ point\ array \rangle$  (with one floating point number only)
- \\_\\_fp\\_&\_o:ww  $\langle floating\ point \rangle$   $\langle floating\ point \rangle$
- \\_\\_fp\\_|\_o:ww  $\langle floating\ point \rangle$   $\langle floating\ point \rangle$
- \\_\\_fp\\_ternary:NwwN, \\_\\_fp\\_ternary\\_auxi:NwwN, \\_\\_fp\\_ternary\\_auxii:NwwN have to be understood.

### 30.2 Tests

\fp\_if\_exist\_p:N Copies of the cs functions defined in I3basics.

```
18481 \prg_new_eq_conditional>NNN \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
18482 \prg_new_eq_conditional>NNN \fp_if_exist:c \cs_if_exist:c { TF , T , F , p }
\fp_if_exist:cTF
```

(End definition for \fp\_if\_exist:NTF. This function is documented on page 204.)

\fp\_if\_nan\_p:n Evaluate and check if the result is a floating point of the same kind as NaN.

```
18483 \prg_new_conditional:Npnn \fp_if_nan:n #1 { TF , T , F , p }
18484 {
18485 \if:w 3 \exp_last_unbraced:Nf __fp_kind:w { __fp_parse:n {\#1} }
18486 \prg_return_true:
18487 \else:
18488 \prg_return_false:
18489 \fi:
18490 }
```

(End definition for \fp\_if\_nan:nTF. This function is documented on page 266.)

### 30.3 Comparison

**\fp\_compare\_p:n** Within floating point expressions, comparison operators are treated as operations, so we evaluate #1, then compare with  $\pm 0$ . Tuples are **true**.

```

\fp_compare_p:nTF
\fp_compare:nTF
\fp_compare_return:w
\fp_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
{
 \exp_after:wN \fp_compare_return:w
 \exp:w \exp_end_continue_f:w \fp_parse:n {#1}
}
\cs_new:Npn \fp_compare_return:w #1#2#3;
{
 \if_charcode:w 0
 \fp_if_type_fp:NTwFw
 #1 { \fp_use_i_delimit_by_s_stop:nw #3 \s_fp_stop }
 \s_fp 1 \s_fp_stop
 \prg_return_false:
 \else:
 \prg_return_true:
 \fi:
}

```

(End definition for **\fp\_compare:nTF** and **\fp\_compare\_return:w**. This function is documented on page 205.)

**\fp\_compare\_p:nNn** Evaluate #1 and #3, using an auxiliary to expand both, and feed the two floating point numbers swapped to **\fp\_compare\_back\_any:ww**, defined below. Compare the result with ‘#2-‘=, which is -1 for <, 0 for =, 1 for > and 2 for ?.

```

\fp_compare_p:nNnTF
\fp_compare:nNnTF
\fp_compare_aux:wn
\fp_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
{
 \if_int_compare:w
 \exp_after:wN \fp_compare_aux:wn
 \exp:w \exp_end_continue_f:w \fp_parse:n {#1} {#3}
 = \fp_int_eval:w '#2 - '=' \fp_int_eval_end:
 \prg_return_true:
 \else:
 \prg_return_false:
 \fi:
}
\cs_new:Npn \fp_compare_aux:wn #1; #2
{
 \exp_after:wN \fp_compare_back_any:ww
 \exp:w \exp_end_continue_f:w \fp_parse:n {#2} #1;
}

```

(End definition for **\fp\_compare:nNnTF** and **\fp\_compare\_aux:wn**. This function is documented on page 204.)

**\fp\_compare\_back\_any:ww**

**\fp\_compare\_back:ww**

**\fp\_compare\_nan:w**

**\fp\_compare\_back\_any:ww**  $\langle y \rangle ; \langle x \rangle$  ;

Expands (in the same way as **\int\_eval:n**) to -1 if  $x < y$ , 0 if  $x = y$ , 1 if  $x > y$ , and 2 otherwise (denoted as  $x?y$ ). If either operand is **nan**, stop the comparison with **\fp\_compare\_nan:w** returning 2. If  $x$  is negative, swap the outputs 1 and -1 (*i.e.*, > and <>); we can henceforth assume that  $x \geq 0$ . If  $y \geq 0$ , and they have the same type, either they are normal and we compare them with **\fp\_compare\_npos:nwnw**, or they

are equal. If  $y \geq 0$ , but of a different type, the highest type is a larger number. Finally, if  $y \leq 0$ , then  $x > y$ , unless both are zero.

```

18523 \cs_new:Npn __fp_compare_back_any:ww #1#2; #3
18524 {
18525 __fp_if_type_fp:NTwFw
18526 #1 { __fp_if_type_fp:NTwFw #3 \use_i:nn \s__fp \use_ii:nn \s__fp_stop }
18527 \s__fp \use_ii:nn \s__fp_stop
18528 __fp_compare_back:ww
18529 {
18530 \cs:w
18531 __fp
18532 __fp_type_from_scan:N #1
18533 _compare_back
18534 __fp_type_from_scan:N #3
18535 :ww
18536 \cs_end:
18537 }
18538 #1#2 ; #3
18539 }
18540 \cs_new:Npn __fp_compare_back:ww
18541 \s__fp __fp_chk:w #1 #2 #3;
18542 \s__fp __fp_chk:w #4 #5 #6;
18543 {
18544 \int_value:w
18545 \if_meaning:w 3 #1 \exp_after:wN __fp_compare_nan:w \fi:
18546 \if_meaning:w 3 #4 \exp_after:wN __fp_compare_nan:w \fi:
18547 \if_meaning:w 2 #5 - \fi:
18548 \if_meaning:w #2 #5
18549 \if_meaning:w #1 #4
18550 \if_meaning:w 1 #1
18551 __fp_compare_npos:nwnw #6; #3;
18552 \else:
18553 0
18554 \fi:
18555 \else:
18556 \if_int_compare:w #4 < #1 - \fi: 1
18557 \fi:
18558 \else:
18559 \if_int_compare:w #1#4 = 0 \exp_stop_f:
18560 0
18561 \else:
18562 1
18563 \fi:
18564 \exp_stop_f:
18565 }
18566 \exp_stop_f:
18567 \cs_new:Npn __fp_compare_nan:w #1 \fi: \exp_stop_f: { 2 \exp_stop_f: }

(End definition for __fp_compare_back_any:ww, __fp_compare_back:ww, and __fp_compare_nan:w.)
```

`\__fp_compare_back_tuple:ww`  
`\__fp_tuple_compare_back:ww`  
`\__fp_tuple_compare_back_tuple:ww`  
`\__fp_tuple_compare_back_loop:w`

Tuple and floating point numbers are not comparable so return 2 in mixed cases or when tuples have a different number of items. Otherwise compare pairs of items with `\__fp_compare_back_any:ww` and if any don't match return 2 (as `\int_value:w 02 \exp_stop_f:`).

```

18568 \cs_new:Npn __fp_compare_back_tuple:ww #1; #2; { 2 }
18569 \cs_new:Npn __fp_tuple_compare_back:ww #1; #2; { 2 }
18570 \cs_new:Npn __fp_tuple_compare_back_tuple:ww
18571 \s__fp_tuple __fp_tuple_chk:w #1;
18572 \s__fp_tuple __fp_tuple_chk:w #2;
18573 {
18574 \int_compare:nNnTF { __fp_array_count:n {#1} } =
18575 { __fp_array_count:n {#2} }
18576 {
18577 \int_value:w 0
18578 __fp_tuple_compare_back_loop:w
18579 #1 { \s__fp \prg_break: } ; @
18580 #2 { \s__fp \prg_break: } ;
18581 \prg_break_point:
18582 \exp_stop_f:
18583 }
18584 { 2 }
18585 }
18586 \cs_new:Npn __fp_tuple_compare_back_loop:w #1#2 ; #3 @ #4#5 ;
18587 {
18588 \use_none:n #1
18589 \use_none:n #4
18590 \if_int_compare:w
18591 __fp_compare_back_any:ww #1 #2 ; #4 #5 ; = 0 \exp_stop_f:
18592 \else:
18593 2 \exp_after:wN \prg_break:
18594 \fi:
18595 __fp_tuple_compare_back_loop:w #3 @
18596 }

```

(End definition for `\__fp_compare_back_tuple:ww` and others.)

`\__fp_compare_npos:nwnw` {`<expo1>`} `<body1>` ; {`<expo2>`} `<body2>` ;  
`\__fp_compare_significand:nnnnnnnn`

Within an `\int_value:w ... \exp_stop_f:` construction, this expands to 0 if the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```

18597 \cs_new:Npn __fp_compare_npos:nwnw #1#2; #3#4;
18598 {
18599 \if_int_compare:w #1 = #3 \exp_stop_f:
18600 __fp_compare_significand:nnnnnnnn #2 #4
18601 \else:
18602 \if_int_compare:w #1 < #3 - \fi: 1
18603 \fi:
18604 }
18605 \cs_new:Npn __fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
18606 {
18607 \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
18608 \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
18609 0
18610 \else:
18611 \if_int_compare:w #3#4 < #7#8 - \fi: 1

```

```

18612 \fi:
18613 \else:
18614 \if_int_compare:w #1#2 < #5#6 - \fi: 1
18615 \fi:
18616 }

```

(End definition for `\_fp_compare_npos:nwnw` and `\_fp_compare_significand:nnnnnnnn`.)

### 30.4 Floating point expression loops

`\fp_do_until:nn`  
`\fp_do_while:nn`  
`\fp_until_do:nn`  
`\fp_while_do:nn`

These are quite easy given the above functions. The `do_until` and `do_while` versions execute the body, then test. The `until_do` and `while_do` do it the other way round.

```

18617 \cs_new:Npn \fp_do_until:nn #1#2
18618 {
18619 #2
18620 \fp_compare:nF {#1}
18621 { \fp_do_until:nn {#1} {#2} }
18622 }
18623 \cs_new:Npn \fp_do_while:nn #1#2
18624 {
18625 #2
18626 \fp_compare:nT {#1}
18627 { \fp_do_while:nn {#1} {#2} }
18628 }
18629 \cs_new:Npn \fp_until_do:nn #1#2
18630 {
18631 \fp_compare:nF {#1}
18632 {
18633 #2
18634 \fp_until_do:nn {#1} {#2}
18635 }
18636 }
18637 \cs_new:Npn \fp_while_do:nn #1#2
18638 {
18639 \fp_compare:nT {#1}
18640 {
18641 #2
18642 \fp_while_do:nn {#1} {#2}
18643 }
18644 }

```

(End definition for `\fp_do_until:nn` and others. These functions are documented on page 206.)

`\fp_do_until:nNnn`  
`\fp_do_while:nNnn`  
`\fp_until_do:nNnn`  
`\fp_while_do:nNnn`

As above but not using the `nNn` syntax.

```

18645 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
18646 {
18647 #4
18648 \fp_compare:nNnF {#1} #2 {#3}
18649 { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
18650 }
18651 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
18652 {
18653 #4
18654 \fp_compare:nNnT {#1} #2 {#3}

```

```

18655 { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
18656 }
18657 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
18658 {
18659 \fp_compare:nNnF {#1} #2 {#3}
18660 {
18661 #4
18662 \fp_until_do:nNnn {#1} #2 {#3} {#4}
18663 }
18664 }
18665 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
18666 {
18667 \fp_compare:nNnT {#1} #2 {#3}
18668 {
18669 #4
18670 \fp_while_do:nNnn {#1} #2 {#3} {#4}
18671 }
18672 }

```

(End definition for `\fp_do_until:nNnn` and others. These functions are documented on page 205.)

### `\fp_step_function:nnnN`

### `\fp_step_function:nnnc`

### `\_fp_step:wwwN`

### `\_fp_step_fp:wwwN`

### `\_fp_step:NnnnnN`

### `\_fp_step:NfnnnN`

The approach here is somewhat similar to `\int_step_function:nnnN`. There are two subtleties: we use the internal parser `\_fp_parse:n` to avoid converting back and forth from the internal representation; and (due to rounding) even a non-zero step does not guarantee that the loop counter increases.

```

18673 \cs_new:Npn \fp_step_function:nnnN #1#2#3
18674 {
18675 \exp_after:wN _fp_step:wwwN
18676 \exp:w \exp_end_continue_f:w _fp_parse_o:n {#1}
18677 \exp:w \exp_end_continue_f:w _fp_parse_o:n {#2}
18678 \exp:w \exp_end_continue_f:w _fp_parse:n {#3}
18679 }
18680 \cs_generate_variant:Nn \fp_step_function:nnnN { nnnc }
18681 % \end{macrocode}
18682 % Only floating point numbers (not tuples) are allowed arguments.
18683 % Only \enquote{normal} floating points (not ± 0,
18684 % $\pm\texttt{inf}$, \texttt{nan}) can be used as step; if positive,
18685 % call \cs{_fp_step:NnnnnN} with argument |>| otherwise |<|. This
18686 % function has one more argument than its integer counterpart, namely
18687 % the previous value, to catch the case where the loop has made no
18688 % progress. Conversion to decimal is done just before calling the
18689 % user's function.
18690 % \begin{macrocode}
18691 \cs_new:Npn _fp_step:wwwN #1#2; #3#4; #5#6; #7
18692 {
18693 _fp_if_type_fp:NTwFw #1 { } \s_fp \prg_break: \s_fp_stop
18694 _fp_if_type_fp:NTwFw #3 { } \s_fp \prg_break: \s_fp_stop
18695 _fp_if_type_fp:NTwFw #5 { } \s_fp \prg_break: \s_fp_stop
18696 \use_i:nnnn { _fp_step_fp:wwwN #1#2; #3#4; #5#6; #7 }
18697 \prg_break_point:
18698 \use:n
18699 {
18700 _fp_error:nfff { fp-step-tuple } { \fp_to_tl:n { #1#2 ; } }
18701 { \fp_to_tl:n { #3#4 ; } } { \fp_to_tl:n { #5#6 ; } }

```

```

18702 }
18703 }
18704 \cs_new:Npn __fp_step_fp:wwwN #1 ; \s__fp __fp_chk:w #2#3#4 ; #5; #6
18705 {
18706 \token_if_eq_meaning:NNTF #2 1
18707 {
18708 \token_if_eq_meaning:NNTF #3 0
18709 { __fp_step:NnnnnN > }
18710 { __fp_step:NnnnnN < }
18711 }
18712 {
18713 \token_if_eq_meaning:NNTF #2 0
18714 {
18715 __kernel_msg_expandable_error:nnn { kernel }
18716 { zero-step } {#6}
18717 }
18718 {
18719 __fp_error:nfn { fp-bad-step } { }
18720 { \fp_to_tl:n { \s__fp __fp_chk:w #2#3#4 ; } } {#6}
18721 }
18722 \use_none:nnnn
18723 }
18724 { #1 ; } { \c_nan_fp } { \s__fp __fp_chk:w #2#3#4 ; } { #5 ; } #6
18725 }
18726 \cs_new:Npn __fp_step:NnnnnN #1#2#3#4#5#6
18727 {
18728 \fp_compare:nNnTF {#2} = {#3}
18729 {
18730 __fp_error:nffn { fp-tiny-step }
18731 { \fp_to_tl:n {#3} } { \fp_to_tl:n {#4} } {#6}
18732 }
18733 {
18734 \fp_compare:nNnF {#2} #1 {#5}
18735 {
18736 \exp_args:Nf #6 { \fp_to_decimal_dispatch:w #2 }
18737 __fp_step:NfnnnN
18738 #1 { \fp_parse:n { #2 + #4 } } {#2} {#4} {#5} #6
18739 }
18740 }
18741 }
18742 \cs_generate_variant:Nn __fp_step:NnnnnN { Nf }
```

(End definition for `\fp_step_function:nnnN` and others. This function is documented on page 207.)

`\fp_step_inline:nnnn`  
`\fp_step_variable:nnnN`

As for `\int_step_inline:nnnn`, create a global function and apply it, following up with a break point.

```

18743 \cs_new_protected:Npn \fp_step_inline:nnnn
18744 {
18745 \int_gincr:N \g__kernel_prg_map_int
18746 \exp_args:NNc __fp_step:NNnnnn
18747 \cs_gset_protected:Npn
18748 { __fp_map_ \int_use:N \g__kernel_prg_map_int :w }
18749 }
18750 \cs_new_protected:Npn \fp_step_variable:nnnN #1#2#3#4#5
```

```

18751 {
18752 \int_gincr:N \g__kernel_prg_map_int
18753 \exp_args:NNo __fp_step:NNnnnn
18754 \cs_gset_protected:Npx
18755 { __fp_map_ \int_use:N \g__kernel_prg_map_int :w }
18756 {#1} {#2} {#3}
18757 {
18758 \tl_set:Nn \exp_not:N #4 {##1}
18759 \exp_not:n {#5}
18760 }
18761 }
18762 \cs_new_protected:Npn __fp_step:NNnnnn #1#2#3#4#5#6
18763 {
18764 #1 #2 ##1 {#6}
18765 \fp_step_function:nnnN {#3} {#4} {#5} #2
18766 \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
18767 }

```

(End definition for `\fp_step_inline:nnnn`, `\fp_step_variable:nnnNn`, and `\__fp_step:NNnnnn`. These functions are documented on page 207.)

```

18768 __kernel_msg_new:nnn { kernel } { fp-step-tuple }
18769 { Tuple~argument~in~fp_step_...~{#1}{#2}{#3}. }
18770 __kernel_msg_new:nnn { kernel } { fp-bad-step }
18771 { Invalid~step-size~#2~in~step-function~#3. }
18772 __kernel_msg_new:nnn { kernel } { fp-tiny-step }
18773 { Tiny~step-size~(#1+#2=#1)~in~step-function~#3. }

```

### 30.5 Extrema

`\__fp_minmax_o:Nw`  
`\__fp_minmax_aux_o:Nw`

First check all operands are floating point numbers. The argument #1 is 2 to find the maximum of an array #2 of floating point numbers, and 0 to find the minimum. We read numbers sequentially, keeping track of the largest (smallest) number found so far. If numbers are equal (for instance  $\pm 0$ ), the first is kept. We append  $-\infty$  ( $\infty$ ), for the case of an empty array. Since no number is smaller (larger) than that, this additional item only affects the maximum (minimum) in the case of `max()` and `min()` with no argument. The weird fp-like trailing marker breaks the loop correctly: see the precise definition of `\__fp_minmax_loop:Nww`.

```

18774 \cs_new:Npn __fp_minmax_o:Nw #1
18775 {
18776 __fp_parse_function_all_fp_o:fnw
18777 { \token_if_eq_meaning:NNTF 0 #1 { min } { max } }
18778 { __fp_minmax_aux_o:Nw #1 }
18779 }
18780 \cs_new:Npn __fp_minmax_aux_o:Nw #1#2 @
18781 {
18782 \if_meaning:w 0 #1
18783 \exp_after:wN __fp_minmax_loop:Nww \exp_after:wN +
18784 \else:
18785 \exp_after:wN __fp_minmax_loop:Nww \exp_after:wN -
18786 \fi:
18787 #2
18788 \s__fp __fp_chk:w 2 #1 \s__fp_exact ;
18789 \s__fp __fp_chk:w { 3 __fp_minmax_break_o:w } ;

```

```
18790 }
```

(End definition for `\_fp_minmax_o:Nw` and `\_fp_minmax_aux_o:Nw`.)

```
_fp_minmax_loop:Nww
```

The first argument is `-` or `+` to denote the case where the currently largest (smallest) number found (first floating point argument) should be replaced by the new number (second floating point argument). If the new number is `nan`, keep that as the extremum, unless that extremum is already a `nan`. Otherwise, compare the two numbers. If the new number is larger (in the case of `max`) or smaller (in the case of `min`), the test yields `true`, and we keep the second number as a new maximum; otherwise we keep the first number. Then loop.

```
18791 \cs_new:Npn _fp_minmax_loop:Nww
18792 #1 \s_fp _fp_chk:w #2#3; \s_fp _fp_chk:w #4#5;
18793 {
18794 \if_meaning:w 3 #4
18795 \if_meaning:w 3 #2
18796 _fp_minmax_auxi:ww
18797 \else:
18798 _fp_minmax_auxii:ww
18799 \fi:
18800 \else:
18801 \if_int_compare:w
18802 _fp_compare_back:ww
18803 \s_fp _fp_chk:w #4#5;
18804 \s_fp _fp_chk:w #2#3;
18805 = #1 1 \exp_stop_f:
18806 _fp_minmax_auxii:ww
18807 \else:
18808 _fp_minmax_auxi:ww
18809 \fi:
18810 \fi:
18811 _fp_minmax_loop:Nww #1
18812 \s_fp _fp_chk:w #2#3;
18813 \s_fp _fp_chk:w #4#5;
18814 }
```

(End definition for `\_fp_minmax_loop:Nww`.)

```
_fp_minmax_auxi:ww
```

```
_fp_minmax_auxii:ww
```

Keep the first/second number, and remove the other.

```
18815 \cs_new:Npn _fp_minmax_auxi:ww #1 \fi: \fi: #2 \s_fp #3 ; \s_fp #4;
18816 { \fi: \fi: #2 \s_fp #3 ; }
18817 \cs_new:Npn _fp_minmax_auxii:ww #1 \fi: \fi: #2 \s_fp #3 ;
18818 { \fi: \fi: #2 }
```

(End definition for `\_fp_minmax_auxi:ww` and `\_fp_minmax_auxii:ww`.)

```
_fp_minmax_break_o:w
```

This function is called from within an `\if_meaning:w` test. Skip to the end of the tests, close the current test with `\fi:`, clean up, and return the appropriate number with one post-expansion.

```
18819 \cs_new:Npn _fp_minmax_break_o:w #1 \fi: \fi: #2 \s_fp #3; #4;
18820 { \fi: _fp_exp_after_o:w \s_fp #3; }
```

(End definition for `\_fp_minmax_break_o:w`.)

## 30.6 Boolean operations

`\_fp_not_o:w` Return `true` or `false`, with two expansions, one to exit the conditional, and one to please `l3fp-parse`. The first argument is provided by `l3fp-parse` and is ignored.

```

18821 \cs_new:Npn _fp_not_o:w #1 \s_fp _fp_chk:w #2#3; 0
18822 {
18823 \if_meaning:w 0 #2
18824 \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
18825 \else:
18826 \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
18827 \fi:
18828 }
18829 \cs_new:Npn _fp_tuple_not_o:w #1 @ { \exp_after:wN \c_zero_fp }
```

(End definition for `\_fp_not_o:w` and `\_fp_tuple_not_o:w`.)

`\_fp_&_o:ww` For `and`, if the first number is zero, return it (with the same sign). Otherwise, return the second one. For `or`, the logic is reversed: if the first number is non-zero, return it, otherwise return the second number: we achieve that by hi-jacking `\_fp_&_o:ww`, inserting an extra argument, `\else:`, before `\s_fp`. In all cases, expand after the floating point number.

```

18830 \group_begin:
18831 \char_set_catcode_letter:N &
18832 \char_set_catcode_letter:N |
18833 \cs_new:Npn _fp_&_o:ww #1 \s_fp _fp_chk:w #2#3;
18834 {
18835 \if_meaning:w 0 #2 #1
18836 _fp_and_return:wNw \s_fp _fp_chk:w #2#3;
18837 \fi:
18838 _fp_exp_after_o:w
18839 }
18840 \cs_new:Npn _fp_&_tuple_o:ww #1 \s_fp _fp_chk:w #2#3;
18841 {
18842 \if_meaning:w 0 #2 #1
18843 _fp_and_return:wNw \s_fp _fp_chk:w #2#3;
18844 \fi:
18845 _fp_exp_after_tuple_o:w
18846 }
18847 \cs_new:Npn _fp_tuple_&_o:ww #1; { _fp_exp_after_o:w }
18848 \cs_new:Npn _fp_tuple_&_tuple_o:ww #1; { _fp_exp_after_tuple_o:w }
18849 \cs_new:Npn _fp_|_o:ww { _fp_&_o:ww \else: }
18850 \cs_new:Npn _fp_|_tuple_o:ww { _fp_&_tuple_o:ww \else: }
18851 \cs_new:Npn _fp_tuple_|_o:ww #1; #2; { _fp_exp_after_tuple_o:w #1; }
18852 \cs_new:Npn _fp_tuple_|_tuple_o:ww #1; #2;
18853 { _fp_exp_after_tuple_o:w #1; }
18854 \group_end:
18855 \cs_new:Npn _fp_and_return:wNw #1; \fi: #2;
18856 { \fi: _fp_exp_after_o:w #1; }
```

(End definition for `\_fp_&_o:ww` and others.)

## 30.7 Ternary operator

\\_\\_fp\\_ternary:NwwN  
\\_\\_fp\\_ternary\\_auxi:NwwN  
\\_\\_fp\\_ternary\\_auxii:NwwN

The first function receives the test and the true branch of the ?: ternary operator. It calls \\_\\_fp\\_ternary\\_auxii:NwwN if the test branch is a floating point number  $\pm 0$ , and otherwise calls \\_\\_fp\\_ternary\\_auxi:NwwN. These functions select one of their two arguments.

```

18857 \cs_new:Npn __fp_ternary:NwwN #1 #2#3@ #4@ #5
18858 {
18859 \if_meaning:w __fp_parse_infix_::N #5
18860 \if_charcode:w 0
18861 __fp_if_type_fp:NTwFw
18862 #2 { \use_i:nn __fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
18863 \s__fp 1 \s__fp_stop
18864 \exp_after:wN \exp_after:wN \exp_after:wN __fp_ternary_auxii:NwwN
18865 \else:
18866 \exp_after:wN \exp_after:wN \exp_after:wN __fp_ternary_auxi:NwwN
18867 \fi:
18868 \exp_after:wN #1
18869 \exp:w \exp_end_continue_f:w
18870 __fp_exp_after_array_f:w #4 \s__fp_expr_stop
18871 \exp_after:wN @
18872 \exp:w
18873 __fp_parse_operand:Nw \c__fp_prec_colon_int
18874 __fp_parse_expand:w
18875 \else:
18876 __kernel_msg_expandable_error:nnnn
18877 { kernel } { fp-missing } { : } { ~for~?: }
18878 \exp_after:wN __fp_parse_continue:NwwN
18879 \exp_after:wN #1
18880 \exp:w \exp_end_continue_f:w
18881 __fp_exp_after_array_f:w #4 \s__fp_expr_stop
18882 \exp_after:wN #5
18883 \exp_after:wN #1
18884 \fi:
18885 }
18886 \cs_new:Npn __fp_ternary_auxi:NwwN #1#2@#3@#4
18887 {
18888 \exp_after:wN __fp_parse_continue:NwN
18889 \exp_after:wN #1
18890 \exp:w \exp_end_continue_f:w
18891 __fp_exp_after_array_f:w #2 \s__fp_expr_stop
18892 #4 #1
18893 }
18894 \cs_new:Npn __fp_ternary_auxii:NwwN #1#2@#3@#4
18895 {
18896 \exp_after:wN __fp_parse_continue:NwN
18897 \exp_after:wN #1
18898 \exp:w \exp_end_continue_f:w
18899 __fp_exp_after_array_f:w #3 \s__fp_expr_stop
18900 #4 #1
18901 }
```

(End definition for \\_\\_fp\\_ternary:NwwN, \\_\\_fp\\_ternary\\_auxi:NwwN, and \\_\\_fp\\_ternary\\_auxii:NwwN.)  
18902

## 31 l3fp-basics Implementation

```
18903 <*package>
18904 <@=fp>
```

The l3fp-basics module implements addition, subtraction, multiplication, and division of two floating points, and the absolute value and sign-changing operations on one floating point. All operations implemented in this module yield the outcome of rounding the infinitely precise result of the operation to the nearest floating point.

Some algorithms used below end up being quite similar to some described in “What Every Computer Scientist Should Know About Floating Point Arithmetic”, by David Goldberg, which can be found at <http://cr.yp.to/2005-590/goldberg.pdf>.

Unary functions.

```
18905 \cs_new:Npn _fp_parse_word_abs:N
18906 { _fp_parse_unary_function:NNN _fp_set_sign_o:w 0 }
18907 \cs_new:Npn _fp_parse_word_logb:N
18908 { _fp_parse_unary_function:NNN _fp_logb_o:w ? }
18909 \cs_new:Npn _fp_parse_word_sign:N
18910 { _fp_parse_unary_function:NNN _fp_sign_o:w ? }
18911 \cs_new:Npn _fp_parse_word_sqrt:N
18912 { _fp_parse_unary_function:NNN _fp_sqrt_o:w ? }
```

(End definition for `\_fp_parse_word_abs:N` and others.)

### 31.1 Addition and subtraction

We define here two functions, `\_fp_-o:ww` and `\_fp+_o:ww`, which perform the subtraction and addition of their two floating point operands, and expand the tokens following the result once.

A more obscure function, `\_fp_add_big_i_o:wNww`, is used in l3fp-expo.

The logic goes as follows:

- `\_fp_-o:ww` calls `\_fp+_o:ww` to do the work, with the sign of the second operand flipped;
- `\_fp+_o:ww` dispatches depending on the type of floating point, calling specialized auxiliaries;
- in all cases except summing two normal floating point numbers, we return one or the other operands depending on the signs, or detect an invalid operation in the case of  $\infty - \infty$ ;
- for normal floating point numbers, compare the signs;
- to add two floating point numbers of the same sign or of opposite signs, shift the significand of the smaller one to match the bigger one, perform the addition or subtraction of significands, check for a carry, round, and pack using the `\_fp-basics_pack_...` functions.

The trickiest part is to round correctly when adding or subtracting normal floating point numbers.

### 31.1.1 Sign, exponent, and special numbers

\\_\\_fp\\_o:ww

The \\_\\_fp\\_o:ww auxiliary has a hook: it takes one argument between the first \s\\_\\_fp and \\_\\_fp\\_chk:w, which is applied to the sign of the second operand. Positioning the hook there means that \\_\\_fp\\_o:ww can still perform the sanity check that it was followed by \s\\_\\_fp.

```

18913 \cs_new:cpx { __fp_o:ww } \s_fp
18914 {
18915 \exp_not:c { __fp_o:ww }
18916 \exp_not:n { \s_fp __fp_neg_sign:N }
18917 }
```

(End definition for \\_\\_fp\\_o:ww.)

\\_\\_fp\\_o:ww

This function is either called directly with an empty #1 to compute an addition, or it is called by \\_\\_fp\\_o:ww with \\_\\_fp\\_neg\\_sign:N as #1 to compute a subtraction, in which case the second operand's sign should be changed. If the *types* #2 and #4 are the same, dispatch to case #2 (0, 1, 2, or 3), where we call specialized functions: thanks to \int\\_value:w, those receive the tweaked *sign<sub>2</sub>* (expansion of #1#5) as an argument. If the *types* are distinct, the result is simply the floating point number with the highest *type*. Since case 3 (used for two nan) also picks the first operand, we can also use it when *type<sub>1</sub>* is greater than *type<sub>2</sub>*. Also note that we don't need to worry about *sign<sub>2</sub>* in that case since the second operand is discarded.

```

18918 \cs_new:cpn { __fp_o:ww }
18919 \s_fp #1 __fp_chk:w #2 #3 ; \s_fp __fp_chk:w #4 #5
18920 {
18921 \if_case:w
18922 \if_meaning:w #2 #4
18923 #2
18924 \else:
18925 \if_int_compare:w #2 > #4 \exp_stop_f:
18926 3
18927 \else:
18928 4
18929 \fi:
18930 \fi:
18931 \exp_stop_f:
18932 \exp_after:wN __fp_add_zeros_o:Nww \int_value:w
18933 \or: \exp_after:wN __fp_add_normal_o:Nww \int_value:w
18934 \or: \exp_after:wN __fp_add_inf_o:Nww \int_value:w
18935 \or: __fp_case_return_i_o:ww
18936 \else: \exp_after:wN __fp_add_return_ii_o:Nww \int_value:w
18937 \fi:
18938 #1 #5
18939 \s_fp __fp_chk:w #2 #3 ;
18940 \s_fp __fp_chk:w #4 #5
18941 }
```

(End definition for \\_\\_fp\\_o:ww.)

\\_\\_fp\\_add\\_return\\_ii\\_o:Nww

Ignore the first operand, and return the second, but using the sign #1 rather than #4. As usual, expand after the floating point.

```

18942 \cs_new:Npn __fp_add_return_ii_o:Nww #1 #2 ; \s_fp __fp_chk:w #3 #4
18943 { __fp_exp_after_o:w \s_fp __fp_chk:w #3 #1 }
```

(End definition for `\_fp_add_return_i_o:Nww`.)

`\_fp_add_zeros_o:Nww` Adding two zeros yields `\c_zero_fp`, except if both zeros were  $-0$ .

```
18944 \cs_new:Npn _fp_add_zeros_o:Nww #1 \s_fp _fp_chk:w 0 #2
18945 {
18946 \if_int_compare:w #2 #1 = 20 \exp_stop_f:
18947 \exp_after:wN _fp_add_return_i_o:Nww
18948 \else:
18949 _fp_case_return_i_o:ww
18950 \fi:
18951 #1
18952 \s_fp _fp_chk:w 0 #2
18953 }
```

(End definition for `\_fp_add_zeros_o:Nww`.)

`\_fp_add_inf_o:Nww` If both infinities have the same sign, just return that infinity, otherwise, it is an invalid operation. We find out if that invalid operation is an addition or a subtraction by testing whether the tweaked  $\langle sign_2 \rangle$  (#1) and the  $\langle sign_2 \rangle$  (#4) are identical.

```
18954 \cs_new:Npn _fp_add_inf_o:Nww
18955 #1 \s_fp _fp_chk:w 2 #2 #3; \s_fp _fp_chk:w 2 #4
18956 {
18957 \if_meaning:w #1 #2
18958 _fp_case_return_i_o:ww
18959 \else:
18960 _fp_case_use:nw
18961 {
18962 \exp_last_unbraced:Nf _fp_invalid_operation_o:Nww
18963 { \token_if_eq_meaning:NNTF #1 #4 + - }
18964 }
18965 \fi:
18966 \s_fp _fp_chk:w 2 #2 #3;
18967 \s_fp _fp_chk:w 2 #4
18968 }
```

(End definition for `\_fp_add_inf_o:Nww`.)

`\_fp_add_normal_o:Nww`  $\_fp_add_normal_o:Nww \langle sign_2 \rangle \s_fp \_fp_chk:w 1 \langle sign_1 \rangle \langle exp_1 \rangle \langle body_1 \rangle ; \s_fp \_fp_chk:w 1 \langle initial sign_2 \rangle \langle exp_2 \rangle \langle body_2 \rangle ;$

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```
18969 \cs_new:Npn _fp_add_normal_o:Nww #1 \s_fp _fp_chk:w 1 #2
18970 {
18971 \if_meaning:w #1#2
18972 \exp_after:wN _fp_add_npos_o:NnwNnw
18973 \else:
18974 \exp_after:wN _fp_sub_npos_o:NnwNnw
18975 \fi:
18976 #2
18977 }
```

(End definition for `\_fp_add_normal_o:Nww`.)

### 31.1.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

```
__fp_add_npos_o:NnwNnw __fp_add_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s__fp __fp_chk:w 1
<initial sign2> <exp2> <body2>
```

Since we are doing an addition, the final sign is  $\langle sign_1 \rangle$ . Start an  $\_\_fp\_int\_eval:w$ , responsible for computing the exponent: the result, and the  $\langle final sign \rangle$  are then given to  $\_\_fp\_sanitize:Nw$  which checks for overflow. The exponent is computed as the largest exponent #2 or #5, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by  $\_\_fp\_add\_big\_i:wNww$  or  $\_\_fp\_add\_big\_ii:wNww$ . We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```
18978 \cs_new:Npn __fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp __fp_chk:w 1 #4 #5
18979 {
18980 \exp_after:wN __fp_sanitize:Nw
18981 \exp_after:wN #1
18982 \int_value:w __fp_int_eval:w
18983 \if_int_compare:w #2 > #5 \exp_stop_f:
18984 #2
18985 \exp_after:wN __fp_add_big_i_o:wNww \int_value:w -
18986 \else:
18987 #5
18988 \exp_after:wN __fp_add_big_ii_o:wNww \int_value:w
18989 \fi:
18990 __fp_int_eval:w #5 - #2 ; #1 #3;
18991 }
```

(End definition for  $\_\_fp\_add\_npos\_o:NnwNnw$ .)

```
__fp_add_big_i_o:wNww <shift> ; <final sign> <body1> ; <body2> ;
```

Used in  $\text{l3fp-expo}$ . Shift the significand of the small number, then add with  $\_\_fp\_add\_significand\_o:NnnwnnnnN$ .

```
18992 \cs_new:Npn __fp_add_big_i_o:wNww #1; #2 #3; #4;
18993 {
18994 __fp_decimate:nNnnnn {#1}
18995 __fp_add_significand_o:NnnwnnnnN
18996 #4
18997 #3
18998 #2
18999 }
19000 \cs_new:Npn __fp_add_big_ii_o:wNww #1; #2 #3; #4;
19001 {
19002 __fp_decimate:nNnnnn {#1}
19003 __fp_add_significand_o:NnnwnnnnN
19004 #3
19005 #4
19006 #2
19007 }
```

(End definition for  $\_\_fp\_add\_big\_i\_o:wNww$  and  $\_\_fp\_add\_big\_ii\_o:wNww$ .)

```
__fp_add_significand_o:NnnwnnnN
__fp_add_significand_pack:NNNNNNN
__fp_add_significand_test_o:N
```

To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as  $0.99 \dots 95 \rightarrow 1.00 \dots 0$ , but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```
19008 \cs_new:Npn __fp_add_significand_o:NnnwnnnN #1 #2#3 #4; #5#6#7#8
19009 {
19010 \exp_after:wN __fp_add_significand_test_o:N
19011 \int_value:w __fp_int_eval:w 1#5#6 + #2
19012 \exp_after:wN __fp_add_significand_pack:NNNNNNN
19013 \int_value:w __fp_int_eval:w 1#7#8 + #3 ; #1
19014 }
19015 \cs_new:Npn __fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
19016 {
19017 \if_meaning:w 2 #1
19018 + 1
19019 \fi:
19020 ; #2 #3 #4 #5 #6 #7 ;
19021 }
19022 \cs_new:Npn __fp_add_significand_test_o:N #1
19023 {
19024 \if_meaning:w 2 #1
19025 \exp_after:wN __fp_add_significand_carry_o:wwwNN
19026 \else:
19027 \exp_after:wN __fp_add_significand_no_carry_o:wwwNN
19028 \fi:
19029 }
```

(End definition for  $\_\_fp\_add\_significand\_o:NnnwnnnN$ ,  $\_\_fp\_add\_significand\_pack:NNNNNNN$ , and  $\_\_fp\_add\_significand\_test\_o:N$ .)

```
__fp_add_significand_no_carry_o:wwwNN
__fp_add_significand_no_carry_o:wwwNN <8d> ; <6d> ; <2d> ; <rounding
digit> <sign>
```

If there's no carry, grab all the digits again and round. The packing function  $\_\_fp\_basics\_pack\_high:NNNNNw$  takes care of the case where rounding brings a carry.

```
19030 \cs_new:Npn __fp_add_significand_no_carry_o:wwwNN
19031 #1; #2; #3#4 ; #5#6
19032 {
19033 \exp_after:wN __fp_basics_pack_high:NNNNNw
19034 \int_value:w __fp_int_eval:w 1 #1
19035 \exp_after:wN __fp_basics_pack_low:NNNNNw
19036 \int_value:w __fp_int_eval:w 1 #2 #3#4
19037 + __fp_round:NNN #6 #4 #5
19038 \exp_after:wN ;
19039 }
```

(End definition for  $\_\_fp\_add\_significand\_no\_carry\_o:wwwNN$ .)

```
__fp_add_significand_carry_o:wwwNN <8d> ; <6d> ; <2d> ; <rounding
digit> <sign>
```

The case where there is a carry is very similar. Rounding can even raise the first digit from 1 to 2, but we don't care.

```

19040 \cs_new:Npn __fp_add_significand_carry_o:wwwNN
 #1; #2; #3#4; #5#6
19042 {
19043 + 1
19044 \exp_after:wN __fp_basics_pack_weird_high:NNNNNNNNw
19045 \int_value:w __fp_int_eval:w 1 1 #1
19046 \exp_after:wN __fp_basics_pack_weird_low:NNNNw
19047 \int_value:w __fp_int_eval:w 1 #2#3 +
19048 \exp_after:wN __fp_round:NNN
19049 \exp_after:wN #6
19050 \exp_after:wN #3
19051 \int_value:w __fp_round_digit:Nw #4 #5 ;
19052 \exp_after:wN ;
19053 }
```

(End definition for `\__fp_add_significand_carry_o:wwwNN`.)

### 31.1.3 Absolute subtraction

```

__fp_sub_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s_fp __fp_chk:w 1
__fp_sub_eq_o:NnwNnw <initial sign2> <exp2> <body2> ;
```

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call `\__fp_sub_npos_i_o:NnwNnw` with the opposite of `<sign1>`.

```

19054 \cs_new:Npn __fp_sub_npos_o:NnwNnw #1#2#3; \s_fp __fp_chk:w 1 #4#5#6;
19055 {
19056 \if_case:w __fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
19057 \exp_after:wN __fp_sub_eq_o:NnwNnw
19058 \or:
19059 \exp_after:wN __fp_sub_npos_i_o:NnwNnw
19060 \else:
19061 \exp_after:wN __fp_sub_npos_ii_o:NnwNnw
19062 \fi:
19063 #1 {#2} #3; {#5} #6;
19064 }
19065 \cs_new:Npn __fp_sub_eq_o:NnwNnw #1#2; #3; { \exp_after:wN \c_zero_fp }
19066 \cs_new:Npn __fp_sub_npos_ii_o:NnwNnw #1 #2; #3;
19067 {
19068 \exp_after:wN __fp_sub_npos_i_o:NnwNnw
19069 \int_value:w __fp_neg_sign:N #1
19070 #3; #2;
19071 }
```

(End definition for `\__fp_sub_npos_o:NnwNnw`, `\__fp_sub_eq_o:NnwNnw`, and `\__fp_sub_npos_ii_o:NnwNnw`.)

`\__fp_sub_npos_i_o:NnwNnw`

After the computation is done, `\__fp_sanitize:Nw` checks for overflow/underflow. It expects the `<final sign>` and the `<exponent>` (delimited by ;). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the `near` auxiliary. Otherwise, decimate  $y$ , then call the `far` auxiliary to evaluate

the difference between the two significands. Note that we decimate by 1 less than one could expect.

```

19072 \cs_new:Npn __fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
19073 {
19074 \exp_after:wN __fp_sanitize:Nw
19075 \exp_after:wN #1
19076 \int_value:w __fp_int_eval:w
19077 #2
19078 \if_int_compare:w #2 = #4 \exp_stop_f:
19079 \exp_after:wN __fp_sub_back_near_o:nnnnnnnnN
19080 \else:
19081 \exp_after:wN __fp_decimate:nNnnnn \exp_after:wN
19082 { \int_value:w __fp_int_eval:w #2 - #4 - 1 \exp_after:wN }
19083 \exp_after:wN __fp_sub_back_far_o:NnnwnnnnN
19084 \fi:
19085 #5
19086 #3
19087 #1
19088 }

```

(End definition for `\__fp_sub_npos_i_o:Nnwnw`.)

```

__fp_sub_back_near_o:nnnnnnnnN {\langle Y_1\rangle} {\langle Y_2\rangle} {\langle Y_3\rangle} {\langle Y_4\rangle} {\langle X_1\rangle}
__fp_sub_back_near_pack:NNNNNNw {\langle X_2\rangle} {\langle X_3\rangle} {\langle X_4\rangle} {\langle final sign\rangle}
__fp_sub_back_near_after:wNNNNNw

```

In this case, the subtraction is exact, so we discard the `{final sign}` #9. The very large shifts of  $10^9$  and  $1.1 \cdot 10^9$  are unnecessary here, but allow the auxiliaries to be reused later. Each integer expression produces a 10 digit result. If the resulting 16 digits start with a 0, then we need to shift the group, padding with trailing zeros.

```

19089 \cs_new:Npn __fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
19090 {
19091 \exp_after:wN __fp_sub_back_near_after:wNNNNNw
19092 \int_value:w __fp_int_eval:w 10#5#6 - #1#2 - 11
19093 \exp_after:wN __fp_sub_back_near_pack:NNNNNNw
19094 \int_value:w __fp_int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
19095 }
19096 \cs_new:Npn __fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
19097 { + #1#2 ; {#3#4#5#6} {#7} ; }
19098 \cs_new:Npn __fp_sub_back_near_after:wNNNNNw 10 #1#2#3#4 #5 ;
19099 {
19100 \if_meaning:w 0 #1
19101 \exp_after:wN __fp_sub_back_shift:wnnnn
19102 \fi:
19103 ; {#1#2#3#4} {#5}
19104 }

```

(End definition for `\__fp_sub_back_near_o:nnnnnnnnN`, `\__fp_sub_back_near_pack:NNNNNNw`, and `\__fp_sub_back_near_after:wNNNNNw`.)

```

__fp_sub_back_shift:wnnnn ; {\langle Z_1\rangle} {\langle Z_2\rangle} {\langle Z_3\rangle} {\langle Z_4\rangle} ;

```

This function is called with  $\langle Z_1 \rangle \leq 999$ . Act with `\number` to trim leading zeros from  $\langle Z_1 \rangle \langle Z_2 \rangle$  (we don't do all four blocks at once, since non-zero blocks would then overflow TeX's integers). If the first two blocks are zero, the auxiliary receives an empty #1 and trims #2#30 from leading zeros, yielding a total shift between 7 and 16 to the exponent. Otherwise we get the shift from #1 alone, yielding a result between 1 and 6. Once the

```

__fp_sub_back_shift:wnnnn
__fp_sub_back_shift_ii:ww
__fp_sub_back_shift_iii:NNNNNNNNw
__fp_sub_back_shift_iv:nnnnw

```

exponent is taken care of, trim leading zeros from #1#2#3 (when #1 is empty, the space before #2#3 is ignored), get four blocks of 4 digits and finally clean up. Trailing zeros are added so that digits can be grabbed safely.

```

19105 \cs_new:Npn __fp_sub_back_shift:wnnnn ; #1#2
19106 {
19107 \exp_after:wN __fp_sub_back_shift_ii:ww
19108 \int_value:w #1 #2 0 ;
19109 }
19110 \cs_new:Npn __fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
19111 {
19112 \if_meaning:w @ #1 @
19113 - 7
19114 - \exp_after:wN \use_i:nnn
19115 \exp_after:wN __fp_sub_back_shift_iii:NNNNNNNNw
19116 \int_value:w #2#3 0 ~ 123456789;
19117 \else:
19118 - __fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
19119 \fi:
19120 \exp_after:wN __fp_pack_twice_four:wNNNNNNNN
19121 \exp_after:wN __fp_pack_twice_four:wNNNNNNNN
19122 \exp_after:wN __fp_sub_back_shift_iv:nnnw
19123 \exp_after:wN ;
19124 \int_value:w
19125 #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
19126 }
19127 \cs_new:Npn __fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
19128 \cs_new:Npn __fp_sub_back_shift_iv:nnnw #1 ; #2 ; { ; #1 ; }
```

(End definition for `\__fp_sub_back_shift:wnnnn` and others.)

`\__fp_sub_back_far_o:NnnwnnnnN` (*rounding*) { $\langle Y'_1 \rangle$ } { $\langle Y'_2 \rangle$ }  
   {*extra-digits*} ; { $\langle X_1 \rangle$ } { $\langle X_2 \rangle$ } { $\langle X_3 \rangle$ } { $\langle X_4 \rangle$ } {*final sign*}

If the difference is greater than  $10^{\langle expo_x \rangle}$ , call the `very_far` auxiliary. If the result is less than  $10^{\langle expo_x \rangle}$ , call the `not_far` auxiliary. If it is too close a call to know yet, namely if  $1\langle Y'_1 \rangle \langle Y'_2 \rangle = \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle 0$ , then call the `quite_far` auxiliary. We use the odd combination of space and semi-colon delimiters to allow the `not_far` auxiliary to grab each piece individually, the `very_far` auxiliary to use `\__fp_pack_eight:wNNNNNNNN`, and the `quite_far` to ignore the significands easily (using the ; delimiter).

```

19129 \cs_new:Npn __fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
19130 {
19131 \if_case:w
19132 \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
19133 \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
19134 0
19135 \else:
19136 \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: 1
19137 \fi:
19138 \else:
19139 \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: 1
19140 \fi:
19141 \exp_stop_f:
19142 \exp_after:wN __fp_sub_back Quite_far_o:wwNN
19143 \or: \exp_after:wN __fp_sub_back_very_far_o:wwwNN
```

```

19144 \else: \exp_after:wN _fp_sub_back_not_far_o:wwwNN
19145 \fi:
19146 #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
19147 }

```

(End definition for `\_fp_sub_back_far_o:NnnnnnnnN`.)

`\_fp_sub_back Quite far_o:wwNN  
\_fp_sub_back Quite far_ii>NN`

The easiest case is when  $x - y$  is extremely close to a power of 10, namely the first digit of  $x$  is 1, and all others vanish when subtracting  $y$ . Then the *rounding* #3 and the *final sign* #4 control whether we get 1 or 0.999999999999999. In the usual round-to-nearest mode, we get 1 whenever the *rounding* digit is less than or equal to 5 (remember that the *rounding* digit is only equal to 5 if there was no further non-zero digit).

```

19148 \cs_new:Npn _fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
19149 {
19150 \exp_after:wN _fp_sub_back_quite_far_ii>NN
19151 \exp_after:wN #3
19152 \exp_after:wN #4
19153 }
19154 \cs_new:Npn _fp_sub_back_quite_far_ii>NN #1#2
19155 {
19156 \if_case:w _fp_round_neg:NNN #2 0 #1
19157 \exp_after:wN \use_i:nn
19158 \else:
19159 \exp_after:wN \use_ii:nn
19160 \fi:
19161 { ; {1000} {0000} {0000} {0000} ; }
19162 { - 1 ; {9999} {9999} {9999} {9999} ; }
19163 }

```

(End definition for `\_fp_sub_back_quite_far_o:wwNN` and `\_fp_sub_back_quite_far_ii>NN`.)

`\_fp_sub_back_not_far_o:wwwNN`

In the present case,  $x$  and  $y$  have different exponents, but  $y$  is large enough that  $x - y$  has a smaller exponent than  $x$ . Decrement the exponent (with -1). Then proceed in a way similar to the `near` auxiliaries seen earlier, but multiplying  $x$  by 10 (#30 and #40 below), and with the added quirk that the *rounding* digit has to be taken into account. Namely, we may have to decrease the result by one unit if `\_fp_round_neg:NNN` returns 1. This function expects the *final sign* #6, the last digit of `1100000000+#40-#2`, and the *rounding* digit. Instead of redoing the computation for the second argument, we note that `\_fp_round_neg:NNN` only cares about its parity, which is identical to that of the last digit of #2.

```

19164 \cs_new:Npn _fp_sub_back_not_far_o:wwwNN #1 ~ #2; #3 ~ #4; #5#6
19165 {
19166 - 1
19167 \exp_after:wN _fp_sub_back_near_after:wNNNNw
19168 \int_value:w _fp_int_eval:w 1#30 - #1 - 11
19169 \exp_after:wN _fp_sub_back_near_pack:NNNNNNw
19170 \int_value:w _fp_int_eval:w 11 0000 0000 + #40 - #2
19171 - \exp_after:wN _fp_round_neg:NNN
19172 \exp_after:wN #6
19173 \use_none:nnnnnn #2 #5
19174 \exp_after:wN ;
19175 }

```

(End definition for `\_fp_sub_back_not_far_o:wwwNN`.)

```
__fp_sub_back_very_far_o:wwwNN
__fp_sub_back_very_far_ii_o:nnNwwNN
```

The case where  $x - y$  and  $x$  have the same exponent is a bit more tricky, mostly because it cannot reuse the same auxiliaries. Shift the  $y$  significand by adding a leading 0. Then the logic is similar to the `not_far` functions above. Rounding is a bit more complicated: we have two *(rounding)* digits #3 and #6 (from the decimation, and from the new shift) to take into account, and getting the parity of the main result requires a computation. The first `\int_value:w` triggers the second one because the number is unfinished; we can thus not use 0 in place of 2 there.

```
19176 \cs_new:Npn __fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
19177 {
19178 __fp_pack_eight:wNNNNNNNN
19179 __fp_sub_back_very_far_ii_o:nnNwwNN
19180 { 0 #1#2#3 #4#5#6#7 }
19181 ;
19182 }
19183 \cs_new:Npn __fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
19184 {
19185 \exp_after:wN __fp_basis_pack_high:NNNNNw
19186 \int_value:w __fp_int_eval:w 1#4 - #1 - 1
19187 \exp_after:wN __fp_basis_pack_low:NNNNNw
19188 \int_value:w __fp_int_eval:w 2#5 - #2
19189 - \exp_after:wN __fp_round_neg:NNN
19190 \exp_after:wN #7
19191 \int_value:w
19192 \if_int_odd:w __fp_int_eval:w #5 - #2 __fp_int_eval_end:
19193 1 \else: 2 \fi:
19194 \int_value:w __fp_round_digit:Nw #3 #6 ;
19195 \exp_after:wN ;
19196 }
```

(End definition for `\_\_fp\_sub\_back\_very\_far\_o:wwwNN` and `\_\_fp\_sub\_back\_very\_far\_ii\_o:nnNwwNN`.)

## 31.2 Multiplication

### 31.2.1 Signs, and special numbers

```
__fp_*_o:ww
```

We go through an auxiliary, which is common with `\_\_fp\/_o:ww`. The first argument is the operation, used for the invalid operation exception. The second is inserted in a formula to dispatch cases slightly differently between multiplication and division. The third is the operation for normal floating points. The fourth is there for extra cases needed in `\_\_fp\/_o:ww`.

```
19197 \cs_new:cpn { __fp_*_o:ww }
19198 {
19199 __fp_mul_cases_o:NnNww
19200 *
19201 { - 2 + }
19202 __fp_mul_npos_o:Nww
19203 { }
19204 }
```

(End definition for `\_\_fp\_*\_o:ww`.)

```
__fp_mul_cases_o:nNnnww
```

Split into 10 cases (12 for division). If both numbers are normal, go to case 0 (same sign) or case 1 (opposite signs): in both cases, call `\_\_fp\_mul\_npos\_o:Nww` to do the work. If

the first operand is `nan`, go to case 2, in which the second operand is discarded; if the second operand is `nan`, go to case 3, in which the first operand is discarded (note the weird interaction with the final test on signs). Then we separate the case where the first number is normal and the second is zero: this goes to cases 4 and 5 for multiplication, 10 and 11 for division. Otherwise, we do a computation which dispatches the products  $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$  to case 4 or 5 depending on the combined sign, the products  $0 \times \infty$  and  $\infty \times 0$  to case 6 or 7 (invalid operation), and the products  $1 \times \infty = \infty \times 1 = \infty \times \infty = \infty$  to cases 8 and 9. Note that the code for these two cases (which return  $\pm\infty$ ) is inserted as argument #4, because it differs in the case of divisions.

```

19205 \cs_new:Npn __fp_mul_cases_o:NnNnww
19206 #1#2#3#4 \s__fp __fp_chk:w #5#6#7; \s__fp __fp_chk:w #8#9
19207 {
19208 \if_case:w __fp_int_eval:w
19209 \if_int_compare:w #5 #8 = 11 ~
19210 1
19211 \else:
19212 \if_meaning:w 3 #8
19213 3
19214 \else:
19215 \if_meaning:w 3 #5
19216 2
19217 \else:
19218 \if_int_compare:w #5 #8 = 10 ~
19219 9 #2 - 2
19220 \else:
19221 (#5 #2 #8) / 2 * 2 + 7
19222 \fi:
19223 \fi:
19224 \fi:
19225 \fi:
19226 \if_meaning:w #6 #9 - 1 \fi:
19227 __fp_int_eval_end:
19228 __fp_case_use:nw { #3 0 }
19229 \or: __fp_case_use:nw { #3 2 }
19230 \or: __fp_case_return_i_o:ww
19231 \or: __fp_case_return_ii_o:ww
19232 \or: __fp_case_return_o:Nww \c_zero_fp
19233 \or: __fp_case_return_o:Nww \c_minus_zero_fp
19234 \or: __fp_case_use:nw { __fp_invalid_operation_o:Nww #1 }
19235 \or: __fp_case_use:nw { __fp_invalid_operation_o:Nww #1 }
19236 \or: __fp_case_return_o:Nww \c_inf_fp
19237 \or: __fp_case_return_o:Nww \c_minus_inf_fp
19238 #4
19239 \fi:
19240 \s__fp __fp_chk:w #5 #6 #7;
19241 \s__fp __fp_chk:w #8 #9
19242 }

```

(End definition for `\__fp_mul_cases_o:nNnnww`.)

### 31.2.2 Absolute multiplication

In this subsection, we perform the multiplication of two positive normal numbers.

```
__fp_mul_npos_o:Nww __fp_mul_npos_o:Nww <final sign> \s_fp __fp_chk:w 1 <sign1> {\<exp1>} <body1> ; \s_fp __fp_chk:w 1 <sign2> {\<exp2>} <body2> ;
```

After the computation, `\_\_fp\_sanitize:Nw` checks for overflow or underflow. As we did for addition, `\_\_fp\_int\_eval:w` computes the exponent, catching any shift coming from the computation in the significand. The `<final sign>` is needed to do the rounding properly in the significand computation. We setup the post-expansion here, triggered by `\_\_fp\_mul\_significand_o:nnnnNnnnn`.

This is also used in l3fp-convert.

```
19243 \cs_new:Npn __fp_mul_npos_o:Nww
19244 #1 \s_fp __fp_chk:w #2 #3 #4 #5 ; \s_fp __fp_chk:w #6 #7 #8 #9 ;
19245 {
19246 \exp_after:wN __fp_sanitize:Nw
19247 \exp_after:wN #1
19248 \int_value:w __fp_int_eval:w
19249 #4 + #8
19250 __fp_mul_significand_o:nnnnNnnnn #5 #1 #9
19251 }
```

(End definition for `\_\_fp\_mul\_npos_o:Nww`.)

```
__fp_mul_significand_o:nnnnNnnnn
__fp_mul_significand_drop:NNNNNw
__fp_mul_significand_keep:NNNNNw
```

Note the three semicolons at the end of the definition. One is for the last `\_\_fp\_mul\_significand\_drop:NNNNNw`; one is for `\_\_fp\_round\_digit:Nw` later on; and one, preceded by `\exp_after:wN`, which is correctly expanded (within an `\_\_fp\_int\_eval:w`), is used by `\_\_fp\_basics\_pack\_low:NNNNNw`.

The product of two 16 digit integers has 31 or 32 digits, but it is impossible to know which one before computing. The place where we round depends on that number of digits, and may depend on all digits until the last in some rare cases. The approach is thus to compute the 5 first blocks of 4 digits (the first one is between 100 and 9999 inclusive), and a compact version of the remaining 3 blocks. Afterwards, the number of digits is known, and we can do the rounding within yet another set of `\_\_fp\_int\_eval:w`.

```
19252 \cs_new:Npn __fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
19253 {
19254 \exp_after:wN __fp_mul_significand_test_f:NNN
19255 \exp_after:wN #5
19256 \int_value:w __fp_int_eval:w 99990000 + #1*#6 +
19257 \exp_after:wN __fp_mul_significand_keep:NNNNNw
19258 \int_value:w __fp_int_eval:w 99990000 + #1*#7 + #2*#6 +
19259 \exp_after:wN __fp_mul_significand_keep:NNNNNw
19260 \int_value:w __fp_int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
19261 \exp_after:wN __fp_mul_significand_drop:NNNNNw
19262 \int_value:w __fp_int_eval:w 99990000 + #1*#9 + #2*#8 +
19263 #3*#7 + #4*#6 +
19264 \exp_after:wN __fp_mul_significand_drop:NNNNNw
19265 \int_value:w __fp_int_eval:w 99990000 + #2*#9 + #3*#8 +
19266 #4*#7 +
19267 \exp_after:wN __fp_mul_significand_drop:NNNNNw
19268 \int_value:w __fp_int_eval:w 99990000 + #3*#9 + #4*#8 +
19269 \exp_after:wN __fp_mul_significand_drop:NNNNNw
19270 \int_value:w __fp_int_eval:w 100000000 + #4*#9 ;
19271 ; \exp_after:wN ;
```

```

19272 }
19273 \cs_new:Npn __fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
19274 { #1#2#3#4#5 ; + #6 }
19275 \cs_new:Npn __fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
19276 { #1#2#3#4#5 ; #6 ; }

```

(End definition for  $\text{\__fp\_mul\_significand\_o:nnnnNnnnn}$ ,  $\text{\__fp\_mul\_significand\_drop:NNNNNw}$ , and  $\text{\__fp\_mul\_significand\_keep:NNNNNw}$ .)

```

__fp_mul_significand_test_f:NNN
__fp_mul_significand_test_f:NNN ⟨sign⟩ 1 ⟨digits 1–8⟩ ; ⟨digits 9–12⟩ ;
⟨digits 13–16⟩ ; + ⟨digits 17–20⟩ + ⟨digits 21–24⟩ + ⟨digits 25–28⟩ + ⟨digits
29–32⟩ ; \exp_after:wN ;

```

If the ⟨digit 1⟩ is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if ⟨digit 1⟩ is zero, we care about digits 17 and 18, and whether further digits are zero.

```

19277 \cs_new:Npn __fp_mul_significand_test_f:NNN #1 #2 #3
19278 {
19279 \if_meaning:w 0 #3
19280 \exp_after:wN __fp_mul_significand_small_f:NNwwN
19281 \else:
19282 \exp_after:wN __fp_mul_significand_large_f:NwwNNNN
19283 \fi:
19284 #1 #3
19285 }

```

(End definition for  $\text{\__fp\_mul\_significand\_test\_f:NNN}$ .)

$\text{\__fp\_mul\_significand\_large\_f:NwwNNNN}$

In this branch, ⟨digit 1⟩ is non-zero. The result is thus ⟨digits 1–16⟩, plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are zero or not. Here,  $\text{\__fp\_round\_digit:Nw}$  takes digits 17 and further (as an integer expression), and replaces it by a ⟨rounding digit⟩, suitable for  $\text{\__fp\_round:NNN}$ .

```

19286 \cs_new:Npn __fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
19287 {
19288 \exp_after:wN __fp_basics_pack_high:NNNNNw
19289 \int_value:w __fp_int_eval:w 1#2
19290 \exp_after:wN __fp_basics_pack_low:NNNNNw
19291 \int_value:w __fp_int_eval:w 1#3#4#5#6#7
19292 + \exp_after:wN __fp_round:NNN
19293 \exp_after:wN #1
19294 \exp_after:wN #7
19295 \int_value:w __fp_round_digit:Nw
19296 }

```

(End definition for  $\text{\__fp\_mul\_significand\_large\_f:NwwNNNN}$ .)

$\text{\__fp\_mul\_significand\_small\_f:NNwwN}$

In this branch, ⟨digit 1⟩ is zero. Our result is thus ⟨digits 2–17⟩, plus some rounding which depends on the digits 17, 18, and whether all subsequent digits are zero or not. The 8 digits 1#3 are followed, after expansion of the `small_pack` auxiliary, by the next digit, to form a 9 digit number.

```

19297 \cs_new:Npn __fp_mul_significand_small_f:NNwwN #1 #2#3; #4#5; #6; + #7
19298 {
19299 - 1
19300 \exp_after:wN __fp_basics_pack_high:NNNNNw
19301 \int_value:w __fp_int_eval:w 1#3#4

```

```

19302 \exp_after:wN _fp_basics_pack_low>NNNNNw
19303 \int_value:w _fp_int_eval:w 1#5#6#7
19304 + \exp_after:wN _fp_round:NNN
19305 \exp_after:wN #
19306 \exp_after:wN #
19307 \int_value:w _fp_round_digit:Nw
19308 }

```

(End definition for `\_fp_mul_significand_small_f:NNwwwN`.)

### 31.3 Division

#### 31.3.1 Signs, and special numbers

Time is now ripe to tackle the hardest of the four elementary operations: division.

`\_fp/_o:ww`

Filtering special floating point is very similar to what we did for multiplications, with a few variations. Invalid operation exceptions display `/` rather than `*`. In the formula for dispatch, we replace `- 2 +` by `-`. The case of normal numbers is treated using `\_fp_div_npos_o:Nww` rather than `\_fp_mul_npos_o:Nww`. There are two additional cases: if the first operand is normal and the second is a zero, then the division by zero exception is raised: cases 10 and 11 of the `\if_case:w` construction in `\_fp_mul_cases_o:NnNnww` are provided as the fourth argument here.

```

19309 \cs_new:cpn { _fp/_o:ww }
19310 {
19311 _fp_mul_cases_o:NnNnww
19312 /
19313 { - }
19314 _fp_div_npos_o:Nww
19315 {
19316 \or:
19317 _fp_case_use:nw
19318 { _fp_division_by_zero_o:NNww \c_inf_fp / }
19319 \or:
19320 _fp_case_use:nw
19321 { _fp_division_by_zero_o:NNww \c_minus_inf_fp / }
19322 }
19323 }

```

(End definition for `\_fp/_o:ww`.)

`\_fp_div_npos_o:Nww`

```

_fp_div_npos_o:Nww ⟨final sign⟩ \s_fp _fp_chk:w 1 ⟨signA⟩ {⟨exp A⟩}
{⟨A1⟩} {⟨A2⟩} {⟨A3⟩} {⟨A4⟩} ; \s_fp _fp_chk:w 1 ⟨signZ⟩ {⟨exp Z⟩}
{⟨Z1⟩} {⟨Z2⟩} {⟨Z3⟩} {⟨Z4⟩} ;

```

We want to compute  $A/Z$ . As for multiplication, `\_fp_sanitize:Nw` checks for overflow or underflow; we provide it with the `⟨final sign⟩`, and an integer expression in which we compute the exponent. We set up the arguments of `\_fp_div_significand_i_o:wnnw`, namely an integer `⟨y⟩` obtained by adding 1 to the first 5 digits of  $Z$  (explanation given soon below), then the four `{⟨Ai⟩}`, then the four `{⟨Zi⟩}`, a semi-colon, and the `⟨final sign⟩`, used for rounding at the end.

```

19324 \cs_new:Npn _fp_div_npos_o:Nww
19325 #1 \s_fp _fp_chk:w 1 #2 #3 #4 ; \s_fp _fp_chk:w 1 #5 #6 #7#8#9;
19326 {

```

```

19327 \exp_after:wN __fp_sanitize:Nw
19328 \exp_after:wN #1
19329 \int_value:w __fp_int_eval:w
19330 #3 - #6
19331 \exp_after:wN __fp_div_significand_i_o:wnnw
19332 \int_value:w __fp_int_eval:w #7 \use_i:nnnn #8 + 1 ;
19333 #4
19334 {\#7}{\#8}\#9 ;
19335 #1
19336 }

```

(End definition for `\__fp_div_npos_o:Nww`.)

### 31.3.2 Work plan

In this subsection, we explain how to avoid overflowing TeX's integers when performing the division of two positive normal numbers.

We are given two numbers,  $A = 0.A_1A_2A_3A_4$  and  $Z = 0.Z_1Z_2Z_3Z_4$ , in blocks of 4 digits, and we know that the first digits of  $A_1$  and of  $Z_1$  are non-zero. To compute  $A/Z$ , we proceed as follows.

- Find an integer  $Q_A \simeq 10^4 A/Z$ .
- Replace  $A$  by  $B = 10^4 A - Q_A Z$ .
- Find an integer  $Q_B \simeq 10^4 B/Z$ .
- Replace  $B$  by  $C = 10^4 B - Q_B Z$ .
- Find an integer  $Q_C \simeq 10^4 C/Z$ .
- Replace  $C$  by  $D = 10^4 C - Q_C Z$ .
- Find an integer  $Q_D \simeq 10^4 D/Z$ .
- Consider  $E = 10^4 D - Q_D Z$ , and ensure correct rounding.

The result is then  $Q = 10^{-4}Q_A + 10^{-8}Q_B + 10^{-12}Q_C + 10^{-16}Q_D + \text{rounding}$ . Since the  $Q_i$  are integers,  $B$ ,  $C$ ,  $D$ , and  $E$  are all exact multiples of  $10^{-16}$ , in other words, computing with 16 digits after the decimal separator yields exact results. The problem is the risk of overflow: in general  $B$ ,  $C$ ,  $D$ , and  $E$  may be greater than 1.

Unfortunately, things are not as easy as they seem. In particular, we want all intermediate steps to be positive, since negative results would require extra calculations at the end. This requires that  $Q_A \leq 10^4 A/Z$  etc. A reasonable attempt would be to define  $Q_A$  as

$$\text{int\_eval:n} \left\{ \frac{A_1 A_2}{Z_1 + 1} - 1 \right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that  $\varepsilon$ -TeX's `\__fp_int_eval:w` rounds divisions instead of truncating (really,  $1/2$  would be sufficient, but we work with integers). We add 1 to  $Z_1$  because  $Z_1 \leq 10^4 Z < Z_1 + 1$  and we need  $Q_A$  to be an underestimate. However, we are now underestimating  $Q_A$  too much: it can be wrong by up to 100, for instance when  $Z = 0.1$  and  $A \simeq 1$ . Then  $B$  could take values up to 10 (maybe more), and a few steps down the line, we would run into arithmetic overflow, since TeX can only handle integers less than roughly  $2 \cdot 10^9$ .

A better formula is to take

$$Q_A = \text{\texttt{int\_eval:n}} \left\{ \frac{10 \cdot A_1 A_2}{\lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1} - 1 \right\}.$$

This is always less than  $10^9 A / (10^5 Z)$ , as we wanted. In words, we take the 5 first digits of  $Z$  into account, and the 8 first digits of  $A$ , using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the  $Q_i$  avoids any overflow. For convenience, let us denote

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that  $\varepsilon$ -TeX rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that  $10^4 < y \leq 10^5$ , and  $999 \leq Q_A \leq 99989$ . Also note that this formula does not cause an overflow as long as  $A < (2^{31} - 1)/10^9 \simeq 2.147 \dots$ , since the numerator involves an integer slightly smaller than  $10^9 A$ .

Let us bound  $B$ :

$$\begin{aligned} 10^5 B &= A_1 A_2 0 + 10 \cdot 0 \cdot A_3 A_4 - 10 \cdot Z_1 \cdot Z_2 \cdot Z_3 \cdot Z_4 \cdot Q_A \\ &< A_1 A_2 0 \cdot \left( 1 - 10 \cdot \frac{Z_1 \cdot Z_2 \cdot Z_3 \cdot Z_4}{y} \right) + \frac{3}{2} \cdot 10 \cdot Z_1 \cdot Z_2 \cdot Z_3 \cdot Z_4 + 10 \\ &\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1 \cdot Z_2 \cdot Z_3 \cdot Z_4)}{y} + \frac{3}{2} y + 10 \\ &\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y. \end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned} 10^5 B &< 10^9 A / y + 1.6y, \\ 10^5 C &< 10^9 B / y + 1.6y, \\ 10^5 D &< 10^9 C / y + 1.6y, \\ 10^5 E &< 10^9 D / y + 1.6y. \end{aligned}$$

The goal is now to prove that none of  $B$ ,  $C$ ,  $D$ , and  $E$  can go beyond  $(2^{31} - 1)/10^9 = 2.147 \dots$

Combining the various inequalities together with  $A < 1$ , we get

$$\begin{aligned} 10^5 B &< 10^9 / y + 1.6y, \\ 10^5 C &< 10^{13} / y^2 + 1.6(y + 10^4), \\ 10^5 D &< 10^{17} / y^3 + 1.6(y + 10^4 + 10^8 / y), \\ 10^5 E &< 10^{21} / y^4 + 1.6(y + 10^4 + 10^8 / y + 10^{12} / y^2). \end{aligned}$$

All of those bounds are convex functions of  $y$  (since every power of  $y$  involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range  $10^4 < y \leq 10^5$ . Thus,

$$\begin{aligned} 10^5 B &< \max(1.16 \cdot 10^5, 1.7 \cdot 10^5), \\ 10^5 C &< \max(1.32 \cdot 10^5, 1.77 \cdot 10^5), \\ 10^5 D &< \max(1.48 \cdot 10^5, 1.777 \cdot 10^5), \\ 10^5 E &< \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5). \end{aligned}$$

All of those bounds are less than  $2.147 \cdot 10^5$ , and we are thus within  $\text{\TeX}'$ s bounds in all cases!

We later need to have a bound on the  $Q_i$ . Their definitions imply that  $Q_A < 10^9 A/y - 1/2 < 10^5 A$  and similarly for the other  $Q_i$ . Thus, all of them are less than 177770.

The last step is to ensure correct rounding. We have

$$A/Z = \sum_{i=1}^4 (10^{-4i} Q_i) + 10^{-16} E/Z$$

exactly. Furthermore, we know that the result is in  $[0.1, 10)$ , hence will be rounded to a multiple of  $10^{-16}$  or of  $10^{-15}$ , so we only need to know the integer part of  $E/Z$ , and a “rounding” digit encoding the rest. Equivalently, we need to find the integer part of  $2E/Z$ , and determine whether it was an exact integer or not (this serves to detect ties). Since

$$\frac{2E}{Z} = 2 \frac{10^5 E}{10^5 Z} \leq 2 \frac{10^5 E}{10^4} < 36,$$

this integer part is between 0 and 35 inclusive. We let  $\varepsilon\text{-}\text{\TeX}$  round

$$P = \text{\int\_eval:n} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

which differs from  $2E/Z$  by at most

$$\frac{1}{2} + 2 \left| \frac{E}{Z} - \frac{E}{10^{-8} Z_1 Z_2} \right| + 2 \left| \frac{10^8 E - E_1 E_2}{Z_1 Z_2} \right| < 1,$$

( $1/2$  comes from  $\varepsilon\text{-}\text{\TeX}$ 's rounding) because each absolute value is less than  $10^{-7}$ . Thus  $P$  is either the correct integer part, or is off by 1; furthermore, if  $2E/Z$  is an integer,  $P = 2E/Z$ . We will check the sign of  $2E - PZ$ . If it is negative, then  $E/Z \in ((P-1)/2, P/2)$ . If it is zero, then  $E/Z = P/2$ . If it is positive, then  $E/Z \in (P/2, (P-1)/2)$ . In each case, we know how to round to an integer, depending on the parity of  $P$ , and the rounding mode.

### 31.3.3 Implementing the significand division

```
__fp_div_significand_i_o:wnnw <y> ; {<A1>} {<A2>} {<A3>} {<A4>}
{<Z1>} {<Z2>} {<Z3>} {<Z4>} ; <sign>
```

Compute  $10^6 + Q_A$  (a 7 digit number thanks to the shift), unbrace  $\langle A_1 \rangle$  and  $\langle A_2 \rangle$ , and prepare the  $\langle \text{continuation} \rangle$  arguments for 4 consecutive calls to  $\text{\_\_fp\_div\_significand\_calc:wnnnnnnnn}$ . Each of these calls needs  $\langle y \rangle$  (#1), and it turns out that

we need post-expansion there, hence the `\int_value:w`. Here, #4 is six brace groups, which give the six first n-type arguments of the `calc` function.

```

19337 \cs_new:Npn __fp_div_significand_i_o:wnnw #1 ; #2#3 #4 ;
19338 {
19339 \exp_after:wN __fp_div_significand_test_o:w
19340 \int_value:w __fp_int_eval:w
19341 \exp_after:wN __fp_div_significand_calc:wwnnnnnnn
19342 \int_value:w __fp_int_eval:w 999999 + #2 #3 0 / #1 ;
19343 #2 #3 ;
19344 #4
19345 { \exp_after:wN __fp_div_significand_ii:wwn \int_value:w #1 }
19346 { \exp_after:wN __fp_div_significand_ii:wwn \int_value:w #1 }
19347 { \exp_after:wN __fp_div_significand_ii:wwn \int_value:w #1 }
19348 { \exp_after:wN __fp_div_significand_iii:wwnnnnn \int_value:w #1 }
19349 }
```

(End definition for `\__fp_div_significand_i_o:wnnw`.)

$$\begin{aligned} & \__fp_div_significand_calc:wwnnnnnn \\ & \__fp_div_significand_calc_i:wwnnnnnn \\ & \__fp_div_significand_calc_ii:wwnnnnnn \end{aligned}$$

`\__fp_div_significand_calc:wwnnnnnnn`  $\langle 10^6 + Q_A \rangle ; \langle A_1 \rangle \langle A_2 \rangle ; \{\langle A_3 \rangle\}$   
 $\{\langle A_4 \rangle\} \{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \{\langle Z_4 \rangle\} \{\langle \text{continuation} \rangle\}$

expands to

$$\langle 10^6 + Q_A \rangle \{\langle \text{continuation} \rangle\} ; \langle B_1 \rangle \langle B_2 \rangle ; \{\langle B_3 \rangle\} \{\langle B_4 \rangle\} \{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \{\langle Z_4 \rangle\}$$

where  $B = 10^4 A - Q_A \cdot Z$ . This function is also used to compute  $C, D, E$  (with the input shifted accordingly), and is used in `l3fp-expo`.

We know that  $0 < Q_A < 1.8 \cdot 10^5$ , so the product of  $Q_A$  with each  $Z_i$  is within  $\text{\TeX}'$ s bounds. However, it is a little bit too large for our purposes: we would not be able to use the usual trick of adding a large power of 10 to ensure that the number of digits is fixed.

The bound on  $Q_A$ , implies that  $10^6 + Q_A$  starts with the digit 1, followed by 0 or 1. We test, and call different auxiliaries for the two cases. An earlier implementation did the tests within the computation, but since we added a  $\langle \text{continuation} \rangle$ , this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

$$\begin{aligned} & 10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5 \#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7) \\ & + 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8), \end{aligned}$$

where  $\langle i \rangle$  stands for the  $10^5$  digit of  $Q_A$ , which is 0 or 1, and  $\#1, \#2, \dots$  are the parameters of either auxiliary. The factors of 10 come from the fact that  $Q_A = 10 \cdot 10^4 \cdot \langle i \rangle + \#1$ . As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most  $10^8$  and the negative contributions can go up to  $10^9$ . Indeed, for the auxiliary with  $\langle i \rangle = 1$ ,  $\#1$  is at most 80000, leading to contributions of at worse  $-8 \cdot 10^8 4$ , while the other negative term is very small  $< 10^6$  (except in the first expression, where we don't care about the number of digits); for the auxiliary with  $\langle i \rangle = 0$ ,  $\#1$  can go up to 99999, but there is no other negative term. Hence, a good choice is  $2 \cdot 10^9$ , which produces totals in the range  $[10^9, 2.1 \cdot 10^9]$ . We are flirting with  $\text{\TeX}'$ s limits once more.

```
19350 \cs_new:Npn __fp_div_significand_calc:wwnnnnnnn 1#1
```

```

19351 {
19352 \if_meaning:w 1 #1
19353 \exp_after:wN __fp_div_significand_calc_i:wwnnnnnnn
19354 \else:
19355 \exp_after:wN __fp_div_significand_calc_ii:wwnnnnnnn
19356 \fi:
19357 }
19358 \cs_new:Npn __fp_div_significand_calc_i:wwnnnnnnn
19359 #1; #2;#3#4 #5#6#7#8 #9
19360 {
19361 1 1 #1
19362 #9 \exp_after:wN ;
19363 \int_value:w __fp_int_eval:w \c_fp_Bigg_leading_shift_int
19364 + #2 - #1 * #5 - #5#60
19365 \exp_after:wN __fp_pack_Bigg:NNNNNNw
19366 \int_value:w __fp_int_eval:w \c_fp_Bigg_middle_shift_int
19367 + #3 - #1 * #6 - #70
19368 \exp_after:wN __fp_pack_Bigg:NNNNNNw
19369 \int_value:w __fp_int_eval:w \c_fp_Bigg_middle_shift_int
19370 + #4 - #1 * #7 - #80
19371 \exp_after:wN __fp_pack_Bigg:NNNNNNw
19372 \int_value:w __fp_int_eval:w \c_fp_Bigg_trailing_shift_int
19373 - #1 * #8 ;
19374 {#5}{#6}{#7}{#8}
19375 }
19376 \cs_new:Npn __fp_div_significand_calc_ii:wwnnnnnnn
19377 #1; #2;#3#4 #5#6#7#8 #9
19378 {
19379 1 0 #1
19380 #9 \exp_after:wN ;
19381 \int_value:w __fp_int_eval:w \c_fp_Bigg_leading_shift_int
19382 + #2 - #1 * #5
19383 \exp_after:wN __fp_pack_Bigg:NNNNNNw
19384 \int_value:w __fp_int_eval:w \c_fp_Bigg_middle_shift_int
19385 + #3 - #1 * #6
19386 \exp_after:wN __fp_pack_Bigg:NNNNNNw
19387 \int_value:w __fp_int_eval:w \c_fp_Bigg_middle_shift_int
19388 + #4 - #1 * #7
19389 \exp_after:wN __fp_pack_Bigg:NNNNNNw
19390 \int_value:w __fp_int_eval:w \c_fp_Bigg_trailing_shift_int
19391 - #1 * #8 ;
19392 {#5}{#6}{#7}{#8}
19393 }

```

(End definition for `\__fp_div_significand_calc:wwnnnnnnn`, `\__fp_div_significand_calc_i:wwnnnnnnn`, and `\__fp_div_significand_calc_ii:wwnnnnnnn`.)

`\__fp_div_significand_ii:wwn`  $\langle y \rangle ; \langle B_1 \rangle ; \{ \langle B_2 \rangle \} \{ \langle B_3 \rangle \} \{ \langle B_4 \rangle \} \{ \langle Z_1 \rangle \}$   
 $\{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \} \langle \text{continuations} \rangle \langle \text{sign} \rangle$

Compute  $Q_B$  by evaluating  $\langle B_1 \rangle \langle B_2 \rangle 0/y - 1$ . The result is output to the left, in an `\__fp_int_eval:w` which we start now. Once that is evaluated (and the other  $Q_i$  also, since later expansions are triggered by this one), a packing auxiliary takes care of placing the digits of  $Q_B$  in an appropriate way for the final addition to obtain  $Q$ . This auxiliary is also used to compute  $Q_C$  and  $Q_D$  with the inputs  $C$  and  $D$  instead of  $B$ .

```

19394 \cs_new:Npn __fp_div_significand_ii:wwn #1; #2;#3
19395 {
19396 \exp_after:wN __fp_div_significand_pack:NNN
19397 \int_value:w __fp_int_eval:w
19398 \exp_after:wN __fp_div_significand_calc:wwnnnnnnn
19399 \int_value:w __fp_int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
19400 }

```

(End definition for `\__fp_div_significand_ii:wwn`.)

```
__fp_div_significand_iii:wwnnnnn <y> ; <E1> ; {<E2>} {<E3>} {<E4>}
{<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>
```

We compute  $P \simeq 2E/Z$  by rounding  $2E_1 E_2 / Z_1 Z_2$ . Note the first 0, which multiplies  $Q_D$  by 10: we later add (roughly)  $5 \cdot P$ , which amounts to adding  $P/2 \simeq E/Z$  to  $Q_D$ , the appropriate correction from a hypothetical  $Q_E$ .

```

19401 \cs_new:Npn __fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
19402 {
19403 0
19404 \exp_after:wN __fp_div_significand_iv:wwnnnnnnn
19405 \int_value:w __fp_int_eval:w (2 * #2 #3) / #6 #7 ; % <- P
19406 #2 ; {#3} {#4} {#5}
19407 {#6} {#7}
19408 }

```

(End definition for `\__fp_div_significand_iii:wwnnnnn`.)

```
__fp_div_significand_iv:wwnnnnnnn <P> ; <E1> ; {<E2>} {<E3>} {<E4>}
{<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>
```

This adds to the current expression ( $10^7 + 10 \cdot Q_D$ ) a contribution of  $5 \cdot P + \text{sign}(T)$  with  $T = 2E - PZ$ . This amounts to adding  $P/2$  to  $Q_D$ , with an extra *(rounding)* digit. This *(rounding)* digit is 0 or 5 if  $T$  does not contribute, *i.e.*, if  $0 = T = 2E - PZ$ , in other words if  $10^{16} A/Z$  is an integer or half-integer. Otherwise it is in the appropriate range, [1, 4] or [6, 9]. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute  $T$  exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation  $\#1 \cdot \#6\#7$  below does not cause an overflow: naively,  $P$  can be up to 35, and  $\#6\#7$  up to  $10^8$ , but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For  $P < 10$ , the product obeys  $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$ .
- For large  $P \geq 3$ , the rounding error on  $P$ , which is at most 1, is less than a factor of 2, hence  $P \leq 4E/Z$ . Also,  $\#6\#7 \leq 10^8 \cdot Z$ , hence  $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$ .

Both inequalities could be made tighter if needed.

Note however that  $P \cdot \#8\#9$  may overflow, since the two factors are now independent, and the result may reach  $3.5 \cdot 10^9$ . Thus we compute the two lower levels separately. The rest is standard, except that we use `+` as a separator (ending integer expressions explicitly).  $T$  is negative if the first character is `-`, it is positive if the first character is neither `0` nor `-`. It is also positive if the first character is `0` and second argument of `\__fp_div_significand_vi:Nw`, a sum of several terms, is also zero. Otherwise, there was an exact agreement:  $T = 0$ .

```

19409 \cs_new:Npn __fp_div_significand_iv:wwnnnnnn #1; #2;#3#4#5 #6#7#8#9
19410 {
19411 + 5 * #1
19412 \exp_after:wN __fp_div_significand_vi:Nw
19413 \int_value:w __fp_int_eval:w -20 + 2*#2#3 - #1*#6#7 +
19414 \exp_after:wN __fp_div_significand_v>NN
19415 \int_value:w __fp_int_eval:w 199980 + 2*#4 - #1*#8 +
19416 \exp_after:wN __fp_div_significand_v>NN
19417 \int_value:w __fp_int_eval:w 200000 + 2*#5 - #1*#9 ;
19418 }
19419 \cs_new:Npn __fp_div_significand_v>NN #1#2 { #1#2 __fp_int_eval_end: + }
19420 \cs_new:Npn __fp_div_significand_vi:Nw #1#2;
19421 {
19422 \if_meaning:w 0 #1
19423 \if_int_compare:w __fp_int_eval:w #2 > 0 + 1 \fi:
19424 \else:
19425 \if_meaning:w - #1 - \else: + \fi: 1
19426 \fi:
19427 ;
19428 }

```

(End definition for  $\__fp\_div\_significand\_iv:wwnnnnnn$ ,  $\__fp\_div\_significand\_v:NNw$ , and  $\__fp\_div\_significand\_vi:Nw$ .)

$\__fp\_div\_significand\_pack:NNN$

At this stage, we are in the following situation: TeX is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.

$$\__fp\_div\_significand\_test_o:w 10^6 + Q_A \__fp\_div\_significand\_pack:NNN 10^6 + Q_B \__fp\_div\_significand\_pack:NNN 10^6 + Q_C \__fp\_div\_significand\_pack:NNN 10^7 + 10 \cdot Q_D + 5 \cdot P + \varepsilon ; \langle sign \rangle$$

Here,  $\varepsilon = \text{sign}(T)$  is 0 in case  $2E = PZ$ , 1 in case  $2E > PZ$ , which means that  $P$  was the correct value, but not with an exact quotient, and  $-1$  if  $2E < PZ$ , i.e.,  $P$  was an overestimate. The packing function we define now does nothing special: it removes the  $10^6$  and carries two digits (for the  $10^5$ 's and the  $10^4$ 's).

```
19429 \cs_new:Npn __fp_div_significand_pack:NNN 1 #1 #2 { + #1 #2 ; }
```

(End definition for  $\__fp\_div\_significand\_pack:NNN$ .)

$\__fp\_div\_significand\_test_o:w$

The reason we know that the first two digits are 1 and 0 is that the final result is known to be between 0.1 (inclusive) and 10, hence  $\tilde{Q}_A$  (the tilde denoting the contribution from the other  $Q_i$ ) is at most 99999, and  $10^6 + \tilde{Q}_A = 10 \dots$ .

It is now time to round. This depends on how many digits the final result will have.

```

19430 \cs_new:Npn __fp_div_significand_test_o:w 10 #1
19431 {
19432 \if_meaning:w 0 #1
19433 \exp_after:wN __fp_div_significand_small_o:wwwNNNNwN
19434 \else:
19435 \exp_after:wN __fp_div_significand_large_o:wwwNNNNwN
19436 \fi:
19437 #1
19438 }

```

(End definition for  $\__fp\_div\_significand\_test_o:w$ .)

```
__fp_div_significand_small_o:wwwNNNNwN
__fp_div_significand_small_o:wwwNNNNwN 0 ⟨4d⟩ ; ⟨4d⟩ ; ⟨4d⟩ ; ⟨5d⟩
; ⟨final sign⟩
```

Standard use of the functions `\__fp_basics_pack_low:NNNNNw` and `\__fp_basics_pack_high:NNNNNw`. We finally get to use the `⟨final sign⟩` which has been sitting there for a while.

```
19439 \cs_new:Npn __fp_div_significand_small_o:wwwNNNNwN
19440 0 #1; #2; #3; #4#5#6#7#8; #9
19441 {
19442 \exp_after:wN __fp_basics_pack_high:NNNNNw
19443 \int_value:w __fp_int_eval:w 1 #1#2
19444 \exp_after:wN __fp_basics_pack_low:NNNNNw
19445 \int_value:w __fp_int_eval:w 1 #3#4#5#6#7
19446 + __fp_round:NNN #9 #7 #8
19447 \exp_after:wN ;
19448 }
```

(End definition for `\__fp_div_significand_small_o:wwwNNNNwN`.)

```
__fp_div significand large o:wwwNNNNwN
__fp_div_significand_large_o:wwwNNNNwN ⟨5d⟩ ; ⟨4d⟩ ; ⟨4d⟩ ; ⟨5d⟩ ;
⟨sign⟩
```

We know that the final result cannot reach 10, hence `1#1#2`, together with contributions from the level below, cannot reach  $2 \cdot 10^9$ . For rounding, we build the `⟨rounding digit⟩` from the last two of our 18 digits.

```
19449 \cs_new:Npn __fp_div_significand_large_o:wwwNNNNwN
19450 #1; #2; #3; #4#5#6#7#8; #9
19451 {
19452 + 1
19453 \exp_after:wN __fp_basics_pack_weird_high:NNNNNNNNw
19454 \int_value:w __fp_int_eval:w 1 #1 #2
19455 \exp_after:wN __fp_basics_pack_weird_low:NNNNw
19456 \int_value:w __fp_int_eval:w 1 #3 #4 #5 #6 +
19457 \exp_after:wN __fp_round:NNN
19458 \exp_after:wN #9
19459 \exp_after:wN #6
19460 \int_value:w __fp_round_digit:Nw #7 #8 ;
19461 \exp_after:wN ;
19462 }
```

(End definition for `\__fp_div_significand_large_o:wwwNNNNwN`.)

### 31.4 Square root

`\__fp_sqrt_o:w` Zeros are unchanged:  $\sqrt{-0} = -0$  and  $\sqrt{+0} = +0$ . Negative numbers (other than  $-0$ ) have no real square root. Positive infinity, and `nan`, are unchanged. Finally, for normal positive numbers, there is some work to do.

```
19463 \cs_new:Npn __fp_sqrt_o:w #1 \s_fp __fp_chk:w #2#3#4; @
19464 {
19465 \if_meaning:w 0 #2 __fp_case_return_same_o:w \fi:
19466 \if_meaning:w 2 #3
19467 __fp_case_use:nw { __fp_invalid_operation_o:nw { sqrt } }
19468 \fi:
19469 \if_meaning:w 1 #2 \else: __fp_case_return_same_o:w \fi:
19470 __fp_sqrt_npos_o:w
```

```

19471 \s_fp \fp_chk:w #2 #3 #4;
19472 }

```

(End definition for `\_fp_sqrt_o:w`.)

```

\fp_sqrt_npos_o:w
\fp_sqrt_npos_auxi_o:wwnnN
\fp_sqrt_npos_auxii_o:wNNNNNNNN

```

Prepare `\_fp_sanitize:Nw` to receive the final sign 0 (the result is always positive) and the exponent, equal to half of the exponent #1 of the argument. If the exponent #1 is even, find a first approximation of the square root of the significand  $10^8 a_1 + a_2 = 10^8 \#2\#3+\#4\#5$  through Newton's method, starting at  $x = 56234133 \simeq 10^{7.75}$ . Otherwise, first shift the significand of the argument by one digit, getting  $a'_1 \in [10^6, 10^7)$  instead of  $[10^7, 10^8)$ , then use Newton's method starting at  $17782794 \simeq 10^{7.25}$ .

```

19473 \cs_new:Npn _fp_sqrt_npos_o:w \s_fp \fp_chk:w 1 0 #1#2#3#4#5;
19474 {
19475 \exp_after:wN _fp_sanitize:Nw
19476 \exp_after:wN 0
19477 \int_value:w _fp_int_eval:w
19478 \if_int_odd:w #1 \exp_stop_f:
19479 \exp_after:wN _fp_sqrt_npos_auxi_o:wwnnN
19480 \fi:
19481 #1 / 2
19482 _fp_sqrt_Newton_o:wwn 56234133; 0; {#2#3} {#4#5} 0
19483 }
19484 \cs_new:Npn _fp_sqrt_npos_auxi_o:wwnnN #1 / 2 #2; 0; #3#4#5
19485 {
19486 (#1 + 1) / 2
19487 _fp_pack_eight:wNNNNNNNN
19488 _fp_sqrt_npos_auxii_o:wNNNNNNNN
19489 ;
19490 0 #3 #4
19491 }
19492 \cs_new:Npn _fp_sqrt_npos_auxii_o:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
19493 { _fp_sqrt_Newton_o:wwn 17782794; 0; {#1} {#2#3#4#5#6#7#8#9} }

```

(End definition for `\_fp_sqrt_npos_o:w`, `\_fp_sqrt_npos_auxi_o:wwnnN`, and `\_fp_sqrt_npos_auxii_o:wNNNNNNNN`.)

```
_fp_sqrt_Newton_o:wwn
```

Newton's method maps  $x \mapsto [(x + [10^8 a_1/x])/2]$  in each iteration, where  $[b/c]$  denotes  $\varepsilon$ -TeX's division. This division rounds the real number  $b/c$  to the closest integer, rounding ties away from zero, hence when  $c$  is even,  $b/c - 1/2 + 1/c \leq [b/c] \leq b/c + 1/2$  and when  $c$  is odd,  $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2 - 1/(2c)$ . For all  $c$ ,  $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2$ .

Let us prove that the method converges when implemented with  $\varepsilon$ -TeX integer division, for any  $10^6 \leq a_1 < 10^8$  and starting value  $10^6 \leq x < 10^8$ . Using the inequalities above and the arithmetic-geometric inequality  $(x + t)/2 \geq \sqrt{xt}$  for  $t = 10^8 a_1/x$ , we find

$$x' = \left[ \frac{x + [10^8 a_1/x]}{2} \right] \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}.$$

After any step of iteration, we thus have  $\delta = x - \sqrt{10^8 a_1} \geq -0.25 + 0.25 \cdot 10^{-8}$ . The new difference  $\delta' = x' - \sqrt{10^8 a_1}$  after one step is bounded above as

$$x' - \sqrt{10^8 a_1} \leq \frac{x + 10^8 a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8 a_1} \leq \frac{\delta}{2} \frac{\delta}{\sqrt{10^8 a_1} + \delta} + \frac{3}{4}.$$

For  $\delta > 3/2$ , this last expression is  $\leq \delta/2 + 3/4 < \delta$ , hence  $\delta$  decreases at each step: since all  $x$  are integers,  $\delta$  must reach a value  $-1/4 < \delta \leq 3/2$ . In this range of values, we get  $\delta' \leq \frac{3}{4} \frac{3}{2\sqrt{10^8 a_1}} + \frac{3}{4} \leq 0.75 + 1.125 \cdot 10^{-7}$ . We deduce that the difference  $\delta = x - \sqrt{10^8 a_1}$  eventually reaches a value in the interval  $[-0.25 + 0.25 \cdot 10^{-8}, 0.75 + 11.25 \cdot 10^{-8}]$ , whose width is  $1 + 11 \cdot 10^{-8}$ . The corresponding interval for  $x$  may contain two integers, hence  $x$  might oscillate between those two values.

However, the fact that  $x \mapsto x - 1$  and  $x - 1 \mapsto x$  puts stronger constraints, which are not compatible: the first implies

$$x + [10^8 a_1/x] \leq 2x - 2$$

hence  $10^8 a_1/x \leq x - 3/2$ , while the second implies

$$x - 1 + [10^8 a_1/(x - 1)] \geq 2x - 1$$

hence  $10^8 a_1/(x - 1) \geq x - 1/2$ . Combining the two inequalities yields  $x^2 - 3x/2 \geq 10^8 a_1 \geq x - 3x/2 + 1/2$ , which cannot hold. Therefore, the iteration always converges to a single integer  $x$ . To stop the iteration when two consecutive results are equal, the function `\_fp_sqrt_Newton_o:wwn` receives the newly computed result as #1, the previous result as #2, and  $a_1$  as #3. Note that  $\varepsilon$ -TeX combines the computation of a multiplication and a following division, thus avoiding overflow in `#3 * 100000000 / #1`. In any case, the result is within  $[10^7, 10^8]$ .

```

19494 \cs_new:Npn _fp_sqrt_Newton_o:wwn #1; #2; #3
19495 {
19496 \if_int_compare:w #1 = #2 \exp_stop_f:
19497 \exp_after:wN _fp_sqrt_auxi_o:NNNNwnnN
19498 \int_value:w _fp_int_eval:w 9999 9999 +
19499 \exp_after:wN _fp_use_none_until_s:w
19500 \fi:
19501 \exp_after:wN _fp_sqrt_Newton_o:wwn
19502 \int_value:w _fp_int_eval:w (#1 + #3 * 1 0000 0000 / #1) / 2 ;
19503 #1; {#3}
19504 }

```

(End definition for `\_fp_sqrt_Newton_o:wwn`.)

`\_fp_sqrt_auxi_o:NNNNwnnN` This function is followed by  $10^8 + x - 1$ , which has 9 digits starting with 1, then ;  $\{\langle a_1 \rangle\} \{\langle a_2 \rangle\} \langle a' \rangle$ . Here,  $x \simeq \sqrt{10^8 a_1}$  and we want to estimate the square root of  $a = 10^{-8} a_1 + 10^{-16} a_2 + 10^{-17} a'$ . We set up an initial underestimate

$$y = (x - 1)10^{-8} + 0.2499998875 \cdot 10^{-8} \lesssim \sqrt{a}.$$

From the inequalities shown earlier, we know that  $y \leq \sqrt{10^{-8} a_1} \leq \sqrt{a}$  and that  $\sqrt{10^{-8} a_1} \leq y + 10^{-8} + 11 \cdot 10^{-16}$  hence (using  $0.1 \leq y \leq \sqrt{a} \leq 1$ )

$$a - y^2 \leq 10^{-8} a_1 + 10^{-8} - y^2 \leq (y + 10^{-8} + 11 \cdot 10^{-16})^2 - y^2 + 10^{-8} < 3.2 \cdot 10^{-8},$$

and  $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y) \leq 16 \cdot 10^{-8}$ . Next, `\_fp_sqrt_auxii_o:NnnnnnnnnN` is called several times to get closer and closer underestimates of  $\sqrt{a}$ . By construction, the underestimates  $y$  are always increasing,  $a - y^2 < 3.2 \cdot 10^{-8}$  for all. Also,  $y < 1$ .

```

19505 \cs_new:Npn _fp_sqrt_auxi_o:NNNNwnnN 1 #1#2#3#4#5;
19506 {

```

```

19507 __fp_sqrt_auxii_o:NnnnnnnnN
19508 __fp_sqrt_auxii_o:wnnnnnnnn
19509 {\#1#\#3#\#4} {\#5} {2499} {9988} {7500}
19510 }

```

(End definition for `\__fp_sqrt_auxi_o:NNNNwnnN.`)

`\__fp_sqrt_auxii_o:NnnnnnnnN` This receives a continuation function #1, then five blocks of 4 digits for  $y$ , then two 8-digit blocks and a single digit for  $a$ . A common estimate of  $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y)$  is  $(a - y^2)/(2y)$ , which leads to alternating overestimates and underestimates. We tweak this, to only work with underestimates (no need then to worry about signs in the computation). Each step finds the largest integer  $j \leq 6$  such that  $10^{4j}(a - y^2) < 2 \cdot 10^8$ , then computes the integer (with  $\varepsilon$ -TEX's rounding division)

$$10^{4j}z = \left[ (|10^{4j}(a - y^2)| - 257) \cdot (0.5 \cdot 10^8) / |10^8y + 1| \right].$$

The choice of  $j$  ensures that  $10^{4j}z < 2 \cdot 10^8 \cdot 0.5 \cdot 10^8 / 10^7 = 10^9$ , thus  $10^9 + 10^{4j}z$  has exactly 10 digits, does not overflow T<sub>E</sub>X's integer range, and starts with 1. Incidentally, since all  $a - y^2 \leq 3.2 \cdot 10^{-8}$ , we know that  $j \geq 3$ .

Let us show that  $z$  is an underestimate of  $\sqrt{a} - y$ . On the one hand,  $\sqrt{a} - y \leq 16 \cdot 10^{-8}$  because this holds for the initial  $y$  and values of  $y$  can only increase. On the other hand, the choice of  $j$  implies that  $\sqrt{a} - y \leq 5(\sqrt{a} + y)(\sqrt{a} - y) = 5(a - y^2) < 10^{9-4j}$ . For  $j = 3$ , the first bound is better, while for larger  $j$ , the second bound is better. For all  $j \in [3, 6]$ , we find  $\sqrt{a} - y < 16 \cdot 10^{-2j}$ . From this, we deduce that

$$10^{4j}(\sqrt{a} - y) = \frac{10^{4j}(a - y^2 - (\sqrt{a} - y)^2)}{2y} \geq \frac{|10^{4j}(a - y^2)| - 257}{2 \cdot 10^{-8}|10^8y + 1|} + \frac{1}{2}$$

where we have replaced the bound  $10^{4j}(16 \cdot 10^{-2j}) = 256$  by 257 and extracted the corresponding term  $1/(2 \cdot 10^{-8}|10^8y + 1|) \geq 1/2$ . Given that  $\varepsilon$ -TEX's integer division obeys  $[b/c] \leq b/c + 1/2$ , we deduce that  $10^{4j}z \leq 10^{4j}(\sqrt{a} - y)$ , hence  $y + z \leq \sqrt{a}$  is an underestimate of  $\sqrt{a}$ , as claimed. One implementation detail: because the computation involves `-#4*#4 - 2*#3*#5 - 2*#2*#6` which may be as low as  $-5 \cdot 10^8$ , we need to use the `pack_big` functions, and the `big` shifts.

```

19511 \cs_new:Npn __fp_sqrt_auxii_o:NnnnnnnnN #1 #2#3#4#5#6 #7#8#9
19512 {
19513 \exp_after:wN #1
19514 \int_value:w __fp_int_eval:w \c__fp_big_leading_shift_int
19515 + #7 - #2 * #2
19516 \exp_after:wN __fp_pack_big:NNNNNNw
19517 \int_value:w __fp_int_eval:w \c__fp_big_middle_shift_int
19518 - 2 * #2 * #3
19519 \exp_after:wN __fp_pack_big:NNNNNNw
19520 \int_value:w __fp_int_eval:w \c__fp_big_middle_shift_int
19521 + #8 - #3 * #3 - 2 * #2 * #4
19522 \exp_after:wN __fp_pack_big:NNNNNNw
19523 \int_value:w __fp_int_eval:w \c__fp_big_middle_shift_int
19524 - 2 * #3 * #4 - 2 * #2 * #5
19525 \exp_after:wN __fp_pack_big:NNNNNNw
19526 \int_value:w __fp_int_eval:w \c__fp_big_middle_shift_int
19527 + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
19528 \exp_after:wN __fp_pack_big:NNNNNNw

```

```

19529 \int_value:w _fp_int_eval:w \c_fp_middle_shift_int
19530 - 2 * #4 * #5 - 2 * #3 * #6
19531 \exp_after:wN _fp_pack_big:NNNNNNw
19532 \int_value:w _fp_int_eval:w \c_fp_middle_shift_int
19533 - #5 * #5 - 2 * #4 * #6
19534 \exp_after:wN _fp_pack_big:NNNNNNw
19535 \int_value:w _fp_int_eval:w
19536 \c_fp_middle_shift_int
19537 - 2 * #5 * #6
19538 \exp_after:wN _fp_pack_big:NNNNNNw
19539 \int_value:w _fp_int_eval:w
19540 \c_fp_trailing_shift_int
19541 - #6 * #6 ;
19542 % (
19543 - 257) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
19544 {#2}{#3}{#4}{#5}{#6} {#7}{#8}#9
19545 }

```

(End definition for `\_fp_sqrt_auxii_o:NnnnnnnnN`.)

We receive here the difference  $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$ , as  $\langle d_2 \rangle ; \{\langle d_3 \rangle\} \dots \{\langle d_{10} \rangle\}$ , where each block has 4 digits, except  $\langle d_2 \rangle$ . This function finds the largest  $j \leq 6$  such that  $10^{4j}(a - y^2) < 2 \cdot 10^8$ , then leaves an open parenthesis and the integer  $\lfloor 10^{4j}(a - y^2) \rfloor$  in an integer expression. The closing parenthesis is provided by the caller `\_fp_sqrt_auxii_o:NnnnnnnnN`, which completes the expression

$$10^{4j}z = \left[ (\lfloor 10^{4j}(a - y^2) \rfloor - 257) \cdot (0.5 \cdot 10^8) \Big/ \lfloor 10^8y + 1 \rfloor \right]$$

for an estimate of  $10^{4j}(\sqrt{a} - y)$ . If  $d_2 \geq 2$ ,  $j = 3$  and the `auxiv` auxiliary receives  $10^{12}z$ . If  $d_2 \leq 1$  but  $10^4d_2 + d_3 \geq 2$ ,  $j = 4$  and the `auxv` auxiliary is called, and receives  $10^{16}z$ , and so on. In all those cases, the `auxviii` auxiliary is set up to add  $z$  to  $y$ , then go back to the `auxii` step with continuation `auxiii` (the function we are currently describing). The maximum value of  $j$  is 6, regardless of whether  $10^{12}d_2 + 10^8d_3 + 10^4d_4 + d_5 \geq 1$ . In this last case, we detect when  $10^{24}z < 10^7$ , which essentially means  $\sqrt{a} - y \lesssim 10^{-17}$ : once this threshold is reached, there is enough information to find the correctly rounded  $\sqrt{a}$  with only one more call to `\_fp_sqrt_auxii_o:NnnnnnnnN`. Note that the iteration cannot be stuck before reaching  $j = 6$ , because for  $j < 6$ , one has  $2 \cdot 10^8 \leq 10^{4(j+1)}(a - y^2)$ , hence

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{\lfloor 10^8y + 1 \rfloor} \geq (20000 - 257) \cdot 0.5 > 0.$$

```

19546 \cs_new:Npn _fp_sqrt_auxii_o:wnnnnnnnn
19547 #1; #2#3#4#5#6#7#8#9
19548 {
19549 \if_int_compare:w #1 > 1 \exp_stop_f:
19550 \exp_after:wN _fp_sqrt_auxiv_o:NNNNNw
19551 \int_value:w _fp_int_eval:w (#1#2 %)
19552 \else:
19553 \if_int_compare:w #1#2 > 1 \exp_stop_f:
19554 \exp_after:wN _fp_sqrt_auxv_o:NNNNNw
19555 \int_value:w _fp_int_eval:w (#1#2#3 %)
19556 \else:
19557 \if_int_compare:w #1#2#3 > 1 \exp_stop_f:
19558 \exp_after:wN _fp_sqrt_auxvi_o:NNNNNw

```

```

19559 \int_value:w __fp_int_eval:w (#1#2#3#4 %)
19560 \else:
19561 \exp_after:wN __fp_sqrt_auxvii_o:NNNNNw
19562 \int_value:w __fp_int_eval:w (#1#2#3#4#5 %)
19563 \fi:
19564 \fi:
19565 \fi:
19566 }
19567 \cs_new:Npn __fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6;
19568 { __fp_sqrt_auxviii_o:nnnnnnnn {#1#2#3#4#5#6} {00000000} }
19569 \cs_new:Npn __fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
19570 { __fp_sqrt_auxviii_o:nnnnnnnn {000#1#2#3#4#5} {#60000} }
19571 \cs_new:Npn __fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
19572 { __fp_sqrt_auxviii_o:nnnnnnnn {000000#1} {#2#3#4#5#6} }
19573 \cs_new:Npn __fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
19574 {
19575 \if_int_compare:w #1#2 = 0 \exp_stop_f:
19576 \exp_after:wN __fp_sqrt_auxxx_o:Nnnnnnnn
19577 \fi:
19578 __fp_sqrt_auxviii_o:nnnnnnnn {00000000} {000#1#2#3#4#5}
19579 }

```

(End definition for  $\_fp_sqrt_auxviii_o:wnnnnnnnn$  and others.)

$\_fp_sqrt_auxviii_o:nnnnnnnn$   
 $\_fp_sqrt_auxix_o:wnwnw$

Simply add the two 8-digit blocks of  $z$ , aligned to the last four of the five 4-digit blocks of  $y$ , then call the `auxii` auxiliary to evaluate  $y'^2 = (y + z)^2$ .

```

19580 \cs_new:Npn __fp_sqrt_auxviii_o:nnnnnnnn #1#2 #3#4#5#6#7
19581 {
19582 \exp_after:wN __fp_sqrt_auxix_o:wnwnw
19583 \int_value:w __fp_int_eval:w #3
19584 \exp_after:wN __fp_basics_pack_low:NNNNNw
19585 \int_value:w __fp_int_eval:w #1 + 1#4#5
19586 \exp_after:wN __fp_basics_pack_low:NNNNNw
19587 \int_value:w __fp_int_eval:w #2 + 1#6#7 ;
19588 }
19589 \cs_new:Npn __fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
19590 {
19591 __fp_sqrt_auxii_o:Nnnnnnnn
19592 __fp_sqrt_auxiii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5}
19593 }

```

(End definition for  $\_fp_sqrt_auxviii_o:nnnnnnnn$  and  $\_fp_sqrt_auxix_o:wnwnw$ .)

$\_fp_sqrt_auxxx_o:Nnnnnnnn$   
 $\_fp_sqrt_auxxi_o:wwnnN$

At this stage,  $j = 6$  and  $10^{24}z < 10^7$ , hence

$$10^7 + 1/2 > 10^{24}z + 1/2 \geq (10^{24}(a - y^2) - 258) \cdot (0.5 \cdot 10^8) / (10^8y + 1),$$

then  $10^{24}(a - y^2) - 258 < 2(10^7 + 1/2)(y + 10^{-8})$ , and

$$10^{24}(a - y^2) < (10^7 + 1290.5)(1 + 10^{-8}/y)(2y) < (10^7 + 1290.5)(1 + 10^{-7})(y + \sqrt{a}),$$

which finally implies  $0 \leq \sqrt{a} - y < 0.2 \cdot 10^{-16}$ . In particular,  $y$  is an underestimate of  $\sqrt{a}$  and  $y + 0.5 \cdot 10^{-16}$  is a (strict) overestimate. There is at exactly one multiple  $m$  of  $0.5 \cdot 10^{-16}$  in the interval  $[y, y + 0.5 \cdot 10^{-16}]$ . If  $m^2 > a$ , then the square root is inexact and

is obtained by rounding  $m - \epsilon$  to a multiple of  $10^{-16}$  (the precise shift  $0 < \epsilon < 0.5 \cdot 10^{-16}$  is irrelevant for rounding). If  $m^2 = a$  then the square root is exactly  $m$ , and there is no rounding. If  $m^2 < a$  then we round  $m + \epsilon$ . For now, discard a few irrelevant arguments #1, #2, #3, and find the multiple of  $0.5 \cdot 10^{-16}$  within  $[y, y + 0.5 \cdot 10^{-16}]$ ; rather, only the last 4 digits #8 of  $y$  are considered, and we do not perform any carry yet. The auxxi auxiliary sets up auxii with a continuation function auxxii instead of auxiii as before. To prevent auxii from giving a negative results  $a - m^2$ , we compute  $a + 10^{-16} - m^2$  instead, always positive since  $m < \sqrt{a} + 0.5 \cdot 10^{-16}$  and  $a \leq 1 - 10^{-16}$ .

```

19594 \cs_new:Npn _fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
19595 {
19596 \exp_after:wN _fp_sqrt_auxxi_o:wwnnN
19597 \int_value:w _fp_int_eval:w
19598 (#8 + 2499) / 5000 * 5000 ;
19599 {#4} {#5} {#6} {#7} ;
19600 }
19601 \cs_new:Npn _fp_sqrt_auxxi_o:wwnnN #1; #2; #3#4#5
19602 {
19603 _fp_sqrt_auxii_o:NnnnnnnnN
19604 _fp_sqrt_auxxii_o:nnnnnnnnw
19605 #2 {#1}
19606 {#3} {#4 + 1 } #5
19607 }

```

(End definition for `\_fp_sqrt_auxx_o:Nnnnnnnn` and `\_fp_sqrt_auxxi_o:wwnnN`.)

The difference  $0 \leq a + 10^{-16} - m^2 \leq 10^{-16} + (\sqrt{a} - m)(\sqrt{a} + m) \leq 2 \cdot 10^{-16}$  was just computed: its first 8 digits vanish, as do the next four, #1, and most of the following four, #2. The guess  $m$  is an overestimate if  $a + 10^{-16} - m^2 < 10^{-16}$ , that is, #1#2 vanishes. Otherwise it is an underestimate, unless  $a + 10^{-16} - m^2 = 10^{-16}$  exactly. For an underestimate, call the auxxiv function with argument 9998. For an exact result call it with 9999, and for an overestimate call it with 10000.

```

19608 \cs_new:Npn _fp_sqrt_auxxi_o:nnnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
19609 {
19610 \if_int_compare:w #1#2 > 0 \exp_stop_f:
19611 \if_int_compare:w #1#2 = 1 \exp_stop_f:
19612 \if_int_compare:w #3#4 = 0 \exp_stop_f:
19613 \if_int_compare:w #5#6 = 0 \exp_stop_f:
19614 \if_int_compare:w #7#8 = 0 \exp_stop_f:
19615 _fp_sqrt_auxxii_o:w
19616 \fi:
19617 \fi:
19618 \fi:
19619 \fi:
19620 \exp_after:wN _fp_sqrt_auxxiv_o:wnnnnnnnN
19621 \int_value:w 9998
19622 \else:
19623 \exp_after:wN _fp_sqrt_auxxiv_o:wnnnnnnnN
19624 \int_value:w 10000
19625 \fi:
19626 ;
19627 }
19628 \cs_new:Npn _fp_sqrt_auxxii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
19629 {

```

```

19630 \fi: \fi: \fi: \fi: \fi:
19631 __fp_sqrt_auxxiv_o:wnnnnnnnN 9999 ;
19632 }

```

(End definition for `\_\_fp_sqrt_auxxiv_o:nnnnnnnnw` and `\_\_fp_sqrt_auxxiii_o:w`.)

`\_\_fp_sqrt_auxxiv_o:wnnnnnnnN`

This receives 9998, 9999 or 10000 as #1 when  $m$  is an underestimate, exact, or an overestimate, respectively. Then comes  $m$  as five blocks of 4 digits, but where the last block #6 may be 0, 5000, or 10000. In the latter case, we need to add a carry, unless  $m$  is an overestimate (#1 is then 10000). Then comes  $a$  as three arguments. Rounding is done by `\_\_fp_round:NNN`, whose first argument is the final sign 0 (square roots are positive). We fake its second argument. It should be the last digit kept, but this is only used when ties are “rounded to even”, and only when the result is exactly half-way between two representable numbers rational square roots of numbers with 16 significant digits have: this situation never arises for the square root, as any exact square root of a 16 digit number has at most 8 significant digits. Finally, the last argument is the next digit, possibly shifted by 1 when there are further nonzero digits. This is achieved by `\_\_fp_round_digit:Nw`, which receives (after removal of the 10000’s digit) one of 0000, 0001, 4999, 5000, 5001, or 9999, which it converts to 0, 1, 4, 5, 6, and 9, respectively.

```

19633 \cs_new:Npn __fp_sqrt_auxxiv_o:wnnnnnnnN #1; #2#3#4#5#6 #7#8#9
19634 {
19635 \exp_after:wN __fp_basics_pack_high:NNNNNw
19636 \int_value:w __fp_int_eval:w 1 0000 0000 + #2#3
19637 \exp_after:wN __fp_basics_pack_low:NNNNNw
19638 \int_value:w __fp_int_eval:w 1 0000 0000
19639 + #4#5
19640 \if_int_compare:w #6 > #1 \exp_stop_f: + 1 \fi:
19641 + \exp_after:wN __fp_round:NNN
19642 \exp_after:wN 0
19643 \exp_after:wN 0
19644 \int_value:w
19645 \exp_after:wN \use_i:nn
19646 \exp_after:wN __fp_round_digit:Nw
19647 \int_value:w __fp_int_eval:w #6 + 19999 - #1 ;
19648 \exp_after:wN ;
19649 }

```

(End definition for `\_\_fp_sqrt_auxxiv_o:wnnnnnnnN`.)

### 31.5 About the sign and exponent

`\_\_fp_logb_o:w` The exponent of a normal number is its  $\langle\text{exponent}\rangle$  minus one.

```

19650 \cs_new:Npn __fp_logb_o:w ? \s_fp __fp_chk:w #1#2; @
19651 {
19652 \if_case:w #1 \exp_stop_f:
19653 __fp_case_use:nw
19654 { __fp_division_by_zero_o:Nnw \c_minus_inf_fp { logb } }
19655 \or: \exp_after:wN __fp_logb_aux_o:w
19656 \or: __fp_case_return_o:Nw \c_inf_fp
19657 \else: __fp_case_return_same_o:w
19658 \fi:
19659 \s_fp __fp_chk:w #1 #2;
19660 }

```

```

19661 \cs_new:Npn __fp_logb_aux_o:w \s__fp __fp_chk:w #1 #2 #3 #4 ;
19662 {
19663 \exp_after:wN __fp_parse:n \exp_after:wN
19664 { \int_value:w \int_eval:w #3 - 1 \exp_after:wN }
19665 }

```

(End definition for `\__fp_logb_o:w` and `\__fp_logb_aux_o:w`.)

`\__fp_sign_o:w` Find the sign of the floating point: `nan`, `+0`, `-0`, `+1` or `-1`.

```

19666 \cs_new:Npn __fp_sign_o:w ? \s__fp __fp_chk:w #1#2; @
19667 {
19668 \if_case:w #1 \exp_stop_f:
19669 __fp_case_return_same_o:w
19670 \or: \exp_after:wN __fp_sign_aux_o:w
19671 \or: \exp_after:wN __fp_sign_aux_o:w
19672 \else: __fp_case_return_same_o:w
19673 \fi:
19674 \s__fp __fp_chk:w #1 #2;
19675 }
19676 \cs_new:Npn __fp_sign_aux_o:w \s__fp __fp_chk:w #1 #2 #3 ;
19677 { \exp_after:wN __fp_set_sign_o:w \exp_after:wN #2 \c_one_fp @ }

```

(End definition for `\__fp_sign_o:w` and `\__fp_sign_aux_o:w`.)

`\__fp_set_sign_o:w` This function is used for the unary minus and for `abs`. It leaves the sign of `nan` invariant, turns negative numbers (sign 2) to positive numbers (sign 0) and positive numbers (sign 0) to positive or negative numbers depending on `#1`. It also expands after itself in the input stream, just like `\__fp+_o:ww`.

```

19678 \cs_new:Npn __fp_set_sign_o:w #1 \s__fp __fp_chk:w #2#3#4; @
19679 {
19680 \exp_after:wN __fp_exp_after_o:w
19681 \exp_after:wN \s__fp
19682 \exp_after:wN __fp_chk:w
19683 \exp_after:wN #2
19684 \int_value:w
19685 \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
19686 #4;
19687 }

```

(End definition for `\__fp_set_sign_o:w`.)

### 31.6 Operations on tuples

Two cases: `abs(<tuple>)` for which `#1` is 0 (invalid for tuples) and `-<tuple>` for which `#1` is 2. In that case, map over all items in the tuple an auxiliary that dispatches to the type-appropriate sign-flipping function.

```

19688 \cs_new:Npn __fp_tuple_set_sign_o:w #1
19689 {
19690 \if_meaning:w 2 #1
19691 \exp_after:wN __fp_tuple_set_sign_aux_o:Nnw
19692 \fi:
19693 __fp_invalid_operation_o:nw { abs }
19694 }
19695 \cs_new:Npn __fp_tuple_set_sign_aux_o:Nnw #1#2#3 @

```

```

19696 { __fp_tuple_map_o:nw __fp_tuple_set_sign_aux_o:w #3 }
19697 \cs_new:Npn __fp_tuple_set_sign_aux_o:w #1#2 ;
19698 {
19699 __fp_change_func_type:NNN #1 __fp_set_sign_o:w
19700 __fp_parse_apply_unary_error:NNw
19701 2 #1 #2 ; @
19702 }
```

(End definition for `\__fp_tuple_set_sign_o:w`, `\__fp_tuple_set_sign_aux_o:Nnw`, and `\__fp_tuple_set_sign_aux_o:w`.)

`\__fp_*_tuple_o:ww`  
`\__fp_tuple_*_o:ww`  
`\__fp_tuple/_o:ww`

For  $\langle\text{number}\rangle * \langle\text{tuple}\rangle$  and  $\langle\text{tuple}\rangle * \langle\text{number}\rangle$  and  $\langle\text{tuple}\rangle / \langle\text{number}\rangle$ , loop through the  $\langle\text{tuple}\rangle$  some code that multiplies or divides by the appropriate  $\langle\text{number}\rangle$ . Importantly we need to dispatch according to the type, and we make sure to apply the operator in the correct order.

```

19703 \cs_new:cpn { __fp_*_tuple_o:ww } #1 ;
19704 { __fp_tuple_map_o:nw { __fp_binary_type_o:Nww * #1 ; } }
19705 \cs_new:cpn { __fp_tuple_*_o:ww } #1 ; #2 ;
19706 { __fp_tuple_map_o:nw { __fp_binary_rev_type_o:Nww * #2 ; } #1 ; }
19707 \cs_new:cpn { __fp_tuple/_o:ww } #1 ; #2 ;
19708 { __fp_tuple_map_o:nw { __fp_binary_rev_type_o:Nww / #2 ; } #1 ; }
```

(End definition for `\__fp_*_tuple_o:ww`, `\__fp_tuple_*_o:ww`, and `\__fp_tuple/_o:ww`.)

`\__fp_tuple+_tuple_o:ww`  
`\__fp_tuple-_tuple_o:ww`

Check the two tuples have the same number of items and map through these a helper that dispatches appropriately depending on the types. This means  $(1,2) + ((1,1),2)$  gives  $(\text{nan}, 4)$ .

```

19709 \cs_set_protected:Npn __fp_tmp:w #1
19710 {
19711 \cs_new:cpn { __fp_tuple_#1_tuple_o:ww }
19712 \s__fp_tuple __fp_tuple_chk:w ##1 ;
19713 \s__fp_tuple __fp_tuple_chk:w ##2 ;
19714 {
19715 \int_compare:nNnTF
19716 { __fp_array_count:n {##1} } = { __fp_array_count:n {##2} }
19717 { __fp_tuple_mapthread_o:nww { __fp_binary_type_o:Nww #1 } }
19718 { __fp_invalid_operation_o:Nww #1 }
19719 \s__fp_tuple __fp_tuple_chk:w {##1} ;
19720 \s__fp_tuple __fp_tuple_chk:w {##2} ;
19721 }
19722 }
19723 __fp_tmp:w +
19724 __fp_tmp:w -
```

(End definition for `\__fp_tuple+_tuple_o:ww` and `\__fp_tuple-_tuple_o:ww`.)

19725 `</package>`

## 32 I3fp-extended implementation

```

19726 <*package>
19727 <@=fp>
```

## 32.1 Description of fixed point numbers

This module provides a few functions to manipulate positive floating point numbers with extended precision (24 digits), but mostly provides functions for fixed-point numbers with this precision (24 digits). Those are used in the computation of Taylor series for the logarithm, exponential, and trigonometric functions. Since we eventually only care about the 16 first digits of the final result, some of the calculations are not performed with the full 24-digit precision. In other words, the last two blocks of each fixed point number may be wrong as long as the error is small enough to be rounded away when converting back to a floating point number. The fixed point numbers are expressed as

```
{⟨a1⟩} {⟨a2⟩} {⟨a3⟩} {⟨a4⟩} {⟨a5⟩} {⟨a6⟩} ;
```

where each ⟨a<sub>i</sub>⟩ is exactly 4 digits (ranging from 0000 to 9999), except ⟨a<sub>1</sub>⟩, which may be any “not-too-large” non-negative integer, with or without leading zeros. Here, “not-too-large” depends on the specific function (see the corresponding comments for details). Checking for overflow is the responsibility of the code calling those functions. The fixed point number  $a$  corresponding to the representation above is  $a = \sum_{i=1}^6 \langle a_i \rangle \cdot 10^{-4i}$ .

Most functions we define here have the form

```
__fp_fixed_⟨calculation⟩:wwn ⟨operand1⟩ ; ⟨operand2⟩ ; {⟨continuation⟩}
```

They perform the ⟨calculation⟩ on the two ⟨operands⟩, then feed the result (6 brace groups followed by a semicolon) to the ⟨continuation⟩, responsible for the next step of the calculation. Some functions only accept an N-type ⟨continuation⟩. This allows constructions such as

```
__fp_fixed_add:wwn ⟨X1⟩ ; ⟨X2⟩ ;
__fp_fixed_mul:wwn ⟨X3⟩ ;
__fp_fixed_add:wwn ⟨X4⟩ ;
```

to compute  $(X_1 + X_2) \cdot X_3 + X_4$ . This turns out to be very appropriate for computing continued fractions and Taylor series.

At the end of the calculation, the result is turned back to a floating point number using \\_\\_fp\\_fixed\\_to\\_float\\_o:wN. This function has to change the exponent of the floating point number: it must be used after starting an integer expression for the overall exponent of the result.

## 32.2 Helpers for numbers with extended precision

\c\\_fp\\_one\\_fixed\\_t1 The fixed-point number 1, used in l3fp-expo.

```
19728 \t1_const:Nn \c_fp_one_fixed_t1
19729 { {10000} {0000} {0000} {0000} {0000} {0000} ; }
```

(End definition for \c\\_fp\\_one\\_fixed\\_t1.)

\\_\\_fp\\_fixed\\_continue:wn This function simply calls the next function.

```
19730 \cs_new:Npn __fp_fixed_continue:wn #1; #2 { #2 #1; }
```

(End definition for \\_\\_fp\\_fixed\\_continue:wn.)

```
__fp_fixed_add_one:wN __fp_fixed_add_one:wN $\langle a \rangle ; \langle continuation \rangle$
This function adds 1 to the fixed point $\langle a \rangle$, by changing a_1 to $10000 + a_1$, then calls
the $\langle continuation \rangle$. This requires <math>a_1 + 10000 < 2^{31}</math>.

19731 \cs_new:Npn __fp_fixed_add_one:wN #1#2; #3
19732 {
19733 \exp_after:wN #3 \exp_after:wN
19734 { \int_value:w __fp_int_eval:w \c_fp_myriad_int + #1 } #2 ;
19735 }
```

(End definition for `\_\_fp\_fixed\_add\_one:wN`.)

`\_\_fp\_fixed\_div\_myriad:wn` Divide a fixed point number by 10000. This is a little bit more subtle than just removing the last group and adding a leading group of zeros: the first group `#1` may have any number of digits, and we must split `#1` into the new first group and a second group of exactly 4 digits. The choice of shifts allows `#1` to be in the range  $[0, 5 \cdot 10^8 - 1]$ .

```
19736 \cs_new:Npn __fp_fixed_div_myriad:wn #1#2#3#4#5#6;
19737 {
19738 \exp_after:wN __fp_fixed_mul_after:wwn
19739 \int_value:w __fp_int_eval:w \c_fp_leading_shift_int
19740 \exp_after:wN __fp_pack:NNNNw
19741 \int_value:w __fp_int_eval:w \c_fp_trailing_shift_int
19742 + #1 ; {#2}{#3}{#4}{#5} ;
19743 }
```

(End definition for `\_\_fp\_fixed\_div\_myriad:wn`.)

`\_\_fp\_fixed\_mul\_after:wwn` The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the  $\langle continuation \rangle$  `#3` in front.

```
19744 \cs_new:Npn __fp_fixed_mul_after:wwn #1; #2; #3 { #3 {#1} #2; }
```

(End definition for `\_\_fp\_fixed\_mul\_after:wwn`.)

### 32.3 Multiplying a fixed point number by a short one

```
__fp_fixed_mul_short:wwn __fp_fixed_mul_short:wwn
{$\langle a_1 \rangle \langle a_2 \rangle \langle a_3 \rangle \langle a_4 \rangle \langle a_5 \rangle \langle a_6 \rangle ; \langle continuation \rangle$
{$\langle b_0 \rangle \langle b_1 \rangle \langle b_2 \rangle ; \langle continuation \rangle$}
Computes the product $c = ab$ of $a = \sum_i \langle a_i \rangle 10^{-4i}$ and $b = \sum_i \langle b_i \rangle 10^{-4i}$, rounds
it to the closest multiple of 10^{-24} , and leaves $\langle continuation \rangle \langle c_1 \rangle \dots \langle c_6 \rangle$; in
the input stream, where each of the $\langle c_i \rangle$ are blocks of 4 digits, except $\langle c_1 \rangle$, which is
any TeX integer. Note that indices for $\langle b \rangle$ start at 0: for instance a second operand of
{0001}{0000}{0000} leaves the first operand unchanged (rather than dividing it by 10^4 ,
as __fp_fixed_mul:wwn would).
```

```
19745 \cs_new:Npn __fp_fixed_mul_short:wwn #1#2#3#4#5#6; #7#8#9;
19746 {
19747 \exp_after:wN __fp_fixed_mul_after:wwn
19748 \int_value:w __fp_int_eval:w \c_fp_leading_shift_int
19749 + #1#7
19750 \exp_after:wN __fp_pack:NNNNw
19751 \int_value:w __fp_int_eval:w \c_fp_middle_shift_int
19752 + #1#8 + #2#7
19753 \exp_after:wN __fp_pack:NNNNw
19754 \int_value:w __fp_int_eval:w \c_fp_middle_shift_int
```

```

19755 + #1*#9 + #2*#8 + #3*#7
19756 \exp_after:wN _fp_pack:NNNNNw
19757 \int_value:w _fp_int_eval:w \c_fp_middle_shift_int
19758 + #2*#9 + #3*#8 + #4*#7
19759 \exp_after:wN _fp_pack:NNNNNw
19760 \int_value:w _fp_int_eval:w \c_fp_middle_shift_int
19761 + #3*#9 + #4*#8 + #5*#7
19762 \exp_after:wN _fp_pack:NNNNNw
19763 \int_value:w _fp_int_eval:w \c_fp_trailing_shift_int
19764 + #4*#9 + #5*#8 + #6*#7
19765 + (#5*#9 + #6*#8 + #6*#9 / \c_fp_myriad_int)
19766 / \c_fp_myriad_int ; ;
19767 }

```

(End definition for `\_fp_fixed_mul_short:wnn`.)

### 32.4 Dividing a fixed point number by a small integer

```

\fp_fixed_div_int:wwN
\fp_fixed_div_int:wnN
\fp_fixed_div_int_auxi:wnn
\fp_fixed_div_int_auxii:wnn
\fp_fixed_div_int_pack:Nw
\fp_fixed_div_int_after:Nw

```

`\_fp_fixed_div_int:wwN`  $\langle a \rangle$  ;  $\langle n \rangle$  ;  $\langle \text{continuation} \rangle$

Divides the fixed point number  $\langle a \rangle$  by the (small) integer  $0 < \langle n \rangle < 10^4$  and feeds the result to the  $\langle \text{continuation} \rangle$ . There is no bound on  $a_1$ .

The arguments of the `i` auxiliary are 1: one of the  $a_i$ , 2:  $n$ , 3: the `ii` or the `iii` auxiliary. It computes a (somewhat tight) lower bound  $Q_i$  for the ratio  $a_i/n$ .

The `ii` auxiliary receives  $Q_i$ ,  $n$ , and  $a_i$  as arguments. It adds  $Q_i$  to a surrounding integer expression, and starts a new one with the initial value 9999, which ensures that the result of this expression has 5 digits. The auxiliary also computes  $a_i - n \cdot Q_i$ , placing the result in front of the 4 digits of  $a_{i+1}$ . The resulting  $a'_{i+1} = 10^4(a_i - n \cdot Q_i) + a_{i+1}$  serves as the first argument for a new call to the `i` auxiliary.

When the `iii` auxiliary is called, the situation looks like this:

```

\fp_fixed_div_int_after:Nw $\langle \text{continuation} \rangle$
-1 + Q_1
\fp_fixed_div_int_pack:Nw 9999 + Q_2
\fp_fixed_div_int_pack:Nw 9999 + Q_3
\fp_fixed_div_int_pack:Nw 9999 + Q_4
\fp_fixed_div_int_pack:Nw 9999 + Q_5
\fp_fixed_div_int_pack:Nw 9999
\fp_fixed_div_int_auxii:wnn Q_6 ; { $\langle n \rangle$ } { $\langle a_6 \rangle$ }

```

where expansion is happening from the last line up. The `iii` auxiliary adds  $Q_6 + 2 \simeq a_6/n + 1$  to the last 9999, giving the integer closest to  $10000 + a_6/n$ .

Each `pack` auxiliary receives 5 digits followed by a semicolon. The first digit is added as a carry to the integer expression above, and the 4 other digits are braced. Each call to the `pack` auxiliary thus produces one brace group. The last brace group is produced by the `after` auxiliary, which places the  $\langle \text{continuation} \rangle$  as appropriate.

```

19768 \cs_new:Npn \fp_fixed_div_int:wwN #1#2#3#4#5#6 ; #7 ; #8
19769 {
19770 \exp_after:wN \fp_fixed_div_int_after:Nw
19771 \exp_after:wN #8
19772 \int_value:w \fp_int_eval:w - 1
19773 \fp_fixed_div_int:wnN
19774 #1; {#7} \fp_fixed_div_int_auxi:wnn

```

```

19775 #2; {#7} __fp_fixed_div_int_auxi:wnn
19776 #3; {#7} __fp_fixed_div_int_auxi:wnn
19777 #4; {#7} __fp_fixed_div_int_auxi:wnn
19778 #5; {#7} __fp_fixed_div_int_auxi:wnn
19779 #6; {#7} __fp_fixed_div_int_auxii:wnn ;
19780 }
19781 \cs_new:Npn __fp_fixed_div_int:wnN #1; #2 #3
19782 {
19783 \exp_after:wN #3
19784 \int_value:w __fp_int_eval:w #1 / #2 - 1 ;
19785 {#2}
19786 {#1}
19787 }
19788 \cs_new:Npn __fp_fixed_div_int_auxi:wnn #1; #2 #3
19789 {
19790 +
19791 \exp_after:wN __fp_fixed_div_int_pack:Nw
19792 \int_value:w __fp_int_eval:w 9999
19793 \exp_after:wN __fp_fixed_div_int:wnN
19794 \int_value:w __fp_int_eval:w #3 - #1*#2 __fp_int_eval_end:
19795 }
19796 \cs_new:Npn __fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + 2 ; }
19797 \cs_new:Npn __fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
19798 \cs_new:Npn __fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }

```

(End definition for `\__fp_fixed_div_int:wwN` and others.)

### 32.5 Adding and subtracting fixed points

```

__fp_fixed_add:wwn
__fp_fixed_sub:wwn
__fp_fixed_add:Nnnnnwnn
__fp_fixed_add:nnNnnnwN
__fp_fixed_add_pack:NNNNNwn
__fp_fixed_add_after:NNNNNwn

```

`\__fp_fixed_add:wwn`  $\langle a \rangle ; \langle b \rangle ; \langle continuation \rangle$

Computes  $a + b$  (resp.  $a - b$ ) and feeds the result to the  $\langle continuation \rangle$ . This function requires  $0 \leq a_1, b_1 \leq 114748$ , its result must be positive (this happens automatically for addition) and its first group must have at most 5 digits:  $(a \pm b)_1 < 100000$ . The two functions only differ by a sign, hence use a common auxiliary. It would be nice to grab the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign,  $a_1, \dots, a_4$ , the rest of  $a$ , and  $b_1$  and  $b_2$ . The second auxiliary receives the rest of  $a$ , the sign multiplying  $b$ , the rest of  $b$ , and the  $\langle continuation \rangle$  as arguments. After going down through the various level, we go back up, packing digits and bringing the  $\langle continuation \rangle$  (#8, then #7) from the end of the argument list to its start.

```

19799 \cs_new:Npn __fp_fixed_add:wwn { __fp_fixed_add:Nnnnnwnn + }
19800 \cs_new:Npn __fp_fixed_sub:wwn { __fp_fixed_add:Nnnnnwnn - }
19801 \cs_new:Npn __fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
19802 {
19803 \exp_after:wN __fp_fixed_add_after:NNNNNwn
19804 \int_value:w __fp_int_eval:w 9 9999 9998 + #2#3 #1 #7#8
19805 \exp_after:wN __fp_fixed_add_pack:NNNNNwn
19806 \int_value:w __fp_int_eval:w 1 9999 9998 + #4#5
19807 __fp_fixed_add:nnNnnnwN #6 #1
19808 }
19809 \cs_new:Npn __fp_fixed_add:nnNnnnwN #1#2 #3 #4#5 #6#7 ; #8
19810 {
19811 #3 #4#5
19812 \exp_after:wN __fp_fixed_add_pack:NNNNNwn

```

```

19813 \int_value:w _fp_int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
19814 }
19815 \cs_new:Npn _fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
19816 { + #1 ; {#7} {#2#3#4#5} {#6} }
19817 \cs_new:Npn _fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
19818 { #7 {#1#2#3#4#5} {#6} }

```

(End definition for `\_fp_fixed_add:wwn` and others.)

### 32.6 Multiplying fixed points

```
_fp_fixed_mul:wwn
_fp_fixed_mul:nnnnnnnw
```

`\_fp_fixed_mul:wwn`  $\langle a \rangle ; \langle b \rangle ; \langle \text{continuation} \rangle$

Computes  $a \times b$  and feeds the result to  $\langle \text{continuation} \rangle$ . This function requires  $0 \leq a_1, b_1 < 10000$ . Once more, we need to play around the limit of 9 arguments for T<sub>E</sub>X macros. Note that we don't need to obtain an exact rounding, contrarily to the `*` operator, so things could be harder. We wish to perform carries in

$$\begin{aligned}
a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left( a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
& \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
& \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 \right) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}$$

where the  $O(10^{-24})$  stands for terms which are at most  $5 \cdot 10^{-24}$ ; ignoring those leads to an error of at most 5 ulp. Note how the first 15 terms only depend on  $a_1, \dots, a_4$  and  $b_1, \dots, b_4$ , while the last 6 terms only depend on  $a_1, a_2, a_5, a_6$ , and the corresponding parts of  $b$ . Hence, the first function grabs  $a_1, \dots, a_4$ , the rest of  $a$ , and  $b_1, \dots, b_4$ , and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The `i` auxiliary receives  $a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6$  and finally the  $\langle \text{continuation} \rangle$  as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The  $\langle \text{continuation} \rangle$  is finally placed in front of the 6 brace groups by `\_fp_fixed_mul_after:wwn`.

```

19819 \cs_new:Npn _fp_fixed_mul:wwn #1#2#3#4 #5; #6#7#8#9
19820 {
19821 \exp_after:wN _fp_fixed_mul_after:wwn
19822 \int_value:w _fp_int_eval:w \c_fp_leading_shift_int
19823 \exp_after:wN _fp_pack:NNNNNw
19824 \int_value:w _fp_int_eval:w \c_fp_middle_shift_int
19825 + #1##6
19826 \exp_after:wN _fp_pack:NNNNNw
19827 \int_value:w _fp_int_eval:w \c_fp_middle_shift_int
19828 + #1##7 + #2##6
19829 \exp_after:wN _fp_pack:NNNNNw
19830 \int_value:w _fp_int_eval:w \c_fp_middle_shift_int
19831 + #1##8 + #2##7 + #3##6
19832 \exp_after:wN _fp_pack:NNNNNw

```

```

19833 \int_value:w _fp_int_eval:w \c_fp_middle_shift_int
19834 + #1#9 + #2#8 + #3#7 + #4#6
19835 \exp_after:wN _fp_pack:NNNNNw
19836 \int_value:w _fp_int_eval:w \c_fp_trailing_shift_int
19837 + #2#9 + #3#8 + #4#7
19838 + (#3#9 + #4#8
19839 + \fp_fixed_mul:nnnnnnnw #5 {#6}{#7} {#1}{#2}
19840 }
19841 \cs_new:Npn \fp_fixed_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
19842 {
19843 #1#4 + #2#3 + #5#8 + #6#7) / \c_myriad_int
19844 + #1#3 + #5#7 ; ;
19845 }

```

(End definition for `\fp_fixed_mul:wwn` and `\fp_fixed_mul:nnnnnnnw`.)

### 32.7 Combining product and sum of fixed points

```

\fp_fixed_mul_add:wwn ⟨a⟩ ; ⟨b⟩ ; ⟨c⟩ ; {⟨continuation⟩}
\fp_fixed_mul_sub_back:wwn ⟨a⟩ ; ⟨b⟩ ; ⟨c⟩ ; {⟨continuation⟩}
\fp_fixed_one_minus_mul:wwn ⟨a⟩ ; ⟨b⟩ ; {⟨continuation⟩}

```

Sometimes called FMA (fused multiply-add), these functions compute  $a \times b + c$ ,  $c - a \times b$ , and  $1 - a \times b$  and feed the result to the `{⟨continuation⟩}`. Those functions require  $0 \leq a_1, b_1, c_1 \leq 10000$ . Since those functions are at the heart of the computation of Taylor expansions, we over-optimize them a bit, and in particular we do not factor out the common parts of the three functions.

For definiteness, consider the task of computing  $a \times b + c$ . We perform carries in

$$\begin{aligned}
a \times b + c = & (a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left( a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
& \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
& \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 + c_5 c_6 \right) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}$$

where  $c_1 c_2$ ,  $c_3 c_4$ ,  $c_5 c_6$  denote the 8-digit number obtained by juxtaposing the two blocks of digits of  $c$ , and  $\cdot$  denotes multiplication. The task is obviously tough because we have 18 brace groups in front of us.

Each of the three function starts the first two levels (the first, corresponding to  $10^{-4}$ , is empty), with  $c_1 c_2$  in the first level, calls the `i` auxiliary with arguments described later, and adds a trailing `+ c5 c6 ; {⟨continuation⟩} ;`. The `+ c5 c6` piece, which is omitted for `\fp_fixed_one_minus_mul:wwn`, is taken in the integer expression for the  $10^{-24}$  level.

```

19846 \cs_new:Npn \fp_fixed_mul_add:wwn #1; #2; #3#4#5#6#7#8;
19847 {
19848 \exp_after:wN \fp_fixed_mul_after:wwn
19849 \int_value:w \fp_int_eval:w \c_fp_big_leading_shift_int
19850 \exp_after:wN \fp_pack_big:NNNNNw

```

```

19851 \int_value:w _fp_int_eval:w \c_fp_big_middle_shift_int + #3 #4
19852 _fp_fixed_mul_add:Nwnnnnnnn +
19853 + #5 #6 ; #2 ; #1 ; #2 ; +
19854 + #7 #8 ; ;
19855 }
19856 \cs_new:Npn _fp_fixed_mul_sub_back:wwn #1; #2; #3#4#5#6#7#8;
19857 {
19858 \exp_after:wN _fp_fixed_mul_after:wwn
19859 \int_value:w _fp_int_eval:w \c_fp_big_leading_shift_int
19860 \exp_after:wN _fp_pack_big:NNNNNNw
19861 \int_value:w _fp_int_eval:w \c_fp_big_middle_shift_int + #3 #4
19862 _fp_fixed_mul_add:Nwnnnnnnn -
19863 + #5 #6 ; #2 ; #1 ; #2 ; -
19864 + #7 #8 ; ;
19865 }
19866 \cs_new:Npn _fp_fixed_one_minus_mul:wwn #1; #2;
19867 {
19868 \exp_after:wN _fp_fixed_mul_after:wwn
19869 \int_value:w _fp_int_eval:w \c_fp_big_leading_shift_int
19870 \exp_after:wN _fp_pack_big:NNNNNNw
19871 \int_value:w _fp_int_eval:w \c_fp_big_middle_shift_int +
19872 1 0000 0000
19873 _fp_fixed_mul_add:Nwnnnnnnn -
19874 ; #2 ; #1 ; #2 ; -
19875 ; ;
19876 }

```

(End definition for `\_fp_fixed_mul_add:wwn`, `\_fp_fixed_mul_sub_back:wwn`, and `\_fp_fixed_one_minus_mul:wwn`.)

```

_fp_fixed_mul_add:Nwnnnnnnn ⟨op⟩ + ⟨c3⟩ ⟨c4⟩ ;
⟨b⟩ ; ⟨a⟩ ; ⟨b⟩ ; ⟨op⟩
+ ⟨c5⟩ ⟨c6⟩ ;

```

Here,  $\langle op \rangle$  is either  $+$  or  $-$ . Arguments #3, #4, #5 are  $\langle b_1 \rangle$ ,  $\langle b_2 \rangle$ ,  $\langle b_3 \rangle$ ; arguments #7, #8, #9 are  $\langle a_1 \rangle$ ,  $\langle a_2 \rangle$ ,  $\langle a_3 \rangle$ . We can build three levels:  $a_1 \cdot b_1$  for  $10^{-8}$ ,  $(a_1 \cdot b_2 + a_2 \cdot b_1)$  for  $10^{-12}$ , and  $(a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4)$  for  $10^{-16}$ . The  $a \cdot b$  products use the sign #1. Note that #2 is empty for `\_fp_fixed_one_minus_mul:wwn`. We call the `ii` auxiliary for levels  $10^{-20}$  and  $10^{-24}$ , keeping the pieces of  $\langle a \rangle$  we've read, but not  $\langle b \rangle$ , since there is another copy later in the input stream.

```

19877 \cs_new:Npn _fp_fixed_mul_add:Nwnnnnnnn #1 #2; #3#4#5#6; #7#8#9
19878 {
19879 #1 #7*#3
19880 \exp_after:wN _fp_pack_big:NNNNNNw
19881 \int_value:w _fp_int_eval:w \c_fp_big_middle_shift_int
19882 #1 #7*#4 #1 #8*#3
19883 \exp_after:wN _fp_pack_big:NNNNNNw
19884 \int_value:w _fp_int_eval:w \c_fp_big_middle_shift_int
19885 #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
19886 \exp_after:wN _fp_pack_big:NNNNNNw
19887 \int_value:w _fp_int_eval:w \c_fp_big_middle_shift_int
19888 #1 _fp_fixed_mul_add:nnnnnnnn {#7}{#8}{#9}
19889 }

```

(End definition for `\_fp_fixed_mul_add:Nwnnnnnnn`.)

```
__fp_fixed_mul_add:nnnnwnnnn \langle a\rangle ; \langle b\rangle ; \langle op\rangle
+ \langle c_5\rangle \langle c_6\rangle ;
```

Level  $10^{-20}$  is  $(a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1)$ , multiplied by the sign, which was inserted by the *i* auxiliary. Then we prepare level  $10^{-24}$ . We don't have access to all parts of  $\langle a\rangle$  and  $\langle b\rangle$  needed to make all products. Instead, we prepare the partial expressions

$$\begin{aligned} & b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1 \\ & b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2. \end{aligned}$$

Obviously, those expressions make no mathematical sense: we complete them with  $a_5 \cdot$  and  $\cdot b_5$ , and with  $a_6 \cdot b_1 + a_5 \cdot$  and  $\cdot b_5 + a_1 \cdot b_6$ , and of course with the trailing  $+ c_5 c_6$ . To do all this, we keep  $a_1, a_5, a_6$ , and the corresponding pieces of  $\langle b\rangle$ .

```
19890 \cs_new:Npn __fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
19891 {
19892 (#1##9 + #2##8 + #3##7 + #4##6)
19893 \exp_after:wN __fp_pack_big:NNNNNNw
19894 \int_value:w __fp_int_eval:w \c_fp_big_trailing_shift_int
19895 __fp_fixed_mul_add:nnnnwnnnN
19896 { #6 + #4##7 + #3##8 + #2##9 + #1 }
19897 { #7 + #4##8 + #3##9 + #2 }
19898 {#1} #5;
19899 {#6}
19900 }
```

(End definition for `\_\_fp\_fixed\_mul\_add:nnnnwnnnn`.)

```
__fp_fixed_mul_add:nnnnwnnwN {\langle partial_1\rangle} {\langle partial_2\rangle}
{\langle a_1\rangle} {\langle a_5\rangle} {\langle a_6\rangle} ; {\langle b_1\rangle} {\langle b_5\rangle} {\langle b_6\rangle} ;
\langle op\rangle + \langle c_5\rangle \langle c_6\rangle ;
```

Complete the  $\langle partial_1\rangle$  and  $\langle partial_2\rangle$  expressions as explained for the *ii* auxiliary. The second one is divided by 10000: this is the carry from level  $10^{-28}$ . The trailing  $+ c_5 c_6$  is taken into the expression for level  $10^{-24}$ . Note that the total of level  $10^{-24}$  is in the interval  $[-5 \cdot 10^8, 6 \cdot 10^8]$  (give or take a couple of 10000), hence adding it to the shift gives a 10-digit number, as expected by the packing auxiliaries. See *l3fp-aux* for the definition of the shifts and packing auxiliaries.

```
19901 \cs_new:Npn __fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
19902 {
19903 #9 (#4* #1 *#7)
19904 #9 (#5##6+#4* #2 *#7+#3##8) / \c_fp_myriad_int
19905 }
```

(End definition for `\_\_fp\_fixed\_mul\_add:nnnnwnnwN`.)

### 32.8 Extended-precision floating point numbers

In this section we manipulate floating point numbers with roughly 24 significant figures (“extended-precision” numbers, in short, “ep”), which take the form of an integer exponent, followed by a comma, then six groups of digits, ending with a semicolon. The first group of digit may be any non-negative integer, while other groups of digits have 4 digits. In other words, an extended-precision number is an exponent ending in a comma, then a fixed point number. The corresponding value is  $0.\langle digits\rangle \cdot 10^{\langle exponent\rangle}$ . This convention differs from floating points.

```

__fp_ep_to_fixed:www
__fp_ep_to_fixed_auxi:www
__fp_ep_to_fixed_auxii:nnnnnnnnwn

19906 \cs_new:Npn __fp_ep_to_fixed:wwn #1,#2
19907 {
19908 \exp_after:wN __fp_ep_to_fixed_auxi:www
19909 \int_value:w __fp_int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
19910 \exp:w \exp_end_continue_f:w
19911 \prg_replicate:nn { 4 - \int_max:nn {#1} { -32 } } { 0 } ;
19912 }
19913 \cs_new:Npn __fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
19914 {
19915 __fp_pack_eight:wNNNNNNNN
19916 __fp_pack_twice_four:wNNNNNNNN
19917 __fp_pack_twice_four:wNNNNNNNN
19918 __fp_pack_twice_four:wNNNNNNNN
19919 __fp_ep_to_fixed_auxii:nnnnnnwn ;
19920 #2 #1#3#4#5#6#7 0000 !
19921 }
19922 \cs_new:Npn __fp_ep_to_fixed_auxii:nnnnnnwn #1#2#3#4#5#6#7; #8! #9
19923 { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }

(End definition for __fp_ep_to_fixed:www, __fp_ep_to_fixed_auxi:www, and __fp_ep_to_fixed_auxii:nnnnnnwn.)

```

```

__fp_ep_to_ep:wwN
__fp_ep_to_ep_loop:N
__fp_ep_to_ep_end:www
__fp_ep_to_ep_zero:ww

```

Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation **#8** is placed before the resulting exponent–mantissa pair. The input exponent may in fact be given as an integer expression. The **loop** auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the **end** auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in  $3 \times 2 = 6$  blocks of four. At the end of the day, remove with **\\_\\_fp\\_use\\_i:ww** any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

```

19924 \cs_new:Npn __fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
19925 {
19926 \exp_after:wN #8
19927 \int_value:w __fp_int_eval:w #1 + 4
19928 \exp_after:wN \use_i:nn
19929 \exp_after:wN __fp_ep_to_ep_loop:N
19930 \int_value:w __fp_int_eval:w 1 0000 0000 + #2 __fp_int_eval_end:
19931 #3#4#5#6#7 ; ;
19932 }
19933 \cs_new:Npn __fp_ep_to_ep_loop:N #1
19934 {
19935 \if_meaning:w 0 #1
19936 - 1
19937 \else:
19938 __fp_ep_to_ep_end:www #1
19939 \fi:
19940 __fp_ep_to_ep_loop:N

```

```

19941 }
19942 \cs_new:Npn __fp_ep_to_ep_end:www
19943 #1 \fi: __fp_ep_to_ep_loop:N #2; #3!
19944 {
19945 \fi:
19946 \if_meaning:w ; #1
19947 - 2 * \c_fp_max_exponent_int
19948 __fp_ep_to_ep_zero:ww
19949 \fi:
19950 __fp_pack_twice_four:wNNNNNNNNN
19951 __fp_pack_twice_four:wNNNNNNNNN
19952 __fp_pack_twice_four:wNNNNNNNNN
19953 __fp_use_i:ww , ;
19954 #1 #2 0000 0000 0000 0000 0000 0000 ;
19955 }
19956 \cs_new:Npn __fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
19957 { \fi: , {1000}{0000}{0000}{0000}{0000} ; }

```

(End definition for `\__fp_ep_to_ep:wwN` and others.)

In `\_fp-trig` we need to compare two extended-precision numbers. This is based on the same function for positive floating point numbers, with an extra test if comparing only 16 decimals is not enough to distinguish the numbers. Note that this function only works if the numbers are normalized so that their first block is in [1000, 9999].

```

19958 \cs_new:Npn __fp_ep_compare:wwww #1,#2#3#4#5#6#7;
19959 { __fp_ep_compare_aux:wwww {#1}{#2}{#3}{#4}{#5}; #6#7; }
19960 \cs_new:Npn __fp_ep_compare_aux:wwww #1:#2:#3, #4#5#6#7#8#9;
19961 {
19962 \if_case:w
19963 __fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
19964 \if_int_compare:w #2 = #8#9 \exp_stop_f:
19965 0
19966 \else:
19967 \if_int_compare:w #2 < #8#9 - \fi: 1
19968 \fi:
19969 \or: 1
19970 \else: -1
19971 \fi:
19972 }

```

(End definition for `\__fp_ep_compare:wwww` and `\__fp_ep_compare_aux:wwww`.)

`\__fp_ep_mul:wwww`  
`\__fp_ep_mul_raw:wwwwN`

Multiply two extended-precision numbers: first normalize them to avoid losing too much precision, then multiply the mantissas #2 and #4 as fixed point numbers, and sum the exponents #1 and #3. The result's first block is in [100, 9999].

```

19973 \cs_new:Npn __fp_ep_mul:wwwn #1,#2; #3,#4;
19974 {
19975 __fp_ep_to_ep:wwN #3,#4;
19976 __fp_fixed_continue:wn
19977 {
19978 __fp_ep_to_ep:wwN #1,#2;
19979 __fp_ep_mul_raw:wwwn
19980 }
19981 __fp_fixed_continue:wn

```

```

19982 }
19983 \cs_new:Npn __fp_ep_mul_raw:wwwN #1,#2; #3,#4; #5
19984 {
19985 __fp_fixed_mul:wwn #2; #4;
19986 { \exp_after:wN #5 \int_value:w __fp_int_eval:w #1 + #3 , }
19987 }

```

(End definition for `\__fp_ep_mul:wwwn` and `\__fp_ep_mul_raw:wwwN`.)

### 32.9 Dividing extended-precision numbers

Divisions of extended-precision numbers are difficult to perform with exact rounding: the technique used in `l3fp-basics` for 16-digit floating point numbers does not generalize easily to 24-digit numbers. Thankfully, there is no need for exact rounding.

Let us call  $\langle n \rangle$  the numerator and  $\langle d \rangle$  the denominator. After a simple normalization step, we can assume that  $\langle n \rangle \in [0.1, 1)$  and  $\langle d \rangle \in [0.1, 1)$ , and compute  $\langle n \rangle / (10\langle d \rangle) \in (0.01, 1)$ . In terms of the 6 blocks of digits  $\langle n_1 \rangle \cdots \langle n_6 \rangle$  and the 6 blocks  $\langle d_1 \rangle \cdots \langle d_6 \rangle$ , the condition translates to  $\langle n_1 \rangle, \langle d_1 \rangle \in [1000, 9999]$ .

We first find an integer estimate  $a \simeq 10^8 / \langle d \rangle$  by computing

$$\begin{aligned} \alpha &= \left[ \frac{10^9}{\langle d_1 \rangle + 1} \right] \\ \beta &= \left[ \frac{10^9}{\langle d_1 \rangle} \right] \\ a &= 10^3 \alpha + (\beta - \alpha) \cdot \left( 10^3 - \left[ \frac{\langle d_2 \rangle}{10} \right] \right) - 1250, \end{aligned}$$

where  $\left[ \frac{\bullet}{\bullet} \right]$  denotes  $\varepsilon$ -TEX's rounding division, which rounds ties away from zero. The idea is to interpolate between  $10^3 \alpha$  and  $10^3 \beta$  with a parameter  $\langle d_2 \rangle / 10^4$ , so that when  $\langle d_2 \rangle = 0$  one gets  $a = 10^3 \beta - 1250 \simeq 10^{12} / \langle d_1 \rangle \simeq 10^8 / \langle d \rangle$ , while when  $\langle d_2 \rangle = 9999$  one gets  $a = 10^3 \alpha - 1250 \simeq 10^{12} / (\langle d_1 \rangle + 1) \simeq 10^8 / \langle d \rangle$ . The shift by 1250 helps to ensure that  $a$  is an underestimate of the correct value. We shall prove that

$$1 - 1.755 \cdot 10^{-5} < \frac{\langle d \rangle a}{10^8} < 1.$$

We can then compute the inverse of  $\langle d \rangle a / 10^8 = 1 - \epsilon$  using the relation  $1 / (1 - \epsilon) \simeq (1 + \epsilon)(1 + \epsilon^2) + \epsilon^4$ , which is correct up to a relative error of  $\epsilon^5 < 1.6 \cdot 10^{-24}$ . This allows us to find the desired ratio as

$$\frac{\langle n \rangle}{\langle d \rangle} = \frac{\langle n \rangle a}{10^8} ((1 + \epsilon)(1 + \epsilon^2) + \epsilon^4).$$

Let us prove the upper bound first (multiplied by  $10^{15}$ ). Note that  $10^7 \langle d \rangle < 10^3 \langle d_1 \rangle + 10^{-1}(\langle d_2 \rangle + 1)$ , and that  $\varepsilon$ -TEX's division  $\left[ \frac{\langle d_2 \rangle}{10} \right]$  underestimates  $10^{-1}(\langle d_2 \rangle + 1)$  by 0.5 at

most, as can be checked for each possible last digit of  $\langle d_2 \rangle$ . Then,

$$10^7 \langle d \rangle a < \left( 10^3 \langle d_1 \rangle + \left[ \frac{\langle d_2 \rangle}{10} \right] + \frac{1}{2} \right) \left( \left( 10^3 - \left[ \frac{\langle d_2 \rangle}{10} \right] \right) \beta + \left[ \frac{\langle d_2 \rangle}{10} \right] \alpha - 1250 \right) \quad (1)$$

$$< \left( 10^3 \langle d_1 \rangle + \left[ \frac{\langle d_2 \rangle}{10} \right] + \frac{1}{2} \right) \quad (2)$$

$$\left( \left( 10^3 - \left[ \frac{\langle d_2 \rangle}{10} \right] \right) \left( \frac{10^9}{\langle d_1 \rangle} + \frac{1}{2} \right) + \left[ \frac{\langle d_2 \rangle}{10} \right] \left( \frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2} \right) - 1250 \right) \quad (3)$$

$$< \left( 10^3 \langle d_1 \rangle + \left[ \frac{\langle d_2 \rangle}{10} \right] + \frac{1}{2} \right) \left( \frac{10^{12}}{\langle d_1 \rangle} - \left[ \frac{\langle d_2 \rangle}{10} \right] \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 750 \right) \quad (4)$$

We recognize a quadratic polynomial in  $[\langle d_2 \rangle / 10]$  with a negative leading coefficient: this polynomial is bounded above, according to  $([\langle d_2 \rangle / 10] + a)(b - c[\langle d_2 \rangle / 10]) \leq (b + ca)^2 / (4c)$ . Hence,

$$10^7 \langle d \rangle a < \frac{10^{15}}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} \left( \langle d_1 \rangle + \frac{1}{2} + \frac{1}{4} 10^{-3} - \frac{3}{8} \cdot 10^{-9} \langle d_1 \rangle (\langle d_1 \rangle + 1) \right)^2$$

Since  $\langle d_1 \rangle$  takes integer values within  $[1000, 9999]$ , it is a simple programming exercise to check that the squared expression is always less than  $\langle d_1 \rangle (\langle d_1 \rangle + 1)$ , hence  $10^7 \langle d \rangle a < 10^{15}$ . The upper bound is proven. We also find that  $\frac{3}{8}$  can be replaced by slightly smaller numbers, but nothing less than  $0.374563\dots$ , and going back through the derivation of the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

$$10^7 \langle d \rangle a > \left( 10^3 \langle d_1 \rangle + \left[ \frac{\langle d_2 \rangle}{10} \right] - \frac{1}{2} \right) \left( \frac{10^{12}}{\langle d_1 \rangle} - \left[ \frac{\langle d_2 \rangle}{10} \right] \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 1750 \right)$$

This time, we want to find the minimum of this quadratic polynomial. Since the leading coefficient is still negative, the minimum is reached for one of the extreme values  $[y/10] = 0$  or  $[y/10] = 100$ , and we easily check the bound for those values.

We have proven that the algorithm gives us a precise enough answer. Incidentally, the upper bound that we derived tells us that  $a < 10^8 / \langle d \rangle \leq 10^9$ , hence we can compute  $a$  safely as a TeX integer, and even add  $10^9$  to it to ease grabbing of all the digits. The lower bound implies  $10^8 - 1755 < a$ , which we do not care about.

`\__fp_ep_div:wwwn` Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range  $[100, 9999]$ , and is placed after the `<continuation>` once we are done. First normalize the inputs so that both first block lie in  $[1000, 9999]$ , then call `\__fp_ep_div_esti:wwwn`  $\langle \text{denominator} \rangle \langle \text{numerator} \rangle$ , responsible for estimating the inverse of the denominator.

```

19988 \cs_new:Npn __fp_ep_div:wwwn #1,#2; #3,#4;
19989 {
19990 __fp_ep_to_ep:wwN #1,#2;
19991 __fp_fixed_continue:wn
19992 {
19993 __fp_ep_to_ep:wwN #3,#4;
19994 __fp_ep_div_esti:wwwn
19995 }
19996 }
```

(End definition for `\_fp\_ep\_div:wwwn`.)

```
_fp_ep_div_esti:wwwn
\fp_ep_div_estii:wwnnwwn
\fp_ep_div_estiii:NNNNNwwn
```

The `esti` function evaluates  $\alpha = 10^9 / (\langle d \rangle + 1)$ , which is used twice in the expression for  $a$ , and combines the exponents #1 and #4 (with a shift by 1 because we later compute  $\langle n \rangle / (10 \langle d \rangle)$ ). Then the `estii` function evaluates  $10^9 + a$ , and puts the exponent #2 after the continuation #7: from there on we can forget exponents and focus on the mantissa. The `estiii` function multiplies the denominator #7 by  $10^{-8}a$  (obtained as  $a$  split into the single digit #1 and two blocks of 4 digits, #2#3#4#5 and #6). The result  $10^{-8}a\langle d \rangle = (1 - \epsilon)$ , and a partially packed  $10^{-9}a$  (as a block of four digits, and five individual digits, not packed by lack of available macro parameters here) are passed to `\_fp\_ep\_div\_epsi:wnNNNNn`, which computes  $10^{-9}a / (1 - \epsilon)$ , that is,  $1 / (10 \langle d \rangle)$  and we finally multiply this by the numerator #8.

```
19997 \cs_new:Npn _fp_ep_div_esti:wwwn #1,#2#3; #4,
19998 {
19999 \exp_after:wN _fp_ep_div_estii:wwnnwwn
20000 \int_value:w _fp_int_eval:w 10 0000 0000 / (#2 + 1)
20001 \exp_after:wN ;
20002 \int_value:w _fp_int_eval:w #4 - #1 + 1 ,
20003 {#2} #3;
20004 }
20005 \cs_new:Npn _fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
20006 {
20007 \exp_after:wN _fp_ep_div_estiii:NNNNNwwn
20008 \int_value:w _fp_int_eval:w 10 0000 0000 - 1750
20009 + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
20010 {#3}{#4}#5; #6; { #7 #2, }
20011 }
20012 \cs_new:Npn _fp_ep_div_estiii:NNNNNwwn 1#1#2#3#4#5#6; #7;
20013 {
20014 \fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
20015 \fp_ep_div_epsi:wnNNNNn {#1#2#3#4}#5#6
20016 \fp_fixed_mul:wwn
20017 }
```

(End definition for `\_fp\_ep\_div\_esti:wwwn`, `\_fp\_ep\_div\_estii:wwnnwwn`, and `\_fp\_ep\_div\_estiii:NNNNNwwn`.)

```
_fp_ep_div_epsi:wnNNNNn
\fp_ep_div_eps_pack:NNNNNw
\fp_ep_div_epsii:wnNNNNn
```

The bounds shown above imply that the `epsi` function's first operand is  $(1 - \epsilon)$  with  $\epsilon \in [0, 1.755 \cdot 10^{-5}]$ . The `epsi` function computes  $\epsilon$  as  $1 - (1 - \epsilon)$ . Since  $\epsilon < 10^{-4}$ , its first block vanishes and there is no need to explicitly use #1 (which is 9999). Then `epsii` evaluates  $10^{-9}a / (1 - \epsilon)$  as  $(1 + \epsilon^2)(1 + \epsilon)(10^{-9}a\epsilon) + 10^{-9}a$ . Importantly, we compute  $10^{-9}a\epsilon$  before multiplying it with the rest, rather than multiplying by  $\epsilon$  and then  $10^{-9}a$ , as this second option loses more precision. Also, the combination of `short_mul` and `div_myriad` is both faster and more precise than a simple `mul`.

```
20018 \cs_new:Npn _fp_ep_div_epsi:wnNNNNn #1#2#3#4#5#6;
20019 {
20020 \exp_after:wN _fp_ep_div_epsii:wnNNNNn
20021 \int_value:w _fp_int_eval:w 1 9998 - #2
20022 \exp_after:wN _fp_ep_div_eps_pack:NNNNNw
20023 \int_value:w _fp_int_eval:w 1 9999 9998 - #3#4
20024 \exp_after:wN _fp_ep_div_eps_pack:NNNNNw
20025 \int_value:w _fp_int_eval:w 2 0000 0000 - #5#6 ; ;
```

```

20027 \cs_new:Npn __fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6;
20028 { + #1 ; {#2#3#4#5} {#6} }
20029 \cs_new:Npn __fp_ep_div_epsilon:wwnNNNNNn 1#1; #2; #3#4#5#6#7#8
20030 {
20031 __fp_fixed_mul:wwn {0000}{#1}#2; {0000}{#1}#2;
20032 __fp_fixed_add_one:wN
20033 __fp_fixed_mul:wwn {10000} {#1} #2 ;
20034 {
20035 __fp_fixed_mul_short:wwn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
20036 __fp_fixed_div_myriad:wn
20037 __fp_fixed_mul:wwn
20038 }
20039 __fp_fixed_add:wwn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
20040 }

```

(End definition for `\__fp_ep_div_epsilon:wwnNNNNNn`, `\__fp_ep_div_eps_pack:NNNNNw`, and `\__fp_ep_div_epsilon:wwnNNNNNn`.)

### 32.10 Inverse square root of extended precision numbers

The idea here is similar to division. Normalize the input, multiplying by powers of 10 until we have  $x \in [0.01, 1)$ . Then find an integer approximation  $r \in [101, 1003]$  of  $10^2/\sqrt{x}$ , as the fixed point of iterations of the Newton method: essentially  $r \mapsto (r + 10^8/(x_1 r))/2$ , starting from a guess that optimizes the number of steps before convergence. In fact, just as there is a slight shift when computing divisions to ensure that some inequalities hold, we replace  $10^8$  by a slightly larger number which ensures that  $r^2 x \geq 10^4$ . This also causes  $r \in [101, 1003]$ . Another correction to the above is that the input is actually normalized to  $[0.1, 1)$ , and we use either  $10^8$  or  $10^9$  in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation  $r$ , we set  $y = 10^{-4}r^2x$  (or rather, the correct power of 10 to get  $y \simeq 1$ ) and compute  $y^{-1/2}$  through another application of Newton's method. This time, the starting value is  $z = 1$ , each step maps  $z \mapsto z(1.5 - 0.5yz^2)$ , and we perform a fixed number of steps. Our final result combines  $r$  with  $y^{-1/2}$  as  $x^{-1/2} = 10^{-2}ry^{-1/2}$ .

```

__fp_ep_isqrt:wwn
__fp_ep_isqrt_aux:wwn
__fp_ep_isqrt_auxii:wwnnnw

```

First normalize the input, then check the parity of the exponent **#1**. If it is even, the result's exponent will be  $-\#1/2$ , otherwise it will be  $(\#1 - 1)/2$  (except in the case where the input was an exact power of 100). The `auxii` function receives as **#1** the result's exponent just computed, as **#2** the starting value for the iteration giving  $r$  (the values 168 and 535 lead to the least number of iterations before convergence, on average), as **#3** and **#4** one empty argument and one 0, depending on the parity of the original exponent, as **#5** and **#6** the normalized mantissa (**#5**  $\in [1000, 9999]$ ), and as **#7** the continuation. It sets up the iteration giving  $r$ : the `esti` function thus receives the initial two guesses **#2** and 0, an approximation **#5** of  $10^4x$  (its first block of digits), and the empty/zero arguments **#3** and **#4**, followed by the mantissa and an altered continuation where we have stored the result's exponent.

```

20041 \cs_new:Npn __fp_ep_isqrt:wwn #1,#2;
20042 {
20043 __fp_ep_to_ep:wwN #1,#2;
20044 __fp_ep_isqrt_auxii:wwn
20045 }
20046 \cs_new:Npn __fp_ep_isqrt_auxii:wwn #1,
20047 {

```

```

20048 \exp_after:wN _fp_ep_isqrt_auxii:wwnnnwn
20049 \int_value:w _fp_int_eval:w
20050 \int_if_odd:nTF {#1}
20051 { (1 - #1) / 2 , 535 , { 0 } { } }
20052 { 1 - #1 / 2 , 168 , { } { 0 } }
20053 }
20054 \cs_new:Npn _fp_ep_isqrt_auxii:wwnnnwn #1, #2, #3#4 #5#6; #7
20055 {
20056 _fp_ep_isqrt_esti:wwwnnwn #2, 0, #5, {#3} {#4}
20057 {#5} #6 ; {#7} #1 ,
20058 }

```

(End definition for `\_fp_ep_isqrt:wn`, `\_fp_ep_isqrt_aux:wn`, and `\_fp_ep_isqrt_auxii:wwnnnwn`.)

`\_fp_ep_isqrt_esti:wwwnnwn`  
`\_fp_ep_isqrt_estii:wwwnnwn`  
`\_fp_ep_isqrt_estiii>NNNNNwwwn`

If the last two approximations gave the same result, we are done: call the `estii` function to clean up. Otherwise, evaluate  $((\langle prev \rangle + 1.005 \cdot 10^8 \text{ or } 9) / (\langle prev \rangle \cdot x)) / 2$ , as the next approximation: omitting the 1.005 factor, this would be Newton's method. We can check by brute force that if #4 is empty (the original exponent was even), the process computes an integer slightly larger than  $100/\sqrt{x}$ , while if #4 is 0 (the original exponent was odd), the result is an integer slightly larger than  $100/\sqrt{x}/10$ . Once we are done, we evaluate  $100r^2/2$  or  $10r^2/2$  (when the exponent is even or odd, respectively) and feed that to `estiii`. This third auxiliary finds  $y_{\text{even}}/2 = 10^{-4}r^2x/2$  or  $y_{\text{odd}}/2 = 10^{-5}r^2x/2$  (again, depending on earlier parity). A simple program shows that  $y \in [1, 1.0201]$ . The number  $y/2$  is fed to `\_fp_ep_isqrt_epsi:wn`, which computes  $1/\sqrt{y}$ , and we finally multiply the result by  $r$ .

```

20059 \cs_new:Npn _fp_ep_isqrt_esti:wwwnnwn #1, #2, #3, #4
20060 {
20061 \if_int_compare:w #1 = #2 \exp_stop_f:
20062 \exp_after:wN _fp_ep_isqrt_estii:wwwnnwn
20063 \fi:
20064 \exp_after:wN _fp_ep_isqrt_esti:wwwnnwn
20065 \int_value:w _fp_int_eval:w
20066 (#1 + 1 0050 0000 #4 / (#1 * #3)) / 2 ,
20067 #1, #3, {#4}
20068 }
20069 \cs_new:Npn _fp_ep_isqrt_estii:wwwnnwn #1, #2, #3, #4#5
20070 {
20071 \exp_after:wN _fp_ep_isqrt_estiii>NNNNNwwwn
20072 \int_value:w _fp_int_eval:w 1000 0000 + #2 * #2 #5 * 5
20073 \exp_after:wN , \int_value:w _fp_int_eval:w 10000 + #2 ;
20074 }
20075 \cs_new:Npn _fp_ep_isqrt_estiii>NNNNNwwwn 1#1#2#3#4#5#6, 1#7#8; #9;
20076 {
20077 _fp_fixed_mul_short:wn #9; {#1} {#2#3#4#5} {#600} ;
20078 _fp_ep_isqrt_epsi:wn
20079 _fp_fixed_mul_short:wn {#7} {#80} {0000} ;
20080 }

```

(End definition for `\_fp_ep_isqrt_esti:wwwnnwn`, `\_fp_ep_isqrt_estii:wwwnnwn`, and `\_fp_ep_isqrt_estiii>NNNNNwwwn`.)

`\_fp_ep_isqrt_epsi:wn`  
`\_fp_ep_isqrt_epsii:wn`

Here, we receive a fixed point number  $y/2$  with  $y \in [1, 1.0201]$ . Starting from  $z = 1$  we iterate  $z \mapsto z(3/2 - z^2y/2)$ . In fact, we start from the first iteration  $z = 3/2 - y/2$  to avoid useless multiplications. The `epsii` auxiliary receives  $z$  as #1 and  $y$  as #2.

```

20081 \cs_new:Npn __fp_ep_isqrt_епsi:wN #1;
20082 {
20083 __fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}; #1;
20084 __fp_ep_isqrt_епsii:wN #1;
20085 __fp_ep_isqrt_епsii:wN #1;
20086 __fp_ep_isqrt_епsii:wN #1;
20087 }
20088 \cs_new:Npn __fp_ep_isqrt_епsii:wwN #1; #2;
20089 {
20090 __fp_fixed_mul:wwn #1; #1;
20091 __fp_fixed_mul_sub_back:wwn #2;
20092 {15000}{0000}{0000}{0000}{0000};
20093 __fp_fixed_mul:wwn #1;
20094 }

```

(End definition for `\__fp_ep_isqrt_епsi:wN` and `\__fp_ep_isqrt_епsii:wwN`.)

### 32.11 Converting from fixed point to floating point

After computing Taylor series, we wish to convert the result from extended precision (with or without an exponent) to the public floating point format. The functions here should be called within an integer expression for the overall exponent of the floating point.

`\__fp_ep_to_float_o:wwN`  
`\__fp_ep_inv_to_float_o:wwN`

An extended-precision number is simply a comma-delimited exponent followed by a fixed point number. Leave the exponent in the current integer expression then convert the fixed point number.

```

20095 \cs_new:Npn __fp_ep_to_float_o:wwN #1,
20096 { + __fp_int_eval:w #1 __fp_fixed_to_float_o:wN }
20097 \cs_new:Npn __fp_ep_inv_to_float_o:wwN #1,#2;
20098 {
20099 __fp_ep_div:wwwn 1,{1000}{0000}{0000}{0000}{0000}; #1,#2;
20100 __fp_ep_to_float_o:wwN
20101 }

```

(End definition for `\__fp_ep_to_float_o:wwN` and `\__fp_ep_inv_to_float_o:wwN`.)

`\__fp_fixed_inv_to_float_o:wwN`

Another function which reduces to converting an extended precision number to a float.

```

20102 \cs_new:Npn __fp_fixed_inv_to_float_o:wN
20103 { __fp_ep_inv_to_float_o:wwN 0, }

```

(End definition for `\__fp_fixed_inv_to_float_o:wN`.)

`\__fp_fixed_to_float_rad_o:wwN`

Converts the fixed point number #1 from degrees to radians then to a floating point number. This could perhaps remain in `l3fp-trig`.

```

20104 \cs_new:Npn __fp_fixed_to_float_rad_o:wN #1;
20105 {
20106 __fp_fixed_mul:wwn #1; {5729}{5779}{5130}{8232}{0876}{7981};
20107 { __fp_ep_to_float_o:wwN 2, }
20108 }

```

(End definition for `\__fp_fixed_to_float_rad_o:wwN`.)

```

__fp_fixed_to_float_o:wN
__fp_fixed_to_float_o:Nw
... __fp_int_eval:w {exponent} __fp_fixed_to_float_o:wN {⟨a1⟩} {⟨a2⟩} {⟨a3⟩}
{⟨a4⟩} {⟨a5⟩} {⟨a6⟩} ; {sign}
yields
⟨exponent’⟩ ; {⟨a’1⟩} {⟨a’2⟩} {⟨a’3⟩} {⟨a’4⟩} ;

```

And the `to_fixed` version gives six brace groups instead of 4, ensuring that  $1000 \leq \langle a'_1 \rangle \leq 9999$ . At this stage, we know that  $\langle a_1 \rangle$  is positive (otherwise, it is sign of an error before), and we assume that it is less than  $10^8$ .<sup>9</sup>

```

20109 \cs_new:Npn __fp_fixed_to_float_o:Nw #1#2;
20110 { __fp_fixed_to_float_o:wN #2; #1 }
20111 \cs_new:Npn __fp_fixed_to_float_o:wN #1#2#3#4#5#6; #7
20112 { % for the 8-digit-at-the-start thing
20113 + __fp_int_eval:w \c__fp_block_int
20114 \exp_after:wN \exp_after:wN
20115 \exp_after:wN __fp_fixed_to_loop:N
20116 \exp_after:wN \use_none:n
20117 \int_value:w __fp_int_eval:w
20118 1 0000 0000 + #1 \exp_after:wN __fp_use_none_stop_f:n
20119 \int_value:w 1#2 \exp_after:wN __fp_use_none_stop_f:n
20120 \int_value:w 1#3#4 \exp_after:wN __fp_use_none_stop_f:n
20121 \int_value:w 1#5#6
20122 \exp_after:wN ;
20123 \exp_after:wN ;
20124 }
20125 \cs_new:Npn __fp_fixed_to_loop:N #1
20126 {
20127 \if_meaning:w 0 #1
20128 - 1
20129 \exp_after:wN __fp_fixed_to_loop:N
20130 \else:
20131 \exp_after:wN __fp_fixed_to_loop_end:w
20132 \exp_after:wN #1
20133 \fi:
20134 }
20135 \cs_new:Npn __fp_fixed_to_loop_end:w #1 #2 ;
20136 {
20137 \if_meaning:w ; #1
20138 \exp_after:wN __fp_fixed_to_float_zero:w
20139 \else:
20140 \exp_after:wN __fp_pack_twice_four:wNNNNNNNN
20141 \exp_after:wN __fp_pack_twice_four:wNNNNNNNN
20142 \exp_after:wN __fp_fixed_to_float_pack:ww
20143 \exp_after:wN ;
20144 \fi:
20145 #1 #2 0000 0000 0000 0000 ;
20146 }
20147 \cs_new:Npn __fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
20148 {
20149 - 2 * \c__fp_max_exponent_int ;
20150 {0000} {0000} {0000} {0000} ;
20151 }

```

---

<sup>9</sup>Bruno: I must double check this assumption.

```

20152 \cs_new:Npn __fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
20153 {
20154 \if_int_compare:w #2 > 4 \exp_stop_f:
20155 \exp_after:wN __fp_fixed_to_float_round_up:wnnnnw
20156 \fi:
20157 ; #1 ;
20158 }
20159 \cs_new:Npn __fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
20160 {
20161 \exp_after:wN __fp_basics_pack_high:NNNNNw
20162 \int_value:w __fp_int_eval:w 1 #1#2
20163 \exp_after:wN __fp_basics_pack_low:NNNNNw
20164 \int_value:w __fp_int_eval:w 1 #3#4 + 1 ;
20165 }

```

(End definition for `\__fp_fixed_to_float_o:wN` and `\__fp_fixed_to_float_o:Nw`.)

```

20166
```

## 33 l3fp-expo implementation

```

20167 /*package)
20168 @C=fp
__fp_parse_word_exp:N
__fp_parse_word_ln:N
__fp_parse_word_fact:N
20169 \cs_new:Npn __fp_parse_word_exp:N
20170 { __fp_parse_unary_function:NNN __fp_exp_o:w ? }
20171 \cs_new:Npn __fp_parse_word_ln:N
20172 { __fp_parse_unary_function:NNN __fp_ln_o:w ? }
20173 \cs_new:Npn __fp_parse_word_fact:N
20174 { __fp_parse_unary_function:NNN __fp_fact_o:w ? }

```

(End definition for `\__fp_parse_word_exp:N`, `\__fp_parse_word_ln:N`, and `\__fp_parse_word_fact:N`.)

### 33.1 Logarithm

#### 33.1.1 Work plan

As for many other functions, we filter out special cases in `\__fp_ln_o:w`. Then `\__fp_ln_npos_o:w` receives a positive normal number, which we write in the form  $a \cdot 10^b$  with  $a \in [0.1, 1]$ .

*The rest of this section is actually not in sync with the code. Or is the code not in sync with the section? In the current code,  $c \in [1, 10]$  is such that  $0.7 \leq ac < 1.4$ .*

We are given a positive normal number, of the form  $a \cdot 10^b$  with  $a \in [0.1, 1)$ . To compute its logarithm, we find a small integer  $5 \leq c < 50$  such that  $0.91 \leq ac/5 < 1.1$ , and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms  $\ln(10)$  and  $\ln(c/5)$  are looked up in a table. The last term is computed using the following Taylor series of  $\ln$  near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + t^2 \left(\frac{1}{3} + t^2 \left(\frac{1}{5} + t^2 \left(\frac{1}{7} + t^2 \left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

where  $t = 1 - 10/(ac + 5)$ . We can now see one reason for the choice of  $ac \sim 5$ : then  $ac + 5 = 10(1 - \epsilon)$  with  $-0.05 < \epsilon \leq 0.045$ , hence

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4) \dots,$$

is not too difficult to compute.

### 33.1.2 Some constants

```
\c_fp_ln_i_fixed_tl
\c_fp_ln_ii_fixed_tl
\c_fp_ln_iii_fixed_tl
\c_fp_ln_iv_fixed_tl
\c_fp_ln_vi_fixed_tl
\c_fp_ln_vii_fixed_tl
\c_fp_ln_viii_fixed_tl
\c_fp_ln_ix_fixed_tl
\c_fp_ln_x_fixed_tl
```

20175 \tl\_const:Nn \c\_fp\_ln\_i\_fixed\_tl { {0000}{0000}{0000}{0000}{0000}{0000}; }
20176 \tl\_const:Nn \c\_fp\_ln\_ii\_fixed\_tl { {6931}{4718}{0559}{9453}{0941}{7232}; }
20177 \tl\_const:Nn \c\_fp\_ln\_iii\_fixed\_tl { {10986}{1228}{8668}{1096}{9139}{5245}; }
20178 \tl\_const:Nn \c\_fp\_ln\_iv\_fixed\_tl { {13862}{9436}{1119}{8906}{1883}{4464}; }
20179 \tl\_const:Nn \c\_fp\_ln\_vi\_fixed\_tl { {17917}{5946}{9228}{0550}{0081}{2477}; }
20180 \tl\_const:Nn \c\_fp\_ln\_vii\_fixed\_tl { {19459}{1014}{9055}{3133}{0510}{5353}; }
20181 \tl\_const:Nn \c\_fp\_ln\_viii\_fixed\_tl { {20794}{4154}{1679}{8359}{2825}{1696}; }
20182 \tl\_const:Nn \c\_fp\_ln\_ix\_fixed\_tl { {21972}{2457}{7336}{2193}{8279}{0490}; }
20183 \tl\_const:Nn \c\_fp\_ln\_x\_fixed\_tl { {23025}{8509}{2994}{0456}{8401}{7991}; }

(End definition for `\c_fp_ln_i_fixed_tl` and others.)

### 33.1.3 Sign, exponent, and special numbers

```
_fp_ln_o:w
```

The logarithm of negative numbers (including  $-\infty$  and  $-0$ ) raises the “invalid” exception. The logarithm of  $+0$  is  $-\infty$ , raising a division by zero exception. The logarithm of  $+\infty$  or a `nan` is itself. Positive normal numbers call `\_fp_ln_npos_o:w`.

```
20184 \cs_new:Npn _fp_ln_o:w #1 \s_fp _fp_chk:w #2#3#4; @
20185 {
20186 \if_meaning:w 2 #3
20187 _fp_case_use:nw { _fp_invalid_operation_o:nw { ln } }
20188 \fi:
20189 \if_case:w #2 \exp_stop_f:
20190 _fp_case_use:nw
20191 { _fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
20192 \or:
20193 \else:
20194 _fp_case_return_same_o:w
20195 \fi:
20196 _fp_ln_npos_o:w \s_fp _fp_chk:w #2#3#4;
20197 }
```

(End definition for `\_fp_ln_o:w`.)

### 33.1.4 Absolute ln

```
_fp_ln_npos_o:w
```

We catch the case of a significand very close to 0.1 or to 1. In all other cases, the final result is at least  $10^{-4}$ , and then an error of  $0.5 \cdot 10^{-20}$  is acceptable.

```
20198 \cs_new:Npn _fp_ln_npos_o:w \s_fp _fp_chk:w 10#1#2#3;
20199 { %^^A todo: ln(1) should be "exact zero", not "underflow"
20200 \exp_after:wN _fp_SANITIZE:Nw
20201 \int_value:w % for the overall sign
```

```

20202 \if_int_compare:w #1 < 1 \exp_stop_f:
20203 2
20204 \else:
20205 0
20206 \fi:
20207 \exp_after:wN \exp_stop_f:
20208 \int_value:w __fp_int_eval:w % for the exponent
20209 __fp_ln_significand:NNNNnnnN #2#3
20210 __fp_ln_exponent:wn {#1}
20211 }

```

(End definition for `\__fp_ln_npos_o:w`.)

`\__fp_ln_significand:NNNNnnnN` `\__fp_ln_significand:NNNNnnnN {X}_1 {X}_2 {X}_3 {X}_4 {continuation}`  
This function expands to

`{continuation} {Y}_1 {Y}_2 {Y}_3 {Y}_4 {Y}_5 {Y}_6 ;`

where  $Y = -\ln(X)$  as an extended fixed point.

```

20212 \cs_new:Npn __fp_ln_significand:NNNNnnnN #1#2#3#4
20213 {
20214 \exp_after:wN __fp_ln_x_ii:wnnnn
20215 \int_value:w
20216 \if_case:w #1 \exp_stop_f:
20217 \or:
20218 \if_int_compare:w #2 < 4 \exp_stop_f:
20219 __fp_int_eval:w 10 - #2
20220 \else:
20221 6
20222 \fi:
20223 \or: 4
20224 \or: 3
20225 \or: 2
20226 \or: 2
20227 \or: 2
20228 \else: 1
20229 \fi:
20230 ; { #1 #2 #3 #4 }
20231 }

```

(End definition for `\__fp_ln_significand:NNNNnnnN`.)

`\__fp_ln_x_ii:wnnnn` We have thus found  $c \in [1, 10]$  such that  $0.7 \leq ac < 1.4$  in all cases. Compute  $1 + x = 1 + ac \in [1.7, 2.4]$ .

```

20232 \cs_new:Npn __fp_ln_x_ii:wnnnn #1; #2#3#4#5
20233 {
20234 \exp_after:wN __fp_ln_div_after:Nw
20235 \cs:w c__fp_ln_ __fp_int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
20236 \int_value:w
20237 \exp_after:wN __fp_ln_x_iv:wnnnnnnnn
20238 \int_value:w __fp_int_eval:
20239 \exp_after:wN __fp_ln_x_iii_var:NNNNNw
20240 \int_value:w __fp_int_eval:w 9999 9990 + #1##2#3 +
20241 \exp_after:wN __fp_ln_x_iii:NNNNNw
20242 \int_value:w __fp_int_eval:w 10 0000 0000 + #1##4#5 ;

```

```

20243 {20000} {0000} {0000} {0000}
20244 } %^A todo: reoptimize (a generalization attempt failed).
20245 \cs_new:Npn __fp_ln_x_iii:NNNNNNw #1#2 #3#4#5#6 #7;
20246 { #1#2; {#3#4#5#6} {#7} }
20247 \cs_new:Npn __fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
20248 {
20249 #1#2#3#4#5 + 1 ;
20250 {#1#2#3#4#5} {#6}
20251 }

```

The Taylor series to be used is expressed in terms of  $t = (x - 1)/(x + 1) = 1 - 2/(x + 1)$ . We now compute the quotient with extended precision, reusing some code from `\__fp_-/_o:ww`. Note that  $1 + x$  is known exactly.

To reuse notations from `l3fp-basics`, we want to compute  $A/Z$  with  $A = 2$  and  $Z = x + 1$ . In `l3fp-basics`, we considered the case where both  $A$  and  $Z$  are arbitrary, in the range  $[0.1, 1)$ , and we had to monitor the growth of the sequence of remainders  $A$ ,  $B$ ,  $C$ , etc. to ensure that no overflow occurred during the computation of the next quotient. The main source of risk was our choice to define the quotient as roughly  $10^9 \cdot A/10^5 \cdot Z$ : then  $A$  was bound to be below  $2.147\cdots$ , and this limit was never far.

In our case, we can simply work with  $10^8 \cdot A$  and  $10^4 \cdot Z$ , because our reason to work with higher powers has gone: we needed the integer  $y \simeq 10^5 \cdot Z$  to be at least  $10^4$ , and now, the definition  $y \simeq 10^4 \cdot Z$  suffices.

Let us thus define  $y = \lfloor 10^4 \cdot Z \rfloor + 1 \in (1.7 \cdot 10^4, 2.4 \cdot 10^4]$ , and

$$Q_1 = \left\lfloor \frac{\lfloor 10^8 \cdot A \rfloor}{y} - \frac{1}{2} \right\rfloor.$$

(The  $1/2$  comes from how  $\varepsilon$ -TeX rounds.) As for division, it is easy to see that  $Q_1 \leq 10^4 A/Z$ , i.e.,  $Q_1$  is an underestimate.

Exactly as we did for division, we set  $B = 10^4 A - Q_1 Z$ . Then

$$\begin{aligned} 10^4 B &\leq A_1 A_2 \cdot A_3 A_4 - \left( \frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z \\ &\leq A_1 A_2 \left( 1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y \\ &\leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y \end{aligned}$$

In the same way, and using  $1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4$ , and convexity, we get

$$\begin{aligned} 10^4 A &= 2 \cdot 10^4 \\ 10^4 B &\leq 10^8 \frac{A}{y} + 1.6y \leq 4.7 \cdot 10^4 \\ 10^4 C &\leq 10^8 \frac{B}{y} + 1.6y \leq 5.8 \cdot 10^4 \\ 10^4 D &\leq 10^8 \frac{C}{y} + 1.6y \leq 6.3 \cdot 10^4 \\ 10^4 E &\leq 10^8 \frac{D}{y} + 1.6y \leq 6.5 \cdot 10^4 \\ 10^4 F &\leq 10^8 \frac{E}{y} + 1.6y \leq 6.6 \cdot 10^4 \end{aligned}$$

Note that we compute more steps than for division: since  $t$  is not the end result, we need to know it with more accuracy (on the other hand, the ending is much simpler, as we don't need an exact rounding for transcendental functions, but just a faithful rounding).

```
__fp_ln_x_iv:wnnnnnnnnn {1 or 2} {8d} ; {<4d>} {<4d>} {fixed-tl}
```

The number is  $x$ . Compute  $y$  by adding 1 to the five first digits.

```
20252 \cs_new:Npn __fp_ln_x_iv:wnnnnnnnnn #1; #2#3#4#5 #6#7#8#9
20253 {
20254 \exp_after:wN __fp_div_significand_pack:NNN
20255 \int_value:w __fp_int_eval:w
20256 __fp_ln_div_i:w #1 ;
20257 #6 #7 ; {#8} {#9}
20258 {#2} {#3} {#4} {#5}
20259 { \exp_after:wN __fp_ln_div_ii:wwn \int_value:w #1 }
20260 { \exp_after:wN __fp_ln_div_ii:wwn \int_value:w #1 }
20261 { \exp_after:wN __fp_ln_div_ii:wwn \int_value:w #1 }
20262 { \exp_after:wN __fp_ln_div_ii:wwn \int_value:w #1 }
20263 { \exp_after:wN __fp_ln_div_vi:wwn \int_value:w #1 }
20264 }
20265 \cs_new:Npn __fp_ln_div_i:w #1;
20266 {
20267 \exp_after:wN __fp_div_significand_calc:wwnnnnnnn
20268 \int_value:w __fp_int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
20269 }
20270 \cs_new:Npn __fp_ln_div_ii:wwn #1; #2;#3 % y; B1;B2 <- for k=1
20271 {
20272 \exp_after:wN __fp_div_significand_pack:NNN
20273 \int_value:w __fp_int_eval:w
20274 \exp_after:wN __fp_div_significand_calc:wwnnnnnnn
20275 \int_value:w __fp_int_eval:w 999999 + #2 #3 / #1 ; % Q2
20276 #2 #3 ;
20277 }
20278 \cs_new:Npn __fp_ln_div_vi:wwn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
20279 {
20280 \exp_after:wN __fp_div_significand_pack:NNN
```

```

20281 \int_value:w \fp_int_eval:w 1000000 + #2 #3 / #1 ; % Q6
20282 }

```

We now have essentially

```

\fp_ln_div_after:Nw <fixed tl>
\fp_div_significand_pack:NNN 106 + Q1
\fp_div_significand_pack:NNN 106 + Q2
\fp_div_significand_pack:NNN 106 + Q3
\fp_div_significand_pack:NNN 106 + Q4
\fp_div_significand_pack:NNN 106 + Q5
\fp_div_significand_pack:NNN 106 + Q6 ;
<exponent> ; <continuation>

```

where  $\langle \text{fixed } tl \rangle$  holds the logarithm of a number in  $[1, 10]$ , and  $\langle \text{exponent} \rangle$  is the exponent. Also, the expansion is done backwards. Then  $\fp_div\_significand\_pack:NNN$  puts things in the correct order to add the  $Q_i$  together and put semicolons between each piece. Once those have been expanded, we get

```

\fp_ln_div_after:Nw <fixed-tl> <1d> ; <4d> ; <4d> ;
<4d> ; <4d> ; <4d> ; <4d> ; <exponent> ;

```

Just as with division, we know that the first two digits are 1 and 0 because of bounds on the final result of the division  $2/(x+1)$ , which is between roughly 0.8 and 1.2. We then compute  $1 - 2/(x+1)$ , after testing whether  $2/(x+1)$  is greater than or smaller than 1.

```

20283 \cs_new:Npn \fp_ln_div_after:Nw #1#2;
20284 {
20285 \if_meaning:w 0 #2
20286 \exp_after:wN \fp_ln_t_small:Nw
20287 \else:
20288 \exp_after:wN \fp_ln_t_large>NNw
20289 \exp_after:wN -
20290 \fi:
20291 #1
20292 }
20293 \cs_new:Npn \fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
20294 {
20295 \exp_after:wN \fp_ln_t_large>NNw
20296 \exp_after:wN + % <sign>
20297 \exp_after:wN #1
20298 \int_value:w \fp_int_eval:w 9999 - #2 \exp_after:wN ;
20299 \int_value:w \fp_int_eval:w 9999 - #3 \exp_after:wN ;
20300 \int_value:w \fp_int_eval:w 9999 - #4 \exp_after:wN ;
20301 \int_value:w \fp_int_eval:w 9999 - #5 \exp_after:wN ;
20302 \int_value:w \fp_int_eval:w 9999 - #6 \exp_after:wN ;
20303 \int_value:w \fp_int_eval:w 1 0000 - #7 ;
20304 }

\fp_ln_t_large>NNw <sign> <fixed tl>
<t1>; <t2>; <t3>; <t4>; <t5>; <t6>;
<exponent> ; <continuation>

```

Compute the square  $t^2$ , and keep  $t$  at the end with its sign. We know that  $t < 0.1765$ , so every piece has at most 4 digits. However, since we were not careful in  $\fp_ln_t_small:w$ , they can have less than 4 digits.

```

20305 \cs_new:Npn __fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
20306 {
20307 \exp_after:wN __fp_ln_square_t_after:w
20308 \int_value:w __fp_int_eval:w 9999 0000 + #3*#3
20309 \exp_after:wN __fp_ln_square_t_pack:NNNNNw
20310 \int_value:w __fp_int_eval:w 9999 0000 + 2*#3*#4
20311 \exp_after:wN __fp_ln_square_t_pack:NNNNNw
20312 \int_value:w __fp_int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
20313 \exp_after:wN __fp_ln_square_t_pack:NNNNNw
20314 \int_value:w __fp_int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
20315 \exp_after:wN __fp_ln_square_t_pack:NNNNNw
20316 \int_value:w __fp_int_eval:w
20317 1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
20318 + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
20319 % ; ;
20320 \exp_after:wN __fp_ln_twice_t_after:w
20321 \int_value:w __fp_int_eval:w -1 + 2*#3
20322 \exp_after:wN __fp_ln_twice_t_pack:Nw
20323 \int_value:w __fp_int_eval:w 9999 + 2*#4
20324 \exp_after:wN __fp_ln_twice_t_pack:Nw
20325 \int_value:w __fp_int_eval:w 9999 + 2*#5
20326 \exp_after:wN __fp_ln_twice_t_pack:Nw
20327 \int_value:w __fp_int_eval:w 9999 + 2*#6
20328 \exp_after:wN __fp_ln_twice_t_pack:Nw
20329 \int_value:w __fp_int_eval:w 9999 + 2*#7
20330 \exp_after:wN __fp_ln_twice_t_pack:Nw
20331 \int_value:w __fp_int_eval:w 10000 + 2*#8 ; ;
20332 { __fp_ln_c:NwNw #1 }
20333 #2
20334 }
20335 \cs_new:Npn __fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
20336 \cs_new:Npn __fp_ln_twice_t_after:w #1; { ; ; {#1} }
20337 \cs_new:Npn __fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6;
20338 { + #1#2#3#4#5 ; {#6} }
20339 \cs_new:Npn __fp_ln_square_t_after:w 1 0 #1#2#3 #4;
20340 { __fp_ln_Taylor:wwNw {0#1#2#3} {#4} }

```

(End definition for `\__fp_ln_x_i:wnnnn.`)

`\__fp_ln_Taylor:wwNw` Denoting  $T = t^2$ , we get

```

__fp_ln_Taylor:wwNw
{⟨T1⟩} {⟨T2⟩} {⟨T3⟩} {⟨T4⟩} {⟨T5⟩} {⟨T6⟩} ; ;
{⟨(2t)1⟩} {⟨(2t)2⟩} {⟨(2t)3⟩} {⟨(2t)4⟩} {⟨(2t)5⟩} {⟨(2t)6⟩} ;
{ __fp_ln_c:NwNw ⟨sign⟩ }
⟨fixed t1⟩ ⟨exponent⟩ ; ⟨continuation⟩

```

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + T \left(\frac{1}{3} + T \left(\frac{1}{5} + T \left(\frac{1}{7} + T \left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

The process looks as follows

```

\loop 5; A;
\div_int 5; 1.0; \add A; \mul T; {\loop \eval 5-2;}
\add 0.2; A; \mul T; {\loop \eval 5-2;}
\mul B; T; {\loop 3;}
\loop 3; C;

```

This uses the routine for dividing a number by a small integer ( $< 10^4$ ).

```

20341 \cs_new:Npn _fp_ln_Taylor:wwNw
20342 { _fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000} ; }
20343 \cs_new:Npn _fp_ln_Taylor_loop:www #1; #2; #3;
20344 {
20345 \if_int_compare:w #1 = 1 \exp_stop_f:
20346 _fp_ln_Taylor_break:w
20347 \fi:
20348 \exp_after:wN _fp_fixed_div_int:wwN \c_fp_one_fixed_tl #1;
20349 _fp_fixed_add:wwn #2;
20350 _fp_fixed_mul:wwn #3;
20351 {
20352 \exp_after:wN _fp_ln_Taylor_loop:www
20353 \int_value:w _fp_int_eval:w #1 - 2 ;
20354 }
20355 #3;
20356 }
20357 \cs_new:Npn _fp_ln_Taylor_break:w \fi: #1 _fp_fixed_add:wwn #2#3; #4 ;
20358 {
20359 \fi:
20360 \exp_after:wN _fp_fixed_mul:wwn
20361 \exp_after:wN { \int_value:w _fp_int_eval:w 10000 + #2 } #3;
20362 }

```

(End definition for `\_fp_ln_Taylor:wwNw`.)

```

_fp_ln_c:NwNw <sign>
{<r1>} {<r2>} {<r3>} {<r4>} {<r5>} {<r6>} ;
<fixed t1> <exponent> ; <continuation>

```

We are now reduced to finding  $\ln(c)$  and  $\langle exponent \rangle \ln(10)$  in a table, and adding it to the mixture. The first step is to get  $\ln(c) - \ln(x) = -\ln(a)$ , then we get  $b \ln(10)$  and add or subtract.

For now,  $\ln(x)$  is given as  $\cdot 10^0$ . Unless both the exponent is 1 and  $c = 1$ , we shift to working in units of  $\cdot 10^4$ , since the final result is at least  $\ln(10/7) \simeq 0.35$ .

```

20363 \cs_new:Npn _fp_ln_c:NwNw #1 #2; #3
20364 {
20365 \if_meaning:w + #1
20366 \exp_after:wN \exp_after:wN \exp_after:wN _fp_fixed_sub:wwn
20367 \else:
20368 \exp_after:wN \exp_after:wN \exp_after:wN _fp_fixed_add:wwn
20369 \fi:
20370 #3 #2 ;
20371 }

```

(End definition for `\_fp_ln_c:NwNw`.)

```

__fp_ln_exponent:wn
{⟨s1⟩} {⟨s2⟩} {⟨s3⟩} {⟨s4⟩} {⟨s5⟩} {⟨s6⟩} ;
{⟨exponent⟩}

```

Compute  $\langle exponent \rangle$  times  $\ln(10)$ . Apart from the cases where  $\langle exponent \rangle$  is 0 or 1, the result is necessarily at least  $\ln(10) \simeq 2.3$  in magnitude. We can thus drop the least significant 4 digits. In the case of a very large (positive or negative) exponent, we can (and we need to) drop 4 additional digits, since the result is of order  $10^4$ . Naively, one would think that in both cases we can drop 4 more digits than we do, but that would be slightly too tight for rounding to happen correctly. Besides, we already have addition and subtraction for 24 digits fixed point numbers.

```

20372 \cs_new:Npn __fp_ln_exponent:wn #1; #
20373 {
20374 \if_case:w #2 \exp_stop_f:
20375 0 __fp_case_return:nw { __fp_fixed_to_float_o:Nw 2 }
20376 \or:
20377 \exp_after:wN __fp_ln_exponent_one:ww \int_value:w
20378 \else:
20379 \if_int_compare:w #2 > 0 \exp_stop_f:
20380 \exp_after:wN __fp_ln_exponent_small:NNww
20381 \exp_after:wN 0
20382 \exp_after:wN __fp_fixed_sub:wwn \int_value:w
20383 \else:
20384 \exp_after:wN __fp_ln_exponent_small:NNww
20385 \exp_after:wN 2
20386 \exp_after:wN __fp_fixed_add:wwn \int_value:w -
20387 \fi:
20388 \fi:
20389 #2; #1;
20390 }

```

Now we painfully write all the cases.<sup>10</sup> No overflow nor underflow can happen, except when computing  $\ln(1)$ .

```

20391 \cs_new:Npn __fp_ln_exponent_one:ww 1; #
20392 {
20393 0
20394 \exp_after:wN __fp_fixed_sub:wwn \c_fp_ln_x_fixed_tl #1;
20395 __fp_fixed_to_float_o:wN 0
20396 }

```

For small exponents, we just drop one block of digits, and set the exponent of the log to 4 (minus any shift coming from leading zeros in the conversion from fixed point to floating point). Note that here the exponent has been made positive.

```

20397 \cs_new:Npn __fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
20398 {
20399 4
20400 \exp_after:wN __fp_fixed_mul:wwn
20401 \c_fp_ln_x_fixed_tl
20402 {#3}{0000}{0000}{0000}{0000} ;
20403 #2
20404 {0000}{#4}{#5}{#6}{#7}{#8};
20405 __fp_fixed_to_float_o:wN #1
20406 }

```

---

<sup>10</sup>Bruno: do rounding.

(End definition for `\_fp_ln_exponent:wn`.)

## 33.2 Exponential

### 33.2.1 Sign, exponent, and special numbers

```
_fp_exp_o:w
20407 \cs_new:Npn _fp_exp_o:w #1 \s_fp _fp_chk:w #2#3#4; @
20408 {
20409 \if_case:w #2 \exp_stop_f:
20410 _fp_case_return_o:Nw \c_one_fp
20411 \or:
20412 \exp_after:wN _fp_exp_normal_o:w
20413 \or:
20414 \if_meaning:w 0 #3
20415 \exp_after:wN _fp_case_return_o:Nw
20416 \exp_after:wN \c_inf_fp
20417 \else:
20418 \exp_after:wN _fp_case_return_o:Nw
20419 \exp_after:wN \c_zero_fp
20420 \fi:
20421 \or:
20422 _fp_case_return_same_o:w
20423 \fi:
20424 \s_fp _fp_chk:w #2#3#4;
20425 }
```

(End definition for `\_fp_exp_o:w`.)

```
_fp_exp_normal_o:w
_fp_exp_pos_o:Nnw
_fp_overflow>NN
20426 \cs_new:Npn _fp_exp_normal_o:w \s_fp _fp_chk:w 1#1
20427 {
20428 \if_meaning:w 0 #1
20429 _fp_exp_pos_o>NNnw + _fp_fixed_to_float_o:wN
20430 \else:
20431 _fp_exp_pos_o>NNnw - _fp_fixed_inv_to_float_o:wN
20432 \fi:
20433 }
20434 \cs_new:Npn _fp_exp_pos_o>NNnw #1#2#3 \fi: #4#5;
20435 {
20436 \fi:
20437 \if_int_compare:w #4 > \c_fp_max_exp_exponent_int
20438 \token_if_eq_charcode:NNTF + #1
20439 { _fp_overflow>NN _fp_overflow:w \c_inf_fp }
20440 { _fp_overflow>NN _fp_underflow:w \c_zero_fp }
20441 \exp:w
20442 \else:
20443 \exp_after:wN _fp_sanitize:Nw
20444 \exp_after:wN 0
20445 \int_value:w #1 _fp_int_eval:w
20446 \if_int_compare:w #4 < 0 \exp_stop_f:
20447 \exp_after:wN \use_i:nn
20448 \else:
20449 \exp_after:wN \use_ii:nn
```

```

20450 \fi:
20451 {
20452 0
20453 __fp_decimate:nNnnnn { - #4 }
20454 __fp_exp_Taylor:Nnnwn
20455 }
20456 {
20457 __fp_decimate:nNnnnn { \c__fp_prec_int - #4 }
20458 __fp_exp_pos_large:NnnNwn
20459 }
20460 #5
20461 {#4}
20462 #1 #2 0
20463 \exp:w
20464 \fi:
20465 \exp_after:wN \exp_end:
20466 }
20467 \cs_new:Npn __fp_exp_overflow>NN #1#2
20468 {
20469 \exp_after:wN \exp_after:wN
20470 \exp_after:wN #1
20471 \exp_after:wN #2
20472 }

```

(End definition for `\__fp_exp_normal_o:w`, `\__fp_exp_pos_o:NnwNw`, and `\__fp_exp_overflow>NN`.)

`\__fp_exp_Taylor:Nnnwn`  
`\__fp_exp_Taylor_loop:www`  
`\__fp_exp_Taylor_break:Nww`

This function is called for numbers in the range  $[10^{-9}, 10^{-1}]$ . We compute 10 terms of the Taylor series. The first argument is irrelevant (rounding digit used by some other functions). The next three arguments, at least 16 digits, delimited by a semicolon, form a fixed point number, so we pack it in blocks of 4 digits.

```

20473 \cs_new:Npn __fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
20474 {
20475 #6
20476 __fp_pack_twice_four:wNNNNNNNN
20477 __fp_pack_twice_four:wNNNNNNNN
20478 __fp_pack_twice_four:wNNNNNNNN
20479 __fp_exp_Taylor_ii:ww
20480 ; #2#3#4 0000 0000 ;
20481 }
20482 \cs_new:Npn __fp_exp_Taylor_ii:ww #1; #2;
20483 { __fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s__fp_stop }
20484 \cs_new:Npn __fp_exp_Taylor_loop:www #1; #2; #3;
20485 {
20486 \if_int_compare:w #1 = 1 \exp_stop_f:
20487 \exp_after:wN __fp_exp_Taylor_break:Nww
20488 \fi:
20489 __fp_fixed_div_int:wwN #3 ; #1 ;
20490 __fp_fixed_add_one:wN
20491 __fp_fixed_mul:wwN #2 ;
20492 {
20493 \exp_after:wN __fp_exp_Taylor_loop:www
20494 \int_value:w __fp_int_eval:w #1 - 1 ;
20495 #2 ;
20496 }

```

```

20497 }
20498 \cs_new:Npn __fp_exp_Taylor_break:Nww #1 #2; #3 \s__fp_stop
20499 { __fp_fixed_add_one:wN #2 ; }
```

(End definition for `\__fp_exp_Taylor:Nnnwn`, `\__fp_exp_Taylor_loop:www`, and `\__fp_exp_Taylor-break:Nww`.)

`\c_fp_exp_intarray` The integer array has  $6 \times 9 \times 4 = 216$  items encoding the values of  $\exp(j \times 10^i)$  for  $j = 1, \dots, 9$  and  $i = -1, \dots, 4$ . Each value is expressed as  $\simeq 10^p \times 0.m_1m_2m_3$  with three 8-digit blocks  $m_1, m_2, m_3$  and an integer exponent  $p$  (one more than the scientific exponent), and these are stored in the integer array as four items:  $p, 10^8 + m_1, 10^8 + m_2, 10^8 + m_3$ . The various exponentials are stored in increasing order of  $j \times 10^i$ .

Storing this data in an integer array makes it slightly harder to access (slower, too), but uses 16 bytes of memory per exponential stored, while storing as tokens used around 40 tokens; tokens have an especially large footprint in Unicode-aware engines.

```

20500 \intarray_const_from_clist:Nn \c_fp_exp_intarray
20501 {
20502 1 , 1 1105 1709 , 1 1807 5647 , 1 6248 1171 ,
20503 1 , 1 1221 4027 , 1 5816 0169 , 1 8339 2107 ,
20504 1 , 1 1349 8588 , 1 0757 6003 , 1 1039 8374 ,
20505 1 , 1 1491 8246 , 1 9764 1270 , 1 3178 2485 ,
20506 1 , 1 1648 7212 , 1 7070 0128 , 1 1468 4865 ,
20507 1 , 1 1822 1188 , 1 0039 0508 , 1 9748 7537 ,
20508 1 , 1 2013 7527 , 1 0747 0476 , 1 5216 2455 ,
20509 1 , 1 2225 5409 , 1 2849 2467 , 1 6045 7954 ,
20510 1 , 1 2459 6031 , 1 1115 6949 , 1 6638 0013 ,
20511 1 , 1 2718 2818 , 1 2845 9045 , 1 2353 6029 ,
20512 1 , 1 7389 0560 , 1 9893 0650 , 1 2272 3043 ,
20513 2 , 1 2008 5536 , 1 9231 8766 , 1 7740 9285 ,
20514 2 , 1 5459 8150 , 1 0331 4423 , 1 9078 1103 ,
20515 3 , 1 1484 1315 , 1 9102 5766 , 1 0342 1116 ,
20516 3 , 1 4034 2879 , 1 3492 7351 , 1 2260 8387 ,
20517 4 , 1 1096 6331 , 1 5842 8458 , 1 5992 6372 ,
20518 4 , 1 2980 9579 , 1 8704 1728 , 1 2747 4359 ,
20519 4 , 1 8103 0839 , 1 2757 5384 , 1 0077 1000 ,
20520 5 , 1 2202 6465 , 1 7948 0671 , 1 6516 9579 ,
20521 9 , 1 4851 6519 , 1 5409 7902 , 1 7796 9107 ,
20522 14 , 1 1068 6474 , 1 5815 2446 , 1 2146 9905 ,
20523 18 , 1 2353 8526 , 1 6837 0199 , 1 8540 7900 ,
20524 22 , 1 5184 7055 , 1 2858 7072 , 1 4640 8745 ,
20525 27 , 1 1142 0073 , 1 8981 5684 , 1 2836 6296 ,
20526 31 , 1 2515 4386 , 1 7091 9167 , 1 0062 6578 ,
20527 35 , 1 5540 6223 , 1 8439 3510 , 1 0525 7117 ,
20528 40 , 1 1220 4032 , 1 9431 7840 , 1 8020 0271 ,
20529 44 , 1 2688 1171 , 1 4181 6135 , 1 4484 1263 ,
20530 87 , 1 7225 9737 , 1 6812 5749 , 1 2581 7748 ,
20531 131 , 1 1942 4263 , 1 9524 1255 , 1 9365 8421 ,
20532 174 , 1 5221 4696 , 1 8976 4143 , 1 9505 8876 ,
20533 218 , 1 1403 5922 , 1 1785 2837 , 1 4107 3977 ,
20534 261 , 1 3773 0203 , 1 0092 9939 , 1 8234 0143 ,
20535 305 , 1 1014 2320 , 1 5473 5004 , 1 5094 5533 ,
20536 348 , 1 2726 3745 , 1 7211 2566 , 1 5673 6478 ,
20537 391 , 1 7328 8142 , 1 2230 7421 , 1 7051 8866 ,
20538 435 , 1 1970 0711 , 1 1401 7046 , 1 9938 8888 ,
```

```

20539 869 , 1 3881 1801 , 1 9428 4368 , 1 5764 8232 ,
20540 1303 , 1 7646 2009 , 1 8905 4704 , 1 8893 1073 ,
20541 1738 , 1 1506 3559 , 1 7005 0524 , 1 9009 7592 ,
20542 2172 , 1 2967 6283 , 1 8402 3667 , 1 0689 6630 ,
20543 2606 , 1 5846 4389 , 1 5650 2114 , 1 7278 5046 ,
20544 3041 , 1 1151 7900 , 1 5080 6878 , 1 2914 4154 ,
20545 3475 , 1 2269 1083 , 1 0850 6857 , 1 8724 4002 ,
20546 3909 , 1 4470 3047 , 1 3316 5442 , 1 6408 6591 ,
20547 4343 , 1 8806 8182 , 1 2566 2921 , 1 5872 6150 ,
20548 8686 , 1 7756 0047 , 1 2598 6861 , 1 0458 3204 ,
20549 13029 , 1 6830 5723 , 1 7791 4884 , 1 1932 7351 ,
20550 17372 , 1 6015 5609 , 1 3095 3052 , 1 3494 7574 ,
20551 21715 , 1 5297 7951 , 1 6443 0315 , 1 3251 3576 ,
20552 26058 , 1 4665 6719 , 1 0099 3379 , 1 5527 2929 ,
20553 30401 , 1 4108 9724 , 1 3326 3186 , 1 5271 5665 ,
20554 34744 , 1 3618 6973 , 1 3140 0875 , 1 3856 4102 ,
20555 39087 , 1 3186 9209 , 1 6113 3900 , 1 6705 9685 ,
20556 }

```

(End definition for `\c_fp_exp_intarray`.)

```
__fp_exp_pos_large:NnnNwn
__fp_exp_large_after:wwn
 __fp_exp_large:NwN
 __fp_exp_intarray:w
__fp_exp_intarray_aux:w
```

The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros). The third argument is the integer part of our number, then we have the decimal part delimited by a semicolon, and finally the exponent, in the range [0, 5]. Remove leading zeros from the integer part: putting #4 in there too ensures that an integer part of 0 is also removed. Then read digits one by one, looking up  $\exp(\langle digit \rangle \cdot 10^{\langle exponent \rangle})$  in a table, and multiplying that to the current total. The loop is done by `\__fp_exp_large:NwN`, whose #1 is the  $\langle exponent \rangle$ , #2 is the current mantissa, and #3 is the  $\langle digit \rangle$ . At the end, `\__fp_exp_large_after:wwn` moves on to the Taylor series, eventually multiplied with the mantissa that we have just computed.

```

20557 \cs_new:Npn __fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
20558 {
20559 \exp_after:wn \exp_after:wn \exp_after:wn __fp_exp_large:NwN
20560 \exp_after:wn \exp_after:wn \exp_after:wn #6
20561 \exp_after:wn \c_fp_one_fixed_tl
20562 \int_value:w #3 #4 \exp_stop_f:
20563 #5 00000 ;
20564 }
20565 \cs_new:Npn __fp_exp_large:NwN #1#2; #3
20566 {
20567 \if_case:w #3 ~
20568 \exp_after:wn __fp_fixed_continue:wn
20569 \else:
20570 \exp_after:wn __fp_exp_intarray:w
20571 \int_value:w __fp_int_eval:w 36 * #1 + 4 * #3 \exp_after:wn ;
20572 \fi:
20573 #2;
20574 {
20575 \if_meaning:w 0 #1
20576 \exp_after:wn __fp_exp_large_after:wwn
20577 \else:
20578 \exp_after:wn __fp_exp_large:NwN
20579 \int_value:w __fp_int_eval:w #1 - 1 \exp_after:wn \scan_stop:
20580 \fi:

```

```

20581 }
20582 }
20583 \cs_new:Npn __fp_exp_intarray:w #1 ;
20584 {
20585 +
20586 __kernel_intarray_item:Nn \c__fp_exp_intarray
20587 { __fp_int_eval:w #1 - 3 \scan_stop: }
20588 \exp_after:wN \use_i:n
20589 \exp_after:wN __fp_fixed_mul:wwn
20590 \int_value:w 0
20591 \exp_after:wN __fp_exp_intarray_aux:w
20592 \int_value:w __kernel_intarray_item:Nn
20593 \c__fp_exp_intarray { __fp_int_eval:w #1 - 2 }
20594 \exp_after:wN __fp_exp_intarray_aux:w
20595 \int_value:w __kernel_intarray_item:Nn
20596 \c__fp_exp_intarray { __fp_int_eval:w #1 - 1 }
20597 \exp_after:wN __fp_exp_intarray_aux:w
20598 \int_value:w __kernel_intarray_item:Nn \c__fp_exp_intarray {#1} ; ;
20599 }
20600 \cs_new:Npn __fp_exp_intarray_aux:w 1 #1#2#3#4#5 ; { ; {#1#2#3#4} {#5} }
20601 \cs_new:Npn __fp_exp_large_after:wwn #1; #2; #3
20602 {
20603 __fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
20604 __fp_fixed_mul:wwn #1;
20605 }

```

(End definition for `\__fp_exp_pos_large:NnnNwn` and others.)

### 33.3 Power

Raising a number  $a$  to a power  $b$  leads to many distinct situations.

| $a^b$           | $-\infty$  | $(-\infty, -0)$ | -integer        | $\pm 0$    | +integer        | $(0, \infty)$ | $+\infty$  | NaN |
|-----------------|------------|-----------------|-----------------|------------|-----------------|---------------|------------|-----|
| $+\infty$       | +0         | +0              | +1              | + $\infty$ | + $\infty$      | + $\infty$    | NaN        | NaN |
| $(1, \infty)$   | +0         | + $ a ^b$       | +1              | + $ a ^b$  | + $ a ^b$       | + $\infty$    | NaN        | NaN |
| +1              | +1         | +1              | +1              | +1         | +1              | +1            | +1         | +1  |
| $(0, 1)$        | + $\infty$ | + $ a ^b$       | +1              | + $ a ^b$  | + $ a ^b$       | +0            | NaN        | NaN |
| +0              | + $\infty$ | + $\infty$      | +1              | +0         | +0              | +0            | NaN        | NaN |
| -0              | + $\infty$ | NaN             | $(-1)^b \infty$ | +1         | $(-1)^b 0$      | +0            | +0         | NaN |
| $(-1, 0)$       | + $\infty$ | NaN             | $(-1)^b  a ^b$  | +1         | $(-1)^b  a ^b$  | NaN           | +0         | NaN |
| -1              | +1         | NaN             | $(-1)^b$        | +1         | $(-1)^b$        | NaN           | +1         | NaN |
| $(-\infty, -1)$ | +0         | NaN             | $(-1)^b  a ^b$  | +1         | $(-1)^b  a ^b$  | NaN           | + $\infty$ | NaN |
| $-\infty$       | +0         | +0              | $(-1)^b 0$      | +1         | $(-1)^b \infty$ | NaN           | + $\infty$ | NaN |
| NaN             | NaN        | NaN             | NaN             | +1         | NaN             | NaN           | NaN        | NaN |

We distinguished in this table the cases of finite (positive or negative) integer exponents, as  $(-1)^b$  is defined in that case. One peculiarity of this operation is that  $\text{NaN}^0 = 1^{\text{NaN}} = 1$ , because this relation is obeyed for any number, even  $\pm\infty$ .

`\__fp_~o:ww` We cram most of the tests into a single function to save csnames. First treat the case  $b = 0$ :  $a^0 = 1$  for any  $a$ , even `nan`. Then test the sign of  $a$ .

- If it is positive, and  $a$  is a normal number, call `\_fp_pow_normal_o:ww` followed by the two `fp a` and `b`. For  $a = +0$  or  $+\infty$ , call `\_fp_pow_zero_or_inf:ww` instead, to return either  $+0$  or  $+\infty$  as appropriate.
- If  $a$  is a `nan`, then skip to the next semicolon (which happens to be conveniently the end of  $b$ ) and return `nan`.
- Finally, if  $a$  is negative, compute  $a^b$  (`\_fp_pow_normal_o:ww` which ignores the sign of its first operand), and keep an extra copy of  $a$  and  $b$  (the second brace group, containing  $\{ b a \}$ , is inserted between  $a$  and  $b$ ). Then do some tests to find the final sign of the result if it exists.

```

20606 \cs_new:cpn { __fp_ \iow_char:N \^ _o:ww }
20607 \s__fp _fp_chk:w #1#2#3; \s__fp _fp_chk:w #4#5#6;
20608 {
20609 \if_meaning:w 0 #4
20610 _fp_case_return_o:Nw \c_one_fp
20611 \fi:
20612 \if_case:w #2 \exp_stop_f:
20613 \exp_after:wN \use_i:nn
20614 \or:
20615 _fp_case_return_o:Nw \c_nan_fp
20616 \else:
20617 \exp_after:wN _fp_pow_neg:www
20618 \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
20619 \fi:
20620 {
20621 \if_meaning:w 1 #1
20622 \exp_after:wN _fp_pow_normal_o:ww
20623 \else:
20624 \exp_after:wN _fp_pow_zero_or_inf:ww
20625 \fi:
20626 \s__fp _fp_chk:w #1#2#3;
20627 }
20628 { \s__fp _fp_chk:w #4#5#6; \s__fp _fp_chk:w #1#2#3; }
20629 \s__fp _fp_chk:w #4#5#6;
20630 }
```

(End definition for `\_fp_ ^ _o:ww`.)

`\_fp_pow_zero_or_inf:ww` Raising  $-0$  or  $-\infty$  to `nan` yields `nan`. For other powers, the result is  $+0$  if  $0$  is raised to a positive power or  $\infty$  to a negative power, and  $+\infty$  otherwise. Thus, if the type of  $a$  and the sign of  $b$  coincide, the result is  $0$ , since those conveniently take the same possible values,  $0$  and  $2$ . Otherwise, either  $a = \pm\infty$  and  $b > 0$  and the result is  $+\infty$ , or  $a = \pm 0$  with  $b < 0$  and we have a division by zero unless  $b = -\infty$ .

```

20631 \cs_new:Npn _fp_pow_zero_or_inf:ww
20632 \s__fp _fp_chk:w #1#2; \s__fp _fp_chk:w #3#4
20633 {
20634 \if_meaning:w 1 #4
20635 _fp_case_return_same_o:w
20636 \fi:
20637 \if_meaning:w #1 #4
20638 _fp_case_return_o:Nw \c_zero_fp
20639 \fi:
```

```

20640 \if_meaning:w 2 #1
20641 __fp_case_return_o:Nw \c_inf_fp
20642 \fi:
20643 \if_meaning:w 2 #3
20644 __fp_case_return_o:Nw \c_inf_fp
20645 \else:
20646 __fp_case_use:nw
20647 {
20648 __fp_division_by_zero_o:NNww \c_inf_fp ^
20649 \s__fp __fp_chk:w #1 #2 ;
20650 }
20651 \fi:
20652 \s__fp __fp_chk:w #3#4
20653 }

```

(End definition for `\_\_fp_pow_zero_or_inf:ww`.)

`\_\_fp_pow_normal_o:ww` We have in front of us  $a$ , and  $b \neq 0$ , we know that  $a$  is a normal number, and we wish to compute  $|a|^b$ . If  $|a| = 1$ , we return 1, unless  $a = -1$  and  $b$  is `nan`. Indeed, returning 1 at this point would wrongly raise “invalid” when the sign is considered. If  $|a| \neq 1$ , test the type of  $b$ :

- 0 Impossible, we already filtered  $b = \pm 0$ .
- 1 Call `\_\_fp_pow_npos_o:Nww`.
- 2 Return  $+\infty$  or  $+0$  depending on the sign of  $b$  and whether the exponent of  $a$  is positive or not.
- 3 Return  $b$ .

```

20654 \cs_new:Npn __fp_pow_normal_o:ww
20655 \s__fp __fp_chk:w 1 #1#2#3; \s__fp __fp_chk:w #4#5
20656 {
20657 \if_int_compare:w __fp_str_if_eq:nn { #2 #3 }
20658 { 1 {1000} {0000} {0000} {0000} } = 0 \exp_stop_f:
20659 \if_int_compare:w #4 #1 = 32 \exp_stop_f:
20660 \exp_after:wN __fp_case_return_ii_o:ww
20661 \fi:
20662 __fp_case_return_o:Nww \c_one_fp
20663 \fi:
20664 \if_case:w #4 \exp_stop_f:
20665 \or:
20666 \exp_after:wN __fp_pow_npos_o:Nww
20667 \exp_after:wN #5
20668 \or:
20669 \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
20670 \if_int_compare:w #2 > 0 \exp_stop_f:
20671 \exp_after:wN __fp_case_return_o:Nww
20672 \exp_after:wN \c_inf_fp
20673 \else:
20674 \exp_after:wN __fp_case_return_o:Nww
20675 \exp_after:wN \c_zero_fp
20676 \fi:
20677 \or:

```

```

20678 _fp_case_return_ii_o:ww
20679 \fi:
20680 \s_fp _fp_chk:w 1 #1 {#2} #3 ;
20681 \s_fp _fp_chk:w #4 #5
20682 }

```

(End definition for `\_fp_pow_normal_o:ww`.)

`\_fp_pow_npos_o:Nww`

We now know that  $a \neq \pm 1$  is a normal number, and  $b$  is a normal number too. We want to compute  $|a|^b = (|x| \cdot 10^n)^{y \cdot 10^p} = \exp((\ln|x| + n \ln(10)) \cdot y \cdot 10^p) = \exp(z)$ . To compute the exponential accurately, we need to know the digits of  $z$  up to the 16-th position. Since the exponential of  $10^5$  is infinite, we only need at most 21 digits, hence the fixed point result of `\_fp_ln_o:w` is precise enough for our needs. Start an integer expression for the decimal exponent of  $e^{|z|}$ . If  $z$  is negative, negate that decimal exponent, and prepare to take the inverse when converting from the fixed point to the floating point result.

```

20683 \cs_new:Npn _fp_pow_npos_o:Nww #1 \s_fp _fp_chk:w 1#2#3
20684 {
20685 \exp_after:wN _fp_sanitize:Nw
20686 \exp_after:wN 0
20687 \int_value:w
20688 \if:w #1 \if_int_compare:w #3 > 0 \exp_stop_f: 0 \else: 2 \fi:
20689 \exp_after:wN _fp_pow_npos_aux>NNnww
20690 \exp_after:wN +
20691 \exp_after:wN _fp_fixed_to_float_o:wN
20692 \else:
20693 \exp_after:wN _fp_pow_npos_aux>NNnww
20694 \exp_after:wN -
20695 \exp_after:wN _fp_fixed_inv_to_float_o:wN
20696 \fi:
20697 {#3}
20698 }

```

(End definition for `\_fp_pow_npos_o:Nww`.)

`\_fp_pow_npos_aux>NNnww`

The first argument is the conversion function from fixed point to float. Then comes an exponent and the 4 brace groups of  $x$ , followed by  $b$ . Compute  $-\ln(x)$ .

```

20699 \cs_new:Npn _fp_pow_npos_aux>NNnww #1#2#3#4#5; \s_fp _fp_chk:w 1#6#7#8;
20700 {
20701 #1
20702 _fp_int_eval:w
20703 _fp_ln_significand>NNNNnnnn #4#5
20704 _fp_pow_exponent:wnN {#3}
20705 _fp_fixed_mul:wwn #8 {0000}{0000} ;
20706 _fp_pow_B:wwN #7;
20707 #1 #2 0 % fixed_to_float_o:wN
20708 }
20709 \cs_new:Npn _fp_pow_exponent:wnN #1; #2
20710 {
20711 \if_int_compare:w #2 > 0 \exp_stop_f:
20712 \exp_after:wN _fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
20713 \exp_after:wN +
20714 \else:
20715 \exp_after:wN _fp_pow_exponent:Nwnnnnnw % -(|n|\ln(10) + (-\ln(x)))
20716 \exp_after:wN -

```

```

20717 \fi:
20718 #2; #1;
20719 }
20720 \cs_new:Npn __fp_pow_exponent:Nwnnnnnw #1#2; #3#4#5#6#7#8;
20721 { %^A todo: use that in ln.
20722 \exp_after:wN __fp_fixed_mul_after:wwn
20723 \int_value:w __fp_int_eval:w \c__fp_leading_shift_int
20724 \exp_after:wN __fp_pack:NNNNNw
20725 \int_value:w __fp_int_eval:w \c__fp_middle_shift_int
20726 #1#2*23025 - #1 #3
20727 \exp_after:wN __fp_pack:NNNNNw
20728 \int_value:w __fp_int_eval:w \c__fp_middle_shift_int
20729 #1 #2*8509 - #1 #4
20730 \exp_after:wN __fp_pack:NNNNNw
20731 \int_value:w __fp_int_eval:w \c__fp_middle_shift_int
20732 #1 #2*2994 - #1 #5
20733 \exp_after:wN __fp_pack:NNNNNw
20734 \int_value:w __fp_int_eval:w \c__fp_middle_shift_int
20735 #1 #2*0456 - #1 #6
20736 \exp_after:wN __fp_pack:NNNNNw
20737 \int_value:w __fp_int_eval:w \c__fp_trailing_shift_int
20738 #1 #2*8401 - #1 #7
20739 #1 (#2*7991 - #8) / 1 0000 ; ;
20740 }
20741 \cs_new:Npn __fp_pow_B:wwN #1#2#3#4#5#6; #7;
20742 {
20743 \if_int_compare:w #7 < 0 \exp_stop_f:
20744 \exp_after:wN __fp_pow_C_neg:w \int_value:w -
20745 \else:
20746 \if_int_compare:w #7 < 22 \exp_stop_f:
20747 \exp_after:wN __fp_pow_C_pos:w \int_value:w
20748 \else:
20749 \exp_after:wN __fp_pow_C_overflow:w \int_value:w
20750 \fi:
20751 \fi:
20752 #7 \exp_after:wN ;
20753 \int_value:w __fp_int_eval:w 10 0000 + #1 __fp_int_eval_end:
20754 #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^A todo: how many 0?
20755 }
20756 \cs_new:Npn __fp_pow_C_overflow:w #1; #2; #3
20757 {
20758 + 2 * \c__fp_max_exponent_int
20759 \exp_after:wN __fp_fixed_continue:wn \c__fp_one_fixed_tl
20760 }
20761 \cs_new:Npn __fp_pow_C_neg:w #1 ; 1
20762 {
20763 \exp_after:wN \exp_after:wN \exp_after:wN __fp_pow_C_pack:w
20764 \prg_replicate:nn {#1} {0}
20765 }
20766 \cs_new:Npn __fp_pow_C_pos:w #1; 1
20767 { __fp_pow_C_pos_loop:wN #1; }
20768 \cs_new:Npn __fp_pow_C_pos_loop:wN #1; #2
20769 {
20770 \if_meaning:w 0 #1

```

```

20771 \exp_after:wN _fp_pow_C_pack:w
20772 \exp_after:wN #2
20773 \else:
20774 \if_meaning:w 0 #2
20775 \exp_after:wN _fp_pow_C_pos_loop:wN \int_value:w
20776 \else:
20777 \exp_after:wN _fp_pow_C_overflow:w \int_value:w
20778 \fi:
20779 _fp_int_eval:w #1 - 1 \exp_after:wN ;
20780 \fi:
20781 }
20782 \cs_new:Npn _fp_pow_C_pack:w
20783 {
20784 \exp_after:wN _fp_exp_large:NwN
20785 \exp_after:wN 5
20786 \c_fp_one_fixed_tl
20787 }

```

(End definition for `\_fp_pow_npos_aux:NNnww`.)

`\_fp_pow_neg:www` This function is followed by three floating point numbers:  $a^b$ ,  $a \in [-\infty, -0]$ , and  $b$ . If  $b$  is an even integer (case  $-1$ ),  $a^b = a^b$ . If  $b$  is an odd integer (case  $0$ ),  $a^b = -a^b$ , obtained by a call to `\_fp_pow_neg_aux:wNN`. Otherwise, the sign is undefined. This is invalid, unless  $a^b$  turns out to be  $+0$  or `nan`, in which case we return that as  $a^b$ . In particular, since the underflow detection occurs before `\_fp_pow_neg:www` is called,  $(-0.1)^{**}(12345.67)$  gives  $+0$  rather than complaining that the sign is not defined.

```

20788 \cs_new:Npn _fp_pow_neg:www \s_fp _fp_chk:w #1#2; #3; #4;
20789 {
20790 \if_case:w _fp_pow_neg_case:w #4 ;
20791 \exp_after:wN _fp_pow_neg_aux:wNN
20792 \or:
20793 \if_int_compare:w _fp_int_eval:w #1 / 2 = 1 \exp_stop_f:
20794 _fp_invalid_operation_o:Nww ^ #3; #4;
20795 \exp:w \exp_end_continue_f:w
20796 \exp_after:wN \exp_after:wN
20797 \exp_after:wN _fp_use_none_until_s:w
20798 \fi:
20799 \fi:
20800 _fp_exp_after_o:w
20801 \s_fp _fp_chk:w #1#2;
20802 }
20803 \cs_new:Npn _fp_pow_neg_aux:wNN #1 \s_fp _fp_chk:w #2#3
20804 {
20805 \exp_after:wN _fp_exp_after_o:w
20806 \exp_after:wN \s_fp
20807 \exp_after:wN _fp_chk:w
20808 \exp_after:wN #2
20809 \int_value:w _fp_int_eval:w 2 - #3 _fp_int_eval_end:
20810 }

```

(End definition for `\_fp_pow_neg:www` and `\_fp_pow_neg_aux:wNN`.)

`\_fp_pow_neg_case:w` This function expects a floating point number, and determines its “parity”. It should be used after `\if_case:w` or in an integer expression. It gives  $-1$  if the number is an `\_fp_pow_neg_case_aux:Nnnw`

even integer, 0 if the number is an odd integer, and 1 otherwise. Zeros and  $\pm\infty$  are even (because very large finite floating points are even), while `nan` is a non-integer. The sign of normal numbers is irrelevant to parity. After `\_fp_decimate:nNnnnn` the argument #1 of `\_fp_pow_neg_case_aux:Nnnw` is a rounding digit, 0 if and only if the number was an integer, and #3 is the 8 least significant digits of that integer.

```

20811 \cs_new:Npn _fp_pow_neg_case:w \s__fp _fp_chk:w #1#2#3;
20812 {
20813 \if_case:w #1 \exp_stop_f:
20814 -1
20815 \or: _fp_pow_neg_case_aux:nnnnn #3
20816 \or: -1
20817 \else: 1
20818 \fi:
20819 \exp_stop_f:
20820 }
20821 \cs_new:Npn _fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
20822 {
20823 \if_int_compare:w #1 > \c_fp_prec_int
20824 -1
20825 \else:
20826 _fp_decimate:nNnnnn { \c_fp_prec_int - #1 }
20827 _fp_pow_neg_case_aux:Nnnw
20828 {#2} {#3} {#4} {#5}
20829 \fi:
20830 }
20831 \cs_new:Npn _fp_pow_neg_case_aux:Nnnw #1#2#3#4 ;
20832 {
20833 \if_meaning:w 0 #1
20834 \if_int_odd:w #3 \exp_stop_f:
20835 0
20836 \else:
20837 -1
20838 \fi:
20839 \else:
20840 1
20841 \fi:
20842 }
```

(End definition for `\_fp_pow_neg_case:w`, `\_fp_pow_neg_case_aux:nnnnn`, and `\_fp_pow_neg_case_aux:Nnnw`.)

### 33.4 Factorial

`\c_fp_fact_max_arg_int` The maximum integer whose factorial fits in the exponent range is 3248, as  $3249! \sim 10^{10000.8}$

```
20843 \int_const:Nn \c_fp_fact_max_arg_int { 3248 }
```

(End definition for `\c_fp_fact_max_arg_int`.)

`\_fp_fact_o:w` First detect  $\pm 0$  and  $+\infty$  and `nan`. Then note that factorial of anything with a negative sign (except  $-0$ ) is undefined. Then call `\_fp_small_int:wTF` to get an integer as the argument, and start a loop. This is not the most efficient way of computing the factorial,

but it works all right. Of course we work with 24 digits instead of 16. It is easy to check that computing factorials with this precision is enough.

```

20844 \cs_new:Npn __fp_fact_o:w #1 \s__fp __fp_chk:w #2#3#4; @
20845 {
20846 \if_case:w #2 \exp_stop_f:
20847 __fp_case_return_o:Nw \c_one_fp
20848 \or:
20849 \or:
20850 \if_meaning:w 0 #3
20851 \exp_after:wN __fp_case_return_same_o:w
20852 \fi:
20853 \or:
20854 __fp_case_return_same_o:w
20855 \fi:
20856 \if_meaning:w 2 #3
20857 __fp_case_use:nw { __fp_invalid_operation_o:fw { fact } }
20858 \fi:
20859 __fp_fact_pos_o:w
20860 \s__fp __fp_chk:w #2 #3 #4 ;
20861 }

```

(End definition for `\__fp_fact_o:w`.)

`\__fp_fact_pos_o:w`  
`\__fp_fact_int_o:w`

Then check the input is an integer, and call `\__fp_factorial_int_o:n` with that int as an argument. If it's too big the factorial overflows. Otherwise call `\__fp_sanitze:Nw` with a positive sign marker 0 and an integer expression that will mop up any exponent in the calculation.

```

20862 \cs_new:Npn __fp_fact_pos_o:w #1;
20863 {
20864 __fp_small_int:wTF #1;
20865 { __fp_fact_int_o:n }
20866 { __fp_invalid_operation_o:fw { fact } #1; }
20867 }
20868 \cs_new:Npn __fp_fact_int_o:n #1
20869 {
20870 \if_int_compare:w #1 > \c__fp_fact_max_arg_int
20871 __fp_case_return:nw
20872 {
20873 \exp_after:wN \exp_after:wN \exp_after:wN __fp_overflow:w
20874 \exp_after:wN \c_inf_fp
20875 }
20876 \fi:
20877 \exp_after:wN __fp_sanitze:Nw
20878 \exp_after:wN 0
20879 \int_value:w __fp_int_eval:w
20880 __fp_fact_loop_o:w #1 . 4 , { 1 } { } { } { } { } { } ;
20881 }

```

(End definition for `\__fp_fact_pos_o:w` and `\__fp_fact_int_o:w`.)

`\__fp_fact_loop_o:w`

The loop receives an integer `#1` whose factorial we want to compute, which we progressively decrement, and the result so far as an extended-precision number `#2` in the form  $\langle\text{exponent}\rangle,\langle\text{mantissa}\rangle;$ . The loop goes in steps of two because we compute `#1*#1-1` as an integer expression (it must fit since `#1` is at most 3248), then multiply with the

result so far. We don't need to fill in most of the mantissa with zeros because `\_fp\_ep\_mul:wwwn` first normalizes the extended precision number to avoid loss of precision. When reaching a small enough number simply use a table of factorials less than  $10^8$ . This limit is chosen because the normalization step cannot deal with larger integers.

```

20882 \cs_new:Npn _fp_fact_loop_o:w #1 . #2 ;
20883 {
20884 \if_int_compare:w #1 < 12 \exp_stop_f:
20885 _fp_fact_small_o:w #1
20886 \fi:
20887 \exp_after:wN _fp_ep_mul:wwwn
20888 \exp_after:wN 4 \exp_after:wN ,
20889 \exp_after:wN { \int_value:w _fp_int_eval:w #1 * (#1 - 1) }
20890 { } { } { } { } { } ;
20891 #2 ;
20892 {
20893 \exp_after:wN _fp_fact_loop_o:w
20894 \int_value:w _fp_int_eval:w #1 - 2 .
20895 }
20896 }
20897 \cs_new:Npn _fp_fact_small_o:w #1 \fi: #2 ; #3 ; #4
20898 {
20899 \fi:
20900 \exp_after:wN _fp_ep_mul:wwwn
20901 \exp_after:wN 4 \exp_after:wN ,
20902 \exp_after:wN
20903 {
20904 \int_value:w
20905 \if_case:w #1 \exp_stop_f:
20906 1 \or: 1 \or: 2 \or: 6 \or: 24 \or: 120 \or: 720 \or: 5040
20907 \or: 40320 \or: 362880 \or: 3628800 \or: 39916800
20908 \fi:
20909 { } { } { } { } { } { } ;
20910 #3 ;
20911 _fp_ep_to_float_o:wwN 0
20912 }

```

(End definition for `\_fp_fact_loop_o:w`.)

```
20913
```

## 34 I3fp-trig Implementation

```
20914 (*package)
```

```
20915 (@@=fp)
```

```

__fp_parse_word_acos:N Unary functions.
__fp_parse_word_acosd:N
__fp_parse_word_acsc:N
__fp_parse_word_acscd:N
__fp_parse_word_asec:N
__fp_parse_word_aecd:N
__fp_parse_word_asin:N
__fp_parse_word_asind:N
 __fp_parse_word_cos:N
__fp_parse_word_cosd:N
 __fp_parse_word_cot:N
__fp_parse_word_cotd:N
 __fp_parse_word_csc:N
__fp_parse_word_cscd:N
 __fp_parse_word_sec:N
__fp_parse_word_secd:N
 __fp_parse_word_sin:N
__fp_parse_word_sind:N

```

```

20923 {
20924 \exp_not:N __fp_parse_unary_function>NNN
20925 \exp_not:c { __fp_#1_o:w }
20926 \exp_not:N \use_i:nn
20927 }
20928 \cs_new:cpx { __fp_parse_word_#1d:N }
20929 {
20930 \exp_not:N __fp_parse_unary_function>NNN
20931 \exp_not:c { __fp_#1_o:w }
20932 \exp_not:N \use_ii:nn
20933 }
20934 }
```

(End definition for `\__fp_parse_word_acos:N` and others.)

`\__fp_parse_word_acot:N` Those functions may receive a variable number of arguments.

```

__fp_parse_word_acotd:N 20935 \cs_new:Npn __fp_parse_word_acot:N
__fp_parse_word_atan:N 20936 { __fp_parse_function>NNN __fp_acot_o:Nw \use_i:nn }
__fp_parse_word_atand:N 20937 \cs_new:Npn __fp_parse_word_acotd:N
20938 { __fp_parse_function>NNN __fp_acot_o:Nw \use_ii:nn }
20939 \cs_new:Npn __fp_parse_word_atan:N
20940 { __fp_parse_function>NNN __fp_atan_o:Nw \use_i:nn }
20941 \cs_new:Npn __fp_parse_word_atand:N
20942 { __fp_parse_function>NNN __fp_atan_o:Nw \use_ii:nn }
```

(End definition for `\__fp_parse_word_acot:N` and others.)

### 34.1 Direct trigonometric functions

The approach for all trigonometric functions (sine, cosine, tangent, cotangent, cosecant, and secant), with arguments given in radians or in degrees, is the same.

- Filter out special cases ( $\pm 0$ ,  $\pm \infty$  and `NaN`).
- Keep the sign for later, and work with the absolute value  $|x|$  of the argument.
- Small numbers ( $|x| < 1$  in radians,  $|x| < 10$  in degrees) are converted to fixed point numbers (and to radians if  $|x|$  is in degrees).
- For larger numbers, we need argument reduction. Subtract a multiple of  $\pi/2$  (in degrees, 90) to bring the number to the range to  $[0, \pi/2)$  (in degrees,  $[0, 90)$ ).
- Reduce further to  $[0, \pi/4]$  (in degrees,  $[0, 45]$ ) using  $\sin x = \cos(\pi/2 - x)$ , and when working in degrees, convert to radians.
- Use the appropriate power series depending on the octant  $\lfloor \frac{x}{\pi/4} \rfloor \bmod 8$  (in degrees, the same formula with  $\pi/4 \rightarrow 45$ ), the sign, and the function to compute.

#### 34.1.1 Filtering special cases

`\__fp_sin_o:w` This function, and its analogs for `cos`, `csc`, `sec`, `tan`, and `cot` instead of `sin`, are followed either by `\use_i:nn` and a float in radians or by `\use_ii:nn` and a float in degrees. The sine of  $\pm 0$  or `NaN` is the same float. The sine of  $\pm \infty$  raises an invalid operation exception with the appropriate function name. Otherwise, call the `trig` function to perform argument reduction and if necessary convert the reduced argument to radians.

Then, `\_fp_sin_series_o>NNwww` is called to compute the Taylor series: this function receives a sign #3, an initial octant of 0, and the function `\_fp_ep_to_float_o:wwN` which converts the result of the series to a floating point directly rather than taking its inverse, since  $\sin(x) = \#3 \sin|x|$ .

```

20943 \cs_new:Npn _fp_sin_o:w #1 \s_fp _fp_chk:w #2#3#4; @
20944 {
20945 \if_case:w #2 \exp_stop_f:
20946 _fp_case_return_same_o:w
20947 \or: _fp_case_use:nw
20948 {
20949 _fp_trig:NNNNwn #1 _fp_sin_series_o:NNwww
20950 _fp_ep_to_float_o:wwN #3 0
20951 }
20952 \or: _fp_case_use:nw
20953 { _fp_invalid_operation_o:fw { #1 { sin } { sind } } }
20954 \else: _fp_case_return_same_o:w
20955 \fi:
20956 \s_fp _fp_chk:w #2 #3 #4;
20957 }
```

(End definition for `\_fp_sin_o:w`.)

`\_fp_cos_o:w` The cosine of  $\pm 0$  is 1. The cosine of  $\pm\infty$  raises an invalid operation exception. The cosine of NaN is itself. Otherwise, the `trig` function reduces the argument to at most half a right-angle and converts if necessary to radians. We then call the same series as for sine, but using a positive sign 0 regardless of the sign of  $x$ , and with an initial octant of 2, because  $\cos(x) = +\sin(\pi/2 + |x|)$ .

```

20958 \cs_new:Npn _fp_cos_o:w #1 \s_fp _fp_chk:w #2#3; @
20959 {
20960 \if_case:w #2 \exp_stop_f:
20961 _fp_case_return_o:Nw \c_one_fp
20962 \or: _fp_case_use:nw
20963 {
20964 _fp_trig:NNNNwn #1 _fp_sin_series_o:NNwww
20965 _fp_ep_to_float_o:wwN 0 2
20966 }
20967 \or: _fp_case_use:nw
20968 { _fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
20969 \else: _fp_case_return_same_o:w
20970 \fi:
20971 \s_fp _fp_chk:w #2 #3;
20972 }
```

(End definition for `\_fp_cos_o:w`.)

`\_fp_csc_o:w` The cosecant of  $\pm 0$  is  $\pm\infty$  with the same sign, with a division by zero exception (see `\_fp_cot_zero_o:Nfw` defined below), which requires the function name. The cosecant of  $\pm\infty$  raises an invalid operation exception. The cosecant of NaN is itself. Otherwise, the `trig` function performs the argument reduction, and converts if necessary to radians before calling the same series as for sine, using the sign #3, a starting octant of 0, and inverting during the conversion from the fixed point sine to the floating point result, because  $\csc(x) = \#3(\sin|x|)^{-1}$ .

```
20973 \cs_new:Npn _fp_csc_o:w #1 \s_fp _fp_chk:w #2#3#4; @
```

```

20974 {
20975 \if_case:w #2 \exp_stop_f:
20976 __fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
20977 \or: __fp_case_use:nw
20978 {
20979 __fp_trig:NNNNNwn #1 __fp_sin_series_o:NNwwww
20980 __fp_inv_to_float_o:wwN #3 0
20981 }
20982 \or: __fp_case_use:nw
20983 { __fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
20984 \else: __fp_case_return_same_o:w
20985 \fi:
20986 \s__fp __fp_chk:w #2 #3 #4;
20987 }

```

(End definition for  $\_\_fp_{csc\_o:w}$ .)

$\_\_fp_{sec\_o:w}$  The secant of  $\pm 0$  is 1. The secant of  $\pm\infty$  raises an invalid operation exception. The secant of  $\text{NaN}$  is itself. Otherwise, the **trig** function reduces the argument and turns it to radians before calling the same series as for sine, using a positive sign 0, a starting octant of 2, and inverting upon conversion, because  $\sec(x) = +1/\sin(\pi/2 + |x|)$ .

```

20988 \cs_new:Npn __fp_sec_o:w #1 \s__fp __fp_chk:w #2#3; @
20989 {
20990 \if_case:w #2 \exp_stop_f:
20991 __fp_case_return_o:Nw \c_one_fp
20992 \or: __fp_case_use:nw
20993 {
20994 __fp_trig:NNNNNwn #1 __fp_sin_series_o:NNwwww
20995 __fp_inv_to_float_o:wwN 0 2
20996 }
20997 \or: __fp_case_use:nw
20998 { __fp_invalid_operation_o:fw { #1 { sec } { secd } } }
20999 \else: __fp_case_return_same_o:w
21000 \fi:
21001 \s__fp __fp_chk:w #2 #3;
21002 }

```

(End definition for  $\_\_fp_{sec\_o:w}$ .)

$\_\_fp_{tan\_o:w}$  The tangent of  $\pm 0$  or  $\text{NaN}$  is the same floating point number. The tangent of  $\pm\infty$  raises an invalid operation exception. Once more, the **trig** function does the argument reduction step and conversion to radians before calling  $\_\_fp_{tan\_series\_o:NNwwww}$ , with a sign #3 and an initial octant of 1 (this shift is somewhat arbitrary). See  $\_\_fp_{cot\_o:w}$  for an explanation of the 0 argument.

```

21003 \cs_new:Npn __fp_tan_o:w #1 \s__fp __fp_chk:w #2#3#4; @
21004 {
21005 \if_case:w #2 \exp_stop_f:
21006 __fp_case_return_same_o:w
21007 \or: __fp_case_use:nw
21008 {
21009 __fp_trig:NNNNNwn #1
21010 __fp_tan_series_o:NNwwww 0 #3 1
21011 }
21012 \or: __fp_case_use:nw

```

```

21013 { __fp_invalid_operation_o:fw { #1 { tan } { tand } } }
21014 \else: __fp_case_return_same_o:w
21015 \fi:
21016 \s__fp __fp_chk:w #2 #3 #4;
21017 }

```

(End definition for `\__fp_tan_o:w`.)

`\__fp_cot_o:w`  
`\__fp_cot_zero_o:Nfw`

The cotangent of  $\pm 0$  is  $\pm\infty$  with the same sign, with a division by zero exception (see `\__fp_cot_zero_o:Nfw`). The cotangent of  $\pm\infty$  raises an invalid operation exception. The cotangent of NaN is itself. We use  $\cot x = -\tan(\pi/2 + x)$ , and the initial octant for the tangent was chosen to be 1, so the octant here starts at 3. The change in sign is obtained by feeding `\__fp_tan_series_o>NNwww` two signs rather than just the sign of the argument: the first of those indicates whether we compute tangent or cotangent. Those signs are eventually combined.

```

21018 \cs_new:Npn __fp_cot_o:w #1 \s__fp __fp_chk:w #2#3#4; @
21019 {
21020 \if_case:w #2 \exp_stop_f:
21021 __fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
21022 \or: __fp_case_use:nw
21023 {
21024 __fp_trig:NNNNNwn #1
21025 __fp_tan_series_o:NNwww 2 #3 3
21026 }
21027 \or: __fp_case_use:nw
21028 { __fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
21029 \else: __fp_case_return_same_o:w
21030 \fi:
21031 \s__fp __fp_chk:w #2 #3 #4;
21032 }
21033 \cs_new:Npn __fp_cot_zero_o:Nfw #1#2#3 \fi:
21034 {
21035 \fi:
21036 \token_if_eq_meaning:NNTF 0 #1
21037 { \exp_args:NNf __fp_division_by_zero_o:Nnw \c_inf_fp }
21038 { \exp_args:NNf __fp_division_by_zero_o:Nnw \c_minus_inf_fp }
21039 {#2}
21040 }

```

(End definition for `\__fp_cot_o:w` and `\__fp_cot_zero_o:Nfw`.)

### 34.1.2 Distinguishing small and large arguments

`\__fp_trig:NNNNNwn`

The first argument is `\use_i:nn` if the operand is in radians and `\use_ii:nn` if it is in degrees. Arguments #2 to #5 control what trigonometric function we compute, and #6 to #8 are pieces of a normal floating point number. Call the `_series` function #2, with arguments #3, either a conversion function (`\__fp_ep_to_float_o:wN` or `\__fp_ep_inv_to_float_o:wN`) or a sign 0 or 2 when computing tangent or cotangent; #4, a sign 0 or 2; the octant, computed in an integer expression starting with #5 and stopped by a period; and a fixed point number obtained from the floating point number by argument reduction (if necessary) and conversion to radians (if necessary). Any argument reduction adjusts the octant accordingly by leaving a (positive) shift into its integer expression. Let us explain the integer comparison. Two of the four `\exp_after:wN` are expanded, the

expansion hits the test, which is true if the float is at least 1 when working in radians, and at least 10 when working in degrees. Then one of the remaining `\exp_after:wN` hits #1, which picks the `trig` or `trigd` function in whichever branch of the conditional was taken. The final `\exp_after:wN` closes the conditional. At the end of the day, a number is `large` if it is  $\geq 1$  in radians or  $\geq 10$  in degrees, and `small` otherwise. All four `trig/trigd` auxiliaries receive the operand as an extended-precision number.

```

21041 \cs_new:Npn __fp_trig:NNNNwn #1#2#3#4#5 \s__fp __fp_chk:w 1#6#7#8;
21042 {
21043 \exp_after:wN #2
21044 \exp_after:wN #3
21045 \exp_after:wN #4
21046 \int_value:w __fp_int_eval:w #5
21047 \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
21048 \if_int_compare:w #7 > #1 0 1 \exp_stop_f:
21049 #1 __fp_trig_large:ww __fp_trigd_large:ww
21050 \else:
21051 #1 __fp_trig_small:ww __fp_trigd_small:ww
21052 \fi:
21053 #7,#8{0000}{0000};
21054 }
```

(End definition for `\__fp_trig:NNNNwn`.)

### 34.1.3 Small arguments

`\__fp_trig_small:ww`

This receives a small extended-precision number in radians and converts it to a fixed point number. Some trailing digits may be lost in the conversion, so we keep the original floating point number around: when computing sine or tangent (or their inverses), the last step is to multiply by the floating point number (as an extended-precision number) rather than the fixed point number. The period serves to end the integer expression for the octant.

```

21055 \cs_new:Npn __fp_trig_small:ww #1,#2;
21056 { __fp_ep_to_fixed:wwn #1,#2; . #1,#2; }
```

(End definition for `\__fp_trig_small:ww`.)

`\__fp_trigd_small:ww`

Convert the extended-precision number to radians, then call `\__fp_trig_small:ww` to massage it in the form appropriate for the `_series` auxiliary.

```

21057 \cs_new:Npn __fp_trigd_small:ww #1,#2;
21058 {
21059 __fp_ep_mul_raw:wwwwN
21060 -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
21061 __fp_trig_small:ww
21062 }
```

(End definition for `\__fp_trigd_small:ww`.)

### 34.1.4 Argument reduction in degrees

`\__fp_trigd_large:ww`  
`\__fp_trigd_large_auxi:nmmnwNNNN`  
`\__fp_trigd_large_auxii:wNw`  
`\__fp_trigd_large_auxiii:www`

Note that  $25 \times 360 = 9000$ , so  $10^{k+1} \equiv 10^k \pmod{360}$  for  $k \geq 3$ . When the exponent #1 is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is  $100 \times \langle block_1 \rangle \cdots \langle block_3 \rangle \langle block_4 \rangle \cdots \langle block_7 \rangle$ , or is equal to

it modulo 360 if the exponent #1 is very large. The first auxiliary finds  $\langle block_1 \rangle + \langle block_2 \rangle$  (mod 9), a single digit, and prepends it to the 4 digits of  $\langle block_3 \rangle$ . It also unpacks  $\langle block_4 \rangle$  and grabs the 4 digits of  $\langle block_7 \rangle$ . The second auxiliary grabs the  $\langle block_3 \rangle$  plus any contribution from the first two blocks as #1, the first digit of  $\langle block_4 \rangle$  (just after the decimal point in hundreds of degrees) as #2, and the three other digits as #3. It finds the quotient and remainder of #1#2 modulo 9, adds twice the quotient to the integer expression for the octant, and places the remainder (between 0 and 8) before #3 to form a new  $\langle block_4 \rangle$ . The resulting fixed point number is  $x \in [0, 0.9]$ . If  $x \geq 0.45$ , we add 1 to the octant and feed  $0.9 - x$  with an exponent of 2 (to compensate the fact that we are working in units of hundreds of degrees rather than degrees) to  $\text{\_\_fp\_trigd\_small:ww}$ . Otherwise, we feed it  $x$  with an exponent of 2. The third auxiliary also discards digits which were not packed into the various  $\langle blocks \rangle$ . Since the original exponent #1 is at least 2, those are all 0 and no precision is lost (#6 and #7 are four 0 each).

```

21063 \cs_new:Npn __fp_trigd_large:ww #1, #2#3#4#5#6#7;
21064 {
21065 \exp_after:wN __fp_pack_eight:wNNNNNNNN
21066 \exp_after:wN __fp_pack_eight:wNNNNNNNN
21067 \exp_after:wN __fp_pack_twice_four:wNNNNNNNN
21068 \exp_after:wN __fp_pack_twice_four:wNNNNNNNN
21069 \exp_after:wN __fp_trigd_large_auxi:nnnnwNNNN
21070 \exp_after:wN ;
21071 \exp:w \exp_end_continue_f:w
21072 \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
21073 #2#3#4#5#6#7 0000 0000 0000 !
21074 }
21075 \cs_new:Npn __fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
21076 {
21077 \exp_after:wN __fp_trigd_large_auxii:wNw
21078 \int_value:w __fp_int_eval:w #1 + #2
21079 - (#1 + #2 - 4) / 9 * 9 __fp_int_eval_end:
21080 #3;
21081 #4; #5{#6#7#8#9};
21082 }
21083 \cs_new:Npn __fp_trigd_large_auxii:wNw #1; #2#3;
21084 {
21085 + (#1#2 - 4) / 9 * 2
21086 \exp_after:wN __fp_trigd_large_auxiii:www
21087 \int_value:w __fp_int_eval:w #1#2
21088 - (#1#2 - 4) / 9 * 9 __fp_int_eval_end: #3 ;
21089 }
21090 \cs_new:Npn __fp_trigd_large_auxiii:www #1; #2; #3!
21091 {
21092 \if_int_compare:w #1 < 4500 \exp_stop_f:
21093 \exp_after:wN __fp_use_i_until_s:nw
21094 \exp_after:wN __fp_fixed_continue:wn
21095 \else:
21096 + 1
21097 \fi:
21098 __fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000}{0000};
21099 {#1}#2{0000}{0000};
21100 { __fp_trigd_small:ww 2, }
21101 }
```

(End definition for `\_fp_trigd_large:ww` and others.)

### 34.1.5 Argument reduction in radians

Arguments greater or equal to 1 need to be reduced to a range where we only need a few terms of the Taylor series. We reduce to the range  $[0, 2\pi]$  by subtracting multiples of  $2\pi$ , then to the smaller range  $[0, \pi/2]$  by subtracting multiples of  $\pi/2$  (keeping track of how many times  $\pi/2$  is subtracted), then to  $[0, \pi/4]$  by mapping  $x \rightarrow \pi/2 - x$  if appropriate. When the argument is very large, say,  $10^{100}$ , an equally large multiple of  $2\pi$  must be subtracted, hence we must work with a very good approximation of  $2\pi$  in order to get a sensible remainder modulo  $2\pi$ .

Specifically, we multiply the argument by an approximation of  $1/(2\pi)$  with 10048 digits, then discard the integer part of the result, keeping 52 digits of the fractional part. From the fractional part of  $x/(2\pi)$  we deduce the octant (quotient of the first three digits by 125). We then multiply by 8 or  $-8$  (the latter when the octant is odd), ignore any integer part (related to the octant), and convert the fractional part to an extended precision number, before multiplying by  $\pi/4$  to convert back to a value in radians in  $[0, \pi/4]$ .

It is possible to prove that given the precision of floating points and their range of exponents, the 52 digits may start at most with 24 zeros. The 5 last digits are affected by carries from computations which are not done, hence we are left with at least  $52 - 24 - 5 = 23$  significant digits, enough to round correctly up to  $0.6 \cdot \text{ulp}$  in all cases.

`\c_fp_trig_intarray` This integer array stores blocks of 8 decimals of  $10^{-16}/(2\pi)$ . Each entry is  $10^8$  plus an 8 digit number storing 8 decimals. In total we store 10112 decimals of  $10^{-16}/(2\pi)$ . The number of decimals we really need is the maximum exponent plus the number of digits we later need, 52, plus 12 (4 – 1 groups of 4 digits). The memory footprint (1/2 byte per digit) is the same as an earlier method of storing the data as a control sequence name, but the major advantage is that we can unpack specific subsets of the digits without unpacking the 10112 decimals.

```

21102 \intarray_const_from_clist:Nn \c_fp_trig_intarray
21103 {
21104 100000000, 100000000, 115915494, 130918953, 135768883, 176337251,
21105 143620344, 159645740, 145644874, 176673440, 158896797, 163422653,
21106 150901138, 102766253, 108595607, 128427267, 157958036, 189291184,
21107 161145786, 152877967, 141073169, 198392292, 139966937, 140907757,
21108 130777463, 196925307, 168871739, 128962173, 197661693, 136239024,
21109 117236290, 111832380, 111422269, 197557159, 140461890, 108690267,
21110 139561204, 189410936, 193784408, 155287230, 199946443, 140024867,
21111 123477394, 159610898, 132309678, 130749061, 166986462, 180469944,
21112 186521878, 181574786, 156696424, 110389958, 174139348, 160998386,
21113 180991999, 162442875, 158517117, 188584311, 117518767, 116054654,
21114 175369880, 109739460, 136475933, 137680593, 102494496, 163530532,
21115 171567755, 103220324, 177781639, 171660229, 146748119, 159816584,
21116 106060168, 103035998, 113391198, 174988327, 186654435, 127975507,
21117 100162406, 177564388, 184957131, 108801221, 199376147, 168137776,
21118 147378906, 133068046, 145797848, 117613124, 127314069, 196077502,
21119 145002977, 159857089, 105690279, 167851315, 125210016, 131774602,
21120 109248116, 106240561, 145620314, 164840892, 148459191, 143521157,
21121 154075562, 100871526, 160680221, 171591407, 157474582, 172259774,
21122 162853998, 175155329, 139081398, 117724093, 158254797, 107332871,
21123 190406999, 175907657, 170784934, 170393589, 182808717, 134256403,
```

21124 166895116, 162545705, 194332763, 112686500, 126122717, 197115321,  
 21125 112599504, 138667945, 103762556, 108363171, 116952597, 158128224,  
 21126 194162333, 143145106, 112353687, 185631136, 136692167, 114206974,  
 21127 169601292, 150578336, 105311960, 185945098, 139556718, 170995474,  
 21128 165104316, 123815517, 158083944, 129799709, 199505254, 138756612,  
 21129 194458833, 106846050, 178529151, 151410404, 189298850, 163881607,  
 21130 176196993, 107341038, 199957869, 118905980, 193737772, 106187543,  
 21131 122271893, 101366255, 126123878, 103875388, 181106814, 106765434,  
 21132 108282785, 126933426, 179955607, 107903860, 160352738, 199624512,  
 21133 159957492, 176297023, 159409558, 143011648, 129641185, 157771240,  
 21134 157544494, 157021789, 176979240, 194903272, 194770216, 164960356,  
 21135 153181535, 144003840, 168987471, 176915887, 163190966, 150696440,  
 21136 147769706, 187683656, 177810477, 197954503, 153395758, 130188183,  
 21137 186879377, 166124814, 195305996, 155802190, 183598751, 103512712,  
 21138 190432315, 180498719, 168687775, 194656634, 162210342, 104440855,  
 21139 149785037, 192738694, 129353661, 193778292, 187359378, 143470323,  
 21140 102371458, 137923557, 111863634, 119294601, 183182291, 196416500,  
 21141 187830793, 131353497, 179099745, 186492902, 167450609, 189368909,  
 21142 145883050, 133703053, 180547312, 132158094, 131976760, 132283131,  
 21143 141898097, 149822438, 133517435, 169898475, 101039500, 168388003,  
 21144 197867235, 199608024, 100273901, 108749548, 154787923, 156826113,  
 21145 199489032, 168997427, 108349611, 149208289, 103776784, 174303550,  
 21146 145684560, 183671479, 130845672, 133270354, 185392556, 120208683,  
 21147 193240995, 162211753, 131839402, 109707935, 170774965, 149880868,  
 21148 160663609, 168661967, 103747454, 121028312, 119251846, 122483499,  
 21149 111611495, 166556037, 196967613, 199312829, 196077608, 127799010,  
 21150 107830360, 102338272, 198790854, 102387615, 157445430, 192601191,  
 21151 100543379, 198389046, 154921248, 129516070, 172853005, 122721023,  
 21152 160175233, 113173179, 175931105, 103281551, 109373913, 163964530,  
 21153 157926071, 180083617, 195487672, 146459804, 173977292, 144810920,  
 21154 109371257, 186918332, 189588628, 139904358, 168666639, 175673445,  
 21155 114095036, 137327191, 174311388, 106638307, 125923027, 159734506,  
 21156 105482127, 178037065, 133778303, 121709877, 134966568, 149080032,  
 21157 169885067, 141791464, 168350828, 116168533, 114336160, 173099514,  
 21158 198531198, 119733758, 144420984, 116559541, 152250643, 139431286,  
 21159 144403838, 183561508, 179771645, 101706470, 167518774, 156059160,  
 21160 187168578, 157939226, 123475633, 117111329, 198655941, 159689071,  
 21161 198506887, 144230057, 151919770, 156900382, 118392562, 120338742,  
 21162 135362568, 108354156, 151729710, 188117217, 195936832, 156488518,  
 21163 174997487, 108553116, 159830610, 113921445, 144601614, 188452770,  
 21164 125114110, 170248521, 173974510, 138667364, 103872860, 109967489,  
 21165 131735618, 112071174, 104788993, 168886556, 192307848, 150230570,  
 21166 157144063, 163863202, 136852010, 174100574, 185922811, 115721968,  
 21167 100397824, 175953001, 166958522, 112303464, 118773650, 143546764,  
 21168 164565659, 171901123, 108476709, 193097085, 191283646, 166919177,  
 21169 169387914, 133315566, 150669813, 121641521, 100895711, 172862384,  
 21170 126070678, 145176011, 113450800, 169947684, 122356989, 162488051,  
 21171 157759809, 153397080, 185475059, 175362656, 149034394, 145420581,  
 21172 178864356, 183042000, 131509559, 147434392, 152544850, 167491429,  
 21173 108647514, 142303321, 133245695, 111634945, 167753939, 142403609,  
 21174 105438335, 152829243, 142203494, 184366151, 146632286, 102477666,  
 21175 166049531, 140657343, 157553014, 109082798, 180914786, 169343492,  
 21176 127376026, 134997829, 195701816, 119643212, 133140475, 176289748,  
 21177 140828911, 174097478, 126378991, 181699939, 148749771, 151989818,

|       |                                                                   |
|-------|-------------------------------------------------------------------|
| 21178 | 172666294, 160183053, 195832752, 109236350, 168538892, 128468247, |
| 21179 | 125997252, 183007668, 156937583, 165972291, 198244297, 147406163, |
| 21180 | 181831139, 158306744, 134851692, 185973832, 137392662, 140243450, |
| 21181 | 119978099, 140402189, 161348342, 173613676, 144991382, 171541660, |
| 21182 | 163424829, 136374185, 106122610, 186132119, 198633462, 184709941, |
| 21183 | 183994274, 129559156, 128333990, 148038211, 175011612, 111667205, |
| 21184 | 119125793, 103552929, 124113440, 131161341, 112495318, 138592695, |
| 21185 | 184904438, 146807849, 109739828, 108855297, 104515305, 139914009, |
| 21186 | 188698840, 188365483, 166522246, 168624087, 125401404, 100911787, |
| 21187 | 142122045, 123075334, 173972538, 114940388, 141905868, 142311594, |
| 21188 | 163227443, 139066125, 116239310, 162831953, 123883392, 113153455, |
| 21189 | 163815117, 152035108, 174595582, 101123754, 135976815, 153401874, |
| 21190 | 107394340, 136339780, 138817210, 104531691, 182951948, 179591767, |
| 21191 | 139541778, 179243527, 161740724, 160593916, 102732282, 187946819, |
| 21192 | 136491289, 149714953, 143255272, 135916592, 198072479, 198580612, |
| 21193 | 169007332, 118844526, 179433504, 155801952, 149256630, 162048766, |
| 21194 | 116134365, 133992028, 175452085, 155344144, 109905129, 182727454, |
| 21195 | 165911813, 122232840, 151166615, 165070983, 175574337, 129548631, |
| 21196 | 120411217, 116380915, 160616116, 157320000, 183306114, 160618128, |
| 21197 | 103262586, 195951602, 146321661, 138576614, 180471993, 127077713, |
| 21198 | 116441201, 159496011, 106328305, 120759583, 148503050, 179095584, |
| 21199 | 198298218, 167402898, 138551383, 123957020, 180763975, 150429225, |
| 21200 | 198476470, 171016426, 197438450, 143091658, 164528360, 132493360, |
| 21201 | 143546572, 137557916, 113663241, 120457809, 196971566, 134022158, |
| 21202 | 180545794, 131328278, 100552461, 132088901, 187421210, 192448910, |
| 21203 | 141005215, 149680971, 113720754, 100571096, 134066431, 135745439, |
| 21204 | 191597694, 135788920, 179342561, 177830222, 137011486, 142492523, |
| 21205 | 192487287, 113132021, 176673607, 156645598, 127260957, 141566023, |
| 21206 | 143787436, 129132109, 174858971, 150713073, 191040726, 143541417, |
| 21207 | 197057222, 165479803, 181512759, 157912400, 125344680, 148220261, |
| 21208 | 173422990, 101020483, 106246303, 137964746, 178190501, 181183037, |
| 21209 | 151538028, 179523433, 141955021, 135689770, 191290561, 143178787, |
| 21210 | 192086205, 174499925, 178975690, 118492103, 124206471, 138519113, |
| 21211 | 188147564, 102097605, 154895793, 178514140, 141453051, 151583964, |
| 21212 | 128232654, 106020603, 131189158, 165702720, 186250269, 191639375, |
| 21213 | 115278873, 160608114, 155694842, 110322407, 177272742, 116513642, |
| 21214 | 134366992, 171634030, 194053074, 180652685, 109301658, 192136921, |
| 21215 | 141431293, 171341061, 157153714, 106203978, 147618426, 150297807, |
| 21216 | 186062669, 169960809, 118422347, 163350477, 146719017, 145045144, |
| 21217 | 161663828, 146208240, 186735951, 102371302, 190444377, 194085350, |
| 21218 | 134454426, 133413062, 163074595, 113830310, 122931469, 134466832, |
| 21219 | 185176632, 182415152, 110179422, 164439571, 181217170, 121756492, |
| 21220 | 119644493, 196532222, 118765848, 182445119, 109401340, 150443213, |
| 21221 | 198586286, 121083179, 139396084, 143898019, 114787389, 177233102, |
| 21222 | 186310131, 148695521, 126205182, 178063494, 157118662, 177825659, |
| 21223 | 188310053, 151552316, 165984394, 109022180, 163144545, 121212978, |
| 21224 | 197344714, 188741258, 126822386, 102360271, 109981191, 152056882, |
| 21225 | 134723983, 158013366, 106837863, 128867928, 161973236, 172536066, |
| 21226 | 185216856, 132011948, 197807339, 158419190, 166595838, 167852941, |
| 21227 | 124187182, 117279875, 106103946, 106481958, 157456200, 160892122, |
| 21228 | 184163943, 173846549, 158993202, 184812364, 133466119, 170732430, |
| 21229 | 195458590, 173361878, 162906318, 150165106, 126757685, 112163575, |
| 21230 | 188696307, 145199922, 100107766, 176830946, 198149756, 122682434, |
| 21231 | 179367131, 108412102, 119520899, 148191244, 140487511, 171059184, |

|       |                                                                   |
|-------|-------------------------------------------------------------------|
| 21232 | 141399078, 189455775, 118462161, 190415309, 134543802, 180893862, |
| 21233 | 180732375, 178615267, 179711433, 123241969, 185780563, 176301808, |
| 21234 | 184386640, 160717536, 183213626, 129671224, 126094285, 140110963, |
| 21235 | 121826276, 151201170, 122552929, 128965559, 146082049, 138409069, |
| 21236 | 107606920, 103954646, 119164002, 115673360, 117909631, 187289199, |
| 21237 | 186343410, 186903200, 157966371, 103128612, 135698881, 176403642, |
| 21238 | 152540837, 109810814, 183519031, 121318624, 172281810, 150845123, |
| 21239 | 169019064, 166322359, 138872454, 163073727, 128087898, 130041018, |
| 21240 | 194859136, 173742589, 141812405, 167291912, 138003306, 134499821, |
| 21241 | 196315803, 186381054, 124578934, 150084553, 128031351, 118843410, |
| 21242 | 107373060, 159565443, 173624887, 171292628, 198074235, 139074061, |
| 21243 | 178690578, 144431052, 174262641, 176783005, 182214864, 162289361, |
| 21244 | 192966929, 192033046, 169332843, 181580535, 164864073, 118444059, |
| 21245 | 195496893, 153773183, 167266131, 130108623, 158802128, 180432893, |
| 21246 | 144562140, 147978945, 142337360, 158506327, 104399819, 132635916, |
| 21247 | 168734194, 136567839, 101281912, 120281622, 195003330, 112236091, |
| 21248 | 185875592, 101959081, 122415367, 194990954, 148881099, 175891989, |
| 21249 | 108115811, 163538891, 163394029, 123722049, 184837522, 142362091, |
| 21250 | 100834097, 156679171, 100841679, 157022331, 178971071, 102928884, |
| 21251 | 189701309, 195339954, 124415335, 106062584, 139214524, 133864640, |
| 21252 | 134324406, 157317477, 155340540, 144810061, 177612569, 108474646, |
| 21253 | 114329765, 143900008, 138265211, 145210162, 136643111, 197987319, |
| 21254 | 102751191, 144121361, 169620456, 193602633, 161023559, 162140467, |
| 21255 | 102901215, 167964187, 135746835, 187317233, 110047459, 163339773, |
| 21256 | 124770449, 118885134, 141536376, 100915375, 164267438, 145016622, |
| 21257 | 113937193, 106748706, 128815954, 164819775, 119220771, 102367432, |
| 21258 | 189062690, 170911791, 194127762, 112245117, 123546771, 115640433, |
| 21259 | 135772061, 166615646, 174474627, 130562291, 133320309, 153340551, |
| 21260 | 138417181, 194605321, 150142632, 180008795, 151813296, 175497284, |
| 21261 | 167018836, 157425342, 150169942, 131069156, 134310662, 160434122, |
| 21262 | 105213831, 158797111, 150754540, 163290657, 102484886, 148697402, |
| 21263 | 187203725, 198692811, 149360627, 140384233, 128749423, 132178578, |
| 21264 | 177507355, 171857043, 178737969, 134023369, 102911446, 196144864, |
| 21265 | 197697194, 134527467, 144296030, 189437192, 154052665, 188907106, |
| 21266 | 162062575, 150993037, 199766583, 167936112, 181374511, 104971506, |
| 21267 | 115378374, 135795558, 167972129, 135876446, 130937572, 103221320, |
| 21268 | 124605656, 161129971, 131027586, 191128460, 143251843, 143269155, |
| 21269 | 129284585, 173495971, 150425653, 199302112, 118494723, 121323805, |
| 21270 | 116549802, 190991967, 168151180, 122483192, 151273721, 199792134, |
| 21271 | 133106764, 121874844, 126215985, 112167639, 167793529, 182985195, |
| 21272 | 185453921, 106957880, 158685312, 132775454, 133229161, 198905318, |
| 21273 | 190537253, 191582222, 192325972, 178133427, 181825606, 148823337, |
| 21274 | 160719681, 101448145, 131983362, 137910767, 112550175, 128826351, |
| 21275 | 183649210, 135725874, 110356573, 189469487, 154446940, 118175923, |
| 21276 | 106093708, 128146501, 185742532, 149692127, 164624247, 183221076, |
| 21277 | 154737505, 168198834, 156410354, 158027261, 125228550, 131543250, |
| 21278 | 139591848, 191898263, 104987591, 115406321, 103542638, 190012837, |
| 21279 | 142615518, 178773183, 175862355, 117537850, 169565995, 170028011, |
| 21280 | 158412588, 170150030, 117025916, 174630208, 142412449, 112839238, |
| 21281 | 105257725, 114737141, 123102301, 172563968, 130555358, 132628403, |
| 21282 | 183638157, 168682846, 143304568, 105994018, 170010719, 152092970, |
| 21283 | 117799058, 132164175, 179868116, 158654714, 177489647, 116547948, |
| 21284 | 183121404, 131836079, 184431405, 157311793, 149677763, 173989893, |
| 21285 | 102277656, 107058530, 140837477, 152640947, 143507039, 152145247, |

```

21286 101683884, 107090870, 161471944, 137225650, 128231458, 172995869,
21287 173831689, 171268519, 139042297, 111072135, 107569780, 137262545,
21288 181410950, 138270388, 198736451, 162848201, 180468288, 120582913,
21289 153390138, 135649144, 130040157, 106509887, 192671541, 174507066,
21290 186888783, 143805558, 135011967, 145862340, 180595327, 124727843,
21291 182925939, 157715840, 136885940, 198993925, 152416883, 178793572,
21292 179679516, 154076673, 192703125, 164187609, 162190243, 104699348,
21293 159891990, 160012977, 174692145, 132970421, 167781726, 115178506,
21294 153008552, 155999794, 102099694, 155431545, 127458567, 104403686,
21295 168042864, 184045128, 181182309, 179349696, 127218364, 192935516,
21296 120298724, 169583299, 148193297, 183358034, 159023227, 105261254,
21297 121144370, 184359584, 194433836, 138388317, 175184116, 108817112,
21298 151279233, 137457721, 193398208, 119005406, 132929377, 175306906,
21299 160741530, 149976826, 147124407, 176881724, 186734216, 185881509,
21300 191334220, 175930947, 117385515, 193408089, 157124410, 163472089,
21301 131949128, 180783576, 131158294, 100549708, 191802336, 165960770,
21302 170927599, 101052702, 181508688, 197828549, 143403726, 142729262,
21303 110348701, 139928688, 153550062, 106151434, 130786653, 196085995,
21304 100587149, 139141652, 106530207, 100852656, 124074703, 166073660,
21305 153338052, 163766757, 120188394, 197277047, 122215363, 138511354,
21306 183463624, 161985542, 159938719, 133367482, 104220974, 149956672,
21307 170250544, 164232439, 157506869, 159133019, 137469191, 142980999,
21308 134242305, 150172665, 121209241, 145596259, 160554427, 159095199,
21309 168243130, 184279693, 171132070, 121049823, 123819574, 171759855,
21310 119501864, 163094029, 175943631, 194450091, 191506160, 149228764,
21311 132319212, 197034460, 193584259, 126727638, 168143633, 109856853,
21312 127860243, 132141052, 133076065, 188414958, 158718197, 107124299,
21313 159592267, 181172796, 144388537, 196763139, 127431422, 179531145,
21314 100064922, 112650013, 132686230, 121550837,
21315 }

```

(End definition for \c\_fp\_trig\_intarray.)

\\_fp\_trig\\_large:ww  
\\_fp\_trig\\_large\\_auxi:w  
\\_fp\_trig\\_large\\_auxii:w  
\\_fp\_trig\\_large\\_auxiii:w

The exponent #1 is between 1 and 10000. We wish to look up decimals  $10^{\#1-16}/(2\pi)$  starting from the digit #1 + 1. Since they are stored in batches of 8, compute  $\lceil \#1/8 \rceil$  and fetch blocks of 8 digits starting there. The numbering of items in \c\_fp\_trig\_intarray starts at 1, so the block  $\lceil \#1/8 \rceil + 1$  contains the digit we want, at one of the eight positions. Each call to \int\_value:w \\_kernel\_intarray\_item:Nn expands the next, until being stopped by \\_fp\_trig\\_large\\_auxiii:w using \exp\_stop\_f:. Once all these blocks are unpacked, the \exp\_stop\_f: and 0 to 7 digits are removed by \use\_none:n...n. Finally, \\_fp\_trig\\_large\\_auxii:w packs 64 digits (there are between 65 and 72 at this point) into groups of 4 and the auxv auxiliary is called.

```

21316 \cs_new:Npn _fp_trig_large:ww #1, #2#3#4#5#6;
21317 {
21318 \exp_after:wN _fp_trig_large_auxi:w
21319 \int_value:w _fp_int_eval:w (#1 - 4) / 8 \exp_after:wN ,
21320 \int_value:w #1 , ;
21321 {#2}{#3}{#4}{#5} ;
21322 }
21323 \cs_new:Npn _fp_trig_large_auxi:w #1, #2,
21324 {
21325 \exp_after:wN \exp_after:wN
21326 \exp_after:wN _fp_trig_large_auxii:w
21327 \cs:w

```

```

21328 use_none:n \prg_replicate:nn { #2 - #1 * 8 } { n }
21329 \exp_after:wN
21330 \cs_end:
21331 \int_value:w
21332 __kernel_intarray_item:Nn \c_fp_trig_intarray
21333 { __fp_int_eval:w #1 + 1 \scan_stop: }
21334 \exp_after:wN __fp_trig_large_auxiii:w \int_value:w
21335 __kernel_intarray_item:Nn \c_fp_trig_intarray
21336 { __fp_int_eval:w #1 + 2 \scan_stop: }
21337 \exp_after:wN __fp_trig_large_auxiii:w \int_value:w
21338 __kernel_intarray_item:Nn \c_fp_trig_intarray
21339 { __fp_int_eval:w #1 + 3 \scan_stop: }
21340 \exp_after:wN __fp_trig_large_auxiii:w \int_value:w
21341 __kernel_intarray_item:Nn \c_fp_trig_intarray
21342 { __fp_int_eval:w #1 + 4 \scan_stop: }
21343 \exp_after:wN __fp_trig_large_auxiii:w \int_value:w
21344 __kernel_intarray_item:Nn \c_fp_trig_intarray
21345 { __fp_int_eval:w #1 + 5 \scan_stop: }
21346 \exp_after:wN __fp_trig_large_auxiii:w \int_value:w
21347 __kernel_intarray_item:Nn \c_fp_trig_intarray
21348 { __fp_int_eval:w #1 + 6 \scan_stop: }
21349 \exp_after:wN __fp_trig_large_auxiii:w \int_value:w
21350 __kernel_intarray_item:Nn \c_fp_trig_intarray
21351 { __fp_int_eval:w #1 + 7 \scan_stop: }
21352 \exp_after:wN __fp_trig_large_auxiii:w \int_value:w
21353 __kernel_intarray_item:Nn \c_fp_trig_intarray
21354 { __fp_int_eval:w #1 + 8 \scan_stop: }
21355 \exp_after:wN __fp_trig_large_auxiii:w \int_value:w
21356 __kernel_intarray_item:Nn \c_fp_trig_intarray
21357 { __fp_int_eval:w #1 + 9 \scan_stop: }
21358 \exp_stop_f:
21359 }
21360 \cs_new:Npn __fp_trig_large_auxii:w
21361 {
21362 __fp_pack_twice_four:wNNNNNNNN __fp_pack_twice_four:wNNNNNNNN
21363 __fp_pack_twice_four:wNNNNNNNN __fp_pack_twice_four:wNNNNNNNN
21364 __fp_pack_twice_four:wNNNNNNNN __fp_pack_twice_four:wNNNNNNNN
21365 __fp_pack_twice_four:wNNNNNNNN __fp_pack_twice_four:wNNNNNNNN
21366 __fp_trig_large_auxv:www ;
21367 }
21368 \cs_new:Npn __fp_trig_large_auxiii:w 1 { \exp_stop_f: }

```

(End definition for `\__fp_trig_large:ww` and others.)

`\__fp_trig_large_auxv:www`  
`\__fp_trig_large_auxvi:wnnnnnnn`  
`\__fp_trig_large_pack:NNNNNw`

First come the first 64 digits of the fractional part of  $10^{#1-16}/(2\pi)$ , arranged in 16 blocks of 4, and ending with a semicolon. Then a few more digits of the same fractional part, ending with a semicolon, then 4 blocks of 4 digits holding the significand of the original argument. Multiply the 16-digit significand with the 64-digit fractional part: the `auxvi` auxiliary receives the significand as `#2#3#4#5` and 16 digits of the fractional part as `#6#7#8#9`, and computes one step of the usual ladder of `pack` functions we use for multiplication (see *e.g.*, `\__fp_fixed_mul:wwn`), then discards one block of the fractional part to set things up for the next step of the ladder. We perform 13 such steps, replacing the last `middle` shift by the appropriate `trailing` shift, then discard the significand and remaining 3 blocks from the fractional part, as there are not enough digits to compute

any more step in the ladder. The last semicolon closes the ladder, and we return control to the `auxvii` auxiliary.

```

21369 \cs_new:Npn __fp_trig_large_auxv:www #1; #2; #3;
21370 {
21371 \exp_after:wN __fp_use_i_until_s:nw
21372 \exp_after:wN __fp_trig_large_auxvii:w
21373 \int_value:w __fp_int_eval:w \c__fp_leading_shift_int
21374 \prg_replicate:nn { 13 }
21375 { __fp_trig_large_auxvi:wnnnnnnnn }
21376 + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
21377 __fp_use_i_until_s:nw
21378 ; #3 #1 ;
21379 }
21380 \cs_new:Npn __fp_trig_large_auxvi:wnnnnnnnn #1; #2#3#4#5#6#7#8#9
21381 {
21382 \exp_after:wN __fp_trig_large_pack:NNNNNw
21383 \int_value:w __fp_int_eval:w \c__fp_middle_shift_int
21384 + #2*#9 + #3*#8 + #4*#7 + #5*#6
21385 #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
21386 }
21387 \cs_new:Npn __fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
21388 { + #1#2#3#4#5 ; #6 }

(End definition for __fp_trig_large_auxv:www, __fp_trig_large_auxvi:wnnnnnnnn, and __fp-
trig_large_pack:NNNNNw.)
```

The `auxvii` auxiliary is followed by 52 digits and a semicolon. We find the octant as the integer part of 8 times what follows, or equivalently as the integer part of  $\#1\#2\#3/125$ , and add it to the surrounding integer expression for the octant. We then compute 8 times the 52-digit number, with a minus sign if the octant is odd. Again, the last `middle` shift is converted to a `trailing` shift. Any integer part (including negative values which come up when the octant is odd) is discarded by `\__fp_use_i_until_s:nw`. The resulting fractional part should then be converted to radians by multiplying by  $2\pi/8$ , but first, build an extended precision number by abusing `\__fp_ep_to_ep_loop:N` with the appropriate trailing markers. Finally, `\__fp_trig_small:ww` sets up the argument for the functions which compute the Taylor series.

```

21389 \cs_new:Npn __fp_trig_large_auxvii:w #1#2#3
21390 {
21391 \exp_after:wN __fp_trig_large_auxviii:ww
21392 \int_value:w __fp_int_eval:w (#1#2#3 - 62) / 125 ;
21393 #1#2#3
21394 }
21395 \cs_new:Npn __fp_trig_large_auxviii:ww #1;
21396 {
21397 + #1
21398 \if_int_odd:w #1 \exp_stop_f:
21399 \exp_after:wN __fp_trig_large_auxix:Nw
21400 \exp_after:wN -
21401 \else:
21402 \exp_after:wN __fp_trig_large_auxix:Nw
21403 \exp_after:wN +
21404 \fi:
21405 }
```

```

21406 \cs_new:Npn __fp_trig_large_auxix:Nw
21407 {
21408 \exp_after:wN __fp_use_i_until_s:nw
21409 \exp_after:wN __fp_trig_large_auxxi:w
21410 \int_value:w __fp_int_eval:w \c__fp_leading_shift_int
21411 \prg_replicate:nn { 13 }
21412 { __fp_trig_large_auxx:wNNNNN }
21413 + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
21414 ;
21415 }
21416 \cs_new:Npn __fp_trig_large_auxxx:wNNNNN #1; #2 #3#4#5#6
21417 {
21418 \exp_after:wN __fp_trig_large_pack:NNNNNw
21419 \int_value:w __fp_int_eval:w \c__fp_middle_shift_int
21420 #2 8 * #3#4#5#6
21421 #1; #2
21422 }
21423 \cs_new:Npn __fp_trig_large_auxxi:w #1;
21424 {
21425 \exp_after:wN __fp_ep_mul_raw:wwwwN
21426 \int_value:w __fp_int_eval:w 0 __fp_ep_to_ep_loop:N #1 ; ; !
21427 0,{7853}{9816}{3397}{4483}{0961}{5661};
21428 __fp_trig_small:ww
21429 }

```

(End definition for `\__fp_trig_large_auxvii:w` and others.)

### 34.1.6 Computing the power series

Here we receive a conversion function `\__fp_ep_to_float_o:wwN` or `\__fp_ep_inv_to_float_o:wwN`, a  $\langle sign \rangle$  (0 or 2), a (non-negative)  $\langle octant \rangle$  delimited by a dot, a  $\langle fixed point \rangle$  number delimited by a semicolon, and an extended-precision number. The auxiliary receives:

- the conversion function #1;
- the final sign, which depends on the octant #3 and the sign #2;
- the octant #3, which controls the series we use;
- the square #4 \* #4 of the argument as a fixed point number, computed with `\__fp_fixed_mul:wwn`;
- the number itself as an extended-precision number.

If the octant is in  $\{1, 2, 5, 6, \dots\}$ , we are near an extremum of the function and we use the series

$$\cos(x) = 1 - x^2 \left( \frac{1}{2!} - x^2 \left( \frac{1}{4!} - x^2 \left( \dots \right) \right) \right).$$

Otherwise, the series

$$\sin(x) = x \left( 1 - x^2 \left( \frac{1}{3!} - x^2 \left( \frac{1}{5!} - x^2 \left( \dots \right) \right) \right) \right)$$

is used. Finally, the extended-precision number is converted to a floating point number with the given sign, and `\_fp_sanitize:Nw` checks for overflow and underflow.

```

21430 \cs_new:Npn _fp_sin_series_o:NNwww #1#2#3 . #4;
21431 {
21432 _fp_fixed_mul:wwn #4; #4;
21433 {
21434 \exp_after:wN _fp_sin_series_aux_o:NNwww
21435 \exp_after:wN #1
21436 \int_value:w
21437 \if_int_odd:w _fp_int_eval:w (#3 + 2) / 4 _fp_int_eval_end:
21438 #2
21439 \else:
21440 \if_meaning:w #2 0 2 \else: 0 \fi:
21441 \fi:
21442 {#3}
21443 }
21444 }
21445 \cs_new:Npn _fp_sin_series_aux_o:NNwww #1#2#3 #4; #5,#6;
21446 {
21447 \if_int_odd:w _fp_int_eval:w #3 / 2 _fp_int_eval_end:
21448 \exp_after:wN \use_i:nn
21449 \else:
21450 \exp_after:wN \use_ii:nn
21451 \fi:
21452 { % 1/18!
21453 _fp_fixed_mul_sub_back:wwn {0000}{0000}{0000}{0001}{5619}{2070};
21454 #4,{0000}{0000}{0000}{0477}{9477}{3324};
21455 _fp_fixed_mul_sub_back:wwn #4;{0000}{0000}{0011}{4707}{4559}{7730};
21456 _fp_fixed_mul_sub_back:wwn #4;{0000}{0000}{2087}{6756}{9878}{6810};
21457 _fp_fixed_mul_sub_back:wwn #4;{0000}{0027}{5573}{1922}{3985}{8907};
21458 _fp_fixed_mul_sub_back:wwn #4;{0000}{2480}{1587}{3015}{8730}{1587};
21459 _fp_fixed_mul_sub_back:wwn #4;{0013}{8888}{8888}{8888}{8889};
21460 _fp_fixed_mul_sub_back:wwn #4;{0416}{6666}{6666}{6666}{6667};
21461 _fp_fixed_mul_sub_back:wwn #4;{5000}{0000}{0000}{0000}{0000};
21462 _fp_fixed_mul_sub_back:wwn#4;{10000}{0000}{0000}{0000}{0000};
21463 { _fp_continue:wn 0, }
21464 }
21465 { % 1/17!
21466 _fp_fixed_mul_sub_back:wwn {0000}{0000}{0000}{0028}{1145}{7254};
21467 #4,{0000}{0000}{0000}{7647}{1637}{3182};
21468 _fp_fixed_mul_sub_back:wwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
21469 _fp_fixed_mul_sub_back:wwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
21470 _fp_fixed_mul_sub_back:wwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
21471 _fp_fixed_mul_sub_back:wwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
21472 _fp_fixed_mul_sub_back:wwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
21473 _fp_fixed_mul_sub_back:wwn #4;{1666}{6666}{6666}{6666}{6667};
21474 _fp_fixed_mul_sub_back:wwn#4;{10000}{0000}{0000}{0000}{0000};
21475 { _fp_ep_mul:wwwn 0, } #5,#6;
21476 }
21477 {
21478 \exp_after:wN _fp_sanitize:Nw
21479 \exp_after:wN #2
21480 \int_value:w _fp_int_eval:w #1
21481 }
```

```

21482 #2
21483 }

```

(End definition for `\_fp_sin_series_o:NNwww` and `\_fp_sin_series_aux_o:NNnwww`.)

```

\fp_tan_series_o:NNwww
\fp_tan_series_aux_o:Nnwww

```

Contrarily to `\_fp_sin_series_o:NNwww` which received a conversion auxiliary as #1, here, #1 is 0 for tangent and 2 for cotangent. Consider first the case of the tangent. The octant #3 starts at 1, which means that it is 1 or 2 for  $|x| \in [0, \pi/2]$ , it is 3 or 4 for  $|x| \in [\pi/2, \pi]$ , and so on: the intervals on which  $\tan|x| \geq 0$  coincide with those for which  $\lfloor(\#3 + 1)/2\rfloor$  is odd. We also have to take into account the original sign of  $x$  to get the sign of the final result; it is straightforward to check that the first `\int_value:w` expansion produces 0 for a positive final result, and 2 otherwise. A similar story holds for  $\cot(x)$ .

The auxiliary receives the sign, the octant, the square of the (reduced) input, and the (reduced) input (an extended-precision number) as arguments. It then computes the numerator and denominator of

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2a_5)))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2b_5))))}.$$

The ratio is computed by `\_fp_ep_div:wwwn`, then converted to a floating point number. For octants #3 (really, quadrants) next to a pole of the functions, the fixed point numerator and denominator are exchanged before computing the ratio. Note that this `\if_int_odd:w` test relies on the fact that the octant is at least 1.

```

21484 \cs_new:Npn \fp_tan_series_o:NNwww #1#2#3. #4;
21485 {
21486 \fp_fixed_mul:wwn #4; #4;
21487 {
21488 \exp_after:wN \fp_tan_series_aux_o:Nnwww
21489 \int_value:w
21490 \if_int_odd:w \fp_int_eval:w #3 / 2 \fp_int_eval_end:
21491 \exp_after:wN \reverse_if:N
21492 \fi:
21493 \if_meaning:w #1#2 2 \else: 0 \fi:
21494 {#3}
21495 }
21496 }
21497 \cs_new:Npn \fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
21498 {
21499 \fp_fixed_mul_sub_back:wwn {0000}{0000}{1527}{3493}{0856}{7059};
21500 #3; {0000}{0159}{6080}{0274}{5257}{6472};
21501 \fp_fixed_mul_sub_back:wwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
21502 \fp_fixed_mul_sub_back:wwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
21503 \fp_fixed_mul_sub_back:wwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
21504 \fp_fixed_mul_sub_back:wwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
21505 { \fp_ep_mul:wwnn 0, } #4,#5;
21506 {
21507 \fp_fixed_mul_sub_back:wwn {0000}{0007}{0258}{0681}{9408}{4706};
21508 #3; {0000}{2343}{7175}{1399}{6151}{7670};
21509 \fp_fixed_mul_sub_back:wwn #3; {0019}{2638}{4588}{9232}{8861}{3691};
21510 \fp_fixed_mul_sub_back:wwn #3; {0536}{6357}{0691}{4344}{6852}{4252};
21511 \fp_fixed_mul_sub_back:wwn #3; {5263}{1578}{9473}{6842}{1052}{6315};
21512 \fp_fixed_mul_sub_back:wwn #3; {10000}{0000}{0000}{0000}{0000}{0000};

```

```

21513 {
21514 \reverse_if:N \if_int_odd:w
21515 __fp_int_eval:w (#2 - 1) / 2 __fp_int_eval_end:
21516 \exp_after:wN __fp_reverse_args:Nww
21517 \fi:
21518 __fp_ep_div:wwwn 0,
21519 }
21520 }
21521 {
21522 \exp_after:wN __fp_sanitize:Nw
21523 \exp_after:wN #1
21524 \int_value:w __fp_int_eval:w __fp_ep_to_float_o:wwN
21525 }
21526 #1
21527 }

```

(End definition for `\__fp_tan_series_o:NNwwww` and `\__fp_tan_series_aux_o:Nnwww`.)

### 34.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arcosecant, and arcsecant) are based on a function often denoted `atan2`. This function is accessed directly by feeding two arguments to arctangent, and is defined by  $\text{atan}(y, x) = \text{atan}(y/x)$  for generic  $y$  and  $x$ . Its advantages over the conventional arctangent is that it takes values in  $[-\pi, \pi]$  rather than  $[-\pi/2, \pi/2]$ , and that it is better behaved in boundary cases. Other inverse trigonometric functions are expressed in terms of `atan` as

$$\text{acos } x = \text{atan}(\sqrt{1 - x^2}, x) \quad (5)$$

$$\text{asin } x = \text{atan}(x, \sqrt{1 - x^2}) \quad (6)$$

$$\text{asec } x = \text{atan}(\sqrt{x^2 - 1}, 1) \quad (7)$$

$$\text{acsc } x = \text{atan}(1, \sqrt{x^2 - 1}) \quad (8)$$

$$\text{atan } x = \text{atan}(x, 1) \quad (9)$$

$$\text{acot } x = \text{atan}(1, x). \quad (10)$$

Rather than introducing a new function, `atan2`, the arctangent function `atan` is overloaded: it can take one or two arguments. In the comments below, following many texts, we call the first argument  $y$  and the second  $x$ , because  $\text{atan}(y, x) = \text{atan}(y/x)$  is the angular coordinate of the point  $(x, y)$ .

As for direct trigonometric functions, the first step in computing  $\text{atan}(y, x)$  is argument reduction. The sign of  $y$  gives that of the result. We distinguish eight regions where the point  $(x, |y|)$  can lie, of angular size roughly  $\pi/8$ , characterized by their “octant”, between 0 and 7 included. In each region, we compute an arctangent as a Taylor series, then shift this arctangent by the appropriate multiple of  $\pi/4$  and sign to get the result. Here is a list of octants, and how we compute the arctangent (we assume  $y > 0$ : otherwise replace  $y$  by  $-y$  below):

0  $0 < |y| < 0.41421x$ , then  $\text{atan} \frac{|y|}{x}$  is given by a nicely convergent Taylor series;

1  $0 < 0.41421x < |y| < x$ , then  $\text{atan} \frac{|y|}{x} = \frac{\pi}{4} - \text{atan} \frac{x-|y|}{x+|y|}$ ;

- 2  $0 < 0.41421|y| < x < |y|$ , then  $\tan \frac{|y|}{x} = \frac{\pi}{4} + \tan \frac{-x+|y|}{x+|y|}$ ;
- 3  $0 < x < 0.41421|y|$ , then  $\tan \frac{|y|}{x} = \frac{\pi}{2} - \tan \frac{x}{|y|}$ ;
- 4  $0 < -x < 0.41421|y|$ , then  $\tan \frac{|y|}{x} = \frac{\pi}{2} + \tan \frac{-x}{|y|}$ ;
- 5  $0 < 0.41421|y| < -x < |y|$ , then  $\tan \frac{|y|}{x} = \frac{3\pi}{4} - \tan \frac{x+|y|}{-x+|y|}$ ;
- 6  $0 < -0.41421x < |y| < -x$ , then  $\tan \frac{|y|}{x} = \frac{3\pi}{4} + \tan \frac{-x-|y|}{-x+|y|}$ ;
- 7  $0 < |y| < -0.41421x$ , then  $\tan \frac{|y|}{x} = \pi - \tan \frac{|y|}{-x}$ .

In the following, we denote by  $z$  the ratio among  $|\frac{y}{x}|$ ,  $|\frac{x}{y}|$ ,  $|\frac{x+y}{x-y}|$ ,  $|\frac{x-y}{x+y}|$  which appears in the right-hand side above.

### 34.2.1 Arctangent and arccotangent

```
__fp_atan_o:Nw
__fp_acot_o:Nw
__fp_atan_default:w
```

The parsing step manipulates `atan` and `acot` like `min` and `max`, reading in an array of operands, but also leaves `\use_i:nn` or `\use_ii:nn` depending on whether the result should be given in radians or in degrees. The helper `\_\_fp_parse_function_one\_two:nnw` checks that the operand is one or two floating point numbers (not tuples) and leaves its second argument or its tail accordingly (its first argument is used for error messages). More precisely if we are given a single floating point number `\_\_fp_atan_default:w` places `\c_one_fp` (expanded) after it; otherwise `\_\_fp_atan_default:w` is omitted by `\_\_fp_parse_function_one\_two:nnw`.

```
21528 \cs_new:Npn __fp_atan_o:Nw #1
21529 {
21530 __fp_parse_function_one_two:nnw
21531 { #1 { atan } { atand } }
21532 { __fp_atan_default:w __fp_atanii_o:Nww #1 }
21533 }
21534 \cs_new:Npn __fp_acot_o:Nw #1
21535 {
21536 __fp_parse_function_one_two:nnw
21537 { #1 { acot } { acotd } }
21538 { __fp_atan_default:w __fp_acotii_o:Nww #1 }
21539 }
21540 \cs_new:Npx __fp_atan_default:w #1#2#3 @ { #1 #2 #3 \c_one_fp @ }
```

(End definition for `\_\_fp_atan_o:Nw`, `\_\_fp_acot_o:Nw`, and `\_\_fp_atan_default:w`.)

```
__fp_atanii_o:Nww
__fp_acotii_o:Nww
```

If either operand is `nan`, we return it. If both are normal, we call `\_\_fp_atan_normal_o:NNnwNnw`. If both are zero or both infinity, we call `\_\_fp_atan_inf_o:NNNw` with argument 2, leading to a result among  $\{\pm\pi/4, \pm 3\pi/4\}$  (in degrees,  $\{\pm 45, \pm 135\}$ ). Otherwise, one is much bigger than the other, and we call `\_\_fp_atan_inf_o:NNNw` with either an argument of 4, leading to the values  $\pm\pi/2$  (in degrees,  $\pm 90$ ), or 0, leading to  $\{\pm 0, \pm\pi\}$  (in degrees,  $\{\pm 0, \pm 180\}$ ). Since  $\text{acot}(x, y) = \text{atan}(y, x)$ , `\_\_fp_acotii_o:ww` simply reverses its two arguments.

```
21541 \cs_new:Npn __fp_atanii_o:Nww
21542 #1 \s__fp __fp_chk:w #2#3#4; \s__fp __fp_chk:w #5 #6 @
21543 {
21544 \if_meaning:w 3 #2 __fp_case_return_i_o:ww \fi:
```

```

21545 \if_meaning:w 3 #5 __fp_case_return_ii_o:ww \fi:
21546 \if_case:w
21547 \if_meaning:w #2 #5
21548 \if_meaning:w 1 #2 10 \else: 0 \fi:
21549 \else:
21550 \if_int_compare:w #2 > #5 \exp_stop_f: 1 \else: 2 \fi:
21551 \fi:
21552 \exp_stop_f:
21553 __fp_case_return:nw { __fp_atan_inf_o:NNNw #1 #3 2 }
21554 \or: __fp_case_return:nw { __fp_atan_inf_o:NNNw #1 #3 4 }
21555 \or: __fp_case_return:nw { __fp_atan_inf_o:NNNw #1 #3 0 }
21556 \fi:
21557 __fp_atan_normal_o:NNnwNnw #1
21558 \s__fp __fp_chk:w #2#3#4;
21559 \s__fp __fp_chk:w #5 #6
21560 }
21561 \cs_new:Npn __fp_acotii_o:Nww #1#2; #3;
21562 { __fp_atanii_o:Nww #1#3; #2; }

```

(End definition for `\__fp_atanii_o:Nww` and `\__fp_acotii_o:Nww`.)

`\__fp_atan_inf_o:NNNw`

This auxiliary is called whenever one number is  $\pm 0$  or  $\pm\infty$  (and neither is NaN). Then the result only depends on the signs, and its value is a multiple of  $\pi/4$ . We use the same auxiliary as for normal numbers, `\__fp_atan_combine_o:NwwwwN`, with arguments the final sign #2; the octant #3; atan  $z/z = 1$  as a fixed point number;  $z = 0$  as a fixed point number; and  $z = 0$  as an extended-precision number. Given the values we provide, atan  $z$  is computed to be 0, and the result is  $[\#3/2] \cdot \pi/4$  if the sign #5 of  $x$  is positive, and  $[(7 - \#3)/2] \cdot \pi/4$  for negative  $x$ , where the divisions are rounded up.

```

21563 \cs_new:Npn __fp_atan_inf_o:NNNw #1#2#3 \s__fp __fp_chk:w #4#5#6;
21564 {
21565 \exp_after:wN __fp_atan_combine_o:NwwwwN
21566 \exp_after:wN #2
21567 \int_value:w __fp_int_eval:w
21568 \if_meaning:w 2 #5 7 - \fi: #3 \exp_after:wN ;
21569 \c__fp_one_fixed_tl
21570 {0000}{0000}{0000}{0000}{0000}{0000};
21571 0,{0000}{0000}{0000}{0000}{0000}; #1
21572 }

```

(End definition for `\__fp_atan_inf_o:NNNw`.)

`\__fp_atan_normal_o:NNnwNnw`

Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like  $\text{atan}(x, \sqrt{1 - x^2})$  without intermediate rounding errors.

```

21573 \cs_new_protected:Npn __fp_atan_normal_o:NNnwNnw
21574 #1 \s__fp __fp_chk:w 1#2#3#4; \s__fp __fp_chk:w 1#5#6#7;
21575 {
21576 __fp_atan_test_o:NwwNwwN
21577 #2 #3, #4{0000}{0000};
21578 #5 #6, #7{0000}{0000}; #1
21579 }

```

(End definition for `\_fp_atan_normal_o:NnwnNnw`.)

`\_fp_atan_test_o:NwwNwwN`

This receives: the sign #1 of  $y$ , its exponent #2, its 24 digits #3 in groups of 4, and similarly for  $x$ . We prepare to call `\_fp_atan_combine_o:NwwwwwN` which expects the sign #1, the octant, the ratio  $(\tan z)/z = 1 - \dots$ , and the value of  $z$ , both as a fixed point number and as an extended-precision floating point number with a mantissa in  $[0.01, 1)$ . For now, we place #1 as a first argument, and start an integer expression for the octant. The sign of  $x$  does not affect  $z$ , so we simply leave a contribution to the octant:  $\langle \text{octant} \rangle \rightarrow 7 - \langle \text{octant} \rangle$  for negative  $x$ . Then we order  $|y|$  and  $|x|$  in a non-decreasing order: if  $|y| > |x|$ , insert 3– in the expression for the octant, and swap the two numbers. The finer test with 0.41421 is done by `\_fp_atan_div:wnwwnw` after the operands have been ordered.

```

21580 \cs_new:Npn _fp_atan_test_o:NwwNwwN #1#2,#3; #4#5,#6;
21581 {
21582 \exp_after:wN _fp_atan_combine_o:NwwwwwN
21583 \exp_after:wN #1
21584 \int_value:w _fp_int_eval:w
21585 \if_meaning:w 2 #4
21586 7 - _fp_int_eval:w
21587 \fi:
21588 \if_int_compare:w
21589 _fp_ep_compare:www #2,#3; #5,#6; > 0 \exp_stop_f:
21590 3 -
21591 \exp_after:wN _fp_reverse_args:Nww
21592 \fi:
21593 _fp_atan_div:wnwwnw #2,#3; #5,#6;
21594 }

```

(End definition for `\_fp_atan_test_o:NwwNwwN`.)

`\_fp_atan_div:wnwwnw`  
`\_fp_atan_near:www`  
`\_fp_atan_near_aux:www`

This receives two positive numbers  $a$  and  $b$  (equal to  $|x|$  and  $|y|$  in some order), each as an exponent and 6 blocks of 4 digits, such that  $0 < a < b$ . If  $0.41421b < a$ , the two numbers are “near”, hence the point  $(y, x)$  that we started with is closer to the diagonals  $\{|y| = |x|\}$  than to the axes  $\{xy = 0\}$ . In that case, the octant is 1 (possibly combined with the 7– and 3– inserted earlier) and we wish to compute  $\tan \frac{b-a}{a+b}$ . Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute  $\tan \frac{a}{b}$ . In any case, call `\_fp_atan_auxi:ww` followed by  $z$ , as a comma-delimited exponent and a fixed point number.

```

21595 \cs_new:Npn _fp_atan_div:wnwwnw #1,#2#3; #4,#5#6;
21596 {
21597 \if_int_compare:w
21598 _fp_int_eval:w 41421 * #5 < #2 000
21599 \if_case:w _fp_int_eval:w #4 - #1 _fp_int_eval_end:
21600 00 \or: 0 \fi:
21601 \exp_stop_f:
21602 \exp_after:wN _fp_atan_near:www
21603 \fi:
21604 0
21605 _fp_ep_div:www #1,{#2}#3; #4,{#5}#6;
21606 _fp_atan_auxi:ww
21607 }
21608 \cs_new:Npn _fp_atan_near:www
21609 0 _fp_ep_div:www #1,#2; #3,

```

```

21610 {
21611 1
21612 __fp_ep_to_fixed:wwn #1 - #3, #2;
21613 __fp_atan_near_aux:wwn
21614 }
21615 \cs_new:Npn __fp_atan_near_aux:wwn #1; #2;
21616 {
21617 __fp_fixed_add:wwn #1; #2;
21618 { __fp_fixed_sub:wwn #2; #1; { __fp_ep_div:wwwn 0, } 0, }
21619 }

```

(End definition for `\_\_fp\_atan\_div:wnwwnw`, `\_\_fp\_atan\_near:wwnn`, and `\_\_fp\_atan\_near\_aux:wwn`.)

`\_\_fp\_atan\_auxi:ww`  
`\_\_fp\_atan\_auxii:w`

Convert  $z$  from a representation as an exponent and a fixed point number in  $[0.01, 1)$  to a fixed point number only, then set up the call to `\_\_fp\_atan\_Taylor\_loop:www`, followed by the fixed point representation of  $z$  and the old representation.

```

21620 \cs_new:Npn __fp_atan_auxi:ww #1,#2;
21621 { __fp_ep_to_fixed:wwn #1,#2; __fp_atan_auxii:w #1,#2; }
21622 \cs_new:Npn __fp_atan_auxii:w #1;
21623 {
21624 __fp_fixed_mul:wwn #1; #1;
21625 {
21626 __fp_atan_Taylor_loop:www 39 ;
21627 {0000}{0000}{0000}{0000}{0000}{0000} ;
21628 }
21629 ! #1;
21630 }

```

(End definition for `\_\_fp\_atan\_auxi:ww` and `\_\_fp\_atan\_auxii:w`.)

`\_\_fp\_atan\_Taylor\_loop:www`  
`\_\_fp\_atan\_Taylor\_break:w`

We compute the series of  $(\tan z)/z$ . A typical intermediate stage has  $\#1 = 2k - 1$ ,  $\#2 = \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\dots - z^2 \frac{1}{39}))$ , and  $\#3 = z^2$ . To go to the next step  $k \rightarrow k - 1$ , we compute  $\frac{1}{2k-1}$ , then subtract from it  $z^2$  times  $\#2$ . The loop stops when  $k = 0$ : then  $\#2$  is  $(\tan z)/z$ , and there is a need to clean up all the unnecessary data, end the integer expression computing the octant with a semicolon, and leave the result  $\#2$  afterwards.

```

21631 \cs_new:Npn __fp_atan_Taylor_loop:www #1; #2; #3;
21632 {
21633 \if_int_compare:w #1 = -1 \exp_stop_f:
21634 __fp_atan_Taylor_break:w
21635 \fi:
21636 \exp_after:wN __fp_fixed_div_int:wwN \c_fp_one_fixed_tl #1;
21637 __fp_rrrot:www __fp_fixed_mul_sub_back:wwn #2; #3;
21638 {
21639 \exp_after:wN __fp_atan_Taylor_loop:www
21640 \int_value:w __fp_int_eval:w #1 - 2 ;
21641 }
21642 #3;
21643 }
21644 \cs_new:Npn __fp_atan_Taylor_break:w
21645 \fi: #1 __fp_fixed_mul_sub_back:wwn #2; #3 !
21646 { \fi: ; #2 ; }

```

(End definition for `\_\_fp\_atan\_Taylor\_loop:www` and `\_\_fp\_atan\_Taylor\_break:w`.)

```
__fp_atan_combine_o:NwwwwN
__fp_atan_combine_aux:ww
```

This receives a  $\langle sign \rangle$ , an  $\langle octant \rangle$ , a fixed point value of  $(\text{atan } z)/z$ , a fixed point number  $z$ , and another representation of  $z$ , as an  $\langle exponent \rangle$  and the fixed point number  $10^{-\langle exponent \rangle} z$ , followed by either  $\text{\use_i:nn}$  (when working in radians) or  $\text{\use_ii:nn}$  (when working in degrees). The function computes the floating point result

$$\langle sign \rangle \left( \left\lceil \frac{\langle octant \rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant \rangle} \frac{\text{atan } z}{z} \cdot z \right), \quad (11)$$

multiplied by  $180/\pi$  if working in degrees, and using in any case the most appropriate representation of  $z$ . The floating point result is passed to  $\text{\_fp\_sanitize:Nw}$ , which checks for overflow or underflow. If the octant is 0, leave the exponent #5 for  $\text{\_fp\_sanitize:Nw}$ , and multiply #3 =  $\frac{\text{atan } z}{z}$  with #6, the adjusted  $z$ . Otherwise, multiply #3 =  $\frac{\text{atan } z}{z}$  with #4 =  $z$ , then compute the appropriate multiple of  $\frac{\pi}{4}$  and add or subtract the product #3 · #4. In both cases, convert to a floating point with  $\text{\_fp\_fixed\_to\_float\_o:wn}$ .

```
21647 \cs_new:Npn __fp_atan_combine_o:NwwwwN #1 #2; #3; #4; #5; #6; #7
21648 {
21649 \exp_after:wN __fp_sanitize:Nw
21650 \exp_after:wN #1
21651 \int_value:w __fp_int_eval:w
21652 \if_meaning:w 0 #2
21653 \exp_after:wN \use_i:nn
21654 \else:
21655 \exp_after:wN \use_ii:nn
21656 \fi:
21657 { #5 __fp_fixed_mul:wwn #3; #6; }
21658 {
21659 __fp_fixed_mul:wwn #3; #4;
21660 {
21661 \exp_after:wN __fp_atan_combine_aux:ww
21662 \int_value:w __fp_int_eval:w #2 / 2 ; #2;
21663 }
21664 }
21665 { #7 __fp_fixed_to_float_o:wn __fp_fixed_to_float_rad_o:wn }
21666 #1
21667 }
21668 \cs_new:Npn __fp_atan_combine_aux:ww #1; #2;
21669 {
21670 __fp_fixed_mul_short:wwn
21671 {7853}{9816}{3397}{4483}{0961}{5661};
21672 {#1}{0000}{0000};
21673 {
21674 \if_int_odd:w #2 \exp_stop_f:
21675 \exp_after:wN __fp_fixed_sub:wwn
21676 \else:
21677 \exp_after:wN __fp_fixed_add:wwn
21678 \fi:
21679 }
21680 }
```

(End definition for  $\text{\_fp\_atan\_combine\_o:NwwwwN}$  and  $\text{\_fp\_atan\_combine\_aux:ww}$ .)

### 34.2.2 Arcsine and arccosine

\\_fp\_asin\_o:w

Again, the first argument provided by l3fp-parse is \use\_i:nn if we are to work in radians and \use\_ii:nn for degrees. Then comes a floating point number. The arcsine of  $\pm 0$  or NaN is the same floating point number. The arcsine of  $\pm\infty$  raises an invalid operation exception. Otherwise, call an auxiliary common with \\_fp\_acos\_o:w, feeding it information about what function is being performed (for “invalid operation” exceptions).

```

21681 \cs_new:Npn _fp_asin_o:w #1 \s_fp _fp_chk:w #2#3; @
21682 {
21683 \if_case:w #2 \exp_stop_f:
21684 _fp_case_return_same_o:w
21685 \or:
21686 _fp_case_use:nw
21687 { _fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
21688 \or:
21689 _fp_case_use:nw
21690 { _fp_invalid_operation_o:fw { #1 { asin } { asind } } }
21691 \else:
21692 _fp_case_return_same_o:w
21693 \fi:
21694 \s_fp _fp_chk:w #2 #3;
21695 }
```

(End definition for \\_fp\_asin\_o:w.)

\\_fp\_acos\_o:w

The arccosine of  $\pm 0$  is  $\pi/2$  (in degrees, 90). The arccosine of  $\pm\infty$  raises an invalid operation exception. The arccosine of NaN is itself. Otherwise, call an auxiliary common with \\_fp\_sin\_o:w, informing it that it was called by acos or acosd, and preparing to swap some arguments down the line.

```

21696 \cs_new:Npn _fp_acos_o:w #1 \s_fp _fp_chk:w #2#3; @
21697 {
21698 \if_case:w #2 \exp_stop_f:
21699 _fp_case_use:nw { _fp_atan_inf_o>NNNw #1 0 4 }
21700 \or:
21701 _fp_case_use:nw
21702 {
21703 _fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
21704 _fp_reverse_args:Nww
21705 }
21706 \or:
21707 _fp_case_use:nw
21708 { _fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
21709 \else:
21710 _fp_case_return_same_o:w
21711 \fi:
21712 \s_fp _fp_chk:w #2 #3;
21713 }
```

(End definition for \\_fp\_acos\_o:w.)

\\_fp\_asin\_normal\_o:NfwNnnnnw

If the exponent #5 is at most 0, the operand lies within  $(-1, 1)$  and the operation is permitted: call \\_fp\_asin\_auxi\_o:NnNww with the appropriate arguments. If the number is exactly  $\pm 1$  (the test works because we know that #5  $\geq 1$ , #6#7  $\geq 10000000$ , #8#9  $\geq 0$ ,

with equality only for  $\pm 1$ ), we also call  $\text{\_fp\_asin\_auxi\_o:NnNww}$ . Otherwise,  $\text{\_fp\_use\_i:ww}$  gets rid of the `asin` auxiliary, and raises instead an invalid operation, because the operand is outside the domain of arcsine or arccosine.

```

21714 \cs_new:Npn _fp_asin_normal_o:NfwNnnnnw
21715 #1#2#3 \s_fp _fp_chk:w 1#4#5#6#7#8#9;
21716 {
21717 \if_int_compare:w #5 < 1 \exp_stop_f:
21718 \exp_after:wN _fp_use_none_until_s:w
21719 \fi:
21720 \if_int_compare:w _fp_int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
21721 \exp_after:wN _fp_use_none_until_s:w
21722 \fi:
21723 _fp_use_i:ww
21724 _fp_invalid_operation_o:fw {#2}
21725 \s_fp _fp_chk:w 1#4{#5}{#6}{#7}{#8}{#9};
21726 _fp_asin_auxi_o:NnNww
21727 #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
21728 }
```

(End definition for  $\text{\_fp\_asin\_normal\_o:NfwNnnnnw}$ .)

$\text{\_fp\_asin\_auxi\_o:NnNww}$   
 $\text{\_fp\_asin\_isqrt:wn}$

We compute  $x/\sqrt{1-x^2}$ . This function is used by `asin` and `acos`, but also by `acsc` and `asec` after inverting the operand, thus it must manipulate extended-precision numbers. First evaluate  $1-x^2$  as  $(1+x)(1-x)$ : this behaves better near  $x=1$ . We do the addition/subtraction with fixed point numbers (they are not implemented for extended-precision floats), but go back to extended-precision floats to multiply and compute the inverse square root  $1/\sqrt{1-x^2}$ . Finally, multiply by the (positive) extended-precision float  $|x|$ , and feed the (signed) result, and the number  $+1$ , as arguments to the arctangent function. When computing the arccosine, the arguments  $x/\sqrt{1-x^2}$  and  $+1$  are swapped by `#2` ( $\text{\_fp\_reverse\_args:Nww}$  in that case) before  $\text{\_fp\_atan\_test\_o:NwwNwwN}$  is evaluated. Note that the arctangent function requires normalized arguments, hence the need for `ep_to_ep` and `continue` after `ep_mul`.

```

21729 \cs_new:Npn _fp_asin_auxi_o:NnNww #1#2#3#4,#5;
21730 {
21731 _fp_ep_to_fixed:wwn #4,#5;
21732 _fp_asin_isqrt:wn
21733 _fp_ep_mul:wwwn #4,#5;
21734 _fp_ep_to_ep:wwN
21735 _fp_fixed_continue:wn
21736 { #2 _fp_atan_test_o:NwwNwwN #3 }
21737 0 1,{1000}{0000}{0000}{0000}{0000}; #1
21738 }
21739 \cs_new:Npn _fp_asin_isqrt:wn #1;
21740 {
21741 \exp_after:wN _fp_fixed_sub:wwn \c_fp_one_fixed_tl #1;
21742 {
21743 _fp_fixed_add_one:wN #1;
21744 _fp_fixed_continue:wn { _fp_ep_mul:wwwn 0, } 0,
21745 }
21746 _fp_ep_isqrt:wwn
21747 }
```

(End definition for  $\text{\_fp\_asin\_auxi\_o:NnNww}$  and  $\text{\_fp\_asin\_isqrt:wn}$ .)

### 34.2.3 Arc cosecant and arcsecant

\\_fp\_acsc\_o:w

Cases are mostly labelled by #2, except when #2 is 2: then we use #3#2, which is 02 = 2 when the number is  $+\infty$  and 22 when the number is  $-\infty$ . The arccosecant of  $\pm 0$  raises an invalid operation exception. The arccosecant of  $\pm\infty$  is  $\pm 0$  with the same sign. The arcosecant of NaN is itself. Otherwise, \\_fp\_acsc\_normal\_o:NfwNnw does some more tests, keeping the function name (acsc or acscd) as an argument for invalid operation exceptions.

```

21748 \cs_new:Npn _fp_acsc_o:w #1 \s_fp _fp_chk:w #2#3#4; @
21749 {
21750 \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
21751 _fp_case_use:nw
21752 { _fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
21753 \or: _fp_case_use:nw
21754 { _fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
21755 \or: _fp_case_return_o:Nw \c_zero_fp
21756 \or: _fp_case_return_same_o:w
21757 \else: _fp_case_return_o:Nw \c_minus_zero_fp
21758 \fi:
21759 \s_fp _fp_chk:w #2 #3 #4;
21760 }

```

(End definition for \\_fp\_acsc\_o:w.)

\\_fp\_asec\_o:w

The arcsecant of  $\pm 0$  raises an invalid operation exception. The arcsecant of  $\pm\infty$  is  $\pi/2$  (in degrees, 90). The arcosecant of NaN is itself. Otherwise, do some more tests, keeping the function name asec (or asecd) as an argument for invalid operation exceptions, and a \\_fp\_reverse\_args:Nww following precisely that appearing in \\_fp\_acos\_o:w.

```

21761 \cs_new:Npn _fp_asec_o:w #1 \s_fp _fp_chk:w #2#3; @
21762 {
21763 \if_case:w #2 \exp_stop_f:
21764 _fp_case_use:nw
21765 { _fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
21766 \or:
21767 _fp_case_use:nw
21768 {
21769 _fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
21770 _fp_reverse_args:Nww
21771 }
21772 \or: _fp_case_use:nw { _fp_atan_inf_o>NNNw #1 0 4 }
21773 \else: _fp_case_return_same_o:w
21774 \fi:
21775 \s_fp _fp_chk:w #2 #3;
21776 }

```

(End definition for \\_fp\_asec\_o:w.)

\\_fp\_acsc\_normal\_o:NfwNnw

If the exponent is non-positive, the operand is less than 1 in absolute value, which is always an invalid operation: complain. Otherwise, compute the inverse of the operand, and feed it to \\_fp\_asin\_auxi\_o:NnNww (with all the appropriate arguments). This computes what we want thanks to  $\text{acsc}(x) = \text{asin}(1/x)$  and  $\text{asec}(x) = \text{acos}(1/x)$ .

```

21777 \cs_new:Npn _fp_acsc_normal_o:NfwNnw #1#2#3 \s_fp _fp_chk:w 1#4#5#6;
21778 {
21779 \int_compare:nNnTF {#5} < 1

```

```

21780 {
21781 __fp_invalid_operation_o:fw {#2}
21782 \s__fp __fp_chk:w 1#4{#5}#6;
21783 }
21784 {
21785 __fp_ep_div:wwwn
21786 1,{1000}{0000}{0000}{0000}{0000}{0000};
21787 #5,#6{0000}{0000};
21788 { __fp_asin_auxi_o:NnNww #1 {#3} #4 }
21789 }
21790 }

(End definition for __fp_acsc_normal_o:NfwNnw.)
```

21791 </package>

## 35 l3fp-convert implementation

```

21792 /*package)
21793 @@=fp)
```

### 35.1 Dealing with tuples

The first argument is for instance \\_\_fp\_to\_tl\_dispatch:w, which converts any floating point object to the appropriate representation. We loop through all items, putting ,~ between all of them and making sure to remove the leading ,~.

```

21794 \cs_new:Npn __fp_tuple_convert:Nw #1 \s__fp_tuple __fp_tuple_chk:w #2 ;
21795 {
21796 \int_case:nnF { __fp_array_count:n {#2} }
21797 {
21798 { 0 } { () }
21799 { 1 } { __fp_tuple_convert_end:w @ { #1 #2 , } }
21800 }
21801 {
21802 __fp_tuple_convert_loop:nNw { } #1
21803 #2 { ? __fp_tuple_convert_end:w } ;
21804 @ { \use_none:nn }
21805 }
21806 }
21807 \cs_new:Npn __fp_tuple_convert_loop:nNw #1#2#3#4; #5 @ #6
21808 {
21809 \use_none:n #3
21810 \exp_args:Nf __fp_tuple_convert_loop:nNw { #2 #3#4 ; } #2 #5
21811 @ { #6 , ~ #1 }
21812 }
21813 \cs_new:Npn __fp_tuple_convert_end:w #1 @ #2
21814 { \exp_after:wN (\exp:w \exp_end_continue_f:w #2) }
```

(End definition for \\_\_fp\_tuple\_convert:Nw, \\_\_fp\_tuple\_convert\_loop:nNw, and \\_\_fp\_tuple\_convert\_end:w.)

## 35.2 Trimming trailing zeros

`\_fp_trim_zeros:w`  
`\_fp_trim_zeros_loop:w`  
`\_fp_trim_zeros_dot:w`  
`\_fp_trim_zeros_end:w`

If #1 ends with a 0, the `loop` auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same `loop` auxiliary. Once the last trailing zero is reached, the second argument is the `dot` auxiliary, which removes a trailing dot if any. We then clean-up with the `end` auxiliary, keeping only the number.

```
21815 \cs_new:Npn _fp_trim_zeros:w #1 ;
21816 {
21817 _fp_trim_zeros_loop:w #1
21818 ; _fp_trim_zeros_loop:w 0; _fp_trim_zeros_dot:w .; \s__fp_stop
21819 }
21820 \cs_new:Npn _fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
21821 \cs_new:Npn _fp_trim_zeros_dot:w #1 .; { _fp_trim_zeros_end:w #1 ; }
21822 \cs_new:Npn _fp_trim_zeros_end:w #1 ; #2 \s__fp_stop { #1 }
```

(End definition for `\_fp_trim_zeros:w` and others.)

## 35.3 Scientific notation

`\fp_to_scientific:N`  
`\fp_to_scientific:c`  
`\fp_to_scientific:n`

The three public functions evaluate their argument, then pass it to `\_fp_to_scientific_dispatch:w`.

```
21823 \cs_new:Npn \fp_to_scientific:N #1
21824 { \exp_after:wN _fp_to_scientific_dispatch:w #1 }
21825 \cs_generate_variant:Nn \fp_to_scientific:N { c }
21826 \cs_new:Npn \fp_to_scientific:n
21827 {
21828 \exp_after:wN _fp_to_scientific_dispatch:w
21829 \exp:w \exp_end_continue_f:w _fp_parse:n
21830 }
```

(End definition for `\fp_to_scientific:N` and `\fp_to_scientific:n`. These functions are documented on page 203.)

We allow tuples.

```
21831 \cs_new:Npn _fp_to_scientific_dispatch:w #1
21832 {
21833 _fp_change_func_type:NNN
21834 #1 _fp_to_scientific:w _fp_to_scientific_recover:w
21835 #1
21836 }
21837 \cs_new:Npn _fp_to_scientific_recover:w #1 #2 ;
21838 {
21839 _fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
21840 nan
21841 }
21842 \cs_new:Npn _fp_tuple_to_scientific:w
21843 { _fp_tuple_convert:Nw _fp_to_scientific_dispatch:w }
```

(End definition for `\_fp_to_scientific_dispatch:w`, `\_fp_to_scientific_recover:w`, and `\_fp_tuple_to_scientific:w`.)

`\_fp_to_scientific:w`  
`\_fp_to_scientific_normal:wnnnnn`  
`\_fp_to_scientific_normal:wNw`

Expressing an internal floating point number in scientific notation is quite easy: no rounding, and the format is very well defined. First cater for the sign: negative numbers (#2 = 2) start with `-`; we then only need to care about positive numbers and `nan`. Then

filter the special cases:  $\pm 0$  are represented as 0; infinities are converted to a number slightly larger than the largest after an “invalid\_operation” exception; `nan` is represented as 0 after an “invalid\_operation” exception. In the normal case, decrement the exponent and unbrace the 4 brace groups, then in a second step grab the first digit (previously hidden in braces) to order the various parts correctly.

```

21844 \cs_new:Npn __fp_to_scientific:w \s__fp __fp_chk:w #1#2
21845 {
21846 \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
21847 \if_case:w #1 \exp_stop_f:
21848 __fp_case_return:nw { 0.000000000000000e0 }
21849 \or: \exp_after:wN __fp_to_scientific_normal:wnnnnn
21850 \or:
21851 __fp_case_use:nw
21852 {
21853 __fp_invalid_operation:nnw
21854 { \fp_to_scientific:N \c__fp_overflowing_fp }
21855 { fp_to_scientific }
21856 }
21857 \or:
21858 __fp_case_use:nw
21859 {
21860 __fp_invalid_operation:nnw
21861 { \fp_to_scientific:N \c_zero_fp }
21862 { fp_to_scientific }
21863 }
21864 \fi:
21865 \s__fp __fp_chk:w #1 #2
21866 }
21867 \cs_new:Npn __fp_to_scientific_normal:wnnnnn
21868 \s__fp __fp_chk:w 1 #1 #2 #3#4#5#6 ;
21869 {
21870 \exp_after:wN __fp_to_scientific_normal:wNw
21871 \exp_after:wN e
21872 \int_value:w __fp_int_eval:w #2 - 1
21873 ; #3 #4 #5 #6 ;
21874 }
21875 \cs_new:Npn __fp_to_scientific_normal:wNw #1 ; #2#3;
21876 { #2.#3 #1 }

(End definition for __fp_to_scientific:w, __fp_to_scientific_normal:wnnnnn, and __fp_to-
scientific_normal:wNw.)
```

### 35.4 Decimal representation

`\fp_to_decimal:N` All three public variants are based on the same `\__fp_to_decimal_dispatch:w` after evaluating their argument to an internal floating point.

`\fp_to_decimal:c`

```

21877 \cs_new:Npn \fp_to_decimal:N #1
21878 { \exp_after:wN __fp_to_decimal_dispatch:w #1 }
21879 \cs_generate_variant:Nn \fp_to_decimal:N { c }
21880 \cs_new:Npn \fp_to_decimal:n
21881 {
21882 \exp_after:wN __fp_to_decimal_dispatch:w
21883 \exp:w \exp_end_continue_f:w __fp_parse:n
21884 }
```

(End definition for `\fp_to_decimal:N` and `\fp_to_decimal:n`. These functions are documented on page 203.)

```
__fp_to_decimal_dispatch:w
__fp_to_decimal_recover:w
__fp_tuple_to_decimal:w
```

We allow tuples.

```
21885 \cs_new:Npn __fp_to_decimal_dispatch:w #1
21886 {
21887 __fp_change_func_type:NNN
21888 #1 __fp_to_decimal:w __fp_to_decimal_recover:w
21889 #1
21890 }
21891 \cs_new:Npn __fp_to_decimal_recover:w #1 #2 ;
21892 {
21893 __fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
21894 nan
21895 }
21896 \cs_new:Npn __fp_tuple_to_decimal:w
21897 { __fp_tuple_convert:Nw __fp_to_decimal_dispatch:w }
```

(End definition for `\__fp_to_decimal_dispatch:w`, `\__fp_to_decimal_recover:w`, and `\__fp_tuple_to_decimal:w`.)

```
__fp_to_decimal:w
\fp_to_decimal_normal:wnnnnn
\fp_to_decimal_large:Nnnn
\fp_to_decimal_huge:wnnnn
```

The structure is similar to `\__fp_to_scientific:w`. Insert `-` for negative numbers. Zero gives 0,  $\pm\infty$  and NaN yield an “invalid operation” exception; note that  $\pm\infty$  produces a very large output, which we don’t expand now since it most likely won’t be needed. Normal numbers with an exponent in the range [1, 15] have that number of digits before the decimal separator: “decimate” them, and remove leading zeros with `\int_value:w`, then trim trailing zeros and dot. Normal numbers with an exponent 16 or larger have no decimal separator, we only need to add trailing zeros. When the exponent is non-positive, the result should be `0.(zeros)(digits)`, trimmed.

```
21898 \cs_new:Npn __fp_to_decimal:w \s__fp __fp_chk:w #1#2
21899 {
21900 \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
21901 \if_case:w #1 \exp_stop_f:
21902 __fp_case_return:nw { 0 }
21903 \or: \exp_after:wN __fp_to_decimal_normal:wnnnnn
21904 \or:
21905 __fp_case_use:nw
21906 {
21907 __fp_invalid_operation:nnw
21908 { \fp_to_decimal:N \c_fp_overflowing_fp }
21909 { fp_to_decimal }
21910 }
21911 \or:
21912 __fp_case_use:nw
21913 {
21914 __fp_invalid_operation:nnw
21915 { 0 }
21916 { fp_to_decimal }
21917 }
21918 \fi:
21919 \s__fp __fp_chk:w #1 #2
21920 }
21921 \cs_new:Npn __fp_to_decimal_normal:wnnnnn
21922 \s__fp __fp_chk:w 1 #1 #2 #3#4#5#6 ;
```

```

21923 {
21924 \int_compare:nNnTF {#2} > 0
21925 {
21926 \int_compare:nNnTF {#2} < \c_fp_prec_int
21927 {
21928 __fp_decimate:nNnnn { \c_fp_prec_int - #2 }
21929 __fp_to_decimal_large:Nnnw
21930 }
21931 {
21932 \exp_after:wN \exp_after:wN
21933 \exp_after:wN __fp_to_decimal_huge:wnnnn
21934 \prg_replicate:nn { #2 - \c_fp_prec_int } { 0 } ;
21935 }
21936 {#3} {#4} {#5} {#6}
21937 }
21938 {
21939 \exp_after:wN __fp_trim_zeros:w
21940 \exp_after:wN 0
21941 \exp_after:wN .
21942 \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
21943 #3#4#5#6 ;
21944 }
21945 }
21946 \cs_new:Npn __fp_to_decimal_large:Nnnw #1#2#3#4;
21947 {
21948 \exp_after:wN __fp_trim_zeros:w \int_value:w
21949 \if_int_compare:w #2 > 0 \exp_stop_f:
21950 #2
21951 \fi:
21952 \exp_stop_f:
21953 #3.#4 ;
21954 }
21955 \cs_new:Npn __fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }

(End definition for __fp_to_decimal:w and others.)

```

### 35.5 Token list representation

**\fp\_to\_tl:N** These three public functions evaluate their argument, then pass it to \\_\_fp\_to\_tl\_dispatch:w.

```

\fp_to_tl:n 21956 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN __fp_to_tl_dispatch:w #1 }
21957 \cs_generate_variant:Nn \fp_to_tl:N { c }
21958 \cs_new:Npn \fp_to_tl:n
21959 {
21960 \exp_after:wN __fp_to_tl_dispatch:w
21961 \exp:w \exp_end_continue_f:w __fp_parse:n
21962 }

```

(End definition for \fp\_to\_tl:N and \fp\_to\_tl:n. These functions are documented on page 204.)

\\_\_fp\_to\_tl\_dispatch:w  
\\_\_fp\_to\_tl\_recover:w  
\\_\_fp\_tuple\_to\_tl:w

```

21963 \cs_new:Npn __fp_to_tl_dispatch:w #1
21964 { __fp_change_func_type:NNN #1 __fp_to_tl:w __fp_to_tl_recover:w #1 }
21965 \cs_new:Npn __fp_to_tl_recover:w #1 #2 ;

```

```

21966 {
21967 _fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
21968 nan
21969 }
21970 \cs_new:Npn _fp_tuple_to_tl:w
21971 { _fp_tuple_convert:Nw _fp_to_tl_dispatch:w }

```

(End definition for `\_fp_to_tl_dispatch:w`, `\_fp_to_tl_recover:w`, and `\_fp_tuple_to_tl:w`.)

```

\fp_to_tl:w
\fp_to_tl_normal:nnnnn
 \fp_to_tl_scientific:wnnnnn
\fp_to_tl_scientific:wNw

```

A structure similar to `\_fp_to_scientific_dispatch:w` and `\_fp_to_decimal_dispatch:w`, but without the “invalid operation” exception. First filter special cases. We express normal numbers in decimal notation if the exponent is in the range  $[-2, 16]$ , and otherwise use scientific notation.

```

21972 \cs_new:Npn _fp_to_tl:w \s_fp _fp_chk:w #1#2
21973 {
21974 \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
21975 \if_case:w #1 \exp_stop_f:
21976 _fp_case_return:nw { 0 }
21977 \or: \exp_after:wN _fp_to_tl_normal:nnnnn
21978 \or: _fp_case_return:nw { inf }
21979 \else: _fp_case_return:nw { nan }
21980 \fi:
21981 }
21982 \cs_new:Npn _fp_to_tl_normal:nnnnn #1
21983 {
21984 \int_compare:nTF
21985 { -2 <= #1 <= \c_fp_prec_int }
21986 { _fp_to_decimal_normal:wnnnnn }
21987 { _fp_to_tl_scientific:wnnnnn }
21988 \s_fp _fp_chk:w 1 0 {#1}
21989 }
21990 \cs_new:Npn _fp_to_tl_scientific:wnnnnn
21991 \s_fp _fp_chk:w 1 #1 #2 #3#4#5#6 ;
21992 {
21993 \exp_after:wN _fp_to_tl_scientific:wNw
21994 \exp_after:wN e
21995 \int_value:w _fp_int_eval:w #2 - 1
21996 ; #3 #4 #5 #6 ;
21997 }
21998 \cs_new:Npn _fp_to_tl_scientific:wNw #1 ; #2#3;
21999 { _fp_trim_zeros:w #2.#3 ; #1 }

```

(End definition for `\_fp_to_tl:w` and others.)

### 35.6 Formatting

This is not implemented yet, as it is not yet clear what a correct interface would be, for this kind of structured conversion from a floating point (or other types of variables) to a string. Ideas welcome.

### 35.7 Convert to dimension or integer

\fp\_to\_dim:N  
\fp\_to\_dim:c  
\fp\_to\_dim:n

All three public variants are based on the same \fp\_to\_dim\_dispatch:w after evaluating their argument to an internal floating point. We only allow floating point numbers, not tuples.

```

22000 \cs_new:Npn \fp_to_dim:N #1
22001 { \exp_after:wN \fp_to_dim_dispatch:w #1 }
22002 \cs_generate_variant:Nn \fp_to_dim:N { c }
22003 \cs_new:Npn \fp_to_dim:n
22004 {
22005 \exp_after:wN \fp_to_dim_dispatch:w
22006 \exp:w \exp_end_continue_f:w \fp_parse:n
22007 }
22008 \cs_new:Npn \fp_to_dim_dispatch:w #1#2 ;
22009 {
22010 \fp_change_func_type:NNN #1 \fp_to_dim:w \fp_to_dim_recover:w
22011 #1 #2 ;
22012 }
22013 \cs_new:Npn \fp_to_dim_recover:w #1
22014 { \fp_invalid_operation:nnw { Opt } { fp_to_dim } }
22015 \cs_new:Npn \fp_to_dim:w #1 ; { \fp_to_decimal:w #1 ; pt }
```

(End definition for \fp\_to\_dim:N and others. These functions are documented on page 203.)

\fp\_to\_int:N  
\fp\_to\_int:c  
\fp\_to\_int:n

For the most part identical to \fp\_to\_dim:N but without pt, and where \fp\_to\_int:w does more work. To convert to an integer, first round to 0 places (to the nearest integer), then express the result as a decimal number: the definition of \fp\_to\_decimal\_dispatch:w is such that there are no trailing dot nor zero.

```

22016 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \fp_to_int_dispatch:w #1 }
22017 \cs_generate_variant:Nn \fp_to_int:N { c }
22018 \cs_new:Npn \fp_to_int:n
22019 {
22020 \exp_after:wN \fp_to_int_dispatch:w
22021 \exp:w \exp_end_continue_f:w \fp_parse:n
22022 }
22023 \cs_new:Npn \fp_to_int_dispatch:w #1#2 ;
22024 {
22025 \fp_change_func_type:NNN #1 \fp_to_int:w \fp_to_int_recover:w
22026 #1 #2 ;
22027 }
22028 \cs_new:Npn \fp_to_int_recover:w #1
22029 { \fp_invalid_operation:nnw { 0 } { fp_to_int } }
22030 \cs_new:Npn \fp_to_int:w #1;
22031 {
22032 \exp_after:wN \fp_to_decimal:w \exp:w \exp_end_continue_f:w
22033 \fp_round:Nnw \fp_round_to_nearest>NNN #1; { 0 }
22034 }
```

(End definition for \fp\_to\_int:N and others. These functions are documented on page 203.)

### 35.8 Convert from a dimension

\dim\_to\_fp:n  
\\_fp\_from\_dim\_test:ww  
\\_fp\_from\_dim:wNw  
\\_fp\_from\_dim:wNNnnnnnn  
\\_fp\_from\_dim:wnnnnnNw

The dimension expression (which can in fact be a glue expression) is evaluated, converted to a number (*i.e.*, expressed in scaled points), then multiplied by  $2^{-16} =$

0.0000152587890625 to give a value expressed in points. The auxiliary `\_fp\_mul\_npos_o:Nww` expects the desired *final sign* and two floating point operands (of the form `\s_fp ... ;`) as arguments. This set of functions is also used to convert dimension registers to floating points while parsing expressions: in this context there is an additional exponent, which is the first argument of `\_fp_from_dim_test:ww`, and is combined with the exponent  $-4$  of  $2^{-16}$ . There is also a need to expand afterwards: this is performed by `\_fp_mul_npos_o:Nww`, and cancelled by `\prg_do_nothing`: here.

```

22035 \cs_new:Npn \dim_to_fp:n #1
22036 {
22037 \exp_after:wN _fp_from_dim_test:ww
22038 \exp_after:wN 0
22039 \exp_after:wN ,
22040 \int_value:w \tex_glueexpr:D #1 ;
22041 }
22042 \cs_new:Npn _fp_from_dim_test:ww #1, #2
22043 {
22044 \if_meaning:w 0 #2
22045 _fp_case_return:nw { \exp_after:wN \c_zero_fp }
22046 \else:
22047 \exp_after:wN _fp_from_dim:wNw
22048 \int_value:w _fp_int_eval:w #1 - 4
22049 \if_meaning:w - #2
22050 \exp_after:wN , \exp_after:wN 2 \int_value:w
22051 \else:
22052 \exp_after:wN , \exp_after:wN 0 \int_value:w #2
22053 \fi:
22054 \fi:
22055 }
22056 \cs_new:Npn _fp_from_dim:wNw #1,#2#3;
22057 {
22058 _fp_pack_twice_four:wNNNNNNNN _fp_from_dim:wNNnnnnnn ;
22059 #3 000 0000 00 {10}987654321; #2 {#1}
22060 }
22061 \cs_new:Npn _fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
22062 { _fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
22063 \cs_new:Npn _fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
22064 {
22065 _fp_mul_npos_o:Nww #7
22066 \s_fp _fp_chk:w 1 #7 {#5} #1 ;
22067 \s_fp _fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
22068 \prg_do_nothing:
22069 }
```

(End definition for `\dim_to_fp:n` and others. This function is documented on page 177.)

### 35.9 Use and eval

**\fp\_use:N** Those public functions are simple copies of the decimal conversions.  
**\fp\_use:c** 22070 `\cs_new_eq:NN \fp_use:N \fp_to_decimal:N`  
**\fp\_eval:n** 22071 `\cs_generate_variant:Nn \fp_use:N { c }`  
   22072 `\cs_new_eq:NN \fp_eval:n \fp_to_decimal:n`

(End definition for `\fp_use:N` and `\fp_eval:n`. These functions are documented on page 204.)

**\fp\_sign:n** Trivial but useful. See the implementation of \fp\_add:Nn for an explanation of why to use \fp\_parse:n, namely, for better error reporting.

```
22073 \cs_new:Npn \fp_sign:n #1
22074 { \fp_to_decimal:n { sign \fp_parse:n {#1} } }
```

(End definition for \fp\_sign:n. This function is documented on page 203.)

**\fp\_abs:n** Trivial but useful. See the implementation of \fp\_add:Nn for an explanation of why to use \fp\_parse:n, namely, for better error reporting.

```
22075 \cs_new:Npn \fp_abs:n #1
22076 { \fp_to_decimal:n { abs \fp_parse:n {#1} } }
```

(End definition for \fp\_abs:n. This function is documented on page 218.)

**\fp\_max:nn** Similar to \fp\_abs:n, for consistency with \int\_max:nn, etc.

```
22077 \cs_new:Npn \fp_max:nn #1#2
22078 { \fp_to_decimal:n { max (\fp_parse:n {#1} , \fp_parse:n {#2}) } }
22079 \cs_new:Npn \fp_min:nn #1#2
22080 { \fp_to_decimal:n { min (\fp_parse:n {#1} , \fp_parse:n {#2}) } }
```

(End definition for \fp\_max:nn and \fp\_min:nn. These functions are documented on page 218.)

### 35.10 Convert an array of floating points to a comma list

Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

```
\cs_new:Npn \fp_array_to_clist_loop:Nw #1#2;
{
 \use_none:n #1
 { , ~ } \fp_to_tl:n { #1 #2 ; }
 \fp_array_to_clist_loop:Nw
}
```

The \use\_ii:nn function is expanded after \fp\_expand:n is done, and it removes ,~ from the start of the representation.

```
22081 \cs_new:Npn \fp_array_to_clist:n #1
22082 {
22083 \tl_if_empty:nF {#1}
22084 {
22085 \exp_last_unbraced:Ne \use_ii:nn
22086 {
22087 \fp_array_to_clist_loop:Nw #1 { ? \prg_break: } ;
22088 \prg_break_point:
22089 }
22090 }
22091 }
22092 \cs_new:Npn \fp_array_to_clist_loop:Nw #1#2;
22093 {
22094 \use_none:n #1
22095 ,
22096 \exp_not:f { \fp_to_tl_dispatch:w #1 #2 ; }
22097 \fp_array_to_clist_loop:Nw
22098 }
```

(End definition for `\_fp_array_to_clist:n` and `\_fp_array_to_clist_loop:Nw`.)  
22099   `</package>`

## 36 I3fp-random Implementation

```
22100 <*package>
22101 <@=fp>
```

`\_fp_parse_word_rand:N` Those functions may receive a variable number of arguments. We won't use the argument `?.`  
`\_fp_parse_word_randint:N`  
22102  `\cs_new:Npn \_fp_parse_word_rand:N`  
22103  `{ \_fp_parse_function:NNN \_fp_rand_o:Nw ? }`  
22104  `\cs_new:Npn \_fp_parse_word_randint:N`  
22105  `{ \_fp_parse_function:NNN \_fp_randint_o:Nw ? }`

(End definition for `\_fp_parse_word_rand:N` and `\_fp_parse_word_randint:N`.)

### 36.1 Engine support

Most engines provide random numbers, but not all. We write the test twice simply in order to write the `false` branch first.

```
22106 \sys_if_rand_exist:F
22107 {
22108 \kernel_msg_new:nnn { kernel } { fp-no-random }
22109 { Random~numbers~unavailable~for~#1 }
22110 \cs_new:Npn _fp_rand_o:Nw ? #1 @
22111 {
22112 \kernel_msg_expandable_error:nnn { kernel } { fp-no-random }
22113 { fp-rand }
22114 \exp_after:wN \c_nan_fp
22115 }
22116 \cs_new_eq:NN _fp_randint_o:Nw _fp_rand_o:Nw
22117 \cs_new:Npn \int_rand:nn #1#2
22118 {
22119 \kernel_msg_expandable_error:nnn { kernel } { fp-no-random }
22120 { \int_rand:nn {#1} {#2} }
22121 \int_eval:n {#1}
22122 }
22123 \cs_new:Npn \int_rand:n #1
22124 {
22125 \kernel_msg_expandable_error:nnn { kernel } { fp-no-random }
22126 { \int_rand:n {#1} }
22127 1
22128 }
22129 }
22130 \sys_if_rand_exist:T
22131 {
```

Obviously, every word "random" below means "pseudo-random", as we have no access to entropy (except a very unreliable source of entropy: the time it takes to run some code).

The primitive random number generator (RNG) is provided as `\tex_uniformdeviate:D`. Under the hood, it maintains an array of 55 28-bit numbers, updated with a linear recursion relation (similar to Fibonacci numbers) modulo  $2^{28}$ . When `\tex_uniformdeviate:D <integer>` is called (for brevity denote by  $N$  the `<integer>`), the next 28-bit number is read from the array, scaled by  $N/2^{28}$ , and rounded. To prevent 0 and  $N$  from appearing half as often as other numbers, they are both mapped to the result 0.

This process means that `\tex_uniformdeviate:D` only gives a uniform distribution from 0 to  $N-1$  if  $N$  is a divisor of  $2^{28}$ , so we will mostly call the RNG with such power of 2 arguments. If  $N$  does not divide  $2^{28}$ , then the relative non-uniformity (difference between probabilities of getting different numbers) is about  $N/2^{28}$ . This implies that detecting deviation from  $1/N$  of the probability of a fixed value  $X$  requires about  $2^{56}/N$  random trials. But collective patterns can reduce this to about  $2^{56}/N^2$ . For instance with  $N = 3 \times 2^k$ , the modulo 3 repartition of such random numbers is biased with a non-uniformity about  $2^k/2^{28}$  (which is much worse than the circa  $3/2^{28}$  non-uniformity from taking directly  $N = 3$ ). This is detectable after about  $2^{56}/2^{2k} = 9 \cdot 2^{56}/N^2$  random numbers. For  $k = 15$ ,  $N = 98304$ , this means roughly  $2^{26}$  calls to the RNG (experimentally this takes at the very least 16 seconds on a 2 giga-hertz processor). While this bias is not quite problematic, it is uncomfortably close to being so, and it becomes worse as  $N$  is increased. In our code, we shall thus combine several results from the RNG.

The RNG has three types of unexpected correlations. First, everything is linear modulo  $2^{28}$ , hence the lowest  $k$  bits of the random numbers only depend on the lowest  $k$  bits of the seed (and of course the number of times the RNG was called since setting the seed). The recommended way to get a number from 0 to  $N-1$  is thus to scale the raw 28-bit integer, as the engine's RNG does. We will go further and in fact typically we discard some of the lowest bits.

Second, suppose that we call the RNG with the same argument  $N$  to get a set of  $K$  integers in  $[0, N-1]$  (throwing away repeats), and suppose that  $N > K^3$  and  $K > 55$ . The recursion used to construct more 28-bit numbers from previous ones is linear:  $x_n = x_{n-55} - x_{n-24}$  or  $x_n = x_{n-55} - x_{n-24} + 2^{28}$ . After rescaling and rounding we find that the result  $N_n \in [0, N-1]$  is among  $N_{n-55} - N_{n-24} + \{-1, 0, 1\}$  modulo  $N$  (a more detailed analysis shows that 0 appears with frequency close to  $3/4$ ). The resulting set thus has more triplets  $(a, b, c)$  than expected obeying  $a = b + c$  modulo  $N$ . Namely it will have of order  $(K - 55) \times 3/4$  such triplets, when one would expect  $K^3/(6N)$ . This starts to be detectable around  $N = 2^{18} > 55^3$  (earlier if one keeps track of positions too, but this is more subtle than it looks because the array of 28-bit integers is read backwards by the engine). Hopefully the correlation is subtle enough to not affect realistic documents so we do not specifically mitigate against this. Since we typically use two calls to the RNG per `\int_rand:nn` we would need to investigate linear relations between the  $x_{2n}$  on the one hand and between the  $x_{2n+1}$  on the other hand. Such relations will have more complicated coefficients than  $\pm 1$ , which alleviates the issue.

Third, consider successive batches of 165 calls to the RNG (with argument  $2^{28}$  or with argument 2 for instance), then most batches have more odd than even numbers. Note that this does not mean that there are more odd than even numbers overall. Similar issues are discussed in Knuth's TAOCP volume 2 near exercise 3.3.2-31. We do not have any mitigation strategy for this.

Ideally, our algorithm should be:

- Uniform. The result should be as uniform as possible assuming that the RNG's underlying 28-bit integers are uniform.

- Uncorrelated. The result should not have detectable correlations between different seeds, similar to the lowest-bit ones mentioned earlier.
- Quick. The algorithm should be fast in  $\text{\TeX}$ , so no “bit twiddling”, but “digit twiddling” is ok.
- Simple. The behaviour must be documentable precisely.
- Predictable. The number of calls to the RNG should be the same for any  $\backslash\text{int\_rand:nn}$ , because then the algorithm can be modified later without changing the result of other uses of the RNG.
- Robust. It should work even for  $\backslash\text{int_rand:nn} \{ - \backslash\text{c\_max\_int} \} \{ \backslash\text{c\_max\_int} \}$  where the range is not representable as an integer. In fact, we also provide later a floating-point  $\text{randint}$  whose range can go all the way up to  $2 \times 10^{16} - 1$  possible values.

Some of these requirements conflict. For instance, uniformity cannot be achieved with a fixed number of calls to the RNG.

Denote by  $\text{random}(N)$  one call to  $\backslash\text{tex_uniformdeviate:D}$  with argument  $N$ , and by  $\text{ediv}(p, q)$  the  $\varepsilon$ - $\text{\TeX}$  rounding division giving  $\lfloor p/q + 1/2 \rfloor$ . Denote by  $\langle \min \rangle$ ,  $\langle \max \rangle$  and  $R = \langle \max \rangle - \langle \min \rangle + 1$  the arguments of  $\backslash\text{int_min:nn}$  and the number of possible outcomes. Note that  $R \in [1, 2^{32} - 1]$  cannot necessarily be represented as an integer (however,  $R - 2^{31}$  can). Our strategy is to get two 28-bit integers  $X$  and  $Y$  from the RNG, split each into 14-bit integers, as  $X = X_1 \times 2^{14} + X_0$  and  $Y = Y_1 \times 2^{14} + Y_0$  then return essentially  $\langle \min \rangle + \lfloor R(X_1 \times 2^{-14} + Y_1 \times 2^{-28} + Y_0 \times 2^{-42} + X_0 \times 2^{-56}) \rfloor$ . For small  $R$  the  $X_0$  term has a tiny effect so we ignore it and we can compute  $R \times Y / 2^{28}$  much more directly by  $\text{random}(R)$ .

- If  $R \leq 2^{17} - 1$  then return  $\text{ediv}(R \text{random}(2^{14}) + \text{random}(R) + 2^{13}, 2^{14}) - 1 + \langle \min \rangle$ . The shifts by  $2^{13}$  and  $-1$  convert  $\varepsilon$ - $\text{\TeX}$  division to truncated division. The bound on  $R$  ensures that the number obtained after the shift is less than  $\backslash\text{c\_max\_int}$ . The non-uniformity is at most of order  $2^{17}/2^{42} = 2^{-25}$ .
- Split  $R = R_2 \times 2^{28} + R_1 \times 2^{14} + R_0$ , where  $R_2 \in [0, 15]$ . Compute  $\langle \min \rangle + R_2 X_1 2^{14} + (R_2 Y_1 + R_1 X_1) + \text{ediv}(R_2 Y_0 + R_1 Y_1 + R_0 X_1 + \text{ediv}(R_2 X_0 + R_0 Y_1 + \text{ediv}((2^{14} R_1 + R_0)(2^{14} Y_0 + X_0), 2^{28}), 2^{14}), 2^{14})$  then map a result of  $\langle \max \rangle + 1$  to  $\langle \min \rangle$ . Writing each  $\text{ediv}$  in terms of truncated division with a shift, and using  $\lfloor (p + \lfloor r/s \rfloor)/q \rfloor = \lfloor (ps + r)/(sq) \rfloor$ , what we compute is equal to  $\lfloor \langle \text{exact} \rangle + 2^{-29} + 2^{-15} + 2^{-1} \rfloor$  with  $\langle \text{exact} \rangle = \langle \min \rangle + R \times 0.X_1 Y_1 Y_0 X_0$ . Given we map  $\langle \max \rangle + 1$  to  $\langle \min \rangle$ , the shift has no effect on uniformity. The non-uniformity is bounded by  $R/2^{56} < 2^{-24}$ . It may be possible to speed up the code by dropping tiny terms such as  $R_0 X_0$ , but the analysis of non-uniformity proves too difficult.

To avoid the overflow when the computation yields  $\langle \max \rangle + 1$  with  $\langle \max \rangle = 2^{31} - 1$  (note that  $R$  is then arbitrary), we compute the result in two pieces. Compute  $\langle \text{first} \rangle = \langle \min \rangle + R_2 X_1 2^{14}$  if  $R_2 < 8$  or  $\langle \min \rangle + 8X_1 2^{14} + (R_2 - 8)X_1 2^{14}$  if  $R_2 \geq 8$ , the expressions being chosen to avoid overflow. Compute  $\langle \text{second} \rangle = R_2 Y_1 + R_1 X_1 + \text{ediv}(\dots)$ , at most  $R_2 2^{14} + R_1 2^{14} + R_0 \leq 2^{28} + 15 \times 2^{14} - 1$ , not at risk of overflowing. We have  $\langle \text{first} \rangle + \langle \text{second} \rangle = \langle \max \rangle + 1 = \langle \min \rangle + R$  if and only if  $\langle \text{second} \rangle = R 12^{14} + R_0 + R_2 2^{14}$  and  $2^{14} R_2 X_1 = 2^{28} R_2 - 2^{14} R_2$  (namely  $R_2 = 0$  or  $X_1 = 2^{14} - 1$ ). In that case, return  $\langle \min \rangle$ , otherwise return  $\langle \text{first} \rangle + \langle \text{second} \rangle$ , which is safe because it is at most  $\langle \max \rangle$ . Note that the decision of what to return

does not need  $\langle first \rangle$  explicitly so we don't actually compute it, just put it in an integer expression in which  $\langle second \rangle$  is eventually added (or not).

- To get a floating point number in  $[0, 1)$  just call the  $R = 10000 \leq 2^{17} - 1$  procedure above to produce four blocks of four digits.
- To get an integer floating point number in a range (whose size can be up to  $2 \times 10^{16} - 1$ ), work with fixed-point numbers: get six times four digits to build a fixed point number, multiply by  $R$  and add  $\langle min \rangle$ . This requires some care because l3fp-extended only supports non-negative numbers.

`\c_kernel_randint_max_int` Constant equal to  $2^{17} - 1$ , the maximal size of a range that `\int_range:nn` can do with its “simple” algorithm.

```
22132 \int_const:Nn \c_kernel_randint_max_int { 131071 }
```

(End definition for `\c_kernel_randint_max_int`.)

`\_kernel_randint:n` Used in an integer expression, `\_kernel_randint:n {R}` gives a random number  $1 + \lfloor (R \text{random}(2^{14}) + \text{random}(R)) / 2^{14} \rfloor$  that is in  $[1, R]$ . Previous code was computing  $\lfloor p / 2^{14} \rfloor$  as  $\text{ediv}(p - 2^{13}, 2^{14})$  but that wrongly gives  $-1$  for  $p = 0$ .

```
22133 \cs_new:Npn _kernel_randint:n #1
22134 {
22135 (#1 * \tex_uniformdeviate:D 16384
22136 + \tex_uniformdeviate:D #1 + 8192) / 16384
22137 }
```

(End definition for `\_kernel_randint:n`.)

`\_fp_rand_myriads:n` Used as `\_fp_rand_myriads:n {XXX}` with one letter X (specifically) per block of four digit we want; it expands to ; followed by the requested number of brace groups, each containing four (pseudo-random) digits. Digits are produced as a random number in  $[10000, 19999]$  for the usual reason of preserving leading zeros.

```
22138 \cs_new:Npn _fp_rand_myriads:n #1
22139 { _fp_rand_myriads_loop:w #1 \prg_break: X \prg_break_point: ; }
22140 \cs_new:Npn _fp_rand_myriads_loop:w #1 X
22141 {
22142 #1
22143 \exp_after:wN _fp_rand_myriads_get:w
22144 \int_value:w _fp_int_eval:w 9999 +
22145 _kernel_randint:n { 10000 }
22146 _fp_rand_myriads_loop:w
22147 }
22148 \cs_new:Npn _fp_rand_myriads_get:w 1 #1 ; { ; {#1} }
```

(End definition for `\_fp_rand_myriads:n`, `\_fp_rand_myriads_loop:w`, and `\_fp_rand_myriads_get:w`.)

## 36.2 Random floating point

```
__fp_rand_o:Nw
__fp_rand_o:w
```

First we check that `random` was called without argument. Then get four blocks of four digits and convert that fixed point number to a floating point number (this correctly sets the exponent). This has a minor bug: if all of the random numbers are zero then the result is correctly 0 but it raises the `underflow` flag; it should not do that.

```
22149 \cs_new:Npn __fp_rand_o:Nw ? #1 @
22150 {
22151 \tl_if_empty:nTF {#1}
22152 {
22153 \exp_after:wN __fp_rand_o:w
22154 \exp:w \exp_end_continue_f:w
22155 __fp_rand_myriads:n { XXXX } { 0000 } { 0000 } ; 0
22156 }
22157 {
22158 __kernel_msg_expandable_error:nnnn
22159 { kernel } { fp-num-args } { rand() } { 0 } { 0 }
22160 \exp_after:wN \c_nan_fp
22161 }
22162 }
22163 \cs_new:Npn __fp_rand_o:w ;
22164 {
22165 \exp_after:wN __fp_sanitize:Nw
22166 \exp_after:wN 0
22167 \int_value:w __fp_int_eval:w \c_zero_int
22168 __fp_fixed_to_float_o:wN
22169 }
```

(End definition for `\__fp_rand_o:Nw` and `\__fp_rand_o:w`.)

## 36.3 Random integer

Enforce that there is one argument (then add first argument 1) or two arguments. Call `\__fp_randint_badarg:w` on each; this function inserts 1 `\exp_stop_f:` to end the `\if_case:w` statement if either the argument is not an integer or if its absolute value is  $\geq 10^{16}$ . Also bail out if `\__fp_compare_back:ww` yields 1, meaning that the bounds are not in the right order. Otherwise an auxiliary converts each argument times  $10^{-16}$  (hence the shift in exponent) to a 24-digit fixed point number (see l3fp-extended). Then compute the number of choices,  $\langle max \rangle + 1 - \langle min \rangle$ . Create a random 24-digit fixed-point number with `\__fp_rand_myriads:n`, then use a fused multiply-add instruction to multiply the number of choices to that random number and add it to  $\langle min \rangle$ . Then truncate to 16 digits (namely select the integer part of  $10^{16}$  times the result) before converting back to a floating point number (`\__fp_sanitize:Nw` takes care of zero). To avoid issues with negative numbers, add 1 to all fixed point numbers (namely  $10^{16}$  to the integers they represent), except of course when it is time to convert back to a float.

```
22170 \cs_new:Npn __fp_randint_o:Nw ?
22171 {
22172 __fp_parse_function_one_two:nnw
22173 { randint }
22174 { __fp_randint_default:w __fp_randint_o:w }
22175 }
22176 \cs_new:Npn __fp_randint_default:w #1 { \exp_after:wN #1 \c_one_fp }
22177 \cs_new:Npn __fp_randint_badarg:w \s_fp __fp_chk:w #1#2#3;
```

```

22178 {
22179 __fp_int:wTF \s_fp __fp_chk:w #1#2#3;
22180 {
22181 \if_meaning:w 1 #1
22182 \if_int_compare:w
22183 __fp_use_i_until_s:nw #3 ; > \c_fp_prec_int
22184 1 \exp_stop_f:
22185 \fi:
22186 \fi:
22187 }
22188 { 1 \exp_stop_f: }
22189 }
22190 \cs_new:Npn __fp_randint_o:w #1; #2; @
22191 {
22192 \if_case:w
22193 __fp_randint_badarg:w #1;
22194 __fp_randint_badarg:w #2;
22195 \if:w 1 __fp_compare_back:ww #2; #1; 1 \exp_stop_f: \fi:
22196 0 \exp_stop_f:
22197 __fp_randint_auxi_o:ww #1; #2;
22198 \or:
22199 __fp_invalid_operation_tl_o:ff
22200 { randint } { __fp_array_to_clist:n { #1; #2; } }
22201 \exp:w
22202 \fi:
22203 \exp_after:wN \exp_end:
22204 }
22205 \cs_new:Npn __fp_randint_auxi_o:ww #1 ; #2 ; #3 \exp_end:
22206 {
22207 \fi:
22208 __fp_randint_auxii:wn #2 ;
22209 { __fp_randint_auxii:wn #1 ; __fp_randint_auxiii_o:ww }
22210 }
22211 \cs_new:Npn __fp_randint_auxii:wn \s_fp __fp_chk:w #1#2#3#4 ;
22212 {
22213 \if_meaning:w 0 #1
22214 \exp_after:wN \use_i:nn
22215 \else:
22216 \exp_after:wN \use_ii:nn
22217 \fi:
22218 { \exp_after:wN __fp_fixed_continue:wn \c_fp_one_fixed_tl }
22219 {
22220 \exp_after:wN __fp_ep_to_fixed:wwn
22221 \int_value:w __fp_int_eval:w
22222 #3 - \c_fp_prec_int , #4 {0000} {0000} ;
22223 {
22224 \if_meaning:w 0 #2
22225 \exp_after:wN \use_i:nnnn
22226 \exp_after:wN __fp_fixed_add_one:wn
22227 \fi:
22228 \exp_after:wN __fp_fixed_sub:wwn \c_fp_one_fixed_tl
22229 }
22230 __fp_fixed_continue:wn
22231 }

```

```

22232 }
22233 \cs_new:Npn __fp_randint_auxiii_o:ww #1 ; #2 ;
22234 {
22235 __fp_fixed_add:wwn #2 ;
22236 {0000} {0000} {0000} {0001} {0000} {0000} ;
22237 __fp_fixed_sub:wwn #1 ;
22238 {
22239 \exp_after:wN \use_i:nn
22240 \exp_after:wN __fp_fixed_mul_add:www
22241 \exp:w \exp_end_continue_f:w __fp_rand_myriads:n { XXXXXX } ;
22242 }
22243 #1 ;
22244 __fp_randint_auxiv_o:ww
22245 #2 ;
22246 __fp_randint_auxv_o:w #1 ; @
22247 }
22248 \cs_new:Npn __fp_randint_auxiv_o:ww #1#2#3#4#5 ; #6#7#8#9
22249 {
22250 \if_int_compare:w
22251 \if_int_compare:w #1#2 > #6#7 \exp_stop_f: 1 \else:
22252 \if_int_compare:w #1#2 < #6#7 \exp_stop_f: - \fi: \fi:
22253 #3#4 > #8#9 \exp_stop_f:
22254 __fp_use_i_until_s:nw
22255 \fi:
22256 __fp_randint_auxv_o:w {#1}{#2}{#3}{#4}#5
22257 }
22258 \cs_new:Npn __fp_randint_auxv_o:w #1#2#3#4#5 ; #6 @
22259 {
22260 \exp_after:wN __fp_SANITIZE:Nw
22261 \int_value:w
22262 \if_int_compare:w #1 < 10000 \exp_stop_f:
22263 2
22264 \else:
22265 0
22266 \exp_after:wN \exp_after:wN
22267 \exp_after:wN __fp_reverse_args:Nww
22268 \fi:
22269 \exp_after:wN __fp_fixed_sub:wwn \c_fp_one_fixed_tl
22270 {#1} {#2} {#3} {#4} {0000} {0000} ;
22271 {
22272 \exp_after:wN \exp_stop_f:
22273 \int_value:w __fp_int_eval:w \c_fp_prec_int
22274 __fp_fixed_to_float_o:wN
22275 }
22276 0
22277 \exp:w \exp_after:wN \exp_end:
22278 }

```

(End definition for `\__fp_randint_o:Nw` and others.)

`\int_rand:nn`  
`\__fp_randint:ww`

Evaluate the argument and filter out the case where the lower bound #1 is more than the upper bound #2. Then determine whether the range is narrower than `\c_kernel_randint_max_int`; #2-#1 may overflow for very large positive #2 and negative #1. If the range is narrow, call `\__kernel_randint:n {<choices>}` where `<choices>` is the number

of possible outcomes. If the range is wide, use somewhat slower code.

```

22279 \cs_new:Npn \int_rand:nn #1#2
22280 {
22281 \int_eval:n
22282 {
22283 \exp_after:wN __fp_randint:ww
22284 \int_value:w \int_eval:n {#1} \exp_after:wN ;
22285 \int_value:w \int_eval:n {#2} ;
22286 }
22287 }
22288 \cs_new:Npn __fp_randint:ww #1; #2;
22289 {
22290 \if_int_compare:w #1 > #2 \exp_stop_f:
22291 __kernel_msg_expandable_error:nnnn
22292 { kernel } { randint-backward-range } {#1} {#2}
22293 __fp_randint:ww #2; #1;
22294 \else:
22295 \if_int_compare:w __fp_int_eval:w #2
22296 \if_int_compare:w #1 > \c_zero_int
22297 - #1 < __fp_int_eval:w
22298 \else:
22299 < __fp_int_eval:w #1 +
22300 \fi:
22301 \c_kernel_randint_max_int
22302 __fp_int_eval_end:
22303 __kernel_randint:n
22304 { __fp_int_eval:w #2 - #1 + 1 __fp_int_eval_end: }
22305 - 1 + #1
22306 \else:
22307 __kernel_randint:nn {#1} {#2}
22308 \fi:
22309 \fi:
22310 }
```

(End definition for `\int_rand:nn` and `\__fp_randint:ww`. This function is documented on page 99.)

`\__kernel_randint:nn`  
`\__fp_randint_split_o:Nw`  
`\__fp_randint_split_aux:w`  
`\__fp_randinat_wide_aux:w`  
`\__fp_randinat_wide_auxii:w`

Any  $n \in [-2^{31} + 1, 2^{31} - 1]$  is uniquely written as  $2^{14}n_1 + n_2$  with  $n_1 \in [-2^{17}, 2^{17} - 1]$  and  $n_2 \in [0, 2^{14} - 1]$ . Calling `\__fp_randint_split_o:Nw n`; gives  $n_1$ ;  $n_2$ ; and expands the next token once. We do this for two random numbers and apply `\__fp_randint_split_o:Nw` twice to fully decompose the range  $R$ . One subtlety is that we compute  $R - 2^{31} = \langle \max \rangle - \langle \min \rangle - (2^{31} - 1) \in [-2^{31} + 1, 2^{31} - 1]$  rather than  $R$  to avoid overflow.

Then we have `\__fp_randint_wide_aux:w`  $\langle X_1 \rangle; \langle X_0 \rangle; \langle Y_1 \rangle; \langle Y_0 \rangle; \langle R_2 \rangle; \langle R_1 \rangle; \langle R_0 \rangle$ ; and we apply the algorithm described earlier.

```

22311 \cs_new:Npn __kernel_randint:nn #1#2
22312 {
22313 #1
22314 \exp_after:wN __fp_randint_wide_aux:w
22315 \int_value:w
22316 \exp_after:wN __fp_randint_split_o:Nw
22317 \tex_uniformdeviate:D 268435456 ;
22318 \int_value:w
22319 \exp_after:wN __fp_randint_split_o:Nw
22320 \tex_uniformdeviate:D 268435456 ;
22321 \int_value:w
```

```

22322 \exp_after:wN __fp_randint_split_o:Nw
22323 \int_value:w __fp_int_eval:w 131072 +
22324 \exp_after:wN __fp_randint_split_o:Nw
22325 \int_value:w
22326 __kernel_int_add:nnn {\#2} { -\#1 } { -\c_max_int } ;
22327 .
22328 }
22329 \cs_new:Npn __fp_randint_split_o:Nw #1#2 ;
22330 {
22331 \if_meaning:w 0 #1
22332 0 \exp_after:wN ; \int_value:w 0
22333 \else:
22334 \exp_after:wN __fp_randint_split_aux:w
22335 \int_value:w __fp_int_eval:w (#1#2 - 8192) / 16384 ;
22336 + #1#2
22337 \fi:
22338 \exp_after:wN ;
22339 }
22340 \cs_new:Npn __fp_randint_split_aux:w #1 ;
22341 {
22342 #1 \exp_after:wN ;
22343 \int_value:w __fp_int_eval:w - #1 * 16384
22344 }
22345 \cs_new:Npn __fp_randint_wide_aux:w #1;#2; #3;#4; #5;#6;#7; .
22346 {
22347 \exp_after:wN __fp_randint_wide_auxii:w
22348 \int_value:w __fp_int_eval:w #5 * #3 + #6 * #1 +
22349 (#5 * #4 + #6 * #3 + #7 * #1 +
22350 (#5 * #2 + #7 * #3 +
22351 (16384 * #6 + #7) * (16384 * #4 + #2) / 268435456) / 16384
22352) / 16384 \exp_after:wN ;
22353 \int_value:w __fp_int_eval:w (#5 + #6) * 16384 + #7 ;
22354 #1 ; #5 ;
22355 }
22356 \cs_new:Npn __fp_randint_wide_auxii:w #1; #2; #3; #4;
22357 {
22358 \if_int_odd:w 0
22359 \if_int_compare:w #1 = #2 \else: \exp_stop_f: \fi:
22360 \if_int_compare:w #4 = \c_zero_int 1 \fi:
22361 \if_int_compare:w #3 = 16383 ~ 1 \fi:
22362 \exp_stop_f:
22363 \exp_after:wN \prg_break:
22364 \fi:
22365 \if_int_compare:w #4 < 8 \exp_stop_f:
22366 + #4 * #3 * 16384
22367 \else:
22368 + 8 * #3 * 16384 + (#4 - 8) * #3 * 16384
22369 \fi:
22370 + #1
22371 \prg_break_point:
22372 }

```

(End definition for `\__kernel randint:nn` and others.)

**\int\_rand:n** Similar to \int\_rand:nn, but needs fewer checks.

```

22373 \cs_new:Npn \int_rand:n #1
22374 {
22375 \int_eval:n
22376 { \exp_args:Nf __fp_randint:n { \int_eval:n {#1} } }
22377 }
22378 \cs_new:Npn __fp_randint:n #1
22379 {
22380 \if_int_compare:w #1 < 1 \exp_stop_f:
22381 __kernel_msg_expandable_error:nnnn
22382 { kernel } { randint-backward-range } { 1 } { #1 }
22383 __fp_randint:ww #1; 1;
22384 \else:
22385 \if_int_compare:w #1 > \c_kernel_randint_max_int
22386 __kernel_randint:nn { 1 } { #1 }
22387 \else:
22388 __kernel_randint:n {#1}
22389 \fi:
22390 \fi:
22391 }
22392 }
22393 ⟨/package⟩

```

(End definition for \int\_rand:n and \\_\_fp\_randint:n. This function is documented on page 99.)

End the initial conditional that ensures these commands are only defined in engines that support random numbers.

## 37 I3fparray implementation

```

22394 ⟨*package⟩
22395 ⟨@=fp⟩

```

In analogy to \3intarray it would make sense to have <@=fparray>, but we need direct access to \\_\_fp\_parse:n from \3fp-parse, and a few other (less crucial) internals of the \3fp family.

### 37.1 Allocating arrays

There are somewhat more than  $(2^{31}-1)^2$  floating point numbers so we store each floating point number as three entries in integer arrays. To avoid having to multiply indices by three or to add 1 etc, a floating point array is just a token list consisting of three tokens: integer arrays of the same size.

\g\_fp\_array\_int Used to generate unique names for the three integer arrays.

```

22396 \int_new:N \g_fp_array_int

```

(End definition for \g\_fp\_array\_int.)

\l\_fp\_array\_loop\_int Used to loop in \fp\_array\_gzero:N.

```

22397 \int_new:N \l_fp_array_loop_int

```

(End definition for \l\_fp\_array\_loop\_int.)

\fpararray\_new:Nn  
\fpararray\_new:cN  
\\_fp\_array\_new:nNNN

Build a three-token token list, then define all three tokens to be integer arrays of the same size. No need to initialize the data: the integer arrays start with zeros, and three zeros denote precisely \c\_zero\_fp, as we want.

```

22398 \cs_new_protected:Npn \fpararray_new:Nn #1#2
22399 {
22400 \tl_new:N #1
22401 \prg_replicate:nn { 3 }
22402 {
22403 \int_gincr:N \g__fp_array_int
22404 \exp_args:NNc \tl_gput_right:Nn #1
22405 { g__fp_array_ _fp_int_to_roman:w \g__fp_array_int _intarray }
22406 }
22407 \exp_last_unbraced:Nfo _fp_array_new:nNNNN
22408 { \int_eval:n {#2} } #1 #1
22409 }
22410 \cs_generate_variant:Nn \fpararray_new:Nn { c }
22411 \cs_new_protected:Npn _fp_array_new:nNNNN #1#2#3#4#5
22412 {
22413 \int_compare:nNnTF {#1} < 0
22414 {
22415 _kernel_msg_error:nnn { kernel } { negative-array-size } {#1}
22416 \cs_undefine:N #1
22417 \int_gsub:Nn \g__fp_array_int { 3 }
22418 }
22419 {
22420 \intarray_new:Nn #2 {#1}
22421 \intarray_new:Nn #3 {#1}
22422 \intarray_new:Nn #4 {#1}
22423 }
22424 }
```

(End definition for \fpararray\_new:Nn and \\_fp\_array\_new:nNNN. This function is documented on page 221.)

\fpararray\_count:N Size of any of the intarrays, here we pick the third.

\fpararray\_count:c

```

22425 \cs_new:Npn \fpararray_count:N #1
22426 {
22427 \exp_after:wN \use_i:nnn
22428 \exp_after:wN \intarray_count:N #1
22429 }
22430 \cs_generate_variant:Nn \fpararray_count:N { c }
```

(End definition for \fpararray\_count:N. This function is documented on page 221.)

## 37.2 Array items

See the \intarray analogue: only names change. The functions \fpararray\_gset:Nnn and \fpararray\_item:Nn share bounds checking. The T branch is used if #3 is within bounds of the array #2.

```

22431 \cs_new:Npn _fp_array_bounds:NNnTF #1#2#3#4#5
22432 {
22433 \if_int_compare:w 1 > #3 \exp_stop_f:
22434 _fp_array_bounds_error>NNn #1 #2 {#3}
22435 #5
```

```

22436 \else:
22437 \if_int_compare:w #3 > \fparray_count:N #2 \exp_stop_f:
22438 _fp_array_bounds_error>NNn #1 #2 {#3}
22439 #5
22440 \else:
22441 #4
22442 \fi:
22443 \fi:
22444 }
22445 \cs_new:Npn _fp_array_bounds_error>NNn #1#2#3
22446 {
22447 #1 { kernel } { out-of-bounds }
22448 { \token_to_str:N #2 } {#3} { \fparray_count:N #2 }
22449 }

```

(End definition for `\_fp_array_bounds>NNnTF` and `\_fp_array_bounds_error>NNn`.)

### `\fparray_gset:Nnn` `\fparray_gset:cnn`

Evaluate, then store exponent in one intarray, sign and 8 digits of mantissa in the next, and 8 trailing digits in the last.

```

22450 \cs_new_protected:Npn \fparray_gset:Nnn #1#2#3
22451 {
22452 \exp_after:wN \exp_after:wN
22453 \exp_after:wN _fp_array_gset>NNNNww
22454 \exp_after:wN #1
22455 \exp_after:wN #1
22456 \int_value:w \int_eval:n {#2} \exp_after:wN ;
22457 \exp:w \exp_end_continue_f:w _fp_parse:n {#3}
22458 }
22459 \cs_generate_variant:Nn \fparray_gset:Nnn { c }
22460 \cs_new_protected:Npn _fp_array_gset>NNNNww #1#2#3#4#5 ; #6 ;
22461 {
22462 _fp_array_bounds>NNnTF _kernel_msg_error:nnxxx #4 {#5}
22463 {
22464 \exp_after:wN _fp_change_func_type>NNN
22465 _fp_use_i_until_s:nw #6 ;
22466 _fp_array_gset:w
22467 _fp_array_gset_recover:Nw
22468 #6 ; {#5} #1 #2 #3
22469 }
22470 { }
22471 }
22472 \cs_new_protected:Npn _fp_array_gset_recover:Nw #1#2 ;
22473 {
22474 _fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
22475 \exp_after:wN #1 \c_nan_fp
22476 }
22477 \cs_new_protected:Npn _fp_array_gset:w \s_fp _fp_chk:w #1#2
22478 {
22479 \if_case:w #1 \exp_stop_f:
22480 _fp_case_return:nw { _fp_array_gset_special:nnNNN {#2} }
22481 \or: \exp_after:wN _fp_array_gset_normal:w
22482 \or: _fp_case_return:nw { _fp_array_gset_special:nnNNN { #2 3 } }
22483 \or: _fp_case_return:nw { _fp_array_gset_special:nnNNN { 1 } }
22484 \fi:

```

```

22485 \s__fp __fp_chk:w #1 #2
22486 }
22487 \cs_new_protected:Npn __fp_array_gset_normal:w
22488 \s__fp __fp_chk:w 1 #1 #2 #3#4#5 ; #6#7#8#9
22489 {
22490 __kernel_intarray_gset:Nnn #7 {#6} {#2}
22491 __kernel_intarray_gset:Nnn #8 {#6}
22492 { \if_meaning:w 2 #1 3 \else: 1 \fi: #3#4 }
22493 __kernel_intarray_gset:Nnn #9 {#6} { 1 \use:nn #5 }
22494 }
22495 \cs_new_protected:Npn __fp_array_gset_special:nnNNN #1#2#3#4#5
22496 {
22497 __kernel_intarray_gset:Nnn #3 {#2} {#1}
22498 __kernel_intarray_gset:Nnn #4 {#2} {0}
22499 __kernel_intarray_gset:Nnn #5 {#2} {0}
22500 }

```

(End definition for `\fparray_gset:Nnn` and others. This function is documented on page 221.)

```

\fparray_gzero:N
\fparray_gzero:c
22501 \cs_new_protected:Npn \fparray_gzero:N #1
22502 {
22503 \int_zero:N \l__fp_array_loop_int
22504 \prg_replicate:nn { \fparray_count:N #1 }
22505 {
22506 \int_incr:N \l__fp_array_loop_int
22507 \exp_after:wN __fp_array_gset_special:nnNNN
22508 \exp_after:wN 0
22509 \exp_after:wN \l__fp_array_loop_int
22510 #1
22511 }
22512 }
22513 \cs_generate_variant:Nn \fparray_gzero:N { c }

```

(End definition for `\fparray_gzero:N`. This function is documented on page 221.)

```

\fparray_item:Nn
\fparray_item:cn
\fparray_item_to_tl:Nn
\fparray_item_to_tl:cn
 __fp_array_item:NwN
 __fp_array_item:NNNnN
 __fp_array_item:N
 __fp_array_item:w
 __fp_array_item_special:w
 __fp_array_item_normal:w
22514 \cs_new:Npn \fparray_item:Nn #1#2
22515 {
22516 \exp_after:wN __fp_array_item:NwN
22517 \exp_after:wN #1
22518 \int_value:w \int_eval:n {#2} ;
22519 __fp_to_decimal:w
22520 }
22521 \cs_generate_variant:Nn \fparray_item:Nn { c }
22522 \cs_new:Npn \fparray_item_to_tl:Nn #1#2
22523 {
22524 \exp_after:wN __fp_array_item:NwN
22525 \exp_after:wN #1
22526 \int_value:w \int_eval:n {#2} ;
22527 __fp_to_tl:w
22528 }
22529 \cs_generate_variant:Nn \fparray_item_to_tl:Nn { c }
22530 \cs_new:Npn __fp_array_item:NwN #1#2 ; #3
22531 {

```

```

22532 __fp_array_bounds:NNnTF __kernel_msg_expandable_error:nnfff #1 {#2}
22533 { \exp_after:wN __fp_array_item:NNNnN #1 {#2} #3 }
22534 { \exp_after:wN #3 \c_nan_fp }
22535 }
22536 \cs_new:Npn __fp_array_item:NNNnN #1#2#3#4
22537 {
22538 \exp_after:wN __fp_array_item:N
22539 \int_value:w __kernel_intarray_item:Nn #2 {#4} \exp_after:wN ;
22540 \int_value:w __kernel_intarray_item:Nn #3 {#4} \exp_after:wN ;
22541 \int_value:w __kernel_intarray_item:Nn #1 {#4} ;
22542 }
22543 \cs_new:Npn __fp_array_item:N #1
22544 {
22545 \if_meaning:w 0 #1 \exp_after:wN __fp_array_item_special:w \fi:
22546 __fp_array_item:w #1
22547 }
22548 \cs_new:Npn __fp_array_item:w #1 #2#3#4#5 #6 ; 1 #7 ;
22549 {
22550 \exp_after:wN __fp_array_item_normal:w
22551 \int_value:w \if_meaning:w #1 1 0 \else: 2 \fi: \exp_stop_f:
22552 #7 ; {#2#3#4#5} {#6} ;
22553 }
22554 \cs_new:Npn __fp_array_item_special:w #1 ; #2 ; #3 ; #4
22555 {
22556 \exp_after:wN #4
22557 \exp:w \exp_end_continue_f:w
22558 \if_case:w #3 \exp_stop_f:
22559 \exp_after:wN \c_zero_fp
22560 \or: \exp_after:wN \c_nan_fp
22561 \or: \exp_after:wN \c_minus_zero_fp
22562 \or: \exp_after:wN \c_inf_fp
22563 \else: \exp_after:wN \c_minus_inf_fp
22564 \fi:
22565 }
22566 \cs_new:Npn __fp_array_item_normal:w #1 #2#3#4#5 #6 ; #7 ; #8 ; #9
22567 { #9 \s_fp __fp_chk:w 1 #1 {#8} #7 {#2#3#4#5} {#6} ; }

```

(End definition for `\fparray_item:Nn` and others. These functions are documented on page 221.)

```
22568 </package>
```

## 38 l3cctab implementation

```

22569 <*package>
22570 <@=cctab>
```

As `LuaTeX` offers engine support for category code tables, and this is entirely lacking from the other engines, we need two complementary approaches. (Some future `XeTeX` may add support, at which point the conditionals below would be different.)

### 38.1 Variables

`\g_cctab_stack_seq` List of catcode tables saved by nested `\cctab_begin:N`, to restore catcodes at the matching `\cctab_end:`. When popped from the `\g_cctab_stack_seq` the table numbers are stored in `\g_cctab_unused_seq` for later reuse.

|                                                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                              | <pre>22571 \seq_new:N \g__cctab_stack_seq 22572 \seq_new:N \g__cctab_unused_seq</pre> <p>(End definition for <code>\g__cctab_stack_seq</code> and <code>\g__cctab_unused_seq</code>.)</p>                                                                                                                                                                                                                                                                       |
| <code>\g__cctab_group_seq</code>                                             | A stack to store the group level when a catcode table started.                                                                                                                                                                                                                                                                                                                                                                                                  |
|                                                                              | <pre>22573 \seq_new:N \g__cctab_group_seq</pre> <p>(End definition for <code>\g__cctab_group_seq</code>.)</p>                                                                                                                                                                                                                                                                                                                                                   |
| <code>\g__cctab_allocate_int</code>                                          | <p>Integer to keep track of what category code table to allocate. In LuaTeX it is only used in format mode to implement <code>\cctab_new:N</code>. In other engines it is used to make csnames for dynamic tables.</p> <pre>22574 \int_new:N \g__cctab_allocate_int</pre> <p>(End definition for <code>\g__cctab_allocate_int</code>.)</p>                                                                                                                      |
| <code>\l__cctab_internal_a_tl</code><br><code>\l__cctab_internal_b_tl</code> | <p>Scratch space. For instance, when popping <code>\g__cctab_stack_seq</code>/<code>\g__cctab_unused_seq</code>, consists of the catcodetable number (integer denotation) in LuaTeX, or of an intarray variable (as a single token) in other engines.</p> <pre>22575 \tl_new:N \l__cctab_internal_a_tl 22576 \tl_new:N \l__cctab_internal_b_tl</pre> <p>(End definition for <code>\l__cctab_internal_a_tl</code> and <code>\l__cctab_internal_b_tl</code>.)</p> |
| <code>\g__cctab_endlinechar_prop</code>                                      | <p>In LuaTeX we store the <code>\endlinechar</code> associated to each <code>\catcodetable</code> in a property list, unless it is the default value 13.</p> <pre>22577 \prop_new:N \g__cctab_endlinechar_prop</pre> <p>(End definition for <code>\g__cctab_endlinechar_prop</code>.)</p>                                                                                                                                                                       |

## 38.2 Allocating category code tables

**\cctab\_new:N**  
**\cctab\_new:c**  
**\cctab\_new:N**  
**\cctab\_gstore:Nnn**

The `\cctab_new:N` auxiliary allocates a new catcode table but does not attempt to set its value consistently across engines. It is used both in `\cctab_new:N`, which sets catcodes to iniTeX values, and in `\cctab_begin:N/\cctab_end:` for dynamically allocated tables.

First, the LuaTeX case. Creating a new category code table is done like other registers. In ConTeXt, `\newcatcodetable` does not include the initialisation, so that is added explicitly.

```
22578 \sys_if_engine_luatex:TF
22579 {
22580 \cs_new_protected:Npn \cctab_new:N #1
22581 {
22582 __kernel_chk_if_free_cs:N #1
22583 __cctab_new:N #1
22584 }
22585 \cs_new_protected:Npn __cctab_new:N #1
22586 {
22587 \newcatcodetable #1
22588 \tex_initcatcodetable:D #1
22589 }
22590 }
```

Now the case for other engines. Here, each table is an integer array. Following the LuaTeX pattern, a new table starts with `iniTeX` codes. The index base is out-by-one, so we have an internal function to handle that. The `iniTeX \endlinechar` is 13.

```

22591 {
22592 \cs_new_protected:Npn __cctab_new:N #1
22593 { \intarray_new:Nn #1 { 257 } }
22594 \cs_new_protected:Npn __cctab_gstore:Nnn #1#2#3
22595 { \intarray_gset:Nnn #1 { \int_eval:n { #2 + 1 } } {#3} }
22596 \cs_new_protected:Npn \cctab_new:N #1
22597 {
22598 __kernel_chk_if_free_cs:N #1
22599 __cctab_new:N #1
22600 \int_step_inline:nn { 256 }
22601 { __kernel_intarray_gset:Nnn #1 {##1} { 12 } }
22602 __kernel_intarray_gset:Nnn #1 { 257 } { 13 }
22603 __cctab_gstore:Nnn #1 { 0 } { 9 }
22604 __cctab_gstore:Nnn #1 { 13 } { 5 }
22605 __cctab_gstore:Nnn #1 { 32 } { 10 }
22606 __cctab_gstore:Nnn #1 { 37 } { 14 }
22607 \int_step_inline:nnn { 65 } { 90 }
22608 { __cctab_gstore:Nnn #1 {##1} { 11 } }
22609 __cctab_gstore:Nnn #1 { 92 } { 0 }
22610 \int_step_inline:nnn { 97 } { 122 }
22611 { __cctab_gstore:Nnn #1 {##1} { 11 } }
22612 __cctab_gstore:Nnn #1 { 127 } { 15 }
22613 }
22614 }
22615 \cs_generate_variant:Nn \cctab_new:N { c }
```

(End definition for `\cctab_new:N`, `\__cctab_new:N`, and `\__cctab_gstore:Nnn`. This function is documented on page 222.)

### 38.3 Saving category code tables

`\__cctab_gset:n` In various functions we need to save the current catcodes (globally) in a table. In LuaTeX, saving the catcodes is a primitives, but the `\endlinechar` needs more work: to avoid filling `\g__cctab_endlinechar_prop` with many entries we special-case the default value 13. In other engines we store 256 current catcodes and the `\endlinechar` in an intarray variable.

```

22616 \sys_if_engine_luatex:TF
22617 {
22618 \cs_new_protected:Npn __cctab_gset:n #1
22619 { \exp_args:Nf __cctab_gset_aux:n { \int_eval:n {#1} } }
22620 \cs_new_protected:Npn __cctab_gset_aux:n #1
22621 {
22622 \tex_savecatcodetable:D #1 \scan_stop:
22623 \int_compare:nNnTF { \tex_endlinechar:D } = { 13 }
22624 { \prop_gremove:Nn \g__cctab_endlinechar_prop {#1} }
22625 {
22626 \prop_gput:NnV \g__cctab_endlinechar_prop {#1}
22627 \tex_endlinechar:D
22628 }
22629 }
```

```

22630 }
22631 {
22632 \cs_new_protected:Npn __cctab_gset:n #1
22633 {
22634 \int_step_inline:nn { 256 }
22635 {
22636 __kernel_intarray_gset:Nnn #1 {##1}
22637 { \char_value_catcode:n { ##1 - 1 } }
22638 }
22639 __kernel_intarray_gset:Nnn #1 { 257 }
22640 { \tex_endlinechar:D }
22641 }
22642 }

```

(End definition for `\__cctab_gset:n` and `\__cctab_gset_aux:n`.)

`\cctab_gset:Nn`  
`\cctab_gset:cn`

Category code tables are always global, so only one version of assignments is needed. Simply run the setup in a group and save the result in a category code table #1, provided it is valid. The internal function is defined above depending on the engine.

```

22643 \cs_new_protected:Npn \cctab_gset:Nn #1#2
22644 {
22645 __cctab_chk_if_valid:NT #1
22646 {
22647 \group_begin:
22648 \cctab_select:N \c_initex_cctab
22649 #2 \scan_stop:
22650 __cctab_gset:n {#1}
22651 \group_end:
22652 }
22653 }
22654 \cs_generate_variant:Nn \cctab_gset:Nn { c }

```

(End definition for `\cctab_gset:Nn`. This function is documented on page [222](#).)

### 38.4 Using category code tables

`\g_cctab_internal_cctab`  
`\_cctab_internal_cctab_name:`

In `LuaTeX`, we must ensure that the saved tables are read-only. This is done by applying the saved table, then switching immediately to a scratch table. Any later catcode assignment will affect that scratch table rather than the saved one. If we simply switched to the saved tables, then `\char_set_catcode_other:N` in the example below would change `\c_document_cctab` and a later use of that table would give the wrong category code to `_`.

```

\use:n
{
 \cctab_begin:N \c_document_cctab
 \char_set_catcode_other:N _
 \cctab_end:
 \cctab_begin:N \c_document_cctab
 \int_compare:nTF { \char_value_catcode:n { '_' } = 8 }
 { \TRUE } { \ERROR }
 \cctab_end:
}

```

We must also make sure that a scratch table is never reused in a nested group: in the following example, the scratch table used by the first `\cctab_begin:N` would be changed globally by the second one issuing `\savecatcodetable`, and after `\group_end:` the wrong category codes (those of `\c_str_cctab`) would be imposed. Note that the inner `\cctab_end:` restores the correct catcodes only locally, so the problem really comes up because of the different grouping level. The simplest is to use a scratch table labeled by the `\currentgrouplevel`. We initialize one of them as an example.

```

\use:n
{
 \cctab_begin:N \c_document_cctab
 \group_begin:
 \cctab_begin:N \c_str_cctab
 \cctab_end:
 \group_end:
 \cctab_end:
}

22655 \sys_if_engine_luatex:T
22656 {
22657 __cctab_new:N \g__cctab_internal_cctab
22658 \cs_new:Npn __cctab_internal_cctab_name:
22659 {
22660 g__cctab_internal
22661 \tex_roman numeral:D \tex_currentgrouplevel:D
22662 _cctab
22663 }
22664 }

(End definition for \g__cctab_internal_cctab and __cctab_internal_cctab_name:.)
```

`\cctab_select:N` The public function simply checks the `\cctab var` exists before using the engine-dependent `\__cctab_select:N`. Skipping these checks would result in low-level engine-dependent errors. First, the LuaTeX case. In other engines, selecting a catcode table is a matter of doing 256 catcode assignments and setting the `\endlinechar`.

```

22665 \cs_new_protected:Npn \cctab_select:N #1
22666 { __cctab_chk_if_valid:NT #1 { __cctab_select:N #1 } }
22667 \cs_generate_variant:Nn \cctab_select:N { c }
22668 \sys_if_engine_luatex:TF
22669 {
22670 \cs_new_protected:Npn __cctab_select:N #1
22671 {
22672 \tex_catcodetable:D #1
22673 \prop_get:NVNTF \g__cctab_endlinechar_prop #1 \l__cctab_internal_a_tl
22674 { \int_set:Nn \tex_endlinechar:D { \l__cctab_internal_a_tl } }
22675 { \int_set:Nn \tex_endlinechar:D { 13 } }
22676 \cs_if_exist:cF { __cctab_internal_cctab_name: }
22677 { \exp_args:Nc __cctab_new:N { __cctab_internal_cctab_name: } }
22678 \exp_args:Nc \tex_savecatcodetable:D { __cctab_internal_cctab_name: }
22679 \exp_args:Nc \tex_catcodetable:D { __cctab_internal_cctab_name: }
22680 }
22681 }
22682 }
```

```

22683 \cs_new_protected:Npn __cctab_select:N #1
22684 {
22685 \int_step_inline:nn { 256 }
22686 {
22687 \char_set_catcode:nn { ##1 - 1 }
22688 { __kernel_intarray_item:Nn #1 {##1} }
22689 }
22690 \int_set:Nn \tex_endlinechar:D
22691 { __kernel_intarray_item:Nn #1 { 257 } }
22692 }
22693 }

```

(End definition for `\cctab_select:N` and `\__cctab_select:N`. This function is documented on page 222.)

`\g__cctab_next_cctab` For `\cctab_begin:N/\cctab_end:` we will need to allocate dynamic tables. This is done here by `\__cctab_begin_aux:`, which puts a table number (in LuaTeX) or name (in other engines) into `\l__cctab_internal_a_tl`. In LuaTeX this simply calls `\__cctab_new:N` and uses the resulting catcodetable number; in other engines we need to give a name to the intarray variable and use that. In LuaTeX, to restore catcodes at `\cctab_end:` we cannot just set `\catcodetable` to its value before `\cctab_begin:N`, because that table may have been altered by other code in the mean time. So we must make sure to save the catcodes in a table we control and restore them at `\cctab_end::`.

```

22694 \sys_if_engine_luatex:TF
22695 {
22696 \cs_new_protected:Npn __cctab_begin_aux:
22697 {
22698 __cctab_new:N \g__cctab_next_cctab
22699 \tl_set:NV \l__cctab_internal_a_tl \g__cctab_next_cctab
22700 \cs_undefine:N \g__cctab_next_cctab
22701 }
22702 }
22703 {
22704 \cs_new_protected:Npn __cctab_begin_aux:
22705 {
22706 \int_gincr:N \g__cctab_allocate_int
22707 \exp_args:Nc __cctab_new:
22708 { g__cctab_ \int_use:N \g__cctab_allocate_int _cctab }
22709 \exp_args:NNc \tl_set:Nn \l__cctab_internal_a_tl
22710 { g__cctab_ \int_use:N \g__cctab_allocate_int _cctab }
22711 }
22712 }

```

(End definition for `\g__cctab_next_cctab` and `\__cctab_begin_aux::`)

`\cctab_begin:N` Check the `\langle ctab var \rangle` exists, to avoid low-level errors. Get in `\l__cctab_internal_a_tl` the number/name of a dynamic table, either from `\g__cctab_unused_seq` where we save tables that are not currently in use, or from `\__cctab_begin_aux:` if none are available. Then save the current catcodes into the table (pointed to by) `\l__cctab_internal_a_tl` and save that table number in a stack before selecting the desired catcodes.

```

22713 \cs_new_protected:Npn \cctab_begin:N #1
22714 {
22715 __cctab_chk_if_valid:NT #1
22716 {

```

```

22717 \seq_gpop:NNF \g__cctab_unused_seq \l__cctab_internal_a_tl
22718 { __cctab_begin_aux: }
22719 \exp_args:Nx __cctab_chk_group_begin:n
22720 { __cctab_nesting_number:N \l__cctab_internal_a_tl }
22721 \seq_gpush:NV \g__cctab_stack_seq \l__cctab_internal_a_tl
22722 \exp_args:NV __cctab_gset:n \l__cctab_internal_a_tl
22723 __cctab_select:N #1
22724 }
22725 }
22726 \cs_generate_variant:Nn \cctab_begin:N { c }

```

(End definition for `\cctab_begin:N`. This function is documented on page 222.)

**\cctab\_end:** Make sure a `\cctab_begin:N` was used some time earlier, get in `\l__cctab_internal_a_tl` the catcode table number/name in which the prevailing catcodes were stored, then restore these catcodes. The dynamic table is now unused hence stored in `\g__cctab_unused_seq` for recycling by later `\cctab_begin:N`.

```

22727 \cs_new_protected:Npn \cctab_end:
22728 {
22729 \seq_gpop:NNTF \g__cctab_stack_seq \l__cctab_internal_a_tl
22730 {
22731 \seq_gpush:NV \g__cctab_unused_seq \l__cctab_internal_a_tl
22732 \exp_args:Nx __cctab_chk_group_end:n
22733 { __cctab_nesting_number:N \l__cctab_internal_a_tl }
22734 __cctab_select:N \l__cctab_internal_a_tl
22735 }
22736 { __kernel_msg_error:nn { kernel } { ctab-extra-end } }
22737 }

```

(End definition for `\cctab_end:`. This function is documented on page 222.)

`\__cctab_chk_group_begin:n` `\__cctab_chk_group_end:n`: Catcode tables are not allowed to be intermixed with groups, so here we check that they are properly nested regarding TeX groups. `\__cctab_chk_group_begin:n` stores the current group level in a stack, and locally defines a dummy control sequence `\__cctab_group_<ctab-level>_chk:`.

`\__cctab_chk_group_end:n` pops the stack, and compares the returned value with `\tex_currentgrouplevel:D`. If they differ, `\cctab_end:` is in a different grouping level than the matching `\cctab_begin:N`. If they are the same, both happened at the same level, however a group might have ended and another started between `\cctab_begin:N` and `\cctab_end::`:

```

\group_begin:
\cctab_begin:N \c_document_cctab
\group_end:
\group_begin:
\cctab_end:
\group_end:

```

In this case checking `\tex_currentgrouplevel:D` is not enough, so we locally define `\__cctab_group_<ctab-level>_chk:`, and then check if it exist in `\cctab_end::`. If it doesn't, we know there was a group end where it shouldn't.

The `<ctab-level>` in the sentinel macro above cannot be replaced by the more convenient `\tex_currentgrouplevel:D` because with the latter we might be tricked. Suppose:

```

\group_begin:
 \cctab_begin:N \c_code_cctab % A
\group_end:
\group_begin:
 \cctab_begin:N \c_code_cctab % B
 \cctab_end: % C
 \cctab_end: % D
\group_end:

```

The line marked with A would start a `cctab` with a sentinel token named `\__cctab_group_1_chk:`, which would disappear at the `\group_end:` that follows. But B would create the same sentinel token, since both are at the same group level. Line C would end the `cctab` from line B correctly, but so would line D because line B created the same sentinel token. Using `<cctab-level>` works correctly because it signals that certain `cctab` level was activated somewhere, but if it doesn't exist when the `\cctab_end:` is reached, we had a problem.

Unfortunately these tests only flag the wrong usage at the `\cctab_end:`, which might be far from the `\cctab_begin:N`. However it isn't possible to signal the wrong usage at the `\group_end:` without using `\tex_aftergroup:D`, which is unsafe in certain types of groups.

The three cases checked here just raise an error, and no recovery is attempted: usually interleaving groups and catcode tables will work predictably.

```

22738 \cs_new_protected:Npn __cctab_chk_group_begin:n #1
22739 {
22740 \seq_gpush:Nx \g__cctab_group_seq
22741 { \int_use:N \tex_currentgrouplevel:D }
22742 \cs_set_eq:cN { __cctab_group_ #1 _chk: } \prg_do_nothing:
22743 }
22744 \cs_new_protected:Npn __cctab_chk_group_end:n #1
22745 {
22746 \seq_gpop>NN \g__cctab_group_seq \l__cctab_internal_b_tl
22747 \bool_lazy_and:nnF
22748 {
22749 \int_compare_p:nNn
22750 { \tex_currentgrouplevel:D } = { \l__cctab_internal_b_tl }
22751 }
22752 { \cs_if_exist_p:c { __cctab_group_ #1 _chk: } }
22753 {
22754 __kernel_msg_error:nnx { kernel } { cctab-group-mismatch }
22755 {
22756 \int_sign:n
22757 { \tex_currentgrouplevel:D - \l__cctab_internal_b_tl }
22758 }
22759 }
22760 \cs_undefine:c { __cctab_group_ #1 _chk: }
22761 }

```

(End definition for `\__cctab_chk_group_begin:n` and `\__cctab_chk_group_end:n`.)

`\__cctab_nesting_number:N` This macro returns the numeric index of the current catcode table. In `LuaTeX` this is just the argument, which is a count reference to a `\catcodetable` register. In other engines, the number is extracted from the `cctab` variable.

```

22762 \sys_if_engine_luatex:TF
22763 { \cs_new:Npn __cctab_nesting_number:N #1 {#1} }
22764 {
22765 \cs_new:Npn __cctab_nesting_number:N #1
22766 {
22767 \exp_after:wN \exp_after:wN \exp_after:wN __cctab_nesting_number:w
22768 \exp_after:wN \token_to_str:N #1
22769 }
22770 \use:x
22771 {
22772 \cs_new:Npn \exp_not:N __cctab_nesting_number:w
22773 ##1 \tl_to_str:n { g__cctab_ } ##2 \tl_to_str:n { _cctab } {##2}
22774 }
22775 }

```

(End definition for `\__cctab_nesting_number:N` and `\__cctab_nesting_number:w`.)

Finally, install some code at the end of the TeX run to check that all `\cctab_begin:N` were ended by some `\cctab_end::`:

```

22776 \cs_if_exist:NT \hook_gput_code:nnn
22777 {
22778 \hook_gput_code:nnn { enddocument/end } { kernel }
22779 {
2280 \seq_if_empty:NF \g__cctab_stack_seq
2281 { __kernel_msg_error:nn { kernel } { cctab-missing-end } }
2282 }
2283 }

```

### 38.5 Category code table conditionals

`\cctab_if_exist:N` Checks whether a `\cctab var` is defined.

```

\cctab_if_exist:c
22784 \prg_new_eq_conditional:NNn \cctab_if_exist:N \cs_if_exist:N
22785 { TF , T , F , p }
22786 \prg_new_eq_conditional:NNn \cctab_if_exist:c \cs_if_exist:c
22787 { TF , T , F , p }

```

(End definition for `\cctab_if_exist:N`. This function is documented on page ??.)

`\__cctab_chk_if_valid:NTF` Checks whether the argument is defined and whether it is a valid `\cctab var`. In LuaTeX the validity of the `\cctab var` is checked by the engine, which complains if the argument is not a `\chardef`ed` constant. In other engines, check if the given command is an intarray variable (the underlying definition is a copy of the `cmt10` font).

```

22788 \prg_new_protected_conditional:Npnn __cctab_chk_if_valid:N #1
22789 { TF , T , F }
22790 {
22791 \cctab_if_exist:NTF #1
22792 {
22793 __cctab_chk_if_valid_aux:NTF #1
22794 { \prg_return_true: }
22795 {
22796 __kernel_msg_error:nnx { kernel } { invalid-cctab }
22797 { \token_to_str:N #1 }
22798 \prg_return_false:
22799 }

```

```

22800 }
22801 {
22802 __kernel_msg_error:nnx { kernel } { command-not-defined }
22803 { \token_to_str:N #1 }
22804 \prg_return_false:
22805 }
22806 }
22807 \sys_if_engine_luatex:TF
22808 {
22809 \cs_new_protected:Npn __cctab_chk_if_valid_aux:NTF #1
22810 {
22811 \int_compare:nNnTF {#1-1} < { \e@alloc@ccodetable@count }
22812 }
22813 \cs_if_exist:NT \c_syst_catcodes_n
22814 {
22815 \cs_gset_protected:Npn __cctab_chk_if_valid_aux:NTF #1
22816 {
22817 \int_compare:nTF { #1 <= \c_syst_catcodes_n }
22818 }
22819 }
22820 }
22821 {
22822 \cs_new_protected:Npn __cctab_chk_if_valid_aux:NTF #1
22823 {
22824 \exp_args:Nf \str_if_in:nTF
22825 { \cs_meaning:N #1 }
22826 { select~font~cmr10~at~ }
22827 }
22828 }

```

(End definition for `\__cctab_chk_if_valid:NTF` and `\__cctab_chk_if_valid_aux:NTF`.)

## 38.6 Constant category code tables

`\cctab_const:Nn` Creates a new  $\langle cctab var \rangle$  then sets it with the current and user-supplied codes.

```

22829 \cs_new_protected:Npn \cctab_const:Nn #1#2
2290 {
2291 \cctab_new:N #1
2292 \cctab_gset:Nn #1 {#2}
2293 }
2294 \cs_generate_variant:Nn \cctab_const:Nn { c }

```

(End definition for `\cctab_const:Nn`. This function is documented on page 222.)

`\c_initex_cctab` Creating category code tables means thinking starting from iniTeX. For all-other and the standard “string” tables that’s easy.

`\c_other_cctab`

`\c_str_cctab`

```

22835 \cctab_new:N \c_initex_cctab
22836 \cctab_const:Nn \c_other_cctab
22837 {
22838 \cctab_select:N \c_initex_cctab
22839 \int_set:Nn \tex_endlinechar:D { -1 }
22840 \int_step_inline:nnn { 0 } { 127 }
22841 { \char_set_catcode_other:n {#1} }
22842 }

```

```

22843 \cctab_const:Nn \c_str_cctab
22844 {
22845 \cctab_select:N \c_other_cctab
22846 \char_set_catcode_space:n { 32 }
22847 }

```

(End definition for `\c_initex_cctab`, `\c_other_cctab`, and `\c_str_cctab`. These variables are documented on page 223.)

### `\c_code_cctab` `\c_document_cctab`

To pick up document-level category codes, we need to delay set up to the end of the format, where that's possible. Also, as there are a *lot* of category codes to set, we avoid using the official interface and store the document codes using internal code. Depending on whether we are in the hook or not, the catcodes may be code or document, so we explicitly set up both correctly.

```

22848 \cs_if_exist:NTF \expl@finalise@setup@@
22849 { \tl_gput_right:Nn \expl@finalise@setup@@ }
22850 { \use:n }
22851 {
22852 __cctab_new:N \c_code_cctab
22853 \group_begin:
22854 \int_set:Nn \tex_endlinechar:D { 32 }
22855 \char_set_catcode_invalid:n { 0 }
22856 \bool_lazy_or:nnTF
22857 { \sys_if_engine_xetex_p: } { \sys_if_engine_luatex_p: }
22858 { \int_step_function:nN { 31 } \char_set_catcode_invalid:n }
22859 { \int_step_function:nN { 31 } \char_set_catcode_active:n }
22860 \int_step_function:nnN { 33 } { 64 } \char_set_catcode_other:n
22861 \int_step_function:nnN { 65 } { 90 } \char_set_catcode_letter:n
22862 \int_step_function:nnN { 91 } { 96 } \char_set_catcode_other:n
22863 \int_step_function:nnN { 97 } { 122 } \char_set_catcode_letter:n
22864 \char_set_catcode_ignore:n { 9 } % tab
22865 \char_set_catcode_other:n { 10 } % lf
22866 \char_set_catcode_active:n { 12 } % ff
22867 \char_set_catcode_end_line:n { 13 } % cr
22868 \char_set_catcode_ignore:n { 32 } % space
22869 \char_set_catcode_parameter:n { 35 } % hash
22870 \char_set_catcode_math_toggle:n { 36 } % dollar
22871 \char_set_catcode_comment:n { 37 } % percent
22872 \char_set_catcode_alignment:n { 38 } % ampersand
22873 \char_set_catcode_letter:n { 58 } % colon
22874 \char_set_catcode_escape:n { 92 } % backslash
22875 \char_set_catcode_math_superscript:n { 94 } % circumflex
22876 \char_set_catcode_letter:n { 95 } % underscore
22877 \char_set_catcode_group_begin:n { 123 } % left brace
22878 \char_set_catcode_other:n { 124 } % pipe
22879 \char_set_catcode_group_end:n { 125 } % right brace
22880 \char_set_catcode_space:n { 126 } % tilde
22881 \char_set_catcode_invalid:n { 127 } % ^^?
22882 \bool_lazy_or:nnF
22883 { \sys_if_engine_xetex_p: } { \sys_if_engine_luatex_p: }
22884 { \int_step_function:nnN { 128 } { 255 } \char_set_catcode_active:n }
22885 __cctab_gset:n { \c_code_cctab }
22886 \group_end:
22887 \cctab_const:Nn \c_document_cctab

```

```

22888 {
22889 \cctab_select:N \c_code_cctab
22890 \int_set:Nn \tex_endlinechar:D { 13 }
22891 \char_set_catcode_space:n { 9 }
22892 \char_set_catcode_space:n { 32 }
22893 \char_set_catcode_other:n { 58 }
22894 \char_set_catcode_math_subscript:n { 95 }
22895 \char_set_catcode_active:n { 126 }
22896 }
22897 }

```

(End definition for `\c_code_cctab` and `\c_document_cctab`. These variables are documented on page 223.)

## 38.7 Messages

```

22898 __kernel_msg_new:nnnn { kernel } { cctab-stack-full }
22899 { The-category-code-table-stack-is-exhausted. }
22900 {
22901 LaTeX-has-been-asked-to-switch-to-a-new-category-code-table,-
22902 but-there-is-no-more-space-to-do-this!
22903 }
22904 __kernel_msg_new:nnnn { kernel } { cctab-extra-end }
22905 { Extra-\iow_char:N\\cctab_end:-ignored-\msg_line_context:.. }
22906 {
22907 LaTeX-came-across-a-\iow_char:N\\cctab_end:-without-a-matching-
22908 \iow_char:N\\cctab_begin:N.-This-command-will-be-ignored.
22909 }
22910 __kernel_msg_new:nnnn { kernel } { cctab-missing-end }
22911 { Missing-\iow_char:N\\cctab_end:-before-end-of-TeX-run. }
22912 {
22913 LaTeX-came-across-more-\iow_char:N\\cctab_begin:N-than-
22914 \iow_char:N\\cctab_end:.
22915 }
22916 __kernel_msg_new:nnnn { kernel } { invalid-cctab }
22917 { Invalid-\iow_char:N\\catcode-table. }
22918 {
22919 You-can-only-switch-to-a-\iow_char:N\\catcode-table-that-is-
22920 initialized-using-\iow_char:N\\cctab_new:N-or-
22921 \iow_char:N\\cctab_const:Nn.
22922 }
22923 __kernel_msg_new:nnnn { kernel } { cctab-group-mismatch }
22924 {
22925 \iow_char:N\\cctab_end:-occurred-in-a-
22926 \int_case:nn {#1}
22927 {
22928 { 0 } { different-group }
22929 { 1 } { higher-group-level }
22930 { -1 } { lower-group-level }
22931 } ~than~
22932 the-matching-\iow_char:N\\cctab_begin:N.
22933 }
22934 {
22935 Catcode-tables-and-groups-must-be-properly-nested,-but-

```

```

22936 you~tried~to~interleave~them.~LaTeX~will~try~to~proceed,~
22937 but~results~may~be~unexpected.
22938 }
22939 </package>

```

## 39 I3sort implementation

```

22940 {*package}
22941 <@@=sort>

```

### 39.1 Variables

\g\_\_sort\_internal\_seq  
\g\_\_sort\_internal\_tl

Sorting happens in a group; the result is stored in those global variables before being copied outside the group to the proper places. For seq and tl this is more efficient than using \use:x (or some \exp\_args:NNNx) to smuggle the definition outside the group since TeX does not need to re-read tokens. Forclist we don't gain anything since the result is converted from seq to list anyways.

```

22942 \seq_new:N \g__sort_internal_seq
22943 \tl_new:N \g__sort_internal_tl

```

(End definition for \g\_\_sort\_internal\_seq and \g\_\_sort\_internal\_tl.)

\l\_\_sort\_length\_int  
\l\_\_sort\_min\_int  
\l\_\_sort\_top\_int  
\l\_\_sort\_max\_int  
\l\_\_sort\_true\_max\_int

The sequence has \l\_\_sort\_length\_int items and is stored from \l\_\_sort\_min\_int to \l\_\_sort\_top\_int - 1. While reading the sequence in memory, we check that \l\_\_sort\_top\_int remains at most \l\_\_sort\_max\_int, precomputed by \l\_\_sort\_compute\_range:. That bound is such that the merge sort only uses \toks registers less than \l\_\_sort\_true\_max\_int, namely those that have not been allocated for use in other code: the user's comparison code could alter these.

```

22944 \int_new:N \l__sort_length_int
22945 \int_new:N \l__sort_min_int
22946 \int_new:N \l__sort_top_int
22947 \int_new:N \l__sort_max_int
22948 \int_new:N \l__sort_true_max_int

```

(End definition for \l\_\_sort\_length\_int and others.)

\l\_\_sort\_block\_int

Merge sort is done in several passes. In each pass, blocks of size \l\_\_sort\_block\_int are merged in pairs. The block size starts at 1, and, for a length in the range  $[2^k + 1, 2^{k+1}]$ , reaches  $2^k$  in the last pass.

```
22949 \int_new:N \l__sort_block_int
```

(End definition for \l\_\_sort\_block\_int.)

\l\_\_sort\_begin\_int  
\l\_\_sort\_end\_int

When merging two blocks, \l\_\_sort\_begin\_int marks the lowest index in the two blocks, and \l\_\_sort\_end\_int marks the highest index, plus 1.

```

22950 \int_new:N \l__sort_begin_int
22951 \int_new:N \l__sort_end_int

```

(End definition for \l\_\_sort\_begin\_int and \l\_\_sort\_end\_int.)

`\l__sort_A_int` When merging two blocks (whose end-points are `beg` and `end`),  $A$  starts from the high end of the low block, and decreases until reaching `beg`. The index  $B$  starts from the top of the range and marks the register in which a sorted item should be put. Finally,  $C$  points to the copy of the high block in the interval of registers starting at `\l__sort_length_int`, upwards.  $C$  starts from the upper limit of that range.

```

22952 \int_new:N \l__sort_A_int
22953 \int_new:N \l__sort_B_int
22954 \int_new:N \l__sort_C_int

```

*(End definition for `\l__sort_A_int`, `\l__sort_B_int`, and `\l__sort_C_int`.)*

`\s__sort_mark` Internal scan marks.

```

22955 \scan_new:N \s__sort_mark
22956 \scan_new:N \s__sort_stop

```

*(End definition for `\s__sort_mark` and `\s__sort_stop`.)*

## 39.2 Finding available `\toks` registers

`\__sort_shrink_range:` After `\__sort_compute_range:` (defined below) determines that `\toks` registers between `\l__sort_min_int` (included) and `\l__sort_true_max_int` (excluded) have not yet been assigned, `\__sort_shrink_range:` computes `\l__sort_max_int` to reflect the need for a buffer when merging blocks in the merge sort. Given  $2^n \leq A \leq 2^n + 2^{n-1}$  registers we can sort  $\lfloor A/2 \rfloor + 2^{n-2}$  items while if we have  $2^n + 2^{n-1} \leq A \leq 2^{n+1}$  registers we can sort  $A - 2^{n-1}$  items. We first find out a power  $2^n$  such that  $2^n \leq A \leq 2^{n+1}$  by repeatedly halving `\l__sort_block_int`, starting at  $2^{15}$  or  $2^{14}$  namely half the total number of registers, then we use the formulas and set `\l__sort_max_int`.

```

22957 \cs_new_protected:Npn __sort_shrink_range:
22958 {
22959 \int_set:Nn \l__sort_A_int
22960 { \l__sort_true_max_int - \l__sort_min_int + 1 }
22961 \int_set:Nn \l__sort_block_int { \c_max_register_int / 2 }
22962 __sort_shrink_range_loop:
22963 \int_set:Nn \l__sort_max_int
22964 {
22965 \int_compare:nNnTF
22966 { \l__sort_block_int * 3 / 2 } > \l__sort_A_int
22967 {
22968 \l__sort_min_int
22969 + (\l__sort_A_int - 1) / 2
22970 + \l__sort_block_int / 4
22971 - 1
22972 }
22973 { \l__sort_true_max_int - \l__sort_block_int / 2 }
22974 }
22975 }
22976 \cs_new_protected:Npn __sort_shrink_range_loop:
22977 {
22978 \if_int_compare:w \l__sort_A_int < \l__sort_block_int
22979 \tex_divide:D \l__sort_block_int 2 \exp_stop_f:
22980 \exp_after:wN __sort_shrink_range_loop:
22981 \fi:
22982 }

```

(End definition for `\__sort_shrink_range:` and `\__sort_shrink_range_loop::`)

```
__sort_compute_range:
__sort_redefine_compute_range:
\c__sort_max_length_int
```

First find out what `\toks` have not yet been assigned. There are many cases. In L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  with no package, available `\toks` range from `\count15 + 1` to `\c_max_register_int` included (this was not altered despite the 2015 changes). When `\loctoks` is defined, namely in plain (e)T<sub>E</sub>X, or when the package etex is loaded in L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub> , redefine `\__sort_compute_range:` to use the range `\count265` to `\count275 - 1`. The elalloc package also defines `\loctoks` but uses yet another number for the upper bound, namely `\e@alloc@top` (minus one). We must check for `\loctoks` every time a sorting function is called, as etex or elalloc could be loaded.

In ConT<sub>E</sub>Xt MkIV the range is from `\c_syst_last_allocated_toks+1` to `\c_max_register_int`, and in MkII it is from `\lastallocatedtoks+1` to `\c_max_register_int`. In all these cases, call `\__sort_shrink_range::`.

```
22983 \cs_new_protected:Npn __sort_compute_range:
22984 {
22985 \int_set:Nn \l__sort_min_int { \tex_count:D 15 + 1 }
22986 \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
22987 __sort_shrink_range:
22988 \if_meaning:w \loctoks \tex_undefined:D \else:
22989 \if_meaning:w \loctoks \scan_stop: \else:
22990 __sort_redefine_compute_range:
22991 __sort_compute_range:
22992 \fi:
22993 \fi:
22994 }
22995 \cs_new_protected:Npn __sort_redefine_compute_range:
22996 {
22997 \cs_if_exist:cTF { ver@elalloc.sty }
22998 {
22999 \cs_gset_protected:Npn __sort_compute_range:
23000 {
23001 \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
23002 \int_set_eq:NN \l__sort_true_max_int \e@alloc@top
23003 __sort_shrink_range:
23004 }
23005 }
23006 {
23007 \cs_gset_protected:Npn __sort_compute_range:
23008 {
23009 \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
23010 \int_set:Nn \l__sort_true_max_int { \tex_count:D 275 }
23011 __sort_shrink_range:
23012 }
23013 }
23014 }
23015 \cs_if_exist:NT \loctoks { __sort_redefine_compute_range: }
23016 \tl_map_inline:nn { \lastallocatedtoks \c_syst_last_allocated_toks }
23017 {
23018 \cs_if_exist:NT #1
23019 {
23020 \cs_gset_protected:Npn __sort_compute_range:
23021 {
23022 \int_set:Nn \l__sort_min_int { #1 + 1 }
```

```

23023 \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
23024 __sort_shrink_range:
23025 }
23026 }
23027 }

(End definition for __sort_compute_range:, __sort_redefine_compute_range:, and \c__sort_max-
length_int.)
```

### 39.3 Protected user commands

\\_\_sort\_main:NNNn Sorting happens in three steps. First store items in \toks registers ranging from \l\_\_sort\_min\_int to \l\_\_sort\_top\_int - 1, while checking that the list is not too long. If we reach the maximum length, that's an error; exit the group. Secondly, sort the array of \toks registers, using the user-defined sorting function: \\_\_sort\_level: calls \\_\_sort\_compare:nn as needed. Finally, unpack the \toks registers (now sorted) into the target tl, or into \g\_\_sort\_internal\_seq for seq andclist. This is done by \\_\_sort\_seq:NNNNn and \\_\_sort\_tl:NNn.

```

23028 \cs_new_protected:Npn __sort_main:NNNn #1#2#3#4
23029 {
23030 __sort_disable_toksdef:
23031 __sort_compute_range:
23032 \int_set_eq:NN \l__sort_top_int \l__sort_min_int
23033 #1 #3
23034 {
23035 \if_int_compare:w \l__sort_top_int = \l__sort_max_int
23036 __sort_too_long_error>NNw #2 #3
23037 \fi:
23038 \tex_toks:D \l__sort_top_int {##1}
23039 \int_incr:N \l__sort_top_int
23040 }
23041 \int_set:Nn \l__sort_length_int
23042 { \l__sort_top_int - \l__sort_min_int }
23043 \cs_set:Npn __sort_compare:nn ##1 ##2 {#4}
23044 \int_set:Nn \l__sort_block_int { 1 }
23045 __sort_level:
23046 }

(End definition for __sort_main:NNNn.)
```

\tl\_sort:Nn Call the main sorting function then unpack \toks registers outside the group into the target token list. The unpacking is done by \\_\_sort\_tl\_toks:w; registers are numbered from \l\_\_sort\_min\_int to \l\_\_sort\_top\_int - 1. For expansion behaviour we need a couple of primitives. The \tl\_gclear:N reduces memory usage. The \prg\_break\_point: is used by \\_\_sort\_main:NNNn when the list is too long.

```

23047 \cs_new_protected:Npn \tl_sort:Nn { __sort_tl:NNn \tl_set_eq:NN }
23048 \cs_generate_variant:Nn \tl_sort:Nn { c }
23049 \cs_new_protected:Npn \tl_gsort:Nn { __sort_tl:NNn \tl_gset_eq:NN }
23050 \cs_generate_variant:Nn \tl_gsort:Nn { c }
23051 \cs_new_protected:Npn __sort_tl:NNn #1#2#3
23052 {
23053 \group_begin:
23054 __sort_main:NNNn \tl_map_inline:Nn \tl_map_break:n #2 {#3}
```

```

23055 __kernel_tl_gset:Nx \g__sort_internal_tl
23056 { __sort_tl_toks:w \l__sort_min_int ; }
23057 \group_end:
23058 #1 #2 \g__sort_internal_tl
23059 \tl_gclear:N \g__sort_internal_tl
23060 \prg_break_point:
23061 }
23062 \cs_new:Npn __sort_tl_toks:w #1 ;
23063 {
23064 \if_int_compare:w #1 < \l__sort_top_int
23065 { \tex_the:D \tex_toks:D #1 }
23066 \exp_after:wN __sort_tl_toks:w
23067 \int_value:w \int_eval:n { #1 + 1 } \exp_after:wN ;
23068 \fi:
23069 }

```

(End definition for `\tl_sort:Nn` and others. These functions are documented on page 53.)

`\seq_sort:Nn` Use the same general framework for seq andclist. Apply the general sorting code, then unpack `\toks` into `\g__sort_internal_seq`. Outside the group copy or convert (forclist) the data to the target variable. The `\seq_gclear:N` reduces memory usage. The `\prg_break_point:` is used by `\__sort_main:NNNn` when the list is too long.

`\seq_sort:cn`

`\seq_gsort:Nn`

`\seq_gsort:cn`

`\clist_sort:Nn`

`\clist_sort:cn`

`\clist_gsort:Nn`

`\clist_gsort:cn`

`\__sort_seq:NNNNn`

```

23070 \cs_new_protected:Npn \seq_sort:Nn
23071 { __sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_set_eq:NN }
23072 \cs_generate_variant:Nn \seq_sort:Nn { c }
23073 \cs_new_protected:Npn \seq_gsort:Nn
23074 { __sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_gset_eq:NN }
23075 \cs_generate_variant:Nn \seq_gsort:Nn { c }
23076 \cs_new_protected:Npn \clist_sort:Nn
23077 {
23078 __sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
23079 \clist_set_from_seq:NN
23080 }
23081 \cs_generate_variant:Nn \clist_sort:Nn { c }
23082 \cs_new_protected:Npn \clist_gsort:Nn
23083 {
23084 __sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
23085 \clist_gset_from_seq:NN
23086 }
23087 \cs_generate_variant:Nn \clist_gsort:Nn { c }
23088 \cs_new_protected:Npn __sort_seq:NNNNn #1#2#3#4#5
23089 {
23090 \group_begin:
23091 __sort_main:NNNn #1 #2 #4 {#5}
23092 \seq_gset_from_inline_x:Nnn \g__sort_internal_seq
23093 {
23094 \int_step_function:nnN
23095 { \l__sort_min_int } { \l__sort_top_int - 1 }
23096 }
23097 { \tex_the:D \tex_toks:D ##1 }
23098 \group_end:
23099 #3 #4 \g__sort_internal_seq
23100 \seq_gclear:N \g__sort_internal_seq
23101 \prg_break_point:

```

```
23102 }
```

(End definition for `\seq_sort:Nn` and others. These functions are documented on page 79.)

## 39.4 Merge sort

`\__sort_level:` This function is called once blocks of size `\l__sort_block_int` (initially 1) are each sorted. If the whole list fits in one block, then we are done (this also takes care of the case of an empty list or a list with one item). Otherwise, go through pairs of blocks starting from 0, then double the block size, and repeat.

```
23103 \cs_new_protected:Npn __sort_level:
23104 {
23105 \if_int_compare:w \l__sort_block_int < \l__sort_length_int
23106 \l__sort_end_int \l__sort_min_int
23107 __sort_merge_blocks:
23108 \tex_advance:D \l__sort_block_int \l__sort_block_int
23109 \exp_after:wN __sort_level:
23110 \fi:
23111 }
```

(End definition for `\__sort_level:..`)

`\__sort_merge_blocks:` This function is called to merge a pair of blocks, starting at the last value of `\l__sort_end_int` (end-point of the previous pair of blocks). If shifting by one block to the right we reach the end of the list, then this pass has ended: the end of the list is sorted already. Otherwise, store the result of that shift in `A`, which indexes the first block starting from the top end. Then locate the end-point (maximum) of the second block: shift `end` upwards by one more block, but keeping it  $\leq \text{top}$ . Copy this upper block of `\toks` registers in registers above `length`, indexed by `C`: this is covered by `\__sort_copy_block:..`. Once this is done we are ready to do the actual merger using `\__sort_merge_blocks_aux:..`, after shifting `A`, `B` and `C` so that they point to the largest index in their respective ranges rather than pointing just beyond those ranges. Of course, once that pair of blocks is merged, move on to the next pair.

```
23112 \cs_new_protected:Npn __sort_merge_blocks:
23113 {
23114 \l__sort_begin_int \l__sort_end_int
23115 \tex_advance:D \l__sort_end_int \l__sort_block_int
23116 \if_int_compare:w \l__sort_end_int < \l__sort_top_int
23117 \l__sort_A_int \l__sort_end_int
23118 \tex_advance:D \l__sort_end_int \l__sort_block_int
23119 \if_int_compare:w \l__sort_end_int > \l__sort_top_int
23120 \l__sort_end_int \l__sort_top_int
23121 \fi:
23122 \l__sort_B_int \l__sort_A_int
23123 \l__sort_C_int \l__sort_top_int
23124 __sort_copy_block:
23125 \int_decr:N \l__sort_A_int
23126 \int_decr:N \l__sort_B_int
23127 \int_decr:N \l__sort_C_int
23128 \exp_after:wN __sort_merge_blocks_aux:
23129 \exp_after:wN __sort_merge_blocks:
23130 \fi:
23131 }
```

(End definition for `\_\_sort\_merge\_blocks`.)

`\_\_sort\_copy\_block`: We wish to store a copy of the “upper” block of `\toks` registers, ranging between the initial value of `\l\_sort_B_int` (included) and `\l\_sort_end_int` (excluded) into a new range starting at the initial value of `\l\_sort_C_int`, namely `\l\_sort_top_int`.

```
23132 \cs_new_protected:Npn __sort_copy_block:
23133 {
23134 \tex_toks:D \l_sort_C_int \tex_toks:D \l_sort_B_int
23135 \int_incr:N \l_sort_C_int
23136 \int_incr:N \l_sort_B_int
23137 \if_int_compare:w \l_sort_B_int = \l_sort_end_int
23138 \use_i:nn
23139 \fi:
23140 __sort_copy_block:
23141 }
```

(End definition for `\_\_sort_copy_block`.)

`\_\_sort_merge_blocks_aux`: At this stage, the first block starts at `\l\_sort_begin_int`, and ends at `\l\_sort_A_int`, and the second block starts at `\l\_sort_top_int` and ends at `\l\_sort_C_int`. The result of the merger is stored at positions indexed by `\l\_sort_B_int`, which starts at `\l\_sort_end_int - 1` and decreases down to `\l\_sort_begin_int`, covering the full range of the two blocks. In other words, we are building the merger starting with the largest values. The comparison function is defined to return either `swapped` or `same`. Of course, this means the arguments need to be given in the order they appear originally in the list.

```
23142 \cs_new_protected:Npn __sort_merge_blocks_aux:
23143 {
23144 \exp_after:wN __sort_compare:nn \exp_after:wN
23145 { \tex_the:D \tex_toks:D \exp_after:wN \l_sort_A_int \exp_after:wN }
23146 \exp_after:wN { \tex_the:D \tex_toks:D \l_sort_C_int }
23147 \prg_do_nothing:
23148 __sort_return_mark:w
23149 __sort_return_mark:w
23150 \s__sort_mark
23151 __sort_return_none_error:
23152 }
```

(End definition for `\_\_sort_merge_blocks_aux`.)

`\sort_return_same`:  
`\sort_return_swapped`: Each comparison should call `\sort_return_same`: or `\sort_return_swapped`: exactly once. If neither is called, `\_\_sort_return_none_error`: is called, since the `return_mark` removes tokens until `\s__sort_mark`. If one is called, the `return_mark` auxiliary removes everything except `\_\_sort_return_same:w` (or its `swapped` analogue) followed by `\_\_sort_return_none_error`:. Finally if two or more are called, `\_\_sort_return_two_error`: ends up before any `\_\_sort_return_mark:w`, so that it produces an error.

```
23153 \cs_new_protected:Npn \sort_return_same:
23154 #1 __sort_return_mark:w #2 \s__sort_mark
23155 {
23156 #1
23157 #2
23158 __sort_return_two_error:
23159 __sort_return_mark:w
```

```

23160 \s__sort_mark
23161 __sort_return_same:w
23162 }
23163 \cs_new_protected:Npn \sort_return_swapped:
23164 #1 __sort_return_mark:w #2 \s__sort_mark
23165 {
23166 #1
23167 #2
23168 __sort_return_two_error:
23169 __sort_return_mark:w
23170 \s__sort_mark
23171 __sort_return_swapped:w
23172 }
23173 \cs_new_protected:Npn __sort_return_mark:w #1 \s__sort_mark { }
23174 \cs_new_protected:Npn __sort_return_none_error:
23175 {
23176 __kernel_msg_error:nxxx { kernel } { return-none }
23177 { \tex_the:D \tex_toks:D \l__sort_A_int }
23178 { \tex_the:D \tex_toks:D \l__sort_C_int }
23179 __sort_return_same:w __sort_return_none_error:
23180 }
23181 \cs_new_protected:Npn __sort_return_two_error:
23182 {
23183 __kernel_msg_error:nxxx { kernel } { return-two }
23184 { \tex_the:D \tex_toks:D \l__sort_A_int }
23185 { \tex_the:D \tex_toks:D \l__sort_C_int }
23186 }

```

(End definition for `\sort_return_same:` and others. These functions are documented on page 224.)

`\_\_sort_return_same:w` If the comparison function returns `same`, then the second argument fed to `\_\_sort_compare:nn` should remain to the right of the other one. Since we build the merger starting from the right, we copy that `\toks` register into the allotted range, then shift the pointers  $B$  and  $C$ , and go on to do one more step in the merger, unless the second block has been exhausted: then the remainder of the first block is already in the correct registers and we are done with merging those two blocks.

```

23187 \cs_new_protected:Npn __sort_return_same:w #1 __sort_return_none_error:
23188 {
23189 \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
23190 \int_decr:N \l__sort_B_int
23191 \int_decr:N \l__sort_C_int
23192 \if_int_compare:w \l__sort_C_int < \l__sort_top_int
23193 \use_i:nn
23194 \fi:
23195 __sort_merge_blocks_aux:
23196 }

```

(End definition for `\_\_sort_return_same:w`.)

`\_\_sort_return_swapped:w` If the comparison function returns `swapped`, then the next item to add to the merger is the first argument, contents of the `\toks` register  $A$ . Then shift the pointers  $A$  and  $B$  to the left, and go for one more step for the merger, unless the left block was exhausted ( $A$  goes below the threshold). In that case, all remaining `\toks` registers in the second block, indexed by  $C$ , are copied to the merger by `\_\_sort_merge_blocks_end:`

```

23197 \cs_new_protected:Npn __sort_return_swapped:w #1 __sort_return_none_error:
23198 {
23199 \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_A_int
23200 \int_decr:N \l__sort_B_int
23201 \int_decr:N \l__sort_A_int
23202 \if_int_compare:w \l__sort_A_int < \l__sort_begin_int
23203 __sort_merge_blocks_end: \use_i:nn
23204 \fi:
23205 __sort_merge_blocks_aux:
23206 }

```

(End definition for `\__sort_return_swapped:w`.)

`\__sort_merge_blocks_end:`: This function's task is to copy the `\toks` registers in the block indexed by  $C$  to the merger indexed by  $B$ . The end can equally be detected by checking when  $B$  reaches the threshold `begin`, or when  $C$  reaches `top`.

```

23207 \cs_new_protected:Npn __sort_merge_blocks_end:
23208 {
23209 \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
23210 \int_decr:N \l__sort_B_int
23211 \int_decr:N \l__sort_C_int
23212 \if_int_compare:w \l__sort_B_int < \l__sort_begin_int
23213 \use_i:nn
23214 \fi:
23215 __sort_merge_blocks_end:
23216 }

```

(End definition for `\__sort_merge_blocks_end`.)

### 39.5 Expandable sorting

Sorting expandably is very different from sorting and assigning to a variable. Since tokens cannot be stored, they must remain in the input stream, and be read through at every step. It is thus necessarily much slower (at best  $O(n^2 \ln n)$ ) than non-expandable sorting functions ( $O(n \ln n)$ ).

A prototypical version of expandable quicksort is as follows. If the argument has no item, return nothing, otherwise partition, using the first item as a pivot (argument #4 of `\__sort:nnNnn`). The arguments of `\__sort:nnNnn` are 1. items less than #4, 2. items greater or equal to #4, 3. comparison, 4. pivot, 5. next item to test. If #5 is the tail of the list, call `\tl_sort:nN` on #1 and on #2, placing #4 in between; `\use:ff` expands the parts to make `\tl_sort:nN` f-expandable. Otherwise, compare #4 and #5 using #3. If they are ordered, place #5 amongst the “greater” items, otherwise amongst the “lesser” items, and continue partitioning.

```

\cs_new:Npn \tl_sort:nN #1#2
{
 \tl_if_blank:nF {#1}
 {
 __sort:nnNnn { } { } #2
 #1 \q__sort_recursion_tail \q__sort_recursion_stop
 }
}
\cs_new:Npn __sort:nnNnn #1#2#3#4#5

```

```

{
 \quark_if_recursion_tail_stop_do:nn {#5}
 { \use:ff { \tl_sort:nN {#1} #3 {#4} } { \tl_sort:nN {#2} #3 } }
 #3 {#4} {#5}
 { __sort:nnNnn {#1} { #2 {#5} } #3 {#4} }
 { __sort:nnNnn { #1 {#5} } {#2} #3 {#4} }
 }
\cs_generate_variant:Nn \use:nn { ff }

```

There are quite a few optimizations available here: the code below is less legible, but more than twice as fast.

In the simple version of the code, `\__sort:nnNnn` is called  $O(n \ln n)$  times on average (the number of comparisons required by the quicksort algorithm). Hence most of our focus is on optimizing that function.

The first speed up is to avoid testing for the end of the list at every call to `\__sort:nnNnn`. For this, the list is prepared by changing each `\langle item \rangle` of the original token list into `\langle command \rangle \{ \langle item \rangle \}`, just like sequences are stored. We arrange things such that the `\langle command \rangle` is the `\langle conditional \rangle` provided by the user: the loop over the `\langle prepared tokens \rangle` then looks like

```

\cs_new:Npn __sort_loop:wNn ... #6#7
{
 #6 {\langle pivot \rangle} {#7} \langle loop big \rangle \langle loop small \rangle
 \langle extra arguments \rangle
}
__sort_loop:wNn ... \langle prepared tokens \rangle
\end-loop \} \s__sort_stop

```

In this example, which matches the structure of `\__sort_quick_split_i:NnnnnNn` and a few other functions below, the `\__sort_loop:wNn` auxiliary normally receives the user's `\langle conditional \rangle` as #6 and an `\langle item \rangle` as #7. This is compared to the `\langle pivot \rangle` (the argument #5, not shown here), and the `\langle conditional \rangle` leaves the `\langle loop big \rangle` or `\langle loop small \rangle` auxiliary, which both have the same form as `\__sort_loop:wNn`, receiving the next pair `\langle conditional \rangle \{ \langle item \rangle \}` as #6 and #7. At the end, #6 is the `\langle end-loop \rangle` function, which terminates the loop.

The second speed up is to minimize the duplicated tokens between the `true` and `false` branches of the conditional. For this, we introduce two versions of `\__sort:nnNnn`, which receive the new item as #1 and place it either into the list #2 of items less than the pivot #4 or into the list #3 of items greater or equal to the pivot.

```

\cs_new:Npn __sort_i:nnnnNn #1#2#3#4#5#6
{
 #5 {#4} {#6} __sort_ii:nnnnNn __sort_i:nnnnNn
 {#6} { #2 {#1} } {#3} {#4}
}
\cs_new:Npn __sort_ii:nnnnNn #1#2#3#4#5#6
{
 #5 {#4} {#6} __sort_ii:nnnnNn __sort_i:nnnnNn
 {#6} {#2} { #3 {#1} } {#4}
}

```

Note that the two functions have the form of `\__sort_loop:wNn` above, receiving as #5 the conditional or a function to end the loop. In fact, the lists #2 and #3 must be made of pairs  $\langle\text{conditional}\rangle \{\langle\text{item}\rangle\}$ , so we have to replace {#6} above by { #5 {#6} }, and {#1} by #1. The actual functions have one more argument, so all argument numbers are shifted compared to this code.

The third speed up is to avoid `\use:ff` using a continuation-passing style: `\__sort_quick_split:NnNn` expects a list followed by `\s__sort_mark {\langle code\rangle}`, and expands to `\langle code\rangle \langle sorted list\rangle`. Sorting the two parts of the list around the pivot is done with

```
__sort_quick_split:NnNn #2 ... \s__sort_mark
{
 __sort_quick_split:NnNn #1 ... \s__sort_mark {\langle code\rangle}
 {\langle pivot\rangle}
}
```

Items which are larger than the  $\langle pivot\rangle$  are sorted, then placed after code that sorts the smaller items, and after the (braced)  $\langle pivot\rangle$ .

The fourth speed up is avoid the recursive call to `\tl_sort:nN` with an empty first argument. For this, we introduce functions similar to the `\__sort_i:nnnnNn` of the last example, but aware of whether the list of  $\langle\text{conditional}\rangle \{\langle\text{item}\rangle\}$  read so far that are less than the pivot, and the list of those greater or equal, are empty or not: see `\__sort_quick_split:NnNn` and functions defined below. Knowing whether the lists are empty or not is useless if we do not use distinct ending codes as appropriate. The splitting auxiliaries communicate to the  $\langle\text{end-loop}\rangle$  function (that is initially placed after the “prepared” list) by placing a specific ending function, ignored when looping, but useful at the end. In fact, the  $\langle\text{end-loop}\rangle$  function does nothing but place the appropriate ending function in front of all its arguments. The ending functions take care of sorting non-empty sublists, placing the pivot in between, and the continuation before.

The final change in fact slows down the code a little, but is required to avoid memory issues: schematically, when `TEX` encounters

```
\use:n { \use:n { \use:n { ... } ... } ... }
```

the argument of the first `\use:n` is not completely read by the second `\use:n`, hence must remain in memory; then the argument of the second `\use:n` is not completely read when grabbing the argument of the third `\use:n`, hence must remain in memory, and so on. The memory consumption grows quadratically with the number of nested `\use:n`. In practice, this means that we must read everything until a trailing `\s__sort_stop` once in a while, otherwise sorting lists of more than a few thousand items would exhaust a typical `TEX`’s memory.

**\tl\_sort:nN**  
`\__sort_quick_prepare:Nnnn`  
`\__sort_quick_prepare_end:NNNnw`  
`\__sort_quick_cleanup:w`

The code within the `\exp_not:f` sorts the list, leaving in most cases a leading `\exp_not:f`, which stops the expansion, letting the result be return within `\exp_not:n`. We filter out the case of a list with no item, which would otherwise cause problems. Then prepare the token list #1 by inserting the conditional #2 before each item. The `prepare` auxiliary receives the conditional as #1, the prepared token list so far as #2, the next prepared item as #3, and the item after that as #4. The loop ends when #4 contains `\prg_break_point:`, then the `prepare_end` auxiliary finds the prepared token list as #4. The scene is then set up for `\__sort_quick_split:NnNn`, which sorts the prepared list

and perform the post action placed after `\s__sort_mark`, namely removing the trailing `\s__sort_stop` and `\s__sort_stop` and leaving `\exp_stop_f:` to stop f-expansion.

```

23217 \cs_new:Npn \tl_sort:nN #1#2
23218 {
23219 \exp_not:f
23220 {
23221 \tl_if_blank:nF {#1}
23222 {
23223 __sort_quick_prepare:Nnnn #2 { } { }
23224 #1
23225 { \prg_break_point: __sort_quick_prepare_end:NNNnw }
23226 \s__sort_stop
23227 }
23228 }
23229 }
23230 \cs_new:Npn __sort_quick_prepare:Nnnn #1#2#3#4
23231 {
23232 \prg_break: #4 \prg_break_point:
23233 __sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
23234 }
23235 \cs_new:Npn __sort_quick_prepare_end:NNNnw #1#2#3#4#5 \s__sort_stop
23236 {
23237 __sort_quick_split:NnNn #4 __sort_quick_end:nnTFNn { }
23238 \s__sort_mark { __sort_quick_cleanup:w \exp_stop_f: }
23239 \s__sort_mark \s__sort_stop
23240 }
23241 \cs_new:Npn __sort_quick_cleanup:w #1 \s__sort_mark \s__sort_stop {#1}

```

(End definition for `\tl_sort:nN` and others. This function is documented on page 53.)

The `only_i`, `only_ii`, `split_i` and `split_ii` auxiliaries receive a useless first argument, the new item `#2` (that they append to either one of the next two arguments), the list `#3` of items less than the pivot, bigger items `#4`, the pivot `#5`, a `<function>` `#6`, and an item `#7`. The `<function>` is the user's `<conditional>` except at the end of the list where it is `\__sort_quick_end:nnTFNn`. The comparison is applied to the `<pivot>` and the `<item>`, and calls the `only_i` or `split_i` auxiliaries if the `<item>` is smaller, and the `only_ii` or `split_ii` auxiliaries otherwise. In both cases, the next auxiliary goes to work right away, with no intermediate expansion that would slow down operations. Note that the argument `#2` left for the next call has the form `<conditional> {<item>}`, so that the lists `#3` and `#4` keep the right form to be fed to the next sorting function. The `split` auxiliary differs from these in that it is missing three of the arguments, which would be empty, and its first argument is always the user's `<conditional>` rather than an ending function.

```

23242 \cs_new:Npn __sort_quick_split:NnNn #1#2#3#4
23243 {
23244 #3 {#2} {#4} __sort_quick_only_ii:NnnnnNn
23245 __sort_quick_only_i:NnnnnNn
23246 __sort_quick_single_end:nnnwnw
23247 { #3 {#4} } { } { } {#2}
23248 }
23249 \cs_new:Npn __sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
23250 {
23251 #6 {#5} {#7} __sort_quick_split_ii:NnnnnNn
23252 __sort_quick_only_i:NnnnnNn

```

```

23253 __sort_quick_only_i_end:nnnwnw
23254 { #6 {#7} } { #3 #2 } { } {#5}
23255 }
23256 \cs_new:Npn __sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
23257 {
23258 #6 {#5} {#7} __sort_quick_only_ii:NnnnnNn
23259 __sort_quick_split_i:NnnnnNn
23260 __sort_quick_only_ii_end:nnnwnw
23261 { #6 {#7} } { } { #4 #2 } {#5}
23262 }
23263 \cs_new:Npn __sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
23264 {
23265 #6 {#5} {#7} __sort_quick_split_ii:NnnnnNn
23266 __sort_quick_split_i:NnnnnNn
23267 __sort_quick_split_end:nnnwnw
23268 { #6 {#7} } { #3 #2 } {#4} {#5}
23269 }
23270 \cs_new:Npn __sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
23271 {
23272 #6 {#5} {#7} __sort_quick_split_ii:NnnnnNn
23273 __sort_quick_split_i:NnnnnNn
23274 __sort_quick_split_end:nnnwnw
23275 { #6 {#7} } {#3} { #4 #2 } {#5}
23276 }

```

(End definition for `\_\_sort_quick_split:NnNn` and others.)

```
__sort_quick_end:nnTFNn
__sort_quick_single_end:nnnwnw
__sort_quick_only_i_end:nnnwnw
__sort_quick_only_ii_end:nnnwnw
__sort_quick_split_end:nnnwnw
```

The `\_\_sort_quick_end:nnTFNn` appears instead of the user's conditional, and receives as its arguments the pivot #1, a fake item #2, a `true` and a `false` branches #3 and #4, followed by an ending function #5 (one of the four auxiliaries here) and another copy #6 of the fake item. All those are discarded except the function #5. This function receives lists #1 and #2 of items less than or greater than the pivot #3, then a continuation code #5 just after `\s\_\_sort_mark`. To avoid a memory problem described earlier, all of the ending functions read #6 until `\s\_\_sort_stop` and place #6 back into the input stream. When the lists #1 and #2 are empty, the `single` auxiliary simply places the continuation #5 before the pivot {#3}. When #2 is empty, #1 is sorted and placed before the pivot {#3}, taking care to feed the continuation #5 as a continuation for the function sorting #1. When #1 is empty, #2 is sorted, and the continuation argument is used to place the continuation #5 and the pivot {#3} before the sorted result. Finally, when both lists are non-empty, items larger than the pivot are sorted, then items less than the pivot, and the continuations are done in such a way to place the pivot in between.

```

23277 \cs_new:Npn __sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}
23278 \cs_new:Npn __sort_quick_single_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
23279 { #5 {#3} #6 \s__sort_stop }
23280 \cs_new:Npn __sort_quick_only_i_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
23281 {
23282 __sort_quick_split:NnNn #1
23283 __sort_quick_end:nnTFNn { } \s__sort_mark {#5}
23284 {#3}
23285 #6 \s__sort_stop
23286 }
23287 \cs_new:Npn __sort_quick_only_ii_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
23288 {

```

```

23289 __sort_quick_split:NnNn #2
23290 __sort_quick_end:nnTFNn { } \s__sort_mark { #5 {#3} }
23291 #6 \s__sort_stop
23292 }
23293 \cs_new:Npn __sort_quick_split_end:nnnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
23294 {
23295 __sort_quick_split:NnNn #2 __sort_quick_end:nnTFNn { } \s__sort_mark
23296 {
23297 __sort_quick_split:NnNn #1
23298 __sort_quick_end:nnTFNn { } \s__sort_mark {#5}
23299 {#3}
23300 }
23301 #6 \s__sort_stop
23302 }

```

(End definition for `\__sort_quick_end:nnTFNn` and others.)

## 39.6 Messages

`\__sort_error:` Bailing out of the sorting code is a bit tricky. It may not be safe to use a delimited argument, so instead we redefine many `\isort` commands to be trivial, with `\__sort_level:` jumping to the break point. This error recovery won't work in a group.

```

23303 \cs_new_protected:Npn __sort_error:
23304 {
23305 \cs_set_eq:NN __sort_merge_blocks_aux: \prg_do_nothing:
23306 \cs_set_eq:NN __sort_merge_blocks: \prg_do_nothing:
23307 \cs_set_protected:Npn __sort_level: { \group_end: \prg_break: }
23308 }

```

(End definition for `\__sort_error:..`)

`\__sort_disable_toksdef:` `\__sort_disabled_toksdef:n` While sorting, `\toksdef` is locally disabled to prevent users from using `\newtoks` or similar commands in their comparison code: the `\toks` registers that would be assigned are in use by `\isort`. In format mode, none of this is needed since there is no `\toks` allocator.

```

23309 \cs_new_protected:Npn __sort_disable_toksdef:
23310 { \cs_set_eq:NN \toksdef __sort_disabled_toksdef:n }
23311 \cs_new_protected:Npn __sort_disabled_toksdef:n #1
23312 {
23313 __kernel_msg_error:nnx { kernel } { toksdef }
23314 { \token_to_str:N #1 }
23315 __sort_error:
23316 \tex_toksdef:D #1
23317 }
23318 __kernel_msg_new:nnnn { kernel } { toksdef }
23319 { Allocation-of-\iow_char:N\\\toks-registers-impossible-while-sorting. }
23320 {
23321 The-comparison-code-used-for-sorting-a-list-has-attempted-to-
23322 define~#1-as-a-new-\iow_char:N\\toks-register-using-
23323 \iow_char:N\\newtoks-
23324 or-a-similar-command.-The-list-will-not-be-sorted.
23325 }

```

(End definition for `\__sort_disable_toksdef:` and `\__sort_disabled_toksdef:n.`)

\\_\\_sort\\_too\\_long\\_error:NNw When there are too many items in a sequence, this is an error, and we clean up properly the mapping over items in the list: break using the type-specific breaking function #1.

```

23326 \cs_new_protected:Npn __sort_too_long_error:NNw #1#2 \fi:
23327 {
23328 \fi:
23329 __kernel_msg_error:nxxxx { kernel } { too-large }
23330 { \token_to_str:N #2 }
23331 { \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
23332 { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
23333 #1 __sort_error:
23334 }
23335 __kernel_msg_new:nnnn { kernel } { too-large }
23336 { The-list~#1-is~too~long~to~be~sorted~by~TeX. }
23337 {
23338 TeX~has~#2~toks~registers~still~available:~
23339 this~only~allows~to~sort~with~up~to~#3~
23340 items.~The~list~will~not~be~sorted.
23341 }
```

(End definition for \\_\\_sort\\_too\\_long\\_error:NNw.)

```

23342 __kernel_msg_new:nnnn { kernel } { return-none }
23343 { The-comparison-code-did-not-return. }
23344 {
23345 When-sorting-a-list,~the~code~to~compare~items~#1~and~#2~
23346 did-not-call~
23347 \iow_char:N\sort_return_same: ~nor~
23348 \iow_char:N\sort_return_swapped: .~
23349 Exactly-one-of~these~should~be~called.
23350 }
23351 __kernel_msg_new:nnnn { kernel } { return-two }
23352 { The-comparison-code-returned-multiple-times. }
23353 {
23354 When-sorting-a-list,~the~code~to~compare~items~#1~and~#2~called~
23355 \iow_char:N\sort_return_same: ~or~
23356 \iow_char:N\sort_return_swapped: ~multiple-times.~
23357 Exactly-one-of~these~should~be~called.
23358 }
23359
```

/package)

## 40 I3tl-analysis implementation

23360 <@@=tl>

### 40.1 Internal functions

\s\\_tl The format used to store token lists internally uses the scan mark \s\\_tl as a delimiter.

(End definition for \s\\_tl.)

### 40.2 Internal format

The task of the I3tl-analysis module is to convert token lists to an internal format which allows us to extract all the relevant information about individual tokens (category code,

character code), as well as reconstruct the token list quickly. This internal format is used in `\l3regex` where we need to support arbitrary tokens, and it is used in conversion functions in `\l3str-convert`, where we wish to support clusters of characters instead of single tokens.

We thus need a way to encode any `\langle token \rangle` (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is to find `\langle tokens \rangle` which both o-expand and x-expand to the given `\langle token \rangle`. Collecting more information about the category code and character code is also useful for regular expressions, since most regexes are catcode-agnostic. The internal format thus takes the form of a succession of items of the form

```
\langle tokens \rangle \s__t1 \langle catcode \rangle \langle char code \rangle \s__t1
```

The `\langle tokens \rangle` o- and x-expand to the original token in the token list or to the cluster of tokens corresponding to one Unicode character in the given encoding (for `\l3str-convert`). The `\langle catcode \rangle` is given as a single hexadecimal digit, 0 for control sequences. The `\langle char code \rangle` is given as a decimal number, -1 for control sequences.

Using delimited arguments lets us build the `\langle tokens \rangle` progressively when doing an encoding conversion in `\l3str-convert`. On the other hand, the delimiter `\s__t1` may not appear unbraced in `\langle tokens \rangle`. This is not a problem because we are careful to wrap control sequences in braces (as an argument to `\exp_not:n`) when converting from a general token list to the internal format.

The current rule for converting a `\langle token \rangle` to a balanced set of `\langle tokens \rangle` which both o-expands and x-expands to it is the following.

- A control sequence `\cs` becomes `\exp_not:n { \cs } \s__t1 0 -1 \s__t1`.
- A begin-group character `{` becomes `\exp_after:wN { \if_false: } \fi: \s__t1 1 \langle char code \rangle \s__t1`.
- An end-group character `}` becomes `\if_false: { \fi: } \s__t1 2 \langle char code \rangle \s__t1`.
- A character with any other category code becomes `\exp_not:n {\langle character \rangle} \s__t1 \langle hex catcode \rangle \langle char code \rangle \s__t1`.

23361 `(*package)`

### 40.3 Variables and helper functions

`\s__t1` The scan mark `\s__t1` is used as a delimiter in the internal format. This is more practical than using a quark, because we would then need to control expansion much more carefully: compare `\int_value:w '#1 \s__t1` with `\int_value:w '#1 \exp_stop_f: \exp_not:N \q_mark` to extract a character code followed by the delimiter in an x-expansion.

23362 `\scan_new:N \s__t1`

*(End definition for `\s__t1`.)*

`\l__tl_analysis_token`  
`\l__tl_analysis_char_token` The tokens in the token list are probed with the TeX primitive `\futurelet`. We use `\l__tl_analysis_token` in that construction. In some cases, we convert the following token to a string before probing it: then the token variable used is `\l__tl_analysis_char_token`.

23363 `\cs_new_eq:NN \l__tl_analysis_token ?`

23364 `\cs_new_eq:NN \l__tl_analysis_char_token ?`

(End definition for `\l__tl_analysis_token` and `\l__tl_analysis_char_token`.)

`\l__tl_analysis_normal_int` The number of normal (N-type argument) tokens since the last special token.

23365 `\int_new:N \l__tl_analysis_normal_int`

(End definition for `\l__tl_analysis_normal_int`.)

`\l__tl_analysis_index_int` During the first pass, this is the index in the array being built. During the second pass, it is equal to the maximum index in the array from the first pass.

23366 `\int_new:N \l__tl_analysis_index_int`

(End definition for `\l__tl_analysis_index_int`.)

`\l__tl_analysis_nesting_int` Nesting depth of explicit begin-group and end-group characters during the first pass. This lets us detect the end of the token list without a reserved end-marker.

23367 `\int_new:N \l__tl_analysis_nesting_int`

(End definition for `\l__tl_analysis_nesting_int`.)

`\l__tl_analysis_type_int` When encountering special characters, we record their “type” in this integer.

23368 `\int_new:N \l__tl_analysis_type_int`

(End definition for `\l__tl_analysis_type_int`.)

`\g__tl_analysis_result_tl` The result of the conversion is stored in this token list, with a succession of items of the form

`<tokens> \s__tl <catcode> <char code> \s__tl`

23369 `\tl_new:N \g__tl_analysis_result_tl`

(End definition for `\g__tl_analysis_result_tl`.)

`\_tl_analysis extract charcode:` Extracting the character code from the meaning of `\l__tl_analysis_token`. This has no error checking, and should only be assumed to work for begin-group and end-group character tokens. It produces a number in the form ‘`<char>`’.

23370 `\cs_new:Npn \_tl_analysis_extract_charcode:`

23371 `{`

23372 `\exp_after:wN \_tl_analysis_extract_charcode_aux:w`

23373 `\token_to_meaning:N \l__tl_analysis_token`

23374 `}`

23375 `\cs_new:Npn \_tl_analysis_extract_charcode_aux:w #1 ~ #2 ~ { ' }`

(End definition for `\_tl_analysis_extract_charcode:` and `\_tl_analysis_extract_charcode_aux:w`.)

`\_tl_analysis_cs_space_count:NN` Counts the number of spaces in the string representation of its second argument, as well as the number of characters following the last space in that representation, and feeds the two numbers as semicolon-delimited arguments to the first argument. When this function is used, the escape character is printable and non-space.

23376 `\cs_new:Npn \_tl_analysis_cs_space_count:NN #1 #2`

23377 `{`

23378 `\exp_after:wN #1`

23379 `\int_value:w \int_eval:w 0`

23380 `\exp_after:wN \_tl_analysis_cs_space_count:w`

23381 `\token_to_str:N #2`

```

23382 \fi: __tl_analysis_cs_space_count_end:w ; ~ !
23383 }
23384 \cs_new:Npn __tl_analysis_cs_space_count:w #1 ~
23385 {
23386 \if_false: #1 #1 \fi:
23387 + 1
23388 __tl_analysis_cs_space_count:w
23389 }
23390 \cs_new:Npn __tl_analysis_cs_space_count_end:w ; #1 \fi: #2 !
23391 { \exp_after:wN ; \int_value:w \str_count_ignore_spaces:n {#1} ; }

(End definition for __tl_analysis_cs_space_count:NN, __tl_analysis_cs_space_count:w, and __tl_analysis_cs_space_count_end:w.)

```

## 40.4 Plan of attack

Our goal is to produce a token list of the form roughly

```

<token 1> \s@_<catcode 1> <char code 1> \s@_<catcode 2> <char code 2> \s@_<catcode N> <char code N> \s@_<catcode N> <char code N>

```

Most but not all tokens can be grabbed as an undelimited (N-type) argument by TeX. The plan is to have a two pass system. In the first pass, locate special tokens, and store them in various `\toks` registers. In the second pass, which is done within an x-expanding assignment, normal tokens are taken in as N-type arguments, and special tokens are retrieved from the `\toks` registers, and removed from the input stream by some means. The whole process takes linear time, because we avoid building the result one item at a time.

We make the escape character printable (backslash, but this later oscillates between slash and backslash): this allows us to distinguish characters from control sequences.

A token has two characteristics: its `\meaning`, and what it looks like for TeX when it is in scanning mode (*e.g.*, when capturing parameters for a macro). For our purposes, we distinguish the following meanings:

- begin-group token (category code 1), either space (character code 32), or non-space;
- end-group token (category code 2), either space (character code 32), or non-space;
- space token (category code 10, character code 32);
- anything else (then the token is always an N-type argument).

The token itself can “look like” one of the following

- a non-active character, in which case its meaning is automatically that associated to its character code and category code, we call it “true” character;
- an active character;
- a control sequence.

The only tokens which are not valid N-type arguments are true begin-group characters, true end-group characters, and true spaces. We detect those characters by scanning ahead with `\futurelet`, then distinguishing true characters from control sequences set equal to them using the `\string` representation.

The second pass is a simple exercise in expandable loops.

\\_\\_tl\\_analysis:n Everything is done within a group, and all definitions are local. We use \group\_align\\_safe\\_begin/end: to avoid problems in case \\_\\_tl\\_analysis:n is used within an alignment and its argument contains alignment tab tokens.

```

23392 \cs_new_protected:Npn __tl_analysis:n #1
23393 {
23394 \group_begin:
23395 \group_align_safe_begin:
23396 __tl_analysis_a:n {#1}
23397 __tl_analysis_b:n {#1}
23398 \group_align_safe_end:
23399 \group_end:
23400 }
```

(End definition for \\_\\_tl\\_analysis:n.)

## 40.5 Disabling active characters

\\_\\_tl\\_analysis\\_disable:n Active characters can cause problems later on in the processing, so we provide a way to disable them, by setting them to undefined. Since Unicode contains too many characters to loop over all of them, we instead do this whenever we encounter a character. For pTeX and upTeX we skip characters beyond [0, 255] because \lccode only allows those values.

```

23401 \group_begin:
23402 \char_set_catcode_active:N \^^@
23403 \cs_new_protected:Npn __tl_analysis_disable:n #1
23404 {
23405 \tex_lccode:D 0 = #1 \exp_stop_f:
23406 \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
23407 }
23408 \bool_lazy_or:nnT
23409 { \sys_if_engine_ptex_p: }
23410 { \sys_if_engine_uptex_p: }
23411 {
23412 \cs_gset_protected:Npn __tl_analysis_disable:n #1
23413 {
23414 \if_int_compare:w 256 > #1 \exp_stop_f:
23415 \tex_lccode:D 0 = #1 \exp_stop_f:
23416 \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
23417 \fi:
23418 }
23419 }
23420 }
```

(End definition for \\_\\_tl\\_analysis\\_disable:n.)

## 40.6 First pass

The goal of this pass is to detect special (non-N-type) tokens, and count how many N-type tokens lie between special tokens. Also, we wish to store some representation of each special token in a \toks register.

We have 11 types of tokens:

1. a true non-space begin-group character;
2. a true space begin-group character;

3. a true non-space end-group character;
4. a true space end-group character;
5. a true space blank space character;
6. an active character;
7. any other true character;
8. a control sequence equal to a begin-group token (category code 1);
9. a control sequence equal to an end-group token (category code 2);
10. a control sequence equal to a space token (character code 32, category code 10);
11. any other control sequence.

Our first tool is `\futurelet`. This cannot distinguish case 8 from 1 or 2, nor case 9 from 3 or 4, nor case 10 from case 5. Those cases are later distinguished by applying the `\string` primitive to the following token, after possibly changing the escape character to ensure that a control sequence's string representation cannot be mistaken for the true character.

In cases 6, 7, and 11, the following token is a valid N-type argument, so we grab it and distinguish the case of a character from a control sequence: in the latter case, `\str_tail:n {<token>}` is non-empty, because the escape character is printable.

`\__tl_analysis_a:n` We read tokens one by one using `\futurelet`. While performing the loop, we keep track of the number of true begin-group characters minus the number of true end-group characters in `\l__tl_analysis_nesting_int`. This reaches  $-1$  when we read the closing brace.

```

23421 \cs_new_protected:Npn __tl_analysis_a:n #1
23422 {
23423 __tl_analysis_disable:n { 32 }
23424 \int_set:Nn \tex_escapechar:D { 92 }
23425 \int_zero:N \l__tl_analysis_normal_int
23426 \int_zero:N \l__tl_analysis_index_int
23427 \int_zero:N \l__tl_analysis_nesting_int
23428 \if_false: { \fi: __tl_analysis_a_loop:w #1 }
23429 \int_decr:N \l__tl_analysis_index_int
23430 }

```

(End definition for `\__tl_analysis_a:n`.)

`\__tl_analysis_a_loop:w` Read one character and check its type.

```

23431 \cs_new_protected:Npn __tl_analysis_a_loop:w
23432 { \tex_futurelet:D \l__tl_analysis_token __tl_analysis_a_type:w }

```

(End definition for `\__tl_analysis_a_loop:w`.)

`\__tl_analysis_a_type:w` At this point, `\l__tl_analysis_token` holds the meaning of the following token. We store in `\l__tl_analysis_type_int` information about the meaning of the token ahead:

- 0 space token;
- 1 begin-group token;

- -1 end-group token;
- 2 other.

The values 0, 1, -1 correspond to how much a true such character changes the nesting level (2 is used only here, and is irrelevant later). Then call the auxiliary for each case. Note that nesting conditionals here is safe because we only skip over `\l__tl_analysis_token` if it matches with one of the character tokens (hence is not a primitive conditional).

```

23433 \cs_new_protected:Npn __tl_analysis_a_type:w
23434 {
23435 \l__tl_analysis_type_int =
23436 \if_meaning:w \l__tl_analysis_token \c_space_token
23437 0
23438 \else:
23439 \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
23440 1
23441 \else:
23442 \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
23443 - 1
23444 \else:
23445 2
23446 \fi:
23447 \fi:
23448 \fi:
23449 \exp_stop_f:
23450 \if_case:w \l__tl_analysis_type_int
23451 \exp_after:wN __tl_analysis_a_space:w
23452 \or: \exp_after:wN __tl_analysis_a_bgroup:w
23453 \or: \exp_after:wN __tl_analysis_a_safe:N
23454 \else: \exp_after:wN __tl_analysis_a_egroup:w
23455 \fi:
23456 }
```

(End definition for `\__tl_analysis_a_type:w`.)

```
__tl_analysis_a_space:w
__tl_analysis_a_space_test:w
```

In this branch, the following token's meaning is a blank space. Apply `\string` to that token: a true blank space gives a space, a control sequence gives a result starting with the escape character, an active character gives something else than a space since we disabled the space. We grab as `\l__tl_analysis_char_token` the first character of the string representation then test it in `\__tl_analysis_a_space_test:w`. Also, since `\__tl_analysis_a_store:` expects the special token to be stored in the relevant `\toks` register, we do that. The extra `\exp_not:n` is unnecessary of course, but it makes the treatment of all tokens more homogeneous. If we discover that the next token was actually a control sequence or an active character instead of a true space, then we step the counter of normal tokens. We now have in front of us the whole string representation of the control sequence, including potential spaces; those will appear to be true spaces later in this pass. Hence, all other branches of the code in this first pass need to consider the string representation, so that the second pass does not need to test the meaning of tokens, only strings.

```

23457 \cs_new_protected:Npn __tl_analysis_a_space:w
23458 {
23459 \tex_afterassignment:D __tl_analysis_a_space_test:w
23460 \exp_after:wN \cs_set_eq:NN
```

```

23461 \exp_after:wN \l__tl_analysis_char_token
23462 \token_to_str:N
23463 }
23464 \cs_new_protected:Npn __tl_analysis_a_space_test:w
23465 {
23466 \if_meaning:w \l__tl_analysis_char_token \c_space_token
23467 \tex_toks:D \l__tl_analysis_index_int { \exp_not:n { ~ } }
23468 __tl_analysis_a_store:
23469 \else:
23470 \int_incr:N \l__tl_analysis_normal_int
23471 \fi:
23472 __tl_analysis_a_loop:w
23473 }

```

(End definition for `\__tl_analysis_a_space:w` and `\__tl_analysis_a_space_test:w`.)

```

__tl_analysis_a_bgroup:w
__tl_analysis_a_egroup:w
__tl_analysis_a_group:nw
__tl_analysis_a_group_aux:w
__tl_analysis_a_group_auxii:w
__tl_analysis_a_group_test:w

```

The token is most likely a true character token with catcode 1 or 2, but it might be a control sequence, or an active character. Optimizing for the first case, we store in a toks register some code that expands to that token. Since we will turn what follows into a string, we make sure the escape character is different from the current character code (by switching between solidus and backslash). To detect the special case of an active character let to the catcode 1 or 2 character with the same character code, we disable the active character with that character code and re-test: if the following token has become undefined we can in fact safely grab it. We are finally ready to turn what follows to a string and test it. This is one place where we need `\l__tl_analysis_char_token` to be a separate control sequence from `\l__tl_analysis_token`, to compare them.

```

23474 \group_begin:
23475 \char_set_catcode_group_begin:N \^@ %
23476 \cs_new_protected:Npn __tl_analysis_a_bgroup:w
23477 { __tl_analysis_a_group:nw { \exp_after:wN \^@ \if_false: } \fi: } }
23478 \char_set_catcode_group_end:N \^@
23479 \cs_new_protected:Npn __tl_analysis_a_egroup:w
23480 { __tl_analysis_a_group:nw { \if_false: { \fi: \^@ } } % }
23481 \group_end:
23482 \cs_new_protected:Npn __tl_analysis_a_group:nw #1
23483 {
23484 \tex_lccode:D 0 = __tl_analysis_extract_charcode: \scan_stop:
23485 \tex_lowercase:D { \tex_toks:D \l__tl_analysis_index_int {#1} }
23486 \if_int_compare:w \tex_lccode:D 0 = \tex_escapechar:D
23487 \int_set:Nn \tex_escapechar:D { 139 - \tex_escapechar:D }
23488 \fi:
23489 __tl_analysis_disable:n { \tex_lccode:D 0 }
23490 \tex_futurelet:D \l__tl_analysis_token __tl_analysis_a_group_aux:w
23491 }
23492 \cs_new_protected:Npn __tl_analysis_a_group_aux:w
23493 {
23494 \if_meaning:w \l__tl_analysis_token \tex_undefined:D
23495 \exp_after:wN __tl_analysis_a_safe:N
23496 \else:
23497 \exp_after:wN __tl_analysis_a_group_auxii:w
23498 \fi:
23499 }
23500 \cs_new_protected:Npn __tl_analysis_a_group_auxii:w
23501 {

```

```

23502 \tex_afterassignment:D __tl_analysis_a_group_test:w
23503 \exp_after:wN \cs_set_eq:NN
23504 \exp_after:wN \l__tl_analysis_char_token
23505 \token_to_str:N
23506 }
23507 \cs_new_protected:Npn __tl_analysis_a_group_test:w
23508 {
23509 \if_charcode:w \l__tl_analysis_token \l__tl_analysis_char_token
23510 __tl_analysis_a_store:
23511 \else:
23512 \int_incr:N \l__tl_analysis_normal_int
23513 \fi:
23514 __tl_analysis_a_loop:w
23515 }

```

(End definition for `\__tl_analysis_a_bgroup:w` and others.)

`\__tl_analysis_a_store:`: This function is called each time we meet a special token; at this point, the `\toks` register `\l__tl_analysis_index_int` holds a token list which expands to the given special token. Also, the value of `\l__tl_analysis_type_int` indicates which case we are in:

- -1 end-group character;
- 0 space character;
- 1 begin-group character.

We need to distinguish further the case of a space character (code 32) from other character codes, because those behave differently in the second pass. Namely, after testing the `\lccode` of 0 (which holds the present character code) we change the cases above to

- -2 space end-group character;
- -1 non-space end-group character;
- 0 space blank space character;
- 1 non-space begin-group character;
- 2 space begin-group character.

This has the property that non-space characters correspond to odd values of `\l__tl_analysis_type_int`. The number of normal tokens until here and the type of special token are packed into a `\skip` register. Finally, we check whether we reached the last closing brace, in which case we stop by disabling the looping function (locally).

```

23516 \cs_new_protected:Npn __tl_analysis_a_store:
23517 {
23518 \tex_advance:D \l__tl_analysis_nesting_int \l__tl_analysis_type_int
23519 \if_int_compare:w \tex_lccode:D 0 = '\ \exp_stop_f:
23520 \tex_advance:D \l__tl_analysis_type_int \l__tl_analysis_type_int
23521 \fi:
23522 \tex_skip:D \l__tl_analysis_index_int
23523 = \l__tl_analysis_normal_int sp
23524 plus \l__tl_analysis_type_int sp \scan_stop:
23525 \int_incr:N \l__tl_analysis_index_int
23526 \int_zero:N \l__tl_analysis_normal_int

```

```

23527 \if_int_compare:w \l__tl_analysis_nesting_int = -1 \exp_stop_f:
23528 \cs_set_eq:NN __tl_analysis_a_loop:w \scan_stop:
23529 \fi:
23530 }

```

(End definition for `\__tl_analysis_a_store:`)

`\__tl_analysis_a_safe:N` This should be the simplest case: since the upcoming token is safe, we can simply grab it in a second pass. If the token is a single character (including space), the `\if_charcode:w` test yields true; we disable a potentially active character (that could otherwise masquerade as the true character in the next pass) and we count one “normal” token. On the other hand, if the token is a control sequence, we should replace it by its string representation for compatibility with other code branches. Instead of slowly looping through the characters with the main code, we use the knowledge of how the second pass works: if the control sequence name contains no space, count that token as a number of normal tokens equal to its string length. If the control sequence contains spaces, they should be registered as special characters by increasing `\l__tl_analysis_index_int` (no need to carefully count character between each space), and all characters after the last space should be counted in the following sequence of “normal” tokens.

```

23531 \cs_new_protected:Npn __tl_analysis_a_safe:N #1
23532 {
23533 \if_charcode:w
23534 \scan_stop:
23535 \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
23536 \scan_stop:
23537 \exp_after:wN \use_i:nn
23538 \else:
23539 \exp_after:wN \use_ii:nn
23540 \fi:
23541 {
23542 __tl_analysis_disable:n { '#1' }
23543 \int_incr:N \l__tl_analysis_normal_int
23544 }
23545 { __tl_analysis_cs_space_count:NN __tl_analysis_a_cs:ww #1 }
23546 __tl_analysis_a_loop:w
23547 }
23548 \cs_new_protected:Npn __tl_analysis_a_cs:ww #1; #2;
23549 {
23550 \if_int_compare:w #1 > 0 \exp_stop_f:
23551 \tex_skip:D \l__tl_analysis_index_int
23552 = \int_eval:n { \l__tl_analysis_normal_int + 1 } sp \exp_stop_f:
23553 \tex_advance:D \l__tl_analysis_index_int #1 \exp_stop_f:
23554 \else:
23555 \tex_advance:D
23556 \fi:
23557 \l__tl_analysis_normal_int #2 \exp_stop_f:
23558 }

```

(End definition for `\__tl_analysis_a_safe:N` and `\__tl_analysis_a_cs:ww`.)

## 40.7 Second pass

The second pass is an exercise in expandable loops. All the necessary information is stored in `\skip` and `\toks` registers.

\\_\\_tl\\_analysis\\_b:n  
\\_\\_tl\\_analysis\\_b\\_loop:w

Start the loop with the index 0. No need for an end-marker: the loop stops by itself when the last index is read. We repeatedly oscillate between reading long stretches of normal tokens, and reading special tokens.

```

23559 \cs_new_protected:Npn __tl_analysis_b:n #1
2360 {
2361 __kernel_tl_gset:Nx \g__tl_analysis_result_tl
2362 {
2363 __tl_analysis_b_loop:w 0; #1
2364 \prg_break_point:
2365 }
2366 }
2367 \cs_new:Npn __tl_analysis_b_loop:w #1;
2368 {
2369 \exp_after:wN __tl_analysis_b_normals:ww
2370 \int_value:w \tex_skip:D #1 ; #1 ;
2371 }

```

(End definition for \\_\\_tl\\_analysis\\_b:n and \\_\\_tl\\_analysis\\_b\\_loop:w.)

\\_\\_tl\\_analysis\\_b\\_normals:ww  
\\_\\_tl\\_analysis\\_b\\_normal:wwN

The first argument is the number of normal tokens which remain to be read, and the second argument is the index in the array produced in the first step. A character's string representation is always one character long, while a control sequence is always longer (we have set the escape character to a printable value). In both cases, we leave \exp\_not:n {\{token\}} \s\\_\\_tl in the input stream (after x-expansion). Here, \exp\_not:n is used rather than \exp\_not:N because #3 could be a macro parameter character or could be \s\\_\\_tl (which must be hidden behind braces in the result).

```

23572 \cs_new:Npn __tl_analysis_b_normals:ww #1;
23573 {
23574 \if_int_compare:w #1 = 0 \exp_stop_f:
23575 __tl_analysis_b_special:w
23576 \fi:
23577 __tl_analysis_b_normal:wwN #1;
23578 }
23579 \cs_new:Npn __tl_analysis_b_normal:wwN #1; #2; #3
23580 {
23581 \exp_not:n { \exp_not:n { #3 } } \s__tl
23582 \if_charcode:w
23583 \scan_stop:
23584 \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
23585 \scan_stop:
23586 \exp_after:wN __tl_analysis_b_char:Nww
23587 \else:
23588 \exp_after:wN __tl_analysis_b_cs:Nww
23589 \fi:
23590 #3 #1; #2;
23591 }

```

(End definition for \\_\\_tl\\_analysis\\_b\\_normals:ww and \\_\\_tl\\_analysis\\_b\\_normal:wwN.)

\\_\\_tl\\_analysis\\_b\\_char:Nww

If the normal token we grab is a character, leave  $\langle catcode \rangle \langle charcode \rangle$  followed by \s\\_\\_tl in the input stream, and call \\_\\_tl\\_analysis\\_b\\_normals:ww with its first argument decremented.

```
23592 \cs_new:Npx __tl_analysis_b_char:Nww #1
```

```

23593 {
23594 \exp_not:N \if_meaning:w #1 \exp_not:N \tex_undefined:D
23595 \token_to_str:N D \exp_not:N \else:
23596 \exp_not:N \if_catcode:w #1 \c_catcode_other_token
23597 \token_to_str:N C \exp_not:N \else:
23598 \exp_not:N \if_catcode:w #1 \c_catcode_letter_token
23599 \token_to_str:N B \exp_not:N \else:
23600 \exp_not:N \if_catcode:w #1 \c_math_toggle_token 3
23601 \exp_not:N \else:
23602 \exp_not:N \if_catcode:w #1 \c_alignment_token 4
23603 \exp_not:N \else:
23604 \exp_not:N \if_catcode:w #1 \c_math_superscript_token 7
23605 \exp_not:N \else:
23606 \exp_not:N \if_catcode:w #1 \c_math_subscript_token 8
23607 \exp_not:N \else:
23608 \exp_not:N \if_catcode:w #1 \c_space_token
23609 \token_to_str:N A \exp_not:N \else:
23610 6
23611 \exp_not:n { \fi: \fi: \fi: \fi: \fi: \fi: }
23612 \exp_not:N \int_value:w '#1 \s_tl
23613 \exp_not:N \exp_after:wN \exp_not:N __tl_analysis_b_normals:ww
23614 \exp_not:N \int_value:w \exp_not:N \int_eval:w - 1 +
23615 }

```

(End definition for `\__tl_analysis_b_char:Nww`.)

If the token we grab is a control sequence, leave 0 –1 (as category code and character code) in the input stream, followed by `\s_tl`, and call `\__tl_analysis_b_normals:ww` with updated arguments.

```

23616 \cs_new:Npn __tl_analysis_b_cs:Nww #1
23617 {
23618 0 -1 \s_tl
23619 __tl_analysis_cs_space_count>NN __tl_analysis_b_cs_test:ww #1
23620 }
23621 \cs_new:Npn __tl_analysis_b_cs_test:ww #1 ; #2 ; #3 ; #4 ;
23622 {
23623 \exp_after:wN __tl_analysis_b_normals:ww
23624 \int_value:w \int_eval:w
23625 \if_int_compare:w #1 = 0 \exp_stop_f:
23626 #3
23627 \else:
23628 \tex_skip:D \int_eval:n { #4 + #1 } \exp_stop_f:
23629 \fi:
23630 - #2
23631 \exp_after:wN ;
23632 \int_value:w \int_eval:n { #4 + #1 } ;
23633 }

```

(End definition for `\__tl_analysis_b_cs:Nww` and `\__tl_analysis_b_cs_test:ww`.)

`\__tl_analysis_b_special:w`  
`\__tl_analysis_b_special_char:wN`  
`\__tl_analysis_b_special_space:w`

Here, `#1` is the current index in the array built in the first pass. Check now whether we reached the end (we shouldn't keep the trailing end-group character that marked the end of the token list in the first pass). Unpack the `\toks` register: when x-expanding again,

we will get the special token. Then leave the category code in the input stream, followed by the character code, and call `\__tl_analysis_b_loop:w` with the next index.

```

23634 \group_begin:
23635 \char_set_catcode_other:N A
23636 \cs_new:Npn __tl_analysis_b_special:w
23637 \fi: __tl_analysis_b_normal:wwN 0 ; #1 ;
23638 {
23639 \fi:
23640 \if_int_compare:w #1 = \l__tl_analysis_index_int
23641 \exp_after:wN \prg_break:
23642 \fi:
23643 \tex_the:D \tex_toks:D #1 \s__tl
23644 \if_case:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
23645 \token_to_str:N A
23646 \or: 1
23647 \or: 1
23648 \else: 2
23649 \fi:
23650 \if_int_odd:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
23651 \exp_after:wN __tl_analysis_b_special_char:wN \int_value:w
23652 \else:
23653 \exp_after:wN __tl_analysis_b_special_space:w \int_value:w
23654 \fi:
23655 \int_eval:n { 1 + #1 } \exp_after:wN ;
23656 \token_to_str:N
23657 }
23658 \group_end:
23659 \cs_new:Npn __tl_analysis_b_special_char:wN #1 ; #2
23660 {
23661 \int_value:w '#2 \s__tl
23662 __tl_analysis_b_loop:w #1 ;
23663 }
23664 \cs_new:Npn __tl_analysis_b_special_space:w #1 ; ~
23665 {
23666 32 \s__tl
23667 __tl_analysis_b_loop:w #1 ;
23668 }

```

(End definition for `\__tl_analysis_b_special:w`, `\__tl_analysis_b_special_char:wN`, and `\__tl_analysis_b_special_space:w`.)

## 40.8 Mapping through the analysis

First obtain the analysis of the token list into `\g__tl_analysis_result_tl`. To allow nested mappings, increase the nesting depth `\g__kernel_prg_map_int` (shared between all modules), then define the looping macro, which has a name specific to that nesting depth. That looping grabs the `tokens`, `catcode` and `char code`; it checks for the end of the loop with `\use_none:n ##2`, normally empty, but which becomes `\tl_map_break:` at the end; it then performs the user's code `#2`, and loops by calling itself. When the loop ends, remember to decrease the nesting depth.

```

23669 \cs_new_protected:Npn \tl_analysis_map_inline:nn #1
23670 {
23671 __tl_analysis:n {#1}

```

```

23672 \int_gincr:N \g__kernel_prg_map_int
23673 \exp_args:Nc __tl_analysis_map_inline_aux:Nn
23674 { __tl_analysis_map_inline_ \int_use:N \g__kernel_prg_map_int :wNw }
23675 }
23676 \cs_new_protected:Npn \tl_analysis_map_inline:Nn #1
23677 { \exp_args:No \tl_analysis_map_inline:nn #1 }
23678 \cs_new_protected:Npn __tl_analysis_map_inline_aux:Nn #1#2
23679 {
23680 \cs_gset_protected:Npn #1 ##1 \s__tl ##2 ##3 \s__tl
23681 {
23682 \use_none:n ##2
23683 __tl_analysis_map_inline_aux:nnn {##1} {##3} {##2}
23684 }
23685 \cs_gset_protected:Npn __tl_analysis_map_inline_aux:nnn ##1##2##3
23686 {
23687 #2
23688 #1
23689 }
23690 \exp_after:wN #1
23691 \g__tl_analysis_result_tl
23692 \s__tl { ? \tl_map_break: } \s__tl
23693 \prg_break_point:Nn \tl_map_break:
23694 { \int_gdecr:N \g__kernel_prg_map_int }
23695 }

```

(End definition for `\tl_analysis_map_inline:nn` and others. These functions are documented on page [225](#).)

## 40.9 Showing the results

`\tl_analysis_show:N` Add to `\__tl_analysis:n` a third pass to display tokens to the terminal. If the token list variable is not defined, throw the same error as `\tl_show:N` by simply calling that function.

```

23696 \cs_new_protected:Npn \tl_analysis_show:N #1
23697 {
23698 \tl_if_exist:NTF #1
23699 {
23700 \exp_args:No __tl_analysis:n {#1}
23701 \msg_show:nnxxxx { LaTeX / kernel } { show-tl-analysis }
23702 { \token_to_str:N #1 } { __tl_analysis_show: } { } { }
23703 }
23704 { \tl_show:N #1 }
23705 }
23706 \cs_new_protected:Npn \tl_analysis_show:n #1
23707 {
23708 __tl_analysis:n {#1}
23709 \msg_show:nnxxxx { LaTeX / kernel } { show-tl-analysis }
23710 { } { __tl_analysis_show: } { } { }
23711 }

```

(End definition for `\tl_analysis_show:N` and `\tl_analysis_show:n`. These functions are documented on page [225](#).)

`\__tl_analysis_show:` Here, `#1` o- and x-expands to the token; `#2` is the category code (one uppercase hexadecimal digit), 0 for control sequences; `#3` is the character code, which we ignore. In the

cases of control sequences and active characters, the meaning may overflow one line, and we want to truncate it. Those cases are thus separated out.

```

23712 \cs_new:Npn __tl_analysis_show:
23713 {
23714 \exp_after:wN __tl_analysis_show_loop:wNw \g__tl_analysis_result_tl
23715 \s__tl { ? \prg_break: } \s__tl
23716 \prg_break_point:
23717 }
23718 \cs_new:Npn __tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl
23719 {
23720 \use_none:n #2
23721 \iow_newline: > \use:nn { ~ } { ~ }
23722 \if_int_compare:w "#2 = 0 \exp_stop_f:
23723 \exp_after:wN __tl_analysis_show_cs:n
23724 \else:
23725 \if_int_compare:w "#2 = 13 \exp_stop_f:
23726 \exp_after:wN \exp_after:wN
23727 \exp_after:wN __tl_analysis_show_active:n
23728 \else:
23729 \exp_after:wN \exp_after:wN
23730 \exp_after:wN __tl_analysis_show_normal:n
23731 \fi:
23732 \fi:
23733 {#1}
23734 __tl_analysis_show_loop:wNw
23735 }
```

(End definition for `\__tl_analysis_show:` and `\__tl_analysis_show_loop:wNw`.)

`\__tl_analysis_show_normal:n` Non-active characters are a simple matter of printing the character, and its meaning. Our test suite checks that begin-group and end-group characters do not mess up TeX's alignment status.

```

23736 \cs_new:Npn __tl_analysis_show_normal:n #1
23737 {
23738 \exp_after:wN \token_to_str:N #1 ~
23739 (\exp_after:wN \token_to_meaning:N #1)
23740 }
```

(End definition for `\__tl_analysis_show_normal:n`.)

`\__tl_analysis_show_value:N` This expands to the value of #1 if it has any.

```

23741 \cs_new:Npn __tl_analysis_show_value:N #1
23742 {
23743 \token_if_expandable:NF #1
23744 {
23745 \token_if_chardef:NTF #1 \prg_break: { }
23746 \token_if_mathchardef:NTF #1 \prg_break: { }
23747 \token_if_dim_register:NTF #1 \prg_break: { }
23748 \token_if_int_register:NTF #1 \prg_break: { }
23749 \token_if_skip_register:NTF #1 \prg_break: { }
23750 \token_if_toks_register:NTF #1 \prg_break: { }
23751 \use_none:nnn
23752 \prg_break_point:
23753 \use:n { \exp_after:wN = \tex_the:D #1 }
```

```

23754 }
23755 }

```

(End definition for `\_\_tl\_analysis\_show\_value:N.`)

`\_\_tl\_analysis\_show\_cs:n`  
`\_\_tl\_analysis\_show\_active:n`  
`\_\_tl\_analysis\_show\_long:nn`  
`\_\_tl\_analysis\_show\_long\_aux:nnnn`

Control sequences and active characters are printed in the same way, making sure not to go beyond the `\l_iow_line_count_int`. In case of an overflow, we replace the last characters by `\c\_\_tl\_analysis\_show\_etc\_str`.

```

23756 \cs_new:Npn __tl_analysis_show_cs:n #1
23757 { \exp_args:No __tl_analysis_show_long:nn {#1} { control~sequence= } }
23758 \cs_new:Npn __tl_analysis_show_active:n #1
23759 { \exp_args:No __tl_analysis_show_long:nn {#1} { active~character= } }
23760 \cs_new:Npn __tl_analysis_show_long:nn #1
23761 {
23762 __tl_analysis_show_long_aux:oofn
23763 { \token_to_str:N #1 }
23764 { \token_to_meaning:N #1 }
23765 { __tl_analysis_show_value:N #1 }
23766 }
23767 \cs_new:Npn __tl_analysis_show_long_aux:nnnn #1#2#3#4
23768 {
23769 \int_compare:nNnTF
23770 { \str_count:n { #1 ~ (#4 #2 #3) } }
23771 > { \l_iow_line_count_int - 3 }
23772 {
23773 \str_range:nnn { #1 ~ (#4 #2 #3) } { 1 }
23774 {
23775 \l_iow_line_count_int - 3
23776 - \str_count:N \c__tl_analysis_show_etc_str
23777 }
23778 \c__tl_analysis_show_etc_str
23779 }
23780 { #1 ~ (#4 #2 #3) }
23781 }
23782 \cs_generate_variant:Nn __tl_analysis_show_long_aux:nnnn { oof }

(End definition for __tl_analysis_show_cs:n and others.)
```

## 40.10 Messages

When a control sequence (or active character) and its meaning are too long to fit in one line of the terminal, the end is replaced by this token list.

```

23783 \tl_const:Nx \c__tl_analysis_show_etc_str % (
23784 { \token_to_str:N \ETC. }

```

(End definition for `\c\_\_tl\_analysis\_show\_etc\_str.`)

```

23785 __kernel_msg_new:nnn { kernel } { show-tl-analysis }
23786 {
23787 The~token~list~ \tl_if_empty:nF {#1} { #1 ~ }
23788 \tl_if_empty:nTF {#2}
23789 { is~empty }
23790 { contains~the~tokens: #2 }
23791 }
23792
```

## 41 I3regex implementation

```
23793 {*package}
23794 {@@=regex}
```

### 41.1 Plan of attack

Most regex engines use backtracking. This allows to provide very powerful features (back-references come to mind first), but it is costly, and raises the problem of catastrophic backtracking. Since TEX is not first and foremost a programming language, complicated code tends to run slowly, and we must use faster, albeit slightly more restrictive, techniques, coming from automata theory.

Given a regular expression of  $n$  characters, we do the following:

- (Compiling.) Analyse the regex, finding invalid input, and convert it to an internal representation.
- (Building.) Convert the compiled regex to a non-deterministic finite automaton (NFA) with  $O(n)$  states which accepts precisely token lists matching that regex.
- (Matching.) Loop through the query token list one token (one “position”) at a time, exploring in parallel every possible path (“active thread”) through the NFA, considering active threads in an order determined by the quantifiers’ greediness.

We use the following vocabulary in the code comments (and in variable names).

- *Group*: index of the capturing group,  $-1$  for non-capturing groups.
- *Position*: each token in the query is labelled by an integer  $\langle position \rangle$ , with  $\text{min\_pos} - 1 \leq \langle position \rangle \leq \text{max\_pos}$ . The lowest and highest positions correspond to imaginary begin and end markers (with inaccessible category code and character code).
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labelled by an integer  $\langle state \rangle$  with  $\text{min\_state} \leq \langle state \rangle < \text{max\_state}$ .
- *Active thread*: state of the NFA that is reached when reading the query token list for the matching. Those threads are ordered according to the greediness of quantifiers.
- *Step*: used when matching, starts at  $0$ , incremented every time a character is read, and is not reset when searching for repeated matches. The integer  $\text{\_1\_regex\_step\_int}$  is a unique id for all the steps of the matching algorithm.

We use `I3intarray` to manipulate arrays of integers (stored into some dimension registers in scaled points). We also abuse TEX’s `\toks` registers, by accessing them directly by number rather than tying them to control sequence using the `\newtoks` allocation functions. Specifically, these arrays and `\toks` are used as follows. When building, `\toks(state)` holds the tests and actions to perform in the  $\langle state \rangle$  of the NFA. When matching,

- `\g__regex_state_active_intarray` holds the last  $\langle step \rangle$  in which each  $\langle state \rangle$  was active.

- `\g__regex_thread_state_intarray` maps each  $\langle thread \rangle$  (with `min_active \leq \langle thread \rangle < max_active`) to the  $\langle state \rangle$  in which the  $\langle thread \rangle$  currently is. The  $\langle threads \rangle$  are ordered starting from the best to the least preferred.
- `\toks\langle thread \rangle` holds the submatch information for the  $\langle thread \rangle$ , as the contents of a property list.
- `\g__regex_charcode_intarray` and `\g__regex_catcode_intarray` hold the character codes and category codes of tokens at each  $\langle position \rangle$  in the query.
- `\g__regex_balance_intarray` holds the balance of begin-group and end-group character tokens which appear before that point in the token list.
- `\toks\langle position \rangle` holds  $\langle tokens \rangle$  which o- and x-expand to the  $\langle position \rangle$ -th token in the query.
- `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray` and `\g__regex_submatch_end_intarray` hold, for each submatch (as would be extracted by `\regex_extract_all:nnN`), the place where the submatch started to be looked for and its two end-points. For historical reasons, the minimum index is twice `max_state`, and the used registers go up to `\l__regex_submatch_int`. They are organized in blocks of `\l__regex_capturing_group_int` entries, each block corresponding to one match with all its submatches stored in consecutive entries.

The code is structured as follows. Variables are introduced in the relevant section. First we present some generic helper functions. Then comes the code for compiling a regular expression, and for showing the result of the compilation. The building phase converts a compiled regex to NFA states, and the automaton is run by the code in the following section. The only remaining brick is parsing the replacement text and performing the replacement. We are then ready for all the user functions. Finally, messages, and a little bit of tracing code.

## 41.2 Helpers

`\_regex_int_eval:w` Access the primitive: performance is key here, so we do not use the slower route via `\int_eval:n`.

```
23795 \cs_new_eq:NN _regex_int_eval:w \tex_numexpr:D
```

(End definition for `\_regex_int_eval:w`.)

`\_regex_standard_escapechar:` Make the `\escapechar` into the standard backslash.

```
23796 \cs_new_protected:Npn _regex_standard_escapechar:
23797 { \int_set:Nn \tex_escapechar:D { '\\" } }
```

(End definition for `\_regex_standard_escapechar:..`)

`\_regex_toks_use:w` Unpack a `\toks` given its number.

```
23798 \cs_new:Npn _regex_toks_use:w { \tex_the:D \tex_toks:D }
```

(End definition for `\_regex_toks_use:w`.)

\\_\\_regex\\_toks\\_clear:N  
\\_\\_regex\\_toks\\_set:Nn  
\\_\\_regex\\_toks\\_set:No

```

23799 \cs_new_protected:Npn __regex_toks_clear:N #1
23800 { __regex_toks_set:Nn #1 { } }
23801 \cs_new_eq:NN __regex_toks_set:Nn \tex_toks:D
23802 \cs_new_protected:Npn __regex_toks_set:No #1
23803 { __regex_toks_set:Nn #1 \exp_after:wN }
```

(End definition for \\_\\_regex\\_toks\\_clear:N and \\_\\_regex\\_toks\\_set:Nn.)

\\_\\_regex\\_toks\\_memcpy:NNn Copy #3 \toks registers from #2 onwards to #1 onwards, like C's memcpy.

```

23804 \cs_new_protected:Npn __regex_toks_memcpy:NNn #1#2#3
23805 {
23806 \prg_replicate:nn {#3}
23807 {
23808 \tex_toks:D #1 = \tex_toks:D #2
23809 \int_incr:N #1
23810 \int_incr:N #2
23811 }
23812 }
```

(End definition for \\_\\_regex\\_toks\\_memcpy:NNn.)

\\_\\_regex\\_toks\\_put\\_left:Nx  
\\_\\_regex\\_toks\\_put\\_right:Nx  
\\_\\_regex\\_toks\\_put\\_right:Nn

During the building phase we wish to add x-expanded material to \toks, either to the left or to the right. The expansion is done “by hand” for optimization (these operations are used quite a lot). The Nn version of \\_\\_regex\\_toks\\_put\\_right:Nx is provided because it is more efficient than x-expanding with \exp\_not:n.

```

23813 \cs_new_protected:Npn __regex_toks_put_left:Nx #1#2
23814 {
23815 \cs_set:Npx __regex_tmp:w { #2 }
23816 \tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN
23817 { \exp_after:wN __regex_tmp:w \tex_the:D \tex_toks:D #1 }
23818 }
23819 \cs_new_protected:Npn __regex_toks_put_right:Nx #1#2
23820 {
23821 \cs_set:Npx __regex_tmp:w {#2}
23822 \tex_toks:D #1 \exp_after:wN
23823 { \tex_the:D \tex_toks:D \exp_after:wN #1 __regex_tmp:w }
23824 }
23825 \cs_new_protected:Npn __regex_toks_put_right:Nn #1#2
23826 { \tex_toks:D #1 \exp_after:wN { \tex_the:D \tex_toks:D #1 #2 } }
```

(End definition for \\_\\_regex\\_toks\\_put\\_left:Nx and \\_\\_regex\\_toks\\_put\\_right:Nx.)

\\_\\_regex\\_curr\\_cs\\_to\\_str:

Expands to the string representation of the token (known to be a control sequence) at the current position \l\\_regex\\_curr\\_pos\\_int. It should only be used in x-expansion to avoid losing a leading space.

```

23827 \cs_new:Npn __regex_curr_cs_to_str:
23828 {
23829 \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
23830 \tex_the:D \tex_toks:D \l_regex_curr_pos_int
23831 }
```

(End definition for \\_\\_regex\\_curr\\_cs\\_to\\_str:.)

#### 41.2.1 Constants and variables

|                                                                                                                                                                                                                                                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \_\_regex\_tmp:w                                                                                                                                                                                                                                          | Temporary function used for various short-term purposes.<br>23832 \cs_new:Npn \_\_regex\_tmp:w { }                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|                                                                                                                                                                                                                                                           | (End definition for \_\_regex\_tmp:w.)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| \l\_\_regex\_internal\_a\_tl<br>\l\_\_regex\_internal\_b\_tl<br>\l\_\_regex\_internal\_a\_int<br>\l\_\_regex\_internal\_b\_int<br>\l\_\_regex\_internal\_c\_int<br>\l\_\_regex\_internal\_bool<br>\l\_\_regex\_internal\_seq<br>\g\_\_regex\_internal\_tl | Temporary variables used for various purposes.<br>23833 \tl_new:N \l\_\_regex\_internal\_a\_tl<br>23834 \tl_new:N \l\_\_regex\_internal\_b\_tl<br>23835 \int_new:N \l\_\_regex\_internal\_a\_int<br>23836 \int_new:N \l\_\_regex\_internal\_b\_int<br>23837 \int_new:N \l\_\_regex\_internal\_c\_int<br>23838 \bool_new:N \l\_\_regex\_internal\_bool<br>23839 \seq_new:N \l\_\_regex\_internal\_seq<br>23840 \tl_new:N \g\_\_regex\_internal\_tl<br><br>(End definition for \l\_\_regex\_internal\_a\_tl and others.)                                                                                                                                                                                                |
| \l\_\_regex\_build\_tl                                                                                                                                                                                                                                    | This temporary variable is specifically for use with the <code>tl_build</code> machinery.<br>23841 \tl_new:N \l\_\_regex\_build\_tl<br><br>(End definition for \l\_\_regex\_build\_tl.)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| \c\_\_regex\_no\_match\_regex                                                                                                                                                                                                                             | This regular expression matches nothing, but is still a valid regular expression. We could use a failing assertion, but I went for an empty class. It is used as the initial value for regular expressions declared using <code>\regex_new:N</code> .<br>23842 \tl_const:Nn \c\_\_regex\_no\_match\_regex<br>23843 {<br>23844 \_\_regex_branch:n<br>23845 { \_\_regex_class:NnnN \c\_\_true\_bool { } { 1 } { 0 } \c\_\_true\_bool }<br>23846 }<br><br>(End definition for \c\_\_regex\_no\_match\_regex.)                                                                                                                                                                                                            |
| \g\_\_regex\_charcode\_intarray<br>\g\_\_regex\_catcode\_intarray<br>\g\_\_regex\_balance\_intarray                                                                                                                                                       | The first thing we do when matching is to go once through the query token list and store the information for each token into <code>\g\_\_regex\_charcode\_intarray</code> , <code>\g\_\_regex\_catcode\_intarray</code> and <code>\toks</code> registers. We also store the balance of begin-group/end-group characters into <code>\g\_\_regex\_balance\_intarray</code> .<br>23847 \intarray_new:Nn \g\_\_regex\_charcode\_intarray { 65536 }<br>23848 \intarray_new:Nn \g\_\_regex\_catcode\_intarray { 65536 }<br>23849 \intarray_new:Nn \g\_\_regex\_balance\_intarray { 65536 }<br><br>(End definition for \g\_\_regex\_charcode\_intarray, \g\_\_regex\_catcode\_intarray, and \g\_\_regex\_balance\_intarray.) |
| \l\_\_regex\_balance\_int                                                                                                                                                                                                                                 | During this phase, <code>\l\_\_regex\_balance\_int</code> counts the balance of begin-group and end-group character tokens which appear before a given point in the token list. This variable is also used to keep track of the balance in the replacement text.<br>23850 \int_new:N \l\_\_regex\_balance\_int<br><br>(End definition for \l\_\_regex\_balance\_int.)                                                                                                                                                                                                                                                                                                                                                 |
| \l\_\_regex\_cs\_name\_tl                                                                                                                                                                                                                                 | This variable is used in <code>\_\_regex\_item\_cs:n</code> to store the csname of the currently-tested token when the regex contains a sub-regex for testing csnames.<br>23851 \tl_new:N \l\_\_regex\_cs\_name\_tl<br><br>(End definition for \l\_\_regex\_cs\_name\_tl.)                                                                                                                                                                                                                                                                                                                                                                                                                                            |

### 41.2.2 Testing characters

```
\c__regex_ascii_min_int
 \c__regex_ascii_max_control_int
\c__regex_ascii_max_int
 23852 \int_const:Nn \c__regex_ascii_min_int { 0 }
 23853 \int_const:Nn \c__regex_ascii_max_control_int { 31 }
 23854 \int_const:Nn \c__regex_ascii_max_int { 127 }

(End definition for \c__regex_ascii_min_int, \c__regex_ascii_max_control_int, and \c__regex_ascii_max_int.)
```

```
\c__regex_ascii_lower_int
 23855 \int_const:Nn \c__regex_ascii_lower_int { 'a - 'A }

(End definition for \c__regex_ascii_lower_int.)
```

### 41.2.3 Internal auxiliaries

```
\q__regex_recursion_stop
Internal recursion quarks.
 23856 \quark_new:N \q__regex_recursion_stop

(End definition for \q__regex_recursion_stop.)
```

Functions to gobble up to a quark.

```
 23857 \cs_new:Npn __regex_use_none_delimit_by_q_recursion_stop:w
 23858 #1 \q__regex_recursion_stop { }
 23859 \cs_new:Npn __regex_use_i_delimit_by_q_recursion_stop:nw
 23860 #1 #2 \q__regex_recursion_stop {#1}

(End definition for __regex_use_none_delimit_by_q_recursion_stop:w and __regex_use_i_delimit-
by_q_recursion_stop:nw.)
```

```
\q__regex_nil
Internal quarks.
 23861 \quark_new:N \q__regex_nil

(End definition for \q__regex_nil.)
```

Branching quark conditional.

```
 23862 __kernel_quark_new_conditional:Nn __regex_quark_if_nil:N { F }

(End definition for __regex_quark_if_nil:nTF.)
```

`\__regex_break_point:TF`    `\__regex_break_true:w`

When testing whether a character of the query token list matches a given character class in the regular expression, we often have to test it against several ranges of characters, checking if any one of those matches. This is done with a structure like

```
⟨test1⟩ ... ⟨testn⟩
__regex_break_point:TF {⟨true code⟩} {⟨false code⟩}
```

If any of the tests succeeds, it calls `\__regex_break_true:w`, which cleans up and leaves `⟨true code⟩` in the input stream. Otherwise, `\__regex_break_point:TF` leaves the `⟨false code⟩` in the input stream.

```
 23863 \cs_new_protected:Npn __regex_break_true:w
 23864 #1 __regex_break_point:TF #2 #3 {#2}
 23865 \cs_new_protected:Npn __regex_break_point:TF #1 #2 { #2 }

(End definition for __regex_break_point:TF and __regex_break_true:w.)
```

\\_\\_regex\\_item\\_reverse:n This function makes showing regular expressions easier, and lets us define \D in terms of \d for instance. There is a subtlety: the end of the query is marked by -2, and thus matches \D and other negated properties; this case is caught by another part of the code.

```
23866 \cs_new_protected:Npn __regex_item_reverse:n #1
23867 {
23868 #1
23869 __regex_break_point:TF { } __regex_break_true:w
23870 }
```

(End definition for \\_\\_regex\\_item\\_reverse:n.)

\\_\\_regex\\_item\\_caseful\\_equal:n Simple comparisons triggering \\_\\_regex\\_break\\_true:w when true.

```
23871 \cs_new_protected:Npn __regex_item_caseful_equal:n #1
23872 {
23873 \if_int_compare:w #1 = \l__regex_curr_char_int
23874 \exp_after:wN __regex_break_true:w
23875 \fi:
23876 }
23877 \cs_new_protected:Npn __regex_item_caseful_range:nn #1 #2
23878 {
23879 \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
23880 \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
23881 \exp_after:wN \exp_after:wN \exp_after:wN __regex_break_true:w
23882 \fi:
23883 \fi:
23884 }
```

(End definition for \\_\\_regex\\_item\\_caseful\\_equal:n and \\_\\_regex\\_item\\_caseful\\_range:nn.)

\\_\\_regex\\_item\\_caseless\\_equal:n For caseless matching, we perform the test both on the `current_char` and on the `case\_changed\_char`. Before doing the second set of tests, we make sure that `case\_changed\_char` has been computed.

```
23885 \cs_new_protected:Npn __regex_item_caseless_equal:n #1
23886 {
23887 \if_int_compare:w #1 = \l__regex_curr_char_int
23888 \exp_after:wN __regex_break_true:w
23889 \fi:
23890 \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
23891 __regex_compute_case_changed_char:
23892 \fi:
23893 \if_int_compare:w #1 = \l__regex_case_changed_char_int
23894 \exp_after:wN __regex_break_true:w
23895 \fi:
23896 }
23897 \cs_new_protected:Npn __regex_item_caseless_range:nn #1 #2
23898 {
23899 \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
23900 \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
23901 \exp_after:wN \exp_after:wN \exp_after:wN __regex_break_true:w
23902 \fi:
23903 \fi:
23904 \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
23905 __regex_compute_case_changed_char:
23906 \fi:
```

```

23907 \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
23908 \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
23909 \exp_after:wN \exp_after:wN \exp_after:wN __regex_break_true:w
23910 \fi:
23911 \fi:
23912 }

```

(End definition for `\__regex_item_caseless_equal:n` and `\__regex_item_caseless_range:nn`.)

`\__regex_compute_case_changed_char:`

This function is called when `\l__regex_case_changed_char_int` has not yet been computed (or rather, when it is set to the marker value `\c_max_int`). If the current character code is in the range [65, 90] (upper-case), then add 32, making it lowercase. If it is in the lower-case letter range [97, 122], subtract 32.

```

23913 \cs_new_protected:Npn __regex_compute_case_changed_char:
23914 {
23915 \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int
23916 \if_int_compare:w \l__regex_curr_char_int > 'Z \exp_stop_f:
23917 \if_int_compare:w \l__regex_curr_char_int > 'z \exp_stop_f: \else:
23918 \if_int_compare:w \l__regex_curr_char_int < 'a \exp_stop_f: \else:
23919 \int_sub:Nn \l__regex_case_changed_char_int
23920 { \c__regex_ascii_lower_int }
23921 \fi:
23922 \fi:
23923 \else:
23924 \if_int_compare:w \l__regex_curr_char_int < 'A \exp_stop_f: \else:
23925 \int_add:Nn \l__regex_case_changed_char_int
23926 { \c__regex_ascii_lower_int }
23927 \fi:
23928 \fi:
23929 }

```

(End definition for `\__regex_compute_case_changed_char:..`)

`\__regex_item_equal:n`  
`\__regex_item_range:nn`

Those must always be defined to expand to a `caseful` (default) or `caseless` version, and not be protected: they must expand when compiling, to hard-code which tests are caseless or caseful.

```

23930 \cs_new_eq:NN __regex_item_equal:n ?
23931 \cs_new_eq:NN __regex_item_range:nn ?

```

(End definition for `\__regex_item_equal:n` and `\__regex_item_range:nn`.)

`\__regex_item_catcode:nT`  
`\__regex_item_catcode_reverse:nT`  
`\__regex_item_catcode:`

The argument is a sum of powers of 4 with exponents given by the allowed category codes (between 0 and 13). Dividing by a given power of 4 gives an odd result if and only if that category code is allowed. If the catcode does not match, then skip the character code tests which follow.

```

23932 \cs_new_protected:Npn __regex_item_catcode:
23933 {
23934 "
23935 \if_case:w \l__regex_curr_catcode_int
23936 1 \or: 4 \or: 10 \or: 40
23937 \or: 100 \or: \or: 1000 \or: 4000
23938 \or: 10000 \or: \or: 100000 \or: 400000
23939 \or: 1000000 \or: 4000000 \else: 1*0
23940 \fi:

```

```

23941 }
23942 \cs_new_protected:Npn __regex_item_catcode:nT #1
23943 {
23944 \if_int_odd:w \int_eval:n { #1 / __regex_item_catcode: } \exp_stop_f:
23945 \exp_after:wN \use:n
23946 \else:
23947 \exp_after:wN \use_none:n
23948 \fi:
23949 }
23950 \cs_new_protected:Npn __regex_item_catcode_reverse:nT #1#2
23951 { __regex_item_catcode:nT {#1} { __regex_item_reverse:n {#2} } }

(End definition for __regex_item_catcode:nT, __regex_item_catcode_reverse:nT, and __regex_item_catcode:..)

```

\\_\_regex\_item\_exact:nn This matches an exact *<category>-<character code>* pair, or an exact control sequence, more precisely one of several possible control sequences.

```

23952 \cs_new_protected:Npn __regex_item_exact:nn #1#2
23953 {
23954 \if_int_compare:w #1 = \l__regex_curr_catcode_int
23955 \if_int_compare:w #2 = \l__regex_curr_char_int
23956 \exp_after:wN \exp_after:wN \exp_after:wN __regex_break_true:w
23957 \fi:
23958 \fi:
23959 }
23960 \cs_new_protected:Npn __regex_item_exact_cs:n #1
23961 {
23962 \int_compare:nNnTF \l__regex_curr_catcode_int = 0
23963 {
23964 __kernel_tl_set:Nx \l__regex_internal_a_tl
23965 { \scan_stop: __regex_curr_cs_to_str: \scan_stop: }
23966 \tl_if_in:noTF { \scan_stop: } { \scan_stop: }
23967 \l__regex_internal_a_tl
23968 { __regex_break_true:w } { }
23969 }
23970 { }
23971 }

```

(End definition for \\_\_regex\_item\_exact:nn and \\_\_regex\_item\_exact\_cs:n.)

\\_\_regex\_item\_cs:n Match a control sequence (the argument is a compiled regex). First test the catcode of the current token to be zero. Then perform the matching test, and break if the csname indeed matches. The three \exp\_after:wN expand the contents of the \toks(*current position*) (of the form \exp\_not:n {*<control sequence>*}) to *<control sequence>*. We store the cs name before building states for the cs, as those states may overlap with toks registers storing the user's input.

```

23972 \cs_new_protected:Npn __regex_item_cs:n #1
23973 {
23974 \int_compare:nNnT \l__regex_curr_catcode_int = 0
23975 {
23976 \group_begin:
23977 __kernel_tl_set:Nx \l__regex_cs_name_tl { __regex_curr_cs_to_str: }
23978 __regex_single_match:
23979 __regex_disable_submatches:

```

```

23980 __regex_build_for_cs:n {#1}
23981 \bool_set_eq:NN \l_regex_saved_success_bool
23982 \g_regex_success_bool
23983 \exp_args:NV __regex_match_cs:n \l_regex_cs_name_tl
23984 \if_meaning:w \c_true_bool \g_regex_success_bool
23985 \group_insert_after:N __regex_break_true:w
23986 \fi:
23987 \bool_gset_eq:NN \g_regex_success_bool
23988 \l_regex_saved_success_bool
23989 \group_end:
23990 }
23991 }
```

(End definition for `\_\_regex_item_cs:n`.)

#### 41.2.4 Character property tests

|                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>\_\_regex_prop_d:</code> | Character property tests for <code>\d</code> , <code>\W</code> , etc. These character properties are not affected by the <code>(?i)</code> option. The characters recognized by each one are as follows: <code>\d=[0-9]</code> , <code>\w=[0-9A-Z_a-z]</code> , <code>\s=[\u\^\^I\^\^J\^\^L\^\^M]</code> , <code>\h=[\u\^\^I]</code> , <code>\v=[\^\^J-\^\^M]</code> , and the upper case counterparts match anything that the lower case does not match. The order in which the various tests appear is optimized for usual mostly lower case letter text. |
| <code>\_\_regex_prop_h:</code> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>\_\_regex_prop_s:</code> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>\_\_regex_prop_v:</code> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>\_\_regex_prop_w:</code> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>\_\_regex_prop_N:</code> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

```

23992 \cs_new_protected:Npn __regex_prop_d:
23993 { __regex_item_caseful_range:nn { '0 } { '9 } }
23994 \cs_new_protected:Npn __regex_prop_h:
23995 {
23996 __regex_item_caseful_equal:n { '\ }
23997 __regex_item_caseful_equal:n { '\^\^I }
23998 }
23999 \cs_new_protected:Npn __regex_prop_s:
24000 {
24001 __regex_item_caseful_equal:n { '\ }
24002 __regex_item_caseful_equal:n { '\^\^I }
24003 __regex_item_caseful_equal:n { '\^\^J }
24004 __regex_item_caseful_equal:n { '\^\^L }
24005 __regex_item_caseful_equal:n { '\^\^M }
24006 }
24007 \cs_new_protected:Npn __regex_prop_v:
24008 { __regex_item_caseful_range:nn { '\^\^J } { '\^\^M } } % lf, vtab, ff, cr
24009 \cs_new_protected:Npn __regex_prop_w:
24010 {
24011 __regex_item_caseful_range:nn { 'a } { 'z }
24012 __regex_item_caseful_range:nn { 'A } { 'Z }
24013 __regex_item_caseful_range:nn { '0 } { '9 }
24014 __regex_item_caseful_equal:n { '_' }
24015 }
24016 \cs_new_protected:Npn __regex_prop_N:
24017 {
24018 __regex_item_reverse:n
24019 { __regex_item_caseful_equal:n { '\^\^J } }
24020 }
```

(End definition for `\_\_regex_prop_d: and others.`)

```

__regex_posix_alnum: POSIX properties. No surprise.

__regex_posix_alpha:
24021 \cs_new_protected:Npn __regex_posix_alnum:
24022 { __regex_posix_alpha: __regex_posix_digit: }
__regex_posix_ascii:
24023 \cs_new_protected:Npn __regex_posix_alpha:
24024 { __regex_posix_lower: __regex_posix_upper: }
__regex_posix_cntrl:
24025 \cs_new_protected:Npn __regex_posix_ascii:
__regex_posix_digit:
24026 {
__regex_posix_graph:
24027 __regex_item_caseful_range:nn
__regex_posix_lower:
24028 \c__regex_ascii_min_int
24029 \c__regex_ascii_max_int
__regex_posix_punct:
24030 }
__regex_posix_space:
24031 \cs_new_eq:NN __regex_posix_blank: __regex_prop_h:
__regex_posix_upper:
24032 \cs_new_protected:Npn __regex_posix_cntrl:
24033 {
__regex_posix_word:
24034 __regex_item_caseful_range:nn
24035 \c__regex_ascii_min_int
24036 \c__regex_ascii_max_control_int
24037 __regex_item_caseful_equal:n \c__regex_ascii_max_int
24038 }
24039 \cs_new_eq:NN __regex_posix_digit: __regex_prop_d:
24040 \cs_new_protected:Npn __regex_posix_graph:
24041 { __regex_item_caseful_range:nn { '!' } { '\~{}' } }
24042 \cs_new_protected:Npn __regex_posix_lower:
24043 { __regex_item_caseful_range:nn { 'a' } { 'z' } }
24044 \cs_new_protected:Npn __regex_posix_print:
24045 { __regex_item_caseful_range:nn { '\`{}' } { '\~{}' } }
24046 \cs_new_protected:Npn __regex_posix_punct:
24047 {
24048 __regex_item_caseful_range:nn { '!' } { '/' }
24049 __regex_item_caseful_range:nn { ':' } { '@' }
24050 __regex_item_caseful_range:nn { '[' } { '[' }
24051 __regex_item_caseful_range:nn { '\{ ' } { '\~{}' }
24052 }
24053 \cs_new_protected:Npn __regex_posix_space:
24054 {
24055 __regex_item_caseful_equal:n { '\`{}' }
24056 __regex_item_caseful_range:nn { '\^{}I' } { '\^{}M' }
24057 }
24058 \cs_new_protected:Npn __regex_posix_upper:
24059 { __regex_item_caseful_range:nn { 'A' } { 'Z' } }
24060 \cs_new_eq:NN __regex_posix_word: __regex_prop_w:
24061 \cs_new_protected:Npn __regex_posix_xdigit:
24062 {
24063 __regex_posix_digit:
24064 __regex_item_caseful_range:nn { 'A' } { 'F' }
24065 __regex_item_caseful_range:nn { 'a' } { 'f' }
24066 }

```

(End definition for `\_\_regex_posix_alnum:` and others.)

#### 41.2.5 Simple character escape

Before actually parsing the regular expression or the replacement text, we go through them once, converting `\n` to the character 10, *etc.* In this pass, we also convert any special

character (\*, ?, {, etc.) or escaped alphanumeric character into a marker indicating that this was a special sequence, and replace escaped special characters and non-escaped alphanumeric characters by markers indicating that those were “raw” characters. The rest of the code can then avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

Usage: `\_regex_escape_use:nnnn <inline 1> <inline 2> <inline 3> {<token list>}`  
The `<token list>` is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function `<inline 1>`, and escaped characters are fed to the function `<inline 2>` within an x-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` are recognized, and those are replaced by the corresponding character, then fed to `<inline 3>`. The result is then left in the input stream. Spaces are ignored unless escaped.

The conversion is done within an x-expanding assignment.

`\_regex_escape_use:nnnn` The result is built in `\l\_regex_internal_a_t1`, which is then left in the input stream. Tracing code is added as appropriate inside this token list. Go through #4 once, applying #1, #2, or #3 as relevant to each character (after de-escaping it).

```

24067 \cs_new_protected:Npn _regex_escape_use:nnnn #1#2#3#4
24068 {
24069 \group_begin:
24070 \tl_clear:N \l__regex_internal_a_t1
24071 \cs_set:Npn _regex_escape_unescaped:N ##1 { #1 }
24072 \cs_set:Npn _regex_escape_escaped:N ##1 { #2 }
24073 \cs_set:Npn _regex_escape_raw:N ##1 { #3 }
24074 _regex_standard_escapechar:
24075 \kernel_tl_gset:Nx \g__regex_internal_tl
24076 { _kernel_str_to_other_fast:n {#4} }
24077 \tl_put_right:Nx \l__regex_internal_a_t1
24078 {
24079 \exp_after:wN _regex_escape_loop:N \g__regex_internal_tl
24080 { break } \prg_break_point:
24081 }
24082 \exp_after:wN
24083 \group_end:
24084 \l__regex_internal_a_t1
24085 }
```

(End definition for `\_regex_escape_use:nnnn`.)

`\_regex_escape_loop:N` `\_regex_escape_loop:N` reads one character: if it is special (space, backslash, or end-marker), perform the associated action, otherwise it is simply an unescaped character. After a backslash, the same is done, but unknown characters are “escaped”.

```

24086 \cs_new:Npn _regex_escape_loop:N #1
24087 {
24088 \cs_if_exist_use:cF { __regex_escape_\token_to_str:N #1:w }
24089 { _regex_escape_unescaped:N #1 }
24090 _regex_escape_loop:N
24091 }
24092 \cs_new:cpn { __regex_escape_ \c_backslash_str :w }
24093 _regex_escape_loop:N #1
24094 {
24095 \cs_if_exist_use:cF { __regex_escape_/\token_to_str:N #1:w }
```

```

24096 { __regex_escape_escaped:N #1 }
24097 __regex_escape_loop:N
24098 }

```

(End definition for \\_\\_regex\\_escape\\_loop:N and \\_\\_regex\\_escape\\_\\:w.)

\\_\\_regex\\_escape\\_unescape:N  
\\_\\_regex\\_escape\\_escaped:N  
\\_\\_regex\\_escape\\_raw:N

Those functions are never called before being given a new meaning, so their definitions here don't matter.

```

24099 \cs_new_eq:NN __regex_escape_unescape:N ?
24100 \cs_new_eq:NN __regex_escape_escaped:N ?
24101 \cs_new_eq:NN __regex_escape_raw:N ?

```

(End definition for \\_\\_regex\\_escape\\_unescape:N, \\_\\_regex\\_escape\\_escaped:N, and \\_\\_regex\\_escape\\_raw:N.)

\\_\\_regex\\_escape\\_break:w  
\\_\\_regex\\_escape\\_\\:w  
\\_\\_regex\\_escape\\_\\/break:w  
\\_\\_regex\\_escape\\_\\/e:w  
\\_\\_regex\\_escape\\_\\/f:w  
\\_\\_regex\\_escape\\_\\/n:w  
\\_\\_regex\\_escape\\_\\/r:w  
\\_\\_regex\\_escape\\_\\/t:w  
\\_\\_regex\\_escape\\_\\:w

The loop is ended upon seeing the end-marker "break", with an error if the string ended in a backslash. Spaces are ignored, and \a, \e, \f, \n, \r, \t take their meaning here.

```

24102 \cs_new_eq:NN __regex_escape_break:w \prg_break:
24103 \cs_new:cpx { __regex_escape_\\/break:w }
24104 {
24105 __kernel_msg_expandable_error:nn { kernel } { trailing-backslash }
24106 \prg_break:
24107 }
24108 \cs_new:cpx { __regex_escape_\\:w } { }
24109 \cs_new:cpx { __regex_escape_\\/e:w }
24110 { \exp_not:N __regex_escape_raw:N \iow_char:N \^\^G }
24111 \cs_new:cpx { __regex_escape_\\/f:w }
24112 { \exp_not:N __regex_escape_raw:N \iow_char:N \^\^I }
24113 \cs_new:cpx { __regex_escape_\\/n:w }
24114 { \exp_not:N __regex_escape_raw:N \iow_char:N \^\^J }
24115 \cs_new:cpx { __regex_escape_\\/r:w }
24116 { \exp_not:N __regex_escape_raw:N \iow_char:N \^\^L }
24117 \cs_new:cpx { __regex_escape_\\/t:w }
24118 { \exp_not:N __regex_escape_raw:N \iow_char:N \^\^M }
24119 \cs_new:cpx { __regex_escape_\\:w }
24120 { \exp_not:N __regex_escape_raw:N \iow_char:N \^\^[}

```

(End definition for \\_\\_regex\\_escape\\_break:w and others.)

\\_\\_regex\\_escape\\_\\:x:w  
\\_\\_regex\\_escape\\_x\\_end:w  
\\_\\_regex\\_escape\\_x\\_large:n

When \x is encountered, \\_\\_regex\\_escape\_x\\_test:N is responsible for grabbing some hexadecimal digits, and feeding the result to \\_\\_regex\\_escape\_x\\_end:w. If the number is too big interrupt the assignment and produce an error, otherwise call \\_\\_regex\\_escape\\_raw:N on the corresponding character token.

```

24121 \cs_new:cpx { __regex_escape_\\:x:w } __regex_escape_loop:N
24122 {
24123 \exp_after:wN __regex_escape_x_end:w
24124 \int_value:w "0 __regex_escape_x_test:N
24125 }
24126 \cs_new:Npn __regex_escape_x_end:w #1 ;
24127 {
24128 \int_compare:nNnTF {#1} > \c_max_char_int
24129 {
24130 __kernel_msg_expandable_error:nnff { kernel } { x-overflow }
24131 {#1} { \int_to_Hex:n {#1} }

```

```

24132 }
24133 {
24134 \exp_last_unbraced:Nf __regex_escape_raw:N
24135 { \char_generate:nn {#1} { 12 } }
24136 }
24137 }
```

(End definition for `\__regex_escape_x:w`, `\__regex_escape_x_end:w`, and `\__regex_escape_x_large:n`.)

`\__regex_escape_x_test:N` Find out whether the first character is a left brace (allowing any number of hexadecimal digits), or not (allowing up to two hexadecimal digits). We need to check for the end-of-string marker. Eventually, call either `\__regex_escape_x_loop:N` or `\__regex_escape_x:N`.

```

24138 \cs_new:Npn __regex_escape_x_test:N #1
24139 {
24140 \str_if_eq:nnTF {#1} { break } { ; }
24141 {
24142 \if_charcode:w \c_space_token #1
24143 \exp_after:wN __regex_escape_x_test:N
24144 \else:
24145 \exp_after:wN __regex_escape_x_testii:N
24146 \exp_after:wN #1
24147 \fi:
24148 }
24149 }
24150 \cs_new:Npn __regex_escape_x_testii:N #1
24151 {
24152 \if_charcode:w \c_left_brace_str #1
24153 \exp_after:wN __regex_escape_x_loop:N
24154 \else:
24155 __regex_hexadecimal_use:NTF #1
24156 { \exp_after:wN __regex_escape_x:N }
24157 { ; \exp_after:wN __regex_escape_loop:N \exp_after:wN #1 }
24158 \fi:
24159 }
```

(End definition for `\__regex_escape_x_test:N` and `\__regex_escape_x_testii:N`.)

`\__regex_escape_x:N` This looks for the second digit in the unbraced case.

```

24160 \cs_new:Npn __regex_escape_x:N #1
24161 {
24162 \str_if_eq:nnTF {#1} { break } { ; }
24163 {
24164 __regex_hexadecimal_use:NTF #1
24165 { ; __regex_escape_loop:N }
24166 { ; __regex_escape_loop:N #1 }
24167 }
24168 }
```

(End definition for `\__regex_escape_x:N`.)

`\__regex_escape_x_loop:N` Grab hexadecimal digits, skip spaces, and at the end, check that there is a right brace, otherwise raise an error outside the assignment.

```
24169 \cs_new:Npn __regex_escape_x_loop:N #1
```

```

24170 {
24171 \str_if_eq:nnTF {#1} { break }
24172 { ; __regex_escape_x_loop_error:n { } {#1} }
24173 {
24174 __regex_hexadecimal_use:NTF #1
24175 { __regex_escape_x_loop:N }
24176 {
24177 \token_if_eq_charcode:NNTF \c_space_token #1
24178 { __regex_escape_x_loop:N }
24179 {
24180 ;
24181 \exp_after:wN
24182 \token_if_eq_charcode:NNTF \c_right_brace_str #1
24183 { __regex_escape_loop:N }
24184 { __regex_escape_x_loop_error:n {#1} }
24185 }
24186 }
24187 }
24188 }
24189 \cs_new:Npn __regex_escape_x_loop_error:n #1
24190 {
24191 __kernel_msg_expandable_error:nnn { kernel } { x-missing-rbrace } {#1}
24192 __regex_escape_loop:N #1
24193 }

```

(End definition for `\__regex_escape_x_loop:N` and `\__regex_escape_x_loop_error:..`)

`\__regex_hexadecimal_use:NTF` T<sub>E</sub>X detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

24194 \prg_new_conditional:Npnn __regex_hexadecimal_use:N #1 { TF }
24195 {
24196 \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
24197 #1 \prg_return_true:
24198 \else:
24199 \if_case:w
24200 \int_eval:n { \exp_after:wN ‘ \token_to_str:N #1 - ‘a }
24201 A
24202 \or: B
24203 \or: C
24204 \or: D
24205 \or: E
24206 \or: F
24207 \else:
24208 \prg_return_false:
24209 \exp_after:wN \use_none:n
24210 \fi:
24211 \prg_return_true:
24212 \fi:
24213 }

```

(End definition for `\__regex_hexadecimal_use:NTF`.)

`\__regex_char_if_alphanumeric:NTF` and `\__regex_char_if_special:NTF` These two tests are used in the first pass when parsing a regular expression. That pass is responsible for finding escaped and non-escaped characters, and recognizing which ones have special meanings and which should be interpreted as “raw” characters. Namely,

- alphanumerics are “raw” if they are not escaped, and may have a special meaning when escaped;
- non-alphanumeric printable ascii characters are “raw” if they are escaped, and may have a special meaning when not escaped;
- characters other than printable ascii are always “raw”.

The code is ugly, and highly based on magic numbers and the ascii codes of characters. This is mostly unavoidable for performance reasons. Maybe the tests can be optimized a little bit more. Here, “alphanumeric” means 0–9, A–Z, a–z; “special” character means non-alphanumeric but printable ascii, from space (hex 20) to del (hex 7E).

```

24214 \prg_new_conditional:Npnn __regex_char_if_special:N #1 { TF }
24215 {
24216 \if_int_compare:w '#1 > 'Z \exp_stop_f:
24217 \if_int_compare:w '#1 > 'z \exp_stop_f:
24218 \if_int_compare:w '#1 < \c__regex_ascii_max_int
24219 \prg_return_true: \else: \prg_return_false: \fi:
24220 \else:
24221 \if_int_compare:w '#1 < 'a \exp_stop_f:
24222 \prg_return_true: \else: \prg_return_false: \fi:
24223 \fi:
24224 \else:
24225 \if_int_compare:w '#1 > '9 \exp_stop_f:
24226 \if_int_compare:w '#1 < 'A \exp_stop_f:
24227 \prg_return_true: \else: \prg_return_false: \fi:
24228 \else:
24229 \if_int_compare:w '#1 < '0 \exp_stop_f:
24230 \if_int_compare:w '#1 < '\ \exp_stop_f:
24231 \prg_return_false: \else: \prg_return_true: \fi:
24232 \else: \prg_return_false: \fi:
24233 \fi:
24234 \fi:
24235 }
24236 \prg_new_conditional:Npnn __regex_char_if_alphanumeric:N #1 { TF }
24237 {
24238 \if_int_compare:w '#1 > 'Z \exp_stop_f:
24239 \if_int_compare:w '#1 > 'z \exp_stop_f:
24240 \prg_return_false:
24241 \else:
24242 \if_int_compare:w '#1 < 'a \exp_stop_f:
24243 \prg_return_false: \else: \prg_return_true: \fi:
24244 \fi:
24245 \else:
24246 \if_int_compare:w '#1 > '9 \exp_stop_f:
24247 \if_int_compare:w '#1 < 'A \exp_stop_f:
24248 \prg_return_false: \else: \prg_return_true: \fi:
24249 \else:
24250 \if_int_compare:w '#1 < '0 \exp_stop_f:
24251 \prg_return_false: \else: \prg_return_true: \fi:
24252 \fi:
24253 \fi:
24254 }

```

(End definition for `\__regex_char_if_alphanumeric:NTF` and `\__regex_char_if_special:NTF`.)

## 41.3 Compiling

A regular expression starts its life as a string of characters. In this section, we convert it to internal instructions, resulting in a “compiled” regular expression. This compiled expression is then turned into states of an automaton in the building phase. Compiled regular expressions consist of the following:

- `\__regex_class:NnnnN <boolean> {<tests>} {<min>} {<more>} <lazyness>`
- `\__regex_group:nnnN {<branches>} {<min>} {<more>} <lazyness>`, also `\__regex_group_no_capture:nnnN` and `\__regex_group_resetting:nnnN` with the same syntax.
- `\__regex_branch:n {<contents>}`
- `\__regex_command_K:`
- `\__regex_assertion:Nn <boolean> {<assertion test>}`, where the `<assertion test>` is `\__regex_b_test:` or `{\__regex_anchor:N <integer>}`

Tests can be the following:

- `\__regex_item_caseful_equal:n {<char code>}`
- `\__regex_item_caseless_equal:n {<char code>}`
- `\__regex_item_caseful_range:nn {<min>} {<max>}`
- `\__regex_item_caseless_range:nn {<min>} {<max>}`
- `\__regex_item_catcode:nT {<catcode bitmap>} {<tests>}`
- `\__regex_item_catcode_reverse:nT {<catcode bitmap>} {<tests>}`
- `\__regex_item_reverse:n {<tests>}`
- `\__regex_item_exact:nn {<catcode>} {<char code>}`
- `\__regex_item_exact_cs:n {<csnames>}`, more precisely given as `<csname> \scan_stop: <csname> \scan_stop: <csname>` and so on in a brace group.
- `\__regex_item_cs:n {<compiled regex>}`

### 41.3.1 Variables used when compiling

We make sure to open the same number of groups as we close.

24255 `\int_new:N \l__regex_group_level_int`

(End definition for `\l__regex_group_level_int`.)

While compiling, ten modes are recognized, labelled  $-63, -23, -6, -2, 0, 2, 3, 6, 23, 63$ . See section 41.3.3. We only define some of these as constants.

24256 `\int_new:N \l__regex_mode_int`

24257 `\int_const:Nn \c__regex_cs_in_class_mode_int { -6 }`

24258 `\int_const:Nn \c__regex_cs_mode_int { -2 }`

24259 `\int_const:Nn \c__regex_outer_mode_int { 0 }`

24260 `\int_const:Nn \c__regex_catcode_mode_int { 2 }`

24261 `\int_const:Nn \c__regex_class_mode_int { 3 }`

24262 `\int_const:Nn \c__regex_catcode_in_class_mode_int { 6 }`

(End definition for `\l_regex_mode_int` and others.)

```
\l_regex_catcodes_int
 \l_regex_default_catcodes_int
\l_regex_catcodes_bool
```

We wish to allow constructions such as `\c[^BE](..\cL[a-z]..)`, where the outer catcode test applies to the whole group, but is superseded by the inner catcode test. For this to work, we need to keep track of lists of allowed category codes: `\l_regex_catcodes_int` and `\l_regex_default_catcodes_int` are bitmaps, sums of  $4^c$ , for all allowed catcodes  $c$ . The latter is local to each capturing group, and we reset `\l_regex_catcodes_int` to that value after each character or class, changing it only when encountering a `\c` escape. The boolean records whether the list of categories of a catcode test has to be inverted: compare `\c[^BE]` and `\c[BE]`.

```
24263 \int_new:N \l_regex_catcodes_int
24264 \int_new:N \l_regex_default_catcodes_int
24265 \bool_new:N \l_regex_catcodes_bool
```

(End definition for `\l_regex_catcodes_int`, `\l_regex_default_catcodes_int`, and `\l_regex_catcodes_bool`.)

```
\c_regex_catcode_C_int
\c_regex_catcode_B_int
\c_regex_catcode_E_int
\c_regex_catcode_M_int
\c_regex_catcode_T_int
\c_regex_catcode_P_int
\c_regex_catcode_U_int
\c_regex_catcode_D_int
\c_regex_catcode_S_int
\c_regex_catcode_L_int
\c_regex_catcode_O_int
\c_regex_catcode_A_int
\c_regex_all_catcodes_int
```

Constants:  $4^c$  for each category, and the sum of all powers of 4.

```
24266 \int_const:Nn \c_regex_catcode_C_int { "1" }
24267 \int_const:Nn \c_regex_catcode_B_int { "4" }
24268 \int_const:Nn \c_regex_catcode_E_int { "10" }
24269 \int_const:Nn \c_regex_catcode_M_int { "40" }
24270 \int_const:Nn \c_regex_catcode_T_int { "100" }
24271 \int_const:Nn \c_regex_catcode_P_int { "1000" }
24272 \int_const:Nn \c_regex_catcode_U_int { "4000" }
24273 \int_const:Nn \c_regex_catcode_D_int { "10000" }
24274 \int_const:Nn \c_regex_catcode_S_int { "100000" }
24275 \int_const:Nn \c_regex_catcode_L_int { "400000" }
24276 \int_const:Nn \c_regex_catcode_O_int { "1000000" }
24277 \int_const:Nn \c_regex_catcode_A_int { "4000000" }
24278 \int_const:Nn \c_regex_all_catcodes_int { "5515155" }
```

(End definition for `\c_regex_catcode_C_int` and others.)

```
\l_regex_internal_regex
```

The compilation step stores its result in this variable.

```
24279 \cs_new_eq:NN \l_regex_internal_regex \c_regex_no_match_regex
```

(End definition for `\l_regex_internal_regex`.)

```
\l_regex_show_prefix_seq
```

This sequence holds the prefix that makes up the line displayed to the user. The various items must be removed from the right, which is tricky with a token list, hence we use a sequence.

```
24280 \seq_new:N \l_regex_show_prefix_seq
```

(End definition for `\l_regex_show_prefix_seq`.)

```
\l_regex_show_lines_int
```

A hack. To know whether a given class has a single item in it or not, we count the number of lines when showing the class.

```
24281 \int_new:N \l_regex_show_lines_int
```

(End definition for `\l_regex_show_lines_int`.)

### 41.3.2 Generic helpers used when compiling

\\_\\_regex\\_two\\_if\\_eq:NNNNTF

Used to compare pairs of things like \\_\\_regex\\_compile\\_special:N ? together. It's often inconvenient to get the catcodes of the character to match so we just compare the character code. Besides, the expanding behaviour of \if:w is very useful as that means we can use \c\_left\_brace\_str and the like.

```
24282 \prg_new_conditional:Npnn __regex_two_if_eq:NNNN #1#2#3#4 { TF }
24283 {
24284 \if_meaning:w #1 #3
24285 \if:w #2 #4
24286 \prg_return_true:
24287 \else:
24288 \prg_return_false:
24289 \fi:
24290 \else:
24291 \prg_return_false:
24292 \fi:
24293 }
```

(End definition for \\_\\_regex\\_two\\_if\\_eq:NNNNTF.)

\\_\\_regex\\_get\\_digits:NTFw  
\\_\\_regex\\_get\\_digits\\_loop:w

If followed by some raw digits, collect them one by one in the integer variable #1, and take the **true** branch. Otherwise, take the **false** branch.

```
24294 \cs_new_protected:Npn __regex_get_digits:NTFw #1#2#3#4#5
24295 {
24296 __regex_if_raw_digit:NNTF #4 #5
24297 { #1 = #5 __regex_get_digits_loop:nw {#2} }
24298 { #3 #4 #5 }
24299 }
24300 \cs_new:Npn __regex_get_digits_loop:nw #1#2#3
24301 {
24302 __regex_if_raw_digit:NNTF #2 #3
24303 { #3 __regex_get_digits_loop:nw {#1} }
24304 { \scan_stop: #1 #2 #3 }
24305 }
```

(End definition for \\_\\_regex\_get\_digits:NTFw and \\_\\_regex\_get\_digits\_loop:w.)

\\_\\_regex\_if\_raw\_digit:NNTF

Test used when grabbing digits for the {m,n} quantifier. It only accepts non-escaped digits.

```
24306 \prg_new_conditional:Npnn __regex_if_raw_digit:NN #1#2 { TF }
24307 {
24308 \if_meaning:w __regex_compile_raw:N #1
24309 \if_int_compare:w 1 < 1 #2 \exp_stop_f:
24310 \prg_return_true:
24311 \else:
24312 \prg_return_false:
24313 \fi:
24314 \else:
24315 \prg_return_false:
24316 \fi:
24317 }
```

(End definition for \\_\\_regex\_if\_raw\_digit:NNTF.)

### 41.3.3 Mode

When compiling the NFA corresponding to a given regex string, we can be in ten distinct modes, which we label by some magic numbers:

- 6  $[\backslash c\{ \dots \}]$  control sequence in a class,
- 2  $\backslash c\{ \dots \}$  control sequence,
- 0 ... outer,
- 2  $\backslash c\dots$  catcode test,
- 6  $[\backslash c\dots]$  catcode test in a class,
- 63  $[\backslash c\{[\dots]\}]$  class inside mode -6,
- 23  $\backslash c\{[\dots]\}$  class inside mode -2,
- 3 [...] class inside mode 0,
- 23  $\backslash c[ \dots ]$  class inside mode 2,
- 63  $[\backslash c[ \dots ]]$  class inside mode 6.

This list is exhaustive, because  $\backslash c$  escape sequences cannot be nested, and character classes cannot be nested directly. The choice of numbers is such as to optimize the most useful tests, and make transitions from one mode to another as simple as possible.

- Even modes mean that we are not directly in a character class. In this case, a left bracket appends 3 to the mode. In a character class, a right bracket changes the mode as  $m \rightarrow (m - 15)/13$ , truncated.
- Grouping, assertion, and anchors are allowed in non-positive even modes (0, -2, -6), and do not change the mode. Otherwise, they trigger an error.
- A left bracket is special in even modes, appending 3 to the mode; in those modes, quantifiers and the dot are recognized, and the right bracket is normal. In odd modes (within classes), the left bracket is normal, but the right bracket ends the class, changing the mode from  $m$  to  $(m - 15)/13$ , truncated; also, ranges are recognized.
- In non-negative modes, left and right braces are normal. In negative modes, however, left braces trigger a warning; right braces end the control sequence, going from -2 to 0 or -6 to 3, with error recovery for odd modes.
- Properties (such as the  $\backslash d$  character class) can appear in any mode.

`\_\_regex_if_in_class:TF` Test whether we are directly in a character class (at the innermost level of nesting). There, many escape sequences are not recognized, and special characters are normal. Also, for every raw character, we must look ahead for a possible raw dash.

```

24318 \cs_new:Npn __regex_if_in_class:TF
24319 {
24320 \if_int_odd:w \l__regex_mode_int
24321 \exp_after:wN \use_i:nn
24322 \else:
24323 \exp_after:wN \use_ii:nn
24324 \fi:
24325 }
```

(End definition for `\_regex_if_in_class:TF`.)

`\_regex_if_in_cs:TF` Right braces are special only directly inside control sequences (at the inner-most level of nesting, not counting groups).

```
24326 \cs_new:Npn _regex_if_in_cs:TF
24327 {
24328 \if_int_odd:w \l__regex_mode_int
24329 \exp_after:wN \use_i:nn
24330 \else:
24331 \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
24332 \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
24333 \else:
24334 \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
24335 \fi:
24336 \fi:
24337 }
```

(End definition for `\_regex_if_in_cs:TF`.)

`\_regex_if_in_class_or_catcode:TF` Assertions are only allowed in modes 0, -2, and -6, *i.e.*, even, non-positive modes.

```
24338 \cs_new:Npn _regex_if_in_class_or_catcode:TF
24339 {
24340 \if_int_odd:w \l__regex_mode_int
24341 \exp_after:wN \use_i:nn
24342 \else:
24343 \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
24344 \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
24345 \else:
24346 \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
24347 \fi:
24348 \fi:
24349 }
```

(End definition for `\_regex_if_in_class_or_catcode:TF`.)

`\_regex_if_within_catcode:TF` This test takes the true branch if we are in a catcode test, either immediately following it (modes 2 and 6) or in a class on which it applies (modes 23 and 63). This is used to tweak how left brackets behave in modes 2 and 6.

```
24350 \cs_new:Npn _regex_if_within_catcode:TF
24351 {
24352 \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
24353 \exp_after:wN \use_i:nn
24354 \else:
24355 \exp_after:wN \use_i:nn
24356 \fi:
24357 }
```

(End definition for `\_regex_if_within_catcode:TF`.)

`\_regex_chk_c_allowed:T` The `\c` escape sequence is only allowed in modes 0 and 3, *i.e.*, not within any other `\c` escape sequence.

```
24358 \cs_new_protected:Npn _regex_chk_c_allowed:T
24359 {
24360 \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
```

```

24361 \exp_after:wN \use:n
24362 \else:
24363 \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
24364 \exp_after:wN \exp_after:wN \exp_after:wN \use:n
24365 \else:
24366 __kernel_msg_error:nn { kernel } { c-bad-mode }
24367 \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
24368 \fi:
24369 \fi:
24370 }

```

(End definition for `\__regex_chk_c_allowed:T.`)

`\__regex_mode_quit_c:` This function changes the mode as it is needed just after a catcode test.

```

24371 \cs_new_protected:Npn __regex_mode_quit_c:
24372 {
24373 \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
24374 \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
24375 \else:
24376 \if_int_compare:w \l__regex_mode_int =
24377 \c__regex_catcode_in_class_mode_int
24378 \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
24379 \fi:
24380 \fi:
24381 }

```

(End definition for `\__regex_mode_quit_c:.`)

#### 41.3.4 Framework

`\__regex_compile:w` Used when compiling a user regex or a regex for the `\c{...}` escape sequence within another regex. Start building a token list within a group (with x-expansion at the outset), and set a few variables (group level, catcodes), then start the first branch. At the end, make sure there are no dangling classes nor groups, close the last branch: we are done building `\l__regex_internal_regex`.

```

24382 \cs_new_protected:Npn __regex_compile:w
24383 {
24384 \group_begin:
24385 \tl_build_begin:N \l__regex_build_tl
24386 \int_zero:N \l__regex_group_level_int
24387 \int_set_eq:NN \l__regex_default_catcodes_int
24388 \c__regex_all_catcodes_int
24389 \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
24390 \cs_set:Npn __regex_item_equal:n { __regex_item_caseful_equal:n }
24391 \cs_set:Npn __regex_item_range:nn { __regex_item_caseful_range:nn }
24392 \tl_build_put_right:Nn \l__regex_build_tl
24393 { __regex_branch:n { \if_false: } \fi: }
24394 }
24395 \cs_new_protected:Npn __regex_compile_end:
24396 {
24397 __regex_if_in_class:TF
24398 {
24399 __kernel_msg_error:nn { kernel } { missing-rbrack }
24400 \use:c { __regex_compile_!}: }

```

```

24401 \prg_do_nothing: \prg_do_nothing:
24402 }
24403 { }
24404 \if_int_compare:w \l__regex_group_level_int > 0 \exp_stop_f:
24405 __kernel_msg_error:nnx { kernel } { missing-rparen }
24406 { \int_use:N \l__regex_group_level_int }
24407 \prg_replicate:nn
24408 { \l__regex_group_level_int }
24409 {
24410 \tl_build_put_right:Nn \l__regex_build_tl
24411 {
24412 \if_false: { \fi: }
24413 \if_false: { \fi: } { 1 } { 0 } \c_true_bool
24414 }
24415 \tl_build_end:N \l__regex_build_tl
24416 \exp_args:NNNo
24417 \group_end:
24418 \tl_build_put_right:Nn \l__regex_build_tl
24419 { \l__regex_build_tl }
24420 }
24421 \fi:
24422 \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
24423 \tl_build_end:N \l__regex_build_tl
24424 \exp_args:NNNx
24425 \group_end:
24426 \tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
24427 }

```

(End definition for `\__regex_compile:w` and `\__regex_compile_end:..`)

`\__regex_compile:n` The compilation is done between `\__regex_compile:w` and `\__regex_compile_end:..`, starting in mode 0. Then `\__regex_escape_use:nnnn` distinguishes special characters, escaped alphanumerics, and raw characters, interpreting `\a`, `\x` and other sequences. The 4 trailing `\prg_do_nothing:` are needed because some functions defined later look up to 4 tokens ahead. Before ending, make sure that any `\c{...}` is properly closed. No need to check that brackets are closed properly since `\__regex_compile_end:` does that. However, catch the case of a trailing `\cL` construction.

```

24428 \cs_new_protected:Npn __regex_compile:n #1
24429 {
24430 __regex_compile:w
24431 __regex_standard_escapechar:
24432 \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
24433 __regex_escape_use:nnnn
24434 {
24435 __regex_char_if_special:NTF ##1
24436 __regex_compile_special:N __regex_compile_raw:N ##1
24437 }
24438 {
24439 __regex_char_if_alphanumeric:NTF ##1
24440 __regex_compile_escaped:N __regex_compile_raw:N ##1
24441 }
24442 { __regex_compile_raw:N ##1 }
24443 { ##1 }
24444 \prg_do_nothing: \prg_do_nothing:

```

```

24445 \prg_do_nothing: \prg_do_nothing:
24446 \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
24447 { __kernel_msg_error:nn { kernel } { c-trailing } }
24448 \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
24449 {
24450 __kernel_msg_error:nn { kernel } { c-missing-rbrace }
24451 __regex_compile_end_cs:
24452 \prg_do_nothing: \prg_do_nothing:
24453 \prg_do_nothing: \prg_do_nothing:
24454 }
24455 __regex_compile_end:
24456 }

```

(End definition for `\__regex_compile:n`.)

`\__regex_compile_escaped:N` If the special character or escaped alphanumeric has a particular meaning in regexes, the corresponding function is used. Otherwise, it is interpreted as a raw character. We distinguish special characters from escaped alphanumeric characters because they behave differently when appearing as an end-point of a range.

```

24457 \cs_new_protected:Npn __regex_compile_special:N #1
24458 {
24459 \cs_if_exist_use:cF { __regex_compile:#1: }
24460 { __regex_compile_raw:N #1 }
24461 }
24462 \cs_new_protected:Npn __regex_compile_escaped:N #1
24463 {
24464 \cs_if_exist_use:cF { __regex_compile_/#1: }
24465 { __regex_compile_raw:N #1 }
24466 }

```

(End definition for `\__regex_compile_escaped:N` and `\__regex_compile_special:N`.)

`\__regex_compile_one:n` This is used after finding one “test”, such as `\d`, or a raw character. If that followed a catcode test (*e.g.*, `\cL`), then restore the mode. If we are not in a class, then the test is “standalone”, and we need to add `\__regex_class:NnnnN` and search for quantifiers. In any case, insert the test, possibly together with a catcode test if appropriate.

```

24467 \cs_new_protected:Npn __regex_compile_one:n #1
24468 {
24469 __regex_mode_quit_c:
24470 __regex_if_in_class:TF { }
24471 {
24472 \tl_build_put_right:Nn \l__regex_build_tl
24473 { __regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
24474 }
24475 \tl_build_put_right:Nx \l__regex_build_tl
24476 {
24477 \if_int_compare:w \l__regex_catcodes_int <
24478 \c__regex_all_catcodes_int
24479 __regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
24480 { \exp_not:N \exp_not:n {#1} }
24481 \else:
24482 \exp_not:N \exp_not:n {#1}
24483 \fi:
24484 }

```

```

24485 \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
24486 _regex_if_in_class:TF { } { _regex_compile_quantifier:w }
24487 }

(End definition for _regex_compile_one:n.)
```

\\_regex\_compile\_abort\_tokens:n  
\ regex compile abort tokens:x

This function places the collected tokens back in the input stream, each as a raw character. Spaces are not preserved.

```

24488 \cs_new_protected:Npn _regex_compile_abort_tokens:n #1
24489 {
24490 \use:x
24491 {
24492 \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
24493 _regex_compile_raw:N
24494 }
24495 }
24496 \cs_generate_variant:Nn _regex_compile_abort_tokens:n { x }
```

(End definition for \\_regex\_compile\_abort\_tokens:n.)

#### 41.3.5 Quantifiers

\\_regex\_compile\_quantifier:w

```

24497 \cs_new_protected:Npn _regex_compile_quantifier:w #1#2
24498 {
24499 \token_if_eq_meaning:NNTF #1 _regex_compile_special:N
24500 {
24501 \cs_if_exist_use:cF { __regex_compile_quantifier_#2:w }
24502 { _regex_compile_quantifier_none: #1 #2 }
24503 }
24504 { _regex_compile_quantifier_none: #1 #2 }
24505 }
```

(End definition for \\_regex\_compile\_quantifier:w.)

\\_regex\_compile\_quantifier\_none:  
\ regex compile quantifier\_abort:xNN

```

24506 \cs_new_protected:Npn _regex_compile_quantifier_none:
24507 {
24508 \tl_build_put_right:Nn \l__regex_build_tl
24509 { \if_false: { \fi: } { 1 } { 0 } \c_false_bool }
24510 }
24511 \cs_new_protected:Npn _regex_compile_quantifier_abort:xNN #1#2#3
24512 {
24513 _regex_compile_quantifier_none:
24514 _kernel_msg_warning:nnxx { kernel } { invalid-quantifier } {#1} {#3}
24515 _regex_compile_abort_tokens:x {#1}
24516 #2 #3
24517 }
```

(End definition for \\_regex\_compile\_quantifier\_none: and \\_regex\_compile\_quantifier\_abort:xNN.)

```
_regex_compile_quantifier_lazyness:nnNN
```

Once the “main” quantifier (?\*, + or a braced construction) is found, we check whether it is lazy (followed by a question mark). We then add to the compiled regex a closing brace (ending `\_regex_class:NnnnN` and friends), the start-point of the range, its end-point, and a boolean, true for lazy and false for greedy operators.

```
24518 \cs_new_protected:Npn _regex_compile_quantifier_lazyness:nnNN #1#2#3#4
24519 {
24520 _regex_two_if_eq:NNNNTF #3 #4 _regex_compile_special:N ?
24521 {
24522 \tl_build_put_right:Nn \l__regex_build_tl
24523 { \if_false: { \fi: } { #1 } { #2 } \c_true_bool }
24524 }
24525 {
24526 \tl_build_put_right:Nn \l__regex_build_tl
24527 { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
24528 #3 #4
24529 }
24530 }
```

(End definition for `\_regex_compile_quantifier_lazyness:nnNN`.)

```
_regex_compile_quantifier?:w
_regex_compile_quantifier*:w
_regex_compile_quantifier+:w
```

For each “basic” quantifier, ?, \*, +, feed the correct arguments to `\_regex_compile-quantifier_lazyness:nnNN`, -1 means that there is no upper bound on the number of repetitions.

```
24531 \cs_new_protected:cpn { __regex_compile_quantifier?:w }
24532 { _regex_compile_quantifier_lazyness:nnNN { 0 } { 1 } }
24533 \cs_new_protected:cpn { __regex_compile_quantifier*:w }
24534 { _regex_compile_quantifier_lazyness:nnNN { 0 } { -1 } }
24535 \cs_new_protected:cpn { __regex_compile_quantifier+:w }
24536 { _regex_compile_quantifier_lazyness:nnNN { 1 } { -1 } }
```

(End definition for `\_regex_compile_quantifier?:w`, `\_regex_compile_quantifier*:w`, and `\_regex_compile_quantifier+:w`.)

```
_regex_compile_quantifier{:w
_regex_compile_quantifier_braced_auxi:w
_regex_compile_quantifier_braced_auxii:w
_regex_compile_quantifier_braced_auxiii:w}
```

Three possible syntaxes: `{(int)}`, `{(int),}`, or `{(int),(int)}`. Any other syntax causes us to abort and put whatever we collected back in the input stream, as raw characters, including the opening brace. Grab a number into `\l__regex_internal_a_int`. If the number is followed by a right brace, the range is  $[a, a]$ . If followed by a comma, grab one more number, and call the `_ii` or `_iii` auxiliaries. Those auxiliaries check for a closing brace, leading to the range  $[a, \infty]$  or  $[a, b]$ , encoded as  $\{a\}\{-1\}$  and  $\{a\}\{b - a\}$ .

```
24537 \cs_new_protected:cpn { __regex_compile_quantifier_ \c_left_brace_str :w }
24538 {
24539 _regex_get_digits:NTFw \l__regex_internal_a_int
24540 { _regex_compile_quantifier_braced_auxi:w }
24541 { _regex_compile_quantifier_abort:xNN { \c_left_brace_str } }
24542 }
24543 \cs_new_protected:Npn _regex_compile_quantifier_braced_auxi:w #1#2
24544 {
24545 \str_case_e:nnF { #1 #2 }
24546 {
24547 { _regex_compile_special:N \c_right_brace_str }
24548 {
24549 \exp_args:No _regex_compile_quantifier_lazyness:nnNN
24550 { \int_use:N \l__regex_internal_a_int } { 0 }
24551 }
24552 }
```

```

24552 { __regex_compile_special:N , }
24553 {
24554 __regex_get_digits:NTFw \l__regex_internal_b_int
24555 { __regex_compile_quantifier_braced_auxiii:w }
24556 { __regex_compile_quantifier_braced_auxii:w }
24557 }
24558 }
24559 {
24560 __regex_compile_quantifier_abort:xNN
24561 { \c_left brace str \int_use:N \l__regex_internal_a_int }
24562 #1 #2
24563 }
24564 }
24565 \cs_new_protected:Npn __regex_compile_quantifier_braced_auxii:w #1#2
24566 {
24567 __regex_two_if_eq:NNNNTF #1 #2 __regex_compile_special:N \c_right brace str
24568 {
24569 \exp_args:No __regex_compile_quantifier_lazyness:nnNN
24570 { \int_use:N \l__regex_internal_a_int } { -1 }
24571 }
24572 {
24573 __regex_compile_quantifier_abort:xNN
24574 { \c_left brace str \int_use:N \l__regex_internal_a_int , }
24575 #1 #2
24576 }
24577 }
24578 \cs_new_protected:Npn __regex_compile_quantifier_braced_auxiii:w #1#2
24579 {
24580 __regex_two_if_eq:NNNNTF #1 #2 __regex_compile_special:N \c_right brace str
24581 {
24582 \if_int_compare:w \l__regex_internal_a_int >
24583 \l__regex_internal_b_int
24584 __kernel_msg_error:nxxx { kernel } { backwards-quantifier }
24585 { \int_use:N \l__regex_internal_a_int }
24586 { \int_use:N \l__regex_internal_b_int }
24587 \int_zero:N \l__regex_internal_b_int
24588 \else:
24589 \int_sub:Nn \l__regex_internal_b_int \l__regex_internal_a_int
24590 \fi:
24591 \exp_args:Noo __regex_compile_quantifier_lazyness:nnNN
24592 { \int_use:N \l__regex_internal_a_int }
24593 { \int_use:N \l__regex_internal_b_int }
24594 }
24595 {
24596 __regex_compile_quantifier_abort:xNN
24597 {
24598 \c_left brace str
24599 \int_use:N \l__regex_internal_a_int ,
24600 \int_use:N \l__regex_internal_b_int
24601 }
24602 #1 #2
24603 }
24604 }

```

(End definition for `\__regex_compile_quantifier_f:w` and others.)

#### 41.3.6 Raw characters

`\_\_regex_compile_raw_error:N`

Within character classes, and following catcode tests, some escaped alphanumeric sequences such as `\b` do not have any meaning. They are replaced by a raw character, after spitting out an error.

```
24605 \cs_new_protected:Npn __regex_compile_raw_error:N #1
24606 {
24607 __kernel_msg_error:nnx { kernel } { bad-escape } {#1}
24608 __regex_compile_raw:N #1
24609 }
```

(*End definition for `\_\_regex_compile_raw_error:N`.*)

`\_\_regex_compile_raw:N`

If we are in a character class and the next character is an unescaped dash, this denotes a range. Otherwise, the current character #1 matches itself.

```
24610 \cs_new_protected:Npn __regex_compile_raw:N #1#2#3
24611 {
24612 __regex_if_in_class:TF
24613 {
24614 __regex_two_if_eq:NNNTF #2 #3 __regex_compile_special:N -
24615 { __regex_compile_range:Nw #1 }
24616 {
24617 __regex_compile_one:n
24618 { __regex_item_equal:n { \int_value:w '#1 } }
24619 #2 #3
24620 }
24621 }
24622 {
24623 __regex_compile_one:n
24624 { __regex_item_equal:n { \int_value:w '#1 } }
24625 #2 #3
24626 }
24627 }
```

(*End definition for `\_\_regex_compile_raw:N`.*)

`\_\_regex_compile_range:Nw`  
`\_\_regex_if_end_range:NNTF`

We have just read a raw character followed by a dash; this should be followed by an end-point for the range. Valid end-points are: any raw character; any special character, except a right bracket. In particular, escaped characters are forbidden.

```
24628 \prg_new_protected_conditional:Npnn __regex_if_end_range:NN #1#2 { TF }
24629 {
24630 \if_meaning:w __regex_compile_raw:N #1
24631 \prg_return_true:
24632 \else:
24633 \if_meaning:w __regex_compile_special:N #1
24634 \if_charcode:w] #2
24635 \prg_return_false:
24636 \else:
24637 \prg_return_true:
24638 \fi:
24639 \else:
24640 \prg_return_false:
24641 \fi:
24642 }
```

```

24643 }
24644 \cs_new_protected:Npn __regex_compile_range:Nw #1#2#3
24645 {
24646 __regex_if_end_range:NNTF #2 #3
24647 {
24648 \if_int_compare:w '#1 > '#3 \exp_stop_f:
24649 __kernel_msg_error:nxxx { kernel } { range-backwards } {#1} {#3}
24650 \else:
24651 \tl_build_put_right:Nx \l__regex_build_tl
24652 {
24653 \if_int_compare:w '#1 = '#3 \exp_stop_f:
24654 __regex_item_equal:n
24655 \else:
24656 __regex_item_range:nn { \int_value:w '#1 }
24657 \fi:
24658 { \int_value:w '#3 }
24659 }
24660 \fi:
24661 }
24662 {
24663 __kernel_msg_warning:nxxx { kernel } { range-missing-end }
24664 {#1} { \c_backslash_str #3 }
24665 \tl_build_put_right:Nx \l__regex_build_tl
24666 {
24667 __regex_item_equal:n { \int_value:w '#1 \exp_stop_f: }
24668 __regex_item_equal:n { \int_value:w '- \exp_stop_f: }
24669 }
24670 #2#3
24671 }
24672 }

```

(End definition for `\__regex_compile_range:Nw` and `\__regex_if_end_range:NNTF`.)

#### 41.3.7 Character properties

`\__regex_compile_..:`  
`\__regex_prop_..:`

In a class, the dot has no special meaning. Outside, insert `\__regex_prop_..:`, which matches any character or control sequence, and refuses -2 (end-marker).

```

24673 \cs_new_protected:cpx { __regex_compile_..: }
24674 {
24675 \exp_not:N __regex_if_in_class:TF
24676 { __regex_compile_raw:N . }
24677 { __regex_compile_one:n \exp_not:c { __regex_prop_..: } }
24678 }
24679 \cs_new_protected:cpn { __regex_prop_..: }
24680 {
24681 \if_int_compare:w \l__regex_curr_char_int > - 2 \exp_stop_f:
24682 \exp_after:wN __regex_break_true:w
24683 \fi:
24684 }

```

(End definition for `\__regex_compile_..:` and `\__regex_prop_..:`)

`\__regex_compile_d:`  
`\__regex_compile_D:`  
`\__regex_compile_h:`  
`\__regex_compile_H:`  
`\__regex_compile_s:`  
`\__regex_compile_S:`  
`\__regex_compile_v:`  
`\__regex_compile_V:`  
`\__regex_compile_w:`  
`\__regex_compile_W:`  
`\__regex_compile_N:`

The constants `\__regex_prop_d:`, etc. hold a list of tests which match the corresponding character class, and jump to the `\__regex_break_point:TF` marker. As for a normal character, we check for quantifiers.

```

24685 \cs_set_protected:Npn __regex_tmp:w #1#2
24686 {
24687 \cs_new_protected:cpx { __regex_compile_/#1: }
24688 { __regex_compile_one:n \exp_not:c { __regex_prop_#1: } }
24689 \cs_new_protected:cpx { __regex_compile_/#2: }
24690 {
24691 __regex_compile_one:n
24692 { __regex_item_reverse:n \exp_not:c { __regex_prop_#1: } }
24693 }
24694 }
24695 __regex_tmp:w d D
24696 __regex_tmp:w h H
24697 __regex_tmp:w s S
24698 __regex_tmp:w v V
24699 __regex_tmp:w w W
24700 \cs_new_protected:cpn { __regex_compile_/_N: }
24701 { __regex_compile_one:n __regex_prop_N: }

(End definition for __regex_compile_/d: and others.)

```

#### 41.3.8 Anchoring and simple assertions

\\_\_regex\_compile\_anchor:NF  
\\_\_regex\_compile\_~:  
\\_\_regex\_compile/\_A:  
\\_\_regex\_compile/\_G:  
\\_\_regex\_compile/\$:  
\\_\_regex\_compile/\_Z:  
\\_\_regex\_compile/\_z:  
In modes where assertions are allowed, anchor to the start of the query, the start of the match, or the end of the query, depending on the integer #1. In other modes, #2 treats the character as raw, with an error for escaped letters (\$ is valid in a class, but \A is definitely a mistake on the user's part).

```

24702 \cs_new_protected:Npn __regex_compile_anchor:NF #1#2
24703 {
24704 __regex_if_in_class_or_catcode:TF {#2}
24705 {
24706 \tl_build_put_right:Nn \l__regex_build_tl
24707 { __regex_assertion:Nn \c_true_bool { __regex_anchor:N #1 } }
24708 }
24709 }
24710 \cs_set_protected:Npn __regex_tmp:w #1#2
24711 {
24712 \cs_new_protected:cpn { __regex_compile_/#1: }
24713 { __regex_compile_anchor:NF #2 { __regex_compile_raw_error:N #1 } }
24714 }
24715 __regex_tmp:w A \l__regex_min_pos_int
24716 __regex_tmp:w G \l__regex_start_pos_int
24717 __regex_tmp:w Z \l__regex_max_pos_int
24718 __regex_tmp:w z \l__regex_max_pos_int
24719 \cs_set_protected:Npn __regex_tmp:w #1#2
24720 {
24721 \cs_new_protected:cpn { __regex_compile_/#1: }
24722 { __regex_compile_anchor:NF #2 { __regex_compile_raw:N #1 } }
24723 }
24724 \exp_args:Nx __regex_tmp:w { \iow_char:N ` } \l__regex_min_pos_int
24725 \exp_args:Nx __regex_tmp:w { \iow_char:N $ } \l__regex_max_pos_int

(End definition for __regex_compile_anchor:NF and others.)

```

\\_\\_regex\\_compile\\_b: Contrarily to ^ and \$, which could be implemented without really knowing what precedes in the token list, this requires more information, namely, the knowledge of the last character code.

```

24726 \cs_new_protected:cpn { __regex_compile/_b: }
24727 {
24728 __regex_if_in_class_or_catcode:TF
24729 { __regex_compile_raw_error:N b }
24730 {
24731 \tl_build_put_right:Nn \l__regex_build_tl
24732 { __regex_assertion:Nn \c_true_bool { __regex_b_test: } }
24733 }
24734 }
24735 \cs_new_protected:cpn { __regex_compile/_B: }
24736 {
24737 __regex_if_in_class_or_catcode:TF
24738 { __regex_compile_raw_error:N B }
24739 {
24740 \tl_build_put_right:Nn \l__regex_build_tl
24741 { __regex_assertion:Nn \c_false_bool { __regex_b_test: } }
24742 }
24743 }

```

(End definition for \\_\\_regex\\_compile\\_b: and \\_\\_regex\\_compile\\_B::)

#### 41.3.9 Character classes

\\_\\_regex\_compile\\_[: Outside a class, right brackets have no meaning. In a class, change the mode ( $m \rightarrow (m - 15)/13$ , truncated) to reflect the fact that we are leaving the class. Look for quantifiers, unless we are still in a class after leaving one (the case of [...] \cL [...] ...]). quantifiers.

```

24744 \cs_new_protected:cpn { __regex_compile_[: }
24745 {
24746 __regex_if_in_class:TF
24747 {
24748 \if_int_compare:w \l__regex_mode_int >
24749 \c__regex_catcode_in_class_mode_int
24750 \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
24751 \fi:
24752 \tex_advance:D \l__regex_mode_int - 15 \exp_stop_f:
24753 \tex_divide:D \l__regex_mode_int 13 \exp_stop_f:
24754 \if_int_odd:w \l__regex_mode_int \else:
24755 \exp_after:wN __regex_compile_quantifier:w
24756 \fi:
24757 }
24758 { __regex_compile_raw:N } }
24759 }

```

(End definition for \\_\\_regex\\_compile\\_[:]::)

\\_\\_regex\_compile\\_[: In a class, left brackets might introduce a POSIX character class, or mean nothing. Immediately following \c(category), we must insert the appropriate catcode test, then parse the class; we pre-expand the catcode as an optimization. Otherwise (modes 0, -2 and -6) just parse the class. The mode is updated later.

```
24760 \cs_new_protected:cpn { __regex_compile_[: }
```

```

24761 {
24762 __regex_if_in_class:TF
24763 { __regex_compile_class_posix_test:w }
24764 {
24765 __regex_if_within_catcode:TF
24766 {
24767 \exp_after:wN __regex_compile_class_catcode:w
24768 \int_use:N \l__regex_catcodes_int ;
24769 }
24770 { __regex_compile_class_normal:w }
24771 }
24772 }

```

(End definition for \\_\\_regex\\_compile\\_[:])

\\_\\_regex\\_compile\\_class\\_normal:w In the “normal” case, we insert \\_\\_regex\\_class:NnnnN *<boolean>* in the compiled code. The *<boolean>* is true for positive classes, and false for negative classes, characterized by a leading ^. The auxiliary \\_\\_regex\\_compile\\_class:TFNN also checks for a leading ] which has a special meaning.

```

24773 \cs_new_protected:Npn __regex_compile_class_normal:w
24774 {
24775 __regex_compile_class:TFNN
24776 { __regex_class:NnnnN \c_true_bool }
24777 { __regex_class:NnnnN \c_false_bool }
24778 }

```

(End definition for \\_\\_regex\\_compile\\_class\\_normal:w.)

\\_\\_regex\\_compile\\_class\\_catcode:w This function is called for a left bracket in modes 2 or 6 (catcode test, and catcode test within a class). In mode 2 the whole construction needs to be put in a class (like single character). Then determine if the class is positive or negative, inserting \\_\\_regex\\_item\\_catcode:nT or the reverse variant as appropriate, each with the current catcodes bitmap #1 as an argument, and reset the catcodes.

```

24779 \cs_new_protected:Npn __regex_compile_class_catcode:w #1;
24780 {
24781 \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
24782 \tl_build_put_right:Nn \l__regex_build_tl
24783 { __regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
24784 \fi:
24785 \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
24786 __regex_compile_class:TFNN
24787 { __regex_item_catcode:nT {#1} }
24788 { __regex_item_catcode_reverse:nT {#1} }
24789 }

```

(End definition for \\_\\_regex\\_compile\\_class\\_catcode:w.)

\\_\\_regex\\_compile\\_class:TFNN If the first character is ^, then the class is negative (use #2), otherwise it is positive (use #1). If the next character is a right bracket, then it should be changed to a raw one.

```

24790 \cs_new_protected:Npn __regex_compile_class:TFNN #1#2#3#4
24791 {
24792 \l__regex_mode_int = \int_value:w \l__regex_mode_int 3 \exp_stop_f:
24793 __regex_two_if_eq:NNNNTF #3 #4 __regex_compile_special:N ^
24794 {

```

```

24795 \tl_build_put_right:Nn \l__regex_build_tl { #2 { \if_false: } \fi: }
24796 __regex_compile_class>NN
24797 }
24798 {
24799 \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
24800 __regex_compile_class>NN #3 #4
24801 }
24802 }
24803 \cs_new_protected:Npn __regex_compile_class>NN #1#2
24804 {
24805 \token_if_eq_charcode:NNTF #2 [
24806 { __regex_compile_raw:N #2 }
24807 { #1 #2 }
24808 }

```

(End definition for `\__regex_compile_class:TFNN` and `\__regex_compile_class>NN`.)

`\__regex_compile_class_posix_test:w`  
`\__regex_compile_class_posix>NNNNw`  
`\__regex_compile_class_posix_loop:w`  
`\__regex_compile_class_posix_end:w`

Here we check for a syntax such as `[:alpha:]`. We also detect `[=` and `[.` which have a meaning in POSIX regular expressions, but are not implemented in `\l3regex`. In case we see `[:`, grab raw characters until hopefully reaching `:]`. If that's missing, or the POSIX class is unknown, abort. If all is right, add the test to the current class, with an extra `\__regex_item_reverse:n` for negative classes.

```

24809 \cs_new_protected:Npn __regex_compile_class_posix_test:w #1#2
24810 {
24811 \token_if_eq_meaning:NNT __regex_compile_special:N #1
24812 {
24813 \str_case:nn { #2 }
24814 {
24815 : { __regex_compile_class_posix>NNNNw }
24816 = {
24817 __kernel_msg_warning:nnx { kernel }
24818 { posix-unsupported } { = }
24819 }
24820 .
24821 __kernel_msg_warning:nnx { kernel }
24822 { posix-unsupported } { . }
24823 }
24824 }
24825 }
24826 __regex_compile_raw:N [#1 #2
24827 }
24828 \cs_new_protected:Npn __regex_compile_class_posix>NNNNw #1#2#3#4#5#6
24829 {
24830 __regex_two_if_eq>NNNNTF #5 #6 __regex_compile_special:N ^
24831 {
24832 \bool_set_false:N \l__regex_internal_bool
24833 __kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
24834 __regex_compile_class_posix_loop:w
24835 }
24836 {
24837 \bool_set_true:N \l__regex_internal_bool
24838 __kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
24839 __regex_compile_class_posix_loop:w #5 #6
24840 }

```

```

24841 }
24842 \cs_new:Npn __regex_compile_class_posix_loop:w #1#2
24843 {
24844 \token_if_eq_meaning:NNTF __regex_compile_raw:N #1
24845 { #2 __regex_compile_class_posix_loop:w }
24846 { \if_false: { \fi: } __regex_compile_class_posix_end:w #1 #2 }
24847 }
24848 \cs_new_protected:Npn __regex_compile_class_posix_end:w #1#2#3#4
24849 {
24850 __regex_two_if_eq>NNNNTF #1 #2 __regex_compile_special:N :
24851 { __regex_two_if_eq>NNNNTF #3 #4 __regex_compile_special:N] }
24852 { \use_ii:nn }
24853 {
24854 \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
24855 {
24856 __regex_compile_one:n
24857 {
24858 \bool_if:NF \l__regex_internal_bool __regex_item_reverse:n
24859 \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : }
24860 }
24861 }
24862 {
24863 __kernel_msg_warning:nnx { kernel } { posix-unknown }
24864 { \l__regex_internal_a_tl }
24865 __regex_compile_abort_tokens:x
24866 {
24867 [: \bool_if:NF \l__regex_internal_bool { ^ }]
24868 \l__regex_internal_a_tl :]
24869 }
24870 }
24871 }
24872 {
24873 __kernel_msg_error:nnxx { kernel } { posix-missing-close }
24874 { [: \l__regex_internal_a_tl] { #2 #4 }
24875 __regex_compile_abort_tokens:x { [: \l__regex_internal_a_tl] }
24876 #1 #2 #3 #4
24877 }
24878 }

```

(End definition for `\__regex_compile_class_posix_test:w` and others.)

#### 41.3.10 Groups and alternations

The contents of a regex group are turned into compiled code in `\l__regex_build_t1`, which ends up with items of the form `\__regex_branch:n {<concatenation>}`. This construction is done using `\t1_build_...` functions within a T<sub>E</sub>X group, which automatically makes sure that options (case-sensitivity and default catcode) are reset at the end of the group. The argument #1 is `\__regex_group:nnN` or a variant thereof. A small subtlety to support `\cL(abc)` as a shorthand for `(\cLa\cLb\cLc)`: exit any pending catcode test, save the category code at the start of the group as the default catcode for that group, and make sure that the catcode is restored to the default outside the group.

```
24879 \cs_new_protected:Npn __regex_compile_group_begin:N #1
```

```

24880 {
24881 \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
24882 __regex_mode_quit_c:
24883 \group_begin:
24884 \tl_build_begin:N \l__regex_build_tl
24885 \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
24886 \int_incr:N \l__regex_group_level_int
24887 \tl_build_put_right:Nn \l__regex_build_tl
24888 { __regex_branch:n { \if_false: } \fi: }
24889 }
24890 \cs_new_protected:Npn __regex_compile_group_end:
24891 {
24892 \if_int_compare:w \l__regex_group_level_int > 0 \exp_stop_f:
24893 \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
24894 \tl_build_end:N \l__regex_build_tl
24895 \exp_args:NNNx
24896 \group_end:
24897 \tl_build_put_right:Nn \l__regex_build_tl { \l__regex_build_tl }
24898 \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
24899 \exp_after:wN __regex_compile_quantifier:w
24900 \else:
24901 _kernel_msg_warning:nn { kernel } { extra-rparen }
24902 \exp_after:wN __regex_compile_raw:N \exp_after:wN)
24903 \fi:
24904 }

```

(End definition for `\__regex_compile_group_begin:N` and `\__regex_compile_group_end:..`)

`\__regex_compile_()`: In a class, parentheses are not special. In a catcode test inside a class, a left parenthesis gives an error, to catch `[a\cL(bcd)e]`. Otherwise check for a ?, denoting special groups, and run the code for the corresponding special group.

```

24905 \cs_new_protected:cpn { __regex_compile_(: }
24906 {
24907 __regex_if_in_class:TF { __regex_compile_raw:N (}
24908 {
24909 \if_int_compare:w \l__regex_mode_int =
24910 \c__regex_catcode_in_class_mode_int
24911 __kernel_msg_error:nn { kernel } { c-lparen-in-class }
24912 \exp_after:wN __regex_compile_raw:N \exp_after:wN (
24913 \else:
24914 \exp_after:wN __regex_compile_lparen:w
24915 \fi:
24916 }
24917 }
24918 \cs_new_protected:Npn __regex_compile_lparen:w #1#2#3#4
24919 {
24920 __regex_two_if_eq:NNNTF #1 #2 __regex_compile_special:N ?
24921 {
24922 \cs_if_exist_use:cF
24923 { __regex_compile_special_group_\token_to_str:N #4 :w }
24924 {
24925 __kernel_msg_warning:nnx { kernel } { special-group-unknown }
24926 { (? #4 }
24927 __regex_compile_group_begin:N __regex_group:nnN

```

```

24928 __regex_compile_raw:N ? #3 #4
24929 }
24930 }
24931 {
24932 __regex_compile_group_begin:N __regex_group:nnnN
24933 #1 #2 #3 #4
24934 }
24935 }
```

(End definition for `\_\_regex\_compile_(..)`)

`\_\_regex\_compile_|:` In a class, the pipe is not special. Otherwise, end the current branch and open another one.

```

24936 \cs_new_protected:cpn { __regex_compile_|: }
24937 {
24938 __regex_if_in_class:TF { __regex_compile_raw:N | }
24939 {
24940 \tl_build_put_right:Nn \l__regex_build_tl
24941 { \if_false: { \fi: } __regex_branch:n { \if_false: } \fi: }
24942 }
24943 }
```

(End definition for `\_\_regex\_compile_|(..)`)

`\_\_regex\_compile_()`: Within a class, parentheses are not special. Outside, close a group.

```

24944 \cs_new_protected:cpn { __regex_compile_(): }
24945 {
24946 __regex_if_in_class:TF { __regex_compile_raw:N() }
24947 { __regex_compile_group_end: }
24948 }
```

(End definition for `\_\_regex\_compile_()..`)

`\_\_regex_compile_special_group_::w`  
`\_\_regex_compile_special_group_|:w`

Non-capturing, and resetting groups are easy to take care of during compilation; for those groups, the harder parts come when building.

```

24949 \cs_new_protected:cpn { __regex_compile_special_group_::w }
24950 { __regex_compile_group_begin:N __regex_group_no_capture:nnnN }
24951 \cs_new_protected:cpn { __regex_compile_special_group_|:w }
24952 { __regex_compile_group_begin:N __regex_group_resetting:nnnN }
```

(End definition for `\_\_regex_compile_special_group_::w` and `\_\_regex_compile_special_group_|:w`)

`\_\_regex_compile_special_group_i:w`  
`\_\_regex_compile_special_group_-:w`

The match can be made case-insensitive by setting the option with `(?i)`; the original behaviour is restored by `(?-i)`. This is the only supported option.

```

24953 \cs_new_protected:Npn __regex_compile_special_group_i:w #1#2
24954 {
24955 __regex_two_if_eq>NNNNTF #1 #2 __regex_compile_special:N)
24956 {
24957 \cs_set:Npn __regex_item_equal:n
24958 { __regex_item_caseless_equal:n }
24959 \cs_set:Npn __regex_item_range:nn
24960 { __regex_item_caseless_range:nn }
24961 }
24962 {
24963 __kernel_msg_warning:nnx { kernel } { unknown-option } { (?i #2 }
```

```

24964 __regex_compile_raw:N (
24965 __regex_compile_raw:N ?
24966 __regex_compile_raw:N i
24967 #1 #2
24968 }
24969 }
24970 \cs_new_protected:cpn { __regex_compile_special_group_-=w } #1#2#3#4
24971 {
24972 __regex_two_if_eq>NNNNTF #1 #2 __regex_compile_raw:N i
24973 { __regex_two_if_eq>NNNNTF #3 #4 __regex_compile_special:N) }
24974 { \use_ii:nn }
24975 {
24976 \cs_set:Npn __regex_item_equal:n
24977 { __regex_item_caseful_equal:n }
24978 \cs_set:Npn __regex_item_range:nn
24979 { __regex_item_caseful_range:nn }
24980 }
24981 {
24982 __kernel_msg_warning:nnx { kernel } { unknown-option } { (?-#2#4 }
24983 __regex_compile_raw:N (
24984 __regex_compile_raw:N ?
24985 __regex_compile_raw:N -
24986 #1 #2 #3 #4
24987 }
24988 }

```

(End definition for `\__regex_compile_special_group_i:w` and `\__regex_compile_special_group_-=w`.)

#### 41.3.11 Catcodes and csnames

`\__regex_compile/_c:` The `\c` escape sequence can be followed by a capital letter representing a character category, by a left bracket which starts a list of categories, or by a brace group holding a regular expression for a control sequence name. Otherwise, raise an error.

```

24989 \cs_new_protected:cpn { __regex_compile/_c: }
24990 { __regex_chk_c_allowed:T { __regex_compile_c_test:NN } }
24991 \cs_new_protected:Npn __regex_compile_c_test:NN #1#2
24992 {
24993 \token_if_eq_meaning:NNTF #1 __regex_compile_raw:N
24994 {
24995 \int_if_exist:cTF { c__regex_catcode_#2_int }
24996 {
24997 \int_set_eq:Nc \l__regex_catcodes_int
24998 { c__regex_catcode_#2_int }
24999 \l__regex_mode_int
25000 = \if_case:w \l__regex_mode_int
25001 \c__regex_catcode_mode_int
25002 \else:
25003 \c__regex_catcode_in_class_mode_int
25004 \fi:
25005 \token_if_eq_charcode:NNT C #2 { __regex_compile_c_C:NN }
25006 }
25007 }
25008 { \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
25009 {

```

```

25010 __kernel_msg_error:nnx { kernel } { c-missing-category } {#2}
25011 #1 #2
25012 }
25013 }
```

(End definition for `\__regex_compile_c:/c:` and `\__regex_compile_c_test>NN.`)

`\__regex_compile_c_C>NN` If `\cC` is not followed by `.` or `(...)` then complain because that construction cannot match anything, except in cases like `\cC[\c{...}]`, where it has no effect.

```

25014 \cs_new_protected:Npn __regex_compile_c_C:NN #1#2
25015 {
25016 \token_if_eq_meaning:NNTF #1 __regex_compile_special:N
25017 {
25018 \token_if_eq_charcode:NNTF #2 .
25019 { \use_none:n }
25020 { \token_if_eq_charcode:NNF #2 () % }
25021 }
25022 { \use:n }
25023 { __kernel_msg_error:nnn { kernel } { c-C-invalid } {#2} }
25024 #1 #2
25025 }
```

(End definition for `\__regex_compile_c_C>NN.`)

`\__regex_compile_c_[:w` `\__regex_compile_c_lbrack_loop:N` When encountering `\c[`, the task is to collect uppercase letters representing character categories. First check for `^` which negates the list of category codes.

```

25026 \cs_new_protected:cpx { __regex_compile_c_[:w } #1#2
25027 {
25028 \l__regex_mode_int
25029 = \if_case:w \l__regex_mode_int
25030 \c__regex_catcode_mode_int
25031 \else:
25032 \c__regex_catcode_in_class_mode_int
25033 \fi:
25034 \int_zero:N \l__regex_catcodes_int
25035 __regex_two_if_eq:NNNNTF #1 #2 __regex_compile_special:N ^
25036 {
25037 \bool_set_false:N \l__regex_catcodes_bool
25038 __regex_compile_c_lbrack_loop:NN
25039 }
25040 {
25041 \bool_set_true:N \l__regex_catcodes_bool
25042 __regex_compile_c_lbrack_loop:NN
25043 #1 #2
25044 }
25045 }
```

`\cs_new_protected:Npn \__regex_compile_c_lbrack_loop:NN #1#2`

```

25046 {
25047 \token_if_eq_meaning:NNTF #1 __regex_compile_raw:N
25048 {
25049 \int_if_exist:cTF { c__regex_catcode_#2_int }
25050 {
25051 \exp_args:Nc __regex_compile_c_lbrack_add:N
25052 { c__regex_catcode_#2_int }
```

```

25054 __regex_compile_c_lbrack_loop:NN
25055 }
25056 }
25057 {
25058 \token_if_eq_charcode:NNTF #2]
25059 { __regex_compile_c_lbrack_end: }
25060 }
25061 {
25062 __kernel_msg_error:nnx { kernel } { c-missing-rbrack } {#2}
25063 __regex_compile_c_lbrack_end:
25064 #1 #2
25065 }
25066 }
25067 \cs_new_protected:Npn __regex_compile_c_lbrack_add:N #1
25068 {
25069 \if_int_odd:w \int_eval:n { \l__regex_catcodes_int / #1 } \exp_stop_f:
25070 \else:
25071 \int_add:Nn \l__regex_catcodes_int {#1}
25072 \fi:
25073 }
25074 \cs_new_protected:Npn __regex_compile_c_lbrack_end:
25075 {
25076 \if_meaning:w \c_false_bool \l__regex_catcodes_bool
25077 \int_set:Nn \l__regex_catcodes_int
25078 { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
25079 \fi:
25080 }

(End definition for __regex_compile_c_[:w and others.)
```

\\_\_regex\_compile\_c\_{:}: The case of a left brace is easy, based on what we have done so far: in a group, compile the regular expression, after changing the mode to forbid nesting \c. Additionally, disable submatch tracking since groups don't escape the scope of \c{...}.

```

25081 \cs_new_protected:cpn { __regex_compile_c_ \c_left_brace_str :w }
25082 {
25083 __regex_compile:w
25084 __regex_disable_submatches:
25085 \l__regex_mode_int
25086 = \if_case:w \l__regex_mode_int
25087 \c__regex_cs_mode_int
25088 \else:
25089 \c__regex_cs_in_class_mode_int
25090 \fi:
25091 }
```

(End definition for \\_\_regex\_compile\_c\_{:})

\\_\_regex\_compile\_{:}: Non-escaped right braces are only special if they appear when compiling the regular expression for a csname, but not within a class: \c{[{}]} matches the control sequences \{ and \}. So, end compiling the inner regex (this closes any dangling class or group). Then insert the corresponding test in the outer regex. As an optimization, if the control sequence test simply consists of several explicit possibilities (branches) then use \\_\_regex\_item\_exact\_cs:n with an argument consisting of all possibilities separated by \scan\_stop:.

```

25092 \flag_new:n { __regex_cs }
25093 \cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
25094 {
25095 __regex_if_in_cs:TF
25096 { __regex_compile_end_cs: }
25097 { \exp_after:wN __regex_compile_raw:N \c_right_brace_str }
25098 }
25099 \cs_new_protected:Npn __regex_compile_end_cs:
25100 {
25101 __regex_compile_end:
25102 \flag_clear:n { __regex_cs }
25103 __kernel_tl_set:Nx \l__regex_internal_a_tl
25104 {
25105 \exp_after:wN __regex_compile_cs_aux:Nn \l__regex_internal_regex
25106 \q__regex_nil \q__regex_nil \q__regex_recursion_stop
25107 }
25108 \exp_args:Nx __regex_compile_one:n
25109 {
25110 \flag_if_raised:nTF { __regex_cs }
25111 { __regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
25112 {
25113 __regex_item_exact_cs:n
25114 { \tl_tail:N \l__regex_internal_a_tl }
25115 }
25116 }
25117 }
25118 \cs_new:Npn __regex_compile_cs_aux:Nn #1#2
25119 {
25120 \cs_if_eq:NNTF #1 __regex_branch:n
25121 {
25122 \scan_stop:
25123 __regex_compile_cs_aux:NNnnnN #2
25124 \q__regex_nil \q__regex_nil \q__regex_nil
25125 \q__regex_nil \q__regex_nil \q__regex_nil \q__regex_recursion_stop
25126 __regex_compile_cs_aux:Nn
25127 }
25128 {
25129 __regex_quark_if_nil:NF #1 { \flag_raise_if_clear:n { __regex_cs } }
25130 __regex_use_none_delimit_by_q_recursion_stop:w
25131 }
25132 }
25133 \cs_new:Npn __regex_compile_cs_aux:NNnnnN #1#2#3#4#5#6
25134 {
25135 \bool_lazy_all:nTF
25136 {
25137 { \cs_if_eq_p:NN #1 __regex_class:NnnnN }
25138 {#2}
25139 { \tl_if_head_eq_meaning_p:nN {#3} __regex_item_caseful_equal:n }
25140 { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
25141 { \int_compare_p:nNn {#5} = { 0 } }
25142 }
25143 {
25144 \prg_replicate:nn {#4}
25145 { \char_generate:nn { \use_i:nn #3 } {12} }

```

```

25146 __regex_compile_cs_aux:NNnnnN
25147 }
25148 {
25149 __regex_quark_if_nil:NF #1
25150 {
25151 \flag_raise_if_clear:n { __regex_cs }
25152 __regex_use_i_delimit_by_q_recursion_stop:nw
25153 }
25154 __regex_use_none_delimit_by_q_recursion_stop:w
25155 }
25156 }
```

(End definition for `\__regex_compile_`: and others.)

#### 41.3.12 Raw token lists with \u

The `\u` escape is invalid in classes and directly following a catcode test. Otherwise, it must be followed by a left brace. We then collect the characters for the argument of `\u` within an x-expanding assignment. In principle we could just wait to encounter a right brace, but this is unsafe: if the right brace was missing, then we would reach the end-markers of the regex, and continue, leading to obscure fatal errors. Instead, we only allow raw and special characters, and stop when encountering a special right brace, any escaped character, or the end-marker.

```

25157 \cs_new_protected:cpn { __regex_compile_u: } #1#2
25158 {
25159 __regex_if_in_class_or_catcode:TF
25160 { __regex_compile_raw_error:N u #1 #2 }
25161 {
25162 __regex_two_if_eq:NNNTF #1 #2 __regex_compile_special:N \c_left_brace_str
25163 {
25164 __kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
25165 __regex_compile_u_loop:NN
25166 }
25167 {
25168 __kernel_msg_error:nn { kernel } { u-missing-lbrace }
25169 __regex_compile_raw:N u #1 #2
25170 }
25171 }
25172 }
25173 \cs_new:Npn __regex_compile_u_loop:NN #1#2
25174 {
25175 \token_if_eq_meaning:NNTF #1 __regex_compile_raw:N
25176 { #2 __regex_compile_u_loop:NN }
25177 {
25178 \token_if_eq_meaning:NNTF #1 __regex_compile_special:N
25179 {
25180 \exp_after:wN \token_if_eqCharCode:NNTF \c_right_brace_str #2
25181 { \if_false: { \fi: } __regex_compile_u_end: }
25182 { #2 __regex_compile_u_loop:NN }
25183 }
25184 {
25185 \if_false: { \fi: }
25186 __kernel_msg_error:nnx { kernel } { u-missing-rbrace } {#2}
25187 __regex_compile_u_end:
```

```

25188 #1 #2
25189 }
25190 }
25191 }

(End definition for __regex_compile_u: and __regex_compile_u_loop:NN.)
```

\\_\\_regex\\_compile\\_u\\_end: Once we have extracted the variable's name, we store the contents of that variable in \l\\_\\_regex\\_internal\\_a\\_tl. The behaviour of \u then depends on whether we are within a \c{...} escape (in this case, the variable is turned to a string), or not.

```

25192 \cs_new_protected:Npn __regex_compile_u_end:
25193 {
25194 \tl_set:Nv \l__regex_internal_a_tl { \l__regex_internal_a_tl }
25195 \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
25196 __regex_compile_u_not_cs:
25197 \else:
25198 __regex_compile_u_in_cs:
25199 \fi:
25200 }
```

(End definition for \\_\\_regex\\_compile\\_u\\_end:.)

\\_\\_regex\\_compile\\_u\\_in\\_cs: When \u appears within a control sequence, we convert the variable to a string with escaped spaces. Then for each character insert a class matching exactly that character, once.

```

25201 \cs_new_protected:Npn __regex_compile_u_in_cs:
25202 {
25203 __kernel_tl_gset:Nx \g__regex_internal_tl
25204 {
25205 \exp_args:No __kernel_str_to_other_fast:n
25206 { \l__regex_internal_a_tl }
25207 }
25208 \tl_build_put_right:Nx \l__regex_build_tl
25209 {
25210 \tl_map_function:NN \g__regex_internal_tl
25211 __regex_compile_u_in_cs_aux:n
25212 }
25213 }
25214 \cs_new:Npn __regex_compile_u_in_cs_aux:n #1
25215 {
25216 __regex_class:NnnnN \c_true_bool
25217 { __regex_item_caseful_equal:n { \int_value:w '#1 } }
25218 { 1 } { 0 } \c_false_bool
25219 }
```

(End definition for \\_\\_regex\\_compile\\_u\\_in\\_cs:.)

\\_\\_regex\\_compile\\_u\\_not\\_cs: In mode 0, the \u escape adds one state to the NFA for each token in \l\\_\\_regex\\_internal\\_a\\_tl. If a given *<token>* is a control sequence, then insert a string comparison test, otherwise, \\_\\_regex\\_item\\_exact:nn which compares catcode and character code.

```

25220 \cs_new_protected:Npn __regex_compile_u_not_cs:
25221 {
25222 \tl_analysis_map_inline:Nn \l__regex_internal_a_tl
25223 }
```

```

25224 \tl_build_put_right:Nx \l__regex_build_tl
25225 {
25226 __regex_class:NnnnN \c_true_bool
25227 {
25228 \if_int_compare:w ##3 = 0 \exp_stop_f:
25229 __regex_item_exact_cs:n
25230 { \exp_after:wN \cs_to_str:N ##1 }
25231 \else:
25232 __regex_item_exact:nn { \int_value:w "##3 } { ##2 }
25233 \fi:
25234 }
25235 { 1 } { 0 } \c_false_bool
25236 }
25237 }
25238 }
```

(End definition for `\__regex_compile_u_not_cs:`)

#### 41.3.13 Other

- `\__regex_compile/_K:`: The `\K` control sequence is currently the only “command”, which performs some action, rather than matching something. It is allowed in the same contexts as `\b`. At the compilation stage, we leave it as a single control sequence, defined later.

```

25239 \cs_new_protected:cpn { __regex_compile/_K: }
25240 {
25241 \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
25242 { \tl_build_put_right:Nn \l__regex_build_tl { __regex_command_K: } }
25243 { __regex_compile_raw_error:N K }
25244 }
```

(End definition for `\__regex_compile/_K:`)

#### 41.3.14 Showing regexes

- `\__regex_show:N`: Within a group and within `\tl_build_begin:N ... \tl_build_end:N` we redefine all the function that can appear in a compiled regex, then run the regex. The result stored in `\l__regex_internal_a_tl` is then meant to be shown.

```

25245 \cs_new_protected:Npn __regex_show:N #1
25246 {
25247 \group_begin:
25248 \tl_build_begin:N \l__regex_build_tl
25249 \cs_set_protected:Npn __regex_branch:n
25250 {
25251 \seq_pop_right:NN \l__regex_show_prefix_seq
25252 \l__regex_internal_a_tl
25253 __regex_show_one:n { +-branch }
25254 \seq_put_right:No \l__regex_show_prefix_seq
25255 \l__regex_internal_a_tl
25256 \use:n
25257 }
25258 \cs_set_protected:Npn __regex_group:nnnN
25259 { __regex_show_group_aux:nnnnN { } }
25260 \cs_set_protected:Npn __regex_group_no_capture:nnnN
25261 { __regex_show_group_aux:nnnnN { ~ (no~capture) } }
```

```

25262 \cs_set_protected:Npn __regex_group_resetting:nNNN
25263 { __regex_show_group_aux:nNNN { ~ (resetting) } }
25264 \cs_set_eq:NN __regex_class:NnnnN __regex_show_class:NnnnN
25265 \cs_set_protected:Npn __regex_command_K:
25266 { __regex_show_one:n { reset~match~start~(\iow_char:N\\K) } }
25267 \cs_set_protected:Npn __regex_assertion:Nn ##1##2
25268 {
25269 __regex_show_one:n
25270 { \bool_if:NF ##1 { negative~ } assertion:~##2 }
25271 }
25272 \cs_set:Npn __regex_b_test: { word~boundary }
25273 \cs_set_eq:NN __regex_anchor:N __regex_show_anchor_to_str:N
25274 \cs_set_protected:Npn __regex_item_caseful_equal:n ##1
25275 { __regex_show_one:n { char~code~\int_eval:n{##1} } }
25276 \cs_set_protected:Npn __regex_item_caseful_range:nn ##1##2
25277 {
25278 __regex_show_one:n
25279 { range~[\int_eval:n{##1}, \int_eval:n{##2}] }
25280 }
25281 \cs_set_protected:Npn __regex_item_caseless_equal:n ##1
25282 { __regex_show_one:n { char~code~\int_eval:n{##1}~(caseless) } }
25283 \cs_set_protected:Npn __regex_item_caseless_range:nn ##1##2
25284 {
25285 __regex_show_one:n
25286 { Range~[\int_eval:n{##1}, \int_eval:n{##2}]~(caseless) }
25287 }
25288 \cs_set_protected:Npn __regex_item_catcode:nT
25289 { __regex_show_item_catcode:NnT \c_true_bool }
25290 \cs_set_protected:Npn __regex_item_catcode_reverse:nT
25291 { __regex_show_item_catcode:NnT \c_false_bool }
25292 \cs_set_protected:Npn __regex_item_reverse:n
25293 { __regex_show_scope:nn { Reversed~match } }
25294 \cs_set_protected:Npn __regex_item_exact:nn ##1##2
25295 { __regex_show_one:n { char~##2,~catcode~##1 } }
25296 \cs_set_eq:NN __regex_item_exact_cs:n __regex_show_item_exact_cs:n
25297 \cs_set_protected:Npn __regex_item_cs:n
25298 { __regex_show_scope:nn { control~sequence } }
25299 \cs_set:cpx { __regex_prop_:: } { __regex_show_one:n { any~token } }
25300 \seq_clear:N \l__regex_show_prefix_seq
25301 __regex_show_push:n { ~ }
25302 \cs_if_exist_use:N #1
25303 \tl_build_end:N \l__regex_build_tl
25304 \exp_args:NNNo
25305 \group_end:
25306 \tl_set:Nn \l__regex_internal_a_tl { \l__regex_build_tl }
25307 }

```

(End definition for `\__regex_show:N`.)

`\__regex_show_one:n` Every part of the final message go through this function, which adds one line to the output, with the appropriate prefix.

```

25308 \cs_new_protected:Npn __regex_show_one:n #
25309 {
25310 \int_incr:N \l__regex_show_lines_int

```

```

25311 \tl_build_put_right:Nx \l__regex_build_tl
25312 {
25313 \exp_not:N \iow_newline:
25314 \seq_map_function:NN \l__regex_show_prefix_seq \use:n
25315 #1
25316 }
25317 }
```

(End definition for `\_regex_show_one:n.`)

`\_regex_show_push:n`  
`\_regex_show_pop:` Enter and exit levels of nesting. The `scope` function prints its first argument as an “introduction”, then performs its second argument in a deeper level of nesting.

```

25318 \cs_new_protected:Npn _regex_show_push:n #1
25319 { \seq_put_right:Nx \l__regex_show_prefix_seq { #1 ~ } }
25320 \cs_new_protected:Npn _regex_show_pop:
25321 { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
25322 \cs_new_protected:Npn _regex_show_scope:nn #1#2
25323 {
25324 _regex_show_one:n {#1}
25325 _regex_show_push:n { ~ }
25326 #2
25327 _regex_show_pop:
25328 }
```

(End definition for `\_regex_show_push:n`, `\_regex_show_pop:`, and `\_regex_show_scope:nn.`)

`\_regex_show_group_aux:nnnn` We display all groups in the same way, simply adding a message, `(no capture)` or `(resetting)`, to special groups. The odd `\use_ii:nn` avoids printing a spurious `+–branch` for the first branch.

```

25329 \cs_new_protected:Npn _regex_show_group_aux:nnnnN #1#2#3#4#5
25330 {
25331 _regex_show_one:n { ,-group~begin #1 }
25332 _regex_show_push:n { | }
25333 \use_ii:nn #2
25334 _regex_show_pop:
25335 _regex_show_one:n
25336 { ‘-group~end _regex_msg_repeated:nnN {#3} {#4} #5 }
25337 }
```

(End definition for `\_regex_show_group_aux:nnnn.`)

`\_regex_show_class:Nnnn` I’m entirely unhappy about this function: I couldn’t find a way to test if a class is a single test. Instead, collect the representation of the tests in the class. If that had more than one line, write `Match` or `Don’t match` on its own line, with the repeating information if any. Then the various tests on lines of their own, and finally a line. Otherwise, we need to evaluate the representation of the tests again (since the prefix is incorrect). That’s clunky, but not too expensive, since it’s only one test.

```

25338 \cs_set:Npn _regex_show_class:NnnnN #1#2#3#4#5
25339 {
25340 \group_begin:
25341 \tl_build_begin:N \l__regex_build_tl
25342 \int_zero:N \l__regex_show_lines_int
25343 _regex_show_push:n {~}
25344 #2
```

```

25345 \int_compare:nTF { \l__regex_show_lines_int = 0 }
25346 {
25347 \group_end:
25348 __regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
25349 }
25350 {
25351 \bool_if:nTF
25352 { #1 && \int_compare_p:n { \l__regex_show_lines_int = 1 } }
25353 {
25354 \group_end:
25355 #2
25356 \tl_build_put_right:Nn \l__regex_build_tl
25357 { __regex_msg_repeated:nnN {#3} {#4} #5 }
25358 }
25359 {
25360 \tl_build_end:N \l__regex_build_tl
25361 \exp_args:NNNo
25362 \group_end:
25363 \tl_set:Nn \l__regex_internal_a_tl \l__regex_build_tl
25364 __regex_show_one:n
25365 {
25366 \bool_if:NTF #1 { Match } { Don't-match }
25367 __regex_msg_repeated:nnN {#3} {#4} #5
25368 }
25369 \tl_build_put_right:Nx \l__regex_build_tl
25370 { \exp_not:o \l__regex_internal_a_tl }
25371 }
25372 }
25373 }
```

(End definition for `\__regex_show_class:NnnnN`.)

`\__regex_show_anchor_to_str:N` The argument is an integer telling us where the anchor is. We convert that to the relevant info.

```

25374 \cs_new:Npn __regex_show_anchor_to_str:N #1
25375 {
25376 anchor~at~
25377 \str_case:nnF { #1 }
25378 {
25379 { \l__regex_min_pos_int } { start~(\iow_char:N\A) }
25380 { \l__regex_start_pos_int } { start-of~match~(\iow_char:N\G) }
25381 { \l__regex_max_pos_int } { end~(\iow_char:N\Z) }
25382 }
25383 { <error:~'#1'~not~recognized> }
25384 }
```

(End definition for `\__regex_show_anchor_to_str:N`.)

`\__regex_show_item_catcode:NnT` Produce a sequence of categories which the catcode bitmap #2 contains, and show it, indenting the tests on which this catcode constraint applies.

```

25385 \cs_new_protected:Npn __regex_show_item_catcode:NnT #1#2
25386 {
25387 \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
25388 \seq_set_filter:NNn \l__regex_internal_seq \l__regex_internal_seq
```

```

25389 { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
25390 __regex_show_scope:nn
25391 {
25392 categories~
25393 \seq_map_function:NN \l__regex_internal_seq \use:n
25394 ,
25395 \bool_if:NF #1 { negative~ } class
25396 }
25397 }
```

(End definition for `\__regex_show_item_catcode:NnT.`)

```

__regex_show_item_exact_cs:n
25398 \cs_new_protected:Npn __regex_show_item_exact_cs:n #1
25399 {
25400 \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
25401 \seq_set_map_x>NNn \l__regex_internal_seq
25402 \l__regex_internal_seq { \iow_char:N\##1 }
25403 __regex_show_one:n
25404 { control-sequence~ \seq_use:Nn \l__regex_internal_seq { ~or~ } }
25405 }
```

(End definition for `\__regex_show_item_exact_cs:n.`)

## 41.4 Building

### 41.4.1 Variables used while building

`\l__regex_min_state_int`  
`\l__regex_max_state_int`

The last state that was allocated is `\l__regex_max_state_int - 1`, so that `\l__regex_max_state_int` always points to a free state. The `min_state` variable is 1 to begin with, but gets shifted in nested calls to the matching code, namely in `\c{...}` constructions.

```

25406 \int_new:N \l__regex_min_state_int
25407 \int_set:Nn \l__regex_min_state_int { 1 }
25408 \int_new:N \l__regex_max_state_int
```

(End definition for `\l__regex_min_state_int` and `\l__regex_max_state_int.`)

`\l__regex_left_state_int`  
`\l__regex_right_state_int`  
`\l__regex_left_state_seq`  
`\l__regex_right_state_seq`

Alternatives are implemented by branching from a `left` state into the various choices, then merging those into a `right` state. We store information about those states in two sequences. Those states are also used to implement group quantifiers. Most often, the left and right pointers only differ by 1.

```

25409 \int_new:N \l__regex_left_state_int
25410 \int_new:N \l__regex_right_state_int
25411 \seq_new:N \l__regex_left_state_seq
25412 \seq_new:N \l__regex_right_state_seq
```

(End definition for `\l__regex_left_state_int` and others.)

`\l__regex_capturing_group_int`

`\l__regex_capturing_group_int` is the next ID number to be assigned to a capturing group. This starts at 0 for the group enclosing the full regular expression, and groups are counted in the order of their left parenthesis, except when encountering `resetting` groups.

```

25413 \int_new:N \l__regex_capturing_group_int
```

(End definition for `\l__regex_capturing_group_int.`)

#### 41.4.2 Framework

This phase is about going from a compiled regex to an NFA. Each state of the NFA is stored in a `\toks`. The operations which can appear in the `\toks` are

- `\__regex_action_start_wildcard`: inserted at the start of the regular expression to make it unanchored.
- `\__regex_action_success`: marks the exit state of the NFA.
- `\__regex_action_cost:n {<shift>}` is a transition from the current `<state>` to `<state> + <shift>`, which consumes the current character: the target state is saved and will be considered again when matching at the next position.
- `\__regex_action_free:n {<shift>}`, and `\__regex_action_free_group:n {<shift>}` are free transitions, which immediately perform the actions for the state `<state> + <shift>` of the NFA. They differ in how they detect and avoid infinite loops. For now, we just need to know that the `group` variant must be used for transitions back to the start of a group.
- `\__regex_action_submatch:n {<key>}` where the `<key>` is a group number followed by `<` or `>` for the beginning or end of group. This causes the current position in the query to be stored as the `<key>` submatch boundary.

We strive to preserve the following properties while building.

- The current capturing group is `capturing_group - 1`, and if a group opened now it would be labelled `capturing_group`.
- The last allocated state is `max_state - 1`, so `max_state` is a free state.
- The `left_state` points to a state to the left of the current group or of the last class.
- The `right_state` points to a newly created, empty state, with some transitions leading to it.
- The `left/right` sequences hold a list of the corresponding end-points of nested groups.

`\__regex_build:n` The `n`-type function first compiles its argument. Reset some variables. Allocate two states, and put a wildcard in state 0 (transitions to state 1 and 0 state). Then build the regex within a (capturing) group numbered 0 (current value of `capturing_group`). Finally, if the match reaches the last state, it is successful.

```

25414 \cs_new_protected:Npn __regex_build:n #1
25415 {
25416 __regex_compile:n [#1]
25417 __regex_build:N \l__regex_internal_regex
25418 }
25419 \cs_new_protected:Npn __regex_build:N #1
25420 {
25421 __regex_standard_escapechar:
25422 \int_zero:N \l__regex_capturing_group_int
25423 \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
25424 __regex_build_new_state:

```

```

25425 __regex_build_new_state:
25426 __regex_toks_put_right:Nn \l__regex_left_state_int
25427 { __regex_action_start_wildcard: }
25428 __regex_group:nnnN {\#1} { 1 } { 0 } \c_false_bool
25429 __regex_toks_put_right:Nn \l__regex_right_state_int
25430 { __regex_action_success: }
25431 }

```

(End definition for `\_\_regex\_build:n` and `\_\_regex\_build:N`.)

`\_\_regex\_build\_for\_cs:n` The matching code relies on some global intarray variables, but only uses a range of their entries. Specifically,

- `\g\_\_regex\_state\_active\_intarray` from `\l\_\_regex\_min\_state\_int` to `\l\_\_regex\_max\_state\_int`
- `\g\_\_regex\_thread\_state\_intarray` from `\l\_\_regex\_min\_active\_int` to `\l\_\_regex\_max\_active\_int`

In fact, some data is stored in `\toks` registers (local) in the same ranges so these ranges mustn't overlap. This is done by setting `\l\_\_regex\_min\_active\_int` to `\l\_\_regex\_max\_state\_int` after building the NFA. Here, in this nested call to the matching code, we need the new versions of these ranges to involve completely new entries of the intarray variables, so we begin by setting (the new) `\l\_\_regex\_min\_state\_int` to (the old) `\l\_\_regex\_max\_active\_int` to use higher entries.

When using a regex to match a cs, we don't insert a wildcard, we anchor at the end, and since we ignore submatches, there is no need to surround the expression with a group. However, for branches to work properly at the outer level, we need to put the appropriate `left` and `right` states in their sequence.

```

25432 \cs_new_protected:Npn __regex_build_for_cs:n #1
25433 {
25434 \int_set_eq:NN \l__regex_min_state_int \l__regex_max_active_int
25435 \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
25436 __regex_build_new_state:
25437 __regex_build_new_state:
25438 __regex_push_lr_states:
25439 #1
25440 __regex_pop_lr_states:
25441 __regex_toks_put_right:Nn \l__regex_right_state_int
25442 {
25443 \if_int_compare:w \l__regex_curr_pos_int = \l__regex_max_pos_int
25444 \exp_after:wn __regex_action_success:
25445 \fi:
25446 }
25447 }

```

(End definition for `\_\_regex\_build\_for\_cs:n`.)

#### 41.4.3 Helpers for building an nfa

`\_\_regex\_push\_lr\_states:` When building the regular expression, we keep track of pointers to the left-end and right-end of each group without help from T<sub>E</sub>X's grouping.

```

25448 \cs_new_protected:Npn __regex_push_lr_states:
25449 {

```

```

25450 \seq_push:No \l__regex_left_state_seq
25451 { \int_use:N \l__regex_left_state_int }
25452 \seq_push:No \l__regex_right_state_seq
25453 { \int_use:N \l__regex_right_state_int }
25454 }
25455 \cs_new_protected:Npn __regex_pop_lr_states:
25456 {
25457 \seq_pop:NN \l__regex_left_state_seq \l__regex_internal_a_t1
25458 \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_t1
25459 \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_t1
25460 \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_t1
25461 }

```

(End definition for `\__regex_push_lr_states:` and `\__regex_pop_lr_states:..`)

`\__regex_build_transition_left:NNN`  
`\__regex_build_transition_right:nNn`

Add a transition from #2 to #3 using the function #1. The `left` function is used for higher priority transitions, and the `right` function for lower priority transitions (which should be performed later). The signatures differ to reflect the differing usage later on. Both functions could be optimized.

```

25462 \cs_new_protected:Npn __regex_build_transition_left:NNN #1#2#3
25463 { __regex_toks_put_left:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }
25464 \cs_new_protected:Npn __regex_build_transition_right:nNn #1#2#3
25465 { __regex_toks_put_right:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }

```

(End definition for `\__regex_build_transition_left:NNN` and `\__regex_build_transition_right:nNn`)

`\__regex_build_new_state:`

Add a new empty state to the NFA. Then update the `left`, `right`, and `max` states, so that the `right` state is the new empty state, and the `left` state points to the previously “current” state.

```

25466 \cs_new_protected:Npn __regex_build_new_state:
25467 {
25468 __regex_toks_clear:N \l__regex_max_state_int
25469 \int_set_eq:NN \l__regex_left_state_int \l__regex_right_state_int
25470 \int_set_eq:NN \l__regex_right_state_int \l__regex_max_state_int
25471 \int_incr:N \l__regex_max_state_int
25472 }

```

(End definition for `\__regex_build_new_state:..`)

`\__regex_build_transitions_laziness:NNNNN`

This function creates a new state, and puts two transitions starting from the old current state. The order of the transitions is controlled by #1, true for lazy quantifiers, and false for greedy quantifiers.

```

25473 \cs_new_protected:Npn __regex_build_transitions_laziness:NNNNN #1#2#3#4#5
25474 {
25475 __regex_build_new_state:
25476 __regex_toks_put_right:Nx \l__regex_left_state_int
25477 {
25478 \if_meaning:w \c_true_bool #1
25479 #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
25480 #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
25481 \else:
25482 #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
25483 #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
25484 \fi:

```

```

25485 }
25486 }

```

(End definition for `\_regex_build_transitions_lazyness:NNNNN`.)

#### 41.4.4 Building classes

`\_regex_class:NnnnN` The arguments are:  $\langle \text{boolean} \rangle \{\langle \text{tests} \rangle\} \{\langle \text{min} \rangle\} \{\langle \text{more} \rangle\} \langle \text{lazyness} \rangle$ . First store the tests with a trailing `\_regex_action_cost:n`, in the true branch of `\_regex_break_point:TF` for positive classes, or the false branch for negative classes. The integer  $\langle \text{more} \rangle$  is 0 for fixed repetitions, -1 for unbounded repetitions, and  $\langle \text{max} \rangle - \langle \text{min} \rangle$  for a range of repetitions.

```

25487 \cs_new_protected:Npn _regex_class:NnnnN #1#2#3#4#5
25488 {
25489 \cs_set:Npx _regex_tests_action_cost:n ##1
25490 {
25491 \exp_not:n { \exp_not:n {#2} }
25492 \bool_if:NTF #1
25493 { _regex_break_point:TF { _regex_action_cost:n {##1} } { } }
25494 { _regex_break_point:TF { } { _regex_action_cost:n {##1} } }
25495 }
25496 \if_case:w - #4 \exp_stop_f:
25497 _regex_class_repeat:n {#3}
25498 \or: _regex_class_repeat:nN {#3} #5
25499 \else: _regex_class_repeat:nnN {#3} {#4} #5
25500 \fi:
25501 }
25502 \cs_new:Npn _regex_tests_action_cost:n { _regex_action_cost:n }

(End definition for _regex_class:NnnnN and _regex_tests_action_cost:n.)
```

`\_regex_class_repeat:n` This is used for a fixed number of repetitions. Build one state for each repetition, with a transition controlled by the tests that we have collected. That works just fine for  $\#1 = 0$  repetitions: nothing is built.

```

25503 \cs_new_protected:Npn _regex_class_repeat:n #1
25504 {
25505 \prg_replicate:nn {#1}
25506 {
25507 _regex_build_new_state:
25508 _regex_build_transition_right:nNn _regex_tests_action_cost:n
25509 \l__regex_left_state_int \l__regex_right_state_int
25510 }
25511 }

(End definition for _regex_class_repeat:n.)
```

`\_regex_class_repeat:nN` This implements unbounded repetitions of a single class (*e.g.* the \* and + quantifiers). If the minimum number  $\#1$  of repetitions is 0, then build a transition from the current state to itself governed by the tests, and a free transition to a new state (hence skipping the tests). Otherwise, call `\_regex_class_repeat:n` for the code to match  $\#1$  repetitions, and add free transitions from the last state to the previous one, and to a new one. In both cases, the order of transitions is controlled by the lazyness boolean  $\#2$ .

```

25512 \cs_new_protected:Npn _regex_class_repeat:nN #1#2
25513 {


```

```

25514 \if_int_compare:w #1 = 0 \exp_stop_f:
25515 _regex_build_transitions_lazyness:NNNNN #2
25516 __regex_action_free:n \l_regex_right_state_int
25517 __regex_tests_action_cost:n \l_regex_left_state_int
25518 \else:
25519 __regex_class_repeat:n {#1}
25520 \int_set_eq:NN \l_regex_internal_a_int \l_regex_left_state_int
25521 _regex_build_transitions_lazyness:NNNNN #2
25522 __regex_action_free:n \l_regex_right_state_int
25523 __regex_action_free:n \l_regex_internal_a_int
25524 \fi:
25525 }

```

(End definition for `\_\_regex_class_repeat:nN`.)

`\_\_regex_class_repeat:nnN`

We want to build the code to match from #1 to #1 + #2 repetitions. Match #1 repetitions (can be 0). Compute the final state of the next construction as `a`. Build #2 > 0 states, each with a transition to the next state governed by the tests, and a transition to the final state `a`. The computation of `a` is safe because states are allocated in order, starting from `max_state`.

```

25526 \cs_new_protected:Npn __regex_class_repeat:nnN #1#2#3
25527 {
25528 __regex_class_repeat:n {#1}
25529 \int_set:Nn \l_regex_internal_a_int
25530 { \l_regex_max_state_int + #2 - 1 }
25531 \prg_replicate:nn { #2 }
25532 {
25533 _regex_build_transitions_lazyness:NNNNN #3
25534 __regex_action_free:n \l_regex_internal_a_int
25535 __regex_tests_action_cost:n \l_regex_right_state_int
25536 }
25537 }

```

(End definition for `\_\_regex_class_repeat:nnN`.)

#### 41.4.5 Building groups

`\_\_regex_group_aux:nnnnN`

Arguments: {`<label>`} {`<contents>`} {`<min>`} {`<more>`} `<lazyness>`. If `<min>` is 0, we need to add a state before building the group, so that the thread which skips the group does not also set the start-point of the submatch. After adding one more state, the `left_state` is the left end of the group, from which all branches stem, and the `right_state` is the right end of the group, and all branches end their course in that state. We store those two integers to be queried for each branch, we build the NFA states for the contents #2 of the group, and we forget about the two integers. Once this is done, perform the repetition: either exactly #3 times, or #3 or more times, or between #3 and #3 + #4 times, with lazyness #5. The `<label>` #1 is used for submatch tracking. Each of the three auxiliaries expects `left_state` and `right_state` to be set properly.

```

25538 \cs_new_protected:Npn __regex_group_aux:nnnnN #1#2#3#4#5
25539 {
25540 \if_int_compare:w #3 = 0 \exp_stop_f:
25541 _regex_build_new_state:
25542 \assert\assert_int:n { \l_regex_max_state_int = \l_regex_right_state_int + 1 }
25543 _regex_build_transition_right:nNn __regex_action_free_group:n

```

```

25544 \l__regex_left_state_int \l__regex_right_state_int
25545 \fi:
25546 __regex_build_new_state:
25547 __regex_push_lr_states:
25548 #2
25549 __regex_pop_lr_states:
25550 \if_case:w - #4 \exp_stop_f:
25551 __regex_group_repeat:nn {#1} {#3}
25552 \or: __regex_group_repeat:nnN {#1} {#3} #5
25553 \else: __regex_group_repeat:nnnN {#1} {#3} {#4} #5
25554 \fi:
25555 }

```

(End definition for `\__regex_group_aux:nnnnN`.)

`\__regex_group:nnnN` Hand to `\__regex_group_aux:nnnnN` the label of that group (expanded), and the group itself, with some extra commands to perform.

```

25556 \cs_new_protected:Npn __regex_group:nnnN #1
25557 {
25558 \exp_args:No __regex_group_aux:nnnnN
25559 { \int_use:N \l__regex_capturing_group_int }
25560 {
25561 \int_incr:N \l__regex_capturing_group_int
25562 #1
25563 }
25564 }
25565 \cs_new_protected:Npn __regex_group_no_capture:nnnN
25566 { __regex_group_aux:nnnnN { -1 } }

```

(End definition for `\__regex_group:nnnN` and `\__regex_group_no_capture:nnnN`.)

`\__regex_group_resetting:nnnN` Again, hand the label `-1` to `\__regex_group_aux:nnnnN`, but this time we work a little bit harder to keep track of the maximum group label at the end of any branch, and to reset the group number at each branch. This relies on the fact that a compiled regex always is a sequence of items of the form `\__regex_branch:n {<branch>}`.

```

25567 \cs_new_protected:Npn __regex_group_resetting:nnnN #1
25568 {
25569 __regex_group_aux:nnnnN { -1 }
25570 {
25571 \exp_args:Noo __regex_group_resetting_loop:nnNn
25572 { \int_use:N \l__regex_capturing_group_int }
25573 { \int_use:N \l__regex_capturing_group_int }
25574 #1
25575 { ?? \prg_break:n } { }
25576 \prg_break_point:
25577 }
25578 }
25579 \cs_new_protected:Npn __regex_group_resetting_loop:nnNn #1#2#3#4
25580 {
25581 \use_none:nn #3 { \int_set:Nn \l__regex_capturing_group_int {#1} }
25582 \int_set:Nn \l__regex_capturing_group_int {#2}
25583 #3 {#4}
25584 \exp_args:Nf __regex_group_resetting_loop:nnNn
25585 { \int_max:nn {#1} { \l__regex_capturing_group_int } }

```

```

25586 {#2}
25587 }

```

(End definition for `\_regex_group_resetting:nnN` and `\_regex_group_resetting_loop:nnNn`.)

`\_regex_branch:n` Add a free transition from the left state of the current group to a brand new state, starting point of this branch. Once the branch is built, add a transition from its last state to the right state of the group. The left and right states of the group are extracted from the relevant sequences.

```

25588 \cs_new_protected:Npn _regex_branch:n #1
25589 {
25590 _regex_build_new_state:
25591 \seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_tl
25592 \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
25593 _regex_build_transition_right:nNn _regex_action_free:n
25594 \l__regex_left_state_int \l__regex_right_state_int
25595 #1
25596 \seq_get:NN \l__regex_right_state_seq \l__regex_internal_a_tl
25597 _regex_build_transition_right:nNn _regex_action_free:n
25598 \l__regex_right_state_int \l__regex_internal_a_tl
25599 }

```

(End definition for `\_regex_branch:n`.)

`\_regex_group_repeat:nn` This function is called to repeat a group a fixed number of times #2; if this is 0 we remove the group altogether (but don't reset the `capturing_group` label). Otherwise, the auxiliary `\_regex_group_repeat_aux:n` copies #2 times the `\toks` for the group, and leaves `internal_a` pointing to the left end of the last repetition. We only record the submatch information at the last repetition. Finally, add a state at the end (the transition to it has been taken care of by the replicating auxiliary).

```

25600 \cs_new_protected:Npn _regex_group_repeat:nn #1#2
25601 {
25602 \if_int_compare:w #2 = 0 \exp_stop_f:
25603 \int_set:Nn \l__regex_max_state_int
25604 { \l__regex_left_state_int - 1 }
25605 _regex_build_new_state:
25606 \else:
25607 _regex_group_repeat_aux:n {#2}
25608 _regex_group_submatches:nNN {#1}
25609 \l__regex_internal_a_int \l__regex_right_state_int
25610 _regex_build_new_state:
25611 \fi:
25612 }

```

(End definition for `\_regex_group_repeat:nn`.)

`\_regex_group_submatches:nNN` This inserts in states #2 and #3 the code for tracking submatches of the group #1, unless inhibited by a label of -1.

```

25613 \cs_new_protected:Npn _regex_group_submatches:nNN #1#2#3
25614 {
25615 \if_int_compare:w #1 > - 1 \exp_stop_f:
25616 _regex_toks_put_left:Nx #2 { _regex_action_submatch:n { #1 < } }
25617 _regex_toks_put_left:Nx #3 { _regex_action_submatch:n { #1 > } }
25618 \fi:
25619 }

```

(End definition for `\_regex_group_submatches:nNN`.)

`\_regex_group_repeat_aux:n` Here we repeat `\toks` ranging from `left_state` to `max_state`,  $\#1 > 0$  times. First add a transition so that the copies “chain” properly. Compute the shift `c` between the original copy and the last copy we want. Shift the `right_state` and `max_state` to their final values. We then want to perform `c` copy operations. At the end, `b` is equal to the `max_state`, and `a` points to the left of the last copy of the group.

```
25620 \cs_new_protected:Npn _regex_group_repeat_aux:n #1
25621 {
25622 _regex_build_transition_right:nNn _regex_action_free:n
25623 \l__regex_right_state_int \l__regex_max_state_int
25624 \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
25625 \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int
25626 \if_int_compare:w \int_eval:n {\#1} > 1 \exp_stop_f:
25627 \int_set:Nn \l__regex_internal_c_int
25628 {
25629 (#1 - 1)
25630 * (\l__regex_internal_b_int - \l__regex_internal_a_int)
25631 }
25632 \int_add:Nn \l__regex_right_state_int { \l__regex_internal_c_int }
25633 \int_add:Nn \l__regex_max_state_int { \l__regex_internal_c_int }
25634 _regex_toks_memcpy>NNn
25635 \l__regex_internal_b_int
25636 \l__regex_internal_a_int
25637 \l__regex_internal_c_int
25638 \fi:
25639 }
```

(End definition for `\_regex_group_repeat_aux:n`.)

`\_regex_group_repeat:nnN` This function is called to repeat a group at least  $n$  times; the case  $n = 0$  is very different from  $n > 0$ . Assume first that  $n = 0$ . Insert submatch tracking information at the start and end of the group, add a free transition from the right end to the “true” left state `a` (remember: in this case we had added an extra state before the left state). This forms the loop, which we break away from by adding a free transition from `a` to a new state.

Now consider the case  $n > 0$ . Repeat the group  $n$  times, chaining various copies with a free transition. Add submatch tracking only to the last copy, then add a free transition from the right end back to the left end of the last copy, either before or after the transition to move on towards the rest of the NFA. This transition can end up before submatch tracking, but that is irrelevant since it only does so when going again through the group, recording new matches. Finally, add a state; we already have a transition pointing to it from `\_regex_group_repeat_aux:n`.

```
25640 \cs_new_protected:Npn _regex_group_repeat:nnN #1#2#3
25641 {
25642 \if_int_compare:w #2 = 0 \exp_stop_f:
25643 _regex_group_submatches:nNN {\#1}
25644 \l__regex_left_state_int \l__regex_right_state_int
25645 \int_set:Nn \l__regex_internal_a_int
25646 { \l__regex_left_state_int - 1 }
25647 _regex_build_transition_right:nNn _regex_action_free:n
25648 \l__regex_right_state_int \l__regex_internal_a_int
25649 _regex_build_new_state:
25650 \if_meaning:w \c_true_bool #3
```

```

25651 __regex_build_transition_left:NNN __regex_action_free:n
25652 \|l_ regex_internal_a_int \|l_ regex_right_state_int
25653 \else:
25654 __regex_build_transition_right:nNn __regex_action_free:n
25655 \|l_ regex_internal_a_int \|l_ regex_right_state_int
25656 \fi:
25657 \else:
25658 __regex_group_repeat_aux:n {\#2}
25659 __regex_group_submatches:nNN {\#1}
25660 \|l_ regex_internal_a_int \|l_ regex_right_state_int
25661 \if_meaning:w \c_true_bool {\#3}
25662 __regex_build_transition_right:nNn __regex_action_free_group:n
25663 \|l_ regex_right_state_int \|l_ regex_internal_a_int
25664 \else:
25665 __regex_build_transition_left:NNN __regex_action_free_group:n
25666 \|l_ regex_right_state_int \|l_ regex_internal_a_int
25667 \fi:
25668 __regex_build_new_state:
25669 \fi:
25670 }

```

(End definition for `\_\_regex\_group\_repeat:nnN`.)

`\_\_regex\_group\_repeat:nnN` We wish to repeat the group between #2 and #2 + #3 times, with a laziness controlled by #4. We insert submatch tracking up front: in principle, we could avoid recording submatches for the first #2 copies of the group, but that forces us to treat specially the case #2 = 0. Repeat that group with submatch tracking #2 + #3 times (the maximum number of repetitions). Then our goal is to add #3 transitions from the end of the #2-th group, and each subsequent groups, to the end. For a lazy quantifier, we add those transitions to the left states, before submatch tracking. For the greedy case, we add the transitions to the right states, after submatch tracking and the transitions which go on with more repetitions. In the greedy case with #2 = 0, the transition which skips over all copies of the group must be added separately, because its starting state does not follow the normal pattern: we had to add it “by hand” earlier.

```

25671 \cs_new_protected:Npn __regex_group_repeat:nnnN {\#1\#2\#3\#4}
25672 {
25673 __regex_group_submatches:nNN {\#1}
25674 \|l_ regex_left_state_int \|l_ regex_right_state_int
25675 __regex_group_repeat_aux:n {\#2 + \#3}
25676 \if_meaning:w \c_true_bool {\#4}
25677 \int_set_eq:NN \|l_ regex_left_state_int \|l_ regex_max_state_int
25678 \prg_replicate:nn {\#3}
25679 {
25680 \int_sub:Nn \|l_ regex_left_state_int
25681 { \|l_ regex_internal_b_int - \|l_ regex_internal_a_int }
25682 __regex_build_transition_left:NNN __regex_action_free:n
25683 \|l_ regex_left_state_int \|l_ regex_max_state_int
25684 }
25685 \else:
25686 \prg_replicate:nn {\#3 - 1}
25687 {
25688 \int_sub:Nn \|l_ regex_right_state_int
25689 { \|l_ regex_internal_b_int - \|l_ regex_internal_a_int }

```

```

25690 __regex_build_transition_right:nNn __regex_action_free:n
25691 \l_regex_right_state_int \l_regex_max_state_int
25692 }
25693 \if_int_compare:w #2 = 0 \exp_stop_f:
25694 \int_set:Nn \l_regex_right_state_int
25695 { \l_regex_left_state_int - 1 }
25696 \else:
25697 \int_sub:Nn \l_regex_right_state_int
25698 { \l_regex_internal_b_int - \l_regex_internal_a_int }
25699 \fi:
25700 __regex_build_transition_right:nNn __regex_action_free:n
25701 \l_regex_right_state_int \l_regex_max_state_int
25702 \fi:
25703 __regex_build_new_state:
25704 }

```

(End definition for `\_\_regex_group_repeat:nnnN`.)

#### 41.4.6 Others

`\_\_regex_assertion:Nn`  
`\_\_regex_b_test:`  
`\_\_regex_anchor:N`

Usage: `\_\_regex_assertion:Nn <boolean> {{<test>}}`, where the `<test>` is either of the two other functions. Add a free transition to a new state, conditionally to the assertion test. The `\_\_regex_b_test:` test is used by the `\b` and `\B` escape: check if the last character was a word character or not, and do the same to the current character. The boundary-markers of the string are non-word characters for this purpose. Anchors at the start or end of match use `\_\_regex_anchor:N`, with a position controlled by the integer #1.

```

25705 \cs_new_protected:Npn __regex_assertion:Nn #1#2
25706 {
25707 __regex_build_new_state:
25708 __regex_toks_put_right:Nx \l_regex_left_state_int
25709 {
25710 \exp_not:n {#2}
25711 __regex_break_point:TF
25712 \bool_if:NF #1 { { } }
25713 {
25714 __regex_action_free:n
25715 {
25716 \int_eval:n
25717 { \l_regex_right_state_int - \l_regex_left_state_int }
25718 }
25719 }
25720 \bool_if:NT #1 { { } }
25721 }
25722 }
25723 \cs_new_protected:Npn __regex_anchor:N #1
25724 {
25725 \if_int_compare:w #1 = \l_regex_curr_pos_int
25726 \exp_after:wN __regex_break_true:w
25727 \fi:
25728 }
25729 \cs_new_protected:Npn __regex_b_test:
25730 {
25731 \group_begin:
25732 \int_set_eq:NN \l_regex_curr_char_int \l_regex_last_char_int

```

```

25733 __regex_prop_w:
25734 __regex_break_point:TF
25735 { \group_end: __regex_item_reverse:n __regex_prop_w: }
25736 { \group_end: __regex_prop_w: }
25737 }

(End definition for __regex_assertion:Nn, __regex_b_test:, and __regex_anchor:N.)
```

\\_\\_regex\\_command\_K: Change the starting point of the 0-th submatch (full match), and transition to a new state, pretending that this is a fresh thread.

```

25738 \cs_new_protected:Npn __regex_command_K:
25739 {
25740 __regex_build_new_state:
25741 __regex_toks_put_right:Nx \l__regex_left_state_int
25742 {
25743 __regex_action_submatch:n { 0< }
25744 \bool_set_true:N \l__regex_fresh_thread_bool
25745 __regex_action_free:n
25746 {
25747 \int_eval:n
25748 { \l__regex_right_state_int - \l__regex_left_state_int }
25749 }
25750 \bool_set_false:N \l__regex_fresh_thread_bool
25751 }
25752 }
```

(End definition for \\_\\_regex\\_command\_K:.)

## 41.5 Matching

We search for matches by running all the execution threads through the NFA in parallel, reading one token of the query at each step. The NFA contains “free” transitions to other states, and transitions which “consume” the current token. For free transitions, the instruction at the new state of the NFA is performed immediately. When a transition consumes a character, the new state is appended to a list of “active states”, stored in `\g__regex_thread_state_intarray`: this thread is made active again when the next token is read from the query. At every step (for each token in the query), we unpack that list of active states and the corresponding submatch props, and empty those.

If two paths through the NFA “collide” in the sense that they reach the same state after reading a given token, then they only differ in how they previously matched, and any future execution would be identical for both. (Note that this would be wrong in the presence of back-references.) Hence, we only need to keep one of the two threads: the thread with the highest priority. Our NFA is built in such a way that higher priority actions always come before lower priority actions, which makes things work.

The explanation in the previous paragraph may make us think that we simply need to keep track of which states were visited at a given step: after all, the loop generated when matching `(a?)*` against `a` is broken, isn’t it? No. The group first matches `a`, as it should, then repeats; it attempts to match `a` again but fails; it skips `a`, and finds out that this state has already been seen at this position in the query: the match stops. The capturing group is (wrongly) `a`. What went wrong is that a thread collided with itself, and the later version, which has gone through the group one more times with an empty match, should have a higher priority than not going through the group.

We solve this by distinguishing “normal” free transitions `\__regex_action_free:n` from transitions `\__regex_action_free_group:n` which go back to the start of the group. The former keeps threads unless they have been visited by a “completed” thread, while the latter kind of transition also prevents going back to a state visited by the current thread.

#### 41.5.1 Variables used when matching

The tokens in the query are indexed from `min_pos` for the first to `max_pos - 1` for the last, and their information is stored in several arrays and `\toks` registers with those numbers. We don’t start from 0 because the `\toks` registers with low numbers are used to hold the states of the NFA. We match without backtracking, keeping all threads in lockstep at the `current_pos` in the query. The starting point of the current match attempt is `start_pos`, and `success_pos`, updated whenever a thread succeeds, is used as the next starting position.

```
25753 \int_new:N \l__regex_min_pos_int
25754 \int_new:N \l__regex_max_pos_int
25755 \int_new:N \l__regex_curr_pos_int
25756 \int_new:N \l__regex_start_pos_int
25757 \int_new:N \l__regex_success_pos_int
```

(End definition for `\l__regex_min_pos_int` and others.)

The character and category codes of the token at the current position; the character code of the token at the previous position; and the character code of the result of changing the case of the current token (`A-Z↔a-z`). This last integer is only computed when necessary, and is otherwise `\c_max_int`. The `current_char` variable is also used in various other phases to hold a character code.

```
25758 \int_new:N \l__regex_curr_char_int
25759 \int_new:N \l__regex_curr_catcode_int
25760 \int_new:N \l__regex_last_char_int
25761 \int_new:N \l__regex_case_changed_char_int
```

(End definition for `\l__regex_curr_char_int` and others.)

`\l__regex_curr_state_int` For every character in the token list, each of the active states is considered in turn. The variable `\l__regex_curr_state_int` holds the state of the NFA which is currently considered: transitions are then given as shifts relative to the current state.

```
25762 \int_new:N \l__regex_curr_state_int
```

(End definition for `\l__regex_curr_state_int`.)

The submatches for the thread which is currently active are stored in the `current_submatches` property list variable. This property list is stored by `\__regex_action_cost:n` into the `\toks` register for the target state of the transition, to be retrieved when matching at the next position. When a thread succeeds, this property list is copied to `\l__regex_success_submatches_prop`: only the last successful thread remains there.

```
25763 \prop_new:N \l__regex_curr_submatches_prop
25764 \prop_new:N \l__regex_success_submatches_prop
```

(End definition for `\l__regex_curr_submatches_prop` and `\l__regex_success_submatches_prop`.)

\l\_\_regex\_step\_int This integer, always even, is increased every time a character in the query is read, and not reset when doing multiple matches. We store in \g\_\_regex\_state\_active\_intarray the last step in which each *<state>* in the NFA was encountered. This lets us break infinite loops by not visiting the same state twice in the same step. In fact, the step we store is equal to **step** when we have started performing the operations of \toks(*state*), but not finished yet. However, once we finish, we store **step + 1** in \g\_\_regex\_state\_active\_intarray. This is needed to track submatches properly (see building phase). The **step** is also used to attach each set of submatch information to a given iteration (and automatically discard it when it corresponds to a past step).

25765 \int\_new:N \l\_\_regex\_step\_int

(End definition for \l\_\_regex\_step\_int.)

\l\_\_regex\_min\_active\_int  
\l\_\_regex\_max\_active\_int All the currently active threads are kept in order of precedence in \g\_\_regex\_thread\_state\_intarray, and the corresponding submatches in the \toks. For our purposes, those serve as an array, indexed from **min\_active** (inclusive) to **max\_active** (excluded). At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and **max\_active** is reset to **min\_active**, effectively clearing the array.

25766 \int\_new:N \l\_\_regex\_min\_active\_int

25767 \int\_new:N \l\_\_regex\_max\_active\_int

(End definition for \l\_\_regex\_min\_active\_int and \l\_\_regex\_max\_active\_int.)

\g\_\_regex\_state\_active\_intarray  
\g\_\_regex\_thread\_state\_intarray \g\_\_regex\_state\_active\_intarray stores the last *<step>* in which each *<state>* was active. \g\_\_regex\_thread\_state\_intarray stores threads to be considered in the next step, more precisely the states in which these threads are.

25768 \intarray\_new:Nn \g\_\_regex\_state\_active\_intarray { 65536 }

25769 \intarray\_new:Nn \g\_\_regex\_thread\_state\_intarray { 65536 }

(End definition for \g\_\_regex\_state\_active\_intarray and \g\_\_regex\_thread\_state\_intarray.)

\l\_\_regex\_every\_match\_tl Every time a match is found, this token list is used. For single matching, the token list is empty. For multiple matching, the token list is set to repeat the matching, after performing some operation which depends on the user function. See \\_\\_regex\_single\_match: and \\_\\_regex\_multi\_match:n.

25770 \tl\_new:N \l\_\_regex\_every\_match\_tl

(End definition for \l\_\_regex\_every\_match\_tl.)

\l\_\_regex\_fresh\_thread\_bool  
\l\_\_regex\_empty\_success\_bool  
\\_regex\_if\_two\_empty\_matches:F When doing multiple matches, we need to avoid infinite loops where each iteration matches the same empty token list. When an empty token list is matched, the next successful match of the same empty token list is suppressed. We detect empty matches by setting \l\_\_regex\_fresh\_thread\_bool to true for threads which directly come from the start of the regex or from the \K command, and testing that boolean whenever a thread succeeds. The function \\_\\_regex\_if\_two\_empty\_matches:F is redefined at every match attempt, depending on whether the previous match was empty or not: if it was, then the function must cancel a purported success if it is empty and at the same spot as the previous match; otherwise, we definitely don't have two identical empty matches, so the function is \use:n.

25771 \bool\_new:N \l\_\_regex\_fresh\_thread\_bool

25772 \bool\_new:N \l\_\_regex\_empty\_success\_bool

25773 \cs\_new\_eq:NN \\_\\_regex\_if\_two\_empty\_matches:F \use:n

*(End definition for \l\_\_regex\_fresh\_thread\_bool, \l\_\_regex\_empty\_success\_bool, and \\_\\_regex\_if\_two\_empty\_matches:F.)*

\g\_\_regex\_success\_bool  
\l\_\_regex\_saved\_success\_bool  
\l\_\_regex\_match\_success\_bool

The boolean \l\_\_regex\_match\_success\_bool is true if the current match attempt was successful, and \g\_\_regex\_success\_bool is true if there was at least one successful match. This is the only global variable in this whole module, but we would need it to be local when matching a control sequence with \c{...}. This is done by saving the global variable into \l\_\_regex\_saved\_success\_bool, which is local, hence not affected by the changes due to inner regex functions.

```
25774 \bool_new:N \g__regex_success_bool
25775 \bool_new:N \l__regex_saved_success_bool
25776 \bool_new:N \l__regex_match_success_bool
```

*(End definition for \g\_\_regex\_success\_bool, \l\_\_regex\_saved\_success\_bool, and \l\_\_regex\_match\_success\_bool.)*

#### 41.5.2 Matching: framework

\\_\\_regex\_match:n  
\\_\\_regex\_match\_cs:n  
\\_\\_regex\_match\_init:

First store the query into \toks registers and arrays (see \\_\\_regex\_query\_set:nnn). Then initialize the variables that should be set once for each user function (even for multiple matches). Namely, the overall matching is not yet successful; none of the states should be marked as visited (\g\_\_regex\_state\_active\_intarray), and we start at step 0; we pretend that there was a previous match ending at the start of the query, which was not empty (to avoid smothering an empty match at the start). Once all this is set up, we are ready for the ride. Find the first match.

```
25777 \cs_new_protected:Npn __regex_match:n #1
25778 {
25779 \int_zero:N \l__regex_balance_int
25780 \int_set:Nn \l__regex_curr_pos_int { 2 * \l__regex_max_state_int }
25781 __regex_query_set:nnn { } { -1 } { -2 }
25782 \int_set_eq:NN \l__regex_min_pos_int \l__regex_curr_pos_int
25783 \tl_analysis_map_inline:nn {#1}
25784 { __regex_query_set:nnn {##1} {"##3} {##2} }
25785 \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
25786 __regex_query_set:nnn { } { -1 } { -2 }
25787 __regex_match_init:
25788 __regex_match_once:
25789 }
25790 \cs_new_protected:Npn __regex_match_cs:n #1
25791 {
25792 \int_zero:N \l__regex_balance_int
25793 \int_set:Nn \l__regex_curr_pos_int
25794 {
25795 \int_max:nn { 2 * \l__regex_max_state_int - \l__regex_min_state_int }
25796 { \l__regex_max_pos_int }
25797 + 1
25798 }
25799 __regex_query_set:nnn { } { -1 } { -2 }
25800 \int_set_eq:NN \l__regex_min_pos_int \l__regex_curr_pos_int
25801 \str_map_inline:nn {#1}
25802 {
25803 __regex_query_set:nnn { \exp_not:n {##1} }
25804 { \tl_if_blank:nTF {##1} { 10 } { 12 } }
```

```

25805 { '##1 }
25806 }
25807 \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
25808 __regex_query_set:nnn { } { -1 } { -2 }
25809 __regex_match_init:
25810 __regex_match_once:
25811 }
25812 \cs_new_protected:Npn __regex_match_init:
25813 {
25814 \bool_gset_false:N \g__regex_success_bool
25815 \int_step_inline:nnn
25816 \l__regex_min_state_int { \l__regex_max_state_int - 1 }
25817 {
25818 __kernel_intarray_gset:Nnn
25819 \g__regex_state_active_intarray {##1} { 1 }
25820 }
25821 \int_set_eq:NN \l__regex_min_active_int \l__regex_max_state_int
25822 \int_zero:N \l__regex_step_int
25823 \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
25824 \int_set:Nn \l__regex_min_submatch_int
25825 { 2 * \l__regex_max_state_int }
25826 \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
25827 \bool_set_false:N \l__regex_empty_success_bool
25828 }

```

(End definition for `\__regex_match:n`, `\__regex_match_cs:n`, and `\__regex_match_init:..`)

`\__regex_match_once:`: This function finds one match, then does some action defined by the `every_match` token list, which may recursively call `\__regex_match_once:..`. First initialize some variables: set the conditional which detects identical empty matches; this match attempt starts at the previous `success_pos`, is not yet successful, and has no submatches yet; clear the array of active threads, and put the starting state 0 in it. We are then almost ready to read our first token in the query, but we actually start one position earlier than the start, and get that token, to set `last_char` properly for word boundaries. Then call `\__regex_match_loop:..`, which runs through the query until the end or until a successful match breaks early.

```

25829 \cs_new_protected:Npn __regex_match_once:
25830 {
25831 \if_meaning:w \c_true_bool \l__regex_empty_success_bool
25832 \cs_set:Npn __regex_if_two_empty_matches:F
25833 {
25834 \int_compare:nNnF
25835 \l__regex_start_pos_int = \l__regex_curr_pos_int
25836 }
25837 \else:
25838 \cs_set_eq:NN __regex_if_two_empty_matches:F \use:n
25839 \fi:
25840 \int_set_eq:NN \l__regex_start_pos_int \l__regex_success_pos_int
25841 \bool_set_false:N \l__regex_match_success_bool
25842 \prop_clear:N \l__regex_curr_submatches_prop
25843 \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
25844 __regex_store_state:n { \l__regex_min_state_int }
25845 \int_set:Nn \l__regex_curr_pos_int
25846 { \l__regex_start_pos_int - 1 }

```

```

25847 __regex_query_get:
25848 __regex_match_loop:
25849 \l__regex_every_match_tl
25850 }

```

(End definition for `\__regex_match_once::`)

`\__regex_single_match:` For a single match, the overall success is determined by whether the only match attempt is a success. When doing multiple matches, the overall matching is successful as soon as any match succeeds. Perform the action #1, then find the next match.

```

25851 \cs_new_protected:Npn __regex_single_match:
25852 {
25853 \tl_set:Nn \l__regex_every_match_tl
25854 {
25855 \bool_gset_eq:NN
25856 \g__regex_success_bool
25857 \l__regex_match_success_bool
25858 }
25859 }
25860 \cs_new_protected:Npn __regex_multi_match:n #1
25861 {
25862 \tl_set:Nn \l__regex_every_match_tl
25863 {
25864 \if_meaning:w \c_true_bool \l__regex_match_success_bool
25865 \bool_gset_true:N \g__regex_success_bool
25866 #1
25867 \exp_after:wN __regex_match_once:
25868 \fi:
25869 }
25870 }

```

(End definition for `\__regex_single_match:` and `\__regex_multi_match:n`.)

`\__regex_match_loop:` At each new position, set some variables and get the new character and category from the query. Then unpack the array of active threads, and clear it by resetting its length (`max_active`). This results in a sequence of `\__regex_use_state_and_submatches:nn` {`(state)`} {`(prop)`}, and we consider those states one by one in order. As soon as a thread succeeds, exit the step, and, if there are threads to consider at the next position, and we have not reached the end of the string, repeat the loop. Otherwise, the last thread that succeeded is what `\__regex_match_once:` matches. We explain the `fresh_thread` business when describing `\__regex_action_wildcard::`.

```

25871 \cs_new_protected:Npn __regex_match_loop:
25872 {
25873 \int_add:Nn \l__regex_step_int { 2 }
25874 \int_incr:N \l__regex_curr_pos_int
25875 \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
25876 \int_set_eq:NN \l__regex_case_changed_char_int \c_max_int
25877 __regex_query_get:
25878 \use:x
25879 {
25880 \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
25881 \int_step_function:nnN
25882 { \l__regex_min_active_int }
25883 { \l__regex_max_active_int - 1 }

```

```

25884 __regex_match_one_active:n
25885 }
25886 \prg_break_point:
25887 \bool_set_false:N \l_regex_fresh_thread_bool
25888 \if_int_compare:w \l_regex_max_active_int > \l_regex_min_active_int
25889 \if_int_compare:w \l_regex_curr_pos_int < \l_regex_max_pos_int
25890 \exp_after:wN \exp_after:wN \exp_after:wN __regex_match_loop:
25891 \fi:
25892 \fi:
25893 }
25894 \cs_new:Npn __regex_match_one_active:n #1
25895 {
25896 __regex_use_state_and_submatches:nn
25897 { __kernel_intarray_item:Nn \g_regex_thread_state_intarray {#1} }
25898 { __regex_toks_use:w #1 }
25899 }

```

(End definition for `\_\_regex_match_loop:` and `\_\_regex_match_one_active:n.`)

#### `\_\_regex_query_set:nnn`

The arguments are: tokens that o and x expand to one token of the query, the catcode, and the character code. Store those, and the current brace balance (used later to check for overall brace balance) in a `\toks` register and some arrays, then update the `balance`.

```

25900 \cs_new_protected:Npn __regex_query_set:nnn #1#2#3
25901 {
25902 __kernel_intarray_gset:Nnn \g_regex_charcode_intarray
25903 { \l_regex_curr_pos_int } {#3}
25904 __kernel_intarray_gset:Nnn \g_regex_catcode_intarray
25905 { \l_regex_curr_pos_int } {#2}
25906 __kernel_intarray_gset:Nnn \g_regex_balance_intarray
25907 { \l_regex_curr_pos_int } { \l_regex_balance_int }
25908 __regex_toks_set:Nn \l_regex_curr_pos_int {#1}
25909 \int_incr:N \l_regex_curr_pos_int
25910 \if_case:w #2 \exp_stop_f:
25911 \or: \int_incr:N \l_regex_balance_int
25912 \or: \int_decr:N \l_regex_balance_int
25913 \fi:
25914 }

```

(End definition for `\_\_regex_query_set:nnn.`)

#### `\_\_regex_query_get:`

Extract the current character and category codes at the current position from the appropriate arrays.

```

25915 \cs_new_protected:Npn __regex_query_get:
25916 {
25917 \l_regex_curr_char_int
25918 = __kernel_intarray_item:Nn \g_regex_charcode_intarray
25919 { \l_regex_curr_pos_int } \scan_stop:
25920 \l_regex_curr_catcode_int
25921 = __kernel_intarray_item:Nn \g_regex_catcode_intarray
25922 { \l_regex_curr_pos_int } \scan_stop:
25923 }

```

(End definition for `\_\_regex_query_get:.`)

#### 41.5.3 Using states of the nfa

`\_\_regex\_use\_state:` Use the current NFA instruction. The state is initially marked as belonging to the current `step`: this allows normal free transition to repeat, but group-repeating transitions won't. Once we are done exploring all the branches it spawned, the state is marked as `step + 1`: any thread hitting it at that point will be terminated.

```

25924 \cs_new_protected:Npn __regex_use_state:
25925 {
25926 __kernel_intarray_gset:Nnn \g_regex_state_active_intarray
25927 { \l_regex_curr_state_int } { \l_regex_step_int }
25928 __regex_toks_use:w \l_regex_curr_state_int
25929 __kernel_intarray_gset:Nnn \g_regex_state_active_intarray
25930 { \l_regex_curr_state_int }
25931 { \int_eval:n { \l_regex_step_int + 1 } }
25932 }

```

(End definition for `\_\_regex\_use\_state:..`)

`\_\_regex\_use\_state\_and\_submatches:nn` This function is called as one item in the array of active threads after that array has been unpacked for a new step. Update the `current_state` and `current_submatches` and use the state if it has not yet been encountered at this step.

```

25933 \cs_new_protected:Npn __regex_use_state_and_submatches:nn #1 #2
25934 {
25935 \int_set:Nn \l_regex_curr_state_int {#1}
25936 \if_int_compare:w
25937 __kernel_intarray_item:Nn \g_regex_state_active_intarray
25938 { \l_regex_curr_state_int }
25939 < \l_regex_step_int
25940 \tl_set:Nn \l_regex_curr_submatches_prop {#2}
25941 \exp_after:wN __regex_use_state:
25942 \fi:
25943 \scan_stop:
25944 }

```

(End definition for `\_\_regex\_use\_state\_and\_submatches:nn.`)

#### 41.5.4 Actions when matching

`\_\_regex_action_start_wildcard:` For an unanchored match, state 0 has a free transition to the next and a costly one to itself, to repeat at the next position. To catch repeated identical empty matches, we need to know if a successful thread corresponds to an empty match. The instruction resetting `\l\_regex_fresh_thread_bool` may be skipped by a successful thread, hence we had to add it to `\_\_regex_match_loop:` too.

```

25945 \cs_new_protected:Npn __regex_action_start_wildcard:
25946 {
25947 \bool_set_true:N \l_regex_fresh_thread_bool
25948 __regex_action_free:n {1}
25949 \bool_set_false:N \l_regex_fresh_thread_bool
25950 __regex_action_cost:n {0}
25951 }

```

(End definition for `\_\_regex_action_start_wildcard:..`)

```
__regex_action_free:n
__regex_action_free_group:n
__regex_action_free_aux:nn
```

These functions copy a thread after checking that the NFA state has not already been used at this position. If not, store submatches in the new state, and insert the instructions for that state in the input stream. Then restore the old value of `\l_\_regex_curr_state_int` and of the current submatches. The two types of free transitions differ by how they test that the state has not been encountered yet: the `group` version is stricter, and will not use a state if it was used earlier in the current thread, hence forcefully breaking the loop, while the “normal” version will revisit a state even within the thread itself.

```
25952 \cs_new_protected:Npn __regex_action_free:n
25953 { __regex_action_free_aux:nn { > \l__regex_step_int \else: } }
25954 \cs_new_protected:Npn __regex_action_free_group:n
25955 { __regex_action_free_aux:nn { < \l__regex_step_int } }
25956 \cs_new_protected:Npn __regex_action_free_aux:nn #1#2
25957 {
25958 \use:x
25959 {
25960 \int_add:Nn \l__regex_curr_state_int {#2}
25961 \exp_not:n
25962 {
25963 \if_int_compare:w
25964 __kernel_intarray_item:Nn \g__regex_state_active_intarray
25965 { \l__regex_curr_state_int }
25966 #1
25967 \exp_after:wN __regex_use_state:
25968 \fi:
25969 }
25970 \int_set:Nn \l__regex_curr_state_int
25971 { \int_use:N \l__regex_curr_state_int }
25972 \tl_set:Nn \exp_not:N \l__regex_curr_submatches_prop
25973 { \exp_not:o \l__regex_curr_submatches_prop }
25974 }
25975 }
}
```

(End definition for `\_\_regex\_action\_free:n`, `\_\_regex\_action\_free\_group:n`, and `\_\_regex\_action\_free\_aux:nn`.)

```
__regex_action_cost:n
```

A transition which consumes the current character and shifts the state by #1. The resulting state is stored in the appropriate array for use at the next position, and we also store the current submatches.

```
25976 \cs_new_protected:Npn __regex_action_cost:n #1
25977 {
25978 \exp_args:Nx __regex_store_state:n
25979 { \int_eval:n { \l__regex_curr_state_int + #1 } }
25980 }
```

(End definition for `\_\_regex\_action\_cost:n`.)

```
__regex_store_state:n
__regex_store_submatches:
```

Put the given state in `\g_\_regex_thread_state_intarray`, and increment the length of the array. Also store the current submatch in the appropriate `\toks`.

```
25981 \cs_new_protected:Npn __regex_store_state:n #1
25982 {
25983 __regex_store_submatches:
25984 __kernel_intarray_gset:Nnn \g__regex_thread_state_intarray
25985 { \l__regex_max_active_int } {#1}
25986 \int_incr:N \l__regex_max_active_int
```

```

25987 }
25988 \cs_new_protected:Npn __regex_store_submatches:
25989 {
25990 __regex_toks_set:No \l__regex_max_active_int
25991 { \l__regex_curr_submatches_prop }
25992 }
```

(End definition for `\__regex_store_state:n` and `\__regex_store_submatches:..`)

`\__regex_disable_submatches:` Some user functions don't require tracking submatches. We get a performance improvement by simply defining the relevant functions to remove their argument and do nothing with it.

```

25993 \cs_new_protected:Npn __regex_disable_submatches:
25994 {
25995 \cs_set_protected:Npn __regex_store_submatches: { }
25996 \cs_set_protected:Npn __regex_action_submatch:n ##1 { }
25997 }
```

(End definition for `\__regex_disable_submatches:..`)

`\__regex_action_submatch:n` Update the current submatches with the information from the current position. Maybe a bottleneck.

```

25998 \cs_new_protected:Npn __regex_action_submatch:n #1
25999 {
26000 \prop_put:Nno \l__regex_curr_submatches_prop {#1}
26001 { \int_use:N \l__regex_curr_pos_int }
26002 }
```

(End definition for `\__regex_action_submatch:n`)

`\__regex_action_success:` There is a successful match when an execution path reaches the last state in the NFA, unless this marks a second identical empty match. Then mark that there was a successful match; it is empty if it is “fresh”; and we store the current position and submatches. The current step is then interrupted with `\prg_break:`, and only paths with higher precedence are pursued further. The values stored here may be overwritten by a later success of a path with higher precedence.

```

26003 \cs_new_protected:Npn __regex_action_success:
26004 {
26005 __regex_if_two_empty_matches:F
26006 {
26007 \bool_set_true:N \l__regex_match_success_bool
26008 \bool_set_eq:NN \l__regex_empty_success_bool
26009 \l__regex_fresh_thread_bool
26010 \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
26011 \prop_set_eq:NN \l__regex_success_submatches_prop
26012 \l__regex_curr_submatches_prop
26013 \prg_break:
26014 }
26015 }
```

(End definition for `\__regex_action_success:..`)

## 41.6 Replacement

### 41.6.1 Variables and helpers used in replacement

\l\_regex\_replacement\_csnames\_int

The behaviour of closing braces inside a replacement text depends on whether a sequences \c{ or \u{ has been encountered. The number of “open” such sequences that should be closed by } is stored in \l\_regex\_replacement\_csnames\_int, and decreased by 1 by each }.

```
26016 \int_new:N \l_regex_replacement_csnames_int
```

(End definition for \l\_regex\_replacement\_csnames\_int.)

\l\_regex\_replacement\_category\_t1  
\l\_regex\_replacement\_category\_seq

This sequence of letters is used to correctly restore categories in nested constructions such as \cL(abc\cD(\_))d).

```
26017 \tl_new:N \l_regex_replacement_category_t1
```

```
26018 \seq_new:N \l_regex_replacement_category_seq
```

(End definition for \l\_regex\_replacement\_category\_t1 and \l\_regex\_replacement\_category\_seq.)

\l\_regex\_balance\_t1

This token list holds the replacement text for \\_\_regex\_replacement\_balance\_one\_match:n while it is being built incrementally.

```
26019 \tl_new:N \l_regex_balance_t1
```

(End definition for \l\_regex\_balance\_t1.)

\\_\_regex\_replacement\_balance\_one\_match:n

This expects as an argument the first index of a set of entries in \g\_regex\_submatch\_begin\_intarray (and related arrays) which hold the submatch information for a given match. It can be used within an integer expression to obtain the brace balance incurred by performing the replacement on that match. This combines the braces lost by removing the match, braces added by all the submatches appearing in the replacement, and braces appearing explicitly in the replacement. Even though it is always redefined before use, we initialize it as for an empty replacement. An important property is that concatenating several calls to that function must result in a valid integer expression (hence a leading + in the actual definition).

```
26020 \cs_new:Npn __regex_replacement_balance_one_match:n #1
26021 { - __regex_submatch_balance:n {#1} }
```

(End definition for \\_\_regex\_replacement\_balance\_one\_match:n.)

\\_\_regex\_replacement\_do\_one\_match:n

The input is the same as \\_\_regex\_replacement\_balance\_one\_match:n. This function is redefined to expand to the part of the token list from the end of the previous match to a given match, followed by the replacement text. Hence concatenating the result of this function with all possible arguments (one call for each match), as well as the range from the end of the last match to the end of the string, produces the fully replaced token list. The initialization does not matter, but (as an example) we set it as for an empty replacement.

```
26022 \cs_new:Npn __regex_replacement_do_one_match:n #1
26023 {
26024 __regex_query_range:nn
26025 { __kernel_intarray_item:Nn \g_regex_submatch_prev_intarray {#1} }
26026 { __kernel_intarray_item:Nn \g_regex_submatch_begin_intarray {#1} }
26027 }
```

(End definition for \\_\_regex\_replacement\_do\_one\_match:n.)

\\_regex\_replacement\_exp\_not:N

This function lets us navigate around the fact that the primitive `\exp_not:n` requires a braced argument. As far as I can tell, it is only needed if the user tries to include in the replacement text a control sequence set equal to a macro parameter character, such as `\c_parameter_token`. Indeed, within an x-expanding assignment, `\exp_not:N #` behaves as a single `#`, whereas `\exp_not:n {#}` behaves as a doubled `##`.

```
26028 \cs_new:Npn _regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
```

(End definition for `\_regex_replacement_exp_not:N`.)

#### 41.6.2 Query and brace balance

\\_regex\_query\_range:nn

\\_regex\_query\_range\_loop:ww

When it is time to extract submatches from the token list, the various tokens are stored in `\toks` registers numbered from `\l_Regex_min_pos_int` inclusive to `\l_Regex_max_pos_int` exclusive. The function `\_regex_query_range:nn {<min>} {<max>}` unpacks registers from the position `<min>` to the position `<max> - 1` included. Once this is expanded, a second x-expansion results in the actual tokens from the query. That second expansion is only done by user functions at the very end of their operation, after checking (and correcting) the brace balance first.

```
26029 \cs_new:Npn _regex_query_range:nn #1#2
26030 {
26031 \exp_after:wN _regex_query_range_loop:ww
26032 \int_value:w _regex_int_eval:w #1 \exp_after:wN ;
26033 \int_value:w _regex_int_eval:w #2 ;
26034 \prg_break_point:
26035 }
26036 \cs_new:Npn _regex_query_range_loop:ww #1 ; #2 ;
26037 {
26038 \if_int_compare:w #1 < #2 \exp_stop_f:
26039 \else:
26040 \exp_after:wN \prg_break:
26041 \fi:
26042 _regex_toks_use:w #1 \exp_stop_f:
26043 \exp_after:wN _regex_query_range_loop:ww
26044 \int_value:w _regex_int_eval:w #1 + 1 ; #2 ;
26045 }
```

(End definition for `\_regex_query_range:nn` and `\_regex_query_range_loop:ww`.)

\\_regex\_query\_submatch:n

Find the start and end positions for a given submatch (of a given match).

```
26046 \cs_new:Npn _regex_query_submatch:n #1
26047 {
26048 _regex_query_range:nn
26049 { _kernel_intarray_item:Nn \g_regex_submatch_begin_intarray {#1} }
26050 { _kernel_intarray_item:Nn \g_regex_submatch_end_intarray {#1} }
26051 }
```

(End definition for `\_regex_query_submatch:n`.)

\\_regex\_submatch\_balance:n

Every user function must result in a balanced token list (unbalanced token lists cannot be stored by TeX). When we unpacked the query, we kept track of the brace balance, hence the contribution from a given range is the difference between the brace balances at the `<max pos>` and `<min pos>`. These two positions are found in the corresponding “submatch” arrays.

```

26052 \cs_new_protected:Npn __regex_submatch_balance:n #1
26053 {
26054 \int_eval:n
26055 {
26056 \int_compare:nNnTF
26057 {
26058 __kernel_intarray_item:Nn
26059 \g__regex_submatch_end_intarray {#1}
26060 }
26061 = 0
26062 { 0 }
26063 {
26064 __kernel_intarray_item:Nn \g__regex_balance_intarray
26065 {
26066 __kernel_intarray_item:Nn
26067 \g__regex_submatch_end_intarray {#1}
26068 }
26069 }
26070 -
26071 \int_compare:nNnTF
26072 {
26073 __kernel_intarray_item:Nn
26074 \g__regex_submatch_begin_intarray {#1}
26075 }
26076 = 0
26077 { 0 }
26078 {
26079 __kernel_intarray_item:Nn \g__regex_balance_intarray
26080 {
26081 __kernel_intarray_item:Nn
26082 \g__regex_submatch_begin_intarray {#1}
26083 }
26084 }
26085 }
26086 }

```

(End definition for `\__regex_submatch_balance:n`.)

#### 41.6.3 Framework

The replacement text is built incrementally. We keep track in `\l__regex_balance_int` of the balance of explicit begin- and end-group tokens and we store in `\l__regex_balance_tl` some code to compute the brace balance from submatches (see its description). Detect unescaped right braces, and escaped characters, with trailing `\prg_do_nothing`: because some of the later function look-ahead. Once the whole replacement text has been parsed, make sure that there is no open csname. Finally, define the `balance_one_match` and `do_one_match` functions.

```

26087 \cs_new_protected:Npn __regex_replacement:n #1
26088 {
26089 \group_begin:
26090 \tl_build_begin:N \l__regex_build_tl
26091 \int_zero:N \l__regex_balance_int
26092 \tl_clear:N \l__regex_balance_tl
26093 __regex_escape_use:nnnn

```

```

26094 {
26095 \if_charcode:w \c_right_brace_str ##1
26096 __regex_replacement_rbrace:N
26097 \else:
26098 __regex_replacement_normal:n
26099 \fi:
26100 ##1
26101 }
26102 { __regex_replacement_escaped:N ##1 }
26103 { __regex_replacement_normal:n ##1 }
26104 {##1}
26105 \prg_do_nothing: \prg_do_nothing:
26106 \if_int_compare:w \l__regex_replacement_csnames_int > 0 \exp_stop_f:
26107 __kernel_msg_error:nnx { kernel } { replacement-missing-rbrace }
26108 { \int_use:N \l__regex_replacement_csnames_int }
26109 \tl_build_put_right:Nx \l__regex_build_tl
26110 { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
26111 \fi:
26112 \seq_if_empty:NF \l__regex_replacement_category_seq
26113 {
26114 __kernel_msg_error:nnx { kernel } { replacement-missing-rparen }
26115 { \seq_count:N \l__regex_replacement_category_seq }
26116 \seq_clear:N \l__regex_replacement_category_seq
26117 }
26118 \cs_gset:Npx __regex_replacement_balance_one_match:n ##1
26119 {
26120 + \int_use:N \l__regex_balance_int
26121 \l__regex_balance_tl
26122 - __regex_submatch_balance:n {##1}
26123 }
26124 \tl_build_end:N \l__regex_build_tl
26125 \exp_args:NNo
26126 \group_end:
26127 __regex_replacement_aux:n \l__regex_build_tl
26128 }
26129 \cs_new_protected:Npn __regex_replacement_aux:n #1
26130 {
26131 \cs_set:Npn __regex_replacement_do_one_match:n ##1
26132 {
26133 __regex_query_range:nn
26134 {
26135 __kernel_intarray_item:Nn
26136 \g__regex_submatch_prev_intarray {##1}
26137 }
26138 {
26139 __kernel_intarray_item:Nn
26140 \g__regex_submatch_begin_intarray {##1}
26141 }
26142 #1
26143 }
26144 }

```

(End definition for `\__regex_replacement:n` and `\__regex_replacement_aux:n`.)

`\__regex_replacement_normal:n` Most characters are simply sent to the output by `\tl_build_put_right:Nn`, unless a

particular category code has been requested: then `\_\_regex_replacement_c_A:w` or a similar auxiliary is called. One exception is right parentheses, which restore the category code in place before the group started. Note that the sequence is non-empty there: it contains an empty entry corresponding to the initial value of `\l\_\_regex_replacement_category_tl`.

```

26145 \cs_new_protected:Npn __regex_replacement_normal:n #1
26146 {
26147 \tl_if_empty:NTF \l__regex_replacement_category_tl
26148 { \tl_build_put_right:Nn \l__regex_build_tl {#1} }
26149 { % (
26150 \token_if_eq_charcode:NNTF #1)
26151 {
26152 \seq_pop:NN \l__regex_replacement_category_seq
26153 \l__regex_replacement_category_tl
26154 }
26155 {
26156 \use:c
26157 {
26158 __regex_replacement_c_
26159 \l__regex_replacement_category_tl :w
26160 }
26161 __regex_replacement_normal:n {#1}
26162 }
26163 }
26164 }
```

(End definition for `\_\_regex_replacement_normal:n`.)

`\_\_regex_replacement_escaped:N`

As in parsing a regular expression, we use an auxiliary built from `#1` if defined. Otherwise, check for escaped digits (standing from submatches from 0 to 9): anything else is a raw character. We use `\token_to_str:N` to give spaces the right category code.

```

26165 \cs_new_protected:Npn __regex_replacement_escaped:N #1
26166 {
26167 \cs_if_exist_use:cF { __regex_replacement_#1:w }
26168 {
26169 \if_int_compare:w 1 < 1#1 \exp_stop_f:
26170 __regex_replacement_put_submatch:n {#1}
26171 \else:
26172 \exp_args:No __regex_replacement_normal:n
26173 { \token_to_str:N #1 }
26174 \fi:
26175 }
26176 }
```

(End definition for `\_\_regex_replacement_escaped:N`.)

#### 41.6.4 Submatches

`\_\_regex_replacement_put_submatch:n`

Insert a submatch in the replacement text. This is dropped if the submatch number is larger than the number of capturing groups. Unless the submatch appears inside a `\c{...}` or `\u{...}` construction, it must be taken into account in the brace balance. Later on, `##1` will be replaced by a pointer to the 0-th submatch for a given match.

There is an \exp\_not:N here as at the point-of-use of \l\_\_regex\_balance\_tl there is an x-type expansion which is needed to get #1 in correctly.

```

26177 \cs_new_protected:Npn __regex_replacement_put_submatch:n #1
26178 {
26179 \if_int_compare:w #1 < \l__regex_capturing_group_int
26180 \tl_build_put_right:Nn \l__regex_build_tl
26181 { __regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
26182 \if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f:
26183 \tl_put_right:Nn \l__regex_balance_tl
26184 {
26185 + __regex_submatch_balance:n
26186 { \exp_not:N \int_eval:n { #1 + ##1 } }
26187 }
26188 \fi:
26189 \fi:
26190 }

```

*(End definition for \\_\_regex\_replacement\_put\_submatch:n.)*

\\_\_regex\_replacement\_g:w  
\\_\_regex\_replacement\_g\_digits>NN

Grab digits for the \g escape sequence in a primitive assignment to the integer \l\_\_regex\_internal\_a\_int. At the end of the run of digits, check that it ends with a right brace.

```

26191 \cs_new_protected:Npn __regex_replacement_g:w #1#2
26192 {
26193 __regex_two_if_eq>NNNNTF
26194 #1 #2 __regex_replacement_normal:n \c_left_brace_str
26195 { \l__regex_internal_a_int = __regex_replacement_g_digits>NN }
26196 { __regex_error>NNN g #1 #2 }
26197 }
26198 \cs_new:Npn __regex_replacement_g_digits>NN #1#2
26199 {
26200 \token_if_eq_meaning>NNNTF #1 __regex_replacement_normal:n
26201 {
26202 \if_int_compare:w 1 < 1#2 \exp_stop_f:
26203 #2
26204 \exp_after:wN \use_i:nnn
26205 \exp_after:wN __regex_replacement_g_digits>NN
26206 \else:
26207 \exp_stop_f:
26208 \exp_after:wN __regex_error>NNN
26209 \exp_after:wN g
26210 \fi:
26211 }
26212 {
26213 \exp_stop_f:
26214 \if_meaning:w __regex_replacement_rbrace:N #1
26215 \exp_args:No __regex_replacement_put_submatch:n
26216 { \int_use:N \l__regex_internal_a_int }
26217 \exp_after:wN \use_none:nn
26218 \else:
26219 \exp_after:wN __regex_error>NNN
26220 \exp_after:wN g
26221 \fi:
26222 }

```

```

26223 #1 #2
26224 }
(End definition for __regex_replacement_g:w and __regex_replacement_g:digits>NN.)
```

#### 41.6.5 Csnames in replacement

\\_\\_regex\\_replacement\_c:w

\c may only be followed by an unescaped character. If followed by a left brace, start a control sequence by calling an auxiliary common with \u. Otherwise test whether the category is known; if it is not, complain.

```

26225 \cs_new_protected:Npn __regex_replacement_c:w #1#2
26226 {
26227 \token_if_eq_meaning:NNTF #1 __regex_replacement_normal:n
26228 {
26229 \exp_after:wN \token_if_eq_charcode:NNTF \c_left_brace_str #2
26230 { __regex_replacement_cu_aux:Nw __regex_replacement_exp_not:N }
26231 {
26232 \cs_if_exist:cTF { __regex_replacement_c:#2:w }
26233 { __regex_replacement_cat:NNN #2 }
26234 { __regex_replacement_error:NNN c #1#2 }
26235 }
26236 }
26237 { __regex_replacement_error:NNN c #1#2 }
26238 }
```

(End definition for \\_\\_regex\\_replacement\\_c:w.)

\\_\\_regex\\_replacement\_cu\_aux:Nw

Start a control sequence with \cs:w, protected from expansion by #1 (either \\_\\_regex\\_replacement\\_exp\\_not:N or \exp\\_not:V), or turned to a string by \tl\_to\_str:V if inside another csname construction \c or \u. We use \tl\_to\_str:V rather than \tl\_to\_str:N to deal with integers and other registers.

```

26239 \cs_new_protected:Npn __regex_replacement_cu_aux:Nw #1
26240 {
26241 \if_case:w \l__regex_replacement_csnames_int
26242 \tl_build_put_right:Nn \l__regex_build_tl
26243 { \exp_not:n { \exp_after:wN #1 \cs:w } }
26244 \else:
26245 \tl_build_put_right:Nn \l__regex_build_tl
26246 { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
26247 \fi:
26248 \int_incr:N \l__regex_replacement_csnames_int
26249 }
```

(End definition for \\_\\_regex\\_replacement\\_cu\\_aux:Nw.)

\\_\\_regex\\_replacement\_u:w

Check that \u is followed by a left brace. If so, start a control sequence with \cs:w, which is then unpacked either with \exp\\_not:V or \tl\_to\\_str:V depending on the current context.

```

26250 \cs_new_protected:Npn __regex_replacement_u:w #1#2
26251 {
26252 __regex_two_if_eq>NNNNTF
26253 #1 #2 __regex_replacement_normal:n \c_left_brace_str
26254 { __regex_replacement_cu_aux:Nw \exp_not:V }
26255 { __regex_replacement_error:NNN u #1#2 }
26256 }
```

(End definition for `\_regex_replacement_u:w`.)

`\_regex_replacement_rbrace:N` Within a `\c{...}` or `\u{...}` construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

```
26257 \cs_new_protected:Npn _regex_replacement_rbrace:N #1
26258 {
26259 \if_int_compare:w \l__regex_replacement_csnames_int > 0 \exp_stop_f:
26260 \tl_build_put_right:Nn \l__regex_build_tl { \cs_end: }
26261 \int_decr:N \l__regex_replacement_csnames_int
26262 \else:
26263 _regex_replacement_normal:n {#1}
26264 \fi:
26265 }
```

(End definition for `\_regex_replacement_rbrace:N`.)

#### 41.6.6 Characters in replacement

`\_regex_replacement_cat:NNN` Here, `#1` is a letter among `BEMTPUDSLOA` and `#2#3` denote the next character. Complain if we reach the end of the replacement or if the construction appears inside `\c{...}` or `\u{...}`, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

```
26266 \cs_new_protected:Npn _regex_replacement_cat:NNN #1#2#3
26267 {
26268 \token_if_eq_meaning:NNTF \prg_do_nothing: #3
26269 { _kernel_msg_error:nn { kernel } { replacement-catcode-end } }
26270 {
26271 \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
26272 {
26273 _kernel_msg_error:nnnn
26274 { kernel } { replacement-catcode-in-cs } {#1} {#3}
26275 #2 #3
26276 }
26277 {
26278 _regex_two_if_eq:NNNNTF #2 #3 _regex_replacement_normal:n (
26279 {
26280 \seq_push:NV \l__regex_replacement_category_seq
26281 \l__regex_replacement_category_tl
26282 \tl_set:Nn \l__regex_replacement_category_tl {#1}
26283 }
26284 {
26285 \token_if_eq_meaning:NNT #2 _regex_replacement_escaped:N
26286 {
26287 _regex_char_if_alphanumeric:NTF #3
26288 {
26289 _kernel_msg_error:nnnn
26290 { kernel } { replacement-catcode-escaped }
26291 {#1} {#3}
26292 }
26293 { }
26294 }
26295 \use:c { __regex_replacement_c_#1:w } #2 #3
26296 }
26297 }
```

```
26298 }
26299 }
```

(End definition for `\_regex_replacement_cat:NNN`.)

We now need to change the category code of the null character many times, hence work in a group. The catcode-specific macros below are defined in alphabetical order; if you are trying to understand the code, start from the end of the alphabet as those categories are simpler than active or begin-group.

```
26300 \group_begin:
```

`\_regex_replacement_char:nNN` The only way to produce an arbitrary character–catcode pair is to use the `\lowercase` or `\uppercase` primitives. This is a wrapper for our purposes. The first argument is the null character with various catcodes. The second and third arguments are grabbed from the input stream: #3 is the character whose character code to reproduce. We could use `\char_generate:nn` but only for some catcodes (active characters and spaces are not supported).

```
26301 \cs_new_protected:Npn _regex_replacement_char:nNN #1#2#3
26302 {
26303 \tex_lccode:D 0 = '#3 \scan_stop:
26304 \tex_lowercase:D { \tl_build_put_right:Nn \l_regex_build_t1 {#1} }
26305 }
```

(End definition for `\_regex_replacement_char:nNN`.)

`\_regex_replacement_c_A:w` For an active character, expansion must be avoided, twice because we later do two x-expansions, to unpack `\toks` for the query, and to expand their contents to tokens of the query.

```
26306 \char_set_catcode_active:N \^@0
26307 \cs_new_protected:Npn _regex_replacement_c_A:w
26308 { _regex_replacement_char:nNN { \exp_not:n { \exp_not:N \^@0 } } }
```

(End definition for `\_regex_replacement_c_A:w`.)

`\_regex_replacement_c_B:w` An explicit begin-group token increases the balance, unless within a `\c{...}` or `\u{...}` construction. Add the desired begin-group character, using the standard `\if_false:` trick. We eventually x-expand twice. The first time must yield a balanced token list, and the second one gives the bare begin-group token. The `\exp_after:wN` is not strictly needed, but is more consistent with l3tl-analysis.

```
26309 \char_set_catcode_group_begin:N \^@0
26310 \cs_new_protected:Npn _regex_replacement_c_B:w
26311 {
26312 \if_int_compare:w \l_regex_replacement_cnames_int = 0 \exp_stop_f:
26313 \int_incr:N \l_regex_balance_int
26314 \fi:
26315 _regex_replacement_char:nNN
26316 { \exp_not:n { \exp_after:wN \^@0 \if_false: } \fi: } }
26317 }
```

(End definition for `\_regex_replacement_c_B:w`.)

|                                           |                                                                                                                                                                                                                                                                                                                                                                                                       |
|-------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>\_\_regex\_replacement_c_C:w</code> | This is not quite catcode-related: when the user requests a character with category “control sequence”, the one-character control symbol is returned. As for the active character, we prepare for two x-expansions.                                                                                                                                                                                   |
|                                           | <pre> 26318   \cs_new_protected:Npn \_\_regex\_replacement_c_C:w #1#2 26319     { 26320       \tl_build_put_right:Nn \l_\_regex_build_tl 26321         { \exp_not:N \exp_not:N \exp_not:c {#2} } 26322     } </pre>                                                                                                                                                                                   |
|                                           | <i>(End definition for \_\_regex\_replacement_c_C:w.)</i>                                                                                                                                                                                                                                                                                                                                             |
| <code>\_\_regex\_replacement_c_D:w</code> | Subscripts fit the mould: \lowercase the null byte with the correct category.                                                                                                                                                                                                                                                                                                                         |
|                                           | <pre> 26323   \char_set_catcode_math_subscript:N \^\^@  26324   \cs_new_protected:Npn \_\_regex\_replacement_c_D:w 26325     { \_\_regex\_replacement_char:nNN { ^^@ } } </pre>                                                                                                                                                                                                                       |
|                                           | <i>(End definition for \_\_regex\_replacement_c_D:w.)</i>                                                                                                                                                                                                                                                                                                                                             |
| <code>\_\_regex\_replacement_c_E:w</code> | Similar to the begin-group case, the second x-expansion produces the bare end-group token.                                                                                                                                                                                                                                                                                                            |
|                                           | <pre> 26326   \char_set_catcode_group_end:N \^\^@  26327   \cs_new_protected:Npn \_\_regex\_replacement_c_E:w 26328     { 26329       \if_int_compare:w \l_\_regex_replacement_csnames_int = 0 \exp_stop_f: 26330         \int_decr:N \l_\_regex_balance_int 26331       \fi: 26332       \_\_regex\_replacement_char:nNN 26333         { \exp_not:n { \if_false: { \fi: ^^@ } } } 26334     } </pre> |
|                                           | <i>(End definition for \_\_regex\_replacement_c_E:w.)</i>                                                                                                                                                                                                                                                                                                                                             |
| <code>\_\_regex\_replacement_c_L:w</code> | Simply \lowercase a letter null byte to produce an arbitrary letter.                                                                                                                                                                                                                                                                                                                                  |
|                                           | <pre> 26335   \char_set_catcode_letter:N \^\^@  26336   \cs_new_protected:Npn \_\_regex\_replacement_c_L:w 26337     { \_\_regex\_replacement_char:nNN { ^^@ } } </pre>                                                                                                                                                                                                                               |
|                                           | <i>(End definition for \_\_regex\_replacement_c_L:w.)</i>                                                                                                                                                                                                                                                                                                                                             |
| <code>\_\_regex\_replacement_c_M:w</code> | No surprise here, we lowercase the null math toggle.                                                                                                                                                                                                                                                                                                                                                  |
|                                           | <pre> 26338   \char_set_catcode_math_toggle:N \^\^@  26339   \cs_new_protected:Npn \_\_regex\_replacement_c_M:w 26340     { \_\_regex\_replacement_char:nNN { ^^@ } } </pre>                                                                                                                                                                                                                          |
|                                           | <i>(End definition for \_\_regex\_replacement_c_M:w.)</i>                                                                                                                                                                                                                                                                                                                                             |
| <code>\_\_regex\_replacement_c_0:w</code> | Lowercase an other null byte.                                                                                                                                                                                                                                                                                                                                                                         |
|                                           | <pre> 26341   \char_set_catcode_other:N \^\^@  26342   \cs_new_protected:Npn \_\_regex\_replacement_c_0:w 26343     { \_\_regex\_replacement_char:nNN { ^^@ } } </pre>                                                                                                                                                                                                                                |
|                                           | <i>(End definition for \_\_regex\_replacement_c_0:w.)</i>                                                                                                                                                                                                                                                                                                                                             |

`\_\_regex\_replacement_c_P:w` For macro parameters, expansion is a tricky issue. We need to prepare for two `x`-expansions and passing through various macro definitions. Note that we cannot replace one `\exp_not:n` by doubling the macro parameter characters because this would misbehave if a mischievous user asks for `\cf{\cP\#}`, since that macro parameter character would be doubled.

```
26344 \char_set_catcode_parameter:N `^` @
26345 \cs_new_protected:Npn __regex_replacement_c_P:w
26346 {
26347 __regex_replacement_char:nNN
26348 { \exp_not:n { \exp_not:n { `^` @ `^` @ `^` @ } } }
26349 }
```

(End definition for `\_regex_replacement_c_P:w`.)

`\_regex_replacement_c_S:w` Spaces are normalized on input by TeX to have character code 32. It is in fact impossible to get a token with character code 0 and category code 10. Hence we use 32 instead of 0 as our base character.

```
26350 \cs_new_protected:Npn __regex_replacement_c_S:w #1#2
26351 {
26352 \if_int_compare:w '#2 = 0 \exp_stop_f:
26353 __kernel_msg_error:nn { kernel } { replacement-null-space }
26354 \fi:
26355 \tex_lccode:D '\ = '#2 \scan_stop:
26356 \tex_lowercase:D { \tl_build_put_right:Nn \l__regex_build_tl {~} }
26357 }
```

(End definition for `\_regex_replacement_c_S:w`.)

\\_\\_regex\\_replacement\\_c\\_T:w No surprise for alignment tabs here. Those are surrounded by the appropriate braces whenever necessary, hence they don't cause trouble in alignment settings.

```
26358 \char_set_catcode_alignment:N `^` @
26359 \cs_new_protected:Npn __regex_replacement_c_T:w
26360 { __regex_replacement_char:nNN { `^` @ } }
```

(End definition for \\_regex\\_replacement\_c\_T:w.)

`\_\_regex\_replacement_c_U:w` Simple call to `\_\_regex\_replacement_char:nNN` which lowercases the math superscript `^~@`.

```
26361 \char_set_catcode_math_superscript:N ^@\n26362 \cs_new_protected:Npn __regex_replacement_c_U:w\n26363 { __regex_replacement_char:nNN { ^@\ } }
```

(End definition for \\_reg\\_replacement\_c\_U:w.)

Restore the catcode of the null byte.

26364 \group\_end:

#### 41.6.7 An error

\\_regex\_replacement\_error:NNN  
Simple error reporting by calling one of the messages `replacement-c`, `replacement-g`, or `replacement-u`.

```
26365 \cs_new_protected:Npn _regex_replacement_error:NNN #1#2#3
26366 {
26367 \kernel_msg_error:nnx { kernel } { replacement-#1 } {#3}
26368 #2 #3
26369 }
```

(End definition for `\_regex_replacement_error:NNN`.)

### 41.7 User functions

\regex\_new:N Before being assigned a sensible value, a regex variable matches nothing.

```
26370 \cs_new_protected:Npn \regex_new:N #1
26371 { \cs_new_eq:NN #1 \c_regex_no_match_regex }
```

(End definition for `\regex_new:N`. This function is documented on page 233.)

\l\_tmpa\_regex The usual scratch space.

```
26372 \regex_new:N \l_tmpa_regex
26373 \regex_new:N \l_tmpb_regex
26374 \regex_new:N \g_tmpa_regex
26375 \regex_new:N \g_tmpb_regex
```

(End definition for `\l_tmpa_regex` and others. These variables are documented on page 235.)

\regex\_set:Nn Compile, then store the result in the user variable with the appropriate assignment function.

```
26376 \cs_new_protected:Npn \regex_set:Nn #1#2
26377 {
26378 __regex_compile:n {#2}
26379 \tl_set_eq:NN #1 \l__regex_internal_regex
26380 }
26381 \cs_new_protected:Npn \regex_gset:Nn #1#2
26382 {
26383 __regex_compile:n {#2}
26384 \tl_gset_eq:NN #1 \l__regex_internal_regex
26385 }
26386 \cs_new_protected:Npn \regex_const:Nn #1#2
26387 {
26388 __regex_compile:n {#2}
26389 \tl_const:Nx #1 { \exp_not:o \l__regex_internal_regex }
26390 }
```

(End definition for `\regex_set:Nn`, `\regex_gset:Nn`, and `\regex_const:Nn`. These functions are documented on page 233.)

\regex\_show:N User functions: the `n` variant requires compilation first. Then show the variable with some appropriate text. The auxiliary is defined in a different section.

```
26391 \cs_new_protected:Npn \regex_show:n #1
26392 {
26393 __regex_compile:n {#1}
26394 __regex_show:N \l__regex_internal_regex
```

```

26395 \msg_show:nxxxx { LaTeX / kernel } { show-regex }
26396 { \tl_to_str:n {#1} } { }
26397 { \l__regex_internal_a_tl } { }
26398 }
26399 \cs_new_protected:Npn \regex_show:N #1
26400 {
26401 __kernel_chk_defined:NT #1
26402 {
26403 __regex_show:N #1
26404 \msg_show:nxxxx { LaTeX / kernel } { show-regex }
26405 { } { \token_to_str:N #1 }
26406 { \l__regex_internal_a_tl } { }
26407 }
26408 }

```

(End definition for `\regex_show:N` and `\regex_show:n`. These functions are documented on page [233](#).)

**\regex\_match:nnTF** Those conditionals are based on a common auxiliary defined later. Its first argument builds the NFA corresponding to the regex, and the second argument is the query token list. Once we have performed the match, convert the resulting boolean to `\prg_return_true:` or `false`.

```

26409 \prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
26410 {
26411 __regex_if_match:nn { __regex_build:n {#1} } {#2}
26412 __regex_return:
26413 }
26414 \prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
26415 {
26416 __regex_if_match:nn { __regex_build:N #1 } {#2}
26417 __regex_return:
26418 }

```

(End definition for `\regex_match:nnTF` and `\regex_match:NnTF`. These functions are documented on page [233](#).)

**\regex\_count:nnN** Again, use an auxiliary whose first argument builds the NFA.

**\regex\_count:NnN**

```

26419 \cs_new_protected:Npn \regex_count:nnN #1
26420 { __regex_count:nnN { __regex_build:n {#1} } }
26421 \cs_new_protected:Npn \regex_count:NnN #1
26422 { __regex_count:nnN { __regex_build:N #1 } }

```

(End definition for `\regex_count:nnN` and `\regex_count:NnN`. These functions are documented on page [234](#).)

**\regex\_extract\_once:nnN** We define here 40 user functions, following a common pattern in terms of `:nnN` auxiliaries, defined in the coming subsections. The auxiliary is handed `\__regex_build:n` or `\__regex_build:N` with the appropriate regex argument, then all other necessary arguments (replacement text, token list, *etc.*) The conditionals call `\__regex_return:` to return either `true` or `false` once matching has been performed.

```

26423 \cs_set_protected:Npn __regex_tmp:w #1#2#3
26424 {
26425 \cs_new_protected:Npn #2 ##1 { #1 { __regex_build:n {##1} } }
26426 \cs_new_protected:Npn #3 ##1 { #1 { __regex_build:N ##1 } }
26427 \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }

```

```

26428 { #1 { __regex_build:n {##1} } {##2} ##3 __regex_return: }
26429 \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
26430 { #1 { __regex_build:N ##1 } {##2} ##3 __regex_return: }
26431 }
26432 __regex_tmp:w __regex_extract_once:nnN
26433 \regex_extract_once:nnN \regex_extract_once:NnN
26434 __regex_tmp:w __regex_extract_all:nnN
26435 \regex_extract_all:nnN \regex_extract_all:NnN
26436 __regex_tmp:w __regex_replace_once:nnN
26437 \regex_replace_once:nnN \regex_replace_once:NnN
26438 __regex_tmp:w __regex_replace_all:nnN
26439 \regex_replace_all:nnN \regex_replace_all:NnN
26440 __regex_tmp:w __regex_split:nnN \regex_split:nnN \regex_split:NnN

(End definition for \regex_extract_once:nnNTF and others. These functions are documented on page
234.)
```

#### 41.7.1 Variables and helpers for user functions

\l\_\_regex\_match\_count\_int The number of matches found so far is stored in \l\_\_regex\_match\_count\_int. This is only used in the \regex\_count:nnN functions.

```
26441 \int_new:N \l__regex_match_count_int
```

(End definition for \l\_\_regex\_match\_count\_int.)

--regex\_begin --regex\_end Those flags are raised to indicate extra begin-group or end-group tokens when extracting submatches.

```
26442 \flag_new:n { __regex_begin }
26443 \flag_new:n { __regex_end }
```

(End definition for --regex\_begin and --regex\_end.)

\l\_\_regex\_min\_submatch\_int \l\_\_regex\_submatch\_int \l\_\_regex\_zeroth\_submatch\_int The end-points of each submatch are stored in two arrays whose index *<submatch>* ranges from \l\_\_regex\_min\_submatch\_int (inclusive) to \l\_\_regex\_submatch\_int (exclusive). Each successful match comes with a 0-th submatch (the full match), and one match for each capturing group: submatches corresponding to the last successful match are labelled starting at zeroth\_submatch. The entry \l\_\_regex\_zeroth\_submatch\_int in \g\_\_regex\_submatch\_prev\_intarray holds the position at which that match attempt started: this is used for splitting and replacements.

```
26444 \int_new:N \l__regex_min_submatch_int
26445 \int_new:N \l__regex_submatch_int
26446 \int_new:N \l__regex_zeroth_submatch_int
```

(End definition for \l\_\_regex\_min\_submatch\_int, \l\_\_regex\_submatch\_int, and \l\_\_regex\_zeroth\_submatch\_int.)

\g\_\_regex\_submatch\_prev\_intarray Hold the place where the match attempt begun and the end-points of each submatch.

```
26447 \intarray_new:Nn \g__regex_submatch_prev_intarray { 65536 }
26448 \intarray_new:Nn \g__regex_submatch_begin_intarray { 65536 }
26449 \intarray_new:Nn \g__regex_submatch_end_intarray { 65536 }
```

(End definition for \g\_\_regex\_submatch\_prev\_intarray, \g\_\_regex\_submatch\_begin\_intarray, and \g\_\_regex\_submatch\_end\_intarray.)

`\__regex_return:` This function triggers either `\prg_return_false:` or `\prg_return_true:` as appropriate to whether a match was found or not. It is used by all user conditionals.

```
26450 \cs_new_protected:Npn __regex_return:
26451 {
26452 \if_meaning:w \c_true_bool \g__regex_success_bool
26453 \prg_return_true:
26454 \else:
26455 \prg_return_false:
26456 \fi:
26457 }
```

(*End definition for `\__regex_return`:*)

#### 41.7.2 Matching

`\__regex_if_match:nn` We don't track submatches, and stop after a single match. Build the NFA with #1, and perform the match on the query #2.

```
26458 \cs_new_protected:Npn __regex_if_match:nn #1#2
26459 {
26460 \group_begin:
26461 __regex_disable_submatches:
26462 __regex_single_match:
26463 #1
26464 __regex_match:n {#2}
26465 \group_end:
26466 }
```

(*End definition for `\__regex_if_match:nn`.*)

`\__regex_count:nnN` Again, we don't care about submatches. Instead of aborting after the first "longest match" is found, we search for multiple matches, incrementing `\l__regex_match_count_int` every time to record the number of matches. Build the NFA and match. At the end, store the result in the user's variable.

```
26467 \cs_new_protected:Npn __regex_count:nnN #1#2#3
26468 {
26469 \group_begin:
26470 __regex_disable_submatches:
26471 \int_zero:N \l__regex_match_count_int
26472 __regex_multi_match:n { \int_incr:N \l__regex_match_count_int }
26473 #1
26474 __regex_match:n {#2}
26475 \exp_args:NNNo
26476 \group_end:
26477 \int_set:Nn #3 { \int_use:N \l__regex_match_count_int }
26478 }
```

(*End definition for `\__regex_count:nnN`.*)

#### 41.7.3 Extracting submatches

`\__regex_extract_once:nnN` Match once or multiple times. After each match (or after the only match), extract the submatches using `\__regex_extract:`. At the end, store the sequence containing all the submatches into the user variable #3 after closing the group.

```

26479 \cs_new_protected:Npn __regex_extract_once:nnN #1#2#3
26480 {
26481 \group_begin:
26482 __regex_single_match:
26483 #1
26484 __regex_match:n {#2}
26485 __regex_extract:
26486 __regex_group_end_extract_seq:N #3
26487 }
26488 \cs_new_protected:Npn __regex_extract_all:nnN #1#2#3
26489 {
26490 \group_begin:
26491 __regex_multi_match:n { __regex_extract: }
26492 #1
26493 __regex_match:n {#2}
26494 __regex_group_end_extract_seq:N #3
26495 }

```

(End definition for `\__regex_extract_once:nnN` and `\__regex_extract_all:nnN`.)

`\__regex_split:nnN` Splitting at submatches is a bit more tricky. For each match, extract all submatches, and replace the zeroth submatch by the part of the query between the start of the match attempt and the start of the zeroth submatch. This is inhibited if the delimiter matched an empty token list at the start of this match attempt. After the last match, store the last part of the token list, which ranges from the start of the match attempt to the end of the query. This step is inhibited if the last match was empty and at the very end: decrement `\l__regex_submatch_int`, which controls which matches will be used.

```

26496 \cs_new_protected:Npn __regex_split:nnN #1#2#3
26497 {
26498 \group_begin:
26499 __regex_multi_match:n
26500 {
26501 \if_int_compare:w
26502 \l__regex_start_pos_int < \l__regex_success_pos_int
26503 __regex_extract:
26504 __kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
26505 { \l__regex_zeroth_submatch_int } { 0 }
26506 __kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
26507 { \l__regex_zeroth_submatch_int }
26508 {
26509 __kernel_intarray_item:Nn \g__regex_submatch_begin_intarray
26510 { \l__regex_zeroth_submatch_int }
26511 }
26512 __kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
26513 { \l__regex_zeroth_submatch_int }
26514 { \l__regex_start_pos_int }
26515 \fi:
26516 }
26517 #1
26518 __regex_match:n {#2}
26519 (assert)\assert_int:n { \l__regex_curr_pos_int = \l__regex_max_pos_int }
26520 __kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
26521 { \l__regex_submatch_int } { 0 }
26522 __kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray

```

```

26523 { \l__regex_submatch_int }
26524 { \l__regex_max_pos_int }
26525 _kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
26526 { \l__regex_submatch_int }
26527 { \l__regex_start_pos_int }
26528 \int_incr:N \l__regex_submatch_int
26529 \if_meaning:w \c_true_bool \l__regex_empty_success_bool
26530 \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
26531 \int_decr:N \l__regex_submatch_int
26532 \fi:
26533 \fi:
26534 _regex_group_end_extract_seq:N #3
26535 }
```

(End definition for `\_regex_group_end_extract_seq:N`.)

`\_regex_group_end_extract_seq:N` The end-points of submatches are stored as entries of two arrays from `\l__regex_min_submatch_int` to `\l__regex_submatch_int` (exclusive). Extract the relevant ranges into `\l__regex_internal_a_t1`. We detect unbalanced results using the two flags `__regex_begin` and `__regex_end`, raised whenever we see too many begin-group or end-group tokens in a submatch.

```

26536 \cs_new_protected:Npn _regex_group_end_extract_seq:N #1
26537 {
26538 \flag_clear:n { __regex_begin }
26539 \flag_clear:n { __regex_end }
26540 \seq_set_from_function:NnN \l__regex_internal_seq
26541 {
26542 \int_step_function:nnN { \l__regex_min_submatch_int }
26543 { \l__regex_submatch_int - 1 }
26544 }
26545 _regex_extract_seq_aux:n
26546 \int_compare:nNnf
26547 {
26548 \flag_height:n { __regex_begin } +
26549 \flag_height:n { __regex_end }
26550 }
26551 = 0
26552 {
26553 _kernel_msg_error:nnxxx { kernel } { result-unbalanced }
26554 { splitting-or-extracting-submatches }
26555 { \flag_height:n { __regex_end } }
26556 { \flag_height:n { __regex_begin } }
26557 }
26558 \seq_set_map_x:NNn \l__regex_internal_seq \l__regex_internal_seq {##1}
26559 \exp_args:NNno
26560 \group_end:
26561 \tl_set:Nn #1 { \l__regex_internal_seq }
26562 }
```

(End definition for `\_regex_group_end_extract_seq:N`.)

`\_regex_extract_seq_aux:n` The `:n` auxiliary builds one item of the sequence of submatches. First compute the brace balance of the submatch, then extract the submatch from the query, adding the appropriate braces and raising a flag if the submatch is not balanced.

```

26563 \cs_new:Npn __regex_extract_seq_aux:n #1
26564 {
26565 \exp_after:wN __regex_extract_seq_aux:ww
26566 \int_value:w __regex_submatch_balance:n {#1} ; #1;
26567 }
26568 \cs_new:Npn __regex_extract_seq_aux:ww #1; #2;
26569 {
26570 \if_int_compare:w #1 < 0 \exp_stop_f:
26571 \flag_raise:n { __regex_end }
26572 \prg_replicate:nn {-#1} { \exp_not:n { { \if_false: } \fi: } }
26573 \fi:
26574 __regex_query_submatch:n {#2}
26575 \if_int_compare:w #1 > 0 \exp_stop_f:
26576 \flag_raise:n { __regex_begin }
26577 \prg_replicate:nn {#1} { \exp_not:n { \if_false: { \fi: } } }
26578 \fi:
26579 }

```

(End definition for `\__regex_extract_seq_aux:n` and `\__regex_extract_seq_aux:ww`.)

`\__regex_extract:`  
`\__regex_extract_b:wn`  
`\__regex_extract_e:wn`

Our task here is to extract from the property list `\l__regex_success_submatches_prop` the list of end-points of submatches, and store them in appropriate array entries, from `\l__regex_zeroth_submatch_int` upwards. We begin by emptying those entries. Then for each `<key>-<value>` pair in the property list update the appropriate entry. This is somewhat a hack: the `<key>` is a non-negative integer followed by `<` or `>`, which we use in a comparison to `-1`. At the end, store the information about the position at which the match attempt started, in `\g__regex_submatch_prev_intarray`.

```

26580 \cs_new_protected:Npn __regex_extract:
26581 {
26582 \if_meaning:w \c_true_bool \g__regex_success_bool
26583 \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
26584 \prg_replicate:nn \l__regex_capturing_group_int
26585 {
26586 __kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
26587 { \l__regex_submatch_int } { 0 }
26588 __kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
26589 { \l__regex_submatch_int } { 0 }
26590 __kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
26591 { \l__regex_submatch_int } { 0 }
26592 \int_incr:N \l__regex_submatch_int
26593 }
26594 \prop_map_inline:Nn \l__regex_success_submatches_prop
26595 {
26596 \if_int_compare:w ##1 - 1 \exp_stop_f:
26597 \exp_after:wN __regex_extract_e:wn \int_value:w
26598 \else:
26599 \exp_after:wN __regex_extract_b:wn \int_value:w
26600 \fi:
26601 __regex_int_eval:w \l__regex_zeroth_submatch_int + ##1 {##2}
26602 }
26603 __kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
26604 { \l__regex_zeroth_submatch_int } { \l__regex_start_pos_int }
26605 \fi:
26606 }

```

```

26607 \cs_new_protected:Npn __regex_extract_b:wn #1 < #2
26608 {
26609 __kernel_intarray_gset:Nnn
26610 \g__regex_submatch_begin_intarray {#1} {#2}
26611 }
26612 \cs_new_protected:Npn __regex_extract_e:wn #1 > #2
26613 { __kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray {#1} {#2} }

(End definition for __regex_extract:, __regex_extract_b:wn, and __regex_extract_e:wn.)

```

#### 41.7.4 Replacement

\\_\_regex\_replace\_once:nnN Build the NFA and the replacement functions, then find a single match. If the match failed, simply exit the group. Otherwise, we do the replacement. Extract submatches. Compute the brace balance corresponding to replacing this match by the replacement (this depends on submatches). Prepare the replaced token list: the replacement function produces the tokens from the start of the query to the start of the match and the replacement text for this match; we need to add the tokens from the end of the match to the end of the query. Finally, store the result in the user's variable after closing the group: this step involves an additional x-expansion, and checks that braces are balanced in the final result.

```

26614 \cs_new_protected:Npn __regex_replace_once:nnN #1#2#3
26615 {
26616 \group_begin:
26617 __regex_single_match:
26618 #1
26619 __regex_replacement:n {#2}
26620 \exp_args:No __regex_match:n { #3 }
26621 \if_meaning:w \c_false_bool \g__regex_success_bool
26622 \group_end:
26623 \else:
26624 __regex_extract:
26625 \int_set:Nn \l__regex_balance_int
26626 {
26627 __regex_replacement_balance_one_match:n
26628 { \l__regex_zeroth_submatch_int }
26629 }
26630 __kernel_tl_set:Nx \l__regex_internal_a_tl
26631 {
26632 __regex_replacement_do_one_match:n
26633 { \l__regex_zeroth_submatch_int }
26634 __regex_query_range:nn
26635 {
26636 __kernel_intarray_item:Nn \g__regex_submatch_end_intarray
26637 { \l__regex_zeroth_submatch_int }
26638 }
26639 { \l__regex_max_pos_int }
26640 }
26641 __regex_group_end_replace:N #3
26642 \fi:
26643 }

(End definition for __regex_replace_once:nnN.)
```

\\_\\_regex\\_replace\\_all:nnN Match multiple times, and for every match, extract submatches and additionally store the position at which the match attempt started. The entries from \l\\_\\_regex\\_min\\_submatch\\_int to \l\\_\\_regex\\_submatch\\_int hold information about submatches of every match in order; each match corresponds to \l\\_\\_regex\\_capturing\\_group\\_int consecutive entries. Compute the brace balance corresponding to doing all the replacements: this is the sum of brace balances for replacing each match. Join together the replacement texts for each match (including the part of the query before the match), and the end of the query.

```

26644 \cs_new_protected:Npn __regex_replace_all:nnN #1#2#3
26645 {
26646 \group_begin:
26647 __regex_multi_match:n { __regex_extract: } #1
26648 __regex_replacement:n {#2}
26649 \exp_args:No __regex_match:n {#3}
26650 \int_set:Nn \l__regex_balance_int
26651 {
26652 0
26653 \int_step_function:nnN
26654 { \l__regex_min_submatch_int }
26655 \l__regex_capturing_group_int
26656 { \l__regex_submatch_int - 1 }
26657 __regex_replacement_balance_one_match:n
26658 }
26659 }
26660 __kernel_tl_set:Nx \l__regex_internal_a_tl
26661 {
26662 \int_step_function:nnnN
26663 { \l__regex_min_submatch_int }
26664 \l__regex_capturing_group_int
26665 { \l__regex_submatch_int - 1 }
26666 __regex_replacement_do_one_match:n
26667 __regex_query_range:nn
26668 \l__regex_start_pos_int \l__regex_max_pos_int
26669 }
26670 __regex_group_end_replace:N #3
26671 }

```

(End definition for \\_\\_regex\_replace\_all:nnN.)

\\_\\_regex\_group\_end\_replace:N If the brace balance is not 0, raise an error. Then set the user's variable #1 to the x-expansion of \l\\_\\_regex\_internal\_a\_tl, adding the appropriate braces to produce a balanced result. And end the group.

```

26672 \cs_new_protected:Npn __regex_group_end_replace:N #1
26673 {
26674 \if_int_compare:w \l__regex_balance_int = 0 \exp_stop_f:
26675 \else:
26676 __kernel_msg_error:nxxxx { kernel } { result-unbalanced }
26677 { replacing }
26678 { \int_max:nn { - \l__regex_balance_int } { 0 } }
26679 { \int_max:nn { \l__regex_balance_int } { 0 } }
26680 \fi:
26681 \use:x
26682 {

```

```

26683 \group_end:
26684 \tl_set:Nn \exp_not:N #1
26685 {
26686 \if_int_compare:w \l__regex_balance_int < 0 \exp_stop_f:
26687 \prg_replicate:nn { - \l__regex_balance_int }
26688 { { \if_false: } \fi: }
26689 \fi:
26690 \l__regex_internal_a_tl
26691 \if_int_compare:w \l__regex_balance_int > 0 \exp_stop_f:
26692 \prg_replicate:nn { \l__regex_balance_int }
26693 { \if_false: { \fi: } }
26694 \fi:
26695 }
26696 }
26697 }
```

(End definition for `\_regex_group_end_replace:N`.)

#### 41.7.5 Storing and showing compiled patterns

### 41.8 Messages

Messages for the preparsing phase.

```

26698 \use:x
26699 {
26700 __kernel_msg_new:nnn { kernel } { trailing-backslash }
26701 { Trailing~escape~char~'\iow_char:N\\'~in~regex~or~replacement. }
26702 __kernel_msg_new:nnn { kernel } { x-missing-rbrace }
26703 {
26704 Missing-brace~'\iow_char:N\\'~in~regex~
26705 '... \iow_char:N\\x\iow_char:N\\{...##1'.
26706 }
26707 __kernel_msg_new:nnn { kernel } { x-overflow }
26708 {
26709 Character~code~##1~too~large~in~
26710 '\iow_char:N\\x\iow_char:N\\{##2\iow_char:N\\}~regex.
26711 }
26712 }
```

Invalid quantifier.

```

26713 __kernel_msg_new:nnnn { kernel } { invalid-quantifier }
26714 { Braced~quantifier~'#1'~may~not~be~followed~by~'#2'. }
26715 {
26716 The~character~'#2'~is~invalid~in~the~braced~quantifier~'#1'~.
26717 The~only~valid~quantifiers~are~'*',~'?',~'+',~'{<int>}',~
26718 '{<min>},~and~'{<min>,<max>}',~optionally~followed~by~'?'.
26719 }
```

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

```

26720 __kernel_msg_new:nnnn { kernel } { missing-rbrack }
26721 { Missing~right~bracket~inserted~in~regular~expression. }
26722 {
26723 LaTeX~was~given~a~regular~expression~where~a~character~class~
26724 was~started~with~'[',~but~the~matching~']'~is~missing.
```

```

26725 }
26726 __kernel_msg_new:nnnn { kernel } { missing-rparen }
26727 {
26728 Missing-right-
26729 \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
26730 inserted-in-regular-expression.
26731 }
26732 {
26733 LaTeX-was-given-a-regular-expression-with-\int_eval:n {#1} ~
26734 more-left-parentheses-than-right-parentheses.
26735 }
26736 __kernel_msg_new:nnnn { kernel } { extra-rparen }
26737 {
26738 Extra-right-parenthesis-ignored-in-regular-expression. }
26739 {
26740 LaTeX-came-across-a-closing-parenthesis-when-no-submatch-group-
26741 was-open.-The-parenthesis-will-be-ignored.
26741 }

```

Some escaped alphanumerics are not allowed everywhere.

```

26742 __kernel_msg_new:nnnn { kernel } { bad-escape }
26743 {
26744 Invalid-escape-'\\iow_char:N\\#1'-
26745 __regex_if_in_cs:TF { within-a-control-sequence. }
26746 {
26747 __regex_if_in_class:TF
26748 { in-a-character-class. }
26749 { following-a-category-test. }
26750 }
26751 }
26752 {
26753 The-escape-sequence-'\\iow_char:N\\#1'-may-not-appear-
26754 __regex_if_in_cs:TF
26755 {
26756 within-a-control-sequence-test-introduced-by-
26757 '\\iow_char:N\\c\\iow_char:N\\{'.
26758 }
26759 {
26760 __regex_if_in_class:TF
26761 { within-a-character-class- }
26762 { following-a-category-test-such-as-'\\iow_char:N\\cL'~ }
26763 because-it-does-not-match-exactly-one-character.
26764 }
26765 }

```

Range errors.

```

26766 __kernel_msg_new:nnnn { kernel } { range-missing-end }
26767 {
26768 Invalid-end-point-for-range-'#1-#2'-in-character-class. }
26769 {
26770 The-end-point-'#2'-of-the-range-'#1-#2'-may-not-serve-as-an-
26771 end-point-for-a-range:-alphanumeric-characters-should-not-be-
26772 escaped,-and-non-alphanumeric-characters-should-be-escaped.
26773 }
26774 __kernel_msg_new:nnnn { kernel } { range-backwards }
26775 {
26776 Range-[#1-#2]-out-of-order-in-character-class. }
26777 }

```

```

26776 In~ranges~of~characters~'[x-y]'~appearing~in~character~classes,~
26777 the~first~character~code~must~not~be~larger~than~the~second.~
26778 Here,~'#1'~has~character~code~\int_eval:n~{'#1},~while~
26779 '#2'~has~character~code~\int_eval:n~{'#2}.
26780 }
26781 Errors related to \c and \u.
26782 __kernel_msg_new:nnnn { kernel } { c-bad-mode }
26783 {
26784 The~'\iow_char:N\\c'~escape~cannot~be~used~within~
26785 a~control~sequence~test~'\iow_char:N\\c{...}'~
26786 nor~another~category~test.~
26787 To~combine~several~category~tests,~use~'\iow_char:N\\c[...]'.
26788 }
26789 __kernel_msg_new:nnnn { kernel } { c-C-invalid }
26790 {
26791 '\iow_char:N\\cC'~should~be~followed~by~'.'~or~'('~or~'#1'.
26792 The~'\iow_char:N\\cC'~construction~restricts~the~next~item~to~be~a~
26793 control~sequence~or~the~next~group~to~be~made~of~control~sequences.~
26794 It~only~makes~sense~to~follow~it~by~'.'~or~by~a~group.
26795 }
26796 __kernel_msg_new:nnnn { kernel } { c-lparen-in-class }
26797 {
26798 Catcode~test~cannot~apply~to~group~in~character~class
26799 Construction~such~as~'\iow_char:N\\cL(abc)'~are~not~allowed~inside~a~
26800 class~'['~because~classes~do~not~match~multiple~characters~at~once.
26801 }
26802 __kernel_msg_new:nnnn { kernel } { c-missing-rbrace }
26803 {
26804 Missing~right~brace~inserted~for~'\iow_char:N\\c'~escape.
26805 LaTeX~was~given~a~regular~expression~where~a~
26806 '\iow_char:N\\c\iow_char:N\\{...}'~construction~was~not~ended~
26807 with~a~closing~brace~'\iow_char:N}'.
26808 }
26809 __kernel_msg_new:nnnn { kernel } { c-missing-rbrack }
26810 {
26811 Missing~right~bracket~inserted~for~'\iow_char:N\\c'~escape.
26812 A~construction~'\iow_char:N\\c[...]'~appears~in~a~
26813 regular~expression,~but~the~closing~']'~is~not~present.
26814 }
26815 __kernel_msg_new:nnnn { kernel } { c-missing-category }
26816 {
26817 Invalid~character~'#1'~following~'\iow_char:N\\c'~escape.
26818 In~regular~expressions,~the~'\iow_char:N\\c'~escape~sequence~
26819 may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
26820 capital~letter~representing~a~character~category,~namely~
26821 one~of~'ABCDELMOPSTU'.
26822 }
26823 __kernel_msg_new:nnnn { kernel } { c-trailing }
26824 {
26825 Trailing~category~code~escape~'\iow_char:N\\c'...
26826 A~regular~expression~ends~with~'\iow_char:N\\c'~followed~
26827 by~a~letter.~It~will~be~ignored.
26828 }

```

```

26829 __kernel_msg_new:nnnn { kernel } { u-missing-lbrace }
26830 { Missing~left~brace~following~'\iow_char:N\\u'~escape. }
26831 {
26832 The~'\iow_char:N\\u'~escape~sequence~must~be~followed~by~
26833 a-brace~group~with~the~name~of~the~variable~to~use.
26834 }
26835 __kernel_msg_new:nnnn { kernel } { u-missing-rbrace }
26836 { Missing~right~brace~inserted~for~'\iow_char:N\\u'~escape. }
26837 {
26838 LaTeX~
26839 \str_if_eq:eeTF { } {#2}
26840 { reached-the-end-of-the-string~ }
26841 { encountered-an-escaped-alphanumeric-character '\iow_char:N\\#2'~ }
26842 when~parsing~the~argument~of~an~
26843 '\iow_char:N\\u\iow_char:N\\{...\\}'~escape.
26844 }

```

Errors when encountering the POSIX syntax [:....:].

```

26845 __kernel_msg_new:nnnn { kernel } { posix-unsupported }
26846 { POSIX~collating~element~'[#1 ~ #1]'~not~supported. }
26847 {
26848 The~'[.foo.]'~and~'[=bar=]'~syntaxes~have~a~special~meaning~
26849 in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
26850 Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
26851 }
26852 __kernel_msg_new:nnnn { kernel } { posix-unknown }
26853 { POSIX~class~'[:#1:]'~unknown. }
26854 {
26855 '[:#1:]'~is~not~among~the~known~POSIX~classes~
26856 '[:alnum:]',~'[:alpha:]',~'[:ascii:]',~'[:blank:]',~
26857 '[:cntrl:]',~'[:digit:]',~'[:graph:]',~'[:lower:]',~
26858 '[:print:]',~'[:punct:]',~'[:space:]',~'[:upper:]',~
26859 '[:word:]',~and~'[:xdigit:]'.
26860 }
26861 __kernel_msg_new:nnnn { kernel } { posix-missing-close }
26862 { Missing~closing~':]'~for~POSIX~class. }
26863 { The~POSIX~syntax~'#1'~must~be~followed~by~':]',~not~'#2'. }

```

In various cases, the result of a l3regex operation can leave us with an unbalanced token list, which we must re-balance by adding begin-group or end-group character tokens.

```

26864 __kernel_msg_new:nnnn { kernel } { result-unbalanced }
26865 { Missing~brace~inserted~when~#1. }
26866 {
26867 LaTeX~was~asked~to~do~some~regular~expression~operation,~
26868 and~the~resulting~token~list~would~not~have~the~same~number~
26869 of~begin~group~and~end~group~tokens.~Braces~were~inserted:~
26870 #2~left,~#3~right.
26871 }

```

Error message for unknown options.

```

26872 __kernel_msg_new:nnnn { kernel } { unknown-option }
26873 { Unknown~option~'#1'~for~regular~expressions. }
26874 {
26875 The~only~available~option~is~'case-insensitive',~toggled~by~

```

```

26876 '(?i)'~and~'(?-i)'.
26877 }
26878 __kernel_msg_new:nnnn { kernel } { special-group-unknown }
26879 { Unknown~special~group~'#1'...~in~a~regular~expression. }
26880 {
26881 The~only~valid~constructions~starting~with~'(~'~are~
26882 '(:~...~)',~'(?|~...~)',~'(?i)',~and~'(?-i)'.
26883 }

Errors in the replacement text.

26884 __kernel_msg_new:nnnn { kernel } { replacement-c }
26885 { Misused~'\iow_char:N\\c'~command~in~a~replacement~text. }
26886 {
26887 In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
26888 can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~
26889 or~a~brace~group,~not~by~'#1'.
26890 }
26891 __kernel_msg_new:nnnn { kernel } { replacement-u }
26892 { Misused~'\iow_char:N\\u'~command~in~a~replacement~text. }
26893 {
26894 In~a~replacement~text,~the~'\iow_char:N\\u'~escape~sequence~
26895 must~be~followed~by~a~brace~group~holding~the~name~of~the~
26896 variable~to~use.
26897 }
26898 __kernel_msg_new:nnnn { kernel } { replacement-g }
26899 {
26900 Missing~brace~for~the~'\iow_char:N\\g'~construction~
26901 in~a~replacement~text.
26902 }
26903 {
26904 In~the~replacement~text~for~a~regular~expression~search,~
26905 submatches~are~represented~either~as~'\iow_char:N \\g{dd..d}',~
26906 or~'\\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
26907 }
26908 __kernel_msg_new:nnnn { kernel } { replacement-catcode-end }
26909 {
26910 Missing~character~for~the~'\iow_char:N\\c<category><character>'~
26911 construction~in~a~replacement~text.
26912 }
26913 {
26914 In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
26915 can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
26916 the~character~category.~Then,~a~character~must~follow.~LaTeX~
26917 reached~the~end~of~the~replacement~when~looking~for~that.
26918 }
26919 __kernel_msg_new:nnnn { kernel } { replacement-catcode-escaped }
26920 {
26921 Escaped~letter~or~digit~after~category~code~in~replacement~text.
26922 }
26923 {
26924 In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
26925 can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
26926 the~character~category.~Then,~a~character~must~follow,~not~
26927 '\iow_char:N\\#2'.
26928 }

```

```

26929 _kernel_msg_new:nnnn { kernel } { replacement-catcode-in-cs }
26930 {
26931 Category~code~'\iow_char:N\\c#1#3'~ignored~inside~
26932 '\iow_char:N\\c\{...\}'~in~a~replacement~text.
26933 }
26934 {
26935 In~a~replacement~text,~the~category~codes~of~the~argument~of~
26936 '\iow_char:N\\c\{...\}'~are~ignored~when~building~the~control-
26937 sequence~name.
26938 }
26939 _kernel_msg_new:nnnn { kernel } { replacement-null-space }
26940 { TeX~cannot~build~a~space~token~with~character~code~0. }
26941 {
26942 You~asked~for~a~character~token~with~category~space,~
26943 and~character~code~0,~for~instance~through~
26944 '\iow_char:N\\cS\iow_char:N\\x00'.~
26945 This~specific~case~is~impossible~and~will~be~replaced~
26946 by~a~normal~space.
26947 }
26948 _kernel_msg_new:nnnn { kernel } { replacement-missing-rbrace }
26949 { Missing~right~brace~inserted~in~replacement~text. }
26950 {
26951 There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
26952 missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
26953 }
26954 _kernel_msg_new:nnnn { kernel } { replacement-missing-rparen }
26955 { Missing~right~parenthesis~inserted~in~replacement~text. }
26956 {
26957 There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
26958 missing~right~
26959 \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
26960 }

```

Used when showing a regex.

```

26961 _kernel_msg_new:nnn { kernel } { show-regex }
26962 {
26963 >~Compiled~regex~
26964 \tl_if_empty:nTF {#1} { variable~ #2 } { {#1} } :
26965 #3
26966 }

```

\\_regex\_msg\_repeated:nnN This is not technically a message, but seems related enough to go there. The arguments are: #1 is the minimum number of repetitions; #2 is the number of allowed extra repetitions (-1 for infinite number), and #3 tells us about laziness.

```

26967 \cs_new:Npn _regex_msg_repeated:nnN #1#2#3
26968 {
26969 \str_if_eq:eeF { #1 #2 } { 1 0 }
26970 {
26971 , ~ repeated ~
26972 \int_case:nnF {#2}
26973 {
26974 { -1 } { #1-or-more-times,~\bool_if:NTF #3 { lazy } { greedy } }
26975 { 0 } { #1-times }
26976 }

```

```

26977 {
26978 between~#1~and~\int_eval:n {#1+#2}~times,~
26979 \bool_if:NTF #3 { lazy } { greedy }
26980 }
26981 }
26982 }
```

(End definition for `\_regex_msg_repeated:nnN`.)

## 41.9 Code for tracing

There is a more extensive implementation of tracing in the l3trial package `l3trace`. Function names are a bit different but could be merged.

`\_regex_trace_push:nnN` Here #1 is the module name (`regex`) and #2 is typically 1. If the module's current tracing level is less than #2 show nothing, otherwise write #3 to the terminal.

```

26983 \cs_new_protected:Npn _regex_trace_push:nnN #1#2#3
26984 { _regex_trace:nnx {#1} {#2} { entering~ \token_to_str:N #3 } }
26985 \cs_new_protected:Npn _regex_trace_pop:nnN #1#2#3
26986 { _regex_trace:nnx {#1} {#2} { leaving~ \token_to_str:N #3 } }
26987 \cs_new_protected:Npn _regex_trace:nnx #1#2#3
26988 {
26989 \int_compare:nNnF
26990 { \int_use:c { g__regex_trace_#1_int } } < {#2}
26991 { \iow_term:x { Trace:~#3 } }
26992 }
```

(End definition for `\_regex_trace_push:nnN`, `\_regex_trace_pop:nnN`, and `\_regex_trace:nnx`.)

`\g__regex_trace_regex_int` No tracing when that is zero.

```
26993 \int_new:N \g__regex_trace_regex_int
```

(End definition for `\g__regex_trace_regex_int`.)

`\_regex_trace_states:n` This function lists the contents of all states of the NFA, stored in `\toks` from 0 to `\l__regex_max_state_int` (excluded).

```

26994 \cs_new_protected:Npn _regex_trace_states:n #1
26995 {
26996 \int_step_inline:nnn
26997 \l__regex_min_state_int
26998 { \l__regex_max_state_int - 1 }
26999 {
27000 _regex_trace:nnx { regex } {#1}
27001 { \iow_char:N \toks ##1 = { _regex_toks_use:w ##1 } }
27002 }
27003 }
```

(End definition for `\_regex_trace_states:n`.)

```
27004
```

## 42 l3box implementation

```
27005 {*package}
27006 <@=box>
```

### 42.1 Support code

`\_box_dim_eval:w` Evaluating a dimension expression expandably. The only difference with `\dim_eval:n` is the lack of `\dim_use:N`, to produce an internal dimension rather than expand it into characters.

```
27007 \cs_new_eq:NN _box_dim_eval:w \tex_dimexpr:D
27008 \cs_new:Npn _box_dim_eval:n #1
27009 { _box_dim_eval:w #1 \scan_stop: }
```

(End definition for `\_box_dim_eval:w` and `\_box_dim_eval:n`.)

### 42.2 Creating and initialising boxes

The following test files are used for this code: `m3box001.lvt`.

`\box_new:N` Defining a new `<box>` register: remember that box 255 is not generally available.

```
27010 \cs_new_protected:Npn \box_new:N #1
27011 {
27012 _kernel_chk_if_free:cs:N #1
27013 \cs:w newbox \cs_end: #1
27014 }
27015 \cs_generate_variant:Nn \box_new:N { c }
```

Clear a `<box>` register.

```
27016 \cs_new_protected:Npn \box_clear:N #1
27017 { \box_set_eq:NN #1 \c_empty_box }
27018 \cs_new_protected:Npn \box_gclear:N #1
27019 { \box_gset_eq:NN #1 \c_empty_box }
27020 \cs_generate_variant:Nn \box_clear:N { c }
27021 \cs_generate_variant:Nn \box_gclear:N { c }
```

Clear or new.

```
27022 \cs_new_protected:Npn \box_clear_new:N #1
27023 { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
27024 \cs_new_protected:Npn \box_gclear_new:N #1
27025 { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
27026 \cs_generate_variant:Nn \box_clear_new:N { c }
27027 \cs_generate_variant:Nn \box_gclear_new:N { c }
```

Assigning the contents of a box to be another box.

```
27028 \cs_new_protected:Npn \box_set_eq:NN #1#2
27029 { \tex_setbox:D #1 \tex_copy:D #2 }
27030 \cs_new_protected:Npn \box_gset_eq:NN #1#2
27031 { \tex_global:D \tex_setbox:D #1 \tex_copy:D #2 }
27032 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
27033 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }

\box_gset_eq:cN
\box_gset_eq:Nc
\box_gset_eq:cc
```

Assigning the contents of a box to be another box, then drops the original box.

```
27034 \cs_new_protected:Npn \box_set_eq_drop:NN #1#2
27035 { \tex_setbox:D #1 \tex_box:D #2 }
27036 \cs_new_protected:Npn \box_gset_eq_drop:NN #1#2
27037 { \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
27038 \cs_generate_variant:Nn \box_set_eq_drop:NN { c , Nc , cc }
27039 \cs_generate_variant:Nn \box_gset_eq_drop:NN { c , Nc , cc }

\box_gset_eq_drop:NN
\box_gset_eq_drop:cN
\box_gset_eq_drop:Nc
\box_gset_eq_drop:cc
\box_if_exist_p:N
\box_if_exist_p:c
\box_if_exist:NTF
\box_if_exist:cTF
```

Copies of the `cs` functions defined in `l3basics`.

```
27040 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
27041 { TF , T , F , p }
27042 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
27043 { TF , T , F , p }
```

## 42.3 Measuring and setting box dimensions

Accessing the height, depth, and width of a `<box>` register.

```
27044 \cs_new_eq:NN \box_ht:N \tex_ht:D
27045 \cs_new_eq:NN \box_dp:N \tex_dp:D
27046 \cs_new_eq:NN \box_wd:N \tex_wd:D
27047 \cs_generate_variant:Nn \box_ht:N { c }
27048 \cs_generate_variant:Nn \box_dp:N { c }
27049 \cs_generate_variant:Nn \box_wd:N { c }

\box_wd:c

\box_set_ht:Nn
\box_set_ht:cn
\box_gset_ht:Nn
\box_gset_ht:cn
\box_set_dp:Nn
\box_set_dp:cn
\box_gset_dp:Nn
\box_gset_dp:cn
\box_set_wd:Nn
\box_set_wd:cn
\box_gset_wd:Nn
\box_gset_wd:cn

27050 \cs_new_protected:Npn \box_set_dp:Nn #1#2
27051 {
27052 \tex_setbox:D #1 = \tex_copy:D #1
27053 \box_dp:N #1 __box_dim_eval:n {#2}
27054 }
27055 \cs_generate_variant:Nn \box_set_dp:Nn { c }
27056 \cs_new_protected:Npn \box_gset_dp:Nn #1#2
27057 { \box_dp:N #1 __box_dim_eval:n {#2} }
27058 \cs_generate_variant:Nn \box_gset_dp:Nn { c }
27059 \cs_new_protected:Npn \box_set_ht:Nn #1#2
27060 {
27061 \tex_setbox:D #1 = \tex_copy:D #1
27062 \box_ht:N #1 __box_dim_eval:n {#2}
27063 }
27064 \cs_generate_variant:Nn \box_set_ht:Nn { c }
27065 \cs_new_protected:Npn \box_gset_ht:Nn #1#2
27066 { \box_ht:N #1 __box_dim_eval:n {#2} }
27067 \cs_generate_variant:Nn \box_gset_ht:Nn { c }
27068 \cs_new_protected:Npn \box_set_wd:Nn #1#2
27069 {
27070 \tex_setbox:D #1 = \tex_copy:D #1
27071 \box_wd:N #1 __box_dim_eval:n {#2}
27072 }
27073 \cs_generate_variant:Nn \box_set_wd:Nn { c }
27074 \cs_new_protected:Npn \box_gset_wd:Nn #1#2
27075 { \box_wd:N #1 __box_dim_eval:n {#2} }
27076 \cs_generate_variant:Nn \box_gset_wd:Nn { c }
```

## 42.4 Using boxes

Using a  $\langle box \rangle$ . These are just T<sub>E</sub>X primitives with meaningful names.

```

27077 \cs_new_eq:NN \box_use_drop:N \tex_box:D
27078 \cs_new_eq:NN \box_use:N \tex_copy:D
27079 \cs_generate_variant:Nn \box_use_drop:N { c }
27080 \cs_generate_variant:Nn \box_use:N { c }

\box_use:c

```

Move box material in different directions. When debugging, the dimension expression #1 is surrounded by parentheses to catch early termination.

```

\box_move_left:nn
\box_move_right:nn
\box_move_up:nn
\box_move_down:nn

27081 \cs_new_protected:Npn \box_move_left:nn #1#2
27082 { \tex_moveleft:D __box_dim_eval:n {#1} #2 }
27083 \cs_new_protected:Npn \box_move_right:nn #1#2
27084 { \tex_moveright:D __box_dim_eval:n {#1} #2 }
27085 \cs_new_protected:Npn \box_move_up:nn #1#2
27086 { \tex_raise:D __box_dim_eval:n {#1} #2 }
27087 \cs_new_protected:Npn \box_move_down:nn #1#2
27088 { \tex_lower:D __box_dim_eval:n {#1} #2 }

```

## 42.5 Box conditionals

The primitives for testing if a  $\langle box \rangle$  is empty/void or which type of box it is.

```

\if_hbox:N \tex_ifhbox:D
\if_vbox:N \tex_ifvbox:D
\if_box_empty:N \tex_ifvoid:D

\if_box_horizontal_p:N
\if_box_horizontal_p:c
\if_box_horizontal:NTF
\if_box_horizontal:cTF
\if_box_vertical_p:N
\if_box_vertical_p:c
\if_box_vertical:NTF
\if_box_vertical:cTF

```

```

27092 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
27093 { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
27094 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
27095 { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
27096 \prg_generate_conditional_variant:Nnn \box_if_horizontal:N
27097 { c } { p , T , F , TF }
27098 \prg_generate_conditional_variant:Nnn \box_if_vertical:N
27099 { c } { p , T , F , TF }

```

Testing if a  $\langle box \rangle$  is empty/void.

```

\if_box_empty_p:N
\if_box_empty_p:c
\if_box_empty:NTF
\if_box_empty:cTF

```

(End definition for `\box_new:N` and others. These functions are documented on page 239.)

## 42.6 The last box inserted

```

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

27104 \cs_new_protected:Npn \box_set_to_last:N #1
27105 { \tex_setbox:D #1 \tex_lastbox:D }
27106 \cs_new_protected:Npn \box_gset_to_last:N #1
27107 { \tex_global:D \tex_setbox:D #1 \tex_lastbox:D }
27108 \cs_generate_variant:Nn \box_set_to_last:N { c }
27109 \cs_generate_variant:Nn \box_gset_to_last:N { c }

```

(End definition for `\box_set_to:last:N` and `\box_gset_to:last:N`. These functions are documented on page 241.)

## 42.7 Constant boxes

`\c_empty_box` A box we never use.

```
27110 \box_new:N \c_empty_box
```

(End definition for `\c_empty_box`. This variable is documented on page 241.)

## 42.8 Scratch boxes

`\l_tmpa_box` Scratch boxes.

`\l_tmpb_box`

```
27111 \box_new:N \l_tmpa_box
```

`\g_tmpa_box`

```
27112 \box_new:N \l_tmpb_box
```

`\g_tmpb_box`

```
27113 \box_new:N \g_tmpa_box
```

```
27114 \box_new:N \g_tmpb_box
```

(End definition for `\l_tmpa_box` and others. These variables are documented on page 242.)

## 42.9 Viewing box contents

T<sub>E</sub>X's `\showbox` is not really that helpful in many cases, and it is also inconsistent with other L<sub>A</sub>T<sub>E</sub>X3 `show` functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

`\box_show:N` Essentially a wrapper around the internal function, but evaluating the breadth and depth arguments now outside the group.

`\box_show:c`

```
27115 \cs_new_protected:Npn \box_show:N #1
```

```
27116 { \box_show:Nnn #1 \c_max_int \c_max_int }
```

```
27117 \cs_generate_variant:Nn \box_show:N { c }
```

```
27118 \cs_new_protected:Npn \box_show:Nnn #1#2#3
```

```
27119 { __box_show>NNff 1 #1 { \int_eval:n {#2} } { \int_eval:n {#3} } }
```

```
27120 \cs_generate_variant:Nn \box_show:Nnn { c }
```

(End definition for `\box_show:N` and `\box_show:Nnn`. These functions are documented on page 242.)

`\box_log:N` Getting T<sub>E</sub>X to write to the log without interruption the run is done by altering the interaction mode. For that, the ε-T<sub>E</sub>X extensions are needed.

`\box_log:c`

```
27121 \cs_new_protected:Npn \box_log:N #1
```

```
27122 { \box_log:Nnn #1 \c_max_int \c_max_int }
```

`\box_log:cnn`

```
27123 \cs_generate_variant:Nn \box_log:N { c }
```

```
27124 \cs_new_protected:Npn \box_log:Nnn
```

```
27125 { \exp_args:No __box_log:nNnn { \tex_the:D \tex_interactionmode:D } }
```

```
27126 \cs_new_protected:Npn __box_log:nNnn #1#2#3#4
```

```
27127 {
```

```
27128 \int_set:Nn \tex_interactionmode:D { 0 }
```

```
27129 __box_show>NNff 0 #2 { \int_eval:n {#3} } { \int_eval:n {#4} }
```

```
27130 \int_set:Nn \tex_interactionmode:D {#1}
```

```
27131 }
```

```
27132 \cs_generate_variant:Nn \box_log:Nnn { c }
```

(End definition for `\box_log:N`, `\box_log:Nnn`, and `\__box_log:nNnn`. These functions are documented on page 242.)

`\__box_show>NNnn`  
`\__box_show>NNff`

The internal auxiliary to actually do the output uses a group to deal with breadth and depth values. The `\use:n` here gives better output appearance. Setting `\tracingonline` and `\errorcontextlines` is used to control what appears in the terminal.

```

27133 \cs_new_protected:Npn __box_show>NNnn #1#2#3#4
27134 {
27135 \box_if_exist:NTF #2
27136 {
27137 \group_begin:
27138 \int_set:Nn \tex_showboxbreadth:D {#3}
27139 \int_set:Nn \tex_showboxdepth:D {#4}
27140 \int_set:Nn \tex_tracingonline:D {#1}
27141 \int_set:Nn \tex_errorcontextlines:D { -1 }
27142 \tex_showbox:D \use:n {#2}
27143 \group_end:
27144 }
27145 {
27146 __kernel_msg_error:nnx { kernel } { variable-not-defined }
27147 { \token_to_str:N #2 }
27148 }
27149 }
27150 \cs_generate_variant:Nn __box_show>NNnn { NNff }
```

(End definition for `\__box_show>NNnn`.)

## 42.10 Horizontal mode boxes

`\hbox:n` (*The test suite for this command, and others in this file, is m3box002.lvt.*)  
 Put a horizontal box directly into the input stream.

```

27151 \cs_new_protected:Npn \hbox:n #1
27152 { \tex_hbox:D \scan_stop: { \color_group_begin: #1 \color_group_end: } }
```

(End definition for `\hbox:n`. This function is documented on page 242.)

```

\hbox_set:Nn
\hbox_set:cN
\hbox_gset:Nn
\hbox_gset:cN
27153 \cs_new_protected:Npn \hbox_set:Nn #1#2
27154 {
27155 \tex_setbox:D #1 \tex_hbox:D
27156 { \color_group_begin: #2 \color_group_end: }
27157 }
27158 \cs_new_protected:Npn \hbox_gset:Nn #1#2
27159 {
27160 \tex_global:D \tex_setbox:D #1 \tex_hbox:D
27161 { \color_group_begin: #2 \color_group_end: }
27162 }
27163 \cs_generate_variant:Nn \hbox_set:Nn { c }
27164 \cs_generate_variant:Nn \hbox_gset:Nn { c }
```

(End definition for `\hbox_set:Nn` and `\hbox_gset:Nn`. These functions are documented on page 243.)

`\hbox_set_to_wd:Nnn`  
`\hbox_set_to_wd:cnn`

Storing material in a horizontal box with a specified width. Again, put the dimension expression in parentheses when debugging.

```

\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn
27165 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
27166 {
27167 \tex_setbox:D #1 \tex_hbox:D to __box_dim_eval:n {#2}
```

```

27168 { \color_group_begin: #3 \color_group_end: }
27169 }
27170 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn #1#2#3
27171 {
27172 \tex_global:D \tex_setbox:D #1 \tex_hbox:D to __box_dim_eval:n {#2}
27173 { \color_group_begin: #3 \color_group_end: }
27174 }
27175 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
27176 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }

```

(End definition for `\hbox_set_to_wd:Nnn` and `\hbox_gset_to_wd:Nnn`. These functions are documented on page 243.)

**\hbox\_set:Nw** Storing material in a horizontal box. This type is useful in environment definitions.

```

\hbox_set:cw
27177 \cs_new_protected:Npn \hbox_set:Nw #1
\hbox_gset:Nw
27178 {
\hbox_gset:cw
27179 \tex_setbox:D #1 \tex_hbox:D
\hbox_set_end:
27180 \c_group_begin_token
\hbox_gset_end:
27181 \color_group_begin:
27182 }
27183 \cs_new_protected:Npn \hbox_gset:Nw #1
27184 {
27185 \tex_global:D \tex_setbox:D #1 \tex_hbox:D
27186 \c_group_begin_token
27187 \color_group_begin:
27188 }
27189 \cs_generate_variant:Nn \hbox_set:Nw { c }
27190 \cs_generate_variant:Nn \hbox_gset:Nw { c }
27191 \cs_new_protected:Npn \hbox_set_end:
27192 {
27193 \color_group_end:
27194 \c_group_end_token
27195 }
27196 \cs_new_eq:NN \hbox_gset_end: \hbox_set_end:

```

(End definition for `\hbox_set:Nw` and others. These functions are documented on page 243.)

**\hbox\_set\_to\_wd:Nnw** Combining the above ideas.

```

\hbox_set_to_wd:cw
27197 \cs_new_protected:Npn \hbox_set_to_wd:Nnw #1#2
27198 {
27199 \tex_setbox:D #1 \tex_hbox:D to __box_dim_eval:n {#2}
27200 \c_group_begin_token
27201 \color_group_begin:
27202 }
27203 \cs_new_protected:Npn \hbox_gset_to_wd:Nnw #1#2
27204 {
27205 \tex_global:D \tex_setbox:D #1 \tex_hbox:D to __box_dim_eval:n {#2}
27206 \c_group_begin_token
27207 \color_group_begin:
27208 }
27209 \cs_generate_variant:Nn \hbox_set_to_wd:Nnw { c }
27210 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnw { c }

```

(End definition for `\hbox_set_to_wd:Nnw` and `\hbox_gset_to_wd:Nnw`. These functions are documented on page 243.)

**\hbox\_to\_wd:nn** Put a horizontal box directly into the input stream.

```

27211 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
27212 {
27213 \tex_hbox:D to __box_dim_eval:n {#1}
27214 { \color_group_begin: #2 \color_group_end: }
27215 }
27216 \cs_new_protected:Npn \hbox_to_zero:n #1
27217 {
27218 \tex_hbox:D to \c_zero_dim
27219 { \color_group_begin: #1 \color_group_end: }
27220 }
```

(End definition for `\hbox_to_wd:nn` and `\hbox_to_zero:n`. These functions are documented on page 243.)

**\hbox\_overlap\_center:n** Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

**\hbox\_overlap\_left:n**

```

27221 \cs_new_protected:Npn \hbox_overlap_center:n #1
27222 { \hbox_to_zero:n { \tex_hss:D #1 \tex_hss:D } }
27223 \cs_new_protected:Npn \hbox_overlap_left:n #1
27224 { \hbox_to_zero:n { \tex_hss:D #1 } }
27225 \cs_new_protected:Npn \hbox_overlap_right:n #1
27226 { \hbox_to_zero:n { #1 \tex_hss:D } }
```

(End definition for `\hbox_overlap_center:n`, `\hbox_overlap_left:n`, and `\hbox_overlap_right:n`. These functions are documented on page 243.)

**\hbox\_unpack:N** Unpacking a box and if requested also clear it.

**\hbox\_unpack:c**

```

27227 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
27228 \cs_new_eq:NN \hbox_unpack_drop:N \tex_unhbox:D
27229 \cs_generate_variant:Nn \hbox_unpack:N { c }
27230 \cs_generate_variant:Nn \hbox_unpack_drop:N { c }
```

(End definition for `\hbox_unpack:N` and `\hbox_unpack_drop:N`. These functions are documented on page 244.)

## 42.11 Vertical mode boxes

TeX ends these boxes directly with the internal `end_graf` routine. This means that there is no `\par` at the end of vertical boxes unless we insert one. Thus all vertical boxes include a `\par` just before closing the color group.

**\vbox:n** The following test files are used for this code: `m3box003.lvt`.

The following test files are used for this code: `m3box003.lvt`.

**\vbox\_top:n** Put a vertical box directly into the input stream.

```

27231 \cs_new_protected:Npn \vbox:n #1
27232 { \tex_vbox:D { \color_group_begin: #1 \par \color_group_end: } }
27233 \cs_new_protected:Npn \vbox_top:n #1
27234 { \tex_vtop:D { \color_group_begin: #1 \par \color_group_end: } }
```

(End definition for `\vbox:n` and `\vbox_top:n`. These functions are documented on page 244.)

```

\ vbox_to_ht:nn Put a vertical box directly into the input stream.
\ vbox_to_zero:n
\ vbox_to_ht:nn
\ vbox_to_zero:n
27235 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
27236 {
27237 \tex_vbox:D to __box_dim_eval:n {#1}
27238 { \color_group_begin: #2 \par \color_group_end: }
27239 }
27240 \cs_new_protected:Npn \vbox_to_zero:n #1
27241 {
27242 \tex_vbox:D to \c_zero_dim
27243 { \color_group_begin: #1 \par \color_group_end: }
27244 }

```

(End definition for `\vbox_to_ht:nn` and others. These functions are documented on page 244.)

```

\ vbox_set:Nn Storing material in a vertical box with a natural height.
\ vbox_set:cn
\ vbox_gset:Nn
\ vbox_gset:cn
27245 \cs_new_protected:Npn \vbox_set:Nn #1#2
27246 {
27247 \tex_setbox:D #1 \tex_vbox:D
27248 { \color_group_begin: #2 \par \color_group_end: }
27249 }
27250 \cs_new_protected:Npn \vbox_gset:Nn #1#2
27251 {
27252 \tex_global:D \tex_setbox:D #1 \tex_vbox:D
27253 { \color_group_begin: #2 \par \color_group_end: }
27254 }
27255 \cs_generate_variant:Nn \vbox_set:Nn { c }
27256 \cs_generate_variant:Nn \vbox_gset:Nn { c }

```

(End definition for `\vbox_set:Nn` and `\vbox_gset:Nn`. These functions are documented on page 244.)

```

\ vbox_set_top:Nn Storing material in a vertical box with a natural height and reference point at the baseline
\ vbox_set_top:cn of the first object in the box.
\ vbox_gset_top:Nn
\ vbox_gset_top:cn
27257 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
27258 {
27259 \tex_setbox:D #1 \tex_vtop:D
27260 { \color_group_begin: #2 \par \color_group_end: }
27261 }
27262 \cs_new_protected:Npn \vbox_gset_top:Nn #1#2
27263 {
27264 \tex_global:D \tex_setbox:D #1 \tex_vtop:D
27265 { \color_group_begin: #2 \par \color_group_end: }
27266 }
27267 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
27268 \cs_generate_variant:Nn \vbox_gset_top:Nn { c }

```

(End definition for `\vbox_set_top:Nn` and `\vbox_gset_top:Nn`. These functions are documented on page 244.)

```

\ vbox_set_to_ht:Nnn Storing material in a vertical box with a specified height.
\ vbox_set_to_ht:cnn
\ vbox_gset_to_ht:Nnn
\ vbox_gset_to_ht:cnn
27269 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
27270 {
27271 \tex_setbox:D #1 \tex_vbox:D to __box_dim_eval:n {#2}
27272 { \color_group_begin: #3 \par \color_group_end: }
27273 }

```

```

27274 \cs_new_protected:Npn \vbox_gset_to_ht:Nnn #1#2#3
27275 {
27276 \tex_global:D \tex_setbox:D #1 \tex_vbox:D to __box_dim_eval:n {#2}
27277 { \color_group_begin: #3 \par \color_group_end: }
27278 }
27279 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }
27280 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }

```

(End definition for `\vbox_set_to_ht:Nnn` and `\vbox_gset_to_ht:Nnn`. These functions are documented on page 245.)

**\vbox\_set:Nw** Storing material in a vertical box. This type is useful in environment definitions.

```

\vbox_set:cw
27281 \cs_new_protected:Npn \vbox_set:Nw #1
\vbox_gset:Nw
27282 {
27283 \tex_setbox:D #1 \tex_vbox:D
\vbox_set_end:
27284 \c_group_begin_token
\vbox_gset_end:
27285 \color_group_begin:
27286 }
27287 \cs_new_protected:Npn \vbox_gset:Nw #1
27288 {
27289 \tex_global:D \tex_setbox:D #1 \tex_vbox:D
27290 \c_group_begin_token
27291 \color_group_begin:
27292 }
27293 \cs_generate_variant:Nn \vbox_set:Nw { c }
27294 \cs_generate_variant:Nn \vbox_gset:Nw { c }
27295 \cs_new_protected:Npn \vbox_set_end:
27296 {
27297 \par
27298 \color_group_end:
27299 \c_group_end_token
27300 }
27301 \cs_new_eq:NN \vbox_gset_end: \vbox_set_end:

```

(End definition for `\vbox_set:Nw` and others. These functions are documented on page 245.)

**\vbox\_set\_to\_ht:Nnw** A combination of the above ideas.

```

\vbox_set_to_ht:cw
27302 \cs_new_protected:Npn \vbox_set_to_ht:Nnw #1#2
\vbox_gset_to_ht:Nnw
27303 {
27304 \tex_setbox:D #1 \tex_vbox:D to __box_dim_eval:n {#2}
27305 \c_group_begin_token
27306 \color_group_begin:
27307 }
27308 \cs_new_protected:Npn \vbox_gset_to_ht:Nnw #1#2
27309 {
27310 \tex_global:D \tex_setbox:D #1 \tex_vbox:D to __box_dim_eval:n {#2}
27311 \c_group_begin_token
27312 \color_group_begin:
27313 }
27314 \cs_generate_variant:Nn \vbox_set_to_ht:Nnw { c }
27315 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnw { c }

```

(End definition for `\vbox_set_to_ht:Nnw` and `\vbox_gset_to_ht:Nnw`. These functions are documented on page 245.)

**\vbox\_unpack:N** Unpacking a box and if requested also clear it.

```
27316 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
27317 \cs_new_eq:NN \vbox_unpack_drop:N \tex_unvbox:D
27318 \cs_generate_variant:Nn \vbox_unpack:N { c }
27319 \cs_generate_variant:Nn \vbox_unpack_drop:N { c }
```

(End definition for **\vbox\_unpack:N** and **\vbox\_unpack\_drop:N**. These functions are documented on page 245.)

**\vbox\_set\_split\_to\_ht:NNn** Splitting a vertical box in two.

```
27320 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
27321 { \tex_setbox:D #1 \tex_vsplit:D #2 to __box_dim_eval:n {#3} }
27322 \cs_generate_variant:Nn \vbox_set_split_to_ht:NNn { c , Nc , cc }
27323 \cs_new_protected:Npn \vbox_gset_split_to_ht:NNn #1#2#3
27324 {
27325 \tex_global:D \tex_setbox:D #1
27326 \tex_vsplit:D #2 to __box_dim_eval:n {#3}
27327 }
27328 \cs_generate_variant:Nn \vbox_gset_split_to_ht:NNn { c , Nc , cc }
```

(End definition for **\vbox\_set\_split\_to\_ht:NNn** and **\vbox\_gset\_split\_to\_ht:NNn**. These functions are documented on page 245.)

## 42.12 Affine transformations

**\l\_box\_angle\_fp**

When rotating boxes, the angle itself may be needed by the engine-dependent code. This is done using the **fp** module so that the value is tidied up properly.

```
27329 \fp_new:N \l_box_angle_fp
```

(End definition for **\l\_box\_angle\_fp**.)

**\l\_box\_cos\_fp**

**\l\_box\_sin\_fp**

These are used to hold the calculated sine and cosine values while carrying out a rotation.

```
27330 \fp_new:N \l_box_cos_fp
27331 \fp_new:N \l_box_sin_fp
```

(End definition for **\l\_box\_cos\_fp** and **\l\_box\_sin\_fp**.)

**\l\_box\_top\_dim**

**\l\_box\_bottom\_dim**

**\l\_box\_left\_dim**

**\l\_box\_right\_dim**

These are the positions of the four edges of a box before manipulation.

```
27332 \dim_new:N \l_box_top_dim
27333 \dim_new:N \l_box_bottom_dim
27334 \dim_new:N \l_box_left_dim
27335 \dim_new:N \l_box_right_dim
```

(End definition for **\l\_box\_top\_dim** and others.)

**\l\_box\_top\_new\_dim**

**\l\_box\_bottom\_new\_dim**

**\l\_box\_left\_new\_dim**

**\l\_box\_right\_new\_dim**

These are the positions of the four edges of a box after manipulation.

```
27336 \dim_new:N \l_box_top_new_dim
27337 \dim_new:N \l_box_bottom_new_dim
27338 \dim_new:N \l_box_left_new_dim
27339 \dim_new:N \l_box_right_new_dim
```

(End definition for **\l\_box\_top\_new\_dim** and others.)

**\l\_box\_internal\_box**

Scratch space, but also needed by some parts of the driver.

```
27340 \box_new:N \l_box_internal_box
```

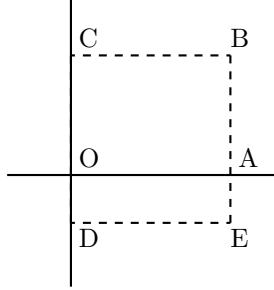


Figure 1: Co-ordinates of a box prior to rotation.

(End definition for `\l_box_internal_box`.)

`\box_rotate:Nn`  
`\box_rotate:cn`  
`\box_grotate:Nn`  
`\box_grotate:cn`

Rotation of a box starts with working out the relevant sine and cosine. The actual rotation is in an auxiliary to keep the flow slightly clearer

```

27341 \cs_new_protected:Npn \box_rotate:Nn #1#2
27342 { __box_rotate:NnN #1 {#2} \hbox_set:Nn }
27343 \cs_generate_variant:Nn \box_rotate:Nn { c }
27344 \cs_new_protected:Npn \box_grotate:Nn #1#2
27345 { __box_rotate:NnN #1 {#2} \hbox_gset:Nn }
27346 \cs_generate_variant:Nn \box_grotate:Nn { c }
27347 \cs_new_protected:Npn __box_rotate:NnN #1#2#3
27348 {
27349 #3 #1
27350 {
27351 \fp_set:Nn \l__box_angle_fp {#2}
27352 \fp_set:Nn \l__box_sin_fp { sind (\l__box_angle_fp) }
27353 \fp_set:Nn \l__box_cos_fp { cosd (\l__box_angle_fp) }
27354 __box_rotate:N #1
27355 }
27356 }
```

The edges of the box are then recorded: the left edge is always at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.

```

27357 \cs_new_protected:Npn __box_rotate:N #1
27358 {
27359 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
27360 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
27361 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
27362 \dim_zero:N \l__box_left_dim
```

The next step is to work out the  $x$  and  $y$  coordinates of vertices of the rotated box in relation to its original coordinates. The box can be visualized with vertices  $B$ ,  $C$ ,  $D$  and  $E$  is illustrated (Figure 1). The vertex  $O$  is the reference point on the baseline, and in this implementation is also the centre of rotation. The formulae are, for a point  $P$  and angle  $\alpha$ :

$$\begin{aligned}
P'_x &= P_x - O_x \\
P'_y &= P_y - O_y \\
P''_x &= (P'_x \cos(\alpha)) - (P'_y \sin(\alpha)) \\
P''_y &= (P'_x \sin(\alpha)) + (P'_y \cos(\alpha)) \\
P'''_x &= P''_x + O_x + L_x \\
P'''_y &= P''_y + O_y
\end{aligned}$$

The “extra” horizontal translation  $L_x$  at the end is calculated so that the leftmost point of the resulting box has  $x$ -coordinate 0. This is desirable as TeX boxes must have the reference point at the left edge of the box. (As  $O$  is always  $(0,0)$ , this part of the calculation is omitted here.)

```

27363 \fp_compare:nNnTF \l_box_sin_fp > \c_zero_fp
27364 {
27365 \fp_compare:nNnTF \l_box_cos_fp > \c_zero_fp
27366 { __box_rotate_quadrant_one: }
27367 { __box_rotate_quadrant_two: }
27368 }
27369 {
27370 \fp_compare:nNnTF \l_box_cos_fp < \c_zero_fp
27371 { __box_rotate_quadrant_three: }
27372 { __box_rotate_quadrant_four: }
27373 }
```

The position of the box edges are now known, but the box at this stage be misplaced relative to the current TeX reference point. So the content of the box is moved such that the reference point of the rotated box is in the same place as the original.

```

27374 \hbox_set:Nn \l_box_internal_box { \box_use:N #1 }
27375 \hbox_set:Nn \l_box_internal_box
27376 {
27377 \tex_kern:D -\l_box_left_new_dim
27378 \hbox:n
27379 {
27380 __box_backend_rotate:Nn
27381 \l_box_internal_box
27382 \l_box_angle_fp
27383 }
27384 }
```

Tidy up the size of the box so that the material is actually inside the bounding box. The result can then be used to reset the original box.

```

27385 \box_set_ht:Nn \l_box_internal_box { \l_box_top_new_dim }
27386 \box_set_dp:Nn \l_box_internal_box { -\l_box_bottom_new_dim }
27387 \box_set_wd:Nn \l_box_internal_box
27388 { \l_box_right_new_dim - \l_box_left_new_dim }
27389 \box_use_drop:N \l_box_internal_box
27390 }
```

These functions take a general point (#1,#2) and rotate its location about the origin, using the previously-set sine and cosine values. Each function gives only one component of the location of the updated point. This is because for rotation of a box each step needs only one value, and so performance is gained by avoiding working out both  $x'$  and  $y'$  at the same time. Contrast this with the equivalent function in the l3coffins module, where both parts are needed.

```

27391 \cs_new_protected:Npn __box_rotate_xdir:nnN #1#2#3
27392 {
27393 \dim_set:Nn #3
27394 {
27395 \fp_to_dim:n
27396 {
27397 \l_box_cos_fp * \dim_to_fp:n {#1}
27398 - \l_box_sin_fp * \dim_to_fp:n {#2}
```

```

27399 }
27400 }
27401 }
27402 \cs_new_protected:Npn __box_rotate_ydir:nnN #1#2#3
27403 {
27404 \dim_set:Nn #3
27405 {
27406 \fp_to_dim:n
27407 {
27408 \l__box_sin_fp * \dim_to_fp:n {#1}
27409 + \l__box_cos_fp * \dim_to_fp:n {#2}
27410 }
27411 }
27412 }

```

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting  $y$ -values, whereas the left and right edges need the  $x$ -values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

```

27413 \cs_new_protected:Npn __box_rotate_quadrant_one:
27414 {
27415 __box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
27416 \l__box_top_new_dim
27417 __box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
27418 \l__box_bottom_new_dim
27419 __box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
27420 \l__box_left_new_dim
27421 __box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
27422 \l__box_right_new_dim
27423 }
27424 \cs_new_protected:Npn __box_rotate_quadrant_two:
27425 {
27426 __box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
27427 \l__box_top_new_dim
27428 __box_rotate_ydir:nnN \l__box_left_dim \l__box_top_dim
27429 \l__box_bottom_new_dim
27430 __box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
27431 \l__box_left_new_dim
27432 __box_rotate_xdir:nnN \l__box_left_dim \l__box_bottom_dim
27433 \l__box_right_new_dim
27434 }
27435 \cs_new_protected:Npn __box_rotate_quadrant_three:
27436 {
27437 __box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
27438 \l__box_top_new_dim
27439 __box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
27440 \l__box_bottom_new_dim
27441 __box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
27442 \l__box_left_new_dim
27443 __box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
27444 \l__box_right_new_dim
27445 }
27446 \cs_new_protected:Npn __box_rotate_quadrant_four:

```

```

27447 {
27448 __box_rotate_ydir:nnN \l_box_left_dim \l_box_top_dim
27449 \l_box_top_new_dim
27450 __box_rotate_ydir:nnN \l_box_right_dim \l_box_bottom_dim
27451 \l_box_bottom_new_dim
27452 __box_rotate_xdir:nnN \l_box_left_dim \l_box_bottom_dim
27453 \l_box_left_new_dim
27454 __box_rotate_xdir:nnN \l_box_right_dim \l_box_top_dim
27455 \l_box_right_new_dim
27456 }

```

(End definition for `\box_rotate:Nn` and others. These functions are documented on page 249.)

`\l_box_scale_x_fp`  
`\l_box_scale_y_fp`

```

27457 \fp_new:N \l_box_scale_x_fp
27458 \fp_new:N \l_box_scale_y_fp

```

(End definition for `\l_box_scale_x_fp` and `\l_box_scale_y_fp`.)

Resizing a box starts by working out the various dimensions of the existing box.

```

27459 \cs_new_protected:Npn \box_resize_to_wd_and_ht_plus_dp:Nnn
27460 {
27461 __box_resize_to_wd_and_ht_plus_dp:NnnN #1{#2}{#3}
27462 \hbox_set:Nn
27463 }
27464 \cs_generate_variant:Nn \box_resize_to_wd_and_ht_plus_dp:Nnn { c }
27465 \cs_new_protected:Npn \box_gresize_to_wd_and_ht_plus_dp:Nnn #1{#2}{#3}
27466 {
27467 __box_resize_to_wd_and_ht_plus_dp:NnnN #1{#2}{#3}
27468 \hbox_gset:Nn
27469 }
27470 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht_plus_dp:Nnn { c }
27471 \cs_new_protected:Npn __box_resize_to_wd_and_ht_plus_dp:NnnN #1{#2}{#3}{#4}
27472 {
27473 #4 #1
27474 {
27475 __box_resize_set_corners:N #1

```

The *x*-scaling and resulting box size is easy enough to work out: the dimension is that given as #2, and the scale is simply the new width divided by the old one.

```

27476 \fp_set:Nn \l_box_scale_x_fp
27477 { \dim_to_fp:n {#2} / \dim_to_fp:n { \l_box_right_dim } }

```

The *y*-scaling needs both the height and the depth of the current box.

```

27478 \fp_set:Nn \l_box_scale_y_fp
27479 {
27480 \dim_to_fp:n {#3}
27481 / \dim_to_fp:n { \l_box_top_dim - \l_box_bottom_dim }
27482 }

```

Hand off to the auxiliary which does the rest of the work.

```

27483 __box_resize:N #1
27484 }
27485 }
27486 \cs_new_protected:Npn __box_resize_set_corners:N #1
27487 {

```

```

27488 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
27489 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
27490 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
27491 \dim_zero:N \l__box_left_dim
27492 }
```

With at least one real scaling to do, the next phase is to find the new edge co-ordinates. In the  $x$  direction this is relatively easy: just scale the right edge. In the  $y$  direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

```

27493 \cs_new_protected:Npn __box_resize:N #1
27494 {
27495 __box_resize>NNN \l__box_right_new_dim
27496 \l__box_scale_x_fp \l__box_right_dim
27497 __box_resize>NNN \l__box_bottom_new_dim
27498 \l__box_scale_y_fp \l__box_bottom_dim
27499 __box_resize>NNN \l__box_top_new_dim
27500 \l__box_scale_y_fp \l__box_top_dim
27501 __box_resize_common:N #1
27502 }
27503 \cs_new_protected:Npn __box_resize>NNN #1#2#3
27504 {
27505 \dim_set:Nn #1
27506 { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }
27507 }
```

(End definition for `\box_resize_to_wd_and_ht_plus_dp:Nnn` and others. These functions are documented on page 248.)

`\box_resize_to_ht:Nn`  
`\box_resize_to_ht:cn`  
`\box_gresize_to_ht:Nn`  
`\box_gresize_to_ht:cn`

`\__box_resize_to_ht:Nnn`  
`\box_resize_to_ht_plus_dp:Nn`  
`\box_resize_to_ht_plus_dp:cn`  
`\box_gresize_to_ht_plus_dp:Nn`  
`\box_gresize_to_ht_plus_dp:cn`  
`\__box_resize_to_ht_plus_dp:Nnn`  
`\box_resize_to_wd:Nn`  
`\box_resize_to_wd:cn`  
`\box_gresize_to_wd:Nn`  
`\box_gresize_to_wd:cn`  
`\__box_resize_to_wd:Nnn`  
`\box_resize_to_wd_and_ht:Nnn`  
`\box_resize_to_wd_and_ht:cnn`  
`\box_gresize_to_wd_and_ht:Nnn`  
`\box_gresize_to_wd_and_ht:cnn`  
`\__box_resize_to_wd_ht:NnnN`

Scaling to a (total) height or to a width is a simplified version of the main resizing operation, with the scale simply copied between the two parts. The internal auxiliary is called using the scaling value twice, as the sign for both parts is needed (as this allows the same internal code to be used as for the general case).

```

27508 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2
27509 { __box_resize_to_ht:Nnn #1 {#2} \hbox_set:Nn }
27510 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c }
27511 \cs_new_protected:Npn \box_gresize_to_ht:Nn #1#2
27512 { __box_resize_to_ht:Nnn #1 {#2} \hbox_gset:Nn }
27513 \cs_generate_variant:Nn \box_gresize_to_ht:Nn { c }
27514 \cs_new_protected:Npn __box_resize_to_ht:Nnn #1#2#3
27515 {
27516 #3 #1
27517 {
27518 __box_resize_set_corners:N #1
27519 \fp_set:Nn \l__box_scale_y_fp
27520 {
27521 \dim_to_fp:n {#2}
27522 / \dim_to_fp:n { \l__box_top_dim }
27523 }
27524 \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
27525 __box_resize:N #1
27526 }
27527 }
27528 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2
```

```

27529 { __box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_set:Nn }
27530 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
27531 \cs_new_protected:Npn \box_gresize_to_ht_plus_dp:Nn #1#2
27532 { __box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_gset:Nn }
27533 \cs_generate_variant:Nn \box_gresize_to_ht_plus_dp:Nn { c }
27534 \cs_new_protected:Npn __box_resize_to_ht_plus_dp:NnN #1#2#3
27535 {
27536 \hbox_set:Nn #1
27537 {
27538 __box_resize_set_corners:N #1
27539 \fp_set:Nn \l__box_scale_y_fp
27540 {
27541 \dim_to_fp:n {#2}
27542 / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
27543 }
27544 \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
27545 __box_resize:N #1
27546 }
27547 }
27548 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
27549 { __box_resize_to_wd:NnN #1 {#2} \hbox_set:Nn }
27550 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
27551 \cs_new_protected:Npn \box_gresize_to_wd:Nn #1#2
27552 { __box_resize_to_wd:NnN #1 {#2} \hbox_gset:Nn }
27553 \cs_generate_variant:Nn \box_gresize_to_wd:Nn { c }
27554 \cs_new_protected:Npn __box_resize_to_wd:NnN #1#2#3
27555 {
27556 #3 #1
27557 {
27558 __box_resize_set_corners:N #1
27559 \fp_set:Nn \l__box_scale_x_fp
27560 { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
27561 \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
27562 __box_resize:N #1
27563 }
27564 }
27565 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
27566 { __box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_set:Nn }
27567 \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }
27568 \cs_new_protected:Npn \box_gresize_to_wd_and_ht:Nnn #1#2#3
27569 { __box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_gset:Nn }
27570 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht:Nnn { c }
27571 \cs_new_protected:Npn __box_resize_to_wd_and_ht:NnnN #1#2#3#4
27572 {
27573 #4 #1
27574 {
27575 __box_resize_set_corners:N #1
27576 \fp_set:Nn \l__box_scale_x_fp
27577 { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
27578 \fp_set:Nn \l__box_scale_y_fp
27579 {
27580 \dim_to_fp:n {#3}
27581 / \dim_to_fp:n { \l__box_top_dim }
27582 }

```

```

27583 __box_resize:N #1
27584 }
27585 }

```

(End definition for `\box_resize_to_ht:Nn` and others. These functions are documented on page 247.)

```

\box_scale:Nnn
\box_scale:cnn
\box_gscale:Nnn
\box_gscale:cnn
_box_scale:NnnN
 __box_scale:N

```

When scaling a box, setting the scaling itself is easy enough. The new dimensions are also relatively easy to find, allowing only for the need to keep them positive in all cases. Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. The code here is split into two as this allows sharing with the auto-resizing functions.

```

27586 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
27587 { __box_scale:NnnN #1 {#2} {#3} \hbox_set:Nn }
27588 \cs_generate_variant:Nn \box_scale:Nnn { c }
27589 \cs_new_protected:Npn \box_gscale:Nnn #1#2#3
27590 { __box_scale:NnnN #1 {#2} {#3} \hbox_gset:Nn }
27591 \cs_generate_variant:Nn \box_gscale:Nnn { c }
27592 \cs_new_protected:Npn __box_scale:NnnN #1#2#3#4
27593 {
27594 #4 #1
27595 {
27596 \fp_set:Nn \l__box_scale_x_fp {#2}
27597 \fp_set:Nn \l__box_scale_y_fp {#3}
27598 __box_scale:N #1
27599 }
27600 }
27601 \cs_new_protected:Npn __box_scale:N #1
27602 {
27603 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
27604 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
27605 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
27606 \dim_zero:N \l__box_left_dim
27607 \dim_set:Nn \l__box_top_new_dim
27608 { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
27609 \dim_set:Nn \l__box_bottom_new_dim
27610 { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
27611 \dim_set:Nn \l__box_right_new_dim
27612 { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
27613 __box_resize_common:N #1
27614 }

```

(End definition for `\box_scale:Nnn` and others. These functions are documented on page 249.)

Although autosizing a box uses dimensions, it has more in common in implementation with scaling. As such, most of the real work here is done elsewhere.

```

27615 \cs_new_protected:Npn \box_autosize_to_wd_and_ht:Nnn #1#2#3
27616 { __box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_set:Nn }
27617 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
27618 \cs_new_protected:Npn \box_gautosize_to_wd_and_ht:Nnn #1#2#3
27619 { __box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_gset:Nn }
27620 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
27621 \cs_new_protected:Npn \box_autosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
27622 {
27623 __box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }

```

```

27624 \hbox_set:Nn
27625 }
27626 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht_plus_dp:Nnn { c }
27627 \cs_new_protected:Npn \box_gautosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
27628 {
27629 __box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }
27630 \hbox_gset:Nn
27631 }
27632 \cs_generate_variant:Nn \box_gautosize_to_wd_and_ht_plus_dp:Nnn { c }
27633 \cs_new_protected:Npn __box_autosize:NnnnN #1#2#3#4#5
27634 {
27635 #5 #1
27636 {
27637 \fp_set:Nn \l__box_scale_x_fp { (#2) / \box_wd:N #1 }
27638 \fp_set:Nn \l__box_scale_y_fp { (#3) / (#4) }
27639 \fp_compare:nNnTF \l__box_scale_x_fp > \l__box_scale_y_fp
27640 { \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp }
27641 { \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp }
27642 __box_scale:N #1
27643 }
27644 }

```

(End definition for `\box_autosize_to_wd_and_ht:Nnn` and others. These functions are documented on page 247.)

`\__box_resize_common:N` The main resize function places its input into a box which start off with zero width, and includes the handles for engine rescaling.

```

27645 \cs_new_protected:Npn __box_resize_common:N #1
27646 {
27647 \hbox_set:Nn \l__box_internal_box
27648 {
27649 __box_backend_scale:Nnn
27650 #1
27651 \l__box_scale_x_fp
27652 \l__box_scale_y_fp
27653 }

```

The new height and depth can be applied directly.

```

27654 \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
27655 {
27656 \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
27657 \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
27658 }
27659 {
27660 \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
27661 \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
27662 }

```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.

```

27663 \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
27664 {
27665 \hbox_to_wd:nn { \l__box_right_new_dim }

```

```

27666 {
27667 \tex_kern:D \l_box_right_new_dim
27668 \box_use_drop:N \l_box_internal_box
27669 \tex_hss:D
27670 }
27671 }
27672 {
27673 \box_set_wd:Nn \l_box_internal_box { \l_box_right_new_dim }
27674 \hbox:n
27675 {
27676 \tex_kern:D \c_zero_dim
27677 \box_use_drop:N \l_box_internal_box
27678 \tex_hss:D
27679 }
27680 }
27681 }

(End definition for _box_resize_common:N.)
```

27682 </package>

## 43 I3coffins Implementation

```

27683 <*package>
27684 <@=coffin>
```

### 43.1 Coffins: data structures and general variables

\l\_coffin\_internal\_box

```

27685 \box_new:N \l_coffin_internal_box
27686 \dim_new:N \l_coffin_internal_dim
27687 \tl_new:N \l_coffin_internal_tl
```

(End definition for \l\_coffin\_internal\_box, \l\_coffin\_internal\_dim, and \l\_coffin\_internal\_tl.)

\c\_coffin\_corners\_prop

The “corners”; of a coffin define the real content, as opposed to the TeX bounding box. They all start off in the same place, of course.

```

27688 \prop_const_from_keyval:Nn \c_coffin_corners_prop
27689 {
27690 tl = { Opt } { Opt } ,
27691 tr = { Opt } { Opt } ,
27692 bl = { Opt } { Opt } ,
27693 br = { Opt } { Opt } ,
27694 }
```

(End definition for \c\_coffin\_corners\_prop.)

\c\_coffin\_poles\_prop

Pole positions are given for horizontal, vertical and reference-point based values.

```

27695 \prop_const_from_keyval:Nn \c_coffin_poles_prop
27696 {
27697 l = { Opt } { Opt } { Opt } { 1000pt } ,
27698 hc = { Opt } { Opt } { Opt } { 1000pt } ,
27699 r = { Opt } { Opt } { Opt } { 1000pt } ,
```

```

27700 b = { Opt } { Opt } { 1000pt } { Opt } ,
27701 vc = { Opt } { Opt } { 1000pt } { Opt } ,
27702 t = { Opt } { Opt } { 1000pt } { Opt } ,
27703 B = { Opt } { Opt } { 1000pt } { Opt } ,
27704 H = { Opt } { Opt } { 1000pt } { Opt } ,
27705 T = { Opt } { Opt } { 1000pt } { Opt } ,
27706 }

```

(End definition for `\c_coffin_poles_prop.`)

`\l_coffin_slope_A_fp` Used for calculations of intersections.

```

27707 \fp_new:N \l_coffin_slope_A_fp
27708 \fp_new:N \l_coffin_slope_B_fp

```

(End definition for `\l_coffin_slope_A_fp` and `\l_coffin_slope_B_fp.`)

`\l_coffin_error_bool` For propagating errors so that parts of the code can work around them.

```
27709 \bool_new:N \l_coffin_error_bool
```

(End definition for `\l_coffin_error_bool.`)

`\l_coffin_offset_x_dim` The offset between two sets of coffin handles when typesetting. These values are corrected from those requested in an alignment for the positions of the handles.  
`\l_coffin_offset_y_dim`

```

27710 \dim_new:N \l_coffin_offset_x_dim
27711 \dim_new:N \l_coffin_offset_y_dim

```

(End definition for `\l_coffin_offset_x_dim` and `\l_coffin_offset_y_dim.`)

`\l_coffin_pole_a_t1` Needed for finding the intersection of two poles.

```

27712 \tl_new:N \l_coffin_pole_a_t1
27713 \tl_new:N \l_coffin_pole_b_t1

```

(End definition for `\l_coffin_pole_a_t1` and `\l_coffin_pole_b_t1.`)

`\l_coffin_x_dim` For calculating intersections and so forth.  
`\l_coffin_y_dim`  
`\l_coffin_x_prime_dim`  
`\l_coffin_y_prime_dim`

```

27714 \dim_new:N \l_coffin_x_dim
27715 \dim_new:N \l_coffin_y_dim
27716 \dim_new:N \l_coffin_x_prime_dim
27717 \dim_new:N \l_coffin_y_prime_dim

```

(End definition for `\l_coffin_x_dim` and others.)

## 43.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

`\_coffin_to_value:N` Coffins are a two-part structure and we rely on the internal nature of box allocation to make everything work. As such, we need an interface to turn coffin identifiers into numbers. For the purposes here, the signature allowed is `N` despite the nature of the underlying primitive.

```
27718 \cs_new_eq:NN _coffin_to_value:N \tex_number:D
```

(End definition for `\_coffin_to_value:N.`)

**\coffin\_if\_exist\_p:N** Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

```

27719 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
27720 {
27721 \cs_if_exist:NTF #1
27722 {
27723 \cs_if_exist:cTF { coffin ~ __coffin_to_value:N #1 ~ poles }
27724 { \prg_return_true: }
27725 { \prg_return_false: }
27726 }
27727 { \prg_return_false: }
27728 }
27729 \prg_generate_conditional_variant:Nnn \coffin_if_exist:N
27730 { c } { p , T , F , TF }
```

(End definition for `\coffin_if_exist:NTF`. This function is documented on page 250.)

**\\_\_coffin\_if\_exist:NT** Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. So a wrapper is provided to deal with this correctly, issuing an error on erroneous use.

```

27731 \cs_new_protected:Npn __coffin_if_exist:NT #1#2
27732 {
27733 \coffin_if_exist:NTF #1
27734 { #2 }
27735 {
27736 __kernel_msg_error:nxx { kernel } { unknown-coffin }
27737 { \token_to_str:N #1 }
27738 }
27739 }
```

(End definition for `\__coffin_if_exist:NT`.)

**\coffin\_clear:N** Clearing coffins means emptying the box and resetting all of the structures.

```

27740 \cs_new_protected:Npn \coffin_clear:N #1
27741 {
27742 __coffin_if_exist:NT #1
27743 {
27744 \box_clear:N #1
27745 __coffin_reset_structure:N #1
27746 }
27747 }
27748 \cs_generate_variant:Nn \coffin_clear:N { c }
27749 \cs_new_protected:Npn \coffin_gclear:N #1
27750 {
27751 __coffin_if_exist:NT #1
27752 {
27753 \box_gclear:N #1
27754 __coffin_greset_structure:N #1
27755 }
27756 }
27757 \cs_generate_variant:Nn \coffin_gclear:N { c }
```

(End definition for `\coffin_clear:N` and `\coffin_gclear:N`. These functions are documented on page 250.)

**\coffin\_new:N** Creating a new coffin means making the underlying box and adding the data structures. The `\debug_suspend:` and `\debug_resume:` functions prevent `\prop_gclear_new:c` from writing useless information to the log file.

```

27758 \cs_new_protected:Npn \coffin_new:N #1
27759 {
27760 \box_new:N #1
27761 \debug_suspend:
27762 \prop_gclear_new:c { coffin ~ __coffin_to_value:N #1 ~ corners }
27763 \prop_gclear_new:c { coffin ~ __coffin_to_value:N #1 ~ poles }
27764 \prop_gset_eq:cN { coffin ~ __coffin_to_value:N #1 ~ corners }
27765 \c_coffin_corners_prop
27766 \prop_gset_eq:cN { coffin ~ __coffin_to_value:N #1 ~ poles }
27767 \c_coffin_poles_prop
27768 \debug_resume:
27769 }
27770 \cs_generate_variant:Nn \coffin_new:N { c }

```

(End definition for `\coffin_new:N`. This function is documented on page 250.)

**\hcoffin\_set:Nn** Horizontal coffins are relatively easy: set the appropriate box, reset the structures then update the handle positions.

```

27771 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
27772 {
27773 __coffin_if_exist:NT #1
27774 {
27775 \hbox_set:Nn #1
27776 {
27777 \color_ensure_current:
27778 #2
27779 }
27780 __coffin_update:N #1
27781 }
27782 }
27783 \cs_generate_variant:Nn \hcoffin_set:Nn { c }
27784 \cs_new_protected:Npn \hcoffin_gset:Nn #1#2
27785 {
27786 __coffin_if_exist:NT #1
27787 {
27788 \hbox_gset:Nn #1
27789 {
27790 \color_ensure_current:
27791 #2
27792 }
27793 __coffin_gupdate:N #1
27794 }
27795 }
27796 \cs_generate_variant:Nn \hcoffin_gset:Nn { c }

```

(End definition for `\hcoffin_set:Nn` and `\hcoffin_gset:Nn`. These functions are documented on page 250.)

**\vcoffin\_set:Nnn** Setting vertical coffins is more complex. First, the material is typeset with a given width. The default handles and poles are set as for a horizontal coffin, before finding the top baseline using a temporary box. No `\color_ensure_current:` here as that would add a

```

__coffin_set_vertical:NnnNN
__coffin_set_vertical_aux:

```

whatsit to the start of the vertical box and mess up the location of the T pole (see *T<sub>EX</sub> by Topic* for discussion of the \vtop primitive, used to do the measuring).

```

27797 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
27798 {
27799 __coffin_set_vertical:NnnNN #1 {#2} {#3}
27800 \vbox_set:Nn __coffin_update:N
27801 }
27802 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }
27803 \cs_new_protected:Npn \vcoffin_gset:Nnn #1#2#3
27804 {
27805 __coffin_set_vertical:NnnNN #1 {#2} {#3}
27806 \vbox_gset:Nn __coffin_gupdate:N
27807 }
27808 \cs_generate_variant:Nn \vcoffin_gset:Nnn { c }
27809 \cs_new_protected:Npn __coffin_set_vertical:NnnNN #1#2#3#4#5
27810 {
27811 __coffin_if_exist:NT #1
27812 {
27813 #4 #1
27814 {
27815 \dim_set:Nn \tex_hsize:D {#2}
27816 __coffin_set_vertical_aux:
27817 #3
27818 }
27819 #5 #1
27820 \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
27821 __coffin_set_pole:Nnx #1 { T }
27822 {
27823 { Opt }
27824 {
27825 \dim_eval:n
27826 { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
27827 }
27828 { 1000pt }
27829 { Opt }
27830 }
27831 \box_clear:N \l__coffin_internal_box
27832 }
27833 }
27834 \cs_new_protected:Npx __coffin_set_vertical_aux:
27835 {
27836 \cs_if_exist:NT \linewidth
27837 { \dim_set_eq:NN \linewidth \tex_hsize:D }
27838 \cs_if_exist:NT \columnwidth
27839 { \dim_set_eq:NN \columnwidth \tex_hsize:D }
27840 }

```

(End definition for \vcoffin\_set:Nnn and others. These functions are documented on page 251.)

**\hcoffin\_set:Nw** These are the “begin”/“end” versions of the above: watch the grouping!

```

\hcoffin_set:cw 27841 \cs_new_protected:Npn \hcoffin_set:Nw #1
\hcoffin_gset:Nw 27842 {
\hcoffin_gset:cw 27843 __coffin_if_exist:NT #1
\hcoffin_set_end: 27844 {
\hcoffin_gset_end:

```

```

27845 \hbox_set:Nw #1 \color_ensure_current:
27846 \cs_set_protected:Npn \hcoffin_set_end:
27847 {
27848 \hbox_set_end:
27849 __coffin_update:N #1
27850 }
27851 }
27852 }
27853 \cs_generate_variant:Nn \hcoffin_set:Nw { c }
27854 \cs_new_protected:Npn \hcoffin_gset:Nw #1
27855 {
27856 __coffin_if_exist:NT #1
27857 {
27858 \hbox_gset:Nw #1 \color_ensure_current:
27859 \cs_set_protected:Npn \hcoffin_gset_end:
27860 {
27861 \hbox_gset_end:
27862 __coffin_gupdate:N #1
27863 }
27864 }
27865 }
27866 \cs_generate_variant:Nn \hcoffin_gset:Nw { c }
27867 \cs_new_protected:Npn \hcoffin_set_end: { }
27868 \cs_new_protected:Npn \hcoffin_gset_end: { }

```

(End definition for `\hcoffin_set:Nw` and others. These functions are documented on page 251.)

### `\vcoffin_set:Nnw`

```

\vc coffin_set:cnw
\vc coffin_set:cnw
\vc coffin_gset:Nnw
\vc coffin_gset:cnw
__coffin_set_vertical:NnNNNNw
\vc coffin_set_end:
\vc coffin_gset_end:
27869 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
27870 {
27871 __coffin_set_vertical:NnNNNNw #1 {#2} \vbox_set:Nw
27872 \vcoffin_set_end:
27873 \vbox_set_end: __coffin_update:N
27874 }
27875 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }
27876 \cs_new_protected:Npn \vcoffin_gset:Nnw #1#2
27877 {
27878 __coffin_set_vertical:NnNNNNw #1 {#2} \vbox_gset:Nw
27879 \vcoffin_gset_end:
27880 \vbox_gset_end: __coffin_gupdate:N
27881 }
27882 \cs_generate_variant:Nn \vcoffin_gset:Nnw { c }
27883 \cs_new_protected:Npn __coffin_set_vertical:NnNNNNw #1#2#3#4#5#6
27884 {
27885 __coffin_if_exist:NT #1
27886 {
27887 #3 #1
27888 \dim_set:Nn \tex_hsize:D {#2}
27889 __coffin_set_vertical_aux:
27890 \cs_set_protected:Npn #4
27891 {
27892 #5
27893 #6 #1
27894 \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }

```

```

27895 __coffin_set_pole:Nnx #1 { T }
27896 {
27897 { Opt }
27898 {
27899 \dim_eval:n
27900 { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
27901 }
27902 { 1000pt }
27903 { Opt }
27904 }
27905 \box_clear:N \l__coffin_internal_box
27906 }
27907 }
27908 }
27909 \cs_new_protected:Npn \vcoffin_set_end: { }
27910 \cs_new_protected:Npn \vcoffin_gset_end: { }

```

(End definition for `\vcoffin_set:Nnw` and others. These functions are documented on page 251.)

**\coffin\_set\_eq:NN** Setting two coffins equal is just a wrapper around other functions.

```

27911 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
27912 {
27913 __coffin_if_exist:NT #1
27914 {
27915 \box_set_eq:NN #1 #2
27916 \prop_set_eq:cc { coffin ~ __coffin_to_value:N #1 ~ corners }
27917 { coffin ~ __coffin_to_value:N #2 ~ corners }
27918 \prop_set_eq:cc { coffin ~ __coffin_to_value:N #1 ~ poles }
27919 { coffin ~ __coffin_to_value:N #2 ~ poles }
27920 }
27921 }
27922 \cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
27923 \cs_new_protected:Npn \coffin_gset_eq:NN #1#2
27924 {
27925 __coffin_if_exist:NT #1
27926 {
27927 \box_gset_eq:NN #1 #2
27928 \prop_gset_eq:cc { coffin ~ __coffin_to_value:N #1 ~ corners }
27929 { coffin ~ __coffin_to_value:N #2 ~ corners }
27930 \prop_gset_eq:cc { coffin ~ __coffin_to_value:N #1 ~ poles }
27931 { coffin ~ __coffin_to_value:N #2 ~ poles }
27932 }
27933 }
27934 \cs_generate_variant:Nn \coffin_gset_eq:NN { c , Nc , cc }

```

(End definition for `\coffin_set_eq:NN` and `\coffin_gset_eq:NN`. These functions are documented on page 250.)

**\c\_empty\_coffin**

Special coffins: these cannot be set up earlier as they need `\coffin_new:N`. The empty coffin is set as a box as the full coffin-setting system needs some material which is not yet available. The empty coffin is created entirely by hand: not everything is in place yet.

```

27935 \coffin_new:N \c_empty_coffin
27936 \coffin_new:N \l__coffin_aligned_coffin
27937 \coffin_new:N \l__coffin_aligned_internal_coffin

```

(End definition for `\c_empty_coffin`, `\l_coffin_aligned_coffin`, and `\l_coffin_aligned_internal_coffin`. This variable is documented on page 254.)

`\l_tmpa_coffin` The usual scratch space.  
`\l_tmpb_coffin` 27938 `\coffin_new:N \l_tmpa_coffin`  
`\g_tmpa_coffin` 27939 `\coffin_new:N \l_tmpb_coffin`  
`\g_tmpb_coffin` 27940 `\coffin_new:N \g_tmpa_coffin`  
27941 `\coffin_new:N \g_tmpb_coffin`

(End definition for `\l_tmpa_coffin` and others. These variables are documented on page 254.)

### 43.3 Measuring coffins

`\coffin_dp:N` Coffins are just boxes when it comes to measurement. However, semantically a separate  
`\coffin_dp:c` set of functions are required.  
`\coffin_ht:N` 27942 `\cs_new_eq:NN \coffin_dp:N \box_dp:N`  
`\coffin_ht:c` 27943 `\cs_new_eq:NN \coffin_dp:c \box_dp:c`  
`\coffin_wd:N` 27944 `\cs_new_eq:NN \coffin_ht:N \box_ht:N`  
`\coffin_wd:c` 27945 `\cs_new_eq:NN \coffin_ht:c \box_ht:c`  
27946 `\cs_new_eq:NN \coffin_wd:N \box_wd:N`  
27947 `\cs_new_eq:NN \coffin_wd:c \box_wd:c`

(End definition for `\coffin_dp:N`, `\coffin_ht:N`, and `\coffin_wd:N`. These functions are documented on page 253.)

### 43.4 Coffins: handle and pole management

`\__coffin_get_pole:NnN` A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

27948 `\cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3`  
27949 {  
27950     `\prop_get:cnNF`  
27951         { `coffin ~ \__coffin_to_value:N #1 ~ poles` } {#2} #3  
27952         {  
27953             `\__kernel_msg_error:nnxx { kernel } { unknown-coffin-pole }`  
27954             { `\exp_not:n {#2}` } { `\token_to_str:N #1` }  
27955             `\tl_set:Nn #3 { { Opt } { Opt } { Opt } { Opt } }`  
27956         }  
27957     }  
27958 }

(End definition for `\__coffin_get_pole:NnN`.)

`\__coffin_reset_structure:N` Resetting the structure is a simple copy job.

27958 `\cs_new_protected:Npn \__coffin_reset_structure:N #1`  
27959 {  
27960     `\prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }`  
27961         `\c__coffin_corners_prop`  
27962     `\prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }`  
27963         `\c__coffin_poles_prop`  
27964 }  
27965 `\cs_new_protected:Npn \__coffin_greset_structure:N #1`  
27966 {  
27967     `\prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }`  
27968         `\c__coffin_corners_prop`

```

27969 \prop_gset_eq:cN { coffin ~ __coffin_to_value:N #1 ~ poles }
27970 \c__coffin_poles_prop
27971 }

```

(End definition for `\__coffin_reset_structure:N` and `\__coffin_greset_structure:N`.)

```

\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn
\coffin_gset_horizontal_pole:Nnn
\coffin_gset_horizontal_pole:cnn
__coffin_set_horizontal_pole:NnnN
\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn
\coffin_gset_vertical_pole:Nnn
\coffin_gset_vertical_pole:cnn
__coffin_set_vertical_pole:NnnN
__coffin_set_pole:Nnn
__coffin_set_pole:Nnx

```

```

27972 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
27973 { __coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_put:cnx }
27974 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
27975 \cs_new_protected:Npn \coffin_gset_horizontal_pole:Nnn #1#2#3
27976 { __coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_gput:cnx }
27977 \cs_generate_variant:Nn \coffin_gset_horizontal_pole:Nnn { c }
27978 \cs_new_protected:Npn __coffin_set_horizontal_pole:NnnN #1#2#3#4
27979 {
27980 __coffin_if_exist:NT #1
27981 {
27982 #4 { coffin ~ __coffin_to_value:N #1 ~ poles }
27983 {#2}
27984 {
27985 { Opt } { \dim_eval:n {#3} }
27986 { 1000pt } { Opt }
27987 }
27988 }
27989 }
27990 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
27991 { __coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_put:cnx }
27992 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
27993 \cs_new_protected:Npn \coffin_gset_vertical_pole:Nnn #1#2#3
27994 { __coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_gput:cnx }
27995 \cs_generate_variant:Nn \coffin_gset_vertical_pole:Nnn { c }
27996 \cs_new_protected:Npn __coffin_set_vertical_pole:NnnN #1#2#3#4
27997 {
27998 __coffin_if_exist:NT #1
27999 {
28000 #4 { coffin ~ __coffin_to_value:N #1 ~ poles }
28001 {#2}
28002 {
28003 { \dim_eval:n {#3} } { Opt }
28004 { Opt } { 1000pt }
28005 }
28006 }
28007 }
28008 \cs_new_protected:Npn __coffin_set_pole:Nnn #1#2#3
28009 {
28010 \prop_put:cnn { coffin ~ __coffin_to_value:N #1 ~ poles }
28011 {#2} {#3}
28012 }
28013 \cs_generate_variant:Nn __coffin_set_pole:Nnn { Nnx }

```

(End definition for `\coffin_set_horizontal_pole:Nnn` and others. These functions are documented on page 251.)

```

__coffin_update:N Simple shortcuts.
__coffin_gupdate:N
28014 \cs_new_protected:Npn __coffin_update:N #1
28015 {
28016 __coffin_reset_structure:N #1
28017 __coffin_update_corners:N #1
28018 __coffin_update_poles:N #1
28019 }
28020 \cs_new_protected:Npn __coffin_gupdate:N #1
28021 {
28022 __coffin_greset_structure:N #1
28023 __coffin_gupdate_corners:N #1
28024 __coffin_gupdate_poles:N #1
28025 }

```

(End definition for `\__coffin_update:N` and `\__coffin_gupdate:N`.)

`\__coffin_update_corners:N`  
`\__coffin_gupdate_corners:N`  
`\__coffin_update_corners:NN`  
`\__coffin_update_corners:NNN`

Updating the corners of a coffin is straight-forward as at this stage there can be no rotation. So the corners of the content are just those of the underlying TeX box.

```

28026 \cs_new_protected:Npn __coffin_update_corners:N #1
28027 {
28028 __coffin_update_corners:NN #1 \prop_put:Nnx
28029 __coffin_update_corners:NN #1 \prop_gput:Nnx
28030 \cs_new_protected:Npn __coffin_update_corners:NN #1#2
28031 {
28032 \exp_args:Nc __coffin_update_corners:NNN
28033 { coffin ~ __coffin_to_value:N #1 ~ corners }
28034 #1 #2
28035 }
28036 \cs_new_protected:Npn __coffin_update_corners:NNN #1#2#3
28037 {
28038 #3 #1
28039 { tl }
28040 { { Opt } { \dim_eval:n { \box_ht:N #2 } } }
28041 #3 #1
28042 { tr }
28043 {
28044 \dim_eval:n { \box_wd:N #2 }
28045 \dim_eval:n { \box_ht:N #2 }
28046 }
28047 #3 #1
28048 { bl }
28049 { { Opt } { \dim_eval:n { -\box_dp:N #2 } } }
28050 #3 #1
28051 { br }
28052 {
28053 \dim_eval:n { \box_wd:N #2 }
28054 \dim_eval:n { -\box_dp:N #2 }
28055 }
28056 }

```

(End definition for `\__coffin_update_corners:N` and others.)

`\__coffin_update_poles:N`  
`\__coffin_gupdate_poles:N`  
`\__coffin_update_poles:NN`  
`\__coffin_update_poles:NNN`

This function is called when a coffin is set, and updates the poles to reflect the nature of size of the box. Thus this function only alters poles where the default position is

dependent on the size of the box. It also does not set poles which are relevant only to vertical coffins.

```

28057 \cs_new_protected:Npn __coffin_update_poles:N #1
28058 { __coffin_update_poles:NN #1 \prop_put:Nnx }
28059 \cs_new_protected:Npn __coffin_gupdate_poles:N #1
28060 { __coffin_update_poles:NN #1 \prop_gput:Nnx }
28061 \cs_new_protected:Npn __coffin_update_poles:NN #1#2
28062 {
28063 \exp_args:Nc __coffin_update_poles:NNN
28064 { coffin ~ __coffin_to_value:N #1 ~ poles }
28065 #1 #2
28066 }
28067 \cs_new_protected:Npn __coffin_update_poles:NNN #1#2#3
28068 {
28069 #3 #1 { hc }
28070 {
28071 { \dim_eval:n { 0.5 \box_wd:N #2 } }
28072 { Opt } { Opt } { 1000pt }
28073 }
28074 #3 #1 { r }
28075 {
28076 { \dim_eval:n { \box_wd:N #2 } }
28077 { Opt } { Opt } { 1000pt }
28078 }
28079 #3 #1 { vc }
28080 {
28081 { Opt }
28082 { \dim_eval:n { (\box_ht:N #2 - \box_dp:N #2) / 2 } }
28083 { 1000pt }
28084 { Opt }
28085 }
28086 #3 #1 { t }
28087 {
28088 { Opt }
28089 { \dim_eval:n { \box_ht:N #2 } }
28090 { 1000pt }
28091 { Opt }
28092 }
28093 #3 #1 { b }
28094 {
28095 { Opt }
28096 { \dim_eval:n { -\box_dp:N #2 } }
28097 { 1000pt }
28098 { Opt }
28099 }
28100 }
```

(End definition for `\__coffin_update_poles:N` and others.)

### 43.5 Coffins: calculation of pole intersections

The lead off in finding intersections is to recover the two poles and then hand off to the auxiliary for the actual calculation. There may of course not be an intersection, for which an error trap is needed.

```

28101 \cs_new_protected:Npn __coffin_calculate_intersection:Nnn #1#2#3
28102 {
28103 __coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
28104 __coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
28105 \bool_set_false:N \l__coffin_error_bool
28106 \exp_last_two_unbraced:Noo
28107 __coffin_calculate_intersection:nnnnnnnn
28108 \l__coffin_pole_a_tl \l__coffin_pole_b_tl
28109 \bool_if:NT \l__coffin_error_bool
28110 {
28111 __kernel_msg_error:nn { kernel } { no-pole-intersection }
28112 \dim_zero:N \l__coffin_x_dim
28113 \dim_zero:N \l__coffin_y_dim
28114 }
28115 }

```

The two poles passed here each have four values (as dimensions),  $(a, b, c, d)$  and  $(a', b', c', d')$ . These are arguments 1–4 and 5–8, respectively. In both cases  $a$  and  $b$  are the co-ordinates of a point on the pole and  $c$  and  $d$  define the direction of the pole. Finding the intersection depends on the directions of the poles, which are given by  $d/c$  and  $d'/c'$ . However, if one of the poles is either horizontal or vertical then one or more of  $c$ ,  $d$ ,  $c'$  and  $d'$  are zero and a special case is needed.

```

28116 \cs_new_protected:Npn __coffin_calculate_intersection:nnnnnnnn
28117 #1#2#3#4#5#6#7#8
28118 {
28119 \dim_compare:nNnTF {#3} = \c_zero_dim

```

The case where the first pole is vertical. So the  $x$ -component of the interaction is at  $a$ . There is then a test on the second pole: if it is also vertical then there is an error.

```

28120 {
28121 \dim_set:Nn \l__coffin_x_dim {#1}
28122 \dim_compare:nNnTF {#7} = \c_zero_dim
28123 { \bool_set_true:N \l__coffin_error_bool }

```

The second pole may still be horizontal, in which case the  $y$ -component of the intersection is  $b'$ . If not,

$$y = \frac{d'}{c'} (a - a') + b'$$

with the  $x$ -component already known to be #1.

```

28124 {
28125 \dim_set:Nn \l__coffin_y_dim
28126 {
28127 \dim_compare:nNnTF {#8} = \c_zero_dim
28128 {#6}
28129 {
28130 \fp_to_dim:n
28131 {
28132 (\dim_to_fp:n {#8} / \dim_to_fp:n {#7})
28133 * (\dim_to_fp:n {#1} - \dim_to_fp:n {#5})
28134 + \dim_to_fp:n {#6}
28135 }
28136 }
28137 }
28138 }
28139 }

```

If the first pole is not vertical then it may be horizontal. If so, then the procedure is essentially the same as that already done but with the  $x$ - and  $y$ -components interchanged.

```

28140 {
28141 \dim_compare:nNnTF {#4} = \c_zero_dim
28142 {
28143 \dim_set:Nn \l__coffin_y_dim {#2}
28144 \dim_compare:nNnTF {#8} = { \c_zero_dim }
28145 { \bool_set_true:N \l__coffin_error_bool }
28146 {

```

Now we deal with the case where the second pole may be vertical, or if not we have

$$x = \frac{c'}{d'} (b - b') + a'$$

which is again handled by the same auxiliary.

```

28147 \dim_set:Nn \l__coffin_x_dim
28148 {
28149 \dim_compare:nNnTF {#7} = \c_zero_dim
28150 {#5}
28151 {
28152 \fp_to_dim:n
28153 {
28154 (\dim_to_fp:n {#7} / \dim_to_fp:n {#8})
28155 * (\dim_to_fp:n {#4} - \dim_to_fp:n {#6})
28156 + \dim_to_fp:n {#5}
28157 }
28158 }
28159 }
28160 }
28161 }

```

The first pole is neither horizontal nor vertical. To avoid even more complexity, we now work out both slopes and pass to an auxiliary.

```

28162 {
28163 \use:x
28164 {
28165 __coffin_calculate_intersection:nnnnnn
28166 { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
28167 { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
28168 }
28169 {#1} {#2} {#5} {#6}
28170 }
28171 }
28172 }

```

Assuming the two poles are not parallel, then the intersection point is found in two steps. First we find the  $x$ -value with

$$x = \frac{sa - s'a' - b + b'}{s - s'}$$

and then finding the  $y$ -value with

$$y = s(x - a) + b$$

```

28173 \cs_set_protected:Npn __coffin_calculate_intersection:nnnnnn #1#2#3#4#5#6
28174 {
28175 \fp_compare:nNnTF {#1} = {#2}
28176 {
28177 \bool_set_true:N \l__coffin_error_bool
28178 \dim_set:Nn \l__coffin_x_dim
28179 {
28180 \fp_to_dim:n
28181 {
28182 (
28183 #1 * \dim_to_fp:n {#3}
28184 - #2 * \dim_to_fp:n {#5}
28185 - \dim_to_fp:n {#4}
28186 + \dim_to_fp:n {#6}
28187)
28188 /
28189 (#1 - #2)
28190 }
28191 }
28192 \dim_set:Nn \l__coffin_y_dim
28193 {
28194 \fp_to_dim:n
28195 {
28196 #1 * (\l__coffin_x_dim - \dim_to_fp:n {#3})
28197 + \dim_to_fp:n {#4}
28198 }
28199 }
28200 }
28201 }
```

(End definition for `\__coffin_calculate_intersection:Nnn`, `\__coffin_calculate_intersection:nnnnnnn`, and `\__coffin_calculate_intersection:nnnnnn`.)

### 43.6 Affine transformations

`\l__coffin_sin_fp` Used for rotations to get the sine and cosine values.

`\l__coffin_cos_fp`

```

28202 \fp_new:N \l__coffin_sin_fp
28203 \fp_new:N \l__coffin_cos_fp
```

(End definition for `\l__coffin_sin_fp` and `\l__coffin_cos_fp`.)

`\l__coffin_bounding_prop` A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```
28204 \prop_new:N \l__coffin_bounding_prop
```

(End definition for `\l__coffin_bounding_prop`.)

`\l__coffin_corners_prop` Used to avoid needing to track scope for intermediate steps.

`\l__coffin_poles_prop`

```

28205 \prop_new:N \l__coffin_corners_prop
28206 \prop_new:N \l__coffin_poles_prop
```

(End definition for `\l__coffin_corners_prop` and `\l__coffin_poles_prop`.)

`\l__coffin_bounding_shift_dim` The shift of the bounding box of a coffin from the real content.

```
28207 \dim_new:N \l__coffin_bounding_shift_dim
```

(End definition for `\l_coffin_bounding_shift_dim`.)

```
\l_coffin_left_corner_dim
\l_coffin_right_corner_dim
\l_coffin_bottom_corner_dim
\l_coffin_top_corner_dim
28208 \dim_new:N \l_coffin_left_corner_dim
28209 \dim_new:N \l_coffin_right_corner_dim
28210 \dim_new:N \l_coffin_bottom_corner_dim
28211 \dim_new:N \l_coffin_top_corner_dim

(End definition for \l_coffin_left_corner_dim and others.)

\coffin_rotate:Nn
\coffin_rotate:cn
\coffin_grotate:Nn
\coffin_grotate:cn
_coffin_rotate:NnNNN
28212 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
28213 { _coffin_rotate:NnNNN #1 {#2} \box_rotate:Nn \prop_set_eq:cN \hbox_set:Nn }
28214 \cs_generate_variant:Nn \coffin_rotate:Nn { c }
28215 \cs_new_protected:Npn \coffin_grotate:Nn #1#2
28216 { _coffin_rotate:NnNNN #1 {#2} \box_grotate:Nn \prop_gset_eq:cN \hbox_gset:Nn }
28217 \cs_generate_variant:Nn \coffin_grotate:Nn { c }
28218 \cs_new_protected:Npn _coffin_rotate:NnNNN #1#2#3#4#5
28219 {
28220 \fp_set:Nn \l_coffin_sin_fp { sind (#2) }
28221 \fp_set:Nn \l_coffin_cos_fp { cosd (#2) }
28222 \prop_set_eq:Nc \l_coffin_corners_prop
28223 { coffin ~ _coffin_to_value:N #1 ~ corners }
28224 \prop_set_eq:Nc \l_coffin_poles_prop
28225 { coffin ~ _coffin_to_value:N #1 ~ poles }
```

Use a local copy of the property lists to avoid needing to pass the name and scope around.

```
28222 \prop_set_eq:Nc \l_coffin_corners_prop
28223 { coffin ~ _coffin_to_value:N #1 ~ corners }
28224 \prop_set_eq:Nc \l_coffin_poles_prop
28225 { coffin ~ _coffin_to_value:N #1 ~ poles }
28226 \prop_map_inline:Nn \l_coffin_corners_prop
28227 { _coffin_rotate_corner:Nnnn #1 {##1} ##2 }
28228 \prop_map_inline:Nn \l_coffin_poles_prop
28229 { _coffin_rotate_pole:Nnnnnn #1 {##1} ##2 }
28230 _coffin_set_bounding:N #1
28231 \prop_map_inline:Nn \l_coffin_bounding_prop
28232 { _coffin_rotate_bounding:nnn {##1} ##2 }
```

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

```
28230 _coffin_set_bounding:N #1
28231 \prop_map_inline:Nn \l_coffin_bounding_prop
28232 { _coffin_rotate_bounding:nnn {##1} ##2 }
```

At this stage, there needs to be a calculation to find where the corners of the content and the box itself will end up.

```
28233 _coffin_find_corner_maxima:N #1
28234 _coffin_find_bounding_shift:
28235 #3 #1 {#2}
```

The correction of the box position itself takes place here. The idea is that the bounding box for a coffin is tight up to the content, and has the reference point at the bottom-left. The  $x$ -direction is handled by moving the content by the difference in the positions of the bounding box and the content left edge. The  $y$ -direction is dealt with by moving the box down by any depth it has acquired. The internal box is used here to allow for the next step.

```

28236 \hbox_set:Nn \l__coffin_internal_box
28237 {
28238 \tex_kern:D
28239 \dim_eval:n
28240 { \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim }
28241 \exp_stop_f:
28242 \box_move_down:nn { \l__coffin_bottom_corner_dim }
28243 { \box_use:N #1 }
28244 }

```

If there have been any previous rotations then the size of the bounding box will be bigger than the contents. This can be corrected easily by setting the size of the box to the height and width of the content. As this operation requires setting box dimensions and these transcend grouping, the safe way to do this is to use the internal box and to reset the result into the target box.

```

28245 \box_set_ht:Nn \l__coffin_internal_box
28246 { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
28247 \box_set_dp:Nn \l__coffin_internal_box { 0pt }
28248 \box_set_wd:Nn \l__coffin_internal_box
28249 { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
28250 #5 #1 { \box_use_drop:N \l__coffin_internal_box }

```

The final task is to move the poles and corners such that they are back in alignment with the box reference point.

```

28251 \prop_map_inline:Nn \l__coffin_corners_prop
28252 { __coffin_shift_corner:Nnnn #1 {##1} ##2 }
28253 \prop_map_inline:Nn \l__coffin_poles_prop
28254 { __coffin_shift_pole:Nnnnnn #1 {##1} ##2 }

```

Update the coffin data.

```

28255 #4 { coffin ~ __coffin_to_value:N #1 ~ corners }
28256 \l__coffin_corners_prop
28257 #4 { coffin ~ __coffin_to_value:N #1 ~ poles }
28258 \l__coffin_poles_prop
28259 }

```

(End definition for `\coffin_rotate:Nn`, `\coffin_grotate:Nn`, and `\__coffin_rotate:NnNNN`. These functions are documented on page 252.)

`\__coffin_set_bounding:N`

The bounding box corners for a coffin are easy enough to find: this is the same code as for the corners of the material itself, but using a dedicated property list.

```

28260 \cs_new_protected:Npn __coffin_set_bounding:N #1
28261 {
28262 \prop_put:Nnx \l__coffin_bounding_prop { tl }
28263 { { Opt } { \dim_eval:n { \box_ht:N #1 } } }
28264 \prop_put:Nnx \l__coffin_bounding_prop { tr }
28265 {
28266 { \dim_eval:n { \box_wd:N #1 } }

```

```

28267 { \dim_eval:n { \box_ht:N #1 } }
28268 }
28269 \dim_set:Nn \l__coffin_internal_dim { -\box_dp:N #1 }
28270 \prop_put:Nnx \l__coffin_bounding_prop { bl }
28271 { { Opt } { \dim_use:N \l__coffin_internal_dim } }
28272 \prop_put:Nnx \l__coffin_bounding_prop { br }
28273 {
28274 { \dim_eval:n { \box_wd:N #1 } }
28275 { \dim_use:N \l__coffin_internal_dim }
28276 }
28277 }

```

(End definition for `\__coffin_set_bounding:N`.)

`\__coffin_rotate_bounding:nnn`  
`\__coffin_rotate_corner:Nnnn`

Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

```

28278 \cs_new_protected:Npn __coffin_rotate_bounding:nnn #1#2#3
28279 {
28280 __coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
28281 \prop_put:Nnx \l__coffin_bounding_prop {#1}
28282 { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
28283 }
28284 \cs_new_protected:Npn __coffin_rotate_corner:Nnnn #1#2#3#4
28285 {
28286 __coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
28287 \prop_put:Nnx \l__coffin_corners_prop {#2}
28288 { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
28289 }

```

(End definition for `\__coffin_rotate_bounding:nnn` and `\__coffin_rotate_corner:Nnnn`.)

`\__coffin_rotate_pole:Nnnnnn`

Rotating a single pole simply means shifting the co-ordinate of the pole and its direction. The rotation here is about the bottom-left corner of the coffin.

```

28290 \cs_new_protected:Npn __coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
28291 {
28292 __coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
28293 __coffin_rotate_vector:nnNN {#5} {#6}
28294 \l__coffin_x_prime_dim \l__coffin_y_prime_dim
28295 \prop_put:Nnx \l__coffin_poles_prop {#2}
28296 {
28297 { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
28298 { \dim_use:N \l__coffin_x_prime_dim }
28299 { \dim_use:N \l__coffin_y_prime_dim }
28300 }
28301 }

```

(End definition for `\__coffin_rotate_pole:Nnnnnn`.)

`\__coffin_rotate_vector:nnNN`

A rotation function, which needs only an input vector (as dimensions) and an output space. The values `\l__coffin_cos_fp` and `\l__coffin_sin_fp` should previously have been set up correctly. Working this way means that the floating point work is kept to a minimum: for any given rotation the sin and cosine values do no change, after all.

```

28302 \cs_new_protected:Npn __coffin_rotate_vector:nnNN #1#2#3#4

```

```

28303 {
28304 \dim_set:Nn #3
28305 {
28306 \fp_to_dim:n
28307 {
28308 \dim_to_fp:n {#1} * \l__coffin_cos_fp
28309 - \dim_to_fp:n {#2} * \l__coffin_sin_fp
28310 }
28311 }
28312 \dim_set:Nn #4
28313 {
28314 \fp_to_dim:n
28315 {
28316 \dim_to_fp:n {#1} * \l__coffin_sin_fp
28317 + \dim_to_fp:n {#2} * \l__coffin_cos_fp
28318 }
28319 }
28320 }

```

(End definition for `\__coffin_rotate_vector:nnNN`.)

`\__coffin_find_corner_maxima:N`  
`\__coffin_find_corner_maxima_aux:nn`

The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

```

28321 \cs_new_protected:Npn __coffin_find_corner_maxima:N #1
28322 {
28323 \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
28324 \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
28325 \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
28326 \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
28327 \prop_map_inline:Nn \l__coffin_corners_prop
28328 { __coffin_find_corner_maxima_aux:nn ##2 }
28329 }
28330 \cs_new_protected:Npn __coffin_find_corner_maxima_aux:nn #1#2
28331 {
28332 \dim_set:Nn \l__coffin_left_corner_dim
28333 { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
28334 \dim_set:Nn \l__coffin_right_corner_dim
28335 { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
28336 \dim_set:Nn \l__coffin_bottom_corner_dim
28337 { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
28338 \dim_set:Nn \l__coffin_top_corner_dim
28339 { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
28340 }

```

(End definition for `\__coffin_find_corner_maxima:N` and `\__coffin_find_corner_maxima_aux:nn`.)

`\__coffin_find_bounding_shift:`  
`\__coffin_find_bounding_shift_aux:nn`

The approach to finding the shift for the bounding box is similar to that for the corners. However, there is only one value needed here and a fixed input property list, so things are a bit clearer.

```

28341 \cs_new_protected:Npn __coffin_find_bounding_shift:
28342 {
28343 \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }

```

```

28344 \prop_map_inline:Nn \l__coffin_bounding_prop
28345 { __coffin_find_bounding_shift_aux:nn ##2 }
28346 }
28347 \cs_new_protected:Npn __coffin_find_bounding_shift_aux:nn #1#2
28348 {
28349 \dim_set:Nn \l__coffin_bounding_shift_dim
28350 { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
28351 }

```

(End definition for `\__coffin_find_bounding_shift:` and `\__coffin_find_bounding_shift_aux:nn`.)

`\__coffin_shift_corner:Nnnn`  
`\__coffin_shift_pole:Nnnnnn`

Shifting the corners and poles of a coffin means subtracting the appropriate values from the  $x$ - and  $y$ -components. For the poles, this means that the direction vector is unchanged.

```

28352 \cs_new_protected:Npn __coffin_shift_corner:Nnnn #1#2#3#4
28353 {
28354 \prop_put:Nnx \l__coffin_corners_prop {#2}
28355 {
28356 { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
28357 { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
28358 }
28359 }
28360 \cs_new_protected:Npn __coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
28361 {
28362 \prop_put:Nnx \l__coffin_poles_prop {#2}
28363 {
28364 { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
28365 { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
28366 {#5} {#6}
28367 }
28368 }

```

(End definition for `\__coffin_shift_corner:Nnnn` and `\__coffin_shift_pole:Nnnnnn`.)

`\l__coffin_scale_x_fp`  
`\l__coffin_scale_y_fp`

```

28369 \fp_new:N \l__coffin_scale_x_fp
28370 \fp_new:N \l__coffin_scale_y_fp

```

(End definition for `\l__coffin_scale_x_fp` and `\l__coffin_scale_y_fp`.)

`\l__coffin_scaled_total_height_dim`  
`\l__coffin_scaled_width_dim`

```

28371 \dim_new:N \l__coffin_scaled_total_height_dim
28372 \dim_new:N \l__coffin_scaled_width_dim

```

(End definition for `\l__coffin_scaled_total_height_dim` and `\l__coffin_scaled_width_dim`.)

`\coffin_resize:Nnn`  
`\coffin_resize:cnn`  
`\coffin_gresize:Nnn`  
`\coffin_gresize:cnn`

`\__coffin_resize:NnnNN`

```

28373 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
28374 {
28375 __coffin_resize:NnnNN #1 {#2} {#3}
28376 \box_resize_to_wd_and_ht_plus_dp:Nnn

```

```

28377 \prop_set_eq:cN
28378 }
28379 \cs_generate_variant:Nn \coffin_resize:Nnn { c }
28380 \cs_new_protected:Npn \coffin_gresize:Nnn #1#2#3
28381 {
28382 __coffin_resize:NnnNN #1 {#2} {#3}
28383 \box_gresize_to_wd_and_ht_plus_dp:Nnn
28384 \prop_gset_eq:cN
28385 }
28386 \cs_generate_variant:Nn \coffin_gresize:Nnn { c }
28387 \cs_new_protected:Npn __coffin_resize:NnnNN #1#2#3#4#5
28388 {
28389 \fp_set:Nn \l__coffin_scale_x_fp
28390 { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
28391 \fp_set:Nn \l__coffin_scale_y_fp
28392 {
28393 \dim_to_fp:n {#3}
28394 / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
28395 }
28396 #4 #1 {#2} {#3}
28397 __coffin_resize_common:NnnN #1 {#2} {#3} #5
28398 }

```

(End definition for `\coffin_resize:Nnn`, `\coffin_gresize:Nnn`, and `\__coffin_resize:NnnNN`. These functions are documented on page 252.)

`\__coffin_resize_common:NnnN` The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```

28399 \cs_new_protected:Npn __coffin_resize_common:NnnN #1#2#3#4
28400 {
28401 \prop_set_eq:Nc \l__coffin_corners_prop
28402 { coffin ~ __coffin_to_value:N #1 ~ corners }
28403 \prop_set_eq:Nc \l__coffin_poles_prop
28404 { coffin ~ __coffin_to_value:N #1 ~ poles }
28405 \prop_map_inline:Nn \l__coffin_corners_prop
28406 { __coffin_scale_corner:Nnnn #1 {##1} ##2 }
28407 \prop_map_inline:Nn \l__coffin_poles_prop
28408 { __coffin_scale_pole:Nnnnnn #1 {##1} ##2 }

```

Negative  $x$ -scaling values place the poles in the wrong location: this is corrected here.

```

28409 \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
28410 {
28411 \prop_map_inline:Nn \l__coffin_corners_prop
28412 { __coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
28413 \prop_map_inline:Nn \l__coffin_poles_prop
28414 { __coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
28415 }
28416 #4 { coffin ~ __coffin_to_value:N #1 ~ corners }
28417 \l__coffin_corners_prop
28418 #4 { coffin ~ __coffin_to_value:N #1 ~ poles }
28419 \l__coffin_poles_prop
28420 }

```

(End definition for `\__coffin_resize_common:NnnN`.)

```
\coffin_scale:Nnn
\coffin_scale:cnn
\coffin_gscale:Nnn
\coffin_gscale:cnn
```

```
\coffin_scale:NnnNN
```

For scaling, the opposite calculation is done to find the new dimensions for the coffin. Only the total height is needed, as this is the shift required for corners and poles. The scaling is done the TeX way as this works properly with floating point values without needing to use the fp module.

```
28421 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
28422 { __coffin_scale:NnnNN #1 {#2} {#3} \box_scale:Nnn \prop_set_eq:cN }
28423 \cs_generate_variant:Nn \coffin_scale:Nnn { c }
28424 \cs_new_protected:Npn \coffin_gscale:Nnn #1#2#3
28425 { __coffin_scale:NnnNN #1 {#2} {#3} \box_gscale:Nnn \prop_gset_eq:cN }
28426 \cs_generate_variant:Nn \coffin_gscale:Nnn { c }
28427 \cs_new_protected:Npn __coffin_scale:NnnNN #1#2#3#4#5
28428 {
28429 \fp_set:Nn \l__coffin_scale_x_fp {#2}
28430 \fp_set:Nn \l__coffin_scale_y_fp {#3}
28431 #4 #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
28432 \dim_set:Nn \l__coffin_internal_dim
28433 { \coffin_ht:N #1 + \coffin_dp:N #1 }
28434 \dim_set:Nn \l__coffin_scaled_total_height_dim
28435 { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
28436 \dim_set:Nn \l__coffin_scaled_width_dim
28437 { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
28438 __coffin_resize_common:NnnN #1
28439 { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
28440 #5
28441 }
```

(End definition for \coffin\_scale:Nnn, \coffin\_gscale:Nnn, and \coffin\_scale:NnnNN. These functions are documented on page 252.)

```
__coffin_scale_vector:nnNN
```

This function scales a vector from the origin using the pre-set scale factors in  $x$  and  $y$ . This is a much less complex operation than rotation, and as a result the code is a lot clearer.

```
28442 \cs_new_protected:Npn __coffin_scale_vector:nnNN #1#2#3#4
28443 {
28444 \dim_set:Nn #3
28445 { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
28446 \dim_set:Nn #4
28447 { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
28448 }
```

(End definition for \\_\_coffin\_scale\_vector:nnNN.)

```
__coffin_scale_corner:Nnnn
__coffin_scale_pole:Nnnnnn
```

Scaling both corners and poles is a simple calculation using the preceding vector scaling.

```
28449 \cs_new_protected:Npn __coffin_scale_corner:Nnnn #1#2#3#4
28450 {
28451 __coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
28452 \prop_put:Nnx \l__coffin_corners_prop {#2}
28453 { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
28454 }
28455 \cs_new_protected:Npn __coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
28456 {
28457 __coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
28458 \prop_put:Nnx \l__coffin_poles_prop {#2}
28459 { }
```

```

28460 { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
28461 {#5} {#6}
28462 }
28463 }

(End definition for __coffin_scale_corner:Nnnn and __coffin_scale_pole:Nnnnnn.)
```

`\__coffin_x_shift_corner:Nnnn`  
`\__coffin_x_shift_pole:Nnnnnn`

These functions correct for the *x* displacement that takes place with a negative horizontal scaling.

```

28464 \cs_new_protected:Npn __coffin_x_shift_corner:Nnnn #1#2#3#4
28465 {
28466 \prop_put:Nnx \l__coffin_corners_prop {#2}
28467 {
28468 { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
28469 }
28470 }
28471 \cs_new_protected:Npn __coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
28472 {
28473 \prop_put:Nnx \l__coffin_poles_prop {#2}
28474 {
28475 { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
28476 {#5} {#6}
28477 }
28478 }
```

(End definition for \\_\_coffin\_x\_shift\_corner:Nnnn and \\_\_coffin\_x\_shift\_pole:Nnnnnn.)

### 43.7 Aligning and typesetting of coffins

`\coffin_join:NnnNnnn`  
`\coffin_join:cnnNnnn`  
`\coffin_join:NnnCnnn`  
`\coffin_join:cnnCnnn`

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which has all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```

28479 \cs_new_protected:Npn \coffin_join:NnnNnnn #1#2#3#4#5#6#7#8
28480 {
28481 __coffin_join:NnnNnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
28482 \coffin_set_eq:NN
28483 }
28484 \cs_generate_variant:Nn \coffin_join:NnnNnnn { c , Nnnc , cnnc }
28485 \cs_new_protected:Npn \coffin_gjoin:NnnNnnn #1#2#3#4#5#6#7#8
28486 {
28487 __coffin_join:NnnNnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
28488 \coffin_gset_eq:NN
28489 }
28490 \cs_generate_variant:Nn \coffin_gjoin:NnnNnnn { c , Nnnc , cnnc }
28491 \cs_new_protected:Npn __coffin_join:NnnNnnnN #1#2#3#4#5#6#7#8#9
28492 {
28493 __coffin_align:NnnNnnnN
28494 #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
```

Correct the placement of the reference point. If the *x*-offset is negative then the reference point of the second box is to the left of that of the first, which is corrected using a kern. On the right side the first box might stick out, which would show up if it is wider than the sum of the *x*-offset and the width of the second box. So a second kern may be needed.

```

28495 \hbox_set:Nn \l__coffin_aligned_coffin
28496 {
28497 \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
28498 { \tex_kern:D -\l__coffin_offset_x_dim }
28499 \hbox_unpack:N \l__coffin_aligned_coffin
28500 \dim_set:Nn \l__coffin_internal_dim
28501 { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
28502 \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
28503 { \tex_kern:D -\l__coffin_internal_dim }
28504 }

```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

28505 __coffin_reset_structure:N \l__coffin_aligned_coffin
28506 \prop_clear:c
28507 {
28508 \coffin ~ __coffin_to_value:N \l__coffin_aligned_coffin
28509 \c_space_tl corners
28510 }
28511 __coffin_update_poles:N \l__coffin_aligned_coffin

```

The structures of the parent coffins are now transferred to the new coffin, which requires that the appropriate offsets are applied. That then depends on whether any shift was needed.

```

28512 \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
28513 {
28514 __coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
28515 __coffin_offset_poles:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
28516 __coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
28517 __coffin_offset_corners:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
28518 }
28519 {
28520 __coffin_offset_poles:Nnn #1 { Opt } { Opt }
28521 __coffin_offset_poles:Nnn #4
28522 { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
28523 __coffin_offset_corners:Nnn #1 { Opt } { Opt }
28524 __coffin_offset_corners:Nnn #4
28525 { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
28526 }
28527 __coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
28528 #9 #1 \l__coffin_aligned_coffin
28529 }

```

(End definition for `\coffin_join:NnnNnnnn`, `\coffin_gjoin:NnnNnnnn`, and `\__coffin_join:NnnNnnnnN`. These functions are documented on page 252.)

`\coffin_attach:NnnNnnnn`

`\coffin_attach:cnnNnnnn`

`\coffin_attach:NnnCnnnn`

`\coffin_attach:cnnCnnnn`

`\coffin_gattach:NnnNnnnn`

`\coffin_gattach:cnnNnnnn`

`\coffin_gattach:NnnCnnnn`

`\coffin_gattach:cnnCnnnn`

`\__coffin_attach:NnnNnnnnN`

`\__coffin_attach_mark:NnnNnnnn`

A more simple version of the above, as it simply uses the size of the first coffin for the new one. This means that the work here is rather simplified compared to the above code. The function used when marking a position is hear also as it is similar but without the structure updates.

```

28530 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
28531 {
28532 __coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
28533 \coffin_set_eq:NN
28534 }

```

```

28535 \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }
28536 \cs_new_protected:Npn \coffin_gattach:NnnNnnnn #1#2#3#4#5#6#7#8
28537 {
28538 __coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
28539 \coffin_gset_eq:NN
28540 }
28541 \cs_generate_variant:Nn \coffin_gattach:NnnNnnnn { c , Nnnc , cnnc }
28542 \cs_new_protected:Npn __coffin_attach:NnnNnnnnN #1#2#3#4#5#6#7#8#9
28543 {
28544 __coffin_align:NnnNnnnnN
28545 #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
28546 \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
28547 \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
28548 \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
28549 __coffin_reset_structure:N \l__coffin_aligned_coffin
28550 \prop_set_eq:cc
28551 {
28552 coffin ~ __coffin_to_value:N \l__coffin_aligned_coffin
28553 \c_space_tl corners
28554 }
28555 { coffin ~ __coffin_to_value:N #1 ~ corners }
28556 __coffin_update_poles:N \l__coffin_aligned_coffin
28557 __coffin_offset_poles:Nnn #1 { Opt } { Opt }
28558 __coffin_offset_poles:Nnn #4
28559 { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
28560 __coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
28561 \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
28562 }
28563 \cs_new_protected:Npn __coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
28564 {
28565 __coffin_align:NnnNnnnnN
28566 #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
28567 \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
28568 \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
28569 \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
28570 \box_set_eq:NN #1 \l__coffin_aligned_coffin
28571 }

```

(End definition for `\coffin_attach:NnnNnnnn` and others. These functions are documented on page 252.)

`\__coffin_align:NnnNnnnnN`

The internal function aligns the two coffins into a third one, but performs no corrections on the resulting coffin poles. The process begins by finding the points of intersection for the poles for each of the input coffins. Those for the first coffin are worked out after those for the second coffin, as this allows the ‘primed’ storage area to be used for the second coffin. The ‘real’ box offsets are then calculated, before using these to re-box the input coffins. The default poles are then set up, but the final result depends on how the bounding box is being handled.

```

28572 \cs_new_protected:Npn __coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
28573 {
28574 __coffin_calculate_intersection:Nnn #4 {#5} {#6}
28575 \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
28576 \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
28577 __coffin_calculate_intersection:Nnn #1 {#2} {#3}

```

```

28578 \dim_set:Nn \l__coffin_offset_x_dim
28579 { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
28580 \dim_set:Nn \l__coffin_offset_y_dim
28581 { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
28582 \hbox_set:Nn \l__coffin_aligned_internal_coffin
28583 {
28584 \box_use:N #1
28585 \tex_kern:D -\box_wd:N #1
28586 \tex_kern:D \l__coffin_offset_x_dim
28587 \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
28588 }
28589 \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
28590 }

```

(End definition for `\__coffin_align:NnnNnnnnN`.)

```
__coffin_offset_poles:Nnn
 __coffin_offset_pole:Nnnnnn
```

Transferring structures from one coffin to another requires that the positions are updated by the offset between the two coffins. This is done by mapping to the property list of the source coffins, moving as appropriate and saving to the new coffin data structures. The test for a `-` means that the structures from the parent coffins are uniquely labelled and do not depend on the order of alignment. The pay off for this is that `-` should not be used in coffin pole or handle names, and that multiple alignments do not result in a whole set of values.

```

28591 \cs_new_protected:Npn __coffin_offset_poles:Nnn #1#2#3
28592 {
28593 \prop_map_inline:cn { coffin ~ __coffin_to_value:N #1 ~ poles }
28594 { __coffin_offset_pole:Nnnnnn #1 {##1} ##2 {#2} {#3} }
28595 }
28596 \cs_new_protected:Npn __coffin_offset_pole:Nnnnnn #1#2#3#4#5#6#7#8
28597 {
28598 \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
28599 \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
28600 \tl_if_in:nTF {#2} { - }
28601 { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
28602 { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
28603 \exp_last_unbraced:NNo __coffin_set_pole:Nnx \l__coffin_aligned_coffin
28604 { \l__coffin_internal_tl }
28605 {
28606 { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
28607 {#5} {#6}
28608 }
28609 }

```

(End definition for `\__coffin_offset_poles:Nnn` and `\__coffin_offset_pole:Nnnnnn`.)

```
__coffin_offset_corners:Nnn
 __coffin_offset_corner:Nnnnn
```

Saving the offset corners of a coffin is very similar, except that there is no need to worry about naming: every corner can be saved here as order is unimportant.

```

28610 \cs_new_protected:Npn __coffin_offset_corners:Nnn #1#2#3
28611 {
28612 \prop_map_inline:cn { coffin ~ __coffin_to_value:N #1 ~ corners }
28613 { __coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
28614 }
28615 \cs_new_protected:Npn __coffin_offset_corner:Nnnnn #1#2#3#4#5#6
28616 {

```

```

28617 \prop_put:cnx
28618 {
28619 coffin ~ __coffin_to_value:N \l__coffin_aligned_coffin
28620 \c_space_tl corners
28621 }
28622 { #1 - #2 }
28623 {
28624 { \dim_eval:n { #3 + #5 } }
28625 { \dim_eval:n { #4 + #6 } }
28626 }
28627 }

```

(End definition for `\__coffin_offset_corners:Nnn` and `\__coffin_offset_corner:Nnnn`.)

The T and B poles need to be recalculated after alignment. These functions find the larger absolute value for the poles, but this is of course only logical when the poles are horizontal.

```

28628 \cs_new_protected:Npn __coffin_update_vertical_poles:NNN #1#2#3
28629 {
28630 __coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
28631 __coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
28632 \exp_last_two_unbraced:Noo __coffin_update_T:nnnnnnnnN
28633 \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
28634 __coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
28635 __coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
28636 \exp_last_two_unbraced:Noo __coffin_update_B:nnnnnnnnN
28637 \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
28638 }
28639 \cs_new_protected:Npn __coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
28640 {
28641 \dim_compare:nNnTF {#2} < {#6}
28642 {
28643 __coffin_set_pole:Nnx #9 { T }
28644 { { Opt } {#6} { 1000pt } { Opt } }
28645 }
28646 {
28647 __coffin_set_pole:Nnx #9 { T }
28648 { { Opt } {#2} { 1000pt } { Opt } }
28649 }
28650 }
28651 \cs_new_protected:Npn __coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
28652 {
28653 \dim_compare:nNnTF {#2} < {#6}
28654 {
28655 __coffin_set_pole:Nnx #9 { B }
28656 { { Opt } {#2} { 1000pt } { Opt } }
28657 }
28658 {
28659 __coffin_set_pole:Nnx #9 { B }
28660 { { Opt } {#6} { 1000pt } { Opt } }
28661 }
28662 }

```

(End definition for `\__coffin_update_vertical_poles:NNN`, `\__coffin_update_T:nnnnnnnnN`, and `\__coffin_update_B:nnnnnnnnN`.)

\c\_coffin\_empty\_coffin An empty-but-horizontal coffin.

```
28663 \coffin_new:N \c_coffin_empty_coffin
28664 \tex_setbox:D \c_coffin_empty_coffin = \tex_hbox:D { }
```

(End definition for \c\_coffin\_empty\_coffin.)

\coffin\_typeset:Nnnnn

\coffin\_typeset:cnnnn

Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

```
28665 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
28666 {
28667 \mode_leave_vertical:
28668 __coffin_align:NnnNnnnnN \c_coffin_empty_coffin { H } { 1 }
28669 #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
28670 \box_use_drop:N \l__coffin_aligned_coffin
28671 }
28672 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }
```

(End definition for \coffin\_typeset:Nnnnn. This function is documented on page 253.)

## 43.8 Coffin diagnostics

\l\_coffin\_display\_coffin Used for printing coffins with data structures attached.

```
28673 \coffin_new:N \l_coffin_display_coffin
28674 \coffin_new:N \l_coffin_display_coord_coffin
28675 \coffin_new:N \l_coffin_display_pole_coffin
```

(End definition for \l\_coffin\_display\_coffin, \l\_coffin\_display\_coord\_coffin, and \l\_coffin\_display\_pole\_coffin.)

\l\_coffin\_display\_handles\_prop

This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```
28676 \prop_new:N \l_coffin_display_handles_prop
28677 \prop_put:Nnn \l_coffin_display_handles_prop { tl }
28678 { { b } { r } { -1 } { 1 } }
28679 \prop_put:Nnn \l_coffin_display_handles_prop { thc }
28680 { { b } { hc } { 0 } { 1 } }
28681 \prop_put:Nnn \l_coffin_display_handles_prop { tr }
28682 { { b } { 1 } { 1 } { 1 } }
28683 \prop_put:Nnn \l_coffin_display_handles_prop { vcl }
28684 { { vc } { r } { -1 } { 0 } }
28685 \prop_put:Nnn \l_coffin_display_handles_prop { vchc }
28686 { { vc } { hc } { 0 } { 0 } }
28687 \prop_put:Nnn \l_coffin_display_handles_prop { vcr }
28688 { { vc } { 1 } { 1 } { 0 } }
28689 \prop_put:Nnn \l_coffin_display_handles_prop { bl }
28690 { { t } { r } { -1 } { -1 } }
28691 \prop_put:Nnn \l_coffin_display_handles_prop { bhc }
28692 { { t } { hc } { 0 } { -1 } }
28693 \prop_put:Nnn \l_coffin_display_handles_prop { br }
28694 { { t } { 1 } { 1 } { -1 } }
28695 \prop_put:Nnn \l_coffin_display_handles_prop { Tl }
28696 { { t } { r } { -1 } { -1 } }
28697 \prop_put:Nnn \l_coffin_display_handles_prop { Thc }
```

```

28698 { { t } { hc } { 0 } { -1 } }
28699 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
28700 { { t } { 1 } { 1 } { -1 } }
28701 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
28702 { { vc } { r } { -1 } { 1 } }
28703 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
28704 { { vc } { hc } { 0 } { 1 } }
28705 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
28706 { { vc } { 1 } { 1 } { 1 } }
28707 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
28708 { { b } { r } { -1 } { -1 } }
28709 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
28710 { { b } { hc } { 0 } { -1 } }
28711 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
28712 { { b } { 1 } { 1 } { -1 } }

(End definition for \l__coffin_display_handles_prop.)

```

\l\_\_coffin\_display\_offset\_dim The standard offset for the label from the handle position when displaying handles.

```

28713 \dim_new:N \l__coffin_display_offset_dim
28714 \dim_set:Nn \l__coffin_display_offset_dim { 2pt }

(End definition for \l__coffin_display_offset_dim.)

```

\l\_\_coffin\_display\_x\_dim \l\_\_coffin\_display\_y\_dim As the intersections of poles have to be calculated to find which ones to print, there is a need to avoid repetition. This is done by saving the intersection into two dedicated values.

```

28715 \dim_new:N \l__coffin_display_x_dim
28716 \dim_new:N \l__coffin_display_y_dim

(End definition for \l__coffin_display_x_dim and \l__coffin_display_y_dim.)

```

\l\_\_coffin\_display\_poles\_prop A property list for printing poles: various things need to be deleted from this to get a “nice” output.

```
28717 \prop_new:N \l__coffin_display_poles_prop
```

(End definition for \l\_\_coffin\_display\_poles\_prop.)

\l\_\_coffin\_display\_font\_t1 Stores the settings used to print coffin data: this keeps things flexible.

```

28718 \tl_new:N \l__coffin_display_font_t1
28719 \cs_if_exist:NTF \AtBeginDocument
28720 { \AtBeginDocument }
28721 { \use:n }
28722 {
28723 __kernel_tl_set:Nx \l__coffin_display_font_t1
28724 {
28725 \cs_if_exist:NT \sffamily { \exp_not:N \sffamily }
28726 \cs_if_exist:NT \tiny { \exp_not:N \tiny }
28727 }
28728 }

(End definition for \l__coffin_display_font_t1.)

```

\\_\_coffin\_color:n Calls \color, and otherwise does nothing if \color is not defined.

```
28729 \cs_if_exist:NTF \AtBeginDocument
28730 { \AtBeginDocument }
28731 { \use:n }
28732 {
28733 \cs_new_protected:Npx __coffin_color:n #1
28734 {
28735 \cs_if_exist:NTF \color_select:n
28736 { \color_select:n {#1} }
28737 {
28738 \cs_if_exist:NT \color
28739 { \exp_not:N \color {#1} }
28740 }
28741 }
28742 }
```

(End definition for \\_\_coffin\_color:n.)

\\_\_coffin\_rule:nn Abstract out creation of rules here until there is a higher-level interface.

```
28743 \cs_new_protected:Npx __coffin_rule:nn #1#2
28744 {
28745 \cs_if_exist:NTF \rule
28746 { \exp_not:N \rule {#1} {#2} }
28747 { \hbox:n { \tex_vrule:D width #1 height #2 \scan_stop: } }
28748 }
```

(End definition for \\_\_coffin\_rule:nn.)

\coffin\_mark\_handle:Nnnn  
\coffin\_mark\_handle:cnnn  
\\_\_coffin\_mark\_handle\_aux:nnnnNnn

Marking a single handle is relatively easy. The standard attachment function is used, meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.

```
28749 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
28750 {
28751 \hcoffin_set:Nn \l__coffin_display_pole_coffin
28752 {
28753 __coffin_color:n {#4}
28754 __coffin_rule:nn { 1pt } { 1pt }
28755 }
28756 __coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
28757 \l__coffin_display_pole_coffin { hc } { vc } { Opt } { Opt }
28758 \hcoffin_set:Nn \l__coffin_display_coord_coffin
28759 {
28760 __coffin_color:n {#4}
28761 \l__coffin_display_font_tl
28762 (\tl_to_str:n { #2 , #3 })
28763 }
28764 \prop_get:NnN \l__coffin_display_handles_prop
28765 { #2 #3 } \l__coffin_internal_tl
28766 \quark_if_no_value:NTF \l__coffin_internal_tl
28767 {
28768 \prop_get:NnN \l__coffin_display_handles_prop
28769 { #3 #2 } \l__coffin_internal_tl
28770 \quark_if_no_value:NTF \l__coffin_internal_tl
```

```

28771 {
28772 __coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
28773 \l__coffin_display_coord_coffin { 1 } { vc }
28774 { 1pt } { Opt }
28775 }
28776 {
28777 \exp_last_unbraced:No __coffin_mark_handle_aux:nnnnNnn
28778 \l__coffin_internal_t1 #1 {#2} {#3}
28779 }
28780 }
28781 {
28782 \exp_last_unbraced:No __coffin_mark_handle_aux:nnnnNnn
28783 \l__coffin_internal_t1 #1 {#2} {#3}
28784 }
28785 }
28786 \cs_new_protected:Npn __coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
28787 {
28788 __coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
28789 \l__coffin_display_coord_coffin {#1} {#2}
28790 { #3 \l__coffin_display_offset_dim }
28791 { #4 \l__coffin_display_offset_dim }
28792 }
28793 \cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }

(End definition for \coffin_mark_handle:Nnnn and __coffin_mark_handle_aux:nnnnNnn. This function is documented on page 253.)
```

**\coffin\_display\_handles:Nn**  
**\coffin\_display\_handles:cn**

Printing the poles starts by removing any duplicates, for which the H poles is used as the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

```

28794 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
28795 {
28796 \hcoffin_set:Nn \l__coffin_display_pole_coffin
28797 {
28798 __coffin_color:n {#2}
28799 __coffin_rule:nn { 1pt } { 1pt }
2880 }
2881 \prop_set_eq:Nc \l__coffin_display_poles_prop
2882 { coffin ~ __coffin_to_value:N #1 ~ poles }
2883 __coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
2884 __coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
2885 \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
2886 { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
2887 __coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
2888 \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
2889 { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
2890 \coffin_set_eq>NN \l__coffin_display_coffin #1
2891 \prop_map_inline:Nn \l__coffin_display_poles_prop
2892 {
2893 \prop_remove:Nn \l__coffin_display_poles_prop {##1}
2894 __coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
2895 }
2896 \box_use_drop:N \l__coffin_display_coffin
```

```
28817 }
```

For each pole there is a check for an intersection, which here does not give an error if none is found. The successful values are stored and used to align the pole coffin with the main coffin for output. The positions are recovered from the preset list if available.

```
28818 \cs_new_protected:Npn __coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
28819 {
28820 \prop_map_inline:Nn \l__coffin_display_poles_prop
28821 {
28822 \bool_set_false:N \l__coffin_error_bool
28823 __coffin_calculate_intersection:nnnnnnnnn {#2} {#3} {#4} {#5} ##2
28824 \bool_if:NF \l__coffin_error_bool
28825 {
28826 \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
28827 \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
28828 __coffin_display_attach:Nnnnn
28829 \l__coffin_display_pole_coffin { hc } { vc }
28830 { Opt } { Opt }
28831 \hcoffin_set:Nn \l__coffin_display_coord_coffin
28832 {
28833 __coffin_color:n {#6}
28834 \l__coffin_display_font_tl
28835 (\tl_to_str:n { #1 , ##1 })
28836 }
28837 \prop_get:NnN \l__coffin_display_handles_prop
28838 { #1 ##1 } \l__coffin_internal_tl
28839 \quark_if_no_value:NTF \l__coffin_internal_tl
28840 {
28841 \prop_get:NnN \l__coffin_display_handles_prop
28842 { ##1 #1 } \l__coffin_internal_tl
28843 \quark_if_no_value:NTF \l__coffin_internal_tl
28844 {
28845 __coffin_display_attach:Nnnnn
28846 \l__coffin_display_coord_coffin { 1 } { vc }
28847 { ipt } { Opt }
28848 }
28849 {
28850 \exp_last_unbraced:No
28851 __coffin_display_handles_aux:nnnn
28852 \l__coffin_internal_tl
28853 }
28854 }
28855 {
28856 \exp_last_unbraced:No __coffin_display_handles_aux:nnnn
28857 \l__coffin_internal_tl
28858 }
28859 }
28860 }
28861 }
28862 \cs_new_protected:Npn __coffin_display_handles_aux:nnnn #1#2#3#4
28863 {
28864 __coffin_display_attach:Nnnnn
28865 \l__coffin_display_coord_coffin {#1} {#2}
28866 { #3 \l__coffin_display_offset_dim }
```

```

28867 { #4 \l__coffin_display_offset_dim }
28868 }
28869 \cs_generate_variant:Nn \coffin_display_handles:Nn { c }

This is a dedicated version of \coffin_attach:NnnNnnnn with a hard-wired first coffin.
As the intersection is already known and stored for the display coffin the code simply
uses it directly, with no calculation.

```

```

28870 \cs_new_protected:Npn __coffin_display_attach:Nnnnn #1#2#3#4#5
28871 {
28872 __coffin_calculate_intersection:Nnn #1 {#2} {#3}
28873 \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
28874 \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
28875 \dim_set:Nn \l__coffin_offset_x_dim
28876 { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
28877 \dim_set:Nn \l__coffin_offset_y_dim
28878 { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
28879 \hbox_set:Nn \l__coffin_aligned_coffin
28880 {
28881 \box_use:N \l__coffin_display_coffin
28882 \tex_kern:D -\box_wd:N \l__coffin_display_coffin
28883 \tex_kern:D \l__coffin_offset_x_dim
28884 \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
28885 }
28886 \box_set_ht:Nn \l__coffin_aligned_coffin
28887 { \box_ht:N \l__coffin_display_coffin }
28888 \box_set_dp:Nn \l__coffin_aligned_coffin
28889 { \box_dp:N \l__coffin_display_coffin }
28890 \box_set_wd:Nn \l__coffin_aligned_coffin
28891 { \box_wd:N \l__coffin_display_coffin }
28892 \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
28893 }

```

(End definition for \coffin\_display\_handles:Nn and others. This function is documented on page 253.)

\coffin\_show\_structure:N For showing the various internal structures attached to a coffin in a way that keeps things relatively readable. If there is no apparent structure then the code complains.

```

\coffin_show_structure:c
\coffin_log_structure:N
\coffin_log_structure:c
__coffin_show_structure>NN
28894 \cs_new_protected:Npn \coffin_show_structure:N
28895 { __coffin_show_structure>NN \msg_show:nxxxxx }
28896 \cs_generate_variant:Nn \coffin_show_structure:N { c }
28897 \cs_new_protected:Npn \coffin_log_structure:N
28898 { __coffin_show_structure>NN \msg_log:nxxxxx }
28899 \cs_generate_variant:Nn \coffin_log_structure:N { c }
28900 \cs_new_protected:Npn __coffin_show_structure>NN #1#2
28901 {
28902 __coffin_if_exist:NT #2
28903 {
28904 #1 { LaTeX / kernel } { show-coffin }
28905 { \token_to_str:N #2 }
28906 {
28907 \iow_newline: >~ ht ~== \dim_eval:n { \coffin_ht:N #2 }
28908 \iow_newline: >~ dp ~== \dim_eval:n { \coffin_dp:N #2 }
28909 \iow_newline: >~ wd ~== \dim_eval:n { \coffin_wd:N #2 }
28910 }
28911 }

```

```

28912 \prop_map_function:cN
28913 { coffin ~ __coffin_to_value:N #2 ~ poles }
28914 \msg_show_item_unbraced:nn
28915 }
28916 {
28917 }
28918 }

```

(End definition for `\coffin_show_structure:N`, `\coffin_log_structure:N`, and `\__coffin_show_structure:NN`. These functions are documented on page 253.)

### 43.9 Messages

```

28919 __kernel_msg_new:nnn { kernel } { no-pole-intersection }
28920 { No-intersection-between-coffin-poles. }
28921 {
28922 LaTeX-was-asked-to-find-the-intersection-between-two-poles,~
28923 but-they-do-not-have-a-unique-meeting-point:~
28924 the-value-(Opt,-Opt)-will-be-used.
28925 }
28926 __kernel_msg_new:nnn { kernel } { unknown-coffin }
28927 { Unknown-coffin-'#1'. }
28928 { The-coffin-'#1'-was-never-defined. }
28929 __kernel_msg_new:nnn { kernel } { unknown-coffin-pole }
28930 { Pole-'#1'-unknown-for-coffin-'#2'. }
28931 {
28932 LaTeX-was-asked-to-find-a-typesetting-pole-for-a-coffin,~
28933 but-either-the-coffin-does-not-exist-or-the-pole-name-is-wrong.
28934 }
28935 __kernel_msg_new:nn { kernel } { show-coffin }
28936 {
28937 Size-of-coffin-#1 : #2 \\
28938 Poles-of-coffin-#1 : #3 .
28939 }
28940
```

## 44 I3color-base Implementation

```

28941 {*package}
28942 @@=color

```

`\l__color_current_tl` The color currently active for foreground (text, *etc.*) material. This is stored in the form of a color model followed by one or more values. There are four pre-defined models, three of which take numerical values in the range [0, 1]:

- **gray** `<gray>` Grayscale color with the `<gray>` value running from 0 (fully black) to 1 (fully white)
- **cmyk** `<cyan>` `<magenta>` `<yellow>` `<black>`
- **rgb** `<red>` `<green>` `<blue>`

Notice that the value are separated by spaces. There is a fourth pre-defined model using a string value and a numerical one:

- **spot** *<name>* *<tint>* A pre-defined spot color, where the *<name>* should be a pre-defined string color name and the *<tint>* should be in the range [0, 1].

Additional models may be created to allow mixing of spot colors. The number of data entries these require will depend on the number of colors to be mixed.

**TEXhackers note:** The content of `\l_color_current_tl` comprises two brace groups, the first containing the color model and the second containing the value(s) applicable in that model.

(*End definition for \l\_color\_current\_tl.*)

**\color\_group\_begin:** Grouping for color is the same as using the basic `\group_begin:` and `\group_end:` functions. However, for semantic reasons, they are renamed here.

```
28943 \cs_new_eq:NN \color_group_begin: \group_begin:
28944 \cs_new_eq:NN \color_group_end: \group_end:
```

(*End definition for \color\_group\_begin: and \color\_group\_end:. These functions are documented on page 255.*)

**\color\_ensure\_current:** A driver-independent wrapper for setting the foreground color to the current color “now”.

```
28945 \cs_new_protected:Npn \color_ensure_current:
28946 {
28947 __color_backend_pickup:N \l_color_current_tl
28948 __color_select:N \l_color_current_tl
28949 }
```

(*End definition for \color\_ensure\_current:. This function is documented on page 255.*)

**\s\_color\_stop** Internal scan marks.

```
28950 \scan_new:N \s_color_stop
```

(*End definition for \s\_color\_stop.*)

**\\_\_color\_select:N** Take an internal color specification and pass it to the driver. This code is needed to ensure the current color but will also be used by the higher-level experimental material.

```
28951 \cs_new_protected:Npn __color_select:N #1
28952 { \exp_after:wN __color_select:nn #1 }
28953 \cs_new_protected:Npn __color_select:nn #1#2
28954 { \use:c { __color_backend_select_ #1 :n } {#2} }
```

(*End definition for \\_\_color\_select:N and \\_\_color\_select:nn.*)

**\l\_color\_current\_tl** The current color, with the model and

```
28955 \tl_new:N \l_color_current_tl
28956 \tl_set:Nn \l_color_current_tl { { gray } { 0 } }
```

(*End definition for \l\_color\_current\_tl.*)

```
28957 </package>
```

## 45 l3luatex implementation

28958 `(*package)`

### 45.1 Breaking out to Lua

28959 `(*tex)`

28960 `@@=lua`

`\_lua_escape:n` Copies of primitives.

28961 `\cs_new_eq:NN \_lua_escape:n \tex_luaescapestring:D`

28962 `\cs_new_eq:NN \_lua_now:n \tex_directlua:D`

28963 `\cs_new_eq:NN \_lua_shipout:n \tex_latelua:D`

(End definition for `\_lua_escape:n`, `\_lua_now:n`, and `\_lua_shipout:n`.)

These functions are set up in l3str for bootstrapping: we want to replace them with a “proper” version at this stage, so clean up.

28964 `\cs_undefine:N \lua_escape:e`

28965 `\cs_undefine:N \lua_now:e`

`\lua_now:n` Wrappers around the primitives. As with engines other than LuaTeX these have to be  
`\lua_now:e` macros, we give them the same status in all cases. When LuaTeX is not in use, simply  
`\lua_shipout_e:n` give an error message/

`\lua_shipout:n`  
`\lua_escape:n`  
`\lua_escape:e`

28966 `\cs_new:Npn \lua_now:e #1 { \_lua_now:n {#1} }`

28967 `\cs_new:Npn \lua_now:n #1 { \lua_now:e { \exp_not:n {#1} } }`

28968 `\cs_new_protected:Npn \lua_shipout_e:n #1 { \_lua_shipout:n {#1} }`

28969 `\cs_new_protected:Npn \lua_shipout:n #1`  
    `{ \lua_shipout_e:n { \exp_not:n {#1} } }`

28970 `\cs_new:Npn \lua_escape:e #1 { \_lua_escape:n {#1} }`

28971 `\cs_new:Npn \lua_escape:n #1 { \lua_escape:e { \exp_not:n {#1} } }`

28972 `\sys_if_engine_luatex:F`  
    `{`

28973 `\clist_map_inline:nn`  
    `{`

28974 `\lua_escape:n , \lua_escape:e ,`  
    `\lua_now:n , \lua_now:e`

28975 `}`

28976 `{`

28977 `\cs_set:Npn #1 ##1`  
    `{`

28978 `\_kernel_msg_expandable_error:nnn`  
    `{ kernel } { luatex-required } { #1 }`

28979 `}`

28980 `}`

28981 `\clist_map_inline:nn`  
    `{ \lua_shipout_e:n , \lua_shipout:n }`

28982 `{`

28983 `\cs_set_protected:Npn #1 ##1`  
    `{`

28984 `\_kernel_msg_error:nnn`  
    `{ kernel } { luatex-required } { #1 }`

28985 `}`

28986 `}`

28987 `\_kernel_msg_error:nnn`  
    `{ kernel } { luatex-required } { #1 }`

28988 `}`

28989 `}`

28990 `\_kernel_msg_error:nnn`  
    `{ kernel } { luatex-required } { #1 }`

28991 `}`

28992 `}`

28993 `\_kernel_msg_error:nnn`  
    `{ kernel } { luatex-required } { #1 }`

28994 `}`

28995 `}`

28996 `}`

(End definition for `\lua_now:n` and others. These functions are documented on page 256.)

## 45.2 Messages

```
28997 __kernel_msg_new:nnn { kernel } { luatex-required }
28998 { LuaTeX-engine-not-in-use!~Ignoring~#1. }
28999 {
29000 The~feature~you~are~using~is~only~available~
29001 with~the~LuaTeX-engine.~LaTeX3~ignored~'#1'.
29002 }
29003 </tex>
```

## 45.3 Lua functions for internal use

```
29004 /*lua>
```

Most of the emulation of pdfTEX here is based heavily on Heiko Oberdiek's pdftex-cmds package.

**13kernel** Create a table for the kernel's own use.  
**ltx.utils**

```
29005 13kernel = 13kernel or { }
29006 local l3kernel = 13kernel
29007 ltx = ltx or {utils={}}
29008 ltx.utils = ltx.utils or {}
29009 local ltxutils = ltx.utils
```

(End definition for 13kernel and ltx.utils. These functions are documented on page 257.)  
Local copies of global tables.

```
29010 local io = io
29011 local kpse = kpse
29012 local lfs = lfs
29013 local math = math
29014 local md5 = md5
29015 local os = os
29016 local string = string
29017 local tex = tex
29018 local texio = texio
29019 local tonumber = tonumber
```

Local copies of standard functions.

```
29020 local abs = math.abs
29021 local byte = string.byte
29022 local floor = math.floor
29023 local format = string.format
29024 local gsub = string.gsub
29025 local lfs_attr = lfs.attributes
29026 local open = io.open
29027 local os_date = os.date
29028 local setcatcode = tex.setcatcode
29029 local sprint = tex.sprint
29030 local cprint = tex.cprint
29031 local write = tex.write
29032 local write_nl = texio.write_nl
29033 local utf8_char = utf8.char
29034
29035 local scan_int = token.scan_int or token.scan_integer
29036 local scan_string = token.scan_string
29037 local scan_keyword = token.scan_keyword
```

```

29038 local put_next = token.put_next
29039
29040 local true_tok = token.create'prg_return_true:'
29041 local false Tok = token.create'prg_return_false:'
29042 local function deprecated(table, name, func)
29043 table[name] = function(...)
29044 write_nl(format("Calling deprecated Lua function %s", name))
29045 table[name] = func
29046 return func(...)
29047 end
29048 end
29049 % \end{macrocode}
29050 %
29051 % Deal with Con\TeX{}t: doesn't use /kpse/ library.
29052 % \begin{macrocode}
29053 local kpse_find = (resolvers and resolvers.findfile) or kpse.find_file

```

`escapehex` An internal auxiliary to convert a string to the matching hex escape. This works on a byte basis: extension to handled UTF-8 input is covered in `pdftexcmds` but is not currently required here.

```

29054 local function escapehex(str)
29055 return (gsub(str, ".", function (ch) return format("%02X", byte(ch)) end))
29056 end

```

(*End definition for `escapehex`.*)

`13kernel.charcat` Creating arbitrary chars using `tex.cprint`. The alternative approach using `token.put_next(token.create)` would be about 10% slower.

```

29058 deprecated(13kernel, 'charcat', function(charcode, catcode)
29059 cprint(catcode, utf8_char(charcode))
29060 end)

```

(*End definition for `13kernel.charcat`. This function is documented on page 257.*)

`13kernel.elapsedtime` Simple timing set up: give the result from the system clock in scaled seconds.

```

13kernel.resettimer
29061 local os_clock = os.clock
29062 local base_clock_time = 0
29063 local function elapsedtime()
29064 local val = (os_clock() - base_clock_time) * 65536 + 0.5
29065 if val > 2147483647 then
29066 val = 2147483647
29067 end
29068 write(format("%d", floor(val)))
29069 end
29070 13kernel.elapsedtime = elapsedtime
29071 local function resettimer()
29072 base_clock_time = os_clock()
29073 end
29074 13kernel.resettimer = resettimer

```

(*End definition for `13kernel.elapsedtime` and `13kernel.resettimer`. These functions are documented on page 257.*)

**ltx.utils.filiedump** Similar comments here to the next function: read the file in binary mode to avoid any line-end weirdness.

```
29075 local function filiedump(name,offset,length)
29076 local file = kpse_find(name,"tex",true)
29077 if not file then return end
29078 local f = open(file,"rb")
29079 if not f then return end
29080 if offset and offset > 0 then
29081 f:seek("set", offset)
29082 end
29083 local data = f:read(length or 'a')
29084 f:close()
29085 return escapehex(data)
29086 end
29087 ltxutils.filiedump = filiedump
29088 deprecated(l3kernel, "filiedump", function(name, offset, length)
29089 local dump = filiedump(name, tonumber(offset), tonumber(length))
29090 if dump then
29091 write(dump)
29092 end
29093 end)
```

(End definition for `ltx.utils.filiedump` and `l3kernel.filiedump`. These functions are documented on page [257](#).)

**md5.HEX** Hash a string and return the hash in uppercase hexadecimal format. In some engines, this is build-in. For traditional LuaTeX, the conversion to hexadecimal has to be done by us.

```
29094 local md5_HEX = md5.HEX
29095 if not md5_HEX then
29096 local md5_sum = md5.sum
29097 function md5_HEX(data)
29098 return escapehex(md5_sum(data))
29099 end
29100 md5.HEX = md5_HEX
29101 end
```

(End definition for `md5.HEX`. This function is documented on page ??.)

**ltx.utils.filemd5sum** Read an entire file and hash it: the hash function itself is a built-in. As Lua is byte-based there is no work needed here in terms of UTF-8 (see `pdftexcmds` and how it handles strings that have passed through LuaTeX). The file is read in binary mode so that no line ending normalisation occurs.

```
29102 local function filemd5sum(name)
29103 local file = kpse_find(name, "tex", true) if not file then return end
29104 local f = open(file, "rb") if not f then return end
29105
29106 local data = f:read("*a")
29107 f:close()
29108 return md5_HEX(data)
29109 end
29110 ltxutils.filemd5sum = filemd5sum
29111 deprecated(l3kernel, "filemd5sum", function(name)
29112 local hash = filemd5sum(name)
```

```

29113 if hash then
29114 write(hash)
29115 end
29116 end)

```

(End definition for `ltx.utils.filemd5sum` and `l3kernel.filemdfivesum`. These functions are documented on page [257](#).)

### `ltx.utils.filemoddate` `l3kernel.filemoddate`

There are two cases: If the C standard library is C99 compliant, we can use `%z` to get the timezone in almost the right format. We only have to add primes and replace a zero or missing offset with Z.

Of course this would be boring, so Windows does things differently. There we have to manually calculate the offset. See procedure `makepdftime` in `utils.c` of pdfTeX.

```

29117 local filemoddate
29118 if os_date'%z':match'^[+-]%-d%d%d%d$' then
29119 local pattern = lpeg.Cs(16 *
29120 (lpeg.Cg(lpeg.S'+-' * '0000' * lpeg.Cc'Z')
29121 + 3 * lpeg.Cc'" * 2 * lpeg.Cc)"
29122 + lpeg.Cc'Z')
29123 * -1)
29124 function filemoddate(name)
29125 local file = kpse_find(name, "tex", true)
29126 if not file then return end
29127 local date = lfs_attr(file, "modification")
29128 if not date then return end
29129 return pattern:match(os_date("D:%Y%m%d%H%M%S%z", date))
29130 end
29131 else
29132 local function filemoddate(name)
29133 local file = kpse_find(name, "tex", true)
29134 if not file then return end
29135 local date = lfs_attr(file, "modification")
29136 if not date then return end
29137 local d = os_date("*t", date)
29138 local u = os_date("!*t", date)
29139 local off = 60 * (d.hour - u.hour) + d.min - u.min
29140 if d.year ~= u.year then
29141 if d.year > u.year then
29142 off = off + 1440
29143 else
29144 off = off - 1440
29145 end
29146 elseif d.yday ~= u.yday then
29147 if d.yday > u.yday then
29148 off = off + 1440
29149 else
29150 off = off - 1440
29151 end
29152 end
29153 local timezone
29154 if off == 0 then
29155 timezone = "Z"
29156 else
29157 if off < 0 then

```

```

29158 timezone = "-"
29159 off = -off
29160 else
29161 timezone = "+"
29162 end
29163 timezone = format("%s%02d'%02d'", timezone, hours // 60, hours % 60)
29164 end
29165 return format("D:%04d%02d%02d%02d%02d%s",
29166 d.year, d.month, d.day, d.hour, d.min, d.sec, timezone)
29167 end
29168 end
29169 ltxutils.filemoddate = filemoddate
29170 deprecated(l3kernel, "filemoddate", function(name)
29171 local hash = filemoddate(name)
29172 if hash then
29173 write(hash)
29174 end
29175 end)

```

(End definition for `ltx.utils.filemoddate` and `l3kernel.filemoddate`. These functions are documented on page 257.)

`ltx.utils.filesize` A simple disk lookup.

```

13kernel.filesize
29176 local function filesize(name)
29177 local file = kpse_find(name, "tex", true)
29178 if file then
29179 local size = lfs_attr(file, "size")
29180 if size then
29181 return size
29182 end
29183 end
29184 end
29185 ltxutils.filesize = filesize
29186 deprecated(l3kernel, "filesize", function(name)
29187 local size = filesize(name)
29188 if size then
29189 write(size)
29190 end
29191 end)

```

(End definition for `ltx.utils.filesize` and `l3kernel.filesize`. These functions are documented on page 257.)

`l3kernel.strcmp` String comparison which gives the same results as pdfTEX's `\pdfstrcmp`, although the ordering should likely not be relied upon!

```

29192 deprecated(l3kernel, "strcmp", function (A, B)
29193 if A == B then
29194 write("0")
29195 elseif A < B then
29196 write("-1")
29197 else
29198 write("1")
29199 end
29200 end)

```

(End definition for `l3kernel.strcmp`. This function is documented on page 258.)

`l3kernel.shellescape` Replicating the pdfTeX log interaction for shell escape.

```
29201 local os_exec = os.execute
29202 deprecated(l3kernel, "shellescape", function(cmd)
29203 local status,msg = os_exec(cmd)
29204 if status == nil then
29205 write_nl("log","runsystem(.. cmd ..)..." .. msg .. "\n")
29206 elseif status == 0 then
29207 write_nl("log","runsystem(.. cmd ..)..." .. "executed\n")
29208 else
29209 write_nl("log","runsystem(.. cmd ..)..." .. "failed " .. (msg or "") .. "\n")
29210 end
29211 end)
```

(End definition for `l3kernel.shellescape`. This function is documented on page 258.)

`luadef` An internal function for defining control sequences from Lua which behave like primitives. This acts as a wrapper around `token.set_lua` which accepts a function instead of an index into the functions table.

```
29212 local luacmd do
29213 local token_create = token.create
29214 local set_lua = token.set_lua
29215 local undefined_cs = token.command_id'undefined_cs'
29216
29217 if not context and not luatexbase then require'ltxluatex' end
29218 if luatexbase then
29219 local new_luafunction = luatexbase.new_luafunction
29220 local functions = lua.get_functions_table()
29221 function luacmd(name, func, ...)
29222 local id
29223 local tok = token_create(name)
29224 if tok.command == undefined_cs then
29225 id = new_luafunction(name)
29226 set_lua(name, id, ...)
29227 else
29228 id = tok.index or tok.mode
29229 end
29230 functions[id] = func
29231 end
29232 elseif context then
29233 local register = context.functions.register
29234 local functions = context.functions.known
29235 function luacmd(name, func, ...)
29236 local tok = token.create(name)
29237 if tok.command == undefined_cs then
29238 token.set_lua(name, register(func), ...)
29239 else
29240 functions[tok.index or tok.mode] = func
29241 end
29242 end
29243 end
29244 end
```

(End definition for luadef.)

```
29245 </lua>
29246 </package>
```

## 46 I3unicode implementation

```
29247 {*package}
29248 @@=char
```

Case changing both for strings and “text” requires data from the Unicode Consortium. Some of this is build in to the format (as `\lccode` and `\uccode` values) but this covers only the simple one-to-one situations and does not fully handle for example case folding.

As only the data needs to remain at the end of this process, everything is set up inside a group. The only thing that is outside is creating a stream: they are global anyway and it is best to force a stream for all engines. For performance reasons, some of the code here is very low-level: the material is read during loading `expl3` in package mode.

```
29249 \ior_new:N \g__char_data_ior
29250 \bool_lazy_or:nnTF { \sys_if_engine_luatex_p: } { \sys_if_engine_xetex_p: }
29251 {
29252 \group_begin:
```

Access the primitive but suppress further expansion: active chars are otherwise an issue.

```
29253 \cs_set:Npn __char_generate_char:n #1
29254 { \tex_detokenize:D \tex_expandafter:D { \tex_Uchar:D "#1" } }
```

A fast local implementation for generating characters; the chars may be active, so we prevent further expansion.

```
29255 \cs_set:Npx __char_generate:n #1
29256 {
29257 \exp_not:N \tex_unexpanded:D \exp_not:N \exp_after:wN
29258 {
29259 \exp_not:N \tex_Ucharcat:D
29260 #1 ~
29261 \tex_catcode:D #1 ~
29262 }
29263 }
```

Parse the main Unicode data file for two things. First, we want the titlecase exceptions: the one-to-one lower- and uppercase mappings it contains are all be covered by the T<sub>E</sub>X data. Second, we need normalization data: at present, just the canonical NFD mappings. Those all yield either one or two codepoints, so the split is relatively easy.

```
29264 \ior_open:Nn \g__char_data_ior { UnicodeData.txt }
29265 \cs_set_protected:Npn __char_data_auxi:w
29266 #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
29267 {
29268 \tl_if_blank:nF {#6}
29269 {
29270 \tl_if_head_eq_charcode:nNF {#6} < % >
29271 { __char_data_auxii:w #1 ; #6 ~ \q_stop }
29272 }
29273 __char_data_auxiii:w #1 ;
```

```

29274
29275 }
29276
29277 \cs_set_protected:Npn __char_data_auxii:w #1 ; #2 ~ #3 \q_stop
29278 {
29279 \tl_const:cx
29280 { c__char_nfd_ __char_generate_char:n {#1} _tl }
29281 {
29282 __char_generate:n { "#2 }
29283 \tl_if_blank:nF {#3}
29284 { __char_generate:n { "#3 } }
29285 }
29286 \cs_set_protected:Npn __char_data_auxiii:w
29287 #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ~ \q_stop
29288 {
29289 \cs_set_nopar:Npn \l__char_tmpa_tl {#7}
29290 \reverse_if:N \if_meaning:w \l__char_tmpa_tl \c_empty_tl
29291 \cs_set_nopar:Npn \l__char_tmpb_t1 {#5}
29292 \reverse_if:N \if_meaning:w \l__char_tmpa_t1 \l__char_tmpb_t1
29293 \tl_const:cx
29294 { c__char_titlecase_ __char_generate_char:n {#1} _tl }
29295 { __char_generate:n { "#7 } }
29296 \fi:
29297 \fi:
29298 }
29299 \group_begin:
29300 \char_set_catcode_space:n { '\ }%
29301 \ior_map_variable:NNn \g__char_data_ior \l__char_tmpa_t1
29302 {%
29303 \if_meaning:w \l__char_tmpa_t1 \c_space_t1
29304 \exp_after:wN \ior_map_break:
29305 \fi:
29306 \exp_after:wN __char_data_auxi:w \l__char_tmpa_t1 \q_stop
29307 }%
29308 \group_end:
29309 \ior_close:N \g__char_data_ior

```

The other data files all use C-style comments so we have to worry about # tokens (and reading as strings). The set up for case folding is in two parts. For the basic (core) mappings, folding is the same as lower casing in most positions so only store the differences. For the more complex foldings, always store the result, splitting up the two or three code points in the input as required.

```

29309 \ior_open:Nn \g__char_data_ior { CaseFolding.txt }
29310 \cs_set_protected:Npn __char_data_auxi:w #1 ;~ #2 ;~ #3 ; #4 \q_stop
29311 {
29312 \if:w \tl_head:n { #2 ? } C
29313 \reverse_if:N \if_int_compare:w
29314 \char_value_lccode:n {"#1} = "#3 ~
29315 \tl_const:cx
29316 { c__char_foldcase_ __char_generate_char:n {#1} _tl }
29317 { __char_generate:n { "#3 } }
29318 \fi:
29319 \else:
29320 \if:w \tl_head:n { #2 ? } F
29321 __char_data_auxii:w #1 ~ #3 ~ \q_stop

```

```

29322 \fi:
29323 \fi:
29324 }
29325 \cs_set_protected:Npn __char_data_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
29326 {
29327 \tl_const:cx { c__char_foldcase_ __char_generate_char:n {#1} _tl }
29328 {
29329 __char_generate:n { "#2 }
29330 __char_generate:n { "#3 }
29331 \tl_if_blank:nF {#4}
29332 { __char_generate:n { \int_value:w "#4 } }
29333 }
29334 }
29335 \ior_str_map_inline:Nn \g__char_data_ior
29336 {
29337 \reverse_if:N \if:w \c_hash_str \tl_head:w #1 \c_hash_str \q_stop
29338 __char_data_auxi:w #1 \q_stop
29339 \fi:
29340 }
29341 \ior_close:N \g__char_data_ior

```

For upper- and lowercasing special situations, there is a bit more to do as we also have title casing to consider, plus we need to stop part-way through the file.

```

29342 \ior_open:Nn \g__char_data_ior { SpecialCasing.txt }
29343 \cs_set_protected:Npn __char_data_auxi:w
29344 #1 ;~ #2 ;~ #3 ;~ #4 ;~ #5 \q_stop
29345 {
29346 \use:n { __char_data_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
29347 \use:n { __char_data_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
29348 \str_if_eq:nnF {#3} {#4}
29349 { \use:n { __char_data_auxii:w #1 ~ title ~ #3 ~ } ~ \q_stop }
29350 }
29351 \cs_set_protected:Npn __char_data_auxii:w
29352 #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
29353 {
29354 \tl_if_empty:nF {#4}
29355 {
29356 \tl_const:cx { c__char_ #2 case_ __char_generate_char:n {#1} _tl }
29357 {
29358 __char_generate:n { "#3 }
29359 __char_generate:n { "#4 }
29360 \tl_if_blank:nF {#5}
29361 { __char_generate:n { "#5 } }
29362 }
29363 }
29364 }
29365 \ior_str_map_inline:Nn \g__char_data_ior
29366 {
29367 \str_if_eq:eeTF
29368 { \tl_head:w #1 \c_hash_str \q_stop }
29369 { \c_hash_str }
29370 {
29371 \str_if_eq:eeT
29372 {#1}
29373 { \c_hash_str \c_space_tl Conditional~Mappings }

```

```

29374 { \ior_map_break: }
29375 }
29376 { __char_data_auxi:w #1 \q_stop }
29377 }
29378 \ior_close:N \g__char_data_ior
29379 \group_end:
29380 }
```

For the 8-bit engines, the above is skipped but there is still some set up required. As case changing can only be applied to bytes, and they have to be in the ASCII range, we define a series of data stores to represent them, and the data are used such that only these are ever case-changed. We do open and close one file to force allocation of a read: this keeps all engines in line.

```

29381 {
29382 \group_begin:
29383 \cs_set_protected:Npn __char_tmp:NN #1#2
29384 {
29385 \quark_if_recursion_tail_stop:N #2
29386 \tl_const:cn { c__char_uppercase_ #2 _tl } {#1}
29387 \tl_const:cn { c__char_lowercase_ #1 _tl } {#2}
29388 \tl_const:cn { c__char_foldcase_ #1 _tl } {#2}
29389 __char_tmp:NN
29390 }
29391 __char_tmp:NN
29392 AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
29393 ? \q_recursion_tail \q_recursion_stop
29394 \ior_open:Nn \g__char_data_ior { UnicodeData.txt }
29395 \ior_close:N \g__char_data_ior
29396 \group_end:
29397 }
29398
```

## 47 I3text implementation

```

29399 /*package)
29400 @@=text}
```

### 47.1 Internal auxiliaries

\s\_\_text\_stop Internal scan marks.

```
29401 \scan_new:N \s__text_stop
```

(End definition for \s\_\_text\_stop.)

\q\_\_text\_nil Internal quarks.

```
29402 \quark_new:N \q__text_nil
```

(End definition for \q\_\_text\_nil.)

\\_\_text\_quark\_if\_nil\_p:n Branching quark conditional.

```
__text_quark_if_nil:nTF 29403 __kernel_quark_new_conditional:Nn __text_quark_if_nil:n { TF }
```

(End definition for \\_\_text\_quark\_if\_nil:nTF.)

```

\q_text_recursion_tail Internal recursion quarks.
\q_text_recursion_stop
29404 \quark_new:N \q_text_recursion_tail
29405 \quark_new:N \q_text_recursion_stop

(End definition for \q_text_recursion_tail and \q_text_recursion_stop.)
```

\_text\_use\_i\_delimit\_by\_q\_recursion\_stop:nw Functions to gobble up to a quark.

```

29406 \cs_new:Npn _text_use_i_delimit_by_q_recursion_stop:nw
29407 #1 #2 \q_text_recursion_stop {#1}

(End definition for _text_use_i_delimit_by_q_recursion_stop:nw.)
```

\\_text\_if\_recursion\_tail\_stop\_do:Nn Functions to query recursion quarks.

```

29408 _kernel_quark_new_test:N _text_if_recursion_tail_stop_do:Nn

(End definition for _text_if_recursion_tail_stop_do:Nn.)
```

## 47.2 Utilities

\\_text\_token\_to\_explicit:N

```

\text_token_to_explicit_char:N
\text_token_to_explicit_cs:N
\text_token_to_explicit_cs_aux:N
\text_token_to_explicit:n
\text_token_to_explicit_auxi:w
\text_token_to_explicit_auxii:w
\text_token_to_explicit_auxiii:w
```

\\_text\_token\_to\_explicit:n

```

\text_token_to_explicit_auxi:w
\text_token_to_explicit_auxii:w
\text_token_to_explicit_auxiii:w
```

\\_text\_token\_to\_explicit:n

```

\group_begin:
29410 \char_set_catcode_active:n { 0 }
29411 \cs_new:Npn _text_token_to_explicit:N #1
29412 {
29413 \if_catcode:w \exp_not:N #1
29414 \if_catcode:w \scan_stop: \exp_not:N #1
29415 \scan_stop:
29416 \else:
29417 \exp_not:N ^^@ \fi:
29418 \exp_after:wN _text_token_to_explicit_cs:N
29419 \else:
29420 \exp_after:wN _text_token_to_explicit_char:N
29421 \fi:
29422 #1
29423 }
29424 \group_end:
```

For control sequences, we can check for macros versus other cases using `\if_meaning:w`, then explicitly check for `\chardef` and `\mathchardef`.

```

29426 \cs_new:Npn _text_token_to_explicit_cs:N #1
29427 {
29428 \exp_after:wN \if_meaning:w \exp_not:N #1 #1
29429 \exp_after:wN \use:nn \exp_after:wN
29430 _text_token_to_explicit_cs_aux:N
29431 \else:
29432 \exp_after:wN \exp_not:n
29433 \fi:
29434 {#1}
29435 }
29436 \cs_new:Npn _text_token_to_explicit_cs_aux:N #1
29437 {
```

```

29438 \bool_lazy_or:nNTF
29439 { \token_if_chardef_p:N #1 }
29440 { \token_if_mathchardef_p:N #1 }
29441 {
29442 \char_generate:nn {#1}
29443 { \char_value_catcode:n {#1} }
29444 }
29445 {#1}
29446 }

```

For character tokens, we need to filter out the implicit characters from those that are explicit. That's done here, then if necessary we work out the category code and generate the char. To avoid issues with alignment tabs, that one is done by elimination rather than looking up the code explicitly. The trick with finding the charcode is that the TeX messages are either the *<something>* character *<char>* or the *<type>* *<char>*.

```

29447 \cs_new:Npn __text_token_to_explicit_char:N #1
29448 {
29449 \if:w
29450 \if_catcode:w ^ \exp_args:No \str_tail:n { \token_to_str:N #1 } ^
29451 \token_to_str:N #1 #1
29452 \else:
29453 AB
29454 \fi:
29455 \exp_after:wN \exp_not:n
29456 \else:
29457 \exp_after:wN __text_token_to_explicit:n
29458 \fi:
29459 {#1}
29460 }
29461 \cs_new:Npn __text_token_to_explicit:n #1
29462 {
29463 \exp_after:wN __text_token_to_explicit_auxi:w
29464 \int_value:w
29465 \if_catcode:w \c_group_begin_token #1 1 \else:
29466 \if_catcode:w \c_group_end_token #1 2 \else:
29467 \if_catcode:w \c_math_toggle_token #1 3 \else:
29468 \if_catcode:w ## #1 6 \else:
29469 \if_catcode:w ^ #1 7 \else:
29470 \if_catcode:w \c_math_subscript_token #1 8 \else:
29471 \if_catcode:w \c_space_token #1 10 \else:
29472 \if_catcode:w A #1 11 \else:
29473 \if_catcode:w + #1 12 \else:
29474 4 \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi:
29475 \exp_after:wN ;
29476 \token_to_meaning:N #1 \s__text_stop
29477 }
29478 \cs_new:Npn __text_token_to_explicit_auxi:w #1 ; #2 \s__text_stop
29479 {
29480 \char_generate:nn
29481 {
29482 \if_int_compare:w #1 < 9 \exp_stop_f:
29483 \exp_after:wN __text_token_to_explicit_auxii:w
29484 \else:
29485 \exp_after:wN __text_token_to_explicit_auxiii:w

```

```

29486 \fi:
29487 #2
29488 }
29489 {#1}
29490 }
29491 \exp_last_unbraced:NNo \cs_new:Npn __text_token_to_explicit_auxii:w
29492 #1 { \tl_to_str:n { character ~ } } { ' }
29493 \cs_new:Npn __text_token_to_explicit_auxiii:w #1 ~ #2 ~ { ' }

(End definition for __text_token_to_explicit:N and others.)

```

\\_\_text\_char\_catcode:N An idea from l3char: we need to get the category code of a specific token, not the general case.

```

29494 \cs_new:Npn __text_char_catcode:N #1
29495 {
29496 \if_catcode:w \exp_not:N #1 \c_math_toggle_token
29497 3
29498 \else:
29499 \if_catcode:w \exp_not:N #1 \c_alignment_token
29500 4
29501 \else:
29502 \if_catcode:w \exp_not:N #1 \c_math_superscript_token
29503 7
29504 \else:
29505 \if_catcode:w \exp_not:N #1 \c_math_subscript_token
29506 8
29507 \else:
29508 \if_catcode:w \exp_not:N #1 \c_space_token
29509 10
29510 \else:
29511 \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
29512 11
29513 \else:
29514 \if_catcode:w \exp_not:N #1 \c_catcode_other_token
29515 12
29516 \else:
29517 13
29518 \fi:
29519 \fi:
29520 \fi:
29521 \fi:
29522 \fi:
29523 \fi:
29524 \fi:
29525 }

```

(End definition for \\_\_text\_char\_catcode:N.)

\\_\_text\_if\_expandable:N<sub>TF</sub> Test for tokens that make sense to expand here: that is more restrictive than the engine view.

```

29526 \prg_new_conditional:Npnn __text_if_expandable:N #1 { T , F , TF }
29527 {
29528 \token_if_expandable:NTF #1
29529 {

```

```

29530 \bool_lazy_any:nTF
29531 {
29532 { \token_if_protected_macro_p:N #1 }
29533 { \token_if_protected_long_macro_p:N #1 }
29534 { \token_if_eq_meaning_p>NN \q__text_recursion_tail #1 }
29535 }
29536 { \prg_return_false: }
29537 { \prg_return_true: }
29538 }
29539 { \prg_return_false: }
29540 }

```

(End definition for `\_text_if_expandable:NTF`.)

### 47.3 Configuration variables

`\l_text_accents_tl` Special cases for accents and letter-like symbols, which in some cases will need to be converted further.  
`\l_text_letterlike_tl`

```

29541 \tl_new:N \l_text_accents_tl
29542 \tl_set:Nn \l_text_accents_tl
29543 { ` ` ^ ~ = \u \. \ " \r \H \v \d \c \k \b \t }
29544 \tl_new:N \l_text_letterlike_tl
29545 \tl_set:Nn \l_text_letterlike_tl
29546 {
29547 \AA \aa
29548 \AE \ae
29549 \DH \dh
29550 \DJ \dj
29551 \IJ \ij
29552 \L \l
29553 \NG \ng
29554 \O \o
29555 \OE \oe
29556 \SS \ss
29557 \TH \th
29558 }

```

(End definition for `\l_text_accents_tl` and `\l_text_letterlike_tl`. These variables are documented on page 263.)

`\l_text_case_exclude_arg_tl` Non-text arguments.

```

29559 \tl_new:N \l_text_case_exclude_arg_tl
29560 \tl_set:Nn \l_text_case_exclude_arg_tl { \begin \cite \end \label \ref }

```

(End definition for `\l_text_case_exclude_arg_tl`. This variable is documented on page 263.)

`\l_text_math_arg_tl` Math mode as arguments.

```

29561 \tl_new:N \l_text_math_arg_tl
29562 \tl_set:Nn \l_text_math_arg_tl { \ensuremath }

```

(End definition for `\l_text_math_arg_tl`. This variable is documented on page 263.)

`\l_text_math_delims_tl` Paired math mode delimiters.

```

29563 \tl_new:N \l_text_math_delims_tl
29564 \tl_set:Nn \l_text_math_delims_tl { $ $ \(\) }

```

(End definition for `\l_text_math_delims_t1`. This variable is documented on page 263.)

`\l_text_expand_exclude_t1` Commands which need not to expand.

```
29565 \tl_new:N \l_text_expand_exclude_t1
29566 \tl_set:Nn \l_text_expand_exclude_t1
29567 { \begin{citemize} \end{label} \ref }
```

(End definition for `\l_text_expand_exclude_t1`. This variable is documented on page 263.)

`\l__text_math_mode_t1` Used to control math mode output: internal as there is a dedicated setter.

```
29568 \tl_new:N \l__text_math_mode_t1
```

(End definition for `\l__text_math_mode_t1`.)

## 47.4 Expansion to formatted text

Markers for implicit char handling.

```
29569 \tex_chardef:D \c__text_chardef_space_token = '\ %
29570 \tex_mathchardef:D \c__text_mathchardef_space_token = '\ %
29571 \tex_chardef:D \c__text_chardef_group_begin_token = '\{ % '\}
29572 \tex_mathchardef:D \c__text_mathchardef_group_begin_token = '\{ % '\} '\{
29573 \tex_chardef:D \c__text_chardef_group_end_token = '\} % '\{
29574 \tex_mathchardef:D \c__text_mathchardef_group_end_token = '\} %
```

(End definition for `\c__text_chardef_space_token` and others.)

`\text_expand:n`

After precautions against & tokens, start a simple loop: that of course means that “text” cannot contain the two recursion quarks. The loop here must be f-type expandable; we have arbitrary user commands which might be protected *and* take arguments, and if the expansion code is used in a typesetting context, that will otherwise explode. (The same issue applies more clearly to case changing: see the example there.)

```
29575 \cs_new:Npn \text_expand:n #1
29576 {
29577 __kernel_exp_not:w \exp_after:wN
29578 {
29579 \exp:w
29580 __text_expand:n {#1}
29581 }
29582 }
29583 \cs_new:Npn __text_expand:n #1
29584 {
29585 \group_align_safe_begin:
29586 __text_expand_loop:w #1
29587 \q__text_recursion_tail \q__text_recursion_stop
29588 __text_expand_result:n { }
29589 }
```

The approach to making the code f-type expandable is to use a marker result token and to shuffle the collected tokens

```
29590 \cs_new:Npn __text_expand_store:n #1
29591 { __text_expand_store:nw {#1} }
29592 \cs_generate_variant:Nn __text_expand_store:n { o }
29593 \cs_new:Npn __text_expand_store:nw #1#2 __text_expand_result:n #3
29594 { #2 __text_expand_result:n { #3 #1 } }
```

```

29595 \cs_new:Npn __text_expand_end:w #1 __text_expand_result:n #
29596 {
29597 \group_align_safe_end:
29598 \exp_end:
29599 #2
2960 }

```

The main loop is a standard “tl action”; groups are handled recursively, while spaces are just passed through. Thus all of the action is in handling N-type tokens.

```

29601 \cs_new:Npn __text_expand_loop:w #1 \q__text_recursion_stop
29602 {
29603 \tl_if_head_is_N_type:nTF {#1}
29604 { __text_expand_N_type:N }
29605 {
29606 \tl_if_head_is_group:nTF {#1}
29607 { __text_expand_group:n }
29608 { __text_expand_space:w }
29609 }
29610 #1 \q__text_recursion_stop
29611 }
29612 \cs_new:Npn __text_expand_group:n #1
29613 {
29614 __text_expand_store:o
29615 {
29616 \exp_after:wN
29617 {
29618 \exp:w
29619 __text_expand:n {#1}
29620 }
29621 }
29622 __text_expand_loop:w
29623 }
29624 \exp_last_unbraced:NNo \cs_new:Npn __text_expand_space:w \c_space_tl
29625 {
29626 __text_expand_store:n {~}
29627 __text_expand_loop:w
29628 }

```

Before we get into the real work, we have to watch out for problematic implicit characters: spaces and grouping tokens. Converting these to explicit characters later would lead to real issues as they are *not* N-type. A space is the easy case, so it’s dealt with first: just insert the explicit token and continue the loop.

```

29629 \cs_new:Npx __text_expand_N_type:N #1
29630 {
29631 \exp_not:N __text_if_recursion_tail_stop_do:Nn #1
29632 { \exp_not:N __text_expand_end:w }
29633 \exp_not:N \bool_lazy_any:nTF
29634 {
29635 { \exp_not:N \token_if_eq_meaning_p:NN #1 \c_space_token }
29636 {
29637 \exp_not:N \token_if_eq_meaning_p:NN #1
29638 \c_text_chardef_space_token
29639 }
29640 {
29641 \exp_not:N \token_if_eq_meaning_p:NN #1

```

```

29642 \c__text_mathchardef_space_token
29643 }
29644 }
29645 { \exp_not:N __text_expand_space:w \c_space_tl }
29646 { \exp_not:N __text_expand_N_type_auxi:N #1 }
29647 }

```

Implicit {} offer two issues. First, the token could be an implicit brace character: we need to avoid turning that into a brace group, so filter out the cases manually. Then we handle the case where an implicit group is present. That is done in an “open-ended” way: there’s the possibility the closing token is hidden somewhere.

```

29648 \cs_new:Npn __text_expand_N_type_auxi:N #
29649 {
29650 \bool_lazy_or:nnTF
29651 { \token_if_eq_meaning_p:NN #1 \c__text_chardef_group_begin_token }
29652 { \token_if_eq_meaning_p:NN #1 \c__text_mathchardef_group_begin_token }
29653 {
29654 __text_expand_store:o \c_left_brace_str
29655 __text_expand_loop:w
29656 }
29657 {
29658 \bool_lazy_or:nnTF
29659 { \token_if_eq_meaning_p:NN #1 \c__text_chardef_group_end_token }
29660 { \token_if_eq_meaning_p:NN #1 \c__text_mathchardef_group_end_token }
29661 {
29662 __text_expand_store:o \c_right_brace_str
29663 __text_expand_loop:w
29664 }
29665 { __text_expand_N_type_auxii:N #1 }
29666 }
29667 }
29668 \cs_new:Npn __text_expand_N_type_auxii:N #
29669 {
29670 \token_if_eq_meaning:NNTF #1 \c_group_begin_token
29671 {
29672 { \if_false: } \fi:
29673 __text_expand_loop:w
29674 }
29675 {
29676 \token_if_eq_meaning:NNTF #1 \c_group_end_token
29677 {
29678 \if_false: { \fi: }
29679 __text_expand_loop:w
29680 }
29681 { __text_expand_N_type_auxiii:N #1 }
29682 }
29683 }

```

The first step in dealing with N-type tokens is to look for math mode material: that needs to be left alone. The starting function has to be split into two as we need \quark\_if\_recursion\_tail\_stop:N first before we can trigger the search. We then look for matching pairs of delimiters, allowing for the case where math mode starts but does not end. Within math mode, we simply pass all the tokens through unchanged, just checking the N-type ones against the end marker.

```

29684 \cs_new:Npn __text_expand_N_type_auxiii:N #1
29685 {
29686 \exp_after:wN __text_expand_math_search:NNN
29687 \exp_after:wN #1 \l_text_math_delims_tl
29688 \q_text_recursion_tail \q_text_recursion_tail
29689 \q_text_recursion_stop
29690 }
29691 \cs_new:Npn __text_expand_math_search:NNN #1#2#3
29692 {
29693 __text_if_recursion_tail_stop_do:Nn #2
29694 { __text_expand_explicit:N #1 }
29695 \token_if_eq_meaning:NNTF #1 #2
29696 {
29697 __text_use_i_delimit_by_q_recursion_stop:nw
29698 {
29699 __text_expand_store:n {#1}
29700 __text_expand_math_loop:Nw #3
29701 }
29702 }
29703 { __text_expand_math_search:NNN #1 }
29704 }
29705 \cs_new:Npn __text_expand_math_loop:Nw #1#2 \q_text_recursion_stop
29706 {
29707 \tl_if_head_is_N_type:nTF {#2}
29708 { __text_expand_math_N_type:NN }
29709 {
29710 \tl_if_head_is_group:nTF {#2}
29711 { __text_expand_math_group:Nn }
29712 { __text_expand_math_space:Nw }
29713 }
29714 #1#2 \q_text_recursion_stop
29715 }
29716 \cs_new:Npn __text_expand_math_N_type:NN #1#2
29717 {
29718 __text_if_recursion_tail_stop_do:Nn #2
29719 { __text_expand_end:w }
29720 __text_expand_store:n {#2}
29721 \token_if_eq_meaning:NNTF #2 #1
29722 { __text_expand_loop:w }
29723 { __text_expand_math_loop:Nw #1 }
29724 }
29725 \cs_new:Npn __text_expand_math_group:Nn #1#2
29726 {
29727 __text_expand_store:n { {#2} }
29728 __text_expand_math_loop:Nw #1
29729 }
29730 \exp_after:wN \cs_new:Npn \exp_after:wN __text_expand_math_space:Nw
29731 \exp_after:wN # \exp_after:wN 1 \c_space_tl
29732 {
29733 __text_expand_store:n { ~ }
29734 __text_expand_math_loop:Nw #1
29735 }

```

At this stage, either we have a control sequence or a simple character: split and handle.

```

29736 \cs_new:Npn __text_expand_explicit:N #1
29737 {
29738 \token_if_cs:NTF #1
29739 { __text_expand_exclude:N #1 }
29740 {
29741 __text_expand_store:n {#1}
29742 __text_expand_loop:w
29743 }
29744 }
```

Next we exclude math commands: this is mainly as there *might* be an `\ensuremath`. We also handle accents, which are basically the same issue but are kept separate for semantic reasons.

```

29745 \cs_new:Npn __text_expand_exclude:N #1
29746 {
29747 \exp_args:Ne __text_expand_exclude:nN
29748 {
29749 \exp_not:V \l_text_math_arg_tl
29750 \exp_not:V \l_text_accents_tl
29751 \exp_not:V \l_text_expand_exclude_tl
29752 }
29753 #1
29754 }
```

- 29755 `\cs_new:Npn \__text_expand_exclude:nN #1#2`
- 29756 {
- 29757     `\__text_expand_exclude>NN #2 #1`
- 29758     `\q_text_recursion_tail \q_text_recursion_stop`
- 29759 }
- 29760 `\cs_new:Npn \__text_expand_exclude>NN #1#2`
- 29761 {
- 29762     `\__text_if_recursion_tail_stop_do:Nn #2`
- 29763     { `\__text_expand_letterlike:N #1` }
- 29764     `\str_if_eq:nnTF {#1} {#2}`
- 29765     {
- 29766         `\__text_use_i_delimit_by_q_recursion_stop:nw`
- 29767         { `\__text_expand_exclude:Nn #1` }
- 29768     }
- 29769     { `\__text_expand_exclude>NN #1` }
- 29770 }
- 29771 `\cs_new:Npn \__text_expand_exclude:Nn #1#2`
- 29772 {
- 29773     `\__text_expand_store:n { #1 {#2} }`
- 29774     `\__text_expand_loop:w`
- 29775 }

Another list of exceptions: these ones take no arguments so are easier to handle.

```

29776 \cs_new:Npn __text_expand_letterlike:N #1
29777 {
29778 \exp_after:wN __text_expand_letterlike>NN \exp_after:wN
29779 #1 \l_text_letterlike_tl
29780 \q_text_recursion_tail \q_text_recursion_stop
29781 }
29782 \cs_new:Npn __text_expand_letterlike>NN #1#2
29783 {
29784 __text_if_recursion_tail_stop_do:Nn #2
```

```

29785 { __text_expand_cs:N #1 }
29786 \cs_if_eq:NNTF #2 #1
29787 {
29788 __text_use_i_delimit_by_q_recursion_stop:nw
29789 {
29790 __text_expand_store:n {#1}
29791 __text_expand_loop:w
29792 }
29793 }
29794 { __text_expand_letterlike>NN #1 }
29795 }

```

$\text{\LaTeX} 2\varepsilon$ 's  $\text{\protect}$  makes life interesting. Where possible, we simply remove it and replace with the “parent” command; of course, the  $\text{\protect}$  might be explicit, in which case we need to leave it alone if it's required.

```

29796 \cs_new:Npx __text_expand_cs:N #
29797 {
29798 \exp_not:N \str_if_eq:nnTF {#1} { \exp_not:N \protect }
29799 { \exp_not:N __text_expand_protect:N }
29800 {
29801 \bool_lazy_and:nnTF
29802 { \cs_if_exist_p:N \fmtname }
29803 { \str_if_eq_p:Vn \fmtname { \LaTeXe } }
29804 { \exp_not:N __text_expand_encoding:N #1 }
29805 { \exp_not:N __text_expand_replace:N #1 }
29806 }
29807 }
29808 \cs_new:Npn __text_expand_protect:N #
29809 {
29810 \exp_args:Ne __text_expand_protect:nN
29811 { \cs_to_str:N #1 } #1
29812 }
29813 \cs_new:Npn __text_expand_protect:nN #1#2
29814 { __text_expand_protect:Nw #2 #1 \q_text_nil #1 ~ \q_text_nil \q_text_nil \s_text_stop
29815 \cs_new:Npn __text_expand_protect:Nw #1 #2 ~ \q_text_nil #3 \q_text_nil #4 \s_text_stop
29816 {
29817 __text_quark_if_nil:nTF {#4}
29818 {
29819 \cs_if_exist:cTF {#2}
29820 { \exp_args:Ne __text_expand_store:n { \exp_not:c {#2} } }
29821 { __text_expand_store:n { \protect #1 } }
29822 }
29823 { __text_expand_store:n { \protect #1 } }
29824 __text_expand_loop:w
29825 }

```

Deal with encoding-specific commands

```

29826 \cs_new:Npn __text_expand_encoding:N #
29827 {
29828 \bool_lazy_or:nnTF
29829 { \cs_if_eq_p:NN #1 \@current@cmd }
29830 { \cs_if_eq_p:NN #1 \@changed@cmd }
29831 { \exp_after:wN __text_expand_loop:w __text_expand_encoding_escape>NN }
29832 { __text_expand_replace:N #1 }
29833 }

```

```

29834 \cs_new:Npn __text_expand_encoding_escape:NN #1#2 { \exp_not:n {#1} }
See if there is a dedicated replacement, and if there is, insert it.
29835 \cs_new:Npn __text_expand_replace:N #1
29836 {
29837 \bool_lazy_and:nnTF
29838 { \cs_if_exist_p:c { l__text_expand_ \token_to_str:N #1 _tl } }
29839 {
29840 \bool_lazy_or_p:nn
29841 { \token_if_cs_p:N #1 }
29842 { \token_if_active_p:N #1 }
29843 }
29844 {
29845 \exp_args:Nv __text_expand_replace:n
29846 { l__text_expand_ \token_to_str:N #1 _tl }
29847 }
29848 { __text_expand_cs_expand:N #1 }
29849 }
29850 \cs_new:Npn __text_expand_replace:n #1 { __text_expand_loop:w #1 }

Finally, expand any macros which can be: this then loops back around to deal with what
they produce. The only issue is if the token is \exp_not:n, as that must apply to the
following balanced text. There might be an \exp_after:wN there, so we check for it.
29851 \cs_new:Npn __text_expand_cs_expand:N #1
29852 {
29853 __text_if_expandable:NTF #1
29854 {
29855 \token_if_eq_meaning:NNTF #1 \exp_not:n
29856 { __text_expand_noexpand:w }
29857 { \exp_after:wN __text_expand_loop:w #1 }
29858 }
29859 {
29860 __text_expand_store:n {#1}
29861 __text_expand_loop:w
29862 }
29863 }
29864 \cs_new:Npn __text_expand_noexpand:w #1#
29865 { __text_expand_noexpand:nn {#1} }
29866 \cs_new:Npn __text_expand_noexpand:nn #1#2
29867 {
29868 #1 __text_expand_store:n #1 {#2}
29869 __text_expand_loop:w
29870 }

```

(End definition for \text\_expand:n and others. This function is documented on page 260.)

### \text\_declare\_expand\_equivalent:Nn

\text\_declare\_expand\_equivalent:cn

Create equivalents to allow replacement.

```

29871 \cs_new_protected:Npn \text_declare_expand_equivalent:Nn #1#2
29872 {
29873 \tl_clear_new:c { l__text_expand_ \token_to_str:N #1 _tl }
29874 \tl_set:cn { l__text_expand_ \token_to_str:N #1 _tl } {#2}
29875 }
29876 \cs_generate_variant:Nn \text_declare_expand_equivalent:Nn { c }

```

(End definition for \text\_declare\_expand\_equivalent:Nn. This function is documented on page 260.)

29877 </package>

## 48 I3text-case implementation

```
29878 {*package}
29879 <@=text>
```

### 48.1 Case changing

\l\_text\_titlecase\_check\_letter\_bool

Needed to determine the route used in titlecasing.

```
29880 \bool_new:N \l_text_titlecase_check_letter_bool
29881 \bool_set_true:N \l_text_titlecase_check_letter_bool
```

(End definition for \l\_text\_titlecase\_check\_letter\_bool. This variable is documented on page 263.)

```
\text_lowercase:n
\text_uppercase:n
\text_titlecase:n
\text_titlecase_first:n
\text_lowercase:nn
\text_uppercase:nn
\text_titlecase:nn
\text_titlecase_first:nn
```

The user level functions here are all wrappers around the internal functions for case changing.

```
29882 \cs_new:Npn \text_lowercase:n #1
29883 { __text_change_case:nnn { lower } { } {#1} }
29884 \cs_new:Npn \text_uppercase:n #1
29885 { __text_change_case:nnn { upper } { } {#1} }
29886 \cs_new:Npn \text_titlecase:n #1
29887 { __text_change_case:nnn { title } { } {#1} }
29888 \cs_new:Npn \text_titlecase_first:n #1
29889 { __text_change_case:nnn { titleonly } { } {#1} }
29890 \cs_new:Npn \text_lowercase:nn #1#2
29891 { __text_change_case:nnn { lower } {#1} {#2} }
29892 \cs_new:Npn \text_uppercase:nn #1#2
29893 { __text_change_case:nnn { upper } {#1} {#2} }
29894 \cs_new:Npn \text_titlecase:nn #1#2
29895 { __text_change_case:nnn { title } {#1} {#2} }
29896 \cs_new:Npn \text_titlecase_first:nn #1#2
29897 { __text_change_case:nnn { titleonly } {#1} {#2} }
```

(End definition for \text\_lowercase:n and others. These functions are documented on page 262.)

```
__text_change_case:nnn
__text_change_case_aux:nnn
__text_change_case_store:o
__text_change_case_store:V
__text_change_case_store:v
__text_change_case_store:e
__text_change_case_store:nw
__text_change_case_result:n
 __text_change_case_end:w
__text_change_case_loop:nnw
__text_change_case_break:w
 __text_change_case_group_lower:nnn
 __text_change_case_group_upper:nnn
 __text_change_case_group_title:nnn
__text_change_case_group_titleonly:nnn
 __text_change_case_space:nnw
 __text_change_case_N_type:nnN
 __text_change_case_N_type_aux:nnN
 __text_change_case_N_type:nnNN
__text_change_case_math_search:nnNNN
 __text_change_case_math_loop:nnNw
__text_change_case_math_N_type:nnNN
 __text_change_case_math_group:nnNn
__text_change_case_math_space:nnNw
 __text_change_case_CS_check:mmN
 __text_change_case_exclude:mmN
 __text_change_case_exclude:nnN
 __text_change_case_exclude:nnNN
```

As for the expansion code, the business end of case changing is the handling of N-type tokens. First, we expand the input fully (so the loops here don't need to worry about awkward look-aheads and the like). Then we split into the different paths.

The code here needs to be f-type expandable to deal with the situation where case changing is applied in running text. There, we might have case changing as a document command and the text containing other non-expandable document commands.

```
\cs_set_eq:NN \MakeLowercase \text_lowercase
...
 \MakeLowercase{\enquote*{A} text}
```

If we use an e-type expansion and wrap each token in \exp\_not:n, that would explode: the document command grabs \exp\_not:n as an argument, and things go badly wrong. So we have to wrap the entire result in exactly one \exp\_not:n, or rather in the kernel version.

```
29898 \cs_new:Npn __text_change_case:nnn #1#2#3
29899 {
2990 __kernel_exp_not:w \exp_after:wN
29901 {
29902 \exp:w
```

```

29903 \exp_args:Nn __text_change_case_aux:nnn
29904 { \text_expand:n {#3} }
29905 {#1} {#2}
29906 }
29907 }
29908 \cs_new:Npn __text_change_case_aux:nnn #1#2#3
29909 {
29910 \group_align_safe_begin:
29911 \cs_if_exist_use:c { __text_change_case_boundary_ #2 _ #3 :Nnnw }
29912 __text_change_case_loop:nnw {#2} {#3} #1
29913 \q_text_recursion_tail \q_text_recursion_stop
29914 __text_change_case_result:n { }
29915 }

```

As for expansion, collect up the tokens for future use.

```

29916 \cs_new:Npn __text_change_case_store:n #1
29917 { __text_change_case_store:nw {#1} }
29918 \cs_generate_variant:Nn __text_change_case_store:n { o , e , V , v }
29919 \cs_new:Npn __text_change_case_store:nw #1#2 __text_change_case_result:n #3
29920 { #2 __text_change_case_result:n { #3 #1 } }
29921 \cs_new:Npn __text_change_case_end:w #1 __text_change_case_result:n #2
29922 {
29923 \group_align_safe_end:
29924 \exp_end:
29925 #2
29926 }

```

The main loop is the standard `tl` action type.

```

29927 \cs_new:Npn __text_change_case_loop:nnw #1#2#3 \q_text_recursion_stop
29928 {
29929 \tl_if_head_is_N_type:nTF {#3}
29930 { __text_change_case_N_type:nnN }
29931 {
29932 \tl_if_head_is_group:nTF {#3}
29933 { \use:c { __text_change_case_group_ #1 :nnn } }
29934 { __text_change_case_space:nnw }
29935 }
29936 {#1} {#2} #3 \q_text_recursion_stop
29937 }
29938 \cs_new:Npn __text_change_case_break:w #1 \q_text_recursion_tail \q_text_recursion_stop
29939 {
29940 __text_change_case_store:n {#1}
29941 __text_change_case_end:w
29942 }

```

For a group, we *could* worry about whether this contains a character or not. However, that would make life very complex for little gain: exactly what a first character is is rather weakly-defined anyway. So if there is a group, we simply assume that a character has been seen, and for title case we switch to the “rest of the tokens” situation. To avoid having too much testing, we use a two-step process here to allow the titlecase functions to be separate.

```

29943 \cs_new:Npn __text_change_case_group_lower:nnn #1#2#3
29944 {
29945 __text_change_case_store:o
29946 {

```

```

29947 \exp_after:wN
29948 {
29949 \exp:w
29950 __text_change_case_aux:nnn {#3} {#1} {#2}
29951 }
29952 }
29953 __text_change_case_loop:nnw {#1} {#2}
29954 }
29955 \cs_new_eq:NN __text_change_case_group_upper:nnn
29956 __text_change_case_group_lower:nnn
29957 \cs_new:Npn __text_change_case_group_title:nnn #1#2#3
29958 {
29959 __text_change_case_store:o
29960 {
29961 \exp_after:wN
29962 {
29963 \exp:w
29964 __text_change_case_aux:nnn {#3} {#1} {#2}
29965 }
29966 }
29967 __text_change_case_loop:nnw { lower } {#2}
29968 }
29969 \cs_new:Npn __text_change_case_group_titleonly:nnn #1#2#3
29970 {
29971 __text_change_case_store:o
29972 {
29973 \exp_after:wN
29974 {
29975 \exp:w
29976 __text_change_case_aux:nnn {#3} {#1} {#2}
29977 }
29978 }
29979 __text_change_case_break:w
29980 }
29981 \use:x
29982 {
29983 \cs_new:Npn \exp_not:N __text_change_case_space:nnw ##1##2 \c_space_tl
29984 }
29985 {
29986 __text_change_case_store:n { ~ }
29987 \cs_if_exist_use:c { __text_change_case_boundary_ #1 _ #2 :Nnnw }
29988 __text_change_case_loop:nnw {#1} {#2}
29989 }

```

The first step of handling N-type tokens is to filter out the end-of-loop. That has to be done separately from the first real step as otherwise we pick up the wrong delimiter. The loop here is the same as the `expand` one, just passing the additional data long. If no close-math token is found then the final clean-up is forced (i.e. there is no assumption of “well-behaved” input in terms of math mode).

```

29990 \cs_new:Npn __text_change_case_N_type:nnN #1#2#3
29991 {
29992 __text_if_recursion_tail_stop_do:Nn #3
29993 { __text_change_case_end:w }
29994 __text_change_case_N_type_aux:nnN {#1} {#2} #3

```

```

29995 }
29996 \cs_new:Npn __text_change_case_N_type_aux:nnN #1#2#3
29997 {
29998 \exp_args:NV __text_change_case_N_type:nnnN
29999 \l_text_math_delims_t1 {#1} {#2} #3
30000 }
30001 \cs_new:Npn __text_change_case_N_type:nnnN #1#2#3#4
30002 {
30003 __text_change_case_math_search:nnNNN {#2} {#3} #4 #1
30004 \q_text_recursion_tail \q_text_recursion_tail
30005 \q_text_recursion_stop
30006 }
30007 \cs_new:Npn __text_change_case_math_search:nnNNN #1#2#3#4#5
30008 {
30009 __text_if_recursion_tail_stop_do:Nn #4
30010 { __text_change_case_cs_check:nnN {#1} {#2} #3 }
30011 \token_if_eq_meaning:NNTF #3 #4
30012 {
30013 __text_use_i_delimit_by_q_recursion_stop:nw
30014 {
30015 __text_change_case_store:n {#3}
30016 __text_change_case_math_loop:nnNw {#1} {#2} #5
30017 }
30018 }
30019 { __text_change_case_math_search:nnNNN {#1} {#2} #3 }
30020 }
30021 \cs_new:Npn __text_change_case_math_loop:nnNw #1#2#3#4 \q_text_recursion_stop
30022 {
30023 \tl_if_head_is_N_type:nTF {#4}
30024 { __text_change_case_math_N_type:nnNN }
30025 {
30026 \tl_if_head_is_group:nTF {#4}
30027 { __text_change_case_math_group:nnNn }
30028 { __text_change_case_math_space:nnNw }
30029 }
30030 {#1} {#2} #3 #4 \q_text_recursion_stop
30031 }
30032 \cs_new:Npn __text_change_case_math_N_type:nnNN #1#2#3#4
30033 {
30034 __text_if_recursion_tail_stop_do:Nn #4
30035 { __text_change_case_end:w }
30036 __text_change_case_store:n {#4}
30037 \token_if_eq_meaning:NNTF #4 #3
30038 { __text_change_case_loop:nnw {#1} {#2} }
30039 { __text_change_case_math_loop:nnNw {#1} {#2} #3 }
30040 }
30041 \cs_new:Npn __text_change_case_math_group:nnNn #1#2#3#4
30042 {
30043 __text_change_case_store:n { {#4} }
30044 __text_change_case_math_loop:nnNw {#1} {#2} #3
30045 }
30046 \use:x
30047 {
30048 \cs_new:Npn \exp_not:N __text_change_case_math_space:nnNw ##1##2##3

```

```

30049 \c_space_tl
30050 }
30051 {
30052 __text_change_case_store:n { ~ }
30053 __text_change_case_math_loop:nnNw {#1} {#2} #3
30054 }

```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: the two routes the code may take are then very different.

```

30055 \cs_new:Npn __text_change_case_cs_check:nnN #1#2#3
30056 {
30057 \token_if_cs:NTF #3
30058 { __text_change_case_exclude:nnN }
30059 { \use:c { __text_change_case_char_ #1 :nnN } }
30060 {#1} {#2} #3
30061 }

```

To deal with a control sequence there is first a need to test if it is on the list which indicate that case changing should be skipped. That's done using a loop as for the other special cases. If a hit is found then the argument is grabbed and passed through as-is.

```

30062 \cs_new:Npn __text_change_case_exclude:nnN #1#2#3
30063 {
30064 \exp_args:Ne __text_change_case_exclude:nnnN
30065 {
30066 \exp_not:V \l_text_math_arg_tl
30067 \exp_not:V \l_text_case_exclude_arg_tl
30068 }
30069 {#1} {#2} #3
30070 }
30071 \cs_new:Npn __text_change_case_exclude:nnnN #1#2#3#4
30072 {
30073 __text_change_case_exclude:nnNN {#2} {#3} #4 #1
30074 \q_text_recursion_tail \q_text_recursion_stop
30075 }
30076 \cs_new:Npn __text_change_case_exclude:nnNN #1#2#3#4
30077 {
30078 __text_if_recursion_tail_stop_do:Nn #4
30079 { \use:c { __text_change_case_letterlike_ #1 :nnN } {#1} {#2} #3 }
30080 \str_if_eq:nnTF {#3} {#4}
30081 {
30082 __text_use_i_delimit_by_q_recursion_stop:nw
30083 { __text_change_case_exclude:nnNn {#1} {#2} #3 }
30084 }
30085 { __text_change_case_exclude:nnNN {#1} {#2} #3 }
30086 }
30087 \cs_new:Npn __text_change_case_exclude:nnNn #1#2#3#4
30088 {
30089 __text_change_case_store:n { #3 {#4} }
30090 __text_change_case_loop:nnw {#1} {#2}
30091 }

```

Letter-like commands may still be present: they are set up using a simple lookup approach, so can easily be handled with no loop. If there is no hit, we are at the end of the process: we loop around. Letter-like chars are all available only in upper- and lowercase, so titlecasing maps to the uppercase version.

```

30092 \cs_new:Npn __text_change_case_letterlike_lower:nnN #1#2#3
30093 { __text_change_case_letterlike:nnnnN {#1} {#1} {#2} #3 }
30094 \cs_new_eq:NN __text_change_case_letterlike_upper:nnN
30095 __text_change_case_letterlike_lower:nnN
30096 \cs_new:Npn __text_change_case_letterlike_title:nnN #1#2#3
30097 { __text_change_case_letterlike:nnnnN { upper } { lower } {#1} {#2} #3 }
30098 \cs_new:Npn __text_change_case_letterlike_titleonly:nnN #1#2#3
30099 { __text_change_case_letterlike:nnnnN { upper } { end } {#1} {#2} #3 }
30100 \cs_new:Npn __text_change_case_letterlike:nnnnN #1#2#3#4#5
30101 {
30102 \cs_if_exist:cTF { c__text_ #1 case_ \token_to_str:N #5 _tl }
30103 {
30104 __text_change_case_store:v
30105 { c__text_ #1 case_ \token_to_str:N #5 _tl }
30106 \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#4}
30107 }
30108 {
30109 __text_change_case_store:n {#5}
30110 \cs_if_exist:cTF
30111 {
30112 c__text_
30113 \str_if_eq:nnTF {#1} { lower } { upper } { lower }
30114 case_ \token_to_str:N #5 _tl
30115 }
30116 { \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#4} }
30117 { __text_change_case_loop:nw {#3} {#4} }
30118 }
30119 }

```

For upper- and lowercase changes, once we get to this stage there are only a couple of questions remaining: is there a language-specific mapping and is there the special case of a terminal sigma. If not, then we pass to a simple character mapping.

```

30120 \cs_new:Npn __text_change_case_char_lower:nnN #1#2#3
30121 {
30122 \cs_if_exist_use:cF { __text_change_case_lower_ #2 :nnnN }
30123 { __text_change_case_lower_sigma:nnnN }
30124 {#1} {#1} {#2} #3
30125 }
30126 \cs_new:Npn __text_change_case_char_upper:nnN #1#2#3
30127 {
30128 \cs_if_exist_use:cF { __text_change_case_upper_ #2 :nnnN }
30129 { __text_change_case_char:nnnN }
30130 {#1} {#1} {#2} #3
30131 }

```

If the current character is an uppercase sigma, the a check is made on the next item in the input. If it is N-type and not a control sequence then there is a look-ahead phase: the logic here is simply based on letters. The one exception is Dutch: see below.

```

30132 \bool_lazy_or:nnTF
30133 { \sys_if_engine_luatex_p: }
30134 { \sys_if_engine_xetex_p: }
30135 {
30136 \cs_new:Npn __text_change_case_lower_sigma:nnnN #1#2#3#4
30137 {
30138 \int_compare:nNnTF { '#4 } = { "03A3 }

```

```

30139 { __text_change_case_lower_sigma:nnNw {#2} {#3} #4 }
30140 { __text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30141 }
30142 \cs_new:Npn __text_change_case_lower_sigma:nnNw #1#2#3#4 \q__text_recursion_stop
30143 {
30144 \tl_if_head_is_N_type:nTF {#4}
30145 { __text_change_case_lower_sigma:NnnN #3 }
30146 {
30147 __text_change_case_store:e
30148 { \char_generate:nn { "03C2 } { __text_char_catcode:N #3 } }
30149 __text_change_case_loop:nnw
30150 }
30151 {#1} {#2} #4 \q__text_recursion_stop
30152 }
30153 \cs_new:Npn __text_change_case_lower_sigma:NnnN #1#2#3#4
30154 {
30155 __text_change_case_store:e
30156 {
30157 \token_if_letter:NTF #4
30158 { \char_generate:nn { "03C3 } { __text_char_catcode:N #1 } }
30159 { \char_generate:nn { "03C2 } { __text_char_catcode:N #1 } }
30160 }
30161 __text_change_case_loop:nnw {#2} {#3} #4
30162 }
30163 }

```

In the 8-bit engines, we have to look ahead once we find the first byte of the possible hit.

```

30164 {
30165 \cs_new:Npn __text_change_case_lower_sigma:nnnN #1#2#3#4
30166 {
30167 \int_compare:nNnTF { '#4 } = { "CE" }
30168 { __text_change_case_lower_sigma:nnnNN }
30169 { __text_change_case_char:nnnN }
30170 {#1} {#2} {#3} #4
30171 }
30172 \cs_new:Npn __text_change_case_lower_sigma:nnnNN #1#2#3#4#5
30173 {
30174 \int_compare:nNnTF { '#5 } = { "A3" }
30175 { __text_change_case_lower_sigma:nnw {#2} {#3} }
30176 { __text_change_case_char:nnnN {#1} {#2} {#3} #4#5 }
30177 }
30178 \cs_new:Npn __text_change_case_lower_sigma:nnw #1#2#3 \q__text_recursion_stop
30179 {
30180 \tl_if_head_is_N_type:nTF {#3}
30181 { __text_change_case_lower_sigma:nnN }
30182 {
30183 __text_change_case_store:V \c__text_final_sigma_tl
30184 __text_change_case_loop:nnw
30185 }
30186 {#1} {#2} #3 \q__text_recursion_stop
30187 }
30188 \cs_new:Npn __text_change_case_lower_sigma:nnN #1#2#3
30189 {
30190 \bool_lazy_or:nnTF
30191 { \token_if_letter_p:N #3 }

```

```

30192 {
30193 \bool_lazy_and_p:nn
30194 { \token_if_active_p:N #3 }
30195 { \int_compare_p:nNn { '#3 } > { "80" } }
30196 }
30197 { __text_change_case_store:V \c__text_sigma_tl }
30198 { __text_change_case_store:V \c__text_final_sigma_tl }
30199 __text_change_case_loop:nnw {#1} {#2} #3
30200 }
30201 }
```

For titlecasing, we need to fully expand the new character to see if it is a letter (or active). But that means looking ahead in the 8-bit case, so we have to grab the required tokens up-front. Life is a lot easier for Unicode TeX's, where we just have one token to worry about. The one wrinkle here is that for look-ahead we'd get into trouble: luckily, only Dutch has that issue.

```

30202 \cs_new:Npx __text_change_case_char_title:nnN #1#2#3
30203 {
30204 \exp_not:N \bool_if:NTF \l_text_titlecase_check_letter_bool
30205 {
30206 \bool_lazy_or:nnTF
30207 { \sys_if_engine_luatex_p: }
30208 { \sys_if_engine_xetex_p: }
30209 { \exp_not:N \token_if_letter:NTF #3 }
30210 {
30211 \exp_not:N \bool_lazy_or:nnTF
30212 { \exp_not:N \token_if_letter_p:N #3 }
30213 { \exp_not:N \token_if_active_p:N #3 }
30214 }
30215 { \exp_not:N \use:c { __text_change_case_char_ #1 :nN } }
30216 { \exp_not:N __text_change_case_char_title:nnN { title } {#1} }
30217 }
30218 { \exp_not:N \use:c { __text_change_case_char_ #1 :nN } }
30219 {#2} #3
30220 }
30221 \cs_new_eq:NN __text_change_case_char_titleonly:nnN
30222 __text_change_case_char_title:nnN
30223 \cs_new:Npn __text_change_case_char_title:nn #1#2
30224 { __text_change_case_char_title:nnN { title } { lower } {#1} #2 }
30225 \cs_new:Npn __text_change_case_char_titleonly:nN #1#2
30226 { __text_change_case_char_title:nnN { title } { end } {#1} #2 }
30227 \cs_new:Npn __text_change_case_char_title:nnN #1#2#3#4
30228 {
30229 \cs_if_exist_use:cF { __text_change_case_title_ #3 :nnnN }
30230 {
30231 \cs_if_exist_use:cF { __text_change_case_upper_ #3 :nnnN }
30232 { __text_change_case_char:nnN }
30233 }
30234 {#1} {#2} {#3} #4
30235 }
```

For Unicode engines we can handle all characters directly. However, for the 8-bit engines the aim is to deal with (a subset of) Unicode (UTF-8) input. They deal with that by making the upper half of the range active, so we look for that and if found work out how many UTF-8 octets there are to deal with. Those can then be grabbed to reconstruct the

full Unicode character, which is then used in a lookup. (As will become obvious below, there is no intention here of covering all of Unicode.)

```

30236 \cs_new:Npn __text_change_case_char:nnnN #1#2#3#4
30237 {
30238 \token_if_active:NTF #4
30239 { __text_change_case_store:n {#4} }
30240 {
30241 __text_change_case_store:e
30242 { \use:c { char_ #1 case :N } #4 }
30243 }
30244 \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#3}
30245 }
30246 \bool_lazy_or:nnF
30247 { \sys_if_engine_luatex_p: }
30248 { \sys_if_engine_xetex_p: }
30249 {
30250 \cs_new_eq:NN __text_change_case_char_aux:nnnN
30251 __text_change_case_char:nnnN
30252 \cs_gset:Npn __text_change_case_char:nnnN #1#2#3#4
30253 {
30254 \int_compare:nNnTF { '#4 } > { "80 }
30255 {
30256 \int_compare:nNnTF { '#4 } < { "E0 }
30257 { __text_change_case_char_UTFviii:nnnNN }
30258 {
30259 \int_compare:nNnTF { '#4 } < { "F0 }
30260 { __text_change_case_char_UTFviii:nnnNNN }
30261 { __text_change_case_char_UTFviii:nnnNNNN }
30262 }
30263 {#1} {#2} {#3} #4
30264 }
30265 { __text_change_case_char_aux:nnnN {#1} {#2} {#3} #4 }
30266 }
30267 \cs_new:Npn __text_change_case_char_UTFviii:nnnNN #1#2#3#4#5
30268 { __text_change_case_char_UTFviii:nnnn {#1} {#2} {#3} {#4#5} }
30269 \cs_new:Npn __text_change_case_char_UTFviii:nnnNNN #1#2#3#4#5#6
30270 { __text_change_case_char_UTFviii:nnnn {#1} {#2} {#3} {#4#5#6} }
30271 \cs_new:Npn __text_change_case_char_UTFviii:nnnNNNNN #1#2#3#4#5#6#7
30272 { __text_change_case_char_UTFviii:nnnn {#1} {#2} {#3} {#4#5#6#7} }
30273 \cs_new:Npn __text_change_case_char_UTFviii:nnnn #1#2#3#4
30274 {
30275 \cs_if_exist:cTF { c_text_ #1 case_ \tl_to_str:n {#4} _tl }
30276 {
30277 __text_change_case_store:v
30278 { c_text_ #1 case_ \tl_to_str:n {#4} _tl }
30279 }
30280 { __text_change_case_store:n {#4} }
30281 \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#3}
30282 }
30283 }
30284 \cs_new:Npn __text_change_case_char_next_lower:nn #1#2
30285 { __text_change_case_loop:nnw {#1} {#2} }
30286 \cs_new_eq:NN __text_change_case_char_next_upper:nn
30287 __text_change_case_char_next_lower:nn

```

```

30288 \cs_new_eq:NN __text_change_case_char_next_title:nn
30289 __text_change_case_char_next_lower:nn
30290 \cs_new_eq:NN __text_change_case_char_next_titleonly:nn
30291 __text_change_case_char_next_lower:nn
30292 \cs_new:Npn __text_change_case_char_next_end:nn #1#2
30293 { __text_change_case_break:w }

```

(End definition for `\__text_change_case:nnn` and others.)

A simple alternative version for German.

```

30294 \bool_lazy_or:nnTF
30295 { \sys_if_engine_luatex_p: }
30296 { \sys_if_engine_xetex_p: }
30297 {
30298 \cs_new:cpx { __text_change_case_upper_de-alt:nnnN } #1#2#3#4
30299 {
30300 \int_compare:nNnTF { '#4 } = { "00DF }
30301 {
30302 __text_change_case_store:e
30303 { \char_generate:nn { "1E9E } { __text_change_case_catcode:N #4 } }
30304 \use:c { __text_change_case_char_next_ #2 :nn }
30305 {#2} {#3}
30306 }
30307 { __text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30308 }
30309 }
30310 {
30311 \cs_new:cpx { __text_change_case_upper_de-alt:nnnN } #1#2#3#4
30312 {
30313 \exp_not:N \int_compare:nNnTF { '#4 } = { "00C3 }
30314 {
30315 \exp_not:c { __text_change_case_upper_de-alt:nnnNN }
30316 {#1} {#2} {#3} #4
30317 }
30318 { \exp_not:N __text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30319 }
30320 \cs_new:cpx { __text_change_case_upper_de-alt:nnnNN } #1#2#3#4#5
30321 {
30322 \int_compare:nNnTF { '#5 } = { "009F }
30323 {
30324 __text_change_case_store:V \c__text_grosses_Eszett_t1
30325 \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#3}
30326 }
30327 { __text_change_case_char:nnnN {#1} {#2} {#3} #4#5 }
30328 }
30329 }

```

(End definition for `\__text_change_case_upper_de-alt:nnnN` and `\__text_change_case_upper_de-alt:nnnNN`.)

For Greek uppercasing, we need to know if characters *in the Greek range* have accents. That means doing a NFD conversion first, then starting a search. As described by the Unicode CLDR, Greek accents need to be found *after* any U+0308 (diaeresis) and are done in two groups to allow for the canonical ordering. The implementation here follows

```

__text_change_case_upper_el:nnN
__text_change_case_upper_el:nnN
__text_change_case_upper_el:mnNw
__text_change_case_upper_el:NnnN
__text_change_case_upper_el:dialytika:nnN
__text_change_case_upper_el_dialytika:N
__text_change_case_upper_el_hiatus:mnNw
__text_change_case_upper_el_hiatus:nnN
__text_change_case_upper_el_gobble:nnw
__text_change_case_upper_el_gobble:nnN
 __text_change_case_if_greek:nTF
 __text_change_case_if_greek:p:n
 __text_change_case_if_greek:nTF
__text_change_case_if_greek_accent:p:n
__text_change_case_if_greek_accent:nTF
__text_change_case_if_greek_accent:nTF

```

the data and examples from ICU (<https://sites.google.com/site/icusite/design/case/greek-upper>), although necessarily the implementation is somewhat different.

```

30330 \bool_lazy_or:nnT
30331 { \sys_if_engine_luatex_p: }
30332 { \sys_if_engine_xetex_p: }
30333 {
30334 \cs_new:Npn __text_change_case_upper_el:nnnN #1#2#3#4
30335 {
30336 __text_change_case_if_greek:nTF { '#4 }
30337 {
30338 \exp_args:Ne __text_change_case_upper_el:nnn
30339 { \char_to_nfd:N #4 } {#2} {#3}
30340 }
30341 { __text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30342 }
30343 \cs_new:Npn __text_change_case_upper_el:nnn #1#2#3
30344 { __text_change_case_upper_el:nnNw {#2} {#3} #1 }

```

At this stage we have the first NFD codepoint as #3. What we need to know is whether after that we have another character token, either from the NFD or directly in the input. If not, we store the changed character at this stage.

```

30345 \cs_new:Npn __text_change_case_upper_el:nnNw #1#2#3#4 \q__text_recursion_stop
30346 {
30347 \tl_if_head_is_N_type:nTF {#4}
30348 { __text_change_case_upper_el:NnnN #3 }
30349 {
30350 __text_change_case_store:e { \char_uppercase:N #3 }
30351 __text_change_case_loop:nnw
30352 }
30353 {#1} {#2} #4 \q__text_recursion_stop
30354 }

```

Now, we check the detail of the next codepoint: again we filter out the not-a-char cases, before checking if it's an dialytika, accent or diacritic. (The latter do not have the same hiatus behavior as accents.)

```

30355 \cs_new:Npn __text_change_case_upper_el:NnnN #1#2#3#4
30356 {
30357 \token_if_cs:NTF #4
30358 {
30359 __text_change_case_store:e { \char_uppercase:N #1 }
30360 __text_change_case_loop:nnw {#2} {#3} #4
30361 }
30362 {
30363 \int_compare:nNnTF { '#4 } = { "0308 }
30364 { __text_change_case_upper_el_dialytika:nnN {#2} {#3} #1 }
30365 {
30366 __text_change_case_if_greek_accent:nTF { '#4 }
30367 { __text_change_case_upper_el_hiatus:nnNw {#2} {#3} #1 }
30368 {
30369 __text_change_case_if_greek_diacritic:nTF { '#4 }
30370 {
30371 __text_change_case_store:e { \char_uppercase:N #1 }
30372 __text_change_case_loop:nnw {#2} {#3}
30373 }

```

```

30374 {
30375 __text_change_case_store:e { \char_uppercase:N #1 }
30376 __text_change_case_loop:nw {#2} {#3} #4
30377 }
30378 }
30379 }
30380 }
30381 }

```

We handle *dialytika* in parts as it's also needed for the hiatus. We know only two letters take it, so we can shortcut here on the second part of the tests.

```

30382 \cs_new:Npn __text_change_case_upper_el_dialytika:nnN #1#2#3
30383 {
30384 __text_change_case_if_takes_dialytika:nTF { '#3 }
30385 { __text_change_case_upper_el_dialytika:N #3 }
30386 { __text_change_case_store:e { \char_uppercase:N #3 } }
30387 __text_change_case_upper_el_gobble:nw {#1} {#2}
30388 }
30389 \cs_new:Npn __text_change_case_upper_el_dialytika:N #1
30390 {
30391 __text_change_case_store:e
30392 {
30393 \bool_lazy_or:nnTF
30394 { \int_compare_p:nNn { '#1 } = { "0399" } }
30395 { \int_compare_p:nNn { '#1 } = { "03B9" } }
30396 { \char_generate:nn { "03AA" } { __text_char_catcode:N #1 } }
30397 { \char_generate:nn { "03AB" } { __text_char_catcode:N #1 } }
30398 }
30399 }

```

Adding a hiatus needs some of the same ideas, but if there is not one we skip this code point, hence needing a separate function.

```

30400 \cs_new:Npn __text_change_case_upper_el_hiatus:nnNw
30401 #1#2#3#4 \q_text_recursion_stop
30402 {
30403 __text_change_case_store:e { \char_uppercase:N #3 }
30404 \tl_if_head_is_N_type:nTF {#4}
30405 { __text_change_case_upper_el_hiatus:nnN }
30406 { __text_change_case_loop:nw }
30407 {#1} {#2} #4 \q_text_recursion_stop
30408 }
30409 \cs_new:Npn __text_change_case_upper_el_hiatus:nnN #1#2#3
30410 {
30411 \token_if_cs:NTF #3
30412 { __text_change_case_loop:nw {#1} {#2} #3 }
30413 {
30414 __text_change_case_if_takes_dialytika:nTF { '#3 }
30415 {
30416 __text_change_case_upper_el_dialytika:N #3
30417 __text_change_case_upper_el_gobble:nw {#1} {#2}
30418 }
30419 { __text_change_case_loop:nw {#1} {#2} #3 }
30420 }
30421 }

```

For clearing out trailing combining marks after we have dealt with the first one.

```

30422 \cs_new:Npn __text_change_case_upper_el_gobble:nnw
30423 #1#2#3 \q__text_recursion_stop
30424 {
30425 \tl_if_head_is_N_type:nTF {#3}
30426 { __text_change_case_upper_el_gobble:nnN }
30427 { __text_change_case_loop:nnw }
30428 {#1} {#2} #3 \q__text_recursion_stop
30429 }
30430 \cs_new:Npn __text_change_case_upper_el_gobble:nnN #1#2#3
30431 {
30432 \bool_lazy_or:nnTF
30433 { \token_if_cs_p:N #3 }
30434 {
30435 ! \bool_lazy_or_p:nn
30436 { __text_change_case_if_greek_accent_p:n { '#3 } }
30437 { __text_change_case_if_greek_diacritic_p:n { '#3 } }
30438 }
30439 { __text_change_case_loop:nnw {#1} {#2} #3 }
30440 { __text_change_case_upper_el_gobble:nnw {#1} {#2} }
30441 }
30442 }
```

Luckily the Greek range is limited and clear.

```

30443 \prg_new_conditional:Npnn __text_change_case_if_greek:n #1 { TF }
30444 {
30445 \if_int_compare:w #1 < "0370 \exp_stop_f:
30446 \prg_return_false:
30447 \else:
30448 \if_int_compare:w #1 > "03FF \exp_stop_f:
30449 \if_int_compare:w #1 < "1F00 \exp_stop_f:
30450 \prg_return_false:
30451 \else:
30452 \if_int_compare:w #1 > "1FFF \exp_stop_f:
30453 \prg_return_false:
30454 \else:
30455 \prg_return_true:
30456 \fi:
30457 \fi:
30458 \else:
30459 \prg_return_true:
30460 \fi:
30461 }
```

We follow ICU in adding a few extras to the accent list here.

```

30463 \prg_new_conditional:Npnn __text_change_case_if_greek_accent:n #1 { TF , p }
30464 {
30465 \if_int_compare:w #1 = "0300 \exp_stop_f:
30466 \prg_return_true:
30467 \else:
30468 \if_int_compare:w #1 = "0301 \exp_stop_f:
30469 \prg_return_true:
30470 \else:
30471 \if_int_compare:w #1 = "0342 \exp_stop_f:
```

```

30472 \prg_return_true:
30473 \else:
30474 \if_int_compare:w #1 = "0302 \exp_stop_f:
30475 \prg_return_true:
30476 \else:
30477 \if_int_compare:w #1 = "0303 \exp_stop_f:
30478 \prg_return_true:
30479 \else:
30480 \if_int_compare:w #1 = "0311 \exp_stop_f:
30481 \prg_return_true:
30482 \else:
30483 \prg_return_false:
30484 \fi:
30485 \fi:
30486 \fi:
30487 \fi:
30488 \fi:
30489 \fi:
30490 }
30491 \prg_new_conditional:Npnn __text_change_case_if_greek_diacritic:n
30492 #1 { TF , p }
30493 {
30494 \if_int_compare:w #1 = "0304 \exp_stop_f:
30495 \prg_return_true:
30496 \else:
30497 \if_int_compare:w #1 = "0306 \exp_stop_f:
30498 \prg_return_true:
30499 \else:
30500 \if_int_compare:w #1 = "0313 \exp_stop_f:
30501 \prg_return_true:
30502 \else:
30503 \if_int_compare:w #1 = "0314 \exp_stop_f:
30504 \prg_return_true:
30505 \else:
30506 \if_int_compare:w #1 = "0343 \exp_stop_f:
30507 \prg_return_true:
30508 \else:
30509 \prg_return_false:
30510 \fi:
30511 \fi:
30512 \fi:
30513 \fi:
30514 \fi:
30515 }
30516 \prg_new_conditional:Npnn __text_change_case_if_takes_dialytika:n #1 { TF }
30517 {
30518 \if_int_compare:w #1 = "0399 \exp_stop_f:
30519 \prg_return_true:
30520 \else:
30521 \if_int_compare:w #1 = "03B9 \exp_stop_f:
30522 \prg_return_true:
30523 \else:
30524 \if_int_compare:w #1 = "03A5 \exp_stop_f:
30525 \prg_return_true:

```

```

30526 \else:
30527 \if_int_compare:w #1 = "03C5 \exp_stop_f:
30528 \prg_return_true:
30529 \else:
30530 \prg_return_false:
30531 \fi:
30532 \fi:
30533 \fi:
30534 \fi:
30535 }

```

(End definition for `\_text_change_case_upper_el:nnN` and others.)

There is one special case in Greek that needs to be picked up based on being an isolated letter. We do that using a test similar to final sigma, but it has to fire off from the space grabber.

```

30536 \bool_lazy_or:nnT
30537 { \sys_if_engine_luatex_p: }
30538 { \sys_if_engine_xetex_p: }
30539 {
30540 \cs_new:Npn _text_change_case_boundary_upper_el:Nnnw
30541 #1#2#3#4 \q_text_recursion_stop
30542 {
30543 \tl_if_head_is_N_type:nTF {#4}
30544 { _text_change_case_boundary_upper_el:nnN }
30545 { _text_change_case_loop:nnw }
30546 {#2} {#3} #4 \q_text_recursion_stop
30547 }
30548 \cs_new:Npn _text_change_case_boundary_upper_el:nnN #1#2#3
30549 {
30550 \bool_lazy_or:nnTF
30551 { \token_if_cs_p:N #3 }
30552 {
30553 ! \bool_lazy_or_p:nn
30554 { \int_compare_p:nNn { '#3 } = { "03AE" } }
30555 { \int_compare_p:nNn { '#3 } = { "1F22" } }
30556 }
30557 { _text_change_case_loop:nnw }
30558 { _text_change_case_boundary_upper_el:nnNw }
30559 {#1} {#2} #3
30560 }
30561 \cs_new:Npn _text_change_case_boundary_upper_el:nnNw
30562 #1#2#3#4 \q_text_recursion_stop
30563 {
30564 \tl_if_head_is_N_type:nTF {#4}
30565 { _text_change_case_boundary_upper_el:NnnN #3 }
30566 {
30567 _text_change_case_store:e
30568 { \char_generate:nn { "0389" } { _text_char_catcode:N #3 } }
30569 _text_change_case_loop:nnw
30570 }
30571 {#1} {#2} #4 \q_text_recursion_stop
30572 }
30573 \cs_new:Npn _text_change_case_boundary_upper_el:NnnN #1#2#3#4

```

```

30574 {
30575 \token_if_letter:NNTF #4
30576 { __text_change_case_loop:nw {#2} {#3} #1#4 }
30577 {
30578 __text_change_case_store:e
30579 { \char_generate:nn { "0389 } { __text_char_catcode:N #1 } }
30580 __text_change_case_loop:nw {#2} {#3} #4
30581 }
30582 }
30583 }

```

(End definition for `\__text_change_case_boundary_upper_el:Nnnw` and others.)

`\__text_change_case_title_el:mnnN`

Titlecasing retains accents, but to prevent the uppercasing code from kicking in, there has to be an explicit function here.

```

30584 \bool_lazy_or:nN
30585 { \sys_if_engine_luatex_p: }
30586 { \sys_if_engine_xetex_p: }
30587 {
30588 \cs_new:Npn __text_change_case_title_el:nnN #1#2#3#4
30589 { __text_change_case_char:nnN {#1} {#2} {#3} #4 }
30590 }

```

(End definition for `\__text_change_case_title_el:nnN`.)

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. The first step is a simple match attempt: look for the three uppercase accented letters which should gain a dot-above char in their lowercase form.

```

30591 \bool_lazy_or:nN
30592 { \sys_if_engine_luatex_p: }
30593 { \sys_if_engine_xetex_p: }
30594 {
30595 \cs_new:Npn __text_change_case_lower_lt:nnnN #1#2#3#4
30596 {
30597 \exp_args:Ne __text_change_case_lower_lt_auxi:nnnN
30598 {
30599 \int_case:nn { '#4 }
30600 {
30601 { "00CC } { "0300 }
30602 { "00CD } { "0301 }
30603 { "0128 } { "0303 }
30604 }
30605 }
30606 {#2} {#3} #4
30607 }

```

If there was a hit, output the result with the dot-above and move on. Otherwise, look for one of the three letters that can take a combining accent: I, J and I-ogonek.

```

30608 \cs_new:Npn __text_change_case_lower_lt_auxi:nnnN #1#2#3#4
30609 {
30610 \tl_if_blank:nTF {#1}
30611 {
30612 \exp_args:Ne __text_change_case_lower_lt_auxii:nnnN

```

```

30613 {
30614 \int_case:nn { '#4 }
30615 {
30616 { "0049 } { "0069 }
30617 { "004A } { "006A }
30618 { "012E } { "012F }
30619 }
30620 }
30621 {#2} {#3} #4
30622 }
30623 {
30624 __text_change_case_store:e
30625 {
30626 \char_generate:nn { "0069 } { __text_char_catcode:N #4 }
30627 \char_generate:nn { "0307 } { __text_char_catcode:N #4 }
30628 \char_generate:nn { "#1 } { __text_char_catcode:N #4 }
30629 }
30630 __text_change_case_loop:nnw {#2} {#3}
30631 }
30632 }
```

Again, branch depending on a hit. If there is one, we output the character then need to look for a combining accent: as usual, we need to be aware of the loop situation.

```

30633 \cs_new:Npn __text_change_case_lower_lt_auxii:nnnN #1#2#3#4
30634 {
30635 \tl_if_blank:nTF {#1}
30636 { __text_change_case_lower_sigma:nnnN {#2} {#2} {#3} #4 }
30637 {
30638 __text_change_case_store:e
30639 { \char_generate:nn { "#1 } { __text_char_catcode:N #4 } }
30640 __text_change_case_lower_lt:nnw {#2} {#3}
30641 }
30642 }
30643 \cs_new:Npn __text_change_case_lower_lt:nnw #1#2#3 \q__text_recursion_stop
30644 {
30645 \tl_if_head_is_N_type:nTF {#3}
30646 { __text_change_case_lower_lt:nnN }
30647 { __text_change_case_loop:nnw }
30648 { "#1 } {#2} #3 \q__text_recursion_stop
30649 }
30650 \cs_new:Npn __text_change_case_lower_lt:nnN #1#2#3
30651 {
30652 \bool_lazy_and:nnT
30653 { ! \token_if_cs_p:N #3 }
30654 {
30655 \bool_lazy_any_p:n
30656 {
30657 { \int_compare_p:nNn { '#3 } = { "0300 } }
30658 { \int_compare_p:nNn { '#3 } = { "0301 } }
30659 { \int_compare_p:nNn { '#3 } = { "0303 } }
30660 }
30661 }
30662 {
30663 __text_change_case_store:e
```

```

30664 { \char_generate:nn { "0307 } { __text_char_catcode:N #3 } }
30665 }
30666 __text_change_case_loop:nnw {#1} {#2} #3
30667 }
30668 }
```

(End definition for `\__text_change_cases_lower_lt:nnnN` and others.)

The uppercasing version: first find i/j/i-ogonek, then look for the combining char: drop it if present.

```

30669 \bool_lazy_or:nnT
30670 { \sys_if_engine_luatex_p: }
30671 { \sys_if_engine_xetex_p: }
30672 {
30673 \cs_new:Npn __text_change_case_upper_lt:nnnN #1#2#3#4
30674 {
30675 \exp_args:Nne __text_change_case_upper_lt_aux:nnnN
30676 {
30677 \int_case:nn { '#4 }
30678 {
30679 { "0069 } { "0049 }
30680 { "006A } { "004A }
30681 { "012F } { "012E }
30682 }
30683 }
30684 {#2} {#3} #4
30685 }
30686 \cs_new:Npn __text_change_case_upper_lt_aux:nnnN #1#2#3#4
30687 {
30688 \tl_if_blank:nTF {#1}
30689 { __text_change_case_char:nnnN { upper } {#2} {#3} #4 }
30690 {
30691 __text_change_case_store:e
30692 { \char_generate:nn {#1} { __text_char_catcode:N #4 } }
30693 __text_change_case_upper_lt:nnw {#2} {#3}
30694 }
30695 }
30696 \cs_new:Npn __text_change_case_upper_lt:nnw #1#2#3 \q__text_recursion_stop
30697 {
30698 \tl_if_head_is_N_type:nTF {#3}
30699 { __text_change_case_upper_lt:nnN }
30700 { \use:c { __text_change_case_char_next_ #1 :nn } }
30701 {#1} {#2} #3 \q__text_recursion_stop
30702 }
30703 \cs_new:Npn __text_change_case_upper_lt:nnN #1#2#3
30704 {
30705 \bool_lazy_and:nnTF
30706 { ! \token_if_cs_p:N #3 }
30707 { \int_compare_p:nNn { '#3 } = { "0307 } }
30708 { \use:c { __text_change_case_char_next_ #1 :nn } {#1} {#2} }
30709 { \use:c { __text_change_case_char_next_ #1 :nn } {#1} {#2} #3 }
30710 }
30711 }
```

(End definition for `\__text_change_cases_upper_lt:nnnN` and others.)

```
_text_change_case_title_nl:nnN
\text_change_case_title_nl:nnw
\text_change_case_title_nl:nnN
```

For Dutch, there is a single look-ahead test for ij when title casing. If the appropriate letters are found, produce IJ and gobble the j/J.

```
30712 \cs_new:Npn \text_change_case_title_nl:nnN #1#2#3#4
30713 {
30714 \bool_lazy_or:nnTF
30715 { \int_compare_p:nNn { '#4 } = { "0049" } }
30716 { \int_compare_p:nNn { '#4 } = { "0069" } }
30717 {
30718 \text_change_case_store:e
30719 { \char_generate:nn { "0049" } { \text_change_case_catcode:N #4 } }
30720 \text_change_case_title_nl:nnw {#2} {#3}
30721 }
30722 { \text_change_case_char:nnN {#1} {#2} {#3} #4 }
30723 }
30724 \cs_new:Npn \text_change_case_title_nl:nnw #1#2#3 \q_text_recursion_stop
30725 {
30726 \tl_if_head_is_N_type:nTF {#3}
30727 { \text_change_case_title_nl:nnN }
30728 { \use:c { \text_change_case_char_next_ #1 :nn } }
30729 {#1} {#2} #3 \q_text_recursion_stop
30730 }
30731 \cs_new:Npn \text_change_case_title_nl:nnN #1#2#3
30732 {
30733 \bool_lazy_and:nnTF
30734 { ! \token_if_cs_p:N #3 }
30735 {
30736 \bool_lazy_or_p:nn
30737 { \int_compare_p:nNn { '#3 } = { "004A" } }
30738 { \int_compare_p:nNn { '#3 } = { "006A" } }
30739 }
30740 {
30741 \text_change_case_store:e
30742 { \char_generate:nn { "004A" } { \text_change_case_catcode:N #3 } }
30743 \use:c { \text_change_case_char_next_ #1 :nn } {#1} {#2}
30744 }
30745 { \use:c { \text_change_case_char_next_ #1 :nn } {#1} {#2} #3 }
30746 }
```

(End definition for \text\_change\_case\_title\_nl:nnN, \text\_change\_case\_title\_nl:nnw, and \text\_change\_case\_title\_nl:nnN.)

```
\text_change_case_lower_tr:nnN
\text_change_case_lower_tr:nnNw
\text_change_case_lower_tr:NnnN
\text_change_case_lower_tr:nnnNN
```

The Turkic languages need special treatment for dotted-i and dotless-i. The lower casing rule can be expressed in terms of searching first for either a dotless-I or a dotted-I. In the latter case the mapping is easy, but in the former there is a second stage search.

```
30747 \bool_lazy_or:nnTF
30748 { \sys_if_engine_luatex_p: }
30749 { \sys_if_engine_xetex_p: }
30750 {
30751 \cs_new:Npn \text_change_case_lower_tr:nnnNN #1#2#3#4
30752 {
30753 \int_compare:nNnTF { '#4 } = { "0049" }
30754 { \text_change_case_lower_tr:nnNw {#1} {#3} #4 }
30755 {
30756 \int_compare:nNnTF { '#4 } = { "0130" }
```

```

30757 {
30758 __text_change_case_store:e
30759 { \char_generate:nn { "0069 } { __text_char_catcode:N #4 } }
30760 __text_change_case_loop:nw {#1} {#3}
30761 }
30762 { __text_change_case_lower_sigma:nnN {#1} {#2} {#3} #4 }
30763 }
30764 }

```

After a dotless-I there may be a dot-above character. If there is then a dotted-i should be produced, otherwise output a dotless-i. When the combination is found both the dotless-I and the dot-above char have to be removed from the input.

```

30765 \cs_new:Npn __text_change_case_lower_tr:nnNw #1#2#3#4 \q__text_recursion_stop
30766 {
30767 \tl_if_head_is_N_type:nTF {#4}
30768 { __text_change_case_lower_tr:NnnN #3 }
30769 {
30770 __text_change_case_store:e
30771 { \char_generate:nn { "0131 } { __text_char_catcode:N #3 } }
30772 __text_change_case_loop:nw
30773 }
30774 {#1} {#2} #4 \q__text_recursion_stop
30775 }
30776 \cs_new:Npn __text_change_case_lower_tr:NnnN #1#2#3#4
30777 {
30778 \bool_lazy_or:nnTF
30779 { \token_if_cs_p:N #4 }
30780 { ! \int_compare_p:nNn { '#4 } = { "0307 } }
30781 {
30782 __text_change_case_store:e
30783 { \char_generate:nn { "0131 } { __text_char_catcode:N #1 } }
30784 __text_change_case_loop:nw {#2} {#3} #4
30785 }
30786 {
30787 __text_change_case_store:e
30788 { \char_generate:nn { "0069 } { __text_char_catcode:N #1 } }
30789 __text_change_case_loop:nw {#2} {#3}
30790 }
30791 }
30792 }

```

For 8-bit engines, dot-above is not available so there is a simple test for an upper-case I. Then we can look for the UTF-8 representation of an upper case dotted-I without the combining char. If it's not there, preserve the UTF-8 sequence as-is. With 8bit engines, we cannot completely preserve category codes, so we have to make some assumptions: output a “normal” i for the dotted case. As the original character here is catcode-13, we have to make a choice about handling of i: generate a “normal” one.

```

30793 {
30794 \cs_new:Npn __text_change_case_lower_tr:nnN #1#2#3#4
30795 {
30796 \int_compare:nNnTF { '#4 } = { "0049 }
30797 {
30798 __text_change_case_store:V \c__text_dotless_i_tl
30799 __text_change_case_loop:nw {#1} {#3}

```

```

30800 }
30801 {
30802 \int_compare:nNnTF { '#4 } = { "00C4 }
30803 { __text_change_case_lower_tr:nnnNN {#1} {#2} {#3} #4 }
30804 { __text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30805 }
30806 }
30807 \cs_new:Npn __text_change_case_lower_tr:nnnNN #1#2#3#4#5
30808 {
30809 \int_compare:nNnTF { '#5 } = { "00B0 }
30810 {
30811 __text_change_case_store:e
30812 {
30813 \char_generate:nn { "0069 }
30814 { \char_value_catcode:n { "0069 } }
30815 }
30816 __text_change_case_loop:nnw {#1} {#3}
30817 }
30818 { __text_change_case_char:nnnN {#1} {#2} {#3} #4#5 }
30819 }
30820 }
```

(End definition for `\__text_change_case_lower_tr:nnnN` and others.)

`\__text_change_case_upper_tr:nnnN` Uppercasing is easier: just one exception with no context.

```

30821 \cs_new:Npx __text_change_case_upper_tr:nnnN #1#2#3#4
30822 {
30823 \exp_not:N \int_compare:nNnTF { '#4 } = { "0069 }
30824 {
30825 \bool_lazy_or:nnTF
30826 { \sys_if_engine_luatex_p: }
30827 { \sys_if_engine_xetex_p: }
30828 {
30829 \exp_not:N __text_change_case_store:e
30830 {
30831 \exp_not:N \char_generate:nn { "0130 }
30832 { \exp_not:N __text_char_catcode:N #4 }
30833 }
30834 }
30835 {
30836 \exp_not:N __text_change_case_store:V
30837 \exp_not:N \c__text_dotted_I_tl
30838 }
30839 \exp_not:N \use:c { __text_change_case_char_next_ #2 :nn } {#2} {#3}
30840 }
30841 { \exp_not:N __text_change_case_char:nnnN {#1} {#2} {#3} #4 }
30842 }
```

(End definition for `\__text_change_case_upper_tr:nnnN`.)

`\__text_change_case_lower_az:nnnN` `\__text_change_case_upper_az:nnnN` Straight copies.

```

30843 \cs_new_eq:NN __text_change_case_lower_az:nnnN
30844 __text_change_case_lower_tr:nnnN
30845 \cs_new_eq:NN __text_change_case_upper_az:nnnN
30846 __text_change_case_upper_tr:nnnN
```

(End definition for `\_text_change_case_lower_az:nnnN` and `\_text_change_case_upper_az:nnnN`.)

## 48.2 Case changing data for 8-bit engines

For cases where there is an 8-bit option in the T1 font set up, a variant is provided in both cases. There are also a few extras for LGR.

```
\c__text_dotless_i_tl
\c__text_dotted_I_tl
\c__text_i_ogonek_tl
\c__text_I_ogonek_tl
\c__text_final_sigma_tl
\c__text_sigma_tl
\c__text_grosses_Eszett_tl

30847 \group_begin:
30848 \bool_lazy_or:nNF
30849 { \sys_if_engine_luatex_p: }
30850 { \sys_if_engine_xetex_p: }
30851 {
30852 \cs_set_protected:Npn _text_tmp:w #1#2
30853 {
30854 \group_begin:
30855 \cs_set_protected:Npn _text_tmp:w ##1##2##3##4
30856 {
30857 \tl_const:Nx #1
30858 {
30859 \exp_after:wN \exp_after:wN \exp_after:wN
30860 \exp_not:N \char_generate:nn {##1} { 13 }
30861 \exp_after:wN \exp_after:wN \exp_after:wN
30862 \exp_not:N \char_generate:nn {##2} { 13 }
30863 \tl_if_blank:nF {##3}
30864 {
30865 \exp_after:wN \exp_after:wN \exp_after:wN
30866 \exp_not:N \char_generate:nn {##3} { 13 }
30867 }
30868 }
30869 }
30870 \use:x
30871 { _text_tmp:w \char_to_utfviii_bytes:n { "#2" } }
30872 \group_end:
30873 }
30874 _text_tmp:w \c__text_dotless_i_tl { 0131 }
30875 _text_tmp:w \c__text_dotted_I_tl { 0130 }
30876 _text_tmp:w \c__text_i_ogonek_tl { 012F }
30877 _text_tmp:w \c__text_I_ogonek_tl { 012E }
30878 _text_tmp:w \c__text_final_sigma_tl { 03C2 }
30879 _text_tmp:w \c__text_sigma_tl { 03C3 }
30880 _text_tmp:w \c__text_grosses_Eszett_tl { 1E9E }
30881 }
30882 \group_end:
```

(End definition for `\c__text_dotless_i_tl` and others.)

For 8-bit engines we now need to define the case-change data for the multi-octet mappings. This data is here not in the `char` module as the multi-byte nature means they are never N-type. These need a list of what code points are doable in T1 so the list is hard coded (there's no saving in loading the mappings dynamically). All of the straight-forward ones have two octets, so that is taken as read.

```
30883 \group_begin:
30884 \bool_lazy_or:nNF
30885 { \sys_if_engine_luatex_p: }
30886 { \sys_if_engine_xetex_p: }
```

```

30887 {
30888 \cs_set_protected:Npn __text_loop:nn #1#2
30889 {
30890 \quark_if_recursion_tail_stop:n {#1}
30891 \use:x
30892 {
30893 __text_tmp:w
30894 \char_to_utfviii_bytes:n { "#1" }
30895 \char_to_utfviii_bytes:n { "#2" }
30896 }
30897 __text_loop:nn
30898 }
30899 \cs_set_protected:Npn __text_tmp:nnnn #1#2#3#4#5
30900 {
30901 \tl_const:cx
30902 {
30903 c__text_ #1 case_
30904 \char_generate:nn {#2} { 12 }
30905 \char_generate:nn {#3} { 12 }
30906 _tl
30907 }
30908 {
30909 \exp_after:wN \exp_after:wN \exp_after:wN
30910 \exp_not:N \char_generate:nn {#4} { 13 }
30911 \exp_after:wN \exp_after:wN \exp_after:wN
30912 \exp_not:N \char_generate:nn {#5} { 13 }
30913 }
30914 }
30915 \cs_set_protected:Npn __text_tmp:w #1#2#3#4#5#6#7#8
30916 {
30917 \tl_const:cx
30918 {
30919 c__text_lowercase_
30920 \char_generate:nn {#1} { 12 }
30921 \char_generate:nn {#2} { 12 }
30922 _tl
30923 }
30924 {
30925 \exp_after:wN \exp_after:wN \exp_after:wN
30926 \exp_not:N \char_generate:nn {#5} { 13 }
30927 \exp_after:wN \exp_after:wN \exp_after:wN
30928 \exp_not:N \char_generate:nn {#6} { 13 }
30929 }
30930 __text_tmp:nnnn { upper } {#5} {#6} {#1} {#2}
30931 __text_tmp:nnnn { title } {#5} {#6} {#1} {#2}
30932 }
30933 __text_loop:nn
30934 { 00C0 } { 00E0 }
30935 { 00C1 } { 00E1 }
30936 { 00C2 } { 00E2 }
30937 { 00C3 } { 00E3 }
30938 { 00C4 } { 00E4 }
30939 { 00C5 } { 00E5 }
30940 { 00C6 } { 00E6 }

```

30941        { 00C7 } { 00E7 }  
30942        { 00C8 } { 00E8 }  
30943        { 00C9 } { 00E9 }  
30944        { 00CA } { 00EA }  
30945        { 00CB } { 00EB }  
30946        { 00CC } { 00EC }  
30947        { 00CD } { 00ED }  
30948        { 00CE } { 00EE }  
30949        { 00CF } { 00EF }  
30950        { 00D0 } { 00F0 }  
30951        { 00D1 } { 00F1 }  
30952        { 00D2 } { 00F2 }  
30953        { 00D3 } { 00F3 }  
30954        { 00D4 } { 00F4 }  
30955        { 00D5 } { 00F5 }  
30956        { 00D6 } { 00F6 }  
30957        { 00D8 } { 00F8 }  
30958        { 00D9 } { 00F9 }  
30959        { 00DA } { 00FA }  
30960        { 00DB } { 00FB }  
30961        { 00DC } { 00FC }  
30962        { 00DD } { 00FD }  
30963        { 00DE } { 00FE }  
30964        { 0100 } { 0101 }  
30965        { 0102 } { 0103 }  
30966        { 0104 } { 0105 }  
30967        { 0106 } { 0107 }  
30968        { 0108 } { 0109 }  
30969        { 010A } { 010B }  
30970        { 010C } { 010D }  
30971        { 010E } { 010F }  
30972        { 0110 } { 0111 }  
30973        { 0112 } { 0113 }  
30974        { 0114 } { 0115 }  
30975        { 0116 } { 0117 }  
30976        { 0118 } { 0119 }  
30977        { 011A } { 011B }  
30978        { 011C } { 011D }  
30979        { 011E } { 011F }  
30980        { 0120 } { 0121 }  
30981        { 0122 } { 0123 }  
30982        { 0124 } { 0125 }  
30983        { 0128 } { 0129 }  
30984        { 012A } { 012B }  
30985        { 012C } { 012D }  
30986        { 012E } { 012F }  
30987        { 0132 } { 0133 }  
30988        { 0134 } { 0135 }  
30989        { 0136 } { 0137 }  
30990        { 0139 } { 013A }  
30991        { 013B } { 013C }  
30992        { 013E } { 013F }  
30993        { 0141 } { 0142 }  
30994        { 0143 } { 0144 }

```

30995 { 0145 } { 0146 }
30996 { 0147 } { 0148 }
30997 { 014A } { 014B }
30998 { 014C } { 014D }
30999 { 014E } { 014F }
31000 { 0150 } { 0151 }
31001 { 0152 } { 0153 }
31002 { 0154 } { 0155 }
31003 { 0156 } { 0157 }
31004 { 0158 } { 0159 }
31005 { 015A } { 015B }
31006 { 015C } { 015D }
31007 { 015E } { 015F }
31008 { 0160 } { 0161 }
31009 { 0162 } { 0163 }
31010 { 0164 } { 0165 }
31011 { 0168 } { 0169 }
31012 { 016A } { 016B }
31013 { 016C } { 016D }
31014 { 016E } { 016F }
31015 { 0170 } { 0171 }
31016 { 0172 } { 0173 }
31017 { 0174 } { 0175 }
31018 { 0176 } { 0177 }
31019 { 0178 } { 00FF }
31020 { 0179 } { 017A }
31021 { 017B } { 017C }
31022 { 017D } { 017E }
31023 { 01CD } { 01CE }
31024 { 01CF } { 01DO }
31025 { 01D1 } { 01D2 }
31026 { 01D3 } { 01D4 }
31027 { 01E2 } { 01E3 }
31028 { 01E6 } { 01E7 }
31029 { 01E8 } { 01E9 }
31030 { 01EA } { 01EB }
31031 { 01F4 } { 01F5 }
31032 { 0218 } { 0219 }
31033 { 021A } { 021B }

```

Add T2 (Cyrillic) as this is doable using a classical \MakeUppercase approach.

```

31034 { 0400 } { 0450 }
31035 { 0401 } { 0451 }
31036 { 0402 } { 0452 }
31037 { 0403 } { 0453 }
31038 { 0404 } { 0454 }
31039 { 0405 } { 0455 }
31040 { 0406 } { 0456 }
31041 { 0407 } { 0457 }
31042 { 0408 } { 0458 }
31043 { 0409 } { 0459 }
31044 { 040A } { 045A }
31045 { 040B } { 045B }
31046 { 040C } { 045C }
31047 { 040D } { 045D }

```

|       |                   |
|-------|-------------------|
| 31048 | { 040E } { 045E } |
| 31049 | { 040F } { 045F } |
| 31050 | { 0410 } { 0430 } |
| 31051 | { 0411 } { 0431 } |
| 31052 | { 0412 } { 0432 } |
| 31053 | { 0413 } { 0433 } |
| 31054 | { 0414 } { 0434 } |
| 31055 | { 0415 } { 0435 } |
| 31056 | { 0416 } { 0436 } |
| 31057 | { 0417 } { 0437 } |
| 31058 | { 0418 } { 0438 } |
| 31059 | { 0419 } { 0439 } |
| 31060 | { 041A } { 043A } |
| 31061 | { 041B } { 043B } |
| 31062 | { 041C } { 043C } |
| 31063 | { 041D } { 043D } |
| 31064 | { 041E } { 043E } |
| 31065 | { 041F } { 043F } |
| 31066 | { 0420 } { 0440 } |
| 31067 | { 0421 } { 0441 } |
| 31068 | { 0422 } { 0442 } |
| 31069 | { 0423 } { 0443 } |
| 31070 | { 0424 } { 0444 } |
| 31071 | { 0425 } { 0445 } |
| 31072 | { 0426 } { 0446 } |
| 31073 | { 0427 } { 0447 } |
| 31074 | { 0428 } { 0448 } |
| 31075 | { 0429 } { 0449 } |
| 31076 | { 042A } { 044A } |
| 31077 | { 042B } { 044B } |
| 31078 | { 042C } { 044C } |
| 31079 | { 042D } { 044D } |
| 31080 | { 042E } { 044E } |
| 31081 | { 042F } { 044F } |

Greek support: everything in the two-octet range.

|       |                   |
|-------|-------------------|
| 31082 | { 0370 } { 0371 } |
| 31083 | { 0372 } { 0373 } |
| 31084 | { 0376 } { 0377 } |
| 31085 | { 03FD } { 037B } |
| 31086 | { 03FE } { 037C } |
| 31087 | { 03FF } { 037D } |
| 31088 | { 0386 } { 03AC } |
| 31089 | { 0388 } { 03AD } |
| 31090 | { 0389 } { 03AE } |
| 31091 | { 038A } { 03AF } |
| 31092 | { 0391 } { 03B1 } |
| 31093 | { 0392 } { 03B2 } |
| 31094 | { 0393 } { 03B3 } |
| 31095 | { 0394 } { 03B4 } |
| 31096 | { 0395 } { 03B5 } |
| 31097 | { 0396 } { 03B6 } |
| 31098 | { 0397 } { 03B7 } |
| 31099 | { 0398 } { 03B8 } |
| 31100 | { 0399 } { 03B9 } |

```

31101 { 039A } { 03BA }
31102 { 039B } { 03BB }
31103 { 039C } { 03BC }
31104 { 039D } { 03BD }
31105 { 039E } { 03BE }
31106 { 039F } { 03BF }
31107 { 03A0 } { 03C0 }
31108 { 03A1 } { 03C1 }
31109 { 03A3 } { 03C3 }
31110 { 03A4 } { 03C4 }
31111 { 03A5 } { 03C5 }
31112 { 03A6 } { 03C6 }
31113 { 03A7 } { 03C7 }
31114 { 03A8 } { 03C8 }
31115 { 03A9 } { 03C9 }
31116 { 03AA } { 03CA }
31117 { 03AB } { 03CB }
31118 { 038C } { 03CC }
31119 { 038E } { 03CD }
31120 { 038F } { 03CE }
31121 { 03CF } { 03D7 }
31122 { 03D8 } { 03D9 }
31123 { 03DA } { 03DB }
31124 { 03DC } { 03DD }
31125 { 03DE } { 03DF }
31126 { 03EO } { 03E1 }
31127 { 03E2 } { 03E3 }
31128 { 03E4 } { 03E5 }
31129 { 03E6 } { 03E7 }
31130 { 03E8 } { 03E9 }
31131 { 03EA } { 03EB }
31132 { 03EC } { 03ED }
31133 { 03EE } { 03EF }
31134 { 03F9 } { 03F2 }
31135 { 037F } { 03F3 }
31136 { 03F7 } { 03F8 }
31137 { 03FA } { 03FB }
31138 \q_recursion_tail ?
31139 \q_recursion_stop

```

Odds and ends for Greek; mainly symbols that are for compatibility, but also things like the terminal sigma. Almost all are uppercase mappings, but there is one that is not!

```

31140 \cs_set_protected:Npn __text_tmp:w #1#2#3
31141 {
31142 \group_begin:
31143 \cs_set_protected:Npn __text_tmp:w ##1##2##3##4##5##6##7##8
31144 {
31145 \tl_const:cx
31146 {
31147 c__text_ #3 case_
31148 \char_generate:nn {##1} { 12 }
31149 \char_generate:nn {##2} { 12 }
31150 _tl
31151 }

```

```

31152 {
31153 \exp_after:wN \exp_after:wN \exp_after:wN
31154 \exp_not:N \char_generate:nn {##5} { 13 }
31155 \exp_after:wN \exp_after:wN \exp_after:wN
31156 \exp_not:N \char_generate:nn {##6} { 13 }
31157 }
31158 }
31159 \use:x
31160 {
31161 _text_tmp:w
31162 \char_to_utfviii_bytes:n { "#1 }
31163 \char_to_utfviii_bytes:n { "#2 }
31164 }
31165 \group_end:
31166 }
31167 _text_tmp:w { 0345 } { 0399 } { upper }
31168 _text_tmp:w { 03C2 } { 03A3 } { upper }
31169 _text_tmp:w { 03D0 } { 0392 } { upper }
31170 _text_tmp:w { 03D1 } { 0398 } { upper }
31171 _text_tmp:w { 03D5 } { 03A6 } { upper }
31172 _text_tmp:w { 03D6 } { 03A0 } { upper }
31173 _text_tmp:w { 03F0 } { 039A } { upper }
31174 _text_tmp:w { 03F1 } { 03A1 } { upper }
31175 _text_tmp:w { 03F4 } { 03B8 } { lower }
31176 _text_tmp:w { 03F5 } { 0395 } { upper }

```

Odds and ends that are not simple one-to-one mappings. These are still two-octet code points.

```

31177 \cs_set_protected:Npn _text_tmp:w #1#2#3
31178 {
31179 \group_begin:
31180 \cs_set_protected:Npn _text_tmp:w ##1##2##3##4
31181 {
31182 \tl_const:cn
31183 {
31184 c__text_ #3 case_
31185 \char_generate:nn {##1} { 12 }
31186 \char_generate:nn {##2} { 12 }
31187 _tl
31188 }
31189 {##2}
31190 }
31191 \use:x
31192 { _text_tmp:w \char_to_utfviii_bytes:n { "#1 } }
31193 \group_end:
31194 }
31195 _text_tmp:w { 00DF } { SS } { upper }
31196 _text_tmp:w { 00DF } { Ss } { title }
31197 _text_tmp:w { 0131 } { I } { upper }

```

Greek support: the three-octet code points.

```

31198 \cs_set_protected:Npn _text_tmp:nnnnnn #1#2#3#4#5#6#7
31199 {
31200 \tl_const:cx
31201 {

```

```

31202 c__text_ #1 case_
31203 \char_generate:nn {#2} { 12 }
31204 \char_generate:nn {#3} { 12 }
31205 \char_generate:nn {#4} { 12 }
31206 _tl
31207 }
31208 {
31209 \exp_after:wN \exp_after:wN \exp_after:wN
31210 \exp_not:N \char_generate:nn {#5} { 13 }
31211 \exp_after:wN \exp_after:wN \exp_after:wN
31212 \exp_not:N \char_generate:nn {#6} { 13 }
31213 \exp_after:wN \exp_after:wN \exp_after:wN
31214 \exp_not:N \char_generate:nn {#7} { 13 }
31215 }
31216 }
31217 \cs_set_protected:Npn __text_tmp:w #1#2#3#4#5#6#7#8
31218 {
31219 \tl_const:cx
31220 {
31221 c__text_lowercase_
31222 \char_generate:nn {#1} { 12 }
31223 \char_generate:nn {#2} { 12 }
31224 \char_generate:nn {#3} { 12 }
31225 _tl
31226 }
31227 {
31228 \exp_after:wN \exp_after:wN \exp_after:wN
31229 \exp_not:N \char_generate:nn {#5} { 13 }
31230 \exp_after:wN \exp_after:wN \exp_after:wN
31231 \exp_not:N \char_generate:nn {#6} { 13 }
31232 \exp_after:wN \exp_after:wN \exp_after:wN
31233 \exp_not:N \char_generate:nn {#7} { 13 }
31234 }
31235 __text_tmp:nnnnnn { upper } {#5} {#6} {#7} {#1} {#2} {#3}
31236 __text_tmp:nnnnnn { title } {#5} {#6} {#7} {#1} {#2} {#3}
31237 }
31238 __text_loop:nn
31239 { 1F08 } { 1F00 }
31240 { 1F09 } { 1F01 }
31241 { 1FOA } { 1F02 }
31242 { 1FOB } { 1F03 }
31243 { 1FOC } { 1F04 }
31244 { 1FOD } { 1F05 }
31245 { 1FOE } { 1F06 }
31246 { 1FOF } { 1F07 }
31247 { 1F18 } { 1F10 }
31248 { 1F19 } { 1F11 }
31249 { 1F1A } { 1F12 }
31250 { 1F1B } { 1F13 }
31251 { 1F1C } { 1F14 }
31252 { 1F1D } { 1F15 }
31253 { 1F28 } { 1F20 }
31254 { 1F29 } { 1F21 }
31255 { 1F2A } { 1F22 }

```

31256 { 1F2B } { 1F23 }  
31257 { 1F2C } { 1F24 }  
31258 { 1F2D } { 1F25 }  
31259 { 1F2E } { 1F26 }  
31260 { 1F2F } { 1F27 }  
31261 { 1F38 } { 1F30 }  
31262 { 1F39 } { 1F31 }  
31263 { 1F3A } { 1F32 }  
31264 { 1F3B } { 1F33 }  
31265 { 1F3C } { 1F34 }  
31266 { 1F3D } { 1F35 }  
31267 { 1F3E } { 1F36 }  
31268 { 1F3F } { 1F37 }  
31269 { 1F48 } { 1F40 }  
31270 { 1F49 } { 1F41 }  
31271 { 1F4A } { 1F42 }  
31272 { 1F4B } { 1F43 }  
31273 { 1F4C } { 1F44 }  
31274 { 1F4D } { 1F45 }  
31275 { 1F59 } { 1F51 }  
31276 { 1F5B } { 1F53 }  
31277 { 1F5D } { 1F55 }  
31278 { 1F5F } { 1F57 }  
31279 { 1F68 } { 1F60 }  
31280 { 1F69 } { 1F61 }  
31281 { 1F6A } { 1F62 }  
31282 { 1F6B } { 1F63 }  
31283 { 1F6C } { 1F64 }  
31284 { 1F6D } { 1F65 }  
31285 { 1F6E } { 1F66 }  
31286 { 1F6F } { 1F67 }  
31287 { 1FBA } { 1F70 }  
31288 { 1FBB } { 1F71 }  
31289 { 1FC8 } { 1F72 }  
31290 { 1FC9 } { 1F73 }  
31291 { 1FCA } { 1F74 }  
31292 { 1FCB } { 1F75 }  
31293 { 1FDA } { 1F76 }  
31294 { 1FDB } { 1F77 }  
31295 { 1FF8 } { 1F78 }  
31296 { 1FF9 } { 1F79 }  
31297 { 1FEA } { 1F7A }  
31298 { 1FEB } { 1F7B }  
31299 { 1FFA } { 1F7C }  
31300 { 1FFB } { 1F7D }  
31301 { 1F88 } { 1F80 }  
31302 { 1F89 } { 1F81 }  
31303 { 1F8A } { 1F82 }  
31304 { 1F8B } { 1F83 }  
31305 { 1F8C } { 1F84 }  
31306 { 1F8D } { 1F85 }  
31307 { 1F8E } { 1F86 }  
31308 { 1F8F } { 1F87 }  
31309 { 1F98 } { 1F90 }

```

31310 { 1F99 } { 1F91 }
31311 { 1F9A } { 1F92 }
31312 { 1F9B } { 1F93 }
31313 { 1F9C } { 1F94 }
31314 { 1F9D } { 1F95 }
31315 { 1F9E } { 1F96 }
31316 { 1F9F } { 1F97 }
31317 { 1FA8 } { 1FA0 }
31318 { 1FA9 } { 1FA1 }
31319 { 1FAA } { 1FA2 }
31320 { 1FAB } { 1FA3 }
31321 { 1FAC } { 1FA4 }
31322 { 1FAD } { 1FA5 }
31323 { 1FAE } { 1FA6 }
31324 { 1FAF } { 1FA7 }
31325 { 1FB8 } { 1FB0 }
31326 { 1FB9 } { 1FB1 }
31327 { 1FBC } { 1FB3 }
31328 { 1FCC } { 1FC3 }
31329 { 1FD8 } { 1FD0 }
31330 { 1FD9 } { 1FD1 }
31331 { 1FE8 } { 1FE0 }
31332 { 1FE9 } { 1FE1 }
31333 { 1FEC } { 1FE5 }
31334 { 1FFC } { 1FF3 }
31335 \q_recursion_tail ?
31336 \q_recursion_stop

```

One three-octet special case for Greek: it also moves to two-octets!

```

31337 \cs_set_protected:Npn __text_tmp:w #1#2#3
31338 {
31339 \group_begin:
31340 \cs_set_protected:Npn __text_tmp:w ##1##2##3##4##5##6##7##8
31341 {
31342 \tl_const:cx
31343 {
31344 c__text_ #3 case_
31345 \char_generate:nn {##1} { 12 }
31346 \char_generate:nn {##2} { 12 }
31347 \char_generate:nn {##3} { 12 }
31348 _tl
31349 }
31350 {
31351 \exp_after:wN \exp_after:wN \exp_after:wN
31352 \exp_not:N \char_generate:nn {##5} { 13 }
31353 \exp_after:wN \exp_after:wN \exp_after:wN
31354 \exp_not:N \char_generate:nn {##6} { 13 }
31355 }
31356 }
31357 \use:x
31358 {
31359 __text_tmp:w
31360 \char_to_utfviii_bytes:n { "#1 }
31361 \char_to_utfviii_bytes:n { "#2 }
31362 }

```

```

31363 \group_end:
31364 }
31365 __text_tmp:w { 1FBE } { 0399 } { upper }
31366 }
31367 \group_end:

```

The (fixed) look-up mappings for letter-like control sequences.

```

31368 \group_begin:
31369 \cs_set_protected:Npn __text_change_case_setup:NN #1#2
31370 {
31371 \quark_if_recursion_tail_stop:N #1
31372 \tl_const:cn { c__text_lowercase_ \token_to_str:N #1 _tl }
31373 { #2 }
31374 \tl_const:cn { c__text_uppercase_ \token_to_str:N #2 _tl }
31375 { #1 }
31376 __text_change_case_setup:NN
31377 }
31378 __text_change_case_setup:NN
31379 \AA \aa
31380 \AE \ae
31381 \DH \dh
31382 \DJ \dj
31383 \IJ \ij
31384 \L \l
31385 \NG \ng
31386 \O \o
31387 \OE \oe
31388 \SS \ss
31389 \TH \th
31390 \q_recursion_tail ?
31391 \q_recursion_stop
31392 \tl_const:cn { c__text_uppercase_ \token_to_str:N \i _tl } { I }
31393 \tl_const:cn { c__text_uppercase_ \token_to_str:N \j _tl } { J }
31394 \group_end:

```

To deal with possible encoding-specific extensions to `\@uclclist`, we check at the end of the preamble. This will therefore only apply to L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  package mode.

```

31395 \cs_if_exist:cT { \@uclclist }
31396 {
31397 \AtBeginDocument
31398 {
31399 \group_begin:
31400 \cs_set_protected:Npn __text_change_case_setup:Nn #1#2
31401 {
31402 \quark_if_recursion_tail_stop:N #1
31403 \tl_if_single_token:nT {#2}
31404 {
31405 \cs_if_exist:cF
31406 { c__text_uppercase_ \token_to_str:N #1 _tl }
31407 {
31408 \tl_const:cn
31409 { c__text_uppercase_ \token_to_str:N #1 _tl }
31410 { #2 }
31411 }
31412 \cs_if_exist:cF

```

```
31413 { c__text_lowercase_ \token_to_str:N #2 _tl }
31414 {
31415 \tl_const:cn
31416 { c__text_lowercase_ \token_to_str:N #2 _tl }
31417 { #1 }
31418 }
31419 }
31420 _text_change_case_setup:Nn
31421 }
31422 \exp_after:wN _text_change_case_setup:Nn \@uclclist
31423 \q_recursion_tail ?
31424 \q_recursion_stop
31425 \group_end:
31426 }
31427 }
31428 </package>
```

## 49 I3text-purify implementation

31429 <\*package>  
31430 <@@=text>

## 49.1 Purifying text

\\_text\\_if\\_recursion\\_tail\\_stop:N Functions to query recursion quarks.

31431 \\_\_kernel\_quark\_new\_test:N \\_\_text\_if\_recursion\_tail\_stop:N

(End definition for `\_text_if_recursion_tail_stop:N`.)

`\text_purify:n` As in the other parts of the module, we start off with a standard “action” loop, with `_text_purify:n` expansion applied up-front.

```

__text_purify_store:n
__text_purify_store:nw
 __text_purify_end:w
 __text_purify_loop:w
 __text_purify_group:n
 __text_purify_space:w
 __text_purify_N_type:N
__text_purify_N_type_aux:n
 __text_purify_math_search:NNN
 __text_purify_math_start:NNw
__text_purify_math_store:n
__text_purify_math_store:nw
 __text_purify_math_end:w
__text_purify_math_loop:NNw
 __text_purify_math_N_type:NNN
 __text_purify_math_group:NNn
 __text_purify_math_space:NNw
__text_purify_math_cmd:N
__text_purify_math_cmd:NN
__text_purify_math_cmd:Nn
 __text_purify_replace:N
 __text_purify_replace:n
 __text_purify_expand:N
 __text_purify_protect:N

31432 \cs_new:Npn \text_purify:n #1
31433 {
31434 \kernel_exp_not:w \exp_after:wn
31435 {
31436 \exp:w
31437 \exp_args:Ne __text_purify:n
31438 { \text_expand:n {#1} }
31439 }
31440 }
31441 \cs_new:Npn __text_purify:n #1
31442 {
31443 \group_align_safe_begin:
31444 __text_purify_loop:w #1
31445 \q_text_recursion_tail \q_text_recursion_stop
31446 __text_purify_result:n { }
31447 }

As for expansion, collect up the tokens for future use.

31448 \cs_new:Npn __text_purify_store:n #1
31449 { __text_purify_store:nw {#1} }
31450 \cs_new:Npn __text_purify_store:nw #1#2 __text_puri
31451 { #2 __text_purify_result:n { #3 #1 } }
```

```

31452 \cs_new:Npn __text_purify_end:w #1 __text_purify_result:n #2
31453 {
31454 \group_align_safe_end:
31455 \exp_end:
31456 #2
31457 }

```

The main loop is a standard “tl action”. Unlike the expansion or case changing, here any groups have to be run inline. Most of the business end is as before in the N-type token processing.

```

31458 \cs_new:Npn __text_purify_loop:w #1 \q__text_recursion_stop
31459 {
31460 \tl_if_head_is_N_type:nTF {#1}
31461 { __text_purify_N_type:N }
31462 {
31463 \tl_if_head_is_group:nTF {#1}
31464 { __text_purify_group:n }
31465 { __text_purify_space:w }
31466 }
31467 #1 \q__text_recursion_stop
31468 }
31469 \cs_new:Npn __text_purify_group:n #1 { __text_purify_loop:w #1 }
31470 \exp_last_unbraced:NNo \cs_new:Npn __text_purify_space:w \c_space_tl
31471 {
31472 __text_purify_store:n { ~ }
31473 __text_purify_loop:w
31474 }

```

The first part of handling math mode is exactly the same as in the other functions: look for a start-of-math mode token and if found start a new loop tracking the closing token.

```

31475 \cs_new:Npn __text_purify_N_type:N #1
31476 {
31477 __text_if_recursion_tail_stop_do:Nn #1 { __text_purify_end:w }
31478 __text_purify_N_type_aux:N #1
31479 }
31480 \cs_new:Npn __text_purify_N_type_aux:N #1
31481 {
31482 \exp_after:wN __text_purify_math_search:NNN
31483 \exp_after:wN #1 \l_text_math_delims_tl
31484 \q__text_recursion_tail ?
31485 \q__text_recursion_stop
31486 }
31487 \cs_new:Npn __text_purify_math_search:NNN #1#2#3
31488 {
31489 __text_if_recursion_tail_stop_do:Nn #2
31490 { __text_purify_math_cmd:N #1 }
31491 \token_if_eq_meaning:NNTF #1 #2
31492 {
31493 __text_use_i_delimit_by_q_recursion_stop:nw
31494 { __text_purify_math_start>NNw #2 #3 }
31495 }
31496 { __text_purify_math_search:NNN #1 }
31497 }
31498 \cs_new:Npn __text_purify_math_start>NNw #1#2#3 \q__text_recursion_stop
31499 {

```

```

31500 __text_purify_math_loop>NNw #1#2#3 \q__text_recursion_stop
31501 __text_purify_math_result:n { }
31502 }
31503 \cs_new:Npn __text_purify_math_store:n #1
31504 { __text_purify_math_store:nw {#1} }
31505 \cs_new:Npn __text_purify_math_store:nw #1#2 __text_purify_math_result:n #3
31506 { #2 __text_purify_math_result:n { #3 #1 } }
31507 \cs_new:Npn __text_purify_math_end:w #1 __text_purify_math_result:n #2
31508 {
31509 __text_purify_store:n { $ #2 $ }
31510 __text_purify_loop:w #1
31511 }
31512 \cs_new:Npn __text_purify_math_stop:Nw #1 __text_purify_math_result:n #2
31513 {
31514 __text_purify_store:n {#1#2}
31515 __text_purify_end:w
31516 }
31517 \cs_new:Npn __text_purify_math_loop>NNw #1#2#3 \q__text_recursion_stop
31518 {
31519 \tl_if_head_is_N_type:nTF {#3}
31520 { __text_purify_math_N_type:NNN }
31521 {
31522 \tl_if_head_is_group:nTF {#3}
31523 { __text_purify_math_group:NNn }
31524 { __text_purify_math_space>NNw }
31525 }
31526 #1#2#3 \q__text_recursion_stop
31527 }
31528 \cs_new:Npn __text_purify_math_N_type:NNN #1#2#3
31529 {
31530 __text_if_recursion_tail_stop_do:Nn #3
31531 { __text_purify_math_stop:Nw #1 }
31532 \token_if_eq_meaning:NNTF #3 #2
31533 { __text_purify_math_end:w }
31534 {
31535 __text_purify_math_store:n {#3}
31536 __text_purify_math_loop>NNw #1#2
31537 }
31538 }
31539 \cs_new:Npn __text_purify_math_group:NNn #1#2#3
31540 {
31541 __text_purify_math_store:n { {#3} }
31542 __text_purify_math_loop>NNw #1#2
31543 }
31544 \exp_after:wN \cs_new:Npn \exp_after:wN __text_purify_math_space>NNw
31545 \exp_after:wN # \exp_after:wN 1
31546 \exp_after:wN # \exp_after:wN 2 \c_space_tl
31547 {
31548 __text_purify_math_store:n { ~ }
31549 __text_purify_math_loop>NNw #1#2
31550 }

```

Then handle math mode as an argument: same outcomes, different input syntax.

```

31551 \cs_new:Npn __text_purify_math_cmd:N #1
31552 {

```

```

31553 \exp_after:wN __text_purify_math_cmd:NN \exp_after:wN #1
31554 \l_text_math_arg_tl \q__text_recursion_tail \q__text_recursion_stop
31555 }
31556 \cs_new:Npn __text_purify_math_cmd:NN #1#2
31557 {
31558 __text_if_recursion_tail_stop_do:Nn #2
31559 { __text_purify_replace:N #1 }
31560 \cs_if_eq:NNTF #2 #1
31561 {
31562 __text_use_i_delimit_by_q_recursion_stop:nw
31563 { __text_purify_math_cmd:n }
31564 }
31565 { __text_purify_math_cmd:NN #1 }
31566 }
31567 \cs_new:Npn __text_purify_math_cmd:n #1
31568 { __text_purify_math_end:w __text_purify_math_result:n {#1} }

```

For N-type tokens, we first look for a string-context replacement before anything else: this can therefore cover anything. Assuming we don't find one, check to see if we can expand control sequences: if not, they have to be dropped. We also allow for L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  \protect: there's an assumption that we don't have \protect { \oops } or similar, but that's also in the expansion code and seems like a reasonable balance.

```

31569 \cs_new:Npn __text_purify_replace:N #1
31570 {
31571 \bool_lazy_and:nnTF
31572 { \cs_if_exist_p:c { __text_purify_ \token_to_str:N #1 _tl } }
31573 {
31574 \bool_lazy_or_p:nn
31575 { \token_if_cs_p:N #1 }
31576 { \token_if_active_p:N #1 }
31577 }
31578 {
31579 \exp_args:Nv __text_purify_replace:n
31580 { __text_purify_ \token_to_str:N #1 _tl }
31581 }
31582 {
31583 \token_if_cs:NTF #1
31584 { __text_purify_expand:N #1 }
31585 {
31586 \exp_args:Ne __text_purify_store:n
31587 { __text_token_to_explicit:N #1 }
31588 __text_purify_loop:w
31589 }
31590 }
31591 }
31592 \cs_new:Npn __text_purify_replace:n #1 { __text_purify_loop:w #1 }
31593 \cs_new:Npn __text_purify_expand:N #1
31594 {
31595 \str_if_eq:nnTF {#1} { \protect }
31596 { __text_purify_protect:N }
31597 {
31598 __text_if_expandable:NTF #1
31599 { \exp_after:wN __text_purify_loop:w #1 }
31600 { __text_purify_loop:w }

```

```

31601 }
31602 }
31603 \cs_new:Npn __text_purify_protect:N #1
31604 {
31605 __text_if_recursion_tail_stop_do:Nn #1 { __text_purify_end:w }
31606 __text_purify_loop:w
31607 }

```

(End definition for `\text_purify:n` and others. This function is documented on page 263.)

```

\text_declare_purify_equivalent:Nn
\text_declare_purify_equivalent:Nx
31608 \cs_new_protected:Npn \text_declare_purify_equivalent:Nn #1#2
31609 {
31610 \tl_clear_new:c { l__text_purify_ \token_to_str:N #1 _tl }
31611 \tl_set:cn { l__text_purify_ \token_to_str:N #1 _tl } {#2}
31612 }
31613 \cs_generate_variant:Nn \text_declare_purify_equivalent:Nn { Nx }

```

(End definition for `\text_declare_purify_equivalent:Nn`. This function is documented on page 263.)

Now pre-define a range of standard commands that need dedicated definitions in purified text. First handle font-related stuff: all of this needs to be disabled.

```

31614 \tl_map_inline:nn
31615 {
31616 \fontencoding
31617 \fontfamily
31618 \fontseries
31619 \fontshape
31620 }
31621 { \text_declare_purify_equivalent:Nn #1 { \use_none:n } }
31622 \text_declare_purify_equivalent:Nn \fontsize { \use_none:nn }
31623 \text_declare_purify_equivalent:Nn \selectfont { }
31624 \text_declare_purify_equivalent:Nn \usefont { \use_none:nnnn }
31625 \tl_map_inline:nn
31626 {
31627 \emph
31628 \text
31629 \textnormal
31630 \textrm
31631 \textsf
31632 \texttt
31633 \textbf
31634 \textmd
31635 \textit
31636 \textsl
31637 \textup
31638 \textsc
31639 \textulc
31640 }
31641 { \text_declare_purify_equivalent:Nn #1 { \use:n } }
31642 \tl_map_inline:nn
31643 {
31644 \normalfont
31645 \rmfamily
31646 \sffamily

```

```

31647 \ttfamily
31648 \bfseries
31649 \mdseries
31650 \itshape
31651 \scshape
31652 \slshape
31653 \upshape
31654 \em
31655 \Huge
31656 \LARGE
31657 \Large
31658 \footnotesize
31659 \huge
31660 \large
31661 \normalsize
31662 \scriptsize
31663 \small
31664 \tiny
31665 }
31666 { \text_declare_purify_equivalent:Nn #1 { } }

```

Environments have to be handled by pure expansion.

\\_\\_text\\_end\\_env:n

```

31667 \text_declare_purify_equivalent:Nn \begin { \use:c }
31668 \text_declare_purify_equivalent:Nn \end { __text_end_env:n }
31669 \cs_new:Npn __text_end_env:n #1 { \cs:w end #1 \cs_end: }

```

(End definition for \\_\\_text\\_end\\_env:n.)

Some common symbols and similar ideas.

```

31670 \text_declare_purify_equivalent:Nn \\ { }
31671 \tl_map_inline:nn
31672 { \{ \} \# \$ \% _ }
31673 { \text_declare_purify_equivalent:Nx #1 { \cs_to_str:N #1 } }

```

Cross-referencing.

```
31674 \text_declare_purify_equivalent:Nn \label { \use_none:n }
```

Spaces.

```

31675 \group_begin:
31676 \char_set_catcode_active:N \~
31677 \use:n
31678 {
31679 \group_end:
31680 \text_declare_purify_equivalent:Nx ~ { \c_space_tl }
31681 }
31682 \text_declare_purify_equivalent:Nn \nobreakspace { ~ }
31683 \text_declare_purify_equivalent:Nn \ { ~ }
31684 \text_declare_purify_equivalent:Nn \, { ~ }

```

## 49.2 Accent and letter-like data for purifying text

In contrast to case changing, both 8-bit and Unicode engines need information for text purification to handle accents and letter-like functions: these all need to be removed. However, the results are of course engine-dependent.

For the letter-like commands, life is relatively easy: they are all simply added as standard exceptions. The only oddity is \SS, which gets converted to two letters. (At some stage an alternative version can presumably be added to `babel` or similar.)

```

31685 \bool_lazy_or:nTF
31686 { \sys_if_engine_luatex_p: }
31687 { \sys_if_engine_xetex_p: }
31688 {
31689 \cs_set_protected:Npn __text_loop:Nn #1#2
31690 {
31691 \quark_if_recursion_tail_stop:N #1
31692 \text_declare_purify_equivalent:Nx #1
31693 {
31694 \char_generate:nn { "#2" }
31695 { \char_value_catcode:n { "#2" } }
31696 }
31697 __text_loop:Nn
31698 }
31699 }
31700 {
31701 \cs_set_protected:Npn __text_loop:Nn #1#2
31702 {
31703 \quark_if_recursion_tail_stop:N #1
31704 \text_declare_purify_equivalent:Nx #1
31705 {
31706 \exp_args:Ne __text_tmp:n
31707 { \char_to_utfviii_bytes:n { "#2" } }
31708 }
31709 __text_loop:Nn
31710 }
31711 \cs_set:Npn __text_tmp:n #1 { __text_tmp:nnnn #1 }
31712 \cs_set:Npn __text_tmp:nnnn #1#2#3#4
31713 {
31714 \exp_after:wN \exp_after:wN \exp_after:wN
31715 \exp_not:N \char_generate:nn {#1} { 13 }
31716 \exp_after:wN \exp_after:wN \exp_after:wN
31717 \exp_not:N \char_generate:nn {#2} { 13 }
31718 }
31719 }
31720 __text_loop:Nn
31721 \AA { 00C5 }
31722 \AE { 00C6 }
31723 \DH { 00D0 }
31724 \DJ { 0110 }
31725 \IJ { 0132 }
31726 \L { 0141 }
31727 \NG { 014A }
31728 \O { 00D8 }
31729 \OE { 0152 }
31730 \TH { 00DE }
31731 \aa { 00E5 }
31732 \ae { 00E6 }
31733 \dh { 00F0 }
31734 \dj { 0111 }
31735 \i { 0131 }

```

```

31736 \j { 0237 }
31737 \ij { 0132 }
31738 \l { 0142 }
31739 \ng { 014B }
31740 \o { 00F8 }
31741 \oe { 0153 }
31742 \ss { 00DF }
31743 \th { 00FE }
31744 \q_recursion_tail ?
31745 \q_recursion_stop
31746 \text_declare_purify_equivalent:Nn \ss { ss }

```

\\_\\_text\\_purify\\_accent:NN      Accent LICR handling is a little more complex. Accents may exist as pre-composed codepoints or as independent glyphs. The former are all saved as single token lists, whilst for the latter the combining accent needs to be re-ordered compared to the character it applies to.

```

31747 \cs_new:Npn __text_purify_accent:NN #1#2
31748 {
31749 \cs_if_exist:cTF
31750 { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
31751 {
31752 \exp_not:v
31753 { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
31754 }
31755 {
31756 \exp_not:n {#2}
31757 \exp_not:v { c__text_purify_ \token_to_str:N #1 _tl }
31758 }
31759 }
31760 \tl_map_inline:Nn \l_text_accents_tl
31761 { \text_declare_purify_equivalent:Nn #1 { __text_purify_accent:NN #1 } }

```

First set up the combining accents.

```

31762 \group_begin:
31763 \cs_set_protected:Npn __text_loop:Nn #1#2
31764 {
31765 \quark_if_recursion_tail_stop:N #1
31766 \tl_const:cx { c__text_purify_ \token_to_str:N #1 _tl }
31767 { __text_tmp:n {#2} }
31768 __text_loop:Nn
31769 }
31770 \bool_lazy_or:nnTF
31771 { \sys_if_engine_luatex_p: }
31772 { \sys_if_engine_xetex_p: }
31773 {
31774 \cs_set:Npn __text_tmp:n #1
31775 {
31776 \char_generate:nn { "#1 }
31777 { \char_value_catcode:n { "#1 } }
31778 }
31779 }
31780 {
31781 \cs_set:Npn __text_tmp:n #1
31782 {

```

```

31783 \exp_args:Nne __text_tmp_aux:n
31784 { \char_to_utfviii_bytes:n { "#1" } }
31785 }
31786 \cs_set:Npn __text_tmp_aux:n #1 { __text_tmp:nnnn #1 }
31787 \cs_set:Npn __text_tmp:nnnn #1#2#3#4
31788 {
31789 \exp_after:wN \exp_after:wN \exp_after:wN
31790 \exp_not:N \char_generate:nn {#1} { 13 }
31791 \exp_after:wN \exp_after:wN \exp_after:wN
31792 \exp_not:N \char_generate:nn {#2} { 13 }
31793 }
31794 }
31795 __text_loop:Nn
31796 \` { 0300 }
31797 \` { 0301 }
31798 \^ { 0302 }
31799 \~ { 0303 }
31800 \= { 0304 }
31801 \u { 0306 }
31802 \. { 0307 }
31803 \" { 0308 }
31804 \r { 030A }
31805 \H { 030B }
31806 \v { 030C }
31807 \d { 0323 }
31808 \c { 0327 }
31809 \k { 0328 }
31810 \b { 0331 }
31811 \t { 0361 }
31812 \q_recursion_tail { }
31813 \q_recursion_stop

```

Now we handle the pre-composed accents: the list here is taken from `puenc.def`. All of the precomposed cases take a single letter as their second argument. We do not try to cover the case where an accent is added to a “real” dotless-i or -j, or a æ/Æ. Rather, we assume that if the UTF-8 character is used, it will have the real accent character too.

```

31814 \cs_set_protected:Npn __text_loop>NNn #1#2#3
31815 {
31816 \quark_if_recursion_tail_stop:N #1
31817 \tl_const:cx
31818 { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
31819 { __text_tmp:n {#3} }
31820 __text_loop>NNn
31821 }
31822 __text_loop>NNn
31823 \` A { 00C0 }
31824 \` A { 00C1 }
31825 \^ A { 00C2 }
31826 \~ A { 00C3 }
31827 \" A { 00C4 }
31828 \r A { 00C5 }
31829 \c C { 00C7 }
31830 \` E { 00C8 }
31831 \` E { 00C9 }

```

|       |        |          |
|-------|--------|----------|
| 31832 | \^ E   | { 00CA } |
| 31833 | \" E   | { 00CB } |
| 31834 | \` I   | { 00CC } |
| 31835 | \' I   | { 00CD } |
| 31836 | \^ I   | { 00CE } |
| 31837 | \" I   | { 00CF } |
| 31838 | \~ N   | { 00D1 } |
| 31839 | \` O   | { 00D2 } |
| 31840 | \' O   | { 00D3 } |
| 31841 | \^ O   | { 00D4 } |
| 31842 | \~ O   | { 00D5 } |
| 31843 | \" O   | { 00D6 } |
| 31844 | \` U   | { 00D9 } |
| 31845 | \' U   | { 00DA } |
| 31846 | \^ U   | { 00DB } |
| 31847 | \" U   | { 00DC } |
| 31848 | \` Y   | { 00DD } |
| 31849 | \' a   | { 00E0 } |
| 31850 | \` a   | { 00E1 } |
| 31851 | \^ a   | { 00E2 } |
| 31852 | \~ a   | { 00E3 } |
| 31853 | \" a   | { 00E4 } |
| 31854 | \r a   | { 00E5 } |
| 31855 | \c c   | { 00E7 } |
| 31856 | \` e   | { 00E8 } |
| 31857 | \' e   | { 00E9 } |
| 31858 | \^ e   | { 00EA } |
| 31859 | \" e   | { 00EB } |
| 31860 | \` i   | { 00EC } |
| 31861 | \` \i  | { 00EC } |
| 31862 | \` i   | { 00ED } |
| 31863 | \` \i  | { 00ED } |
| 31864 | \^ i   | { 00EE } |
| 31865 | \^ \i  | { 00EE } |
| 31866 | \\" i  | { 00EF } |
| 31867 | \\" \i | { 00EF } |
| 31868 | \~ n   | { 00F1 } |
| 31869 | \` o   | { 00F2 } |
| 31870 | \' o   | { 00F3 } |
| 31871 | \^ o   | { 00F4 } |
| 31872 | \~ o   | { 00F5 } |
| 31873 | \\" o  | { 00F6 } |
| 31874 | \` u   | { 00F9 } |
| 31875 | \' u   | { 00FA } |
| 31876 | \^ u   | { 00FB } |
| 31877 | \\" u  | { 00FC } |
| 31878 | \` y   | { 00FD } |
| 31879 | \\" y  | { 00FF } |
| 31880 | \= A   | { 0100 } |
| 31881 | \= a   | { 0101 } |
| 31882 | \u A   | { 0102 } |
| 31883 | \u a   | { 0103 } |
| 31884 | \k A   | { 0104 } |
| 31885 | \k a   | { 0105 } |

|       |       |          |
|-------|-------|----------|
| 31886 | \' C  | { 0106 } |
| 31887 | \' c  | { 0107 } |
| 31888 | \^ C  | { 0108 } |
| 31889 | \^ c  | { 0109 } |
| 31890 | \. C  | { 010A } |
| 31891 | \. c  | { 010B } |
| 31892 | \v C  | { 010C } |
| 31893 | \v c  | { 010D } |
| 31894 | \v D  | { 010E } |
| 31895 | \v d  | { 010F } |
| 31896 | \= E  | { 0112 } |
| 31897 | \= e  | { 0113 } |
| 31898 | \u E  | { 0114 } |
| 31899 | \u e  | { 0115 } |
| 31900 | \. E  | { 0116 } |
| 31901 | \. e  | { 0117 } |
| 31902 | \k E  | { 0118 } |
| 31903 | \k e  | { 0119 } |
| 31904 | \v E  | { 011A } |
| 31905 | \v e  | { 011B } |
| 31906 | \^ G  | { 011C } |
| 31907 | \^ g  | { 011D } |
| 31908 | \u G  | { 011E } |
| 31909 | \u g  | { 011F } |
| 31910 | \. G  | { 0120 } |
| 31911 | \. g  | { 0121 } |
| 31912 | \c G  | { 0122 } |
| 31913 | \c g  | { 0123 } |
| 31914 | \^ H  | { 0124 } |
| 31915 | \^ h  | { 0125 } |
| 31916 | \~ I  | { 0128 } |
| 31917 | \~ i  | { 0129 } |
| 31918 | \~ \i | { 0129 } |
| 31919 | \= I  | { 012A } |
| 31920 | \= i  | { 012B } |
| 31921 | \= \i | { 012B } |
| 31922 | \u I  | { 012C } |
| 31923 | \u i  | { 012D } |
| 31924 | \u \i | { 012D } |
| 31925 | \k I  | { 012E } |
| 31926 | \k i  | { 012F } |
| 31927 | \k \i | { 012F } |
| 31928 | \. I  | { 0130 } |
| 31929 | \^ J  | { 0134 } |
| 31930 | \^ j  | { 0135 } |
| 31931 | \^ \j | { 0135 } |
| 31932 | \c K  | { 0136 } |
| 31933 | \c k  | { 0137 } |
| 31934 | \' L  | { 0139 } |
| 31935 | \' l  | { 013A } |
| 31936 | \c L  | { 013B } |
| 31937 | \c l  | { 013C } |
| 31938 | \v L  | { 013D } |
| 31939 | \v l  | { 013E } |

|       |      |          |
|-------|------|----------|
| 31940 | \. L | { 013F } |
| 31941 | \. l | { 0140 } |
| 31942 | \' N | { 0143 } |
| 31943 | \' n | { 0144 } |
| 31944 | \c N | { 0145 } |
| 31945 | \c n | { 0146 } |
| 31946 | \v N | { 0147 } |
| 31947 | \v n | { 0148 } |
| 31948 | \= O | { 014C } |
| 31949 | \= o | { 014D } |
| 31950 | \u O | { 014E } |
| 31951 | \u o | { 014F } |
| 31952 | \H O | { 0150 } |
| 31953 | \H o | { 0151 } |
| 31954 | \' R | { 0154 } |
| 31955 | \' r | { 0155 } |
| 31956 | \c R | { 0156 } |
| 31957 | \c r | { 0157 } |
| 31958 | \v R | { 0158 } |
| 31959 | \v r | { 0159 } |
| 31960 | \' S | { 015A } |
| 31961 | \' s | { 015B } |
| 31962 | \^ S | { 015C } |
| 31963 | \^ s | { 015D } |
| 31964 | \c S | { 015E } |
| 31965 | \c s | { 015F } |
| 31966 | \v S | { 0160 } |
| 31967 | \v s | { 0161 } |
| 31968 | \c T | { 0162 } |
| 31969 | \c t | { 0163 } |
| 31970 | \v T | { 0164 } |
| 31971 | \v t | { 0165 } |
| 31972 | \~ U | { 0168 } |
| 31973 | \~ u | { 0169 } |
| 31974 | \= U | { 016A } |
| 31975 | \= u | { 016B } |
| 31976 | \u U | { 016C } |
| 31977 | \u u | { 016D } |
| 31978 | \r U | { 016E } |
| 31979 | \r u | { 016F } |
| 31980 | \H U | { 0170 } |
| 31981 | \H u | { 0171 } |
| 31982 | \k U | { 0172 } |
| 31983 | \k u | { 0173 } |
| 31984 | \^ W | { 0174 } |
| 31985 | \^ w | { 0175 } |
| 31986 | \^ Y | { 0176 } |
| 31987 | \^ y | { 0177 } |
| 31988 | \" Y | { 0178 } |
| 31989 | \' Z | { 0179 } |
| 31990 | \' z | { 017A } |
| 31991 | \. Z | { 017B } |
| 31992 | \. z | { 017C } |
| 31993 | \v Z | { 017D } |

```

31994 \v z { 017E }
31995 \v A { 01CD }
31996 \v a { 01CE }
31997 \v I { 01CF }
31998 \v \i { 01D0 }
31999 \v i { 01D0 }
32000 \v \O { 01D1 }
32001 \v o { 01D2 }
32002 \v U { 01D3 }
32003 \v u { 01D4 }
32004 \v G { 01E6 }
32005 \v g { 01E7 }
32006 \v K { 01E8 }
32007 \v k { 01E9 }
32008 \k O { 01EA }
32009 \k o { 01EB }
32010 \v \j { 01FO }
32011 \v j { 01FO }
32012 \` G { 01F4 }
32013 \` g { 01F5 }
32014 \` N { 01F8 }
32015 \` n { 01F9 }
32016 \` \AE { 01FC }
32017 \` \ae { 01FD }
32018 \` \O { 01FE }
32019 \` \o { 01FF }
32020 \v H { 021E }
32021 \v h { 021F }
32022 \. A { 0226 }
32023 \. a { 0227 }
32024 \c E { 0228 }
32025 \c e { 0229 }
32026 \. \O { 022E }
32027 \. \o { 022F }
32028 \= Y { 0232 }
32029 \= y { 0233 }
32030 \q_recursion_tail ? { }
32031 \q_recursion_stop
32032 \group_end:

```

(End definition for `\_text_purify_accent:NN`.)

```
32033 </package>
```

## 50 I3legacy Implementation

```

32034 <*package>
32035 <@=legacy>

```

`\legacy_if_p:n` A friendly wrapper.

```

\legacy_if:nTF
32036 \prg_new_conditional:Npnn \legacy_if:n #1 { p , T , F , TF }
32037 {
32038 \exp_args:Nc \if_meaning:w { if#1 } \iftrue
32039 \prg_return_true:

```

```

32040 \else:
32041 \prg_return_false:
32042 \fi:
32043 }

```

(End definition for `\legacy_if:nTF`. This function is documented on page 264.)

```

32044
```

## 51 l3candidates Implementation

```

32045
```

### 51.1 Additions to l3box

```

32046
```

#### 51.1.1 Viewing part of a box

`\box_clip:N` A wrapper around the driver-dependent code.

```

\box_clip:c
32047 \cs_new_protected:Npn \box_clip:N #1
32048 { \hbox_set:Nn #1 { __box_backend_clip:N #1 } }
\box_gclip:N
\box_gclip:c
32049 \cs_generate_variant:Nn \box_clip:N { c }
32050 \cs_new_protected:Npn \box_gclip:N #1
32051 { \hbox_gset:Nn #1 { __box_backend_clip:N #1 } }
32052 \cs_generate_variant:Nn \box_gclip:N { c }

```

(End definition for `\box_clip:N` and `\box_gclip:N`. These functions are documented on page 265.)

`\box_set_trim:Nnnnn` Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

```

\box_set_trim:cnnnn
\box_gset_trim:Nnnnn
\box_gset_trim:cnnnn
__box_set_trim:NnnnnN
32053 \cs_new_protected:Npn \box_set_trim:Nnnnn #1#2#3#4#5
32054 { __box_set_trim:NnnnnN #1 {##2} {##3} {##4} {##5} \box_set_eq:NN }
32055 \cs_generate_variant:Nn \box_set_trim:Nnnnn { c }
32056 \cs_new_protected:Npn \box_gset_trim:Nnnnn #1#2#3#4#5
32057 { __box_set_trim:NnnnnN #1 {##2} {##3} {##4} {##5} \box_gset_eq:NN }
32058 \cs_generate_variant:Nn \box_gset_trim:Nnnnn { c }
32059 \cs_new_protected:Npn __box_set_trim:NnnnnN #1#2#3#4#5#6
32060 {
32061 \hbox_set:Nn \l__box_internal_box
32062 {
32063 \tex_kern:D - __box_dim_eval:n {##2}
32064 \box_use:N #1
32065 \tex_kern:D - __box_dim_eval:n {##4}
32066 }

```

For the height and depth, there is a need to watch the baseline is respected. Material always has to stay on the correct side, so trimming has to check that there is enough material to trim. First, the bottom edge. If there is enough depth, simply set the depth, or if not move down so the result is zero depth. `\box_move_down:nn` is used in both cases so the resulting box always contains a `\lower` primitive. The internal box is used here as it allows safe use of `\box_set_dp:Nn`.

```

32067 \dim_compare:nNnTF { \box_dp:N #1 } > {##3}
32068 {
32069 \hbox_set:Nn \l__box_internal_box

```

```

32070 {
32071 \box_move_down:nn \c_zero_dim
32072 { \box_use_drop:N \l_box_internal_box }
32073 }
32074 \box_set_dp:Nn \l_box_internal_box { \box_dp:N #1 - (#3) }
32075 }
32076 {
32077 \hbox_set:Nn \l_box_internal_box
32078 {
32079 \box_move_down:nn { (#3) - \box_dp:N #1 }
32080 { \box_use_drop:N \l_box_internal_box }
32081 }
32082 \box_set_dp:Nn \l_box_internal_box \c_zero_dim
32083 }
```

Same thing, this time from the top of the box.

```

32084 \dim_compare:nNnTF { \box_ht:N \l_box_internal_box } > {#5}
32085 {
32086 \hbox_set:Nn \l_box_internal_box
32087 {
32088 \box_move_up:nn \c_zero_dim
32089 { \box_use_drop:N \l_box_internal_box }
32090 }
32091 \box_set_ht:Nn \l_box_internal_box
32092 { \box_ht:N \l_box_internal_box - (#5) }
32093 }
32094 {
32095 \hbox_set:Nn \l_box_internal_box
32096 {
32097 \box_move_up:nn { (#5) - \box_ht:N \l_box_internal_box }
32098 { \box_use_drop:N \l_box_internal_box }
32099 }
32100 \box_set_ht:Nn \l_box_internal_box \c_zero_dim
32101 }
32102 #6 #1 \l_box_internal_box
32103 }
```

(End definition for `\box_set_trim:Nnnnn`, `\box_gset_trim:Nnnnn`, and `\_box_set_trim:NnnnnN`. These functions are documented on page 266.)

`\box_set_viewport:Nnnnn`  
`\box_set_viewport:cnnnn`

The same general logic as for the trim operation, but with absolute dimensions. As a result, there are some things to watch out for in the vertical direction.

```

32104 \cs_new_protected:Npn \box_set_viewport:Nnnnn #1#2#3#4#5
32105 { _box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
32106 \cs_generate_variant:Nn \box_set_viewport:Nnnnn { c }
32107 \cs_new_protected:Npn \box_gset_viewport:Nnnnn #1#2#3#4#5
32108 { _box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
32109 \cs_generate_variant:Nn \box_gset_viewport:Nnnnn { c }
32110 \cs_new_protected:Npn _box_set_viewport:NnnnnN #1#2#3#4#5#6
32111 {
32112 \hbox_set:Nn \l_box_internal_box
32113 {
32114 \tex_kern:D - _box_dim_eval:n {#2}
32115 \box_use:N #1
32116 \tex_kern:D _box_dim_eval:n { #4 - \box_wd:N #1 }
```

```

32117 }
32118 \dim_compare:nNnTF {#3} < \c_zero_dim
32119 {
32120 \hbox_set:Nn \l__box_internal_box
32121 {
32122 \box_move_down:nn \c_zero_dim
32123 { \box_use_drop:N \l__box_internal_box }
32124 }
32125 \box_set_dp:Nn \l__box_internal_box { - __box_dim_eval:n {#3} }
32126 }
32127 {
32128 \hbox_set:Nn \l__box_internal_box
32129 { \box_move_down:nn {#3} { \box_use_drop:N \l__box_internal_box } }
32130 \box_set_dp:Nn \l__box_internal_box \c_zero_dim
32131 }
32132 \dim_compare:nNnTF {#5} > \c_zero_dim
32133 {
32134 \hbox_set:Nn \l__box_internal_box
32135 {
32136 \box_move_up:nn \c_zero_dim
32137 { \box_use_drop:N \l__box_internal_box }
32138 }
32139 \box_set_ht:Nn \l__box_internal_box
32140 {
32141 (#5)
32142 \dim_compare:nNnT {#3} > \c_zero_dim
32143 { - (#3) }
32144 }
32145 }
32146 {
32147 \hbox_set:Nn \l__box_internal_box
32148 {
32149 \box_move_up:nn { - __box_dim_eval:n {#5} }
32150 { \box_use_drop:N \l__box_internal_box }
32151 }
32152 \box_set_ht:Nn \l__box_internal_box \c_zero_dim
32153 }
32154 #6 #1 \l__box_internal_box
32155 }

```

(End definition for `\box_set_viewport:Nnnnn`, `\box_gset_viewport:Nnnnn`, and `\__box_viewport:NnnnnN`. These functions are documented on page 266.)

## 51.2 Additions to l3flag

```
32156 <@=flag>
```

`\flag_raise_if_clear:n` It might be faster to just call the “trap” function in all cases but conceptually the function name suggests we should only run it if the flag is zero in case the “trap” made customizable in the future.

```

32157 \cs_new:Npn \flag_raise_if_clear:n #
32158 {
32159 \if_cs_exist:w flag~#1~0 \cs_end:
32160 \else:
32161 \cs:w flag~#1 \cs_end: 0 ;

```

```

32162 \fi:
32163 }

```

(End definition for `\flag_raise_if_clear:n`. This function is documented on page 267.)

### 51.3 Additions to l3msg

```
32164 ⟨@@=msg⟩
```

```
\msg_show_eval:Nn
\msg_log_eval:Nn
__msg_show_eval:nnN
```

A short-hand used for `\int_show:n` and similar functions that passes to `\tl_show:n` the result of applying #1 (a function such as `\int_eval:n`) to the expression #2. The use of f-expansion ensures that #1 is expanded in the scope in which the show command is called, rather than in the group created by `\iow_wrap:nnnN`. This is only important for expressions involving the `\currentgrouplevel` or `\currentgrouptype`. On the other hand we want the expression to be converted to a string with the usual escape character, hence within the wrapping code.

```

32165 \cs_new_protected:Npn \msg_show_eval:Nn #1#2
32166 { \exp_args:Nf __msg_show_eval:nnN { #1 {#2} } {#2} \tl_show:n }
32167 \cs_new_protected:Npn \msg_log_eval:Nn #1#2
32168 { \exp_args:Nf __msg_show_eval:nnN { #1 {#2} } {#2} \tl_log:n }
32169 \cs_new_protected:Npn __msg_show_eval:nnN #1#2#3 { #3 { #2 = #1 } }

```

(End definition for `\msg_show_eval:Nn`, `\msg_log_eval:Nn`, and `\__msg_show_eval:nnN`. These functions are documented on page 268.)

`\msg_show_item:n`  
`\msg_show_item_unbraced:n`  
`\msg_show_item:nn`  
`\msg_show_item_unbraced:nn`

Each item in the variable is formatted using one of the following functions. We cannot use `\\"` and so on because these short-hands cannot be used inside the arguments of messages, only when defining the messages.

```

32170 \cs_new:Npx \msg_show_item:n #1
32171 { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n { {#1} } }
32172 \cs_new:Npx \msg_show_item_unbraced:n #1
32173 { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n {#1} }
32174 \cs_new:Npx \msg_show_item:nn #1#2
32175 {
32176 \iow_newline: > \use:nn { ~ } { ~ }
32177 \exp_not:N \tl_to_str:n { {#1} }
32178 \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
32179 \exp_not:N \tl_to_str:n { {#2} }
32180 }
32181 \cs_new:Npx \msg_show_item_unbraced:nn #1#2
32182 {
32183 \iow_newline: > \use:nn { ~ } { ~ }
32184 \exp_not:N \tl_to_str:n {#1}
32185 \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
32186 \exp_not:N \tl_to_str:n {#2}
32187 }

```

(End definition for `\msg_show_item:n` and others. These functions are documented on page 268.)

### 51.4 Additions to l3prg

```
32188 ⟨@@=bool⟩
```

```
\bool_set_inverse:N
\bool_set_inverse:c
\bool_gset_inverse:N
\bool_gset_inverse:c
```

Set to `false` or `true` locally or globally.

```
32189 \cs_new_protected:Npn \bool_set_inverse:N #1
```

```

32190 { \bool_if:NTF #1 { \bool_set_false:N } { \bool_set_true:N } #1 }
32191 \cs_generate_variant:Nn \bool_set_inverse:N { c }
32192 \cs_new_protected:Npn \bool_gset_inverse:N #1
32193 { \bool_if:NTF #1 { \bool_gset_false:N } { \bool_gset_true:N } #1 }
32194 \cs_generate_variant:Nn \bool_gset_inverse:N { c }

```

(End definition for `\bool_set_inverse:N` and `\bool_gset_inverse:N`. These functions are documented on page 268.)

`\s__bool_mark` Internal scan marks.

`\s__bool_stop`

```

32195 \scan_new:N \s__bool_mark
32196 \scan_new:N \s__bool_stop

```

(End definition for `\s__bool_mark` and `\s__bool_stop`.)

`\bool_case_true:nTF` For boolean cases the overall idea is the same as for `\tl_case:nn(TF)` as described in l3tl.

```

32197 \cs_new:Npn \bool_case_true:nTF
32198 { \exp:w __bool_case:NnTF \c_true_bool }
32199 \cs_new:Npn \bool_case_true:nT #1#2
32200 { \exp:w __bool_case:NnTF \c_true_bool {#1} {#2} { } }
32201 \cs_new:Npn \bool_case_true:nF #1
32202 { \exp:w __bool_case:NnTF \c_true_bool {#1} { } }
32203 \cs_new:Npn \bool_case_true:n
32204 { \exp:w __bool_case:NnTF \c_true_bool {#1} { } { } }
32205 \cs_new:Npn \bool_case_false:nTF
32206 { \exp:w __bool_case:NnTF \c_false_bool }
32207 \cs_new:Npn \bool_case_false:nT #1#2
32208 { \exp:w __bool_case:NnTF \c_false_bool {#1} {#2} { } }
32209 \cs_new:Npn \bool_case_false:nF #1
32210 { \exp:w __bool_case:NnTF \c_false_bool {#1} { } }
32211 \cs_new:Npn \bool_case_false:n
32212 { \exp:w __bool_case:NnTF \c_false_bool {#1} { } { } }
32213 \cs_new:Npn __bool_case:NnTF #1#2#3#4
32214 {
32215 \bool_if:NTF #1 __bool_case_true:w __bool_case_false:w
32216 #2 #1 { } \s__bool_mark {#3} \s__bool_mark {#4} \s__bool_stop
32217 }
32218 \cs_new:Npn __bool_case_true:w #1#2
32219 {
32220 \bool_if:nTF {#1}
32221 { __bool_case_end:nw {#2} }
32222 { __bool_case_true:w }
32223 }
32224 \cs_new:Npn __bool_case_false:w #1#2
32225 {
32226 \bool_if:nTF {#1}
32227 { __bool_case_false:w }
32228 { __bool_case_end:nw {#2} }
32229 }
32230 \cs_new:Npn __bool_case_end:nw #1#2#3 \s__bool_mark #4#5 \s__bool_stop
32231 { \exp_end: #1 #4 }

```

(End definition for `\bool_case_true:nTF` and others. These functions are documented on page 269.)

## 51.5 Additions to `I3prop`

32232 `<@0=prop>`

`\_prop_use_i_delimit_by_s_stop:nw`

Functions to gobble up to a scan mark.

32233 `\cs_new:Npn \_prop_use_i_delimit_by_s_stop:nw #1 #2 \s__prop_stop {#1}`

(End definition for `\_prop_use_i_delimit_by_s_stop:nw`.)

`\prop_rand_key_value:N`

`\prop_rand_key_value:c`

`\_prop_rand_item:w`

Contrarily to `clist`, `seq` and `t1`, there is no function to get an item of a `prop` given an integer between 1 and the number of items, so we write the appropriate code. There is no bounds checking because `\int_rand:nn` is always within bounds. The initial `\int_value:w` is stopped by the first `\s__prop` in #1.

```
32234 \cs_new:Npn \prop_rand_key_value:N #1
32235 {
32236 \prop_if_empty:NF #1
32237 {
32238 \exp_after:wN _prop_rand_item:w
32239 \int_value:w \int_rand:nn { 1 } { \prop_count:N #1 }
32240 #1 \s__prop_stop
32241 }
32242 }
32243 \cs_generate_variant:Nn \prop_rand_key_value:N { c }
32244 \cs_new:Npn _prop_rand_item:w #1 \s__prop _prop_pair:wn #2 \s__prop #3
32245 {
32246 \int_compare:nNnf {#1} > 1
32247 { _prop_use_i_delimit_by_s_stop:nw { \exp_not:n { {#2} {#3} } } }
32248 \exp_after:wN _prop_rand_item:w
32249 \int_value:w \int_eval:n { #1 - 1 } \s__prop
32250 }
```

(End definition for `\prop_rand_key_value:N` and `\_prop_rand_item:w`. This function is documented on page 269.)

## 51.6 Additions to `I3seq`

32251 `<@0=seq>`

`\seq_mapthread_function:NNN`

`\seq_mapthread_function:NcN`

`\seq_mapthread_function:cNN`

`\seq_mapthread_function:ccN`

`\_seq_mapthread_function:wNN`

`\_seq_mapthread_function:wNw`

`\_seq_mapthread_function:Nnnnn`

The idea is to first expand both sequences, adding the usual `{ ? \prg_break: } { }` to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of #2 and #5, which for items in both sequences are `\s__seq \_seq_item:n`. The function to be mapped are then be applied to the two entries. When the code hits the end of one of the sequences, the break material stops the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```
32252 \cs_new:Npn \seq_mapthread_function:NNN #1#2#3
32253 { \exp_after:wN _seq_mapthread_function:wNN #2 \s__seq_stop #1 #3 }
32254 \cs_new:Npn _seq_mapthread_function:wNN \s__seq #1 \s__seq_stop #2#3
32255 {
32256 \exp_after:wN _seq_mapthread_function:wNw #2 \s__seq_stop #3
32257 #1 { ? \prg_break: } { }
32258 \prg_break_point:
32259 }
32260 \cs_new:Npn _seq_mapthread_function:wNw \s__seq #1 \s__seq_stop #2
32261 { }
```

```

32262 __seq_mapthread_function:Nnnwnn #2
32263 #1 { ? \prg_break: } { }
32264 \s__seq_stop
32265 }
32266 \cs_new:Npn __seq_mapthread_function:Nnnwnn #1#2#3#4 \s__seq_stop #5#6
32267 {
32268 \use_none:n #2
32269 \use_none:n #5
32270 #1 {#3} {#6}
32271 __seq_mapthread_function:Nnnwnn #1 #4 \s__seq_stop
32272 }
32273 \cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc , c , cc }

(End definition for \seq_mapthread_function:NNN and others. This function is documented on page 270.)

```

**\seq\\_set\\_filter:NNn**  
**\seq\\_gset\\_filter:NNn**  
**\\_\\_seq\\_set\\_filter:NNNn**

Similar to \seq\\_map\\_inline:Nn, without a \prg\\_break\\_point: because the user's code is performed within the evaluation of a boolean expression, and skipping out of that would break horribly. The \\_\\_seq\\_wrap\\_item:n function inserts the relevant \\_\\_seq\\_item:n without expansion in the input stream, hence in the x-expanding assignment.

```

32274 \cs_new_protected:Npn \seq_set_filter:NNn
32275 { __seq_set_filter:NNNn __kernel_tl_set:Nx }
32276 \cs_new_protected:Npn \seq_gset_filter:NNn
32277 { __seq_set_filter:NNNn __kernel_tl_gset:Nx }
32278 \cs_new_protected:Npn __seq_set_filter:NNNn #1#2#3#4
32279 {
32280 __seq_push_item_def:n { \bool_if:nT {#4} { __seq_wrap_item:n {##1} } }
32281 #1 #2 { #3 }
32282 __seq_pop_item_def:
32283 }

```

(End definition for \seq\\_set\\_filter:NNn, \seq\\_gset\\_filter:NNn, and \\_\\_seq\\_set\\_filter:NNNn. These functions are documented on page 270.)

Set \\_\\_seq\\_item:n then map it using the loop code.

```

32284 \cs_new_protected:Npn \seq_set_from_inline_x:Nnn
32285 { __seq_set_from_inline_x:NNnn __kernel_tl_set:Nx }
32286 \cs_new_protected:Npn \seq_gset_from_inline_x:Nnn
32287 { __seq_set_from_inline_x:NNnn __kernel_tl_gset:Nx }
32288 \cs_new_protected:Npn __seq_set_from_inline_x:NNnn #1#2#3#4
32289 {
32290 __seq_push_item_def:n { \exp_not:N __seq_item:n {#4} }
32291 #1 #2 { \s__seq #3 __seq_item:n }
32292 __seq_pop_item_def:
32293 }

```

(End definition for \seq\\_set\\_from\\_inline\\_x:Nnn, \seq\\_gset\\_from\\_inline\\_x:Nnn, and \\_\\_seq\\_set\\_from\\_inline\\_x:NNnn. These functions are documented on page 270.)

Reuse \seq\\_set\\_from\\_inline\\_x:Nnn.

```

32294 \cs_new_protected:Npn \seq_set_from_function:NnN #1#2#3
32295 { \seq_set_from_inline_x:Nnn #1 {#2} { #3 {##1} } }
32296 \cs_new_protected:Npn \seq_gset_from_function:NnN #1#2#3
32297 { \seq_gset_from_inline_x:Nnn #1 {#2} { #3 {##1} } }

```

(End definition for \seq\\_set\\_from\\_function:NnN and \seq\\_gset\\_from\\_function:NnN. These functions are documented on page 270.)

## 51.7 Additions to l3sys

32298 ⟨@@=sys⟩

\c\_sys\_engine\_version\_str Various different engines, various different ways to extract the data!

```
32299 \str_const:Nx \c_sys_engine_version_str
32300 {
32301 \str_case:on \c_sys_engine_str
32302 {
32303 { pdftex }
32304 {
32305 \fp_eval:n { round(\int_use:N \tex_pdftexversion:D / 100 , 2) }
32306 .
32307 \tex_pdftexrevision:D
32308 }
32309 { ptx }
32310 {
32311 \cs_if_exist:NT \tex_ptexversion:D
32312 {
32313 p
32314 \int_use:N \tex_ptexversion:D
32315 .
32316 \int_use:N \tex_ptexminorversion:D
32317 \tex_ptexrevision:D
32318 -
32319 \int_use:N \tex_epTeXversion:D
32320 }
32321 }
32322 { luatex }
32323 {
32324 \fp_eval:n { round(\int_use:N \tex_luatexversion:D / 100, 2) }
32325 .
32326 \tex_luatexrevision:D
32327 }
32328 { uptex }
32329 {
32330 \cs_if_exist:NT \tex_ptexversion:D
32331 {
32332 p
32333 \int_use:N \tex_ptexversion:D
32334 .
32335 \int_use:N \tex_ptexminorversion:D
32336 \tex_ptexrevision:D
32337 -
32338 u
32339 \int_use:N \tex_uptexversion:D
32340 \tex_uptexrevision:D
32341 -
32342 \int_use:N \tex_epTeXversion:D
32343 }
32344 }
32345 { xetex }
32346 {
32347 \int_use:N \tex_XeTeXversion:D
32348 \tex_XeTeXrevision:D
```

```

32349 }
32350 }
32351 }
```

(End definition for `\c_sys_engine_version_str`. This variable is documented on page [271](#).)

## 51.8 Additions to `I3file`

`\ior_shell_open:Nn`  
`\__ior_shell_open:nN`

Actually much easier than either the standard open or input versions! When calling `\__kernel_ior_open:Nn` the file the pipe is added to signal a shell command, but the quotes are not added yet—they are added later by `\__kernel_file_name_quote:n`.

```

32353 \cs_new_protected:Npn \ior_shell_open:Nn #1#2
32354 {
32355 \sys_if_shell:TF
32356 { \exp_args:No __ior_shell_open:nN { \tl_to_str:n {#2} } #1 }
32357 { __kernel_msg_error:nn { kernel } { pipe-failed } }
32358 }
32359 \cs_new_protected:Npn __ior_shell_open:nN #1#2
32360 {
32361 \tl_if_in:nnTF {#1} { " }
32362 {
32363 __kernel_msg_error:nnx
32364 { kernel } { quote-in-shell } {#1}
32365 }
32366 { __kernel_ior_open:Nn #2 { |#1 } }
32367 }
32368 __kernel_msg_new:nnnn { kernel } { pipe-failed }
32369 { Cannot-run-piped-system-commands. }
32370 {
32371 LaTeX-tried-to-call-a-system-process-but-this-was-not-possible.\\
32372 Try-the"--shell-escape"-(or"--enable-pipes")-option.
32373 }
```

(End definition for `\ior_shell_open:Nn` and `\__ior_shell_open:nN`. This function is documented on page [267](#).)

## 51.9 Additions to `I3tl`

### 51.9.1 Building a token list

`\__tl_build_begin:N`  $\langle tl\ var \rangle$   
`\__tl_build_end:N`  $\langle tl\ var \rangle$

Between `\__tl_build_begin:N`  $\langle tl\ var \rangle$  and `\__tl_build_end:N`  $\langle tl\ var \rangle$ , the  $\langle tl\ var \rangle$  has the structure

```
\exp_end: ... \exp_end: __tl_build_last:NNn $\langle assignment \rangle$ $\langle next\ tl \rangle$
{ $\langle left \rangle$ } $\langle right \rangle$
```

where  $\langle right \rangle$  is not braced. The “data” it represents is  $\langle left \rangle$  followed by the “data” of  $\langle next\ tl \rangle$  followed by  $\langle right \rangle$ . The  $\langle next\ tl \rangle$  is a token list variable whose name is that of  $\langle tl\ var \rangle$  followed by ‘’. There are between 0 and 4 `\exp_end:` to keep track of when  $\langle left \rangle$  and  $\langle right \rangle$  should be put into the  $\langle next\ tl \rangle$ . The  $\langle assignment \rangle$  is `\cs_set_nopar:Npx` if the variable is local, and `\cs_gset_nopar:Npx` if it is global.

```
\tl_build_begin:N
\tl_build_gbegin:N
__tl_build_begin>NN
__tl_build_begin>NNN
```

First construct the  $\langle next \ tl \rangle$ : using a prime here conflicts with the usual `expl3` convention but we need a name that can be derived from #1 without any external data such as a counter. Empty that  $\langle next \ tl \rangle$  and setup the structure. The local and global versions only differ by a single function `\cs_(g)set_nopar:Npx` used for all assignments: this is important because only that function is stored in the  $\langle tl \ var \rangle$  and  $\langle next \ tl \rangle$  for subsequent assignments. In principle `\__tl_build_begin>NNN` could use `\tl_(g)clear_new:N` to empty #1 and make sure it is defined, but logging the definition does not seem useful so we just do #3 #1 {} to clear it locally or globally as appropriate.

```
32375 \cs_new_protected:Npn \tl_build_begin:N #1
32376 { __tl_build_begin>NN \cs_set_nopar:Npx #1 }
32377 \cs_new_protected:Npn \tl_build_gbegin:N #1
32378 { __tl_build_begin>NN \cs_gset_nopar:Npx #1 }
32379 \cs_new_protected:Npn __tl_build_begin>NN #1#2
32380 { \exp_args:Nc __tl_build_begin>NNN { \cs_to_str:N #2 } #2 #1 }
32381 \cs_new_protected:Npn __tl_build_begin>NNN #1#2#3
32382 {
32383 #3 #1 {}
32384 #3 #2
32385 {
32386 \exp_not:n { \exp_end: \exp_end: \exp_end: \exp_end: }
32387 \exp_not:n { __tl_build_last>NNn #3 #1 {} }
32388 }
32389 }
```

(End definition for `\tl_build_begin:N` and others. These functions are documented on page 272.)

```
\tl_build_clear:N
\tl_build_gclear:N
```

The `begin` and `gbegin` functions already clear enough to make the token list variable effectively empty. Eventually the `begin` and `gbegin` functions should check that #1' is empty or undefined, while the `clear` and `gclear` functions ought to empty #1', #1'' and so on, similar to `\tl_build_end:N`. This only affects memory usage.

```
32390 \cs_new_eq:NN \tl_build_clear:N \tl_build_begin:N
32391 \cs_new_eq:NN \tl_build_gclear:N \tl_build_gbegin:N
```

(End definition for `\tl_build_clear:N` and `\tl_build_gclear:N`. These functions are documented on page 272.)

```
\tl_build_put_right:Nn
\tl_build_put_right:Nx
\tl_build_gput_right:Nn
\tl_build_gput_right:Nx
__tl_build_last>NNn
__tl_build_put:nn
__tl_build_put:nw
```

Similar to `\tl_put_right:Nn`, but apply `\exp:w` to #1. Most of the time this just removes one `\exp_end:`. When there are none left, `\__tl_build_last>NNn` is expanded instead. It resets the definition of the  $\langle tl \ var \rangle$  by ending the `\exp_not:n` and the definition early. Then it makes sure the  $\langle next \ tl \rangle$  (its argument #1) is set-up and starts a new definition. Then `\__tl_build_put:nn` and `\__tl_build_put:nw` place the  $\langle left \rangle$  part of the original  $\langle tl \ var \rangle$  as appropriate for the definition of the  $\langle next \ tl \rangle$  (the  $\langle right \rangle$  part is left in the right place without ever becoming a macro argument). We use `\exp_after:wN` rather than some `\exp_args:No` to avoid reading arguments that are likely very long token lists. We use `\cs_(g)set_nopar:Npx` rather than `\tl_(g)set:Nx` partly for the same reason and partly because the assignments are interrupted by brace tricks, which implies that the assignment does not simply set the token list to an x-expansion of the second argument.

```
32392 \cs_new_protected:Npn \tl_build_put_right:Nn #1#2
32393 {
32394 \cs_set_nopar:Npx #1
32395 { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
32396 }
32397 \cs_new_protected:Npn \tl_build_put_right:Nx #1#2
```

```

32398 {
32399 \cs_set_nopar:Npx #1
32400 { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
32401 }
32402 \cs_new_protected:Npn \tl_build_gput_right:Nn #1#2
32403 {
32404 \cs_gset_nopar:Npx #1
32405 { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
32406 }
32407 \cs_new_protected:Npn \tl_build_gput_right:Nx #1#2
32408 {
32409 \cs_gset_nopar:Npx #1
32410 { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
32411 }
32412 \cs_new_protected:Npn __tl_build_last>NNn #1#2
32413 {
32414 \if_false: { { \fi:
32415 \exp_end: \exp_end: \exp_end: \exp_end: \exp_end:
32416 __tl_build_last>NNn #1 #2 { }
32417 }
32418 }
32419 \if_meaning:w \c_empty_tl #2
32420 __tl_build_begin>NN #1 #2
32421 \fi:
32422 #1 #2
32423 {
32424 \exp_after:wN \exp_not:n \exp_after:wN
32425 {
32426 \exp:w \if_false: } } \fi:
32427 \exp_after:wN __tl_build_put:nn \exp_after:wN {#2}
32428 }
32429 \cs_new_protected:Npn __tl_build_put:nn #1#2 { __tl_build_put:nw {#2} #1 }
32430 \cs_new_protected:Npn __tl_build_put:nw #1#2 __tl_build_last>NNn #3#4#5
32431 { #2 __tl_build_last>NNn #3 #4 { #1 #5 } }

```

(End definition for `\tl_build_put_right:Nn` and others. These functions are documented on page 273.)

See `\tl_build_put_right:Nn` for all the machinery. We could easily provide `\tl_build_put_left:Nnn`, by just add the `\{right\}` material after the `\{(left)\}` in the x-expanding assignment.

```

32432 \cs_new_protected:Npn \tl_build_put_left:Nn #1
32433 { __tl_build_put_left>NNn \cs_set_nopar:Npx #1 }
32434 \cs_generate_variant:Nn \tl_build_put_left:Nn { Nx }
32435 \cs_new_protected:Npn \tl_build_gput_left:Nn #1
32436 { __tl_build_put_left>NNn \cs_gset_nopar:Npx #1 }
32437 \cs_generate_variant:Nn \tl_build_gput_left:Nn { Nx }
32438 \cs_new_protected:Npn __tl_build_put_left>NNn #1#2#3
32439 {
32440 #1 #2
32441 {
32442 \exp_after:wN \exp_not:n \exp_after:wN
32443 {
32444 \exp:w \exp_after:wN __tl_build_put:nn
32445 \exp_after:wN {#2} {#3}

```

```

32446 }
32447 }
32448 }
```

(End definition for `\tl_build_put_left:Nn`, `\tl_build_gput_left:Nn`, and `\_tl_build_put_left:NNn`. These functions are documented on page 273.)

```

\tl_build_get:NN
__tl_build_get:NNN
__tl_build_get:w
__tl_build_get_end:w
```

The idea is to expand the  $\langle tl \ var \rangle$  then the  $\langle next \ tl \rangle$  and so on, all within an x-expanding assignment, and wrap as appropriate in `\exp_not:n`. The various  $\langle left \rangle$  parts are left in the assignment as we go, which enables us to expand the  $\langle next \ tl \rangle$  at the right place. The various  $\langle right \rangle$  parts are eventually picked up in one last `\exp_not:n`, with a brace trick to wrap all the  $\langle right \rangle$  parts together.

```

32449 \cs_new_protected:Npn \tl_build_get:NN
32450 { __tl_build_get:NNN __kernel_tl_set:Nx }
32451 \cs_new_protected:Npn __tl_build_get:NNN #1#2#3
32452 { #1 #3 { \if_false: { \fi: \exp_after:wN __tl_build_get:w #2 } } }
32453 \cs_new:Npn __tl_build_get:w #1 __tl_build_last>NNn #2#3#4
32454 {
32455 \exp_not:n {#4}
32456 \if_meaning:w \c_empty_tl #3
32457 \exp_after:wN __tl_build_get_end:w
32458 \fi:
32459 \exp_after:wN __tl_build_get:w #3
32460 }
32461 \cs_new:Npn __tl_build_get_end:w #1#2#3
32462 { \exp_after:wN \exp_not:n \exp_after:wN { \if_false: } \fi: }
```

(End definition for `\tl_build_get:NN` and others. This function is documented on page 273.)

```

\tl_build_end:N
\tl_build_gend:N
__tl_build_end_loop>NNN
```

Get the data then clear the  $\langle next \ tl \rangle$  recursively until finding an empty one. It is perhaps wasteful to repeatedly use `\cs_to_sr:N`. The local/global scope is checked by `\tl_set:Nx` or `\tl_gset:Nx`.

```

32463 \cs_new_protected:Npn \tl_build_end:N #1
32464 {
32465 __tl_build_get:NNN __kernel_tl_set:Nx #1 #1
32466 \exp_args:Nc __tl_build_end_loop>NNN { \cs_to_str:N #1 ' } \tl_clear:N
32467 }
32468 \cs_new_protected:Npn \tl_build_gend:N #1
32469 {
32470 __tl_build_get:NNN __kernel_tl_gset:Nx #1 #1
32471 \exp_args:Nc __tl_build_end_loop>NNN { \cs_to_str:N #1 ' } \tl_gclear:N
32472 }
32473 \cs_new_protected:Npn __tl_build_end_loop>NNN #1#2
32474 {
32475 \if_meaning:w \c_empty_tl #1
32476 \exp_after:wN \use_none:nnnnnn
32477 \fi:
32478 #2 #1
32479 \exp_args:Nc __tl_build_end_loop>NNN { \cs_to_str:N #1 ' } #2
32480 }
```

(End definition for `\tl_build_end:N`, `\tl_build_gend:N`, and `\__tl_build_end_loop>NNN`. These functions are documented on page 273.)

### 51.9.2 Other additions to **I3tl**

```
\tl_range_braced:Nnn
\tl_range_braced:cnn
\tl_range_braced:nnn
\tl_range_unbraced:Nnn
\tl_range_unbraced:cnn
\tl_range_unbraced:nnn
 _tl_range_braced:w
__tl_range_collect_braced:w
__tl_range_unbraced:w
 _tl_range_collect_unbraced:w
```

For the braced version `\_tl_range_braced:w` sets up `\_tl_range_collect_braced:w` which stores items one by one in an argument after the semicolon. The unbraced version is almost identical. The version preserving braces and spaces starts by deleting spaces before the argument to avoid collecting them, and sets up `\_tl_range_collect:nnn` with a first argument of the form `{ {<collected>} <tokens> }`, whose head is the collected tokens and whose tail is what remains of the original token list. This form makes it easier to move tokens to the `<collected>` tokens.

```
32481 \cs_new:Npn \tl_range_braced:Nnn { \exp_args:No \tl_range_braced:nnn }
32482 \cs_generate_variant:Nn \tl_range_braced:Nnn { c }
32483 \cs_new:Npn \tl_range_braced:nnn { _tl_range:Nnnn _tl_range_braced:w }
32484 \cs_new:Npn \tl_range_unbraced:Nnn
 { \exp_args:No \tl_range_unbraced:nnn }
32485 \cs_generate_variant:Nn \tl_range_unbraced:Nnn { c }
32486 \cs_new:Npn \tl_range_unbraced:nnn
 { _tl_range:Nnnn _tl_range_unbraced:w }
32487 \cs_new:Npn _tl_range_braced:w #1 ; #2
 { _tl_range_collect_braced:w #1 ; { } #2 }
32489 \cs_new:Npn _tl_range_unbraced:w #1 ; #2
 { _tl_range_collect_unbraced:w #1 ; { } #2 }
32490 \cs_new:Npn _tl_range_collect_braced:w #1 ; #2#3
 {
32491 \if_int_compare:w #1 > 1 \exp_stop_f:
 \exp_after:wN _tl_range_collect_braced:w
 \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
32492 \fi:
32493 { #2 {#3} }
32494 }
32495 \cs_new:Npn _tl_range_collect_unbraced:w #1 ; #2#3
 {
32496 \if_int_compare:w #1 > 1 \exp_stop_f:
 \exp_after:wN _tl_range_collect_unbraced:w
 \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
32497 \fi:
32498 { #2 #3 }
32499 }
```

(End definition for `\tl_range_braced:Nnn` and others. These functions are documented on page 272.)

## 51.10 Additions to **I3token**

```
\c_catcode_active_space_tl
```

While `\char_generate:nn` can produce active characters in some engines it cannot in general. It would be possible to simply change the catcode of space but then the code would need to avoid all spaces, making it quite unreadable. Instead we use the primitive `\tex_lowercase:D` trick.

```
32509 \group_begin:
32510 \char_set_catcode_active:N *
32511 \char_set_lccode:nn { '*' } { '\ }
32512 \tex_lowercase:D { \tl_const:Nn \c_catcode_active_space_tl { * } }
32513 \group_end:
```

(End definition for `\c_catcode_active_space_tl`. This variable is documented on page 273.)

```
32514 <@=peek>
```

```
\l_peek_collect_tl
32515 \tl_new:N \l_peek_collect_tl

(End definition for \l_peek_collect_tl.)
```

Most of the work is done by `\__peek_execute_branches_...`, which calls either `\__peek_true:w` or `\__peek_false:w` according to whether the next token `\l_peek_token` matches the search token (stored in `\l_peek_search_token` and `\l_peek_search_tl`). Here, in the `true` case we run `\__peek_collect_true:w`, which generally calls `\__peek_collect:N` to store the peeked token into `\l_peek_collect_tl`, except in special non-N-type cases (begin-group, end-group, or space), where a frozen token is stored. The `true` branch calls `\__peek_execute_branches_...` to fetch more matching tokens. Once there are no more, `\__peek_false_aux:n` closes the safe-align group and runs the user's inline code.

```
32516 \cs_new_protected:Npn \peek_catcode_collect_inline:Nn
32517 { __peek_collect>NNn __peek_execute_branches_catcode: }
32518 \cs_new_protected:Npn \peekCharCode_collect_inline:Nn
32519 { __peek_collect>NNn __peek_execute_branches_charcode: }
32520 \cs_new_protected:Npn \peek_meaning_collect_inline:Nn
32521 { __peek_collect>NNn __peek_execute_branches_meaning: }
32522 \cs_new_protected:Npn __peek_collect:NNn #1#2#3
32523 {
32524 \group_align_safe_begin:
32525 \cs_set_eq:NN \l_peek_search_token #2
32526 \tl_set:Nn \l_peek_search_tl {#2}
32527 \tl_clear:N \l_peek_collect_tl
32528 \cs_set:Npn __peek_false:w
32529 { \exp_args:No __peek_false_aux:n \l_peek_collect_tl }
32530 \cs_set:Npn __peek_false_aux:n ##1
32531 {
32532 \group_align_safe_end:
32533 #3
32534 }
32535 \cs_set_eq:NN __peek_true:w __peek_collect_true:w
32536 \cs_set:Npn __peek_true_aux:w { \peek_after:Nw #1 }
32537 __peek_true_aux:w
32538 }
32539 \cs_new_protected:Npn __peek_collect_true:w
32540 {
32541 \if_case:w
32542 \if_catcode:w \exp_not:N \l_peek_token { 1 \exp_stop_f: \fi:
32543 \if_catcode:w \exp_not:N \l_peek_token } 2 \exp_stop_f: \fi:
32544 \if_meaning:w \l_peek_token \c_space_token 3 \exp_stop_f: \fi:
32545 0 \exp_stop_f:
32546 \exp_after:wN __peek_collect:N
32547 \or: __peek_collect_remove:nw { \c_group_begin_token }
32548 \or: __peek_collect_remove:nw { \c_group_end_token }
32549 \or: __peek_collect_remove:nw { ~ }
32550 \fi:
32551 }
32552 \cs_new_protected:Npn __peek_collect:N #1
32553 {
32554 \tl_put_right:Nn \l_peek_collect_tl {#1}
```

```

32555 __peek_true_aux:w
32556 }
32557 \cs_new_protected:Npn __peek_collect_remove:nw #1
32558 {
32559 \tl_put_right:Nn \l__peek_collect_tl {#1}
32560 \exp_after:wN __peek_true_remove:w
32561 }

```

(End definition for `\peek_catcode_collect_inline:Nn` and others. These functions are documented on page 274.)

```

32562
```

## 52 I3deprecation implementation

```

32563
```

```

32564
```

```

32565
```

### 52.1 Helpers and variables

`\l_deprecation_grace_period_bool`

This is set to `true` when the deprecated command that is being defined is in its grace period, meaning between the time it becomes an error by default and the time 6 months later where even `undo-recent-deprecations` stops restoring it.

```

32566 \bool_new:N \l_deprecation_grace_period_bool

```

(End definition for `\l_deprecation_grace_period_bool`.)

`\s_deprecation_mark`

`\s_deprecation_stop`

```

32567 \scan_new:N \s_deprecation_mark
32568 \scan_new:N \s_deprecation_stop

```

(End definition for `\s_deprecation_mark` and `\s_deprecation_stop`.)

`\_deprecation_date_compare:nNnTF`

`\_deprecation_date_compare_aux:w`

Expects `#1` and `#3` to be dates in the format YYYY-MM-DD (but accepts YYYY or YYYY-MM too, filling in zeros for the missing data). Compares them using `#2` (one of `<`, `=`, `>`).

```

32569 \cs_new:Npn _deprecation_date_compare:nNnTF #1#2#3
32570 { _deprecation_date_compare_aux:w #1 -0-0- \s_deprecation_mark #2 #3 -0-0- \s_deprecation_stop
32571 \cs_new:Npn _deprecation_date_compare_aux:w
32572 #1 - #2 - #3 - #4 \s_deprecation_mark #5 #6 - #7 - #8 - #9 \s_deprecation_stop
32573 {
32574 \int_compare:nNnTF {#1} = {#6}
32575 {
32576 \int_compare:nNnTF {#2} = {#7}
32577 { \int_compare:nNnTF {#3} #5 {#8} }
32578 { \int_compare:nNnTF {#2} #5 {#7} }
32579 }
32580 { \int_compare:nNnTF {#1} #5 {#6} }
32581 }

```

(End definition for `\_deprecation_date_compare:nNnTF` and `\_deprecation_date_compare_aux:w`.)

`\g_kernel_deprecation_undo_recent_bool`

```

32582 \bool_new:N \g_kernel_deprecation_undo_recent_bool

```

(End definition for `\g_kernel_deprecation_undo_recent_bool`.)

`\__deprecation_not_yet_deprecated:nTF`  
`\__deprecation_minus_six_months:w`

Receives a deprecation `<date>` and runs the true (false) branch if the `expl3` date is earlier (later) than `<date>`. If `undo-recent-deprecations` is used we subtract 6 months to the `expl3` date (equivalently add 6 months to the `<date>`). In addition, if the `expl3` date is between `<date>` and `<date>` plus 6 months, `\l_deprecation_grace_period_bool` is set to `true`, otherwise `false`.

```
32583 \cs_new_protected:Npn __deprecation_not_yet_deprecated:nTF #1
32584 {
32585 \bool_set_false:N \l_deprecation_grace_period_bool
32586 \exp_args:No __deprecation_date_compare:nNnTF { \ExplLoaderFileDate } < {#1}
32587 { \use_i:nn }
32588 {
32589 \exp_args:Nf __deprecation_date_compare:nNnTF
32590 {
32591 \exp_after:wN __deprecation_minus_six_months:w
32592 \ExplLoaderFileDate -0-0- \s_deprecation_stop
32593 } < {#1}
32594 {
32595 \bool_set_true:N \l_deprecation_grace_period_bool
32596 \bool_if:NTF \g_kernel_deprecation_undo_recent_bool
32597 }
32598 { \use_ii:nn }
32599 }
32600 }
32601 \cs_new:Npn __deprecation_minus_six_months:w #1 - #2 - #3 - #4 \s_deprecation_stop
32602 {
32603 \int_compare:nNnTF {#2} > 6
32604 { #1 - \int_eval:n { #2 - 6 } - #3 }
32605 { \int_eval:n { #1 - 1 } - \int_eval:n { #2 + 6 } - #3 }
32606 }
```

(End definition for `\__deprecation_not_yet_deprecated:nTF` and `\__deprecation_minus_six_months:w`.)

## 52.2 Patching definitions to deprecate

```
__kernel_patch_deprecation:nnNNpn {\<date>} {\<replacement>} {\<definition>}
{\<function>} {\<parameters>} {\<code>}}
```

defines the `<function>` to produce a warning and run its `<code>`, or to produce an error and not run any `<code>`, depending on the `expl3` date.

- If the `expl3` date is less than the `<date>` (plus 6 months in case `undo-recent-deprecations` is used) then we define the `<function>` to produce a warning and run its code. The warning is actually suppressed in two cases:
  - if neither `undo-recent-deprecations` nor `enable-debug` are in effect we may be in an end-user's document so it is suppressed;
  - if the command is expandable then we cannot produce a warning.
- Otherwise, we define the `<function>` to produce an error.

In both cases we additionally make `\debug_on:n {deprecation}` turn the *(function)* into an `\outer` error, and `\debug_off:n {deprecation}` restore whatever the behaviour was without `\debug_on:n {deprecation}`.

In later sections we use the `\l3doc` key `deprecated` with a date equal to that *(date)* plus 6 months, so that `\l3doc` will complain if we forget to remove the stale *(parameters)* and *{(code)}*.

In the explanations below, *(definition)* *(function)* *(parameters)* *{(code)}* or assignments that only differ in the scope of the *(definition)* will be called “the standard definition”.

```
_kernel_patch_deprecation:nnNpn
 _deprecation_patch_aux:nnNnn
 _deprecation_warn_once:nnNnn
__deprecation_patch_aux:Nn
 _deprecation_just_error:nnNN
```

(The parameter text is grabbed using `#5#`.) The arguments of `\_kernel_deprecation_code:nn` are run upon `\debug_on:n {deprecation}` and `\debug_off:n {deprecation}`, respectively. In both scenarios we the *(function)* may be `\outer` so we undefine it with `\tex_let:D` before redefining it, with `\__kernel_deprecation_error:Nnn` or with some code added shortly.

Then check the date (taking into account `undo-recent-deprecations`) to see if the command should be deprecated right away (`false` branch of `\_deprecation_not_yet_DEPRECATED:nTF`), in which case `\_deprecation_just_error:nnNN` makes *(function)* into an error (not `\outer`), ignoring its *(parameters)* and *(code)* completely.

Otherwise distinguish cases where we should give a warning from those where we shouldn't: warnings can only happen for protected commands, and we only want them if either `undo-recent-deprecations` or `enable-debug` is in force, not for standard users.

```
32607 \cs_new_protected:Npn _kernel_patch_deprecation:nnNpn #1#2#3#4#5#
32608 { _deprecation_patch_aux:nnNnn {#1} {#2} {#3} {#4} {#5} }
32609 \cs_new_protected:Npn _deprecation_patch_aux:nnNnn #1#2#3#4#5#6
32610 {
32611 _kernel_deprecation_code:nn
32612 {
32613 \tex_let:D #4 \scan_stop:
32614 __kernel_deprecation_error:Nnn #4 {#2} {#1}
32615 }
32616 { \tex_let:D #4 \scan_stop: }
32617 _deprecation_not_yet_DEPRECATED:nTF {#1}
32618 {
32619 \bool_if:nTF
32620 {
32621 \cs_if_eq_p:NN #3 \cs_gset_protected:Npn &&
32622 _kernel_if_debug:TF
32623 { \c_true_bool } { \g__kernel_deprecation_undo_recent_bool }
32624 }
32625 { __deprecation_warn_once:nnNnn {#1} {#2} {#4} {#5} {#6} }
32626 { __deprecation_patch_aux:Nn #3 { #4 #5 {#6} } }
32627 }
32628 { __deprecation_just_error:nnNN {#1} {#2} {#3} {#4} }
32629 }
```

In case we want a warning, the *(function)* is defined to produce such a warning without grabbing any argument, then redefine itself to the standard definition that the *(function)* should have, with arguments, and call that definition. The x-type expansion and `\exp-not:n` avoid needing to double the `#`, which we could not do anyways. We then deal with the code for `\debug_off:n {deprecation}`: presumably someone doing that does not need the warning so we simply do the standard definition.

```

32630 \cs_new_protected:Npn __deprecation_warn_once:nnNnn #1#2#3#4#5
32631 {
32632 \cs_gset_protected:Npx #3
32633 {
32634 __kernel_if_debug:TF
32635 {
32636 \exp_not:N __kernel_msg_warning:nxxxx
32637 { kernel } { deprecated-command }
32638 {#1}
32639 { \token_to_str:N #3 }
32640 { \tl_to_str:n {#2} }
32641 }
32642 {
32643 \exp_not:n { \cs_gset_protected:Npn #3 #4 {#5} }
32644 \exp_not:N #3
32645 }
32646 __kernel_deprecation_code:nn { }
32647 { \cs_set_protected:Npn #3 #4 {#5} }
32648 }

```

In case we want neither warning nor error, the *<function>* is given its standard definition. Here #1 is `\cs_new:Npn` or `\cs_new_protected:Npn` and #2 is *<function> <parameters> {<code>}*, so #1#2 performs the assignment. For `\debug_off:n {deprecation}` we want to use the same assignment but with a different scope, hence the `\cs_if_eq:NNTF` test.

```

32649 \cs_new_protected:Npn __deprecation_patch_aux:Nn #1#2
32650 {
32651 #1 #2
32652 \cs_if_eq:NNTF #1 \cs_gset_protected:Npn
32653 { __kernel_deprecation_code:nn { } { \cs_set_protected:Npn #2 } }
32654 { __kernel_deprecation_code:nn { } { \cs_set:Npn #2 } }
32655 }

```

Finally, if we want an error we reuse the same `\__deprecation_patch_aux:Nn` as the previous case. Indeed, we want `\debug_off:n {deprecation}` to make the *<function>* into an error, just like it is by default. The error is expandable or not, and the last argument of the error message is empty or is `grace` to denote the case where we are in the 6 month grace period, in which case the error message is more detailed.

```

32656 \cs_new_protected:Npn __deprecation_just_error:nnNN #1#2#3#4
32657 {
32658 \exp_args:NNx __deprecation_patch_aux:Nn #3
32659 {
32660 \exp_not:N #4
32661 {
32662 \cs_if_eq:NNTF #3 \cs_gset_protected:Npn
32663 { \exp_not:N __kernel_msg_error:nnnnnn }
32664 { \exp_not:N __kernel_msg_expandable_error:nnnnnn }
32665 { kernel } { deprecated-command }
32666 {#1}
32667 { \token_to_str:N #4 }
32668 { \tl_to_str:n {#2} }
32669 { \bool_if:NT \l_deprecation_grace_period_bool { grace } }
32670 }
32671 }
32672 }

```

(End definition for `\_kernel_patch_deprecation:nnNNpn` and others.)

`\_kernel_deprecation_error:Nnn`

The `\outer` definition here ensures the command cannot appear in an argument. Use this auxiliary on all commands that have been removed since 2015.

```
32673 \cs_new_protected:Npn _kernel_deprecation_error:Nnn #1#2#3
32674 {
32675 \tex_protected:D \tex_outer:D \tex_edef:D #1
32676 {
32677 \exp_not:N _kernel_msg_expandable_error:nnnnn
32678 { kernel } { deprecated-command }
32679 { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
32680 \exp_not:N _kernel_msg_error:nnxxx
32681 { kernel } { deprecated-command }
32682 { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
32683 }
32684 }
```

(End definition for `\_kernel_deprecation_error:Nnn`.)

```
32685 _kernel_msg_new:nnn { kernel } { deprecated-command }
32686 {
32687 \tl_if_blank:nF {#3} { Use~\tl_trim_spaces:n {#3} ~not~ }
32688 #2-deprecated-on~#1.
32689 \str_if_eq:nnT {#4} { grace }
32690 {
32691 \c_space_tl
32692 For~6~months~after~that~date~one~can~restore~a~deprecated~
32693 command~by~loading~the~expl3~package~with~the~option~
32694 'undo-recent-deprecations'.
32695 }
32696 }
```

### 52.3 Removed functions

`\_deprecation_old_protected:Nnn`  
`\_deprecation_old:Nnn`

Short-hands for old commands whose definition does not matter anymore, i.e., commands past the grace period.

```
32697 \cs_new_protected:Npn _deprecation_old_protected:Nnn #1#2#3
32698 {
32699 _kernel_patch_deprecation:nnNNpn {#3} {#2}
32700 \cs_gset_protected:Npn #1 { }
32701 }
32702 \cs_new_protected:Npn _deprecation_old:Nnn #1#2#3
32703 {
32704 _kernel_patch_deprecation:nnNNpn {#3} {#2}
32705 \cs_gset:Npn #1 { }
32706 }
32707 _deprecation_old:Nnn \box_resize:Nnn
32708 { \box_resize_to_wd_and_ht_plus_dp:Nnn } { 2019-01-01 }
32709 _deprecation_old:Nnn \box_use_clear:N
32710 { \box_use_drop:N } { 2019-01-01 }
32711 _deprecation_old:Nnn \c_job_name_tl
32712 { \c_sys_jobname_str } { 2017-01-01 }
32713 _deprecation_old:Nnn \c_minus_one
32714 { -1 } { 2019-01-01 }
```

```

32715 __deprecation_old:Nnn \c_zero
32716 { 0 } { 2020-01-01 }
32717 __deprecation_old:Nnn \c_one
32718 { 1 } { 2020-01-01 }
32719 __deprecation_old:Nnn \c_two
32720 { 2 } { 2020-01-01 }
32721 __deprecation_old:Nnn \c_three
32722 { 3 } { 2020-01-01 }
32723 __deprecation_old:Nnn \c_four
32724 { 4 } { 2020-01-01 }
32725 __deprecation_old:Nnn \c_five
32726 { 5 } { 2020-01-01 }
32727 __deprecation_old:Nnn \c_six
32728 { 6 } { 2020-01-01 }
32729 __deprecation_old:Nnn \c_seven
32730 { 7 } { 2020-01-01 }
32731 __deprecation_old:Nnn \c_eight
32732 { 8 } { 2020-01-01 }
32733 __deprecation_old:Nnn \c_nine
32734 { 9 } { 2020-01-01 }
32735 __deprecation_old:Nnn \c_ten
32736 { 10 } { 2020-01-01 }
32737 __deprecation_old:Nnn \c_eleven
32738 { 11 } { 2020-01-01 }
32739 __deprecation_old:Nnn \c_twelve
32740 { 12 } { 2020-01-01 }
32741 __deprecation_old:Nnn \c_thirteen
32742 { 13 } { 2020-01-01 }
32743 __deprecation_old:Nnn \c_fourteen
32744 { 14 } { 2020-01-01 }
32745 __deprecation_old:Nnn \c_fifteen
32746 { 15 } { 2020-01-01 }
32747 __deprecation_old:Nnn \c_sixteen
32748 { 16 } { 2020-01-01 }
32749 __deprecation_old:Nnn \c_thirty_two
32750 { 32 } { 2020-01-01 }
32751 __deprecation_old:Nnn \c_one_hundred
32752 { 100 } { 2020-01-01 }
32753 __deprecation_old:Nnn \c_two_hundred_fifty_five
32754 { 255 } { 2020-01-01 }
32755 __deprecation_old:Nnn \c_two_hundred_fifty_six
32756 { 256 } { 2020-01-01 }
32757 __deprecation_old:Nnn \c_one_thousand
32758 { 1000 } { 2020-01-01 }
32759 __deprecation_old:Nnn \c_ten_thousand
32760 { 10000 } { 2020-01-01 }
32761 __deprecation_old:Nnn \dim_case:nnn
32762 { \dim_case:nnF } { 2015-07-14 }
32763 __deprecation_old:Nnn \file_add_path:nN
32764 { \file_get_full_name:nN } { 2019-01-01 }
32765 __deprecation_old_protected:Nnn \file_if_exist_input:nT
32766 { \file_if_exist:nT and~ \file_input:n } { 2018-03-05 }
32767 __deprecation_old_protected:Nnn \file_if_exist_input:nTF
32768 { \file_if_exist:nT and~ \file_input:n } { 2018-03-05 }

```

```

32769 __deprecation_old:Nnn \file_list:
32770 { \file_log_list: } { 2019-01-01 }
32771 __deprecation_old:Nnn \file_path_include:n
32772 { \seq_put_right:Nn \l_file_search_path_seq } { 2019-01-01 }
32773 __deprecation_old:Nnn \file_path_remove:n
32774 { \seq_remove_all:Nn \l_file_search_path_seq } { 2019-01-01 }
32775 __deprecation_old:Nnn \g_file_current_name_tl
32776 { \g_file_curr_name_str } { 2019-01-01 }
32777 __deprecation_old:Nnn \int_case:nnn
32778 { \int_case:nnF } { 2015-07-14 }
32779 __deprecation_old:Nnn \int_from_binary:n
32780 { \int_from_bin:n } { 2016-01-05 }
32781 __deprecation_old:Nnn \int_from_hexadecimal:n
32782 { \int_from_hex:n } { 2016-01-05 }
32783 __deprecation_old:Nnn \int_from_octal:n
32784 { \int_from_oct:n } { 2016-01-05 }
32785 __deprecation_old:Nnn \int_to_binary:n
32786 { \int_to_bin:n } { 2016-01-05 }
32787 __deprecation_old:Nnn \int_to_hexadecimal:n
32788 { \int_to_hex:n } { 2016-01-05 }
32789 __deprecation_old:Nnn \int_to_octal:n
32790 { \int_to_oct:n } { 2016-01-05 }
32791 __deprecation_old_protected:Nnn \ior_get_str:NN
32792 { \ior_str_get:NN } { 2018-03-05 }
32793 __deprecation_old:Nnn \ior_list_streams:
32794 { \ior_show_list: } { 2019-01-01 }
32795 __deprecation_old:Nnn \ior_log_streams:
32796 { \ior_log_list: } { 2019-01-01 }
32797 __deprecation_old:Nnn \iow_list_streams:
32798 { \iow_show_list: } { 2019-01-01 }
32799 __deprecation_old:Nnn \iow_log_streams:
32800 { \iow_log_list: } { 2019-01-01 }
32801 __deprecation_old:Nnn \lua_escape_x:n
32802 { \lua_escape:e } { 2020-01-01 }
32803 __deprecation_old:Nnn \lua_now_x:n
32804 { \lua_now:e } { 2020-01-01 }
32805 __deprecation_old_protected:Nnn \lua_shipout_x:n
32806 { \lua_shipout_e:n } { 2020-01-01 }
32807 __deprecation_old:Nnn \luatex_if_engine_p:
32808 { \sys_if_engine_luatex_p: } { 2017-01-01 }
32809 __deprecation_old:Nnn \luatex_if_engine:F
32810 { \sys_if_engine_luatex:F } { 2017-01-01 }
32811 __deprecation_old:Nnn \luatex_if_engine:T
32812 { \sys_if_engine_luatex:T } { 2017-01-01 }
32813 __deprecation_old:Nnn \luatex_if_engine:TF
32814 { \sys_if_engine_luatex:TF } { 2017-01-01 }
32815 __deprecation_old_protected:Nnn \msg_interrupt:nnn
32816 { [Defined~error~message] } { 2020-01-01 }
32817 __deprecation_old_protected:Nnn \msg_log:n
32818 { \iow_log:n } { 2020-01-01 }
32819 __deprecation_old_protected:Nnn \msg_term:n
32820 { \iow_term:n } { 2020-01-01 }
32821 __deprecation_old:Nnn \pdftex_if_engine_p:
32822 { \sys_if_engine_pdftex_p: } { 2017-01-01 }

```

```

32823 __deprecation_old:Nnn \pdftex_if_engine:F
32824 { \sys_if_engine_pdftex:F } { 2017-01-01 }
32825 __deprecation_old:Nnn \pdftex_if_engine:T
32826 { \sys_if_engine_pdftex:T } { 2017-01-01 }
32827 __deprecation_old:Nnn \pdftex_if_engine:TF
32828 { \sys_if_engine_pdftex:TF } { 2017-01-01 }
32829 __deprecation_old:Nnn \prop_get:cn
32830 { \prop_item:cn } { 2016-01-05 }
32831 __deprecation_old:Nnn \prop_get:Nn
32832 { \prop_item:Nn } { 2016-01-05 }
32833 __deprecation_old:Nnn \quark_if_recursion_tail_break:N
32834 { } { 2015-07-14 }
32835 __deprecation_old:Nnn \quark_if_recursion_tail_break:n
32836 { } { 2015-07-14 }
32837 __deprecation_old:Nnn \scan_align_safe_stop:
32838 { \protected~commands } { 2017-01-01 }
32839 __deprecation_old:Nnn \sort_ordered:
32840 { \sort_return_same: } { 2019-01-01 }
32841 __deprecation_old:Nnn \sort_reversed:
32842 { \sort_return_swapped: } { 2019-01-01 }
32843 __deprecation_old:Nnn \str_case:nnn
32844 { \str_case:nnF } { 2015-07-14 }
32845 __deprecation_old:Nnn \str_case:onn
32846 { \str_case:onF } { 2015-07-14 }
32847 __deprecation_old:Nnn \str_case_x:nn
32848 { \str_case_e:nn } { 2020-01-01 }
32849 __deprecation_old:Nnn \str_case_x:nnn
32850 { \str_case_e:nnF } { 2015-07-14 }
32851 __deprecation_old:Nnn \str_case_x:nnT
32852 { \str_case_e:nnT } { 2020-01-01 }
32853 __deprecation_old:Nnn \str_case_x:nnTF
32854 { \str_case_e:nnTF } { 2020-01-01 }
32855 __deprecation_old:Nnn \str_case_x:nnF
32856 { \str_case_e:nnF } { 2020-01-01 }
32857 __deprecation_old:Nnn \str_if_eq_x:p:nn
32858 { \str_if_eq_p:ee } { 2020-01-01 }
32859 __deprecation_old:Nnn \str_if_eq_x:nnT
32860 { \str_if_eq:eeT } { 2020-01-01 }
32861 __deprecation_old:Nnn \str_if_eq_x:nnF
32862 { \str_if_eq:eeF } { 2020-01-01 }
32863 __deprecation_old:Nnn \str_if_eq_x:nnTF
32864 { \str_if_eq:eeTF } { 2020-01-01 }
32865 __deprecation_old_protected:Nnn \tl_show_analysis:N
32866 { \tl_analysis_show:N } { 2020-01-01 }
32867 __deprecation_old_protected:Nnn \tl_show_analysis:n
32868 { \tl_analysis_show:n } { 2020-01-01 }
32869 __deprecation_old:Nnn \tl_case:cnn
32870 { \tl_case:cnf } { 2015-07-14 }
32871 __deprecation_old:Nnn \tl_case:Nnn
32872 { \tl_case:Nnf } { 2015-07-14 }
32873 __deprecation_old_protected:Nnn \tl_to_lowercase:n
32874 { \tex_lowercase:D } { 2018-03-05 }
32875 __deprecation_old_protected:Nnn \tl_to_uppercase:n
32876 { \tex_uppercase:D } { 2018-03-05 }

```

```

32877 __deprecation_old:Nnn \token_new:Nn
32878 { \cs_new_eq:NN } { 2019-01-01 }
32879 __deprecation_old:Nnn \xetex_if_engine_p:
32880 { \sys_if_engine_xetex_p: } { 2017-01-01 }
32881 __deprecation_old:Nnn \xetex_if_engine:F
32882 { \sys_if_engine_xetex:F } { 2017-01-01 }
32883 __deprecation_old:Nnn \xetex_if_engine:T
32884 { \sys_if_engine_xetex:T } { 2017-01-01 }
32885 __deprecation_old:Nnn \xetex_if_engine:TF
32886 { \sys_if_engine_xetex:TF } { 2017-01-01 }

(End definition for __deprecation_old_protected:Nnn and __deprecation_old:Nnn.)

```

## 52.4 Loading the patches

When loaded first, the patches are simply read here. Here the deprecation code is loaded with the lower-level \\_\_kernel\_... macro because we don't want it to flip the \g\_\_-sys\_deprecation\_bool boolean, so that the deprecation code can be re-loaded later (when using `undo-recent-deprecations`).

```

32887 \group_begin:
32888 \cs_set_protected:Npn \ProvidesExplFile
32889 {
32890 \char_set_catcode_space:n { '\ }
32891 \ProvidesExplFileAux
32892 }
32893 \cs_set_protected:Npx \ProvidesExplFileAux #1#2#3#4
32894 {
32895 \group_end:
32896 \cs_if_exist:NTF \ProvidesFile
32897 { \exp_not:N \ProvidesFile {#1} [#2~v#3~#4] }
32898 { \iow_log:x { File:-#1~#2~v#3~#4 } }
32899 }
32900 \cs_gset_protected:Npn __kernel_sys_configuration_load:n #1
32901 { \file_input:n { #1 .def } }
32902 __kernel_sys_configuration_load:n { 13deprecation }
32903 </kernel>
32904 (*patches)

Standard file identification.

32905 \ProvidesExplFile{13deprecation.def}{2019-04-06}{}{L3 Deprecated functions}

```

## 52.5 Deprecated l3box functions

```

\box_set_eq_clear:NN
\box_set_eq_clear:cN
\box_set_eq_clear:Nc
\box_set_eq_clear:cc
\box_gset_eq_clear:NN
\box_gset_eq_clear:cN
\box_gset_eq_clear:Nc
\box_gset_eq_clear:cc

32906 __kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \box_set_eq_drop:N }
32907 \cs_gset_protected:Npn \box_set_eq_clear:NN #1#2
32908 { \tex_setbox:D #1 \tex_box:D #2 }
32909 __kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \box_gset_eq_drop:N }
32910 \cs_gset_protected:Npn \box_gset_eq_clear:NN #1#2
32911 { \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
32912 \cs_generate_variant:Nn \box_set_eq_clear:NN { c , Nc , cc }
32913 \cs_generate_variant:Nn \box_gset_eq_clear:NN { c , Nc , cc }

```

(End definition for `\box_set_eq_clear:NN` and `\box_gset_eq_clear:NN`.)

```
\hbox_unpack_clear:N
\hbox_unpack_clear:c
32914 __kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \hbox_unpack_drop:N }
32915 \cs_gset_protected:Npn \hbox_unpack_clear:N
32916 { \hbox_unpack_drop:N }
32917 \cs_generate_variant:Nn \hbox_unpack_clear:N { c }

(End definition for \hbox_unpack_clear:N.)

\vbox_unpack_clear:N
\vbox_unpack_clear:c
32918 __kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \vbox_unpack_drop:N }
32919 \cs_gset_protected:Npn \vbox_unpack_clear:N
32920 { \vbox_unpack_drop:N }
32921 \cs_generate_variant:Nn \vbox_unpack_clear:N { c }

(End definition for \vbox_unpack_clear:N.)
```

## 52.6 Deprecated `I3str` functions

32922 `<@=str>`

```
\str_lower_case:n
\str_lower_case:f
\str_upper_case:n
\str_upper_case:f
\str_fold_case:n
\str_fold_case:V
32923 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_lowercase:n }
32924 \cs_gset:Npn \str_lower_case:n { \str_lowercase:n }
32925 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_lowercase:f }
32926 \cs_gset:Npn \str_lower_case:f { \str_lowercase:f }
32927 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_uppercase:n }
32928 \cs_gset:Npn \str_uppercase:n { \str_uppercase:n }
32929 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_uppercase:f }
32930 \cs_gset:Npn \str_uppercase:f { \str_uppercase:f }
32931 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_foldcase:n }
32932 \cs_gset:Npn \str_fold_case:n { \str_foldcase:n }
32933 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \str_foldcase:V }
32934 \cs_gset:Npn \str_fold_case:V { \str_foldcase:V }

(End definition for \str_lower_case:n, \str_upper_case:n, and \str_fold_case:n.)
```

`\str_declare_eight_bit_encoding:nnn`

This command was made internal, with one more argument. There is no easy way to compute a reasonable value for that extra argument so we take a value that is big enough to accomodate all of Unicode.

```
32935 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { }
32936 \cs_gset_protected:Npn \str_declare_eight_bit_encoding:nnn #1
32937 { __str_declare_eight_bit_encoding:nnnn {#1} { 1114112 } }

(End definition for \str_declare_eight_bit_encoding:nnn.)
```

## 52.7 Deprecated `I3seq` functions

`\seq_indexed_map_inline:Nn`

`\seq_indexed_map_function:NN`

```
32938 __kernel_patch_deprecation:nnNNpn { 2022-07-01 } { \seq_map_indexed_inline:Nn }
32939 \cs_gset:Npn \seq_indexed_map_inline:Nn { \seq_map_indexed_inline:Nn }
32940 __kernel_patch_deprecation:nnNNpn { 2022-07-01 } { \seq_map_indexed_function:NN }
32941 \cs_gset:Npn \seq_indexed_map_function:NN { \seq_map_indexed_function:NN }

(End definition for \seq_indexed_map_inline:Nn and \seq_indexed_map_function:NN.)
```

### 52.7.1 Deprecated **I3tl** functions

```

32942 <@@=tl>

\tl_set_from_file:Nnn
\tl_set_from_file:cnn
\tl_gset_from_file:Nnn
\tl_gset_from_file:cnn
\tl_set_from_file_x:Nnn
\tl_set_from_file_x:cnn
\tl_gset_from_file_x:Nnn
\tl_gset_from_file_x:cnn

32943 __kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \file_get:nnN }
32944 \cs_gset_protected:Npn \tl_set_from_file:Nnn #1#2#3
32945 { \file_get:nnN {#3} {#2} #1 }
32946 \cs_generate_variant:Nn \tl_set_from_file:Nnn { c }
32947 __kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \file_get:nnN }
32948 \cs_gset_protected:Npn \tl_gset_from_file:Nnn #1#2#3
32949 {
32950 \group_begin:
32951 \file_get:nnN {#3} {#2} \l__tl_internal_a_tl
32952 \tl_gset_eq:NN #1 \l__tl_internal_a_tl
32953 \group_end:
32954 }
32955 \cs_generate_variant:Nn \tl_gset_from_file:Nnn { c }
32956 __kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \file_get:nnN }
32957 \cs_gset_protected:Npn \tl_set_from_file_x:Nnn #1#2#3
32958 {
32959 \group_begin:
32960 \file_get:nnN {#3} {#2} \l__tl_internal_a_tl
32961 #2 \scan_stop:
32962 __kernel_tl_set:Nx \l__tl_internal_a_tl { \l__tl_internal_a_tl }
32963 \exp_args:NNNo \group_end:
32964 \tl_set:Nn #1 \l__tl_internal_a_tl
32965 }
32966 \cs_generate_variant:Nn \tl_set_from_file_x:Nnn { c }
32967 __kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \file_get:nnN }
32968 \cs_gset_protected:Npn \tl_gset_from_file_x:Nnn #1#2#3
32969 {
32970 \group_begin:
32971 \file_get:nnN {#3} {#2} \l__tl_internal_a_tl
32972 #2 \scan_stop:
32973 __kernel_tl_gset:Nx #1 { \l__tl_internal_a_tl }
32974 \group_end:
32975 }
32976 \cs_generate_variant:Nn \tl_gset_from_file_x:Nnn { c }

(End definition for \tl_set_from_file:Nnn and others.)

```

```

\tl_lower_case:n
\tl_lower_case:nn
\tl_upper_case:n
\tl_upper_case:nn
\tl_mixed_case:n
\tl_mixed_case:nn

32977 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_lowercase:n }
32978 \cs_gset:Npn \tl_lower_case:n #1
32979 { \text_lowercase:n {#1} }
32980 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_lowercase:nn }
32981 \cs_gset:Npn \tl_lower_case:nn #1#2
32982 { \text_lowercase:nn {#1} {#2} }
32983 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_uppercase:n }
32984 \cs_gset:Npn \tl_upper_case:n #1
32985 { \text_uppercase:n {#1} }
32986 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_uppercase:nn }
32987 \cs_gset:Npn \tl_upper_case:nn #1#2
32988 { \text_uppercase:nn {#1} {#2} }

```

```

32989 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_titlecase:n }
32990 \cs_gset:Npn \tl_mixed_case:n #1
32991 { \text_titlecase:n {#1} }
32992 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \text_titlecase:nn }
32993 \cs_gset:Npn \tl_mixed_case:nn #1#2
32994 { \text_titlecase:nn {#1} {#2} }

```

(End definition for `\tl_lower_case:n` and others.)

## 52.8 Deprecated **I3token** functions

```

\token_get_prefix_spec:N
\token_get_arg_spec:N
\token_get_replacement_spec:N
32995 __kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \cs_prefix_spec:N }
32996 \cs_gset:Npn \token_get_prefix_spec:N { \cs_prefix_spec:N }
32997 __kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \cs_argument_spec:N }
32998 \cs_gset:Npn \token_get_arg_spec:N { \cs_argument_spec:N }
32999 __kernel_patch_deprecation:nnNNpn { 2021-01-01 } { \cs_replacement_spec:N }
33000 \cs_gset:Npn \token_get_replacement_spec:N { \cs_replacement_spec:N }

```

(End definition for `\token_get_prefix_spec:N`, `\token_get_arg_spec:N`, and `\token_get_replacement_spec:N`.)

```

\char_lower_case:N
\char_upper_case:N
\char_mixed_case:Nn
\char_fold_case:N
\char_str_lower_case:N
\char_str_upper_case:N
\char_str_mixed_case:Nn
\char_str_fold_case:N
33001 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_lowercase:N }
33002 \cs_gset:Npn \char_lower_case:N { \char_lowercase:N }
33003 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_uppercase:N }
33004 \cs_gset:Npn \char_upper_case:N { \char_uppercase:N }
33005 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_titlecase:N }
33006 \cs_gset:Npn \char_mixed_case:N { \char_titlecase:N }
33007 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_foldcase:N }
33008 \cs_gset:Npn \char_fold_case:N { \char_foldcase:N }
33009 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_str_lowercase:N }
33010 \cs_gset:Npn \char_str_lower_case:N { \char_str_lowercase:N }
33011 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_str_uppercase:N }
33012 \cs_gset:Npn \char_str_upper_case:N { \char_str_uppercase:N }
33013 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_str_titlecase:N }
33014 \cs_gset:Npn \char_str_mixed_case:N { \char_str_titlecase:N }
33015 __kernel_patch_deprecation:nnNNpn { 2022-01-01 } { \char_str_foldcase:N }
33016 \cs_gset:Npn \char_str_fold_case:N { \char_str_foldcase:N }

```

(End definition for `\char_lower_case:N` and others.)

## 52.9 Deprecated **I3file** functions

```

\c_term_ior
33017 __kernel_patch_deprecation:nnNNpn { 2021-01-01 } { -1 }
33018 \cs_gset_protected:Npn \c_term_ior { -1 \scan_stop: }

```

(End definition for `\c_term_ior`.)

```

33019 </patches>
33020 </package>

```

# Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

| Symbols          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| !                | 213                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| \"               | 10509, 10512,<br>29543, 31803, 31827, 31833, 31837,<br>31843, 31847, 31853, 31859, 31866,<br>31867, 31873, 31877, 31879, 31988                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \# .             | 5324, 5632, 10509, 13109, 31672                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| \\$ .            | 5323, 5630, 10509, 10512, 24725, 31672                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| \% .             | 5325, 5640, 10509, 13111, 31672                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| \& .             | 5316, 5631, 10509, 10512, 11898                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| \&& .            | 212                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| \' .             | 29543, 31797, 31824,<br>31831, 31835, 31840, 31845, 31848,<br>31850, 31857, 31862, 31863, 31870,<br>31875, 31878, 31886, 31887, 31934,<br>31935, 31942, 31943, 31954, 31955,<br>31960, 31961, 31989, 31990, 32012,<br>32013, 32016, 32017, 32018, 32019                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| \( .             | 13622, 13808, 13883, 29564                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| \) .             | 29564                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| *                | 213                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| \* .             | 5009, 5032, 10885, 10887, 10891, 10899                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| \** .            | 213                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| +. .             | 212, 213                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \,. .            | 14839, 31684                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| \-. .            | 212, 213                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \-. .            | 193                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| \. .             | 29543, 31802,<br>31890, 31891, 31900, 31901, 31910,<br>31911, 31928, 31940, 31941, 31991,<br>31992, 32022, 32023, 32026, 32027                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| / .              | 213                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| \/. .            | 192                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| \: .             | 5322                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| \::: .           | 37, 340, 366,<br>2243, 2244, 2245, 2246, 2247, 2248,<br>2249, 2251, 2255, 2256, 2259, 2262,<br>2269, 2272, 2275, 2281, 2368, 2444,<br>2446, 2451, 2458, 2462, 2465, 2470,<br>2485, 2526, 2527, 2528, 2529, 2540                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| \:::N .          | 37, 2247, 2368, 2529                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| \:::V .          | 37, 2275                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \:::V_unbraced . | 37, 2443                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \:::c .          | 37, 2249                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \:::e .          | 37, 2253, 2368                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \:::e_unbraced . | 37, 2443, 2485                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \:::error .      | 156, 12712                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| \:::f .          | 37, 2262, 2528                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \:::f_unbraced . | 37, 2443                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \:::n .          | 37, 377, 2246, 2526, 2529                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| \:::o .          | 37, 2251, 2527                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \:::o_unbraced . | 37, 2443, 2526, 2527, 2528, 2529                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| \:::p .          | 37, 340, 2248                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| \:::v .          | 37, 2275                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \:::v_unbraced . | 37, 2443                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \:::x .          | 37, 2269                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \:::x_unbraced . | 37, 2443, 2540                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| < .              | 212                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| = .              | 212                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| \= .             | 14840,<br>29543, 31800, 31880, 31881, 31896,<br>31897, 31919, 31920, 31921, 31948,<br>31949, 31974, 31975, 32028, 32029                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| > .              | 212                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| ? .              | 212                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| ? commands:      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| ?: .             | 212                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| \\" .            | 2182, 5318,<br>5627, 5870, 5871, 5874, 6196, 6199,<br>6200, 6201, 6202, 6207, 6213, 6218,<br>6225, 6382, 6385, 6386, 6387, 6389,<br>6395, 6400, 6405, 6556, 6563, 10509,<br>11801, 11819, 11821, 11826, 11827,<br>11851, 11861, 11868, 11883, 12327,<br>12335, 12342, 12354, 12355, 12380,<br>12381, 12388, 12409, 12411, 12412,<br>12444, 12457, 12458, 12471, 12538,<br>12571, 12587, 12591, 12596, 12603,<br>13114, 14124, 14128, 14129, 14131,<br>14137, 14139, 14144, 14145, 14147,<br>14148, 14150, 14152, 14164, 15926,<br>15938, 15944, 16726, 16729, 16730,<br>16731, 16738, 16741, 16742, 22905,<br>22907, 22908, 22911, 22913, 22914,<br>22917, 22919, 22920, 22921, 22925,<br>22932, 23319, 23322, 23323, 23347,<br>23348, 23355, 23356, 23797, 25266,<br>25379, 25380, 25381, 25402, 26701,<br>26705, 26710, 26744, 26753, 26757,<br>26762, 26782, 26784, 26785, 26787,<br>26790, 26792, 26799, 26803, 26806,<br>26810, 26812, 26816, 26818, 26824,<br>26826, 26830, 26832, 26836, 26841,<br>26843, 26885, 26887, 26892, 26894,<br>26900, 26905, 26906, 26910, 26914, |

|                                                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
|-----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------|----------------------------|
| \` . . . . .                                              | 26924, 26927, 26931, 26932, 26936, 26944, 27001, 28937, 31670, 32371                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | acos . . . . .             | 215                        |
| \{ . . . . .                                              | 5319, 5628, 5875, 10509, 13108, 24051, 26705, 26710, 26757, 26806, 26843, 26932, 26936, 29571, 29572, 29573, 31672                                                                                                                                                                                                                                                                                                                                                                                                                          | acosd . . . . .            | 215                        |
| \} . . . . .                                              | 5320, 5629, 5875, 10509, 13110, 26704, 26710, 26807, 26843, 26932, 26936, 29571, 29572, 29573, 29574, 31672                                                                                                                                                                                                                                                                                                                                                                                                                                 | acot . . . . .             | 216                        |
| \^ . . . . .                                              | 88, 1870, 2570, 5321, 5633, 5989, 5990, 6323, 6324, 6505, 6506, 6507, 10509, 10512, 10514, 10520, 10571, 11905, 13026, 13062, 20606, 23402, 23475, 23478, 23997, 24002, 24003, 24004, 24005, 24008, 24019, 24056, 24110, 24112, 24114, 24116, 24118, 24120, 24724, 26306, 26309, 26323, 26326, 26335, 26338, 26341, 26344, 26358, 26361, 29543, 31798, 31825, 31832, 31836, 31841, 31846, 31851, 31858, 31864, 31865, 31871, 31876, 31888, 31889, 31906, 31907, 31914, 31915, 31929, 31930, 31931, 31962, 31963, 31984, 31985, 31986, 31987 | acotd . . . . .            | 216                        |
| \~ . . . . .                                              | 213                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | acsc . . . . .             | 215                        |
| \_ . . . . .                                              | 5327, 10509, 10512, 31672                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | acscd . . . . .            | 215                        |
| \` . . . . .                                              | 29543, 31796, 31823, 31830, 31834, 31839, 31844, 31849, 31856, 31860, 31861, 31869, 31874, 32014, 32015                                                                                                                                                                                                                                                                                                                                                                                                                                     | \adjdemerits . . . . .     | 199                        |
| \  . . . . .                                              | 212                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | \adjustspacing . . . . .   | 912                        |
| \~ . . . . .                                              | 3835, 5326, 5635, 10509, 10512, 13112, 24041, 24045, 24051, 29543, 31676, 31799, 31826, 31838, 31842, 31852, 31868, 31872, 31916, 31917, 31918, 31972, 31973                                                                                                                                                                                                                                                                                                                                                                                | \advance . . . . .         | 200                        |
| \u . . . . .                                              | 191, 1774, 5009, 5032, 5634, 5875, 6199, 6200, 6201, 10509, 10877, 12382, 12401, 12508, 12644, 13115, 14148, 23519, 23996, 24001, 24045, 24055, 24230, 26355, 29299, 29569, 29570, 31683, 32511, 32890                                                                                                                                                                                                                                                                                                                                      | \AE . . . . .              | 29548, 31380, 31722, 32016 |
| \a . . . . .                                              | 5010, 5033                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | \ae . . . . .              | 29548, 31380, 31732, 32017 |
| \AA . . . . .                                             | 29547, 31379, 31721                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | \afterassignment . . . . . | 201                        |
| \aa . . . . .                                             | 29547, 31379, 31731                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | \aftergroup . . . . .      | 202                        |
| \above . . . . .                                          | 194                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | \alignmark . . . . .       | 787                        |
| \abovedisplayshortskip . . . . .                          | 195                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | \aligntab . . . . .        | 788                        |
| \abovedisplayskip . . . . .                               | 196                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | \asec . . . . .            | 215                        |
| \abovewithdelims . . . . .                                | 197                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | \asecd . . . . .           | 215                        |
| \abs . . . . .                                            | 213                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | \asin . . . . .            | 215                        |
| \accent . . . . .                                         | 198                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | \asind . . . . .           | 215                        |
| assert commands:                                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \assert_int:n . . . . .                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| atan . . . . .                                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| atand . . . . .                                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \AtBeginDocument . . . . .                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| 28719, 28720, 28729, 28730, 31397                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \atop . . . . .                                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \atopwithdelims . . . . .                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \attribute . . . . .                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \attributedef . . . . .                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \automaticdiscretionary . . . . .                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \automatichyphenmode . . . . .                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \automatichyphenpenalty . . . . .                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \autospacing . . . . .                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \autoxspacing . . . . .                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
|                                                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| <b>B</b>                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \b . . . . .                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \badness . . . . .                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \baselineskip . . . . .                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \batchmode . . . . .                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \begin . . . . .                                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| 226, 230, 14118, 14121, 18690, 29052, 29560, 29567, 31667 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| begin internal commands:                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| __regex_begin . . . . .                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \begincsname . . . . .                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \begingroup . . . . .                                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| 3, 20, 24, 29, 33, 51, 107, 185, 208                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \beginL . . . . .                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \beginR . . . . .                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \belowdisplayshortskip . . . . .                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \belowdisplayskip . . . . .                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \bfseries . . . . .                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \binoppenalty . . . . .                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \bodydir . . . . .                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |
| \bodydirection . . . . .                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                            |                            |

bool commands:

- \bool\_case\_false:n ..... 269, [32197](#)
- \bool\_case\_false:nTF ..... 269, [32197](#), 32207, 32209
- \bool\_case\_true:n ..... 269, [32197](#)
- \bool\_case\_true:nTF ..... 269, [32197](#), 32199, 32201
- \bool\_const:Nn ..... 108, [9082](#)
- \bool\_do\_until:Nn ..... 111, [9279](#)
- \bool\_do\_until:nn ..... 112, [9285](#)
- \bool\_do\_while:Nn ..... 111, [9279](#)
- \bool\_do\_while:nn ..... 112, [9285](#)
- .bool\_gset:N ..... 187, [15265](#)
- \bool\_gset:Nn ..... 108, [9104](#)
- \bool\_gset\_eq:NN ..... 108, [9100](#), 23987, 25855
- \bool\_gset\_false:N ..... 108, 5511, 5520, [9088](#), 25814, 32193
- .bool\_gset\_inverse:N ..... 187, [15273](#)
- \bool\_gset\_inverse:N ..... 268, [32189](#)
- \bool\_gset\_true:N ..... 108, 5501, [9088](#), 9540, 9546, 25865, 32193
- \bool\_if:NTF ..... 108, 149, 2051, 5515, 5524, [9121](#), 9274, 9276, 9280, 9282, 9538, 9544, 13329, 13336, 15016, 15232, 15241, 15431, 15433, 15435, 15481, 15483, 15485, 15523, 15525, 15527, 15543, 15545, 15547, 15586, 15627, 15646, 15648, 15653, 15660, 15724, 15753, 15763, 15791, 24858, 24867, 25270, 25348, 25366, 25395, 25492, 25712, 25720, 26974, 26979, 28109, 28824, 30204, 32190, 32193, 32215, 32596, 32669
- \bool\_if:nTF ..... 108, 110, 112, 112, 112, 593, 9135, [9153](#), 9224, 9231, 9250, 9257, 9266, 9287, 9296, 9300, 9309, 9379, 25351, 32220, 32226, 32280, 32619
- \bool\_if\_exist:NTF ..... 109, [9149](#), 15033, 15049
- \bool\_if\_exist\_p:N ..... 109, [9149](#)
- \bool\_if\_p:N ..... 108, [9121](#)
- \bool\_if\_p:n ..... 110, 534, 9085, 9107, 9112, [9153](#), [9161](#), 9231, 9257, 9263, 9267
- \bool\_lazy\_all:nTF ..... 110, 110, 110, [9211](#), 25135
- \bool\_lazy\_all\_p:n ..... 110, [9211](#)
- \bool\_lazy\_and:nnTF ..... 110, 110, 110, [9228](#), 9451, 9669, 12927, 13431, 14105, 22747, 29801, 29837, 30652, 30705, 30733, 31571
- \bool\_lazy\_and\_p:nn ..... 110, [110](#), [9228](#), 30193
- \bool\_lazy\_any:nTF ..... 110, 111, 111, 5643, 5667, 5689, 5859, [9237](#), 13720, 29530, 29633
- \bool\_lazy\_any\_p:n ..... 110, 111, [9237](#), 13436, 30655
- \bool\_lazy\_or:nnTF ..... 110, 111, 6641, [9254](#), 9434, 9507, 10870, 22856, 22882, 23408, 29250, 29438, 29650, 29658, 29828, 30132, 30190, 30206, 30211, 30246, 30294, 30330, 30393, 30432, 30536, 30550, 30584, 30591, 30669, 30714, 30747, 30778, 30825, 30848, 30884, 31685, 31770
- \bool\_lazy\_or\_p:nn ..... 111, [9254](#), 29840, 30435, 30553, 30736, 31574
- \bool\_log:N ..... 108, [9136](#)
- \bool\_log:n ..... 109, [9130](#)
- \bool\_new:N ..... 107, 5364, 9080, 9145, 9146, 9147, 9148, 9534, 9535, 13057, 14921, 14922, 14929, 14930, 14934, 15033, 15049, 23838, 24265, 25771, 25772, 25774, 25775, 25776, 27709, 29880, 32566, 32582
- \bool\_not\_p:n ..... 111, [9263](#), 13434
- .bool\_set:N ..... 187, [15265](#)
- \bool\_set:Nn ..... 108, 527, [9104](#)
- \bool\_set\_eq:NN ..... 108, [9100](#), 23981, 26008
- \bool\_set\_false:N ..... 108, 164, [9088](#), 13163, 13305, 13313, 13321, 13331, 13338, 14958, 15425, 15426, 15427, 15477, 15478, 15482, 15518, 15526, 15528, 15537, 15538, 15548, 15569, 15634, 24832, 25037, 25750, 25827, 25841, 25887, 25949, 28105, 28822, 32190, 32585
- .bool\_set\_inverse:N ..... 187, [15273](#)
- \bool\_set\_inverse:N ..... 268, [32189](#)
- \bool\_set\_true:N ..... 108, 178, [9088](#), 13291, 14953, 15432, 15434, 15436, 15476, 15484, 15486, 15519, 15520, 15524, 15539, 15544, 15546, 15564, 15641, 24837, 25041, 25744, 25947, 26007, 28123, 28145, 28176, 29881, 32190, 32595
- \bool\_show:N ..... 108, [9136](#)
- \bool\_show:n ..... 108, [9130](#)
- \bool\_until\_do:Nn ..... 111, [9273](#)
- \bool\_until\_do:nn ..... 112, [9285](#)
- \bool\_while\_do:Nn ..... 111, [9273](#)
- \bool\_while\_do:nn ..... 112, [9285](#)
- \bool\_xor:nnTF ..... 111, [9264](#)
- \bool\_xor\_p:nn ..... 111, [9264](#)

```

\c_false_bool ... 22, 107, 326, 359,
 528, 530, 531, 531, 531, 532, 1628,
 1680, 1681, 1712, 1731, 1736, 1768,
 1787, 1988, 1995, 2903, 3177, 9080,
 9091, 9095, 9202, 9225, 9231, 9249,
 9390, 24509, 24527, 24741, 24777,
 25076, 25218, 25235, 25291, 25428,
 26621, 32206, 32208, 32210, 32212
\g_tmpa_bool 109, 9145
\l_tmpa_bool 109, 9145
\g_tmpb_bool 109, 9145
\l_tmpb_bool 109, 9145
\c_true_bool 22, 107, 326, 392, 528,
 530, 531, 531, 531, 532, 1680, 1712,
 1768, 1786, 2009, 9089, 9093, 9203,
 9204, 9223, 9251, 9257, 9384, 23845,
 23984, 24413, 24473, 24523, 24707,
 24732, 24776, 24783, 25216, 25226,
 25289, 25478, 25650, 25661, 25676,
 25831, 25864, 26452, 26529, 26582,
 32198, 32200, 32202, 32204, 32623
bool internal commands:
__bool_!:_Nw 9182
__bool_&_0: 9194
__bool_&_1: 9194
__bool_&_2: 9194
__bool_(:Nw 9187
__bool_)_0: 9194
__bool_)_1: 9194
__bool_)_2: 9194
__bool_case:NnTF 32197
__bool_case_end:nw 32197
__bool_case_false:w 32197
__bool_case_true:w 32197
__bool_choose>NNN .. 9189, 9193, 9194
__bool_get_next>NN
 . 531, 531, 9169, 9172, 9184, 9190,
 9205, 9206, 9207, 9208, 9209, 9210
__bool_if_p:n 9161
__bool_if_p_aux:w 530, 9161
__bool_if_recursion_tail_stop-
 do:nn 9120, 9223, 9249
__bool_lazy_all:n 9211
__bool_lazy_any:n 9237
__bool_p:Nw 9192
__bool_show>NN 9136
__bool_to_str:n 9130, 9143
__bool_use_i_delimit_by_q-
 recursion_stop:nw 9118, 9225, 9251
__bool_|_0: 9194
__bool_|_1: 9194
__bool_|_2: 9194
\botmark 212
\botmarks 518
\box 213
box commands:
\box_autosize_to_wd_and_ht:Nnn ...
 247, 27615
\box_autosize_to_wd_and_ht_plus-
 dp:Nnn 247, 27615
\box_clear:N 239,
 239, 27016, 27023, 27744, 27831, 27905
\box_clear_new:N 239, 27022
\box_clip:N 265, 266, 266, 32047
\box_dp:N 240, 17214, 27044,
 27053, 27057, 27360, 27489, 27604,
 27623, 27629, 27942, 27943, 28049,
 28054, 28082, 28096, 28269, 28547,
 28568, 28889, 32067, 32074, 32079
\box_gautosize_to_wd_and_ht:Nnn .
 247, 27615
\box_gautosize_to_wd_and_ht-
 plus_dp:Nnn 247, 27615
\box_gclear:N 239, 27016, 27025, 27753
\box_gclear_new:N 239, 27022
\box_gclip:N 265, 32047
\box_gresize_to_ht:Nn 247, 27508
\box_gresize_to_ht_plus_dp:Nn ...
 248, 27508
\box_gresize_to_wd:Nn ... 248, 27508
\box_gresize_to_wd_and_ht:Nnn ...
 248, 27508
\box_gresize_to_wd_and_ht_plus-
 dp:Nnn 248, 27459, 28383
\box_grotate:Nn ... 249, 27341, 28216
\box_gscale:Nnn ... 249, 27586, 28425
\box_gset_dp:Nn 240, 27050
\box_gset_eq>NNN ...
 . 239, 27019, 27028, 27927, 32057, 32108
\box_gset_eq_clear>NNN 32906
\box_gset_eq_drop:N 32909
\box_gset_eq_drop>NNN 246, 27034
\box_gset_ht:Nn 240, 27050
\box_gset_to_last:N 241, 27104
\box_gset_trim:Nnnnn 266, 32053
\box_gset_viewport:Nnnnn .. 266, 32104
\box_gset_wd:Nn 241, 27050
\box_ht:N 240, 17213, 27044,
 27062, 27066, 27359, 27488, 27603,
 27616, 27619, 27623, 27629, 27826,
 27900, 27944, 27945, 28040, 28045,
 28082, 28089, 28263, 28267, 28546,
 28567, 28887, 32084, 32092, 32097
\box_if_empty:NTF 241, 27100
\box_if_empty_p:N 241, 27100
\box_if_exist:NTF ...
 . 239, 27023, 27025, 27040, 27135
\box_if_exist_p:N 239, 27040

```

\box\_if\_horizontal:NTF . . . . . 241, [27092](#)  
\box\_if\_horizontal\_p:N . . . . . 241, [27092](#)  
\box\_if\_vertical:NTF . . . . . 241, [27092](#)  
\box\_if\_vertical\_p:N . . . . . 241, [27092](#)  
\box\_log:N . . . . . 242, [27121](#)  
\box\_log:Nnn . . . . . 242, [27121](#)  
\box\_move\_down:nn 240, [1188](#), [27081](#),  
28242, [32071](#), [32079](#), [32122](#), [32129](#)  
\box\_move\_left:nn . . . . . 240, [27081](#)  
\box\_move\_right:nn . . . . . 240, [27081](#)  
\box\_move\_up:nn 240, [27081](#), [28587](#),  
28884, [32088](#), [32097](#), [32136](#), [32149](#)  
\box\_new:N . . . . . 239,  
239, [27010](#), [27110](#), [27111](#), [27112](#),  
27113, [27114](#), [27340](#), [27685](#), [27760](#)  
\box\_resize:Nnn . . . . . 32707  
\box\_resize\_to\_ht:Nn . . . . . 247, [27508](#)  
\box\_resize\_to\_ht\_plus\_dp:Nn . . . . .  
248, [27508](#)  
\box\_resize\_to\_wd:Nn . . . . . 248, [27508](#)  
\box\_resize\_to\_wd\_and\_ht:Nnn . . . . .  
248, [27508](#)  
\box\_resize\_to\_wd\_and\_ht\_plus\_-  
dp:Nnn . . . . . 248, [27459](#), [28376](#), [32708](#)  
\box\_rotate:Nn . . . . . 249, [27341](#), [28213](#)  
\box\_scale:Nnn . . . . . 249, [27586](#), [28422](#)  
\box\_set\_dp:Nn . . . . .  
240, [1188](#), [27050](#), [27386](#),  
27657, [27660](#), [28247](#), [28547](#), [28568](#),  
28888, [32074](#), [32082](#), [32125](#), [32130](#)  
\box\_set\_eq:NN . . . . . 239, [27017](#), [27028](#),  
27915, [28570](#), [28892](#), [32054](#), [32105](#)  
\box\_set\_eq\_clear:NN . . . . . 32906  
\box\_set\_eq\_drop:N . . . . . 32906  
\box\_set\_eq\_drop:NN . . . . . 246, [27034](#)  
\box\_set\_ht:Nn . . . . . 240, [27050](#), [27385](#),  
27656, [27661](#), [28245](#), [28546](#), [28567](#),  
28886, [32091](#), [32100](#), [32139](#), [32152](#)  
\box\_set\_to\_last:N . . . . . 241, [27104](#)  
\box\_set\_trim:Nnnnn . . . . . 266, [32053](#)  
\box\_set\_viewport:Nnnnn . . . . . 266, [32104](#)  
\box\_set\_wd:Nn . . . . . 241, [27050](#), [27387](#),  
27673, [28248](#), [28548](#), [28569](#), [28890](#)  
\box\_show:N . . . . . 242, [246](#), [27115](#)  
\box\_show:Nnn . . . . . 242, [27115](#)  
\box\_use:N . . . . . 239, [240](#),  
240, [27077](#), [27374](#), [28243](#), [28584](#),  
28587, [28881](#), [28884](#), [32064](#), [32115](#)  
\box\_use\_clear:N . . . . . 32709  
\box\_use\_drop:N . . . . . 246, [27077](#),  
27389, [27668](#), [27677](#), [28250](#), [28670](#),  
28816, [32072](#), [32080](#), [32089](#), [32098](#),  
32123, [32129](#), [32137](#), [32150](#), [32710](#)  
\box\_wd:N . . . . . 240,  
17212, [27044](#), [27071](#), [27075](#), [27361](#),  
27490, [27605](#), [27637](#), [27946](#), [27947](#),  
28044, [28053](#), [28071](#), [28076](#), [28266](#),  
28274, [28468](#), [28475](#), [28501](#), [28548](#),  
28569, [28585](#), [28882](#), [28891](#), [32116](#)  
\c\_empty\_box . . . . .  
239, [241](#), [241](#), [27017](#), [27019](#), [27110](#)  
\g\_tmpa\_box . . . . . 242, [27111](#)  
\l\_tmpa\_box . . . . . 242, [27111](#)  
\g\_tmpb\_box . . . . . 242, [27111](#)  
\l\_tmpb\_box . . . . . 242, [27111](#)  
box internal commands:  
\l\_box\_angle\_fp . . . . .  
27329, [27351](#), [27352](#), [27353](#), [27382](#)  
\l\_box\_autosize:NnnnN . . . . . 27615  
\l\_box\_backend\_clip:N . . . . . 32048, [32051](#)  
\l\_box\_backend\_rotate:Nn . . . . . 27380  
\l\_box\_backend\_scale:Nnn . . . . . 27649  
\l\_box\_bottom\_dim . . . . . 27332,  
27360, [27417](#), [27421](#), [27426](#), [27432](#),  
27437, [27441](#), [27450](#), [27452](#), [27481](#),  
27489, [27498](#), [27542](#), [27604](#), [27610](#)  
\l\_box\_bottom\_new\_dim . . . . .  
27336, [27386](#), [27418](#), [27429](#), [27440](#),  
27451, [27497](#), [27609](#), [27657](#), [27661](#)  
\l\_box\_cos\_fp . . . . . 27330,  
27353, [27365](#), [27370](#), [27397](#), [27409](#)  
\l\_box\_dim\_eval:n . . . . .  
27007, [27053](#), [27057](#), [27062](#),  
27066, [27071](#), [27075](#), [27082](#), [27084](#),  
27086, [27088](#), [27167](#), [27172](#), [27199](#),  
27205, [27213](#), [27237](#), [27271](#), [27276](#),  
27304, [27310](#), [27321](#), [27326](#), [32063](#),  
32065, [32114](#), [32116](#), [32125](#), [32149](#)  
\l\_box\_dim\_eval:w . . . . . 27007  
\l\_box\_internal\_box 27340, [27374](#),  
27375, [27381](#), [27385](#), [27386](#), [27387](#),  
27389, [27647](#), [27656](#), [27657](#), [27660](#),  
27661, [27668](#), [27673](#), [27677](#), [32061](#),  
32069, [32072](#), [32074](#), [32077](#), [32080](#),  
32082, [32084](#), [32086](#), [32089](#), [32091](#),  
32092, [32095](#), [32097](#), [32098](#), [32100](#),  
32102, [32112](#), [32120](#), [32123](#), [32125](#),  
32128, [32129](#), [32130](#), [32134](#), [32137](#),  
32139, [32147](#), [32150](#), [32152](#), [32154](#)  
\l\_box\_left\_dim . . . . . 27332, [27362](#),  
27417, [27419](#), [27428](#), [27432](#), [27437](#),  
27443, [27448](#), [27452](#), [27491](#), [27606](#)  
\l\_box\_left\_new\_dim 27336, [27377](#),  
27388, [27420](#), [27431](#), [27442](#), [27453](#)  
\l\_box\_log:nNnn . . . . . 27121  
\l\_box\_resize:N . . . . .  
27459, [27525](#), [27545](#), [27562](#), [27583](#)

```

__box_resize:NNN 27459
__box_resize_common:N 27501, 27613, 27645
__box_resize_set_corners:N .. 27459, 27518, 27538, 27558, 27575
__box_resize_to_ht:NnN 27508
__box_resize_to_ht_plus_dp:NnN .. 27508
__box_resize_to_wd:NnN 27508
__box_resize_to_wd_and_ht:NnnN .. 27566, 27569, 27571
__box_resize_to_wd_and_ht_plus_dp:NnnN .. 27459
__box_resize_to_wd_ht:NnnN .. 27508
\l_box_right_dim .. 27332, 27361, 27415, 27421, 27426, 27430, 27439, 27441, 27450, 27454, 27477, 27490, 27496, 27560, 27577, 27605, 27612
\l_box_right_new_dim .. 27336, 27388, 27422, 27433, 27444, 27455, 27495, 27611, 27665, 27667, 27673
__box_rotate:N 27341
__box_rotate:NnnN 27341
__box_rotate_quadrant_four: .. 27341, 27446
__box_rotate_quadrant_one: .. 27341, 27413
__box_rotate_quadrant_three: .. 27341, 27435
__box_rotate_quadrant_two: .. 27341, 27424
__box_rotate_xdir:nnN 27341, 27391, 27419, 27421, 27430, 27432, 27441, 27443, 27452, 27454
__box_rotate_ydir:nnN 27341, 27402, 27415, 27417, 27426, 27428, 27437, 27439, 27448, 27450
__box_scale:N 27586, 27642
__box_scale:NnnN 27586
\l_box_scale_x_fp .. 27457, 27476, 27496, 27524, 27544, 27559, 27561, 27576, 27596, 27612, 27637, 27639, 27640, 27641, 27651, 27663
\l_box_scale_y_fp .. 27457, 27478, 27498, 27500, 27519, 27524, 27539, 27544, 27561, 27578, 27597, 27608, 27610, 27638, 27639, 27640, 27641, 27652, 27654
__box_set_trim:NnnnnN 32053
__box_set_viewport:NnnnnN .. 32105, 32108, 32110
__box_show>NNnn .. 27119, 27129, 27133
\l_box_sin_fp .. 27330, 27352, 27363, 27398, 27408
\l_box_top_dim 27332, 27359, 27415, 27419, 27428, 27430, 27439, 27443, 27448, 27454, 27481, 27488, 27500, 27522, 27542, 27581, 27603, 27608
\l_box_top_new_dim .. 27336, 27385, 27416, 27427, 27438, 27449, 27499, 27607, 27656, 27660
__box_viewport:NnnnnN 32104
\boxdir 799
\boxdirection 800
\boxmaxdepth 214
\bp 218
\breakafterdirmode 801
\brokenpenalty 215

C
\c .. 29543, 31808, 31829, 31855, 31912, 31913, 31932, 31933, 31936, 31937, 31944, 31945, 31956, 31957, 31964, 31965, 31968, 31969, 32024, 32025
\catcode 124, 125, 126, 127, 128, 129, 130, 131, 132, 136, 137, 138, 139, 140, 141, 142, 143, 144, 216
catcode commands:
 \c_catcode_active_space_t1 273, 32509
 \c_catcode_active_t1 134, 581, 10898, 10958
 \c_catcode_letter_token 134, 581, 10831, 10880, 10948, 23598, 29511
 \c_catcode_other_space_t1 130, 645, 10877, 13071, 13115, 13195, 13284, 13360
 \c_catcode_other_token 134, 581, 10834, 10880, 10953, 23596, 29514
\catcodetable 802
cc 218
cctab commands:
 \cctab_begin:N .. 222, 222, 934, 935, 938, 939, 940, 940, 941, 942, 22713
 \cctab_const:Nn 222, 222, 22829, 22836, 22843, 22887
 \cctab_end: 222, 222, 934, 935, 938, 939, 940, 941, 942, 22727
 \cctab_gset:Nn 222, 222, 22643, 22832
 \cctab_if_exist:N 22784
 \cctab_if_exist:NTF 223, 22791
 \cctab_if_exist_p:N 223
 \cctab_new:N 222, 935, 935, 22578, 22831, 22835
 \cctab_select:N .. 46, 46, 222, 222, 222, 22648, 22665, 22838, 22845, 22889
 \c_code_cctab 223, 22848
 \c_document_cctab ... 223, 937, 22848
 \c_initex_cctab ... 223, 22648, 22835

```

\c\_other\_cctab ..... 223, 22835  
 \c\_str\_cctab ..... 223, 938, 22835  
 cctab internal commands:  
   \g\_\_cctab\_allocate\_int .....  
     ..... 22574, 22706, 22708, 22710  
   \\_\_cctab\_begin\_aux: .....  
     ..... 939, 939, 22694, 22718  
   \\_\_cctab\_chk\_group\_begin:n .....  
     ..... 940, 22719, 22738  
   \\_\_cctab\_chk\_group\_end:n .....  
     ..... 940, 22732, 22738  
   \\_\_cctab\_chk\_if\_valid:NTF .....  
     ..... 22645, 22666, 22715, 22788  
   \\_\_cctab\_chk\_if\_valid\_aux:NTF . 22788  
   \g\_\_cctab\_endlinechar\_prop .....  
     .... 936, 22577, 22624, 22626, 22673  
   \g\_\_cctab\_group\_seq .....  
     ..... 22573, 22740, 22746  
   \\_\_cctab\_gset:n .....  
     ..... 22616, 22650, 22722, 22885  
   \\_\_cctab\_gset\_aux:n ..... 22616  
   \\_\_cctab\_gstore:Nnn ..... 22578  
   \l\_\_cctab\_internal\_a\_t1 .....  
     939, 939, 940, 22575, 22673, 22674,  
     22699, 22709, 22717, 22720, 22721,  
     22722, 22729, 22731, 22733, 22734  
   \l\_\_cctab\_internal\_b\_t1 .....  
     ..... 22575, 22746, 22750, 22757  
   \g\_\_cctab\_internal\_cctab ..... 22655  
   \\_\_cctab\_internal\_cctab\_name: ...  
     .. 22655, 22676, 22677, 22678, 22679  
   \\_\_cctab\_nesting\_number:N .....  
     ..... 22720, 22733, 22762  
   \\_\_cctab\_nesting\_number:w ..... 22762  
   \\_\_cctab\_new:N .. 935, 939, 22578,  
     22657, 22677, 22698, 22707, 22852  
   \g\_\_cctab\_next\_cctab ..... 22694  
   \\_\_cctab\_select:N .....  
     ..... 938, 22665, 22723, 22734  
   \g\_\_cctab\_stack\_seq .....  
     934, 935, 22571, 22721, 22729, 22780  
   \g\_\_cctab\_unused\_seq .....  
     934, 935, 939, 940, 22571, 22717, 22731  
 ceil ..... 214  
 \char ..... 217, 11064  
 char commands:  
   \l\_char\_active\_seq .. 133, 158, 10507  
   \char\_fold\_case:N ..... 33001  
   \char\_foldcase:N .....  
     ..... 130, 10756, 33007, 33008  
   \char\_generate:nn 45, 130, 386, 440,  
     1050, 1200, 3816, 3832, 5425, 5698,  
     5714, 10534, 10749, 10753, 10784,  
     10813, 10868, 10877, 12644, 24135,  
       25145, 29442, 29480, 30148, 30158,  
       30159, 30303, 30396, 30397, 30568,  
       30579, 30626, 30627, 30628, 30639,  
       30664, 30692, 30719, 30742, 30759,  
       30771, 30783, 30788, 30813, 30831,  
       30860, 30862, 30866, 30904, 30905,  
       30910, 30912, 30920, 30921, 30926,  
       30928, 31148, 31149, 31154, 31156,  
       31185, 31186, 31203, 31204, 31205,  
       31210, 31212, 31214, 31222, 31223,  
       31224, 31229, 31231, 31233, 31345,  
       31346, 31347, 31352, 31354, 31694,  
       31715, 31717, 31776, 31790, 31792  
   \char\_gset\_active\_eq>NN .. 129, 10513  
   \char\_gset\_active\_eq:nN .. 129, 10513  
   \char\_lower\_case:N ..... 33001  
   \char\_lowercase:N .....  
     ..... 130, 10756, 33001, 33002  
   \char\_mixed\_case:N ..... 33006  
   \char\_mixed\_case:Nn ..... 33001  
   \char\_set\_active\_eq>NN .. 129, 10513  
   \char\_set\_active\_eq:nN .. 129, 10513  
   \char\_set\_catcode:nn .. 132, 153,  
     154, 155, 156, 157, 158, 159, 160,  
     161, 10413, 10420, 10422, 10424,  
     10426, 10428, 10430, 10432, 10434,  
     10436, 10438, 10440, 10442, 10444,  
     10446, 10448, 10450, 10452, 10454,  
     10456, 10458, 10460, 10462, 10464,  
     10466, 10468, 10470, 10472, 10474,  
     10476, 10478, 10480, 10482, 22687  
   \char\_set\_catcode\_active:N .....  
     . 131, 10419, 10514, 10571, 10899,  
     11898, 23402, 26306, 31676, 32510  
   \char\_set\_catcode\_active:n .....  
     . 131, 10451, 10622, 14839, 14840,  
     22859, 22866, 22884, 22895, 29410  
   \char\_set\_catcode\_alignment:N .....  
     . 131, 5631, 10419, 10887, 26358  
   \char\_set\_catcode\_alignment:n .....  
     . 131, 171, 10451, 10606, 22872  
   \char\_set\_catcode\_comment:N .....  
     . 131, 5640, 10419  
   \char\_set\_catcode\_comment:n .....  
     . 131, 10451, 22871  
   \char\_set\_catcode\_end\_line:N ...  
     ..... 131, 10419  
   \char\_set\_catcode\_end\_line:n ...  
     ..... 131, 10451, 22867  
   \char\_set\_catcode\_escape:N .....  
     ..... 131, 5627, 10419  
   \char\_set\_catcode\_escape:n .....  
     ..... 131, 10451, 22874

```

\char_set_catcode_group_begin:N 131, 5628, 10419, 23475, 26309
\char_set_catcode_group_begin:n 131, 10451, 10599, 22877
\char_set_catcode_group_end:N 131, 5629, 10419, 23478, 26326
\char_set_catcode_group_end:n 131, 10451, 10601, 22879
\char_set_catcode_ignore:N 131, 5634, 10419
\char_set_catcode_ignore:n 131, 168, 169, 10451, 22864, 22868
\char_set_catcode_invalid:N 131, 10419
\char_set_catcode_invalid:n 131, 10451, 22855, 22858, 22881
\char_set_catcode_letter:N 131, 5637, 10419, 18831, 18832, 26335
\char_set_catcode_letter:n 131, 172, 174, 10451, 10618, 22861, 22863, 22873, 22876
\char_set_catcode_math_subscript:N 131, 10419, 10891, 26323
\char_set_catcode_math_subscript:n 131, 10451, 10613, 22894
\char_set_catcode_math_superscript:N 131, 5633, 10419, 26361
\char_set_catcode_math_superscript:n 131, 173, 10451, 10611, 22875
\char_set_catcode_math_toggle:N 131, 5630, 10419, 10885, 26338
\char_set_catcode_math_toggle:n 131, 10451, 10604, 22870
\char_set_catcode_other:N 131, 937, 5639, 5989, 5990, 6323, 6324, 6505, 6506, 6507, 10419, 23635, 26341
\char_set_catcode_other:n 131, 170, 175, 10451, 10573, 10620, 22841, 22860, 22862, 22865, 22878, 22893
\char_set_catcode_parameter:N 131, 5632, 10419, 26344
\char_set_catcode_parameter:n 131, 10451, 10609, 22869
\char_set_catcode_space:N 131, 5635, 10419
\char_set_catcode_space:n 131, 10451, 10616, 14050, 22846, 22880, 22891, 22892, 29299, 32890
\char_set_lccode:mn 132, 10483, 10520, 10626, 10627, 11894, 11895, 11896, 11897, 32511
\char_set_mathcode:nn 133, 10483
\char_set_sfcode:nn 133, 10483
\char_set_uccode:nn 132, 10483
\char_show_value_catcode:n 132, 10413
\char_show_value_lccode:n 132, 10483
\char_show_value_mathcode:n 133, 10483
\char_show_value_sfcode:n 133, 10483
\char_show_value_uccode:n 133, 10483
\l_char_special_seq 133, 10507
\char_str_fold_case:N 33001
\char_str_foldcase:N 130, 10756, 33015, 33016
\char_str_lower_case:N 33001
\char_str_lowercase:N 130, 10756, 33009, 33010
\char_str_mixed_case:N 33014
\char_str_mixed_case:Nn 33001
\char_str_titlecase:N 130, 10756, 33013, 33014
\char_str_upper_case:N 33001
\char_str_uppercase:N 130, 10756, 33011, 33012
\char_titlecase:N 130, 10756, 33005, 33006
\char_to_nfd:N 273, 10734, 30339
\char_to_utfviii_bytes:n 273, 6654, 10656, 30871, 30894, 30895, 31162, 31163, 31192, 31360, 31361, 31707, 31784
\char_upper_case:N 33001
\char_uppercase:N 130, 10756, 30350, 30359, 30371, 30375, 30386, 30403, 33003, 33004
\char_value_catcode:n 132, 153, 154, 155, 156, 157, 158, 159, 160, 161, 3828, 3832, 10413, 10753, 22637, 29443, 30814, 31695, 31777
\char_value_lccode:n 132, 10483, 10757, 10770, 10847, 10857, 29314
\char_value_mathcode:n 133, 10483
\char_value_sfcode:n 133, 10483
\char_value_uccode:n 133, 10483, 10759, 10849
char internal commands:
__char_change_case>NN 10756
__char_change_case:nN 10756
__char_change_case:NN 10756
__char_change_case:nNN 10756
__char_change_case:nN 10756
__char_change_case:NNNN 10756
__char_change_case_catcode:N 10749, 10756
__char_change_case_multi:nN 10756
__char_change_case_multi:NNNW 10756

```

\\_\_char\_data\_auxi:w ..... 29265,  
   29305, 29310, 29338, 29343, 29376  
 \\_\_char\_data\_auxii:w .....  
   ..... 29271, 29275, 29321,  
   29325, 29346, 29347, 29349, 29351  
 \\_\_char\_data\_auxiii:w .. 29273, 29285  
 \g\_\_char\_data\_ior .. 29249, 29264,  
   29300, 29308, 29309, 29335, 29341,  
   29342, 29365, 29378, 29394, 29395  
 \\_\_char\_generate:n ..... 29255,  
   29280, 29282, 29294, 29317, 29329,  
   29330, 29332, 29358, 29359, 29361  
 \\_\_char\_generate\_aux:nn ..... 10534  
 \\_\_char\_generate\_aux:nw ..... 10534  
 \\_\_char\_generate\_aux:w .. 10536, 10540  
 \\_\_char\_generate\_auxii:nw ... 10534  
 \\_\_char\_generate\_char:n .. 29253,  
   29278, 29293, 29316, 29327, 29356  
 \\_\_char\_generate\_invalid\_-  
   catcode: ..... 10534  
 \\_\_char\_int\_to\_roman:w .....  
   ..... 10533, 10631, 10650  
 \\_\_char\_quark\_if\_no\_value:NTF ...  
   ..... 10412, 10791, 10793  
 \\_\_char\_quark\_if\_no\_value\_p:N .. 10412  
 \\_\_char\_str\_change\_case:nN .. 10756  
 \\_\_char\_str\_change\_case:nNN .. 10756  
 \\_\_char\_tmp:n .....  
   ..... 10624, 10635, 10638, 10640  
 \\_\_char\_tmp:NN .. 29383, 29389, 29391  
 \\_\_char\_tmp:nN .. 10515, 10526, 10527  
 \l\_\_char\_tmp\_t1 ..... 10534  
 \l\_\_char\_tmpa\_t1 ..... 29288,  
   29289, 29291, 29300, 29302, 29305  
 \l\_\_char\_tmpb\_t1 ..... 29290, 29291  
 \\_\_char\_to\_nfd:n ..... 10734  
 \\_\_char\_to\_nfd:Nw ..... 10734  
 \\_\_char\_to\_utfviii\_bytes\_auxi:n ..  
   ..... 10656  
 \\_\_char\_to\_utfviii\_bytes\_-  
   auxii:Nnn ..... 10656  
 \\_\_char\_to\_utfviii\_bytes\_-  
   auxiii:n ..... 10656  
 \\_\_char\_to\_utfviii\_bytes\_end: .. 10656  
 \\_\_char\_to\_utfviii\_bytes\_-  
   output:nnn ..... 10656  
 \\_\_char\_to\_utfviii\_bytes\_-  
   outputi:nw ..... 10656  
 \\_\_char\_to\_utfviii\_bytes\_-  
   outputii:nw ..... 10656  
 \\_\_char\_to\_utfviii\_bytes\_-  
   outputiv:nw ..... 10656  
 \chardef ..... 134, 146, 218, 942  
 choice commands:  
   .choice: ..... 187, 15281  
 choices commands:  
   .choices:nn ..... 187, 15283  
 \cite ..... 29560, 29567  
 \cleaders ..... 219  
 \clearmarks ..... 803  
 clist commands:  
   \clist\_clear:N ..... 120,  
   120, 9871, 9888, 10051, 15464, 15506  
   \clist\_clear\_new:N ..... 120, 9875  
   \clist\_concat:NNN 120, 9914, 9940, 9953  
   \clist\_const:Nn ..... 120, 9868  
   \clist\_count:N .....  
   ..... 125, 127, 10251, 10280, 10312, 10378  
   \clist\_count:n 125, 10251, 10343, 10369  
   \clist\_gclear:N .... 120, 9871, 9890  
   \clist\_gclear\_new:N ..... 120, 9875  
   \clist\_gconcat:NNN .....  
   ..... 120, 9914, 9942, 9955  
   \clist\_get>NN ..... 126, 9965  
   \clist\_get:NNTF ..... 126, 10002  
   \clist\_gpop>NN ..... 126, 9976  
   \clist\_gpop:NNTF ..... 127, 10002  
   \clist\_gpush:Nn ..... 127, 10027  
   \clist\_gput\_left:Nn .....  
   ... 121, 9939, 10035, 10036, 10037,  
   10038, 10039, 10040, 10041, 10042  
   \clist\_gput\_right:Nn ..... 121, 9952  
   \clist\_gremove\_all:Nn ... 122, 10061  
   \clist\_gremove\_duplicates:N .....  
   ..... 122, 10045  
   \clist\_greverse:N ..... 122, 10100  
   \clist\_gset:N ..... 187, 15293  
   \clist\_gset:Nn ..... 121, 5727, 9933  
   \clist\_gset\_eq>NN .. 120, 9879, 10048  
   \clist\_gset\_from\_seq>NN .....  
   ..... 120, 9887, 10064, 23085  
   \clist\_gsort:Nn ... 122, 10118, 23070  
   \clist\_if\_empty:NTF .....  
   ..... 123, 9923, 10085, 10118,  
   10175, 10205, 10225, 10377, 15143  
   \clist\_if\_empty:nTF ..... 123, 10122  
   \clist\_if\_empty\_p:N ..... 123, 10118  
   \clist\_if\_empty\_p:n ..... 123, 10122  
   \clist\_if\_exist:NTF .....  
   ..... 120, 9929, 10278, 13918, 14018  
   \clist\_if\_exist\_p:N ..... 120, 9929  
   \clist\_if\_in:NnTF .....  
   ..... 119, 123, 10054, 10136  
   \clist\_if\_in:nnTF .. 123, 10136, 16599  
   \clist\_item:Nn .....  
   ..... 127, 566, 566, 10309, 10378

```

\clist_item:nn 127, 566, 10340, 10373
\clist_log:N 128, 10381
\clist_log:n 128, 10395
\clist_map_break:
..... 124, 10179, 10184, 10193,
10197, 10213, 10231, 10247, 15688
\clist_map_break:n ... 125, 10156,
10247, 15642, 15717, 23078, 23084
\clist_map_function:NN .. 38, 123,
7562, 7572, 10159, 10173, 10256, 10391
\clist_map_function:Nn 563
\clist_map_function:nN
..... 123, 270, 270, 563, 5730,
7567, 7577, 7588, 10189, 10400, 15819
\clist_map_inline:Nn
123, 124, 561, 9786, 10052, 10203,
15637, 15679, 15708, 23078, 23084
\clist_map_inline:nn
124, 3215, 10203, 14197,
15102, 15194, 16082, 28975, 28987
\clist_map_variable:NNn .. 124, 10223
\clist_map_variable:nNn .. 124, 10223
\clist_new:N
119, 120, 550, 9866, 10043,
10402, 10403, 10404, 10405, 14917
\clist_pop:NN 126, 9976
\clist_pop:NNTF 126, 10002
\clist_push:Nn 127, 10027
\clist_put_left:Nn
121, 9939, 10027, 10028, 10029,
10030, 10031, 10032, 10033, 10034
\clist_put_right:Nn
121, 9952, 10055, 15750, 15760, 15788
\clist_rand_item:N 127, 10368
\clist_rand_item:n .. 117, 127, 10368
\clist_remove_all:Nn 122, 9801, 10061
\clist_remove_duplicates:N
119, 122, 10045
\clist_reverse:N 122, 10100
\clist_reverse:n
122, 559, 10101, 10103, 10106
\clist_set:N 187, 15293
\clist_set:Nn 121, 9933, 9940, 9942,
9953, 9955, 10142, 10219, 10236, 15142
\clist_set_eq:NN
120, 9879, 10046, 15622
\clist_set_from_seq:NN
120, 9887, 10062, 23079
\clist_show:N 127, 128, 10381
\clist_show:n 128, 128, 10395
\clist_sort:Nn 122, 10118, 23070
\clist_use:Nn 126, 10276
\clist_use:Nnnn 125, 499, 10276
\c_empty_clist
128, 9810, 9967, 9982, 10004, 10018
\l_foo_clist 224
\g_tmptpa_clist 128, 10402
\l_tmptpa_clist 128, 10402
\g_tmptpb_clist 128, 10402
\l_tmptpb_clist 128, 10402
clist internal commands:
__clist_concat:NNNN 9914
__clist_count:n 10251
__clist_count:w 10251
__clist_get:wN 9965, 10007
__clist_if_empty_n:w 10122
__clist_if_empty_n:wNw 10122
__clist_if_in_return:nnN 10136
__clist_if_recursion_tail_-
break:nN 9818, 10184, 10197
__clist_if_recursion_tail_-
stop:n ... 9818, 9835, 10241, 10272
__clist_if_wrap:nTF
551, 9840, 9865, 9906, 10067, 10148
__clist_if_wrap:w 551, 9840
\l__clist_internal_clist
..... 554, 9811, 9945,
9946, 9958, 9959, 10142, 10143,
10144, 10219, 10220, 10236, 10237
\l__clist_internal_remove_clist .
.. 10043, 10051, 10054, 10055, 10057
\l__clist_internal_remove_seq .
..... 10043, 10069, 10070, 10071
__clist_item:nnnN 10309, 10342
__clist_item_n:nw 10340
__clist_item_n_end:n 10340
__clist_item_N_loop:nw 10309
__clist_item_n_loop:nw 10340
__clist_item_n_strip:n 10340
__clist_item_n_strip:w 10340
__clist_map_function:Nw
..... 561, 10173, 10210
__clist_map_function_n:Nn 562, 10189
__clist_map_unbrace:Nw .. 562, 10189
__clist_map_variable:Nnw 10223
__clist_pop:NNN 9976
__clist_pop:wN 9976
__clist_pop:wwNNN .. 555, 9976, 10021
__clist_pop_TF:NNN 10002
__clist_put_left>NNNn 9939
__clist_put_right>NNNn 9952
__clist_rand_item:nn 10368
__clist_remove_all: 10061
__clist_remove_all>NNNn 10061
__clist_remove_all:w ... 558, 10061
__clist_remove_duplicates>NN . 10045
__clist_reverse:wwNww ... 559, 10106

```

```

__clist_reverse_end:ww 559, 10106
__clist_sanitize:n 9827, 9869, 9934, 9936
__clist_sanitize:Nn 551, 9827
__clist_set_from_seq:n 9887
__clist_set_from_seq:NNNN 9887
__clist_show:NN 10381
__clist_show:Nn 10395
__clist_tmp:w 558, 9820, 10074,
 10096, 10150, 10159, 10163, 10165
__clist_trim_next:w 551,
 562, 9821, 9830, 9838, 10192, 10200
__clist_use:nwn 10276
__clist_use:nwwwnwn 564, 10276
__clist_use:wnn 10276
__clist_use_i_delimit_by_s-
 stop:nw 9814, 10336
__clist_use_none_delimit_by_s-
 stop:w 558, 9814, 10077, 10322, 10327
__clist_wrap_item:w 551, 9836, 9864
\closein 220
\closeout 221
\clubpenalties 519
\clubpenalty 222
\cm 218
code commands:
 .code:n 187, 15291
coffin commands:
 \coffin_attach:NnnNnnnn
 252, 1118, 28530
 \coffin_clear:N 250, 27740
 \coffin_display_handles:Nn 253, 28794
 \coffin_dp:N
 253, 27942, 28394, 28433, 28908
 \coffin_gattach:NnnNnnnn 252, 28530
 \coffin_gclear:N 250, 27740
 \coffin_gjoin:NnnNnnnn 252, 28479
 \coffin_gresize:Nnn 252, 28373
 \coffin_grotate:Nn 252, 28212
 \coffin_gsacle:Nnn 252, 28421
 \coffin_gset_eq:NN
 250, 27911, 28488, 28539
 \coffin_gset_horizontal_pole:Nnn
 251, 27972
 \coffin_gset_vertical_pole:Nnn
 251, 27972
 \coffin_ht:N
 253, 27942, 28394, 28433, 28907
 \coffin_if_exist:NTF 250, 27719, 27733
 \coffin_if_exist_p:N 250, 27719
 \coffin_join:NnnNnnnn 252, 28479
 \coffin_log_structure:N 254, 28894
 \coffin_mark_handle:Nnnn 253, 28749
\coffin_new:N
 250, 1093, 27758, 27935,
 27936, 27937, 27938, 27939, 27940,
 27941, 28663, 28673, 28674, 28675
\coffin_resize:Nnn 252, 28373
\coffin_rotate:Nn 252, 28212
\coffin_scale:Nnn 252, 28421
\coffin_scale:NnnNN 28421
\coffin_set_eq:NN 250, 27911,
 28482, 28533, 28561, 28589, 28810
\coffin_set_horizontal_pole:Nnn .
 251, 27972
\coffin_set_vertical_pole:Nnn .
 251, 27972
\coffin_show_structure:N
 253, 254, 28894
\coffin_typeset:Nnnnn 253, 28665
\coffin_wd:N
 253, 27942, 28390, 28437, 28909
\c_empty_coffin 254, 27935
\g_tmmpa_coffin 254, 27938
\l_tmmpa_coffin 254, 27938
\g_tmppb_coffin 254, 27938
\l_tmppb_coffin 254, 27938
coffin internal commands:
 __coffin_align:NnnNnnnnN . . .
 28493, 28544, 28565, 28572, 28668
 \l__coffin_aligned_coffin . . .
 27935, 28494,
 28495, 28499, 28505, 28508, 28511,
 28527, 28528, 28545, 28546, 28547,
 28548, 28549, 28552, 28556, 28560,
 28561, 28566, 28567, 28568, 28569,
 28570, 28603, 28619, 28669, 28670,
 28879, 28886, 28888, 28890, 28892
 \l__coffin_aligned_internal_coffin . . .
 27935, 28582, 28589
 __coffin_attach:NnnNnnnnN 28530
 __coffin_attach_mark:NnnNnnnn . .
 28530, 28756, 28772, 28788
 \l__coffin_bottom_corner_dim . .
 28208, 28242, 28246,
 28325, 28336, 28337, 28357, 28365
 \l__coffin_bounding_prop . .
 28204, 28231, 28262,
 28264, 28270, 28272, 28281, 28344
 \l__coffin_bounding_shift_dim . .
 28207, 28240, 28343, 28349, 28350
 __coffin_calculate_intersection:Nnn
 28101, 28574, 28577, 28872
 __coffin_calculate_intersection:nnnnnn
 28101
 __coffin_calculate_intersection:nnnnnnnn
 28101, 28823

```

```

__coffin_color:n
.. 28729, 28753, 28760, 28798, 28833
\c__coffin_corners_prop
.. 27688, 27765, 27961, 27968
\l__coffin_corners_prop
.. 28205, 28222, 28226, 28251,
28256, 28287, 28327, 28354, 28401,
28405, 28411, 28417, 28452, 28466
\l__coffin_cos_fp
.. 1101, 1103, 28202, 28221, 28308, 28317
__coffin_display_attach:Nnnn 28794
__coffin_display_coffin
.. 28673, 28810, 28816, 28881,
28882, 28887, 28889, 28891, 28892
\l__coffin_display_coord_coffin .
.. 28673, 28758,
28773, 28789, 28831, 28846, 28865
\l__coffin_display_font_t1 .
.. 28718, 28761, 28834
__coffin_display_handles_-
aux:nnnn 28794
__coffin_display_handles_-
aux:nnnnnn 28794
\l__coffin_display_handles_prop .
.. 28676, 28764, 28768, 28837, 28841
\l__coffin_display_offset_dim ...
.. 28713, 28790, 28791, 28866, 28867
\l__coffin_display_pole_coffin ..
.. 28673, 28751, 28757, 28796, 28829
\l__coffin_display_poles_prop ...
.. 28717, 28801,
28806, 28809, 28811, 28813, 28820
\l__coffin_display_x_dim
.. 28715, 28826, 28876
\l__coffin_display_y_dim
.. 28715, 28827, 28878
\c__coffin_empty_coffin 28663, 28668
\l__coffin_error_bool
.. 27709, 28105, 28109,
28123, 28145, 28176, 28822, 28824
__coffin_find_bounding_shift: ..
.. 28234, 28341
__coffin_find_bounding_shift_-
aux:nn 28341
__coffin_find_corner_maxima:N ..
.. 28233, 28321
__coffin_find_corner_maxima_-
aux:nn 28321
__coffin_get_pole:NnN
.. 27948, 28103, 28104, 28630, 28631,
28634, 28635, 28803, 28804, 28807
__coffin_greset_structure:N ...
.. 27754, 27958, 28022
__coffin_gupdate:N
.. 27793, 27806, 27862, 27880, 28014
__coffin_gupdate_corners:N
.. 28023, 28026
__coffin_gupdate_poles:N
.. 28024, 28057
__coffin_if_exist:NTF
.. 27731, 27742, 27751, 27773,
27786, 27811, 27843, 27856, 27885,
27913, 27925, 27980, 27998, 28902
\l__coffin_internal_box
.. 27685, 27820,
27826, 27831, 27894, 27900, 27905,
28236, 28245, 28247, 28248, 28250
\l__coffin_internal_dim
.. 27685, 28269, 28271, 28275,
28432, 28435, 28500, 28502, 28503
\l__coffin_internal_t1 ... 27685,
28601, 28602, 28604, 28765, 28766,
28769, 28770, 28778, 28783, 28838,
28839, 28842, 28843, 28852, 28857
__coffin_join:NnnNnnnnN 28479
\l__coffin_left_corner_dim
.. 28208, 28240, 28249,
28326, 28332, 28333, 28356, 28364
__coffin_mark_handle_aux:nnnnNn
.. 28749
__coffin_offset_corner:Nnnnn . 28610
__coffin_offset_corners:Nnn ...
.. 28516, 28517, 28523, 28524, 28610
__coffin_offset_pole:Nnnnnnn . 28591
__coffin_offset_poles:Nnn
.. 28514, 28515,
28520, 28521, 28557, 28558, 28591
\l__coffin_offset_x_dim
.. 27710, 28497, 28498, 28501,
28512, 28514, 28516, 28522, 28525,
28559, 28578, 28586, 28875, 28883
\l__coffin_offset_y_dim
.. 27710, 28515, 28517, 28522, 28525,
28559, 28580, 28587, 28877, 28884
\l__coffin_pole_a_t1
.. 27712, 28103, 28108, 28630, 28633,
28634, 28637, 28803, 28805, 28808
\l__coffin_pole_b_t1 ... 27712,
28104, 28108, 28631, 28633, 28635,
28637, 28804, 28805, 28807, 28808
\c__coffin_poles_prop
.. 27695, 27767, 27963, 27970
\l__coffin_poles_prop
.. 28205, 28224, 28228,
28253, 28258, 28295, 28362, 28403,
28407, 28413, 28419, 28458, 28473

```

```

__coffin_reset_structure:N . . .
 .. 27745, 27958, 28016, 28505, 28549
__coffin_resize:NnnNN 28373
__coffin_resize_common:NnnN . .
 28397, 28399, 28438
\l__coffin_right_corner_dim . .
 .. 28208, 28249, 28324, 28334, 28335
__coffin_rotate:NnnNN 28212
__coffin_rotate_bounding:nm . .
 28232, 28278
__coffin_rotate_corner:Nnnn . .
 28227, 28278
__coffin_rotate_pole:Nnnnnn . .
 28229, 28290
__coffin_rotate_vector:nnNN . .
 .. 28280, 28286, 28292, 28293, 28302
__coffin_rule:nn 28743, 28754, 28799
__coffin_scale:NnnNN . . .
 28422, 28425, 28427
__coffin_scale_corner:Nnnn . .
 28406, 28449
__coffin_scale_pole:Nnnnnn . .
 28408, 28449
__coffin_scale_vector:nnNN . .
 28442, 28451, 28457
\l__coffin_scale_x_fp 28369, 28389,
 28409, 28429, 28431, 28437, 28445
\l__coffin_scale_y_fp 28369,
 28391, 28430, 28431, 28435, 28447
\l__coffin_scaled_total_height_-
 dim 28371, 28434, 28439
\l__coffin_scaled_width_dim . .
 28371, 28436, 28439
__coffin_set_bounding:N 28230, 28260
__coffin_set_horizontal_-
 pole:NnnN 27972
__coffin_set_pole:Nnn . . .
 27821, 27895, 27972,
 28603, 28643, 28647, 28655, 28659
__coffin_set_vertical:NnnNN . 27797
__coffin_set_vertical:NnNNNNw 27869
__coffin_set_vertical_aux: . .
 27797, 27889
__coffin_set_vertical_pole:NnnN . .
 27972
__coffin_shift_corner:Nnnn . .
 28252, 28352
__coffin_shift_pole:Nnnnnn . .
 28254, 28352
__coffin_show_structure>NN . . 28894
\l__coffin_sin_fp
 1101, 1103, 28202, 28220, 28309, 28316
\l__coffin_slope_A_fp 27707
\l__coffin_slope_B_fp 27707

__coffin_to_value:N . . .
 27718, 27723, 27762, 27763, 27764,
 27766, 27916, 27917, 27918, 27919,
 27928, 27929, 27930, 27931, 27951,
 27960, 27962, 27967, 27969, 27982,
 28000, 28010, 28033, 28064, 28223,
 28225, 28255, 28257, 28402, 28404,
 28416, 28418, 28508, 28552, 28555,
 28593, 28612, 28619, 28802, 28913
\l__coffin_top_corner_dim . . .
 .. 28208, 28246, 28323, 28338, 28339
__coffin_update:N . . .
 .. 27780, 27800, 27849, 27873, 28014
__coffin_update_B:mnnnnnnnnN . 28628
__coffin_update_corners:N . .
 28017, 28026
__coffin_update_corners:NN .. 28026
__coffin_update_corners:NNN .. 28026
__coffin_update_poles:N . . .
 28018, 28057, 28511, 28556
__coffin_update_poles:NN . . . 28057
__coffin_update_poles:NNN . . . 28057
__coffin_update_T:mnnnnnnnnN . 28628
__coffin_update_vertical_-
 poles:NNN 28527, 28560, 28628
\l__coffin_x_dim 27714, 28112,
 28121, 28147, 28178, 28196, 28280,
 28282, 28286, 28288, 28292, 28297,
 28451, 28453, 28457, 28460, 28575,
 28579, 28598, 28606, 28826, 28873
\l__coffin_x_prime_dim . . .
 27714, 28294,
 28298, 28575, 28579, 28873, 28876
__coffin_x_shift_corner:Nnnn . .
 28412, 28464
__coffin_x_shift_pole:Nnnnnn . .
 28414, 28464
\l__coffin_y_dim 27714,
 28113, 28125, 28143, 28192, 28280,
 28282, 28286, 28288, 28292, 28297,
 28451, 28453, 28457, 28460, 28576,
 28581, 28599, 28606, 28827, 28874
\l__coffin_y_prime_dim . . .
 27714, 28294,
 28299, 28576, 28581, 28874, 28878
\color 28738, 28739
color commands:
 \color_ensure_current: . 255, 1090,
 27777, 27790, 27845, 27858, 28945
 \color_group_begin: . . .
 255, 255, 27152,
 27156, 27161, 27168, 27173, 27181,
 27187, 27201, 27207, 27214, 27219,
 27232, 27234, 27238, 27243, 27248,

```

27253, 27260, 27265, 27272, 27277,  
 27285, 27291, 27306, 27312, 28943  
 \color\_group\_end: ..... 255, 255,  
 27152, 27156, 27161, 27168, 27173,  
 27193, 27214, 27219, 27232, 27234,  
 27238, 27243, 27248, 27253, 27260,  
 27265, 27272, 27277, 27298, 28943  
 \color\_select:n ..... 28735, 28736  
 color internal commands:  
   \\_\_color\_backend\_pickup:N ..... 28947  
   \\_\_color\_current\_tl .....  
     ... 1120, 28943, 28947, 28948, 28955  
   \\_\_color\_select:N ..... 28948, 28951  
   \\_\_color\_select:nn ..... 28951  
 \columnwidth ..... 27838, 27839  
 \copy ..... 223  
 \copyfont ..... 913  
 cos ..... 215  
 cosd ..... 215  
 cot ..... 215  
 cotd ..... 215  
 \count ..... 224, 11073  
 \countdef ..... 225  
 \cr ..... 226  
 \crampeddisplaystyle ..... 804  
 \crampedscriptscriptstyle ..... 805  
 \crampedscriptstyle ..... 807  
 \crampedtextstyle ..... 808  
 \crcr ..... 227  
 \creationdate ..... 777  
 \cs ..... 18685  
 cs commands:  
   \cs:w ..... 17, 1048, 1048, 1432,  
     1454, 1456, 1509, 1816, 1844, 2037,  
     2101, 2250, 2299, 2308, 2310, 2314,  
     2315, 2316, 2378, 2384, 2390, 2396,  
     2423, 2425, 2430, 2437, 2438, 2503,  
     2507, 2546, 3164, 5436, 5442, 8298,  
     8384, 9021, 9073, 9320, 9322, 10740,  
     14214, 14511, 14558, 14624, 14650,  
     15742, 16375, 16394, 16461, 17270,  
     17459, 17491, 17909, 17935, 17948,  
     17984, 18028, 18530, 20235, 21327,  
     26243, 26246, 27013, 31669, 32161  
   \cs\_argument\_spec:N .....  
     ... 18, 2187, 32997, 32998  
   \cs\_end: ..... 17, 366, 1432, 1454,  
     1456, 1460, 1509, 1810, 1816, 1838,  
     1844, 1964, 2037, 2101, 2250, 2299,  
     2308, 2310, 2314, 2315, 2316, 2378,  
     2384, 2390, 2396, 2423, 2425, 2430,  
     2437, 2438, 2503, 2507, 2546, 3164,  
     5442, 5445, 8298, 8384, 9021, 9026,  
     9054, 9063, 9073, 9317, 9323, 9325,  
     9327, 9329, 9331, 9333, 9335, 9337,  
     9339, 9341, 9343, 10740, 14214,  
     14511, 14558, 14624, 14650, 15742,  
     16378, 16394, 16469, 17273, 17463,  
     17495, 17915, 17941, 17954, 17987,  
     18031, 18536, 20235, 21330, 26110,  
     26260, 27013, 31669, 32159, 32161  
 \cs\_generate\_from\_arg\_count>NNnm  
   ..... 14, 2017, 2059  
 \cs\_generate\_variant:Nn .....  
   ..... 10, 25, 27, 28, 107,  
   266, 360, 2863, 3206, 3208, 3210,  
   3212, 3326, 3327, 3397, 3404, 3428,  
   3591, 3602, 3603, 3608, 3609, 3614,  
   3615, 3618, 3619, 3636, 3637, 3663,  
   3664, 3665, 3666, 3667, 3668, 3709,  
   3710, 3711, 3712, 3713, 3714, 3715,  
   3716, 3751, 3752, 3753, 3754, 3755,  
   3756, 3757, 3758, 3806, 3807, 3808,  
   3809, 3873, 3874, 3875, 3876, 3937,  
   3938, 3943, 3944, 4114, 4132, 4146,  
   4155, 4177, 4182, 4184, 4193, 4205,  
   4206, 4243, 4246, 4251, 4252, 4228,  
   4339, 4537, 4548, 4549, 4572, 4579,  
   4581, 4658, 4679, 4681, 4732, 4747,  
   4748, 4751, 4752, 4763, 4785, 4786,  
   4787, 4788, 4821, 4822, 4827, 4828,  
   4896, 4954, 4972, 4998, 5049, 5110,  
   5188, 5207, 5245, 5260, 5277, 5278,  
   5279, 5292, 5334, 5337, 7538, 7541,  
   7544, 7547, 7550, 7579, 7580, 7581,  
   7582, 7583, 7584, 7590, 7626, 7627,  
   7632, 7633, 7655, 7656, 7657, 7658,  
   7663, 7664, 7665, 7666, 7683, 7684,  
   7709, 7710, 7727, 7728, 7784, 7785,  
   7835, 7848, 7849, 7867, 7893, 7894,  
   7946, 7952, 7972, 8001, 8009, 8026,  
   8027, 8104, 8127, 8141, 8175, 8177,  
   8300, 8321, 8336, 8337, 8342, 8343,  
   8345, 8347, 8360, 8361, 8362, 8363,  
   8372, 8373, 8374, 8375, 8380, 8381,  
   8383, 8990, 8994, 9081, 9087, 9096,  
   9097, 9098, 9099, 9102, 9103, 9114,  
   9115, 9137, 9139, 9277, 9278, 9283,  
   9284, 9621, 9648, 9870, 9910, 9911,  
   9912, 9913, 9927, 9928, 9937, 9938,  
   9948, 9949, 9950, 9951, 9961, 9962,  
   9963, 9964, 9975, 10000, 10001,  
   10059, 10060, 10098, 10099, 10104,  
   10105, 10188, 10222, 10246, 10259,  
   10299, 10308, 10332, 10339, 10380,  
   10382, 10384, 10529, 10530, 10531,  
   10532, 10732, 10788, 11409, 11415,  
   11418, 11421, 11424, 11427, 11448,

11456, 11464, 11523, 11524, 11525,  
11526, 11533, 11534, 11553, 11554,  
11555, 11556, 11569, 11579, 11615,  
11617, 11619, 11621, 11638, 11639,  
11701, 11716, 11730, 11736, 11738,  
12266, 12679, 12680, 12681, 12682,  
12704, 12705, 12706, 12707, 12736,  
12741, 12775, 12796, 12954, 12973,  
12981, 12993, 13008, 13011, 13029,  
13137, 13636, 13645, 13646, 13647,  
13928, 14012, 14216, 14222, 14226,  
14227, 14232, 14233, 14242, 14243,  
14246, 14249, 14257, 14258, 14266,  
14267, 14510, 14540, 14560, 14566,  
14569, 14570, 14575, 14576, 14585,  
14586, 14588, 14590, 14595, 14596,  
14601, 14602, 14623, 14631, 14632,  
14634, 14652, 14658, 14663, 14664,  
14669, 14670, 14679, 14680, 14682,  
14684, 14689, 14690, 14695, 14696,  
14700, 14702, 14950, 15046, 15062,  
15117, 15124, 15191, 15258, 15264,  
15441, 15455, 15461, 15471, 15497,  
15503, 15513, 15553, 15591, 15985,  
15987, 16017, 16059, 16068, 16077,  
16086, 16094, 16113, 16115, 16129,  
16150, 16193, 16716, 16719, 18419,  
18426, 18427, 18428, 18431, 18432,  
18435, 18436, 18441, 18442, 18449,  
18450, 18451, 18452, 18454, 18456,  
18680, 18742, 21825, 21879, 21957,  
22002, 22017, 22071, 22410, 22430,  
22459, 22513, 22521, 22529, 22615,  
22654, 22667, 22726, 22834, 23048,  
23050, 23072, 23075, 23081, 23087,  
23782, 24496, 27015, 27020, 27021,  
27026, 27027, 27032, 27033, 27038,  
27039, 27047, 27048, 27049, 27055,  
27058, 27064, 27067, 27073, 27076,  
27079, 27080, 27108, 27109, 27117,  
27120, 27123, 27132, 27150, 27163,  
27164, 27175, 27176, 27189, 27190,  
27209, 27210, 27229, 27230, 27255,  
27256, 27267, 27268, 27279, 27280,  
27293, 27294, 27314, 27315, 27318,  
27319, 27322, 27328, 27343, 27346,  
27464, 27470, 27510, 27513, 27530,  
27533, 27550, 27553, 27567, 27570,  
27588, 27591, 27617, 27620, 27626,  
27632, 27748, 27757, 27770, 27783,  
27796, 27802, 27808, 27853, 27866,  
27875, 27882, 27922, 27934, 27974,  
27977, 27992, 27995, 28013, 28214,  
28217, 28379, 28386, 28423, 28426,  
28484, 28490, 28535, 28541, 28672,  
28793, 28869, 28896, 28899, 29592,  
29876, 29918, 31613, 32049, 32052,  
32055, 32058, 32106, 32109, 32191,  
32194, 32243, 32273, 32434, 32437,  
32482, 32486, 32912, 32913, 32917,  
32921, 32946, 32955, 32966, 32976  
\cs\_gset:Nn . . . . . 14, 2032, 2096  
.cs\_gset:Np . . . . . 187, 15301  
\cs\_gset:Npn 10, 12, 1493, 1912, 1926,  
1928, 3478, 3479, 3516, 3558, 7976,  
10586, 11791, 11793, 11831, 13618,  
15310, 15312, 17150, 30252, 32705,  
32924, 32926, 32928, 32930, 32932,  
32934, 32939, 32941, 32978, 32981,  
32984, 32987, 32990, 32993, 32996,  
32998, 33000, 33002, 33004, 33006,  
33008, 33010, 33012, 33014, 33016  
\cs\_gset:Npx . . . . . 12,  
1493, 1913, 1926, 1929, 7981, 26118  
\cs\_gset\_eq:NN 15, 1944, 1961, 1969,  
3589, 3617, 5593, 5597, 7536, 7755,  
7986, 7991, 9093, 9095, 9742, 10527,  
11413, 11704, 11712, 12793, 12990  
\cs\_gset\_nopar:Nn . . . . . 14, 2032, 2096  
\cs\_gset\_nopar:Npn . . . . .  
12, 1493, 1910, 1918, 1922, 3286  
\cs\_gset\_nopar:Npx . 12, 379, 1196,  
1493, 1911, 1918, 1923, 3572, 3585,  
3595, 3600, 32378, 32404, 32409, 32436  
\cs\_gset\_protected:Nn 14, 2032, 2096  
.cs\_gset\_protected:Np . . . . . 187, 15301  
\cs\_gset\_protected:Npn . . . . .  
12, 1493, 1916,  
1938, 1940, 4136, 4958, 8036, 8608,  
10208, 11707, 12891, 14468, 15314,  
15316, 18747, 22815, 22999, 23007,  
23020, 23412, 23680, 23685, 32621,  
32643, 32652, 32662, 32700, 32900,  
32907, 32910, 32915, 32919, 32936,  
32944, 32948, 32957, 32968, 33018  
\cs\_gset\_protected:Npx . . . . .  
12, 1493, 1917,  
1938, 1941, 8619, 14475, 18754, 32632  
\cs\_gset\_protected\_nopar:Nn . . . . .  
14, 2032, 2096  
\cs\_gset\_protected\_nopar:Npn . . . . .  
12, 1493, 1914, 1932, 1934  
\cs\_gset\_protected\_nopar:Npx . . . . .  
12, 1493, 1915, 1932, 1935  
\cs\_if\_eq:NNTF . . . . . 22, 1205,  
2128, 2135, 2136, 2139, 2140, 2143,  
2144, 9784, 11841, 15214, 16915,

16925, 16951, 16953, 16955, 17155,  
 25120, 29786, 31560, 32652, 32662  
`\cs_if_eq_p:N` ..... . 22, 2128, 25137, 29829, 29830, 32621  
`\cs_if_exist` ..... 237  
`\cs_if_exist:N` 22, 3638, 3639, 7634,  
 7636, 8348, 8350, 9149, 9151, 9929,  
 9931, 11640, 11642, 14234, 14236,  
 14577, 14579, 14671, 14673, 18481,  
 18482, 22784, 22786, 27040, 27042  
`\cs_if_exist:NTF` ..... 16, 22, 305, 357, 437, 584, 617, 1796,  
 1853, 1855, 1857, 1859, 1861, 1863,  
 1865, 1867, 2148, 2253, 2323, 2359,  
 2456, 2480, 2548, 2579, 2830, 3096,  
 4293, 5564, 5573, 5577, 5591, 7739,  
 8323, 8324, 8325, 8326, 9002, 9003,  
 9049, 9395, 9396, 9397, 9399, 9403,  
 9432, 9667, 9782, 10583, 10736,  
 11037, 11056, 11762, 12578, 12725,  
 12729, 12755, 12812, 12939, 12943,  
 13385, 13560, 13616, 13648, 13657,  
 13800, 13876, 14030, 14083, 14195,  
 14964, 15069, 15160, 15165, 15620,  
 15662, 15670, 15682, 15694, 15700,  
 15711, 15727, 15814, 15875, 15883,  
 16027, 17148, 17322, 18409, 22676,  
 22776, 22813, 22848, 22997, 23015,  
 23018, 24854, 26232, 27721, 27723,  
 27836, 27838, 28719, 28725, 28726,  
 28729, 28735, 28738, 28745, 29819,  
 30102, 30110, 30275, 31395, 31405,  
 31412, 31749, 32311, 32330, 32896  
`\cs_if_exist_p:N` ..... . 22, 305, 1796, 9463, 10871,  
 10872, 22752, 29802, 29838, 31572  
`\cs_if_exist_use:N` ..... 16, 328, 1852, 12146, 12164, 13106,  
 15696, 15715, 25302, 29911, 29987  
`\cs_if_exist_use:NTF` ..... 16, 1852, 1854, 1856,  
 1862, 1864, 3126, 3195, 3431, 9757,  
 16597, 17280, 17282, 24088, 24095,  
 24459, 24464, 24501, 24922, 25008,  
 26167, 30122, 30128, 30229, 30231  
`\cs_if_free:NTF` ..... 22, 105, 617, 1824, 1893, 3038, 3065, 12136  
`\cs_if_free_p:N` ... 21, 22, 105, 1824  
`\cs_log:N` ..... 16, 337, 2173  
`\cs_meaning:N` ..... 15, 314,  
 1441, 1457, 1465, 2185, 9664, 22825  
`\cs_new:Nn` ..... 12, 106, 2032, 2096  
`\cs_new:Npn` ..... 10, 11, 14, 105,  
 105, 364, 955, 1205, 1583, 1600,  
 1902, 1926, 1930, 2003, 2005, 2007,  
 2015, 2067, 2133, 2134, 2135, 2136,  
 2137, 2138, 2139, 2140, 2141, 2142,  
 2143, 2144, 2189, 2193, 2202, 2211,  
 2220, 2223, 2232, 2233, 2243, 2244,  
 2245, 2246, 2247, 2248, 2249, 2251,  
 2255, 2259, 2262, 2275, 2281, 2287,  
 2298, 2300, 2307, 2309, 2311, 2318,  
 2319, 2321, 2325, 2329, 2335, 2337,  
 2342, 2347, 2353, 2361, 2368, 2369,  
 2375, 2381, 2387, 2393, 2399, 2406,  
 2413, 2420, 2427, 2434, 2443, 2444,  
 2446, 2451, 2458, 2462, 2465, 2475,  
 2476, 2478, 2482, 2485, 2486, 2488,  
 2490, 2496, 2502, 2504, 2510, 2512,  
 2519, 2526, 2527, 2528, 2529, 2530,  
 2532, 2541, 2543, 2546, 2547, 2550,  
 2554, 2557, 2559, 2564, 2574, 2577,  
 2581, 2599, 2600, 2602, 2608, 2613,  
 2615, 2621, 2641, 2643, 2645, 2658,  
 2665, 2680, 2686, 2692, 2697, 2698,  
 2721, 2751, 2758, 2764, 2792, 2798,  
 2803, 2809, 2858, 2859, 2860, 2861,  
 2930, 2951, 2973, 2976, 2984, 2997,  
 3012, 3023, 3055, 3160, 3162, 3296,  
 3302, 3310, 3317, 3324, 3328, 3334,  
 3367, 3377, 3380, 3484, 3493, 3526,  
 3531, 3533, 3542, 3548, 3553, 3580,  
 3801, 3862, 3930, 3965, 4054, 4057,  
 4058, 4059, 4060, 4071, 4086, 4091,  
 4096, 4101, 4106, 4108, 4117, 4119,  
 4125, 4127, 4147, 4153, 4156, 4178,  
 4180, 4183, 4185, 4194, 4199, 4204,  
 4207, 4218, 4219, 4220, 4221, 4228,  
 4235, 4237, 4244, 4255, 4267, 4276,  
 4282, 4288, 4290, 4295, 4300, 4307,  
 4312, 4318, 4329, 4330, 4331, 4332,  
 4340, 4380, 4389, 4408, 4410, 4423,  
 4430, 4446, 4457, 4465, 4471, 4474,  
 4479, 4491, 4497, 4498, 4500, 4508,  
 4514, 4521, 4523, 4525, 4538, 4540,  
 4542, 4550, 4558, 4564, 4571, 4573,  
 4578, 4580, 4582, 4583, 4591, 4603,  
 4612, 4621, 4626, 4632, 4655, 4656,  
 4657, 4659, 4704, 4720, 4721, 4874,  
 4879, 4884, 4889, 4894, 4899, 4905,  
 4910, 4915, 4920, 4925, 4927, 4933,  
 4935, 4943, 4945, 4947, 4973, 4999,  
 5001, 5003, 5014, 5023, 5026, 5037,  
 5046, 5048, 5050, 5058, 5060, 5067,  
 5088, 5098, 5103, 5108, 5109, 5111,  
 5119, 5121, 5129, 5135, 5141, 5160,  
 5162, 5171, 5177, 5184, 5186, 5189,  
 5199, 5206, 5208, 5216, 5221, 5226,

5237, 5244, 5246, 5252, 5254, 5259,  
5261, 5267, 5268, 5273, 5274, 5275,  
5276, 5280, 5285, 5290, 5293, 5295,  
5303, 5308, 5374, 5382, 5389, 5430,  
5432, 5438, 5444, 5446, 5451, 5456,  
5472, 5488, 5502, 5599, 5605, 5649,  
5655, 5687, 5697, 5708, 5734, 5741,  
5801, 5808, 5830, 5840, 5909, 5919,  
5956, 6018, 6059, 6061, 6078, 6084,  
6107, 6137, 6158, 6167, 6244, 6264,  
6284, 6307, 6314, 6341, 6444, 6458,  
6485, 6494, 6496, 6517, 6522, 6528,  
6533, 6601, 6624, 6636, 6645, 6651,  
6656, 7524, 7618, 7624, 7654, 7667,  
7722, 7803, 7833, 7861, 7866, 7888,  
7923, 7925, 7933, 7939, 7947, 7953,  
7955, 7957, 7966, 8002, 8010, 8028,  
8043, 8053, 8081, 8099, 8103, 8105,  
8128, 8129, 8130, 8137, 8139, 8202,  
8207, 8209, 8210, 8216, 8224, 8230,  
8248, 8256, 8264, 8277, 8279, 8286,  
8288, 8384, 8391, 8405, 8410, 8416,  
8427, 8432, 8439, 8441, 8443, 8445,  
8447, 8449, 8451, 8461, 8466, 8471,  
8476, 8481, 8483, 8489, 8507, 8515,  
8523, 8529, 8535, 8543, 8551, 8557,  
8563, 8570, 8586, 8596, 8598, 8634,  
8648, 8654, 8686, 8718, 8720, 8722,  
8728, 8734, 8746, 8754, 8766, 8774,  
8807, 8840, 8842, 8844, 8846, 8848,  
8853, 8858, 8863, 8868, 8869, 8870,  
8871, 8872, 8873, 8874, 8875, 8876,  
8877, 8878, 8879, 8880, 8881, 8882,  
8883, 8884, 8893, 8894, 8903, 8909,  
8911, 8920, 8927, 8933, 8935, 8937,  
8953, 8964, 8987, 9020, 9060, 9061,  
9070, 9071, 9118, 9134, 9161, 9162,  
9171, 9172, 9182, 9187, 9192, 9194,  
9202, 9203, 9204, 9205, 9206, 9207,  
9208, 9209, 9210, 9211, 9221, 9237,  
9247, 9263, 9273, 9275, 9279, 9281,  
9285, 9293, 9298, 9306, 9312, 9319,  
9321, 9323, 9324, 9326, 9328, 9330,  
9332, 9334, 9336, 9338, 9340, 9342,  
9344, 9349, 9350, 9351, 9352, 9353,  
9354, 9355, 9356, 9357, 9358, 9368,  
9370, 9694, 9696, 9814, 9815, 9821,  
9827, 9833, 9863, 9864, 9903, 9999,  
10095, 10097, 10106, 10113, 10116,  
10129, 10135, 10173, 10182, 10189,  
10195, 10202, 10247, 10249, 10251,  
10269, 10276, 10300, 10301, 10304,  
10306, 10309, 10317, 10333, 10340,  
10348, 10350, 10364, 10366, 10367,  
10368, 10370, 10375, 10415, 10485,  
10491, 10497, 10503, 10534, 10540,  
10577, 10641, 10656, 10661, 10705,  
10707, 10709, 10712, 10715, 10721,  
10727, 10733, 10734, 10746, 10756,  
10758, 10760, 10769, 10771, 10780,  
10786, 10789, 10799, 10805, 10812,  
10814, 10846, 10848, 10850, 10856,  
10858, 10864, 10985, 11015, 11161,  
11173, 11174, 11182, 11191, 11200,  
11214, 11215, 11216, 11217, 11220,  
11276, 11284, 11286, 11288, 11298,  
11308, 11400, 11510, 11557, 11563,  
11570, 11578, 11658, 11664, 11686,  
11695, 11717, 11724, 11731, 11733,  
11757, 11830, 11865, 11927, 11932,  
11937, 11939, 11941, 11943, 11949,  
11958, 11964, 11970, 12115, 12117,  
12134, 12267, 12632, 12641, 12647,  
12659, 12664, 12669, 12674, 12683,  
12696, 12698, 12700, 12702, 12708,  
12875, 12877, 12951, 13034, 13042,  
13080, 13097, 13152, 13203, 13212,  
13231, 13232, 13240, 13246, 13254,  
13264, 13269, 13275, 13281, 13354,  
13356, 13358, 13406, 13417, 13428,  
13463, 13468, 13474, 13483, 13485,  
13494, 13496, 13497, 13499, 13547,  
13552, 13570, 13572, 13583, 13584,  
13585, 13587, 13605, 13693, 13695,  
13697, 13702, 13707, 13709, 13711,  
13718, 13732, 13742, 13752, 13759,  
13763, 13770, 13843, 13956, 13961,  
13966, 13971, 13977, 13991, 14029,  
14097, 14210, 14268, 14273, 14275,  
14283, 14291, 14299, 14301, 14313,  
14319, 14332, 14334, 14336, 14338,  
14340, 14348, 14353, 14358, 14363,  
14368, 14370, 14376, 14378, 14386,  
14394, 14400, 14406, 14414, 14422,  
14428, 14434, 14441, 14455, 14489,  
14491, 14497, 14511, 14512, 14519,  
14527, 14529, 14531, 14617, 14620,  
14624, 14626, 14629, 14697, 14725,  
14727, 14734, 14736, 14738, 14749,  
14756, 14760, 14768, 14777, 14781,  
14789, 14791, 14799, 14803, 14810,  
14817, 14824, 14833, 14843, 14849,  
14855, 14856, 14857, 14858, 14859,  
14871, 14883, 14891, 14896, 14902,  
15029, 15736, 15738, 15803, 15812,  
15818, 15820, 15825, 15829, 15833,  
15837, 15843, 15855, 15859, 15863,  
15986, 15988, 15990, 16001, 16060,

16062, 16069, 16075, 16093, 16095,  
16103, 16200, 16201, 16202, 16203,  
16204, 16205, 16206, 16207, 16208,  
16209, 16219, 16243, 16245, 16247,  
16256, 16258, 16265, 16277, 16278,  
16280, 16290, 16300, 16310, 16320,  
16328, 16330, 16337, 16339, 16340,  
16345, 16352, 16366, 16368, 16384,  
16385, 16393, 16395, 16404, 16406,  
16418, 16423, 16427, 16432, 16434,  
16436, 16438, 16440, 16447, 16449,  
16457, 16459, 16471, 16473, 16475,  
16477, 16501, 16503, 16505, 16506,  
16507, 16509, 16511, 16513, 16515,  
16533, 16548, 16549, 16555, 16571,  
16577, 16703, 16704, 16705, 16706,  
16707, 16708, 16709, 16714, 16717,  
16763, 16765, 16767, 16769, 16775,  
16779, 16781, 16790, 16791, 16800,  
16813, 16826, 16833, 16847, 16863,  
16875, 16886, 16896, 16902, 16913,  
16923, 16949, 16960, 16977, 16988,  
16993, 17013, 17015, 17026, 17031,  
17044, 17067, 17068, 17072, 17089,  
17090, 17114, 17122, 17140, 17169,  
17195, 17199, 17202, 17204, 17210,  
17222, 17234, 17241, 17247, 17255,  
17278, 17293, 17312, 17320, 17335,  
17350, 17361, 17371, 17381, 17386,  
17395, 17412, 17425, 17430, 17436,  
17438, 17445, 17475, 17503, 17519,  
17530, 17535, 17553, 17571, 17582,  
17597, 17602, 17613, 17623, 17633,  
17649, 17697, 17702, 17709, 17717,  
17723, 17728, 17732, 17749, 17757,  
17789, 17806, 17820, 17839, 17847,  
17856, 17865, 17876, 17878, 17892,  
17902, 17903, 17920, 17927, 17932,  
17945, 17958, 17963, 17993, 18007,  
18037, 18038, 18042, 18059, 18081,  
18083, 18094, 18126, 18130, 18145,  
18162, 18186, 18188, 18190, 18192,  
18202, 18207, 18218, 18230, 18241,  
18254, 18274, 18292, 18294, 18306,  
18312, 18320, 18334, 18341, 18352,  
18359, 18373, 18477, 18479, 18496,  
18518, 18523, 18540, 18567, 18568,  
18569, 18570, 18586, 18597, 18605,  
18617, 18623, 18629, 18637, 18645,  
18651, 18657, 18665, 18673, 18691,  
18704, 18726, 18774, 18780, 18791,  
18815, 18817, 18819, 18821, 18829,  
18833, 18840, 18847, 18848, 18849,  
18850, 18851, 18852, 18855, 18857,  
18886, 18894, 18905, 18907, 18909,  
18911, 18918, 18942, 18944, 18954,  
18969, 18978, 18992, 19000, 19008,  
19015, 19022, 19030, 19040, 19054,  
19065, 19066, 19072, 19089, 19096,  
19098, 19105, 19110, 19127, 19128,  
19129, 19148, 19154, 19164, 19176,  
19183, 19197, 19205, 19243, 19252,  
19273, 19275, 19277, 19286, 19297,  
19309, 19324, 19337, 19350, 19358,  
19376, 19394, 19401, 19409, 19419,  
19420, 19429, 19430, 19439, 19449,  
19463, 19473, 19484, 19492, 19494,  
19505, 19511, 19546, 19567, 19569,  
19571, 19573, 19580, 19589, 19594,  
19601, 19608, 19628, 19633, 19650,  
19661, 19666, 19676, 19678, 19688,  
19695, 19697, 19703, 19705, 19707,  
19711, 19730, 19731, 19736, 19744,  
19745, 19768, 19781, 19788, 19796,  
19797, 19798, 19799, 19800, 19801,  
19809, 19815, 19817, 19819, 19841,  
19846, 19856, 19866, 19877, 19890,  
19901, 19906, 19913, 19922, 19924,  
19933, 19942, 19956, 19958, 19960,  
19973, 19983, 19988, 19997, 20005,  
20012, 20018, 20027, 20029, 20041,  
20046, 20054, 20059, 20069, 20075,  
20081, 20088, 20095, 20097, 20102,  
20104, 20109, 20111, 20125, 20135,  
20147, 20152, 20159, 20169, 20171,  
20173, 20184, 20198, 20212, 20232,  
20245, 20247, 20252, 20265, 20270,  
20278, 20283, 20293, 20305, 20335,  
20336, 20337, 20339, 20341, 20343,  
20357, 20363, 20372, 20391, 20397,  
20407, 20426, 20434, 20467, 20473,  
20482, 20484, 20498, 20557, 20565,  
20583, 20600, 20601, 20606, 20631,  
20654, 20683, 20699, 20709, 20720,  
20741, 20756, 20761, 20766, 20768,  
20782, 20788, 20803, 20811, 20821,  
20831, 20844, 20862, 20868, 20882,  
20897, 20935, 20937, 20939, 20941,  
20943, 20958, 20973, 20988, 21003,  
21018, 21033, 21041, 21055, 21057,  
21063, 21075, 21083, 21090, 21316,  
21323, 21360, 21368, 21369, 21380,  
21387, 21389, 21395, 21406, 21416,  
21423, 21430, 21445, 21484, 21497,  
21528, 21534, 21541, 21561, 21563,  
21580, 21595, 21608, 21615, 21620,  
21622, 21631, 21644, 21647, 21668,  
21681, 21696, 21714, 21729, 21739,

21748, 21761, 21777, 21794, 21807,  
 21813, 21815, 21820, 21821, 21822,  
 21823, 21826, 21831, 21837, 21842,  
 21844, 21867, 21875, 21877, 21880,  
 21885, 21891, 21896, 21898, 21921,  
 21946, 21955, 21956, 21958, 21963,  
 21965, 21970, 21972, 21982, 21990,  
 21998, 22000, 22003, 22008, 22013,  
 22015, 22016, 22018, 22023, 22028,  
 22030, 22035, 22042, 22056, 22061,  
 22063, 22073, 22075, 22077, 22079,  
 22081, 22092, 22102, 22104, 22110,  
 22117, 22123, 22133, 22138, 22140,  
 22148, 22149, 22163, 22170, 22176,  
 22177, 22190, 22205, 22211, 22233,  
 22248, 22258, 22279, 22288, 22311,  
 22329, 22340, 22345, 22356, 22373,  
 22378, 22425, 22431, 22445, 22514,  
 22522, 22530, 22536, 22543, 22548,  
 22554, 22566, 22658, 22763, 22765,  
 22772, 23062, 23217, 23230, 23235,  
 23241, 23242, 23249, 23256, 23263,  
 23270, 23277, 23278, 23280, 23287,  
 23293, 23370, 23375, 23376, 23384,  
 23390, 23567, 23572, 23579, 23616,  
 23621, 23636, 23659, 23664, 23712,  
 23718, 23736, 23741, 23756, 23758,  
 23760, 23767, 23798, 23827, 23832,  
 23857, 23859, 24086, 24092, 24103,  
 24108, 24121, 24126, 24138, 24150,  
 24160, 24169, 24189, 24300, 24318,  
 24326, 24338, 24350, 24842, 25118,  
 25133, 25173, 25214, 25374, 25502,  
 25894, 26020, 26022, 26028, 26029,  
 26036, 26046, 26198, 26563, 26568,  
 26967, 27008, 28966, 28967, 28971,  
 28972, 29406, 29411, 29426, 29436,  
 29447, 29461, 29478, 29491, 29493,  
 29494, 29575, 29583, 29590, 29593,  
 29595, 29601, 29612, 29624, 29648,  
 29668, 29684, 29691, 29705, 29716,  
 29725, 29730, 29736, 29745, 29755,  
 29760, 29771, 29776, 29782, 29808,  
 29813, 29815, 29826, 29834, 29835,  
 29850, 29851, 29864, 29866, 29882,  
 29884, 29886, 29888, 29890, 29892,  
 29894, 29896, 29898, 29908, 29916,  
 29919, 29921, 29927, 29938, 29943,  
 29957, 29969, 29983, 29990, 29996,  
 30001, 30007, 30021, 30032, 30041,  
 30048, 30055, 30062, 30071, 30076,  
 30087, 30092, 30096, 30098, 30100,  
 30120, 30126, 30136, 30142, 30153,  
 30165, 30172, 30178, 30188, 30223,  
 30225, 30227, 30236, 30267, 30269,  
 30271, 30273, 30284, 30292, 30298,  
 30320, 30334, 30343, 30345, 30355,  
 30382, 30389, 30400, 30409, 30422,  
 30430, 30540, 30548, 30561, 30573,  
 30588, 30595, 30608, 30633, 30643,  
 30650, 30673, 30686, 30696, 30703,  
 30712, 30724, 30731, 30751, 30765,  
 30776, 30794, 30807, 31432, 31441,  
 31448, 31450, 31452, 31458, 31469,  
 31470, 31475, 31480, 31487, 31498,  
 31503, 31505, 31507, 31512, 31517,  
 31528, 31539, 31544, 31551, 31556,  
 31567, 31569, 31592, 31593, 31603,  
 31669, 31747, 32157, 32197, 32199,  
 32201, 32203, 32205, 32207, 32209,  
 32211, 32213, 32218, 32224, 32230,  
 32233, 32234, 32244, 32252, 32254,  
 32260, 32266, 32453, 32461, 32481,  
 32483, 32484, 32487, 32489, 32491,  
 32493, 32501, 32569, 32571, 32601  
`\cs_new:Npx` . . . . . 11, 34, 34, 358,  
`\cs_new:Npn` . . . . . 359, 1902, 1926, 1931, 2889, 6658,  
`\cs_new:Nn` . . . . . 10260, 10270, 13079, 13091, 13450,  
`\cs_new:NnN` . . . . . 13458, 13596, 16359, 17181, 17769,  
`\cs_new:Nx` . . . . . 18913, 20922, 20928, 21540, 23592,  
`\cs_new_eq:NN` . . . . . 24109, 24111, 24113, 24115, 24117,  
`\cs_new_eq:NNN` . . . . . 24119, 29629, 29796, 30202, 30311,  
`\cs_new_eq:NNN` . . . . . 30821, 32170, 32172, 32174, 32181  
`\cs_new_eq:NNN` . . . . . 15, 106,  
`\cs_new_eq:NNN` . . . . . 329, 331, 579, 1716, 1944, 2222,  
`\cs_new_eq:NNN` . . . . . 2231, 2268, 2545, 2839, 2840, 2841,  
`\cs_new_eq:NNN` . . . . . 2842, 2843, 2844, 2845, 2846, 2847,  
`\cs_new_eq:NNN` . . . . . 2848, 2849, 2850, 2851, 2852, 2853,  
`\cs_new_eq:NNN` . . . . . 2856, 2857, 3098, 3294, 3568, 3571,  
`\cs_new_eq:NNN` . . . . . 3584, 3585, 3931, 3932, 4731, 4745,  
`\cs_new_eq:NNN` . . . . . 4746, 4749, 4750, 4816, 4837, 5332,  
`\cs_new_eq:NNN` . . . . . 5333, 5335, 5336, 5666, 5682, 5684,  
`\cs_new_eq:NNN` . . . . . 6672, 7531, 7551, 7552, 7553, 7554,  
`\cs_new_eq:NNN` . . . . . 7555, 7556, 7557, 7558, 7782, 8142,  
`\cs_new_eq:NNN` . . . . . 8143, 8144, 8145, 8146, 8147, 8148,  
`\cs_new_eq:NNN` . . . . . 8149, 8150, 8151, 8152, 8153, 8154,  
`\cs_new_eq:NNN` . . . . . 8155, 8156, 8157, 8158, 8159, 8160,  
`\cs_new_eq:NNN` . . . . . 8161, 8162, 8163, 8164, 8165, 8166,  
`\cs_new_eq:NNN` . . . . . 8167, 8195, 8196, 8197, 8198, 8199,  
`\cs_new_eq:NNN` . . . . . 8327, 8328, 8331, 8382, 8633, 8989,  
`\cs_new_eq:NNN` . . . . . 8993, 9078, 9080, 9100, 9101, 9381,  
`\cs_new_eq:NNN` . . . . . 9382, 9383, 9384, 9387, 9388, 9389,  
`\cs_new_eq:NNN` . . . . . 9390, 9658, 9810, 9866, 9867, 9871,  
`\cs_new_eq:NNN` . . . . . 9872, 9873, 9874, 9875, 9876, 9877,  
`\cs_new_eq:NNN` . . . . . 9878, 9879, 9880, 9881, 9882, 9883,  
`\cs_new_eq:NNN` . . . . . 9884, 9885, 9886, 10027, 10028,  
`\cs_new_eq:NNN` . . . . . 10029, 10030, 10031, 10032, 10033,  
`\cs_new_eq:NNN` . . . . . 10034, 10035, 10036, 10037, 10038,

10039, 10040, 10041, 10042, 10533,  
10585, 10886, 10888, 10889, 10890,  
10892, 10895, 10896, 11210, 11211,  
11212, 11428, 11429, 11430, 11431,  
11432, 11433, 11434, 11435, 12735,  
12757, 12809, 12953, 13035, 13546,  
13834, 13875, 13943, 13949, 14205,  
14206, 14207, 14509, 14539, 14543,  
14544, 14622, 14625, 14628, 14633,  
14637, 14638, 14699, 14701, 14705,  
14706, 15957, 15958, 16197, 16198,  
16199, 16403, 16570, 16846, 16874,  
16882, 16883, 16884, 16893, 16895,  
17696, 17815, 17816, 17817, 18418,  
18429, 18430, 22070, 22072, 22116,  
23363, 23364, 23795, 23801, 23930,  
23931, 24031, 24039, 24060, 24099,  
24100, 24101, 24102, 24279, 25773,  
26371, 27007, 27044, 27045, 27046,  
27077, 27078, 27089, 27090, 27091,  
27196, 27227, 27228, 27301, 27316,  
27317, 27718, 27942, 27943, 27944,  
27945, 27946, 27947, 28943, 28944,  
28961, 28962, 28963, 29955, 30094,  
30221, 30250, 30286, 30288, 30290,  
30843, 30845, 32390, 32391, 32878  
\cs\_new\_nopar:Nn . . . . . 13, 2032, 2096  
\cs\_new\_nopar:Npn . . . . . . . . .  
  . . . . . 11, 329, 330, 1902, 1918, 1924  
\cs\_new\_nopar:Npx 11, 1902, 1918, 1925  
\cs\_new\_protected:Nn . . . . . 13, 2032, 2096  
\cs\_new\_protected:Npn . . . . . . . . .  
  . . . . . . . . . 11, 364, 1205,  
  1587, 1604, 1902, 1938, 1942, 1944,  
  1945, 1946, 1947, 1948, 1949, 1950,  
  1951, 1952, 1957, 1958, 1959, 1960,  
  1962, 2017, 2027, 2029, 2040, 2049,  
  2146, 2155, 2157, 2159, 2161, 2163,  
  2171, 2173, 2174, 2176, 2177, 2179,  
  2234, 2269, 2441, 2470, 2540, 2571,  
  2863, 2876, 2894, 2898, 2901, 2910,  
  3034, 3051, 3061, 3070, 3094, 3102,  
  3114, 3122, 3133, 3135, 3137, 3139,  
  3141, 3152, 3154, 3167, 3175, 3186,  
  3205, 3207, 3209, 3211, 3213, 3283,  
  3385, 3387, 3398, 3405, 3411, 3429,  
  3438, 3443, 3448, 3453, 3458, 3464,  
  3469, 3476, 3482, 3491, 3501, 3503,  
  3505, 3507, 3509, 3514, 3559, 3586,  
  3592, 3597, 3604, 3606, 3610, 3612,  
  3616, 3617, 3620, 3628, 3651, 3653,  
  3655, 3657, 3659, 3661, 3669, 3674,  
  3679, 3687, 3689, 3694, 3699, 3707,  
  3717, 3719, 3724, 3732, 3734, 3736,

10483, 10487, 10489, 10493, 10495,  
10499, 10501, 10505, 10517, 11221,  
11223, 11225, 11230, 11237, 11246,  
11264, 11266, 11268, 11270, 11272,  
11274, 11357, 11374, 11379, 11386,  
11391, 11393, 11410, 11416, 11419,  
11422, 11425, 11441, 11449, 11457,  
11465, 11470, 11477, 11479, 11481,  
11486, 11488, 11500, 11502, 11511,  
11517, 11527, 11535, 11544, 11602,  
11603, 11604, 11623, 11625, 11627,  
11702, 11735, 11737, 11739, 11765,  
11773, 11778, 11780, 11787, 11789,  
11796, 11837, 11866, 11886, 11891,  
11902, 11979, 11981, 12022, 12105,  
12119, 12144, 12166, 12167, 12180,  
12185, 12211, 12220, 12222, 12224,  
12241, 12268, 12270, 12272, 12274,  
12281, 12735, 12739, 12753, 12764,  
12785, 12797, 12798, 12799, 12826,  
12828, 12839, 12841, 12860, 12862,  
12864, 12879, 12881, 12883, 12889,  
12896, 12905, 12907, 12909, 12914,  
12953, 12957, 12960, 12974, 12982,  
12994, 12995, 12996, 13006, 13009,  
13012, 13018, 13024, 13031, 13033,  
13043, 13074, 13085, 13103, 13138,  
13161, 13173, 13192, 13196, 13285,  
13301, 13310, 13318, 13327, 13334,  
13340, 13506, 13540, 13631, 13685,  
13776, 13778, 13780, 13782, 13792,  
13820, 13893, 13898, 13904, 13905,  
13910, 13929, 13944, 13950, 14001,  
14006, 14013, 14014, 14015, 14042,  
14055, 14067, 14077, 14079, 14081,  
14090, 14098, 14211, 14217, 14223,  
14224, 14228, 14230, 14238, 14240,  
14244, 14247, 14250, 14252, 14259,  
14261, 14342, 14464, 14471, 14483,  
14541, 14545, 14555, 14561, 14567,  
14568, 14571, 14573, 14581, 14583,  
14587, 14589, 14591, 14593, 14597,  
14599, 14635, 14639, 14647, 14653,  
14659, 14661, 14665, 14667, 14675,  
14677, 14681, 14683, 14685, 14687,  
14691, 14693, 14703, 14707, 14942,  
14944, 14951, 14956, 14961, 14974,  
14981, 14985, 14994, 14999, 15008,  
15014, 15031, 15047, 15063, 15065,  
15067, 15083, 15094, 15096, 15098,  
15115, 15118, 15125, 15140, 15153,  
15158, 15169, 15176, 15178, 15192,  
15202, 15230, 15239, 15248, 15259,  
15265, 15267, 15269, 15271, 15273,  
15275, 15277, 15279, 15281, 15283,  
15285, 15287, 15289, 15291, 15293,  
15295, 15297, 15299, 15301, 15303,  
15305, 15307, 15309, 15311, 15313,  
15315, 15317, 15319, 15321, 15323,  
15325, 15327, 15329, 15331, 15333,  
15335, 15337, 15339, 15341, 15343,  
15345, 15347, 15349, 15351, 15353,  
15355, 15357, 15359, 15361, 15363,  
15365, 15367, 15369, 15371, 15373,  
15375, 15377, 15379, 15381, 15383,  
15385, 15387, 15389, 15391, 15393,  
15395, 15397, 15399, 15401, 15403,  
15405, 15407, 15409, 15411, 15413,  
15415, 15417, 15419, 15421, 15442,  
15444, 15450, 15456, 15462, 15469,  
15472, 15491, 15498, 15504, 15511,  
15514, 15533, 15554, 15556, 15562,  
15567, 15572, 15592, 15597, 15601,  
15607, 15612, 15618, 15632, 15658,  
15677, 15692, 15706, 15722, 15746,  
15770, 15802, 15888, 15890, 15892,  
15964, 15973, 16008, 16010, 16018,  
16029, 16037, 16044, 16050, 16078,  
16087, 16112, 16114, 16116, 16127,  
16132, 16137, 16151, 16156, 16161,  
16176, 16187, 16210, 16213, 16324,  
16595, 16612, 16614, 16616, 16618,  
16646, 16648, 16650, 16652, 16672,  
16674, 16676, 16678, 16680, 16682,  
16684, 16686, 16688, 18417, 18420,  
18422, 18424, 18433, 18434, 18437,  
18439, 18443, 18444, 18445, 18446,  
18447, 18453, 18455, 18457, 18462,  
18464, 18743, 18750, 18762, 21573,  
22398, 22411, 22450, 22460, 22472,  
22477, 22487, 22495, 22501, 22580,  
22585, 22592, 22594, 22596, 22618,  
22620, 22632, 22643, 22665, 22670,  
22683, 22696, 22704, 22713, 22727,  
22738, 22744, 22809, 22822, 22829,  
22957, 22976, 22983, 22995, 23028,  
23047, 23049, 23051, 23070, 23073,  
23076, 23082, 23088, 23103, 23112,  
23132, 23142, 23153, 23163, 23173,  
23174, 23181, 23187, 23197, 23207,  
23303, 23309, 23311, 23326, 23392,  
23403, 23421, 23431, 23433, 23457,  
23464, 23476, 23479, 23482, 23492,  
23500, 23507, 23516, 23531, 23548,  
23559, 23669, 23676, 23678, 23696,  
23706, 23796, 23799, 23802, 23804,  
23813, 23819, 23825, 23863, 23865,  
23866, 23871, 23877, 23885, 23897,

23913, 23932, 23942, 23950, 23952,  
23960, 23972, 23992, 23994, 23999,  
24007, 24009, 24016, 24021, 24023,  
24025, 24032, 24040, 24042, 24044,  
24046, 24053, 24058, 24061, 24067,  
24294, 24358, 24371, 24382, 24395,  
24428, 24457, 24462, 24467, 24488,  
24497, 24506, 24511, 24518, 24531,  
24533, 24535, 24537, 24543, 24565,  
24578, 24605, 24610, 24644, 24679,  
24700, 24702, 24712, 24721, 24726,  
24735, 24744, 24760, 24773, 24779,  
24790, 24803, 24809, 24828, 24848,  
24879, 24890, 24905, 24918, 24936,  
24944, 24949, 24951, 24953, 24970,  
24989, 24991, 25014, 25026, 25046,  
25067, 25074, 25081, 25093, 25099,  
25157, 25192, 25201, 25220, 25239,  
25245, 25308, 25318, 25320, 25322,  
25329, 25385, 25398, 25414, 25419,  
25432, 25448, 25455, 25462, 25464,  
25466, 25473, 25487, 25503, 25512,  
25526, 25538, 25556, 25565, 25567,  
25579, 25588, 25600, 25613, 25620,  
25640, 25671, 25705, 25723, 25729,  
25738, 25777, 25790, 25812, 25829,  
25851, 25860, 25871, 25900, 25915,  
25924, 25933, 25945, 25952, 25954,  
25956, 25976, 25981, 25988, 25993,  
25998, 26003, 26052, 26087, 26129,  
26145, 26165, 26177, 26191, 26225,  
26239, 26250, 26257, 26266, 26301,  
26307, 26310, 26318, 26324, 26327,  
26336, 26339, 26342, 26345, 26350,  
26359, 26362, 26365, 26370, 26376,  
26381, 26386, 26391, 26399, 26419,  
26421, 26425, 26426, 26450, 26458,  
26467, 26479, 26488, 26496, 26536,  
26580, 26607, 26612, 26614, 26644,  
26672, 26983, 26985, 26987, 26994,  
27010, 27016, 27018, 27022, 27024,  
27028, 27030, 27034, 27036, 27050,  
27056, 27059, 27065, 27068, 27074,  
27081, 27083, 27085, 27087, 27104,  
27106, 27115, 27118, 27121, 27124,  
27126, 27133, 27151, 27153, 27158,  
27165, 27170, 27177, 27183, 27191,  
27197, 27203, 27211, 27216, 27221,  
27223, 27225, 27231, 27233, 27235,  
27240, 27245, 27250, 27257, 27262,  
27269, 27274, 27281, 27287, 27295,  
27302, 27308, 27320, 27323, 27341,  
27344, 27347, 27357, 27391, 27402,  
27413, 27424, 27435, 27446, 27459,  
27465, 27471, 27486, 27493, 27503,  
27508, 27511, 27514, 27528, 27531,  
27534, 27548, 27551, 27554, 27565,  
27568, 27571, 27586, 27589, 27592,  
27601, 27615, 27618, 27621, 27627,  
27633, 27645, 27731, 27740, 27749,  
27758, 27771, 27784, 27797, 27803,  
27809, 27841, 27854, 27867, 27868,  
27869, 27876, 27883, 27909, 27910,  
27911, 27923, 27948, 27958, 27965,  
27972, 27975, 27978, 27990, 27993,  
27996, 28008, 28014, 28020, 28026,  
28028, 28030, 28036, 28057, 28059,  
28061, 28067, 28101, 28116, 28212,  
28215, 28218, 28260, 28278, 28284,  
28290, 28302, 28321, 28330, 28341,  
28347, 28352, 28360, 28373, 28380,  
28387, 28399, 28421, 28424, 28427,  
28442, 28449, 28455, 28464, 28471,  
28479, 28485, 28491, 28530, 28536,  
28542, 28563, 28572, 28591, 28596,  
28610, 28615, 28628, 28639, 28651,  
28665, 28749, 28786, 28794, 28818,  
28862, 28870, 28894, 28897, 28900,  
28945, 28951, 28953, 28968, 28969,  
29871, 31608, 32047, 32050, 32053,  
32056, 32059, 32104, 32107, 32110,  
32165, 32167, 32169, 32189, 32192,  
32274, 32276, 32278, 32284, 32286,  
32288, 32294, 32296, 32353, 32359,  
32375, 32377, 32379, 32381, 32392,  
32397, 32402, 32407, 32412, 32429,  
32430, 32432, 32435, 32438, 32449,  
32451, 32463, 32468, 32473, 32516,  
32518, 32520, 32522, 32539, 32552,  
32557, 32583, 32607, 32609, 32630,  
32649, 32656, 32673, 32697, 32702  
\cs\_new\_protected:Npx 11, 358, 359,  
363, 1902, 1938, 1943, 2034, 2098,  
2878, 2882, 2887, 3046, 3050, 4758,  
10523, 11323, 11338, 11343, 11348,  
11990, 11992, 11994, 11996, 11998,  
12007, 12009, 12011, 12290, 12292,  
12294, 12296, 12298, 12307, 12309,  
12311, 12776, 13522, 13916, 24673,  
24687, 24689, 27834, 28733, 28743  
\cs\_new\_protected:Nn . . . . .  
..... 13, 2032, 2096  
\cs\_new\_protected\_nopar:Npn . . . . .  
..... 11, 1902, 1919, 1932, 1936  
\cs\_new\_protected\_nopar:Npx . . . . .  
..... 11, 1902, 1932, 1937  
\cs\_prefix\_spec:N . . . . .  
..... 18, 2187, 32995, 32996

```

\cs_replacement_spec:N 18, 2187, 15901, 32999, 33000
\cs_set:Nn 13, 334, 2032, 2096
.cs_set:Np 187, 15301
\cs_set:Npn 10, 11,
105, 105, 321, 330, 334, 599, 1479,
1509, 1516, 1518, 1521, 1522, 1523,
1524, 1525, 1526, 1527, 1528, 1529,
1530, 1531, 1532, 1533, 1534, 1535,
1536, 1537, 1538, 1539, 1540, 1541,
1543, 1544, 1545, 1546, 1547, 1548,
1549, 1550, 1551, 1574, 1576, 1578,
1581, 1598, 1647, 1650, 1712, 1760,
1762, 1764, 1766, 1771, 1777, 1778,
1782, 1789, 1792, 1852, 1854, 1856,
1858, 1860, 1862, 1864, 1866, 1885,
1902, 1918, 1926, 1926, 2032, 2096,
3524, 3546, 3851, 3908, 4036, 4806,
6435, 6592, 8232, 8240, 9662, 10074,
10163, 10572, 10874, 11234, 11504,
11782, 11784, 13050, 13372, 15302,
15304, 15841, 16621, 16629, 16638,
16655, 16663, 16691, 23043, 24071,
24072, 24073, 24390, 24391, 24957,
24959, 24976, 24978, 25272, 25299,
25338, 25832, 26131, 28981, 29253,
31711, 31712, 31774, 31781, 31786,
31787, 32528, 32530, 32536, 32654
\cs_set:Npx 11,
340, 702, 1479, 1926, 1927, 3910,
5792, 5820, 10150, 11232, 11251,
11257, 13108, 13109, 13110, 13111,
13112, 23815, 23821, 25489, 29255
\cs_set_eq:NN 15, 106,
331, 528, 1714, 1944, 2882, 2900,
3050, 3616, 5513, 5522, 6337, 7717,
7718, 7720, 7874, 7875, 7886, 9027,
9089, 9091, 9411, 10526, 10745,
10923, 11228, 11249, 11256, 13114,
13115, 13116, 13117, 13119, 13121,
13122, 15129, 15145, 15149, 15197,
15208, 15218, 22742, 23305, 23306,
23310, 23460, 23503, 23528, 25264,
25273, 25296, 25838, 32525, 32535
\cs_set_nopar:Nn 13, 2032, 2096
\cs_set_nopar:Npn 10, 11, 134, 330,
1479, 1508, 1566, 1567, 1918, 1920,
15085, 15156, 15616, 29288, 29290
\cs_set_nopar:Npx 11, 698,
714, 1196, 1479, 1512, 1918, 1921,
2271, 2472, 3584, 14976, 14989,
14996, 15003, 15004, 15134, 15604,
15609, 32376, 32394, 32399, 32433
\cs_set_protected:Nn . 13, 2032, 2096
.cs_set_protected:Np 187, 15301
\cs_set_protected:Npn 10, 11, 165, 331, 1479,
1495, 1497, 1499, 1501, 1503, 1505,
1510, 1553, 1554, 1559, 1564, 1565,
1568, 1580, 1582, 1584, 1585, 1586,
1588, 1597, 1599, 1601, 1602, 1603,
1605, 1614, 1626, 1652, 1669, 1688,
1696, 1704, 1713, 1715, 1717, 1729,
1743, 1780, 1868, 1881, 1883, 1887,
1889, 1891, 1899, 1904, 1938, 1938,
1972, 1993, 3068, 3229, 4043, 4253,
4463, 4727, 4754, 5938, 5991, 7791,
10515, 10624, 11011, 11030, 11355,
11910, 11976, 12277, 12279, 12630,
12758, 13030, 13032, 13171, 13252,
13352, 13368, 13802, 14609, 14723,
14869, 15116, 15306, 15308, 15776,
16576, 17070, 17146, 17730, 17804,
17818, 18040, 18057, 18092, 18128,
18143, 18160, 19709, 23307, 24685,
24710, 24719, 25249, 25258, 25260,
25262, 25265, 25267, 25274, 25276,
25281, 25283, 25288, 25290, 25292,
25294, 25297, 25995, 25996, 26423,
27846, 27859, 27890, 28173, 28990,
29265, 29275, 29285, 29310, 29325,
29343, 29351, 29383, 30852, 30855,
30888, 30899, 30915, 31140, 31143,
31177, 31180, 31198, 31217, 31337,
31340, 31369, 31400, 31689, 31701,
31763, 31814, 32647, 32653, 32888
\cs_set_protected:Npx .. 11, 151,
1479, 1938, 1939, 12152, 15120, 32893
\cs_set_protected_nopar:Nn 14, 2032, 2096
\cs_set_protected_nopar:Npn 12, 330, 1479, 1932, 1932
\cs_set_protected_nopar:Npx 12, 1479, 1932, 1933
\cs_show:N 16, 16, 22, 337, 2173
\cs_split_function:N 17, 1593, 1610, 1722, 1723, 1780,
2004, 2045, 2869, 3172, 3401, 3422
\cs_to_sr:N 1199
\cs_to_str:N 4,
17, 51, 60, 325, 325, 326, 357, 429,
1771, 1786, 2667, 2833, 3529, 3551,
5316, 5317, 5318, 5319, 5320, 5321,
5322, 5323, 5324, 5325, 5326, 5327,
13035, 16574, 23829, 25230, 29811,
31673, 32380, 32466, 32471, 32479
\cs_undefine:N 15, 525, 704, 1960, 4490,
```

4714, 12316, 12317, 12318, 12760,  
 22416, 22700, 22760, 28964, 28965  
 cs internal commands:  
 \\_\_cs\_count\_signature:N . . . . . 2003  
 \\_\_cs\_count\_signature:n . . . . . 2003  
 \\_\_cs\_count\_signature:nnN . . . . . 2003  
 \\_\_cs\_generate\_from\_signature:n . . . . . 2054, 2067  
 \\_\_cs\_generate\_from\_signature:NNn . . . . . 2036, 2040  
 \\_\_cs\_generate\_from\_signature:nnNNNn . . . . . 2044, 2049  
 \\_\_cs\_generate\_internal\_c:NN . . . . . 3135  
 \\_\_cs\_generate\_internal\_end:w . . . . . 3118, 3152  
 \\_\_cs\_generate\_internal\_long:nnnNNn . . . . . 3156, 3160  
 \\_\_cs\_generate\_internal\_long:w . . . . . 3119, 3154  
 \\_\_cs\_generate\_internal\_loop:nwnw . . . . . 3116,  
     3122, 3134, 3136, 3138, 3140, 3143  
 \\_\_cs\_generate\_internal\_N:NN . . . . . 3133  
 \\_\_cs\_generate\_internal\_n:NN . . . . . 3137  
 \\_\_cs\_generate\_internal\_one\_-  
     go:NNn . . . . . 364, 3091, 3114  
 \\_\_cs\_generate\_internal\_other:NN . . . . . 3127, 3141  
 \\_\_cs\_generate\_internal\_test:Nw . . . . . 3076, 3098, 3102  
 \\_\_cs\_generate\_internal\_test\_-  
     aux:w . . . . . 3078, 3094, 3099, 3105, 3108  
 \\_\_cs\_generate\_internal\_variant:n . . . . . 368, 3041, 3046, 3226, 3232  
 \\_\_cs\_generate\_internal\_variant:NNn . . . . . 364, 3066, 3070  
 \\_\_cs\_generate\_internal\_variant:wwNnw . . . . . 3048, 3061  
 \\_\_cs\_generate\_internal\_variant\_-  
     loop:n . . . . . 3046  
 \\_\_cs\_generate\_internal\_x:NN . . . . . 3139  
 \\_\_cs\_generate\_variant:N . . . . . 2865, 2878  
 \\_\_cs\_generate\_variant:n . . . . . 3167  
 \\_\_cs\_generate\_variant:nnNN . . . . . 2868, 2901  
 \\_\_cs\_generate\_variant:nnNnn . . . . . 3167  
 \\_\_cs\_generate\_variant:Nnnw . . . . . 2908, 2910  
 \\_\_cs\_generate\_variant:w . . . . . 3167  
 \\_\_cs\_generate\_variant:ww . . . . . 2878  
 \\_\_cs\_generate\_variant:wwNN . . . . . 360, 361, 2917, 3034  
 \\_\_cs\_generate\_variant:wwNw . . . . . 2878  
 \\_\_cs\_generate\_variant\_F\_-  
     form:nnm . . . . . 3167  
 \\_\_cs\_generate\_variant\_loop:nNwN  
     . . . . . 360, 361, 2918, 2930  
 \\_\_cs\_generate\_variant\_loop\_-  
     base:N . . . . . 2930  
 \\_\_cs\_generate\_variant\_loop\_-  
     end:nwwwNNnn . . . . . 360, 361, 2920, 2930  
 \\_\_cs\_generate\_variant\_loop\_-  
     invalid>NNwNNnn . . . . . 361, 2930  
 \\_\_cs\_generate\_variant\_loop\_-  
     long:wNNnn . . . . . 361, 2923, 2930  
 \\_\_cs\_generate\_variant\_loop\_-  
     same:w . . . . . 361, 2930  
 \\_\_cs\_generate\_variant\_loop\_-  
     special>NNwNNnn . . . . . 2930, 3029  
 \\_\_cs\_generate\_variant\_p\_-  
     form:nnn . . . . . 3167  
 \\_\_cs\_generate\_variant\_same:N . . . . .  
     . . . . . 361, 2975, 3023  
 \\_\_cs\_generate\_variant\_T\_-  
     form:nnn . . . . . 3167  
 \\_\_cs\_generate\_variant\_TF\_-  
     form:nnn . . . . . 3167  
 \\_\_cs\_get\_function\_name:N . . . . . 324  
 \\_\_cs\_get\_function\_signature:N . . . . . 324  
 \\_\_cs\_parm\_from\_arg\_count\_-  
     test:nnTF . . . . . 1972  
 \\_\_cs\_split\_function\_auxi:w . . . . . 1780  
 \\_\_cs\_split\_function\_auxii:w . . . . . 1780  
 \\_\_cs\_tmp:w . . . . . 325, 358,  
     363, 364, 368, 1780, 1795, 1902,  
     1918, 1920, 1921, 1922, 1923, 1924,  
     1925, 1926, 1927, 1928, 1929, 1930,  
     1931, 1932, 1933, 1934, 1935, 1936,  
     1937, 1938, 1939, 1940, 1941, 1942,  
     1943, 2032, 2072, 2073, 2074, 2075,  
     2076, 2077, 2078, 2079, 2080, 2081,  
     2082, 2083, 2084, 2085, 2086, 2087,  
     2088, 2089, 2090, 2091, 2092, 2093,  
     2094, 2095, 2096, 2104, 2105, 2106,  
     2107, 2108, 2109, 2110, 2111, 2112,  
     2113, 2114, 2115, 2116, 2117, 2118,  
     2119, 2120, 2121, 2122, 2123, 2124,  
     2125, 2126, 2127, 2882, 2900, 3042,  
     3050, 3068, 3113, 3229, 3236, 3237,  
     3238, 3239, 3240, 3241, 3242, 3243,  
     3244, 3245, 3246, 3247, 3248, 3249,  
     3250, 3251, 3252, 3253, 3254, 3255,  
     3256, 3257, 3258, 3259, 3260, 3261,  
     3262, 3263, 3264, 3265, 3266, 3267,  
     3268, 3269, 3270, 3271, 3272, 3273,  
     3274, 3275, 3276, 3277, 3278, 3279  
 \\_\_cs\_to\_str:N . . . . . 325, 1771

\\_\_cs\_to\_str:w ..... 325, 1771  
 \\_\_cs\_use\_i\_delimit\_by\_s\_stop:nw ..... 2859, 3180  
 \\_\_cs\_use\_none\_delimit\_by\_q\_- recursion\_stop:w ..... 2859, 2906, 2913, 3193  
 \\_\_cs\_use\_none\_delimit\_by\_s\_- stop:w ..... 2859, 3184  
 csc ..... 215  
 cscd ..... 215  
 \csname ..... 4, 21, 25, 30, 34, 35, 46, 68, 70, 71, 72, 83, 90, 111, 115, 134, 228  
 \cssstring ..... 809  
 \currentcjktoken ..... 1113, 1170  
 \currentgrouplevel ..... 520  
 \currentgroupype ..... 521  
 \currentifbranch ..... 522  
 \currentiflevel ..... 523  
 \currentiftype ..... 524  
 \currentspacingmode ..... 1114  
 \currentxspacingmode ..... 1115

**D**

\d ..... 29543, 31807  
 \date ..... 305  
 \day ..... 229, 1315, 9686  
 dd ..... 218  
 \deadcycles ..... 230  
 debug commands:  
 \debug\_off: ..... 305  
 \debug\_off:n ..... 24, 1204, 1204, 1205, 1205, 1554  
 \debug\_on: ..... 305  
 \debug\_on:n ..... 24, 1204, 1204, 1554  
 \debug\_resume: ..... 24, 1090, 1564, 27768  
 \debug\_suspend: ..... 24, 1090, 1564, 27761

debug internal commands:  
 \g\_\_debug\_deprecation\_off\_tl ..... 1566  
 \g\_\_debug\_deprecation\_on\_tl ..... 1566

\def ..... 52, 53, 54, 89, 91, 92, 108, 109, 112, 123, 147, 186, 231

default commands:  
 .default:n ..... 188, 15317  
 \defaulthyphenchar ..... 232  
 \defaultskewchar ..... 233  
 deg ..... 217  
 \delcode ..... 234  
 \delimiter ..... 235  
 \delimiterfactor ..... 236  
 \delimitershortfall ..... 237

deprecation internal commands:  
 \\_\_deprecation\_date\_compare:nNnTF ..... 32569, 32586, 32589

\\_\_deprecation\_date\_compare\_- aux:w ..... 32569  
 \l\_\_deprecation\_grace\_period\_- bool 1203, 32566, 32585, 32595, 32669  
 \\_\_deprecation\_just\_error:nnNN ..... 1204, 32607  
 \\_\_deprecation\_minus\_six\_- months:w ..... 32583  
 \\_\_deprecation\_not\_yet\_DEPRECATED:nTF ..... 1204, 32583, 32617  
 \\_\_deprecation\_old:Nnn ..... 32697  
 \\_\_deprecation\_old\_protected:Nnn ..... 32697  
 \\_\_deprecation\_patch\_aux:Nn ..... 1205, 32607  
 \\_\_deprecation\_patch\_aux:nnNNnn ..... 32607  
 \\_\_deprecation\_warn\_once:nnNnn ..... 32607  
 \detokenize ..... 46, 134, 525  
 \DH ..... 29549, 31381, 31723  
 \dh ..... 29549, 31381, 31733

dim commands:

\dim\_abs:n ..... 171, 680, 14268  
 \dim\_add:Nn ..... 171, 14250  
 \dim\_case:nn ..... 174, 14348  
 \dim\_case:nnn ..... 32761  
 \dim\_case:mnTF ..... 174, 14348, 14353, 14358, 32762  
 \dim\_compare:nNnTF ..... 172, 173, 174, 174, 174, 175, 204, 14303, 14372, 14408, 14416, 14425, 14431, 14443, 14446, 14457, 28119, 28122, 28127, 28141, 28144, 28149, 28497, 28502, 28512, 28641, 28653, 32067, 32084, 32118, 32132, 32142  
 \dim\_compare:nTF ..... 172, 173, 175, 175, 175, 175, 14308, 14380, 14388, 14397, 14403  
 \dim\_compare\_p:n ..... 173, 14308  
 \dim\_compare\_p:nNn ..... 172, 14303  
 \dim\_const:Nn ..... 170, 673, 683, 14217, 14547, 14548, 15960  
 \dim\_do\_until:nn ..... 175, 14378  
 \dim\_do\_until:nNnn ..... 174, 14406  
 \dim\_do\_while:nn ..... 175, 14378  
 \dim\_do\_while:nNnn ..... 174, 14406  
 \dim\_eval:n ..... 172, 173, 176, 176, 673, 1069, 14220, 14351, 14356, 14361, 14366, 14461, 14489, 14542, 14546, 27825, 27899, 27985, 28003, 28040, 28044, 28045, 28049, 28053, 28054, 28071, 28076, 28082, 28089, 28096, 28239, 28263, 28266, 28267, 28274, 28356,

28357, 28364, 28365, 28468, 28475,  
 28624, 28625, 28907, 28908, 28909  
`\dim_gadd:Nn` ..... 171, 14250  
`.dim_gset:N` ..... 188, 15325  
`\dim_gset:Nn` ..... 171, 673, 14238  
`\dim_gset_eq:NN` ..... 171, 14244  
`\dim_gsub:Nn` ..... 171, 14250  
`\dim_gzero:N` ..... 170, 14223, 14231  
`\dim_gzero_new:N` ..... 170, 14228  
`\dim_if_exist:NTF` .....  
       ..... 170, 14229, 14231, 14234  
`\dim_if_exist_p:N` ..... 170, 14234  
`\dim_log:N` ..... 178, 14543  
`\dim_log:n` ..... 178, 14543  
`\dim_max:nn` .. 171, 14268, 28335, 28339  
`\dim_min:nn` .....  
       .... 171, 14268, 28333, 28337, 28350  
`\dim_new:N` .....  
       ... 170, 170, 14211, 14219, 14229,  
           14231, 14549, 14550, 14551, 14552,  
           27332, 27333, 27334, 27335, 27336,  
           27337, 27338, 27339, 27686, 27710,  
           27711, 27714, 27715, 27716, 27717,  
           28207, 28208, 28209, 28210, 28211,  
           28371, 28372, 28713, 28715, 28716  
`\dim_ratio:nn` .. 172, 681, 14299, 14536  
`.dim_set:N` ..... 188, 15325  
`\dim_set:Nn` ..... 171, 14238,  
       27359, 27360, 27361, 27393, 27404,  
       27488, 27489, 27490, 27505, 27603,  
       27604, 27605, 27607, 27609, 27611,  
       27815, 27888, 28121, 28125, 28143,  
       28147, 28178, 28192, 28269, 28304,  
       28312, 28323, 28324, 28325, 28326,  
       28332, 28334, 28336, 28338, 28343,  
       28349, 28432, 28434, 28436, 28444,  
       28446, 28500, 28575, 28576, 28578,  
       28580, 28598, 28599, 28714, 28826,  
       28827, 28873, 28874, 28875, 28877  
`\dim_set_eq:NN` 171, 14244, 27837, 27839  
`\dim_show:N` ..... 177, 14539  
`\dim_show:n` ..... 178, 682, 14541  
`\dim_sign:n` ..... 176, 14491  
`\dim_step_function:nnnN` .....  
       ..... 175, 679, 14434, 14486  
`\dim_step_inline:nnnn` ... 175, 14464  
`\dim_step_variable:nnnNn` .. 176, 14464  
`\dim_sub:Nn` ..... 171, 14250  
`\dim_to_decimal:n` .....  
       ..... 176, 14512, 14528, 14533  
`\dim_to_decimal_in_bp:n` .....  
       ..... 177, 177, 14527  
`\dim_to_decimal_in_sp:n` .... 177,  
       177, 770, 14529, 17208, 17245, 17843  
`\dim_to_decimal_in_unit:nn` 177, 14531  
`\dim_to_fp:n` .. 177, 770, 789, 14539,  
       22035, 27397, 27398, 27408, 27409,  
       27477, 27480, 27481, 27506, 27521,  
       27522, 27541, 27542, 27560, 27577,  
       27580, 27581, 28132, 28133, 28134,  
       28154, 28155, 28156, 28166, 28167,  
       28183, 28184, 28185, 28186, 28196,  
       28197, 28308, 28309, 28316, 28317,  
       28390, 28393, 28394, 28445, 28447  
`\dim_until_do:nn` ..... 175, 14378  
`\dim_until_do:nNnn` ..... 174, 14406  
`\dim_use:N` ..... 176,  
       176, 1069, 14271, 14277, 14278,  
       14279, 14285, 14286, 14287, 14311,  
       14330, 14490, 14494, 14509, 14515,  
       28271, 28275, 28282, 28288, 28297,  
       28298, 28299, 28453, 28460, 28606  
`\dim_while_do:nn` ..... 175, 14378  
`\dim_while_do:nNnn` ..... 175, 14406  
`\dim_zero:N` 170, 170, 14223, 14229,  
       27362, 27491, 27606, 28112, 28113  
`\dim_zero_new:N` ..... 170, 14228  
`\c_max_dim` ..... 178, 181, 723,  
       14547, 14642, 15989, 16031, 16039,  
       28323, 28324, 28325, 28326, 28343  
`\g_tmpa_dim` ..... 178, 14549  
`\l_tmpa_dim` ..... 178, 14549  
`\g_tmpb_dim` ..... 178, 14549  
`\l_tmpb_dim` ..... 178, 14549  
`\c_zero_dim` .....  
       . 178, 14443, 14446, 14499, 14547,  
       14641, 16056, 27218, 27242, 27676,  
       28119, 28122, 28127, 28141, 28144,  
       28149, 28497, 28502, 28512, 32071,  
       32082, 32088, 32100, 32118, 32122,  
       32130, 32132, 32136, 32142, 32152  
**dim internal commands:**  
`\_dim_abs:N` ..... 14268  
`\_dim_case:nnTF` ..... 14348  
`\_dim_case:nw` ..... 14348  
`\_dim_case_end:nw` ..... 14348  
`\_dim_compare:w` ..... 14308  
`\_dim_compare:wNN` ..... 675, 14308  
`\_dim_compare_!:_w` ..... 14308  
`\_dim_compare_<:_w` ..... 14308  
`\_dim_compare_=:_w` ..... 14308  
`\_dim_compare_>:_w` ..... 14308  
`\_dim_compare_end:w` .. 14316, 14340  
`\_dim_compare_error:` ... 675, 14308  
`\_dim_eval:w` ..... 681, 14205,  
       14239, 14241, 14251, 14255, 14260,  
       14264, 14271, 14277, 14278, 14279,  
       14285, 14286, 14287, 14302, 14305,

14311, 14330, 14335, 14437, 14438,  
 14439, 14490, 14494, 14515, 14530  
`\_dim_eval_end`: ..... 14205,  
 14239, 14241, 14251, 14255, 14260,  
 14264, 14271, 14281, 14289, 14302,  
 14305, 14490, 14494, 14515, 14530  
`\_dim_maxmin:wwN` ..... 14268  
`\_dim_ratio:n` ..... 14299  
`\_dim_sign:Nw` ..... 14491  
`\_dim_step:NnnnN` ..... 14434  
`\_dim_step>NNnnnn` ..... 14464  
`\_dim_step:wwwN` ..... 14434  
`\_dim_tmp:w` ..... 674  
`\_dim_to_decimal:w` ..... 14512  
`\_dim_use_none_delimit_by_s_-`  
 stop:w ..... 14210, 14326  
`\dimen` ..... 238, 11072  
`\dimendef` ..... 239  
`\dimexpr` ..... 526  
`\directlua` ..... 6, 37, 39, 810  
`\disablecjktoken` ..... 1171  
`\discretionary` ..... 240  
`\disinhibitglue` ..... 1116  
`\displayindent` ..... 241  
`\displaylimits` ..... 242  
`\displaystyle` ..... 243  
`\displaywidowpenalties` ..... 527  
`\displaywidowpenalty` ..... 244  
`\displaywidth` ..... 245  
`\divide` ..... 246  
`\DJ` ..... 29550, 31382, 31724  
`\dj` ..... 29550, 31382, 31734  
`\do` ..... 1221  
`\doublehyphendemerits` ..... 247  
`\dp` ..... 248  
`\draftmode` ..... 914  
`\dtou` ..... 1117  
`\dump` ..... 249  
`\dviextension` ..... 811  
`\dvifeedback` ..... 812  
`\dvivariable` ..... 813

**E**

`\edef` ..... 33, 98, 121, 250  
`\efcode` ..... 689  
`\elapsedtime` ..... 778  
`\else` 5, 26, 28, 69, 73, 76, 79, 80, 84, 85, 251  
 else commands:  
`\else:` ..... 23, 101, 101, 102, 106,  
 113, 113, 165, 184, 249, 249, 249,  
 319, 320, 326, 358, 403, 533, 815,  
 1417, 1462, 1640, 1648, 1674, 1800,  
 1803, 1812, 1818, 1828, 1831, 1840,  
 1846, 1966, 1988, 1997, 2011, 2069,

2070, 2131, 2293, 2573, 2725, 2753,  
 2768, 2776, 2813, 2883, 2934, 2935,  
 2937, 2941, 2953, 2954, 2955, 2956,  
 2957, 2958, 2959, 2960, 2961, 3025,  
 3026, 3028, 3077, 3107, 3194, 3306,  
 3344, 3352, 3363, 3373, 3392, 3416,  
 3420, 3462, 3521, 3538, 3949, 3959,  
 3974, 3983, 3993, 4009, 4023, 4067,  
 4082, 4349, 4367, 4385, 4393, 4403,  
 4419, 4442, 4453, 4459, 4605, 4617,  
 4666, 4669, 4672, 4843, 4850, 4856,  
 5092, 5148, 5151, 5154, 5166, 5181,  
 5395, 5403, 5411, 5558, 5609, 5610,  
 5614, 5619, 5660, 5713, 5813, 6064,  
 6094, 6097, 6127, 6130, 6147, 6150,  
 6251, 6256, 6274, 6293, 6296, 6345,  
 6350, 6353, 6468, 6480, 6489, 6611,  
 6616, 7733, 7811, 7820, 8220, 8231,  
 8252, 8268, 8271, 8292, 8330, 8430,  
 8457, 8495, 8503, 8804, 8837, 8888,  
 9005, 9030, 9056, 9065, 9125, 9157,  
 9177, 9199, 9217, 9233, 9243, 9259,  
 9269, 9361, 9363, 9365, 9367, 9969,  
 9984, 10006, 10020, 10545, 10548,  
 10556, 10562, 10591, 10597, 10669,  
 10680, 10700, 10818, 10821, 10824,  
 10827, 10830, 10833, 10836, 10905,  
 10910, 10915, 10920, 10927, 10934,  
 10939, 10944, 10949, 10954, 10959,  
 10964, 10969, 10974, 10996, 11002,  
 11005, 11041, 11044, 11178, 11187,  
 11195, 11204, 11241, 11280, 11294,  
 11303, 11313, 11370, 11668, 12818,  
 13849, 13858, 13869, 14274, 14295,  
 14306, 14316, 14341, 14501, 14504,  
 15994, 16251, 16268, 16269, 16284,  
 16294, 16389, 16465, 16527, 16530,  
 16544, 16562, 16566, 16804, 16817,  
 16837, 16865, 16866, 16888, 16909,  
 16932, 16933, 16966, 16983, 17001,  
 17036, 17040, 17076, 17093, 17099,  
 17103, 17107, 17266, 17299, 17307,  
 17340, 17344, 17356, 17366, 17376,  
 17407, 17420, 17455, 17465, 17484,  
 17497, 17510, 17514, 17525, 17548,  
 17565, 17577, 17591, 17604, 17608,  
 17616, 17618, 17628, 17639, 17655,  
 17671, 17677, 17682, 17689, 17711,  
 17741, 17764, 17792, 17795, 17971,  
 17975, 17982, 18001, 18015, 18019,  
 18026, 18048, 18065, 18071, 18103,  
 18135, 18151, 18171, 18212, 18227,  
 18260, 18262, 18268, 18283, 18336,  
 18487, 18503, 18514, 18552, 18555,

|                                    |                                                           |
|------------------------------------|-----------------------------------------------------------|
| 18558, 18561, 18592, 18601, 18610, | 31654                                                     |
| 18613, 18784, 18797, 18800, 18807, | 218                                                       |
| 18825, 18849, 18850, 18865, 18875, | 252                                                       |
| 18924, 18927, 18936, 18948, 18959, | 31627                                                     |
| 18973, 18986, 19026, 19060, 19080, | 1172                                                      |
| 19117, 19135, 19138, 19144, 19158, | 253,                                                      |
| 19193, 19211, 19214, 19217, 19220, | 302, 18681, 29049, 29560, 29567, 31668                    |
| 19281, 19354, 19424, 19425, 19434, | end internal commands:                                    |
| 19469, 19552, 19556, 19560, 19622, | <u>_regex_end</u> . . . . . 26442                         |
| 19657, 19672, 19937, 19966, 19970, | \endcsname . . . . . 4, 21, 25, 30, 34, 35, 46,           |
| 20130, 20139, 20193, 20204, 20220, | 68, 70, 71, 72, 83, 90, 111, 115, 134, 254                |
| 20228, 20287, 20367, 20378, 20383, | \endgroup . . . . . 19, 20, 24, 29, 36, 52, 102, 119, 255 |
| 20417, 20430, 20442, 20448, 20569, | \endinput . . . . . 103, 256                              |
| 20577, 20616, 20623, 20645, 20673, | \endL . . . . . 528                                       |
| 20688, 20692, 20714, 20745, 20748, | \endlinechar . . . . . 133, 145, 257                      |
| 20773, 20776, 20817, 20825, 20836, | \endR . . . . . 529                                       |
| 20839, 20954, 20969, 20984, 20999, | \enquote . . . . . 18683                                  |
| 21014, 21029, 21050, 21095, 21401, | \ensuremath . . . . . 1140, 29562                         |
| 21439, 21440, 21449, 21493, 21548, | \epTeXinputencoding . . . . . 1118                        |
| 21549, 21550, 21654, 21676, 21691, | \epTeXversion . . . . . 1119                              |
| 21709, 21757, 21773, 21979, 22046, | \eqno . . . . . 258                                       |
| 22051, 22215, 22251, 22264, 22294, | \errhelp . . . . . 94, 259                                |
| 22298, 22306, 22333, 22359, 22367, | \errmessage . . . . . 95, 260                             |
| 22384, 22387, 22436, 22440, 22492, | \ERROR . . . . . 10637                                    |
| 22551, 22563, 22988, 22989, 23438, | \errorcontextlines . . . . . 261                          |
| 23441, 23444, 23454, 23469, 23496, | \errorstopmode . . . . . 262                              |
| 23511, 23538, 23554, 23587, 23595, | \escapechar . . . . . 263                                 |
| 23597, 23599, 23601, 23603, 23605, | escapehex . . . . . 29054                                 |
| 23607, 23609, 23627, 23648, 23652, | \ETC . . . . . 23784                                      |
| 23724, 23728, 23917, 23918, 23923, | \eTeXglueshrinkorder . . . . . 814                        |
| 23924, 23939, 23946, 24144, 24154, | \eTeXgluestretchorder . . . . . 815                       |
| 24198, 24207, 24219, 24220, 24222, | \eTeXrevision . . . . . 530                               |
| 24224, 24227, 24228, 24231, 24232, | \eTeXversion . . . . . 531                                |
| 24241, 24243, 24245, 24248, 24249, | \etoksapp . . . . . 816                                   |
| 24251, 24287, 24290, 24311, 24314, | \etokspre . . . . . 817                                   |
| 24322, 24330, 24333, 24342, 24345, | \euc . . . . . 1120                                       |
| 24354, 24362, 24365, 24375, 24481, | \everycr . . . . . 264                                    |
| 24588, 24632, 24636, 24639, 24650, | \everydisplay . . . . . 265                               |
| 24655, 24754, 24900, 24913, 25002, | \everyeof . . . . . 532                                   |
| 25031, 25070, 25088, 25197, 25231, | \everyhbox . . . . . 266                                  |
| 25481, 25499, 25518, 25553, 25606, | \everyjob . . . . . 44, 45, 267                           |
| 25653, 25657, 25664, 25685, 25696, | \everymath . . . . . 268                                  |
| 25837, 25953, 26039, 26097, 26171, | \everypar . . . . . 269                                   |
| 26206, 26218, 26244, 26262, 26454, | \everyvbox . . . . . 270                                  |
| 26598, 26623, 26675, 27093, 27095, | ex . . . . . 218                                          |
| 27101, 29319, 29416, 29420, 29431, | \exceptionpenalty . . . . . 818                           |
| 29452, 29456, 29465, 29466, 29467, | \exhyphenpenalty . . . . . 271                            |
| 29468, 29469, 29470, 29471, 29472, | exp . . . . . 213                                         |
| 29473, 29484, 29498, 29501, 29504, | exp commands:                                             |
| 29507, 29510, 29513, 29516, 30447, | \exp:w . . . . . 36, 36, 37,                              |
| 30451, 30454, 30458, 30467, 30470, | 319, 325, 341, 342, 343, 350, 351,                        |
| 30473, 30476, 30479, 30482, 30496, | 406, 407, 423, 536, 551, 628, 646,                        |
| 30499, 30502, 30505, 30508, 30520, | 760, 761, 763, 764, 766, 767, 786,                        |
| 30523, 30526, 30529, 32040, 32160  | 790, 791, 1197, 1439, 1575, 1577,                         |

2265, 2278, 2284, 2332, 2336, 2340,  
2345, 2351, 2357, 2373, 2385, 2391,  
2397, 2402, 2404, 2411, 2418, 2449,  
2454, 2463, 2468, 2477, 2479, 2487,  
2494, 2500, 2508, 2517, 2524, 2538,  
2558, 2562, 2567, 2569, 2606, 2788,  
3147, 3848, 4079, 4088, 4093, 4098,  
4103, 4529, 4610, 4876, 4881, 4886,  
4891, 4907, 4912, 4917, 4922, 5075,  
5084, 5139, 8463, 8468, 8473, 8478,  
9168, 9314, 9830, 9838, 9898, 10192,  
10200, 10536, 12634, 13217, 14315,  
14350, 14355, 14360, 14365, 16297,  
16412, 16416, 16780, 16906, 16907,  
16908, 16909, 17028, 17046, 17075,  
17119, 17131, 17136, 17144, 17152,  
17173, 17179, 17251, 17264, 17265,  
17274, 17287, 17305, 17306, 17326,  
17339, 17343, 17365, 17393, 17406,  
17419, 17443, 17454, 17464, 17483,  
17496, 17509, 17512, 17524, 17547,  
17576, 17590, 17607, 17627, 17638,  
17644, 17654, 17700, 17707, 17738,  
17753, 17761, 17778, 17794, 17798,  
17807, 17844, 17853, 17862, 17867,  
17869, 17880, 17882, 17897, 17900,  
17907, 17918, 18004, 18052, 18070,  
18073, 18087, 18100, 18150, 18168,  
18239, 18251, 18280, 18282, 18286,  
18288, 18346, 18356, 18366, 18378,  
18494, 18511, 18521, 18676, 18677,  
18678, 18869, 18872, 18880, 18890,  
18898, 19910, 20441, 20463, 20618,  
20795, 21071, 21814, 21829, 21846,  
21883, 21900, 21942, 21961, 21974,  
22006, 22021, 22032, 22154, 22201,  
22241, 22277, 22457, 22557, 29579,  
29618, 29902, 29949, 29963, 29975,  
31436, 32198, 32200, 32202, 32204,  
32206, 32208, 32210, 32212, 32395,  
32400, 32405, 32410, 32426, 32444  
\exp\_after:wN . . . . . 33,  
35, 36, 318, 322, 340, 342, 399, 412,  
518, 616, 738, 760, 761, 763, 764,  
828, 829, 890, 891, 961, 983, 1050,  
1142, 1197, 1436, 1454, 1456, 1461,  
1463, 1575, 1577, 1631, 1655, 1673,  
1675, 1694, 1702, 1710, 1734, 1739,  
1746, 1775, 1779, 1784, 1795, 1811,  
1813, 1816, 1839, 1841, 1844, 1965,  
1967, 1976, 1996, 1998, 2037, 2101,  
2197, 2206, 2215, 2227, 2237, 2243,  
2250, 2252, 2264, 2265, 2277, 2278,  
2283, 2284, 2289, 2294, 2296, 2299,

7924, 7959, 7960, 7961, 8045, 8046,  
8048, 8049, 8057, 8058, 8085, 8086,  
8112, 8113, 8116, 8212, 8226, 8231,  
8234, 8235, 8242, 8243, 8259, 8260,  
8281, 8282, 8291, 8402, 8407, 8412,  
8435, 8437, 8565, 8566, 8567, 8592,  
8593, 8776, 8804, 8809, 8837, 8850,  
8860, 8887, 8889, 8890, 8898, 8915,  
8959, 9021, 9028, 9064, 9066, 9073,  
9166, 9184, 9189, 9193, 9315, 9575,  
9576, 9577, 9673, 9829, 9837, 9898,  
9970, 9985, 10007, 10021, 10082,  
10090, 10096, 10191, 10199, 10283,  
10284, 10287, 10288, 10536, 10537,  
10580, 10645, 10646, 10647, 10738,  
10739, 10981, 11000, 11048, 11156,  
11185, 11186, 11188, 11194, 11197,  
11240, 11243, 11279, 11281, 11291,  
11292, 11293, 11295, 11301, 11302,  
11304, 11311, 11312, 11314, 11364,  
11369, 11371, 11377, 11507, 11688,  
11689, 11690, 11919, 12128, 12129,  
12635, 12636, 12637, 12638, 12710,  
12711, 12754, 12900, 12918, 12958,  
13147, 13150, 13200, 13209, 13212,  
13215, 13216, 13218, 13258, 13324,  
13355, 13376, 13388, 13444, 13530,  
13531, 13532, 13947, 14051, 14270,  
14274, 14277, 14278, 14285, 14286,  
14310, 14315, 14326, 14329, 14436,  
14437, 14438, 14493, 14514, 14613,  
14977, 15005, 15018, 15173, 15181,  
15582, 15742, 15743, 15822, 15845,  
16012, 16013, 16014, 16032, 16040,  
16064, 16065, 16109, 16250, 16252,  
16253, 16271, 16272, 16273, 16283,  
16285, 16293, 16295, 16302, 16303,  
16304, 16305, 16306, 16307, 16312,  
16313, 16314, 16315, 16316, 16317,  
16318, 16361, 16374, 16377, 16388,  
16390, 16405, 16409, 16410, 16411,  
16414, 16415, 16479, 16481, 16508,  
16512, 16537, 16541, 16558, 16565,  
16567, 16635, 16643, 16660, 16669,  
16715, 16780, 16849, 16850, 16851,  
16911, 16921, 16940, 16946, 16965,  
16967, 16969, 16980, 16981, 16984,  
16995, 16999, 17006, 17007, 17018,  
17019, 17028, 17035, 17037, 17038,  
17046, 17075, 17093, 17094, 17097,  
17098, 17100, 17101, 17105, 17106,  
17108, 17109, 17118, 17119, 17124,  
17130, 17136, 17144, 17152, 17171,  
17172, 17175, 17176, 17178, 17185,  
17186, 17188, 17206, 17207, 17235,  
17238, 17243, 17244, 17249, 17250,  
17252, 17261, 17262, 17263, 17264,  
17267, 17268, 17269, 17272, 17287,  
17304, 17305, 17315, 17316, 17326,  
17338, 17342, 17355, 17357, 17365,  
17375, 17377, 17383, 17388, 17390,  
17392, 17398, 17399, 17403, 17405,  
17417, 17418, 17440, 17442, 17448,  
17451, 17453, 17457, 17462, 17467,  
17468, 17478, 17479, 17481, 17482,  
17485, 17489, 17494, 17508, 17511,  
17523, 17532, 17539, 17540, 17541,  
17542, 17544, 17546, 17557, 17558,  
17559, 17560, 17562, 17564, 17566,  
17567, 17568, 17574, 17575, 17585,  
17589, 17590, 17592, 17593, 17594,  
17599, 17605, 17606, 17617, 17619,  
17626, 17627, 17629, 17630, 17637,  
17643, 17653, 17721, 17734, 17735,  
17736, 17737, 17751, 17752, 17754,  
17759, 17760, 17775, 17777, 17794,  
17798, 17807, 17841, 17842, 17843,  
17849, 17850, 17851, 17852, 17858,  
17859, 17860, 17861, 17868, 17881,  
17889, 17895, 17896, 17898, 17899,  
17905, 17906, 17908, 17934, 17947,  
17969, 17970, 17972, 17973, 17980,  
17981, 17983, 17986, 17998, 17999,  
18000, 18002, 18003, 18004, 18013,  
18014, 18016, 18017, 18024, 18025,  
18027, 18030, 18045, 18046, 18047,  
18050, 18051, 18052, 18062, 18063,  
18064, 18067, 18068, 18069, 18072,  
18076, 18085, 18086, 18097, 18098,  
18099, 18102, 18104, 18105, 18106,  
18133, 18134, 18136, 18137, 18138,  
18148, 18149, 18150, 18152, 18153,  
18154, 18166, 18167, 18170, 18172,  
18173, 18174, 18194, 18195, 18196,  
18197, 18198, 18199, 18200, 18210,  
18211, 18213, 18214, 18215, 18221,  
18232, 18233, 18234, 18235, 18236,  
18237, 18238, 18239, 18244, 18245,  
18246, 18247, 18248, 18249, 18250,  
18266, 18267, 18269, 18270, 18277,  
18278, 18279, 18284, 18285, 18287,  
18303, 18318, 18327, 18337, 18343,  
18344, 18345, 18350, 18363, 18364,  
18365, 18371, 18493, 18510, 18520,  
18545, 18546, 18593, 18675, 18783,  
18785, 18824, 18826, 18829, 18864,  
18866, 18868, 18871, 18878, 18879,  
18882, 18883, 18888, 18889, 18896,

18897, 18932, 18933, 18934, 18936,  
18947, 18972, 18974, 18980, 18981,  
18985, 18988, 19010, 19012, 19025,  
19027, 19033, 19035, 19038, 19044,  
19046, 19048, 19049, 19050, 19052,  
19057, 19059, 19061, 19065, 19068,  
19074, 19075, 19079, 19081, 19082,  
19083, 19091, 19093, 19094, 19101,  
19107, 19114, 19115, 19120, 19121,  
19122, 19123, 19142, 19143, 19144,  
19150, 19151, 19152, 19157, 19159,  
19167, 19169, 19171, 19172, 19174,  
19185, 19187, 19189, 19190, 19195,  
19246, 19247, 19254, 19255, 19257,  
19259, 19261, 19264, 19267, 19269,  
19271, 19280, 19282, 19288, 19290,  
19292, 19293, 19294, 19300, 19302,  
19304, 19305, 19306, 19327, 19328,  
19331, 19339, 19341, 19345, 19346,  
19347, 19348, 19353, 19355, 19362,  
19365, 19368, 19371, 19380, 19383,  
19386, 19389, 19396, 19398, 19404,  
19412, 19414, 19416, 19433, 19435,  
19442, 19444, 19447, 19453, 19455,  
19457, 19458, 19459, 19461, 19475,  
19476, 19479, 19497, 19499, 19501,  
19513, 19516, 19519, 19522, 19525,  
19528, 19531, 19534, 19538, 19550,  
19554, 19558, 19561, 19576, 19582,  
19584, 19586, 19596, 19620, 19623,  
19635, 19637, 19641, 19642, 19643,  
19645, 19646, 19648, 19655, 19663,  
19664, 19670, 19671, 19677, 19680,  
19681, 19682, 19683, 19691, 19733,  
19738, 19740, 19747, 19750, 19753,  
19756, 19759, 19762, 19770, 19771,  
19783, 19791, 19793, 19803, 19805,  
19812, 19821, 19823, 19826, 19829,  
19832, 19835, 19848, 19850, 19858,  
19860, 19868, 19870, 19880, 19883,  
19886, 19893, 19908, 19909, 19926,  
19928, 19929, 19986, 19999, 20001,  
20007, 20020, 20022, 20024, 20048,  
20062, 20064, 20071, 20073, 20114,  
20115, 20116, 20118, 20119, 20120,  
20122, 20123, 20129, 20131, 20132,  
20138, 20140, 20141, 20142, 20143,  
20155, 20161, 20163, 20200, 20207,  
20214, 20234, 20235, 20237, 20239,  
20241, 20254, 20259, 20260, 20261,  
20262, 20263, 20267, 20272, 20274,  
20280, 20286, 20288, 20289, 20295,  
20296, 20297, 20298, 20299, 20300,  
20301, 20302, 20307, 20309, 20311,  
20313, 20315, 20320, 20322, 20324,  
20326, 20328, 20330, 20348, 20352,  
20360, 20361, 20366, 20368, 20377,  
20380, 20381, 20382, 20384, 20385,  
20386, 20394, 20400, 20412, 20415,  
20416, 20418, 20419, 20443, 20444,  
20447, 20449, 20465, 20469, 20470,  
20471, 20487, 20493, 20559, 20560,  
20561, 20568, 20570, 20571, 20576,  
20578, 20579, 20588, 20589, 20591,  
20594, 20597, 20613, 20617, 20618,  
20622, 20624, 20660, 20666, 20667,  
20669, 20671, 20672, 20674, 20675,  
20685, 20686, 20689, 20690, 20691,  
20693, 20694, 20695, 20712, 20713,  
20715, 20716, 20722, 20724, 20727,  
20730, 20733, 20736, 20744, 20747,  
20749, 20752, 20759, 20763, 20771,  
20772, 20775, 20777, 20779, 20784,  
20785, 20791, 20796, 20797, 20805,  
20806, 20807, 20808, 20851, 20873,  
20874, 20877, 20878, 20887, 20888,  
20889, 20893, 20900, 20901, 20902,  
21043, 21044, 21045, 21047, 21065,  
21066, 21067, 21068, 21069, 21070,  
21077, 21086, 21093, 21094, 21318,  
21319, 21325, 21326, 21329, 21334,  
21337, 21340, 21343, 21346, 21349,  
21352, 21355, 21371, 21372, 21382,  
21391, 21399, 21400, 21402, 21403,  
21408, 21409, 21418, 21425, 21434,  
21435, 21448, 21450, 21478, 21479,  
21488, 21491, 21516, 21522, 21523,  
21565, 21566, 21568, 21582, 21583,  
21591, 21602, 21636, 21639, 21649,  
21650, 21653, 21655, 21661, 21675,  
21677, 21718, 21721, 21741, 21814,  
21824, 21828, 21846, 21849, 21870,  
21871, 21878, 21882, 21900, 21903,  
21932, 21933, 21939, 21940, 21941,  
21948, 21956, 21960, 21974, 21977,  
21993, 21994, 22001, 22005, 22016,  
22020, 22032, 22037, 22038, 22039,  
22045, 22047, 22050, 22052, 22114,  
22143, 22153, 22160, 22165, 22166,  
22176, 22203, 22214, 22216, 22218,  
22220, 22225, 22226, 22228, 22239,  
22240, 22260, 22266, 22267, 22269,  
22272, 22277, 22283, 22284, 22314,  
22316, 22319, 22322, 22324, 22332,  
22334, 22338, 22342, 22347, 22352,  
22363, 22427, 22428, 22452, 22453,  
22454, 22455, 22456, 22464, 22475,  
22481, 22507, 22508, 22509, 22516,

22517, 22524, 22525, 22533, 22534,  
 22538, 22539, 22540, 22545, 22550,  
 22556, 22559, 22560, 22561, 22562,  
 22563, 22767, 22768, 22980, 23066,  
 23067, 23109, 23128, 23129, 23144,  
 23145, 23146, 23372, 23378, 23380,  
 23391, 23451, 23452, 23453, 23454,  
 23460, 23461, 23477, 23495, 23497,  
 23503, 23504, 23535, 23537, 23539,  
 23569, 23584, 23586, 23588, 23613,  
 23623, 23631, 23641, 23651, 23653,  
 23655, 23690, 23714, 23723, 23726,  
 23727, 23729, 23730, 23738, 23739,  
 23753, 23803, 23816, 23817, 23822,  
 23823, 23826, 23829, 23874, 23881,  
 23888, 23894, 23901, 23909, 23945,  
 23947, 23956, 24079, 24082, 24123,  
 24143, 24145, 24146, 24153, 24156,  
 24157, 24181, 24200, 24209, 24321,  
 24323, 24329, 24332, 24334, 24341,  
 24344, 24346, 24353, 24355, 24361,  
 24364, 24367, 24682, 24755, 24767,  
 24899, 24902, 24912, 24914, 25097,  
 25105, 25180, 25230, 25444, 25726,  
 25867, 25890, 25941, 25967, 26031,  
 26032, 26040, 26043, 26204, 26205,  
 26208, 26209, 26217, 26219, 26220,  
 26229, 26243, 26246, 26316, 26565,  
 26597, 26599, 28952, 29257, 29303,  
 29305, 29419, 29421, 29428, 29429,  
 29432, 29455, 29457, 29463, 29475,  
 29483, 29485, 29577, 29616, 29686,  
 29687, 29730, 29731, 29778, 29831,  
 29857, 29900, 29947, 29961, 29973,  
 30859, 30861, 30865, 30909, 30911,  
 30925, 30927, 31153, 31155, 31209,  
 31211, 31213, 31228, 31230, 31232,  
 31351, 31353, 31422, 31434, 31482,  
 31483, 31544, 31545, 31546, 31553,  
 31599, 31714, 31716, 31789, 31791,  
 32238, 32248, 32253, 32256, 32395,  
 32400, 32405, 32410, 32424, 32427,  
 32442, 32444, 32445, 32452, 32457,  
 32459, 32462, 32476, 32496, 32497,  
 32504, 32505, 32546, 32560, 32591  
**\exp\_args:cc** . . . . . 29, 1453, 2307  
**\exp\_args:Nc** 27, 29, 335, 1453, 1457,  
 1465, 1679, 1692, 1700, 1708, 1900,  
 1919, 1945, 1950, 1957, 2016, 2028,  
 2100, 2133, 2134, 2135, 2136, 2158,  
 2162, 2307, 2877, 3888, 4138, 4732,  
 9041, 12651, 12687, 12886, 15901,  
 16929, 17168, 18056, 18080, 18110,  
 18112, 18114, 18116, 18118, 18120,  
 18122, 18124, 18142, 18158, 18159,  
 18178, 18180, 22677, 22678, 22679,  
 22707, 23673, 25052, 28032, 28063,  
 32038, 32380, 32466, 32471, 32479  
**\exp\_args:Ncc** . . . . . 31,  
 1947, 1951, 1959, 2141, 2142, 2143,  
 2144, 2307, 3471, 3511, 5549, 5749  
**\exp\_args:Nccc** . . . . . 32, 2307  
**\exp\_args:Ncco** . . . . . 32, 2406  
**\exp\_args:Ncxx** . . . . . 32, 3255  
**\exp\_args:Ncf** . . . . . 31, 2347  
**\exp\_args:NcNc** . . . . . 32, 2406  
**\exp\_args:NcNo** . . . . . 32, 2406  
**\exp\_args:Ncno** . . . . . 32, 3255  
**\exp\_args:NcnV** . . . . . 32, 3255  
**\exp\_args:Ncnx** . . . . . 32, 3255  
**\exp\_args:Nco** . . . . . 31, 345, 2347  
**\exp\_args:Ncoo** . . . . . 32, 3255  
**\exp\_args:NcV** . . . . . 31, 2347  
**\exp\_args:Ncv** . . . . . 31, 2347  
**\exp\_args:Ncvv** . . . . . 32, 3255  
**\exp\_args:Ncx** . . . . . 31, 3229, 12643  
**\exp\_args:Ne** . . . . . 30, 341,  
 1518, 2260, 2323, 2555, 4686, 6638,  
 6653, 10748, 10752, 12649, 12685,  
 13408, 13410, 13489, 13549, 13571,  
 13699, 13708, 13727, 13737, 13747,  
 13760, 13963, 29747, 29810, 29820,  
 29903, 30064, 30338, 30597, 30612,  
 30675, 31437, 31586, 31706, 31783  
**\exp\_args:Nee** . 31, 3229, 13838, 13993  
**\exp\_args:Neee** . . . . . 32, 3255, 13713  
**\exp\_args:Nf** . . . . .  
 . . . . . 30, 2004, 2335, 2601, 2667,  
 2702, 2716, 3837, 4552, 4553, 4569,  
 4587, 4595, 4599, 4623, 4629, 4639,  
 5052, 5054, 5113, 5115, 5131, 5448,  
 5453, 5804, 5832, 6354, 6355, 6519,  
 6530, 7927, 7928, 7944, 8464, 8469,  
 8474, 8479, 8650, 8719, 8721, 8739,  
 8748, 8759, 8768, 8906, 8923, 9161,  
 10272, 10323, 10337, 10358, 10369,  
 10418, 10488, 10494, 10500, 10506,  
 10658, 10778, 10862, 11945, 13250,  
 13308, 14351, 14356, 14361, 14366,  
 16083, 18736, 21810, 22376, 22619,  
 22824, 25584, 32166, 32168, 32589  
**\exp\_args:Nff** . . 31, 3229, 4593, 16578  
**\exp\_args:Nffo** . . . . . 32, 3255  
**\exp\_args:Nfo** . . . . . 31, 3229  
**\exp\_args>NNc** . . . . .  
 . . . . . 31, 314, 1946, 1949, 1958,  
 2030, 2137, 2138, 2139, 2140, 2175,  
 2178, 2307, 3147, 8038, 8607, 8618,

12754, 12869, 12870, 12958, 14467,  
 14474, 18746, 18753, 22404, 22709  
`\exp_args:Nnc` ..... 31, 3229  
`\exp_args>NNcf` ..... 32, 3255  
`\exp_args:NNe` ..... 31, 2347  
`\exp_args:Nne` ..... 31, 3229  
`\exp_args>NNf` ..... 31, 2347, 12753, 12957, 13134,  
 14460, 16158, 16163, 21037, 21038  
`\exp_args:Nnf` ..... 31, 3229, 3815, 10801, 15899  
`\exp_args:Nnff` ..... 32, 3255, 10807  
`\exp_args:Nnncc` ..... 32, 3255  
`\exp_args:Nnnf` ..... 32, 3255  
`\exp_args:NNNNo` ..... 32, 2318, 24416,  
 25304, 25361, 26475, 26559, 32963  
`\exp_args:NNnno` ..... 32, 3255  
`\exp_args:Nnno` ..... 32, 3255  
`\exp_args:NNNV` ..... 32, 2406  
`\exp_args:NNNv` ..... 32, 2406, 12872  
`\exp_args:NNnV` ..... 32, 3255  
`\exp_args:NNNx` ..... 32, 946, 3255, 24424, 24895  
`\exp_args:NNnx` ..... 32, 3255  
`\exp_args:Nnnx` ..... 32, 3255  
`\exp_args:NNo` ..... 25, 31, 2318,  
 2610, 3773, 8638, 11501, 15740, 26125  
`\exp_args:Nno` ..... 31, 3229, 3517,  
 3800, 3861, 3969, 3978, 4317, 4332,  
 4423, 4457, 9583, 10227, 12843,  
 13539, 14318, 16620, 16628, 16637,  
 16654, 16662, 16690, 17194, 17198  
`\exp_args:NNoo` ..... 32, 3255  
`\exp_args:NNox` ..... 32, 3255  
`\exp_args:Nnox` ..... 32, 3255  
`\exp_args:NNV` ..... 31, 2347  
`\exp_args:NNv` ..... 31, 2347  
`\exp_args:NnV` ..... 31, 3229  
`\exp_args:Nnv` ..... 31, 3229  
`\exp_args:NNVV` ..... 32, 3255  
`\exp_args:NNx` ..... 31,  
 2183, 3229, 14020, 14034, 14100, 32658  
`\exp_args:Nnx` ..... 31, 3229, 4479  
`\exp_args:No` ..... 27,  
 30, 1197, 2167, 2172, 2318, 2610,  
 2614, 2644, 2682, 2745, 2762, 2800,  
 3113, 3136, 3143, 3215, 3232, 3788,  
 4027, 4028, 4029, 4056, 4057, 4058,  
 4059, 4060, 4126, 4145, 4154, 4169,  
 4245, 4248, 4250, 4331, 4340, 4545,  
 4547, 4571, 4578, 4580, 4637, 4646,  
 4944, 4971, 4976, 4990, 5048, 5059,  
 5109, 5120, 5187, 5206, 5244, 5259,  
 5677, 5730, 6016, 7621, 8638, 8725,  
 8731, 9558, 9573, 10089, 10101,  
 10103, 10138, 10143, 10352, 10356,  
 12122, 12902, 13016, 13046, 13142,  
 13527, 13591, 14619, 15288, 15322,  
 15350, 15372, 15443, 15452, 15458,  
 15493, 15500, 15555, 15740, 15773,  
 23677, 23700, 23757, 23759, 24492,  
 24549, 24569, 25205, 25558, 26172,  
 26215, 26620, 26650, 27125, 29450,  
 32356, 32481, 32485, 32529, 32586  
`\exp_args:Noc` ..... 31, 3229  
`\exp_args:Nof` ..... 31, 3229, 3830  
`\exp_args:Noo` ..... 31, 3229, 24591, 25571  
`\exp_args:Noof` ..... 32, 3255  
`\exp_args:Nooo` ..... 32, 3255  
`\exp_args:Noox` ..... 32, 3255  
`\exp_args:Nox` ..... 31, 3229  
`\exp_args:NV` 30, 2335, 13284, 13360,  
 13515, 14080, 14085, 15286, 15320,  
 15348, 15370, 22722, 23983, 29998  
`\exp_args:Nv` ... 30, 2335, 29845, 31579  
`\exp_args:NVo` ..... 31, 3229  
`\exp_args:NVV` ..... 31, 2347, 13195  
`\exp_args:Nx` ..... 31,  
 1974, 2441, 3156, 5566, 9043,  
 9143, 10628, 11892, 13052, 15290,  
 15324, 15352, 15374, 18460, 22719,  
 22732, 24724, 24725, 25108, 25978  
`\exp_args:Nxo` ..... 31, 3229  
`\exp_args:Nxx` ..... 31, 3229  
`\exp_args_generate:n` 266, 3213, 12646  
`\exp_end:` .....  
 ... 36, 36, 319, 322, 325, 342, 343,  
 350, 351, 399, 406, 407, 423, 537,  
 628, 761, 790, 1196, 1197, 1440,  
 1680, 1693, 1701, 1709, 2296, 2305,  
 2569, 2599, 2789, 3147, 3864, 4118,  
 4499, 4618, 4934, 5108, 8490, 9346,  
 9349, 9350, 9351, 9352, 9353, 9354,  
 9355, 9356, 9357, 9359, 9825, 10567,  
 10577, 10580, 10586, 10594, 10641,  
 10646, 12638, 14377, 16911, 17874,  
 20465, 22203, 22205, 22277, 29598,  
 29924, 31455, 32231, 32386, 32415  
`\exp_end_continue_f:nw` ..... 37, 2569  
`\exp_end_continue_f:w` .....  
 ... 36, 37, 341, 763,  
 764, 2265, 2336, 2373, 2397, 2468,  
 2487, 2500, 2524, 2538, 2558, 2569,  
 4079, 9168, 9898, 12563, 13217,  
 14315, 16297, 16412, 16416, 17028,  
 17046, 17067, 17131, 17136, 17144,  
 17152, 17173, 17251, 17287, 17295,

17326, 17700, 17707, 17753, 17800,  
 17807, 17844, 17888, 17894, 17897,  
 17907, 17918, 18087, 18280, 18282,  
 18286, 18288, 18346, 18356, 18366,  
 18378, 18494, 18511, 18521, 18676,  
 18677, 18678, 18869, 18880, 18890,  
 18898, 19910, 20618, 20795, 21071,  
 21814, 21829, 21846, 21883, 21900,  
 21942, 21961, 21974, 22006, 22021,  
 22032, 22154, 22241, 22457, 22557  
`\exp_last_two_unbraced:Nnn` . . . . .  
     . . . . . 33, 2541, 28106, 28632, 28636  
`\exp_last_unbraced:Nco` 33, 2475, 10210  
`\exp_last_unbraced:NcV` . . . . . 33, 2475  
`\exp_last_unbraced:Ne` 33, 2475, 22085  
`\exp_last_unbraced:Nf` . . . . .  
     . . . . . 33, 2475, 3400, 3421, 3528, 3550,  
     5900, 6185, 6372, 6544, 8737, 8757,  
     16099, 16573, 18485, 18962, 24134  
`\exp_last_unbraced:Nfo` 33, 2475, 22407  
`\exp_last_unbraced:NNf` . . . . . 33, 2475  
`\exp_last_unbraced:NNNf` 33, 2475, 9106  
`\exp_last_unbraced:NNNNf` . . . . .  
     . . . . . . . . . . . 33, 2475, 9111  
`\exp_last_unbraced:NNNNo` . . . . .  
     . . . . . 33, 2475, 2893, 2897, 3060,  
     5302, 6017, 15028, 16365, 16383, 29491  
`\exp_last_unbraced:NNNo` . . . . . 33, 2475  
`\exp_last_unbraced:NnNo` . . . . . 33, 2475  
`\exp_last_unbraced:NNNV` . . . . . 33, 2475  
`\exp_last_unbraced:NNo` . . . . .  
     . . . . . 33, 2475, 4507, 10177,  
     13049, 13457, 28603, 29624, 31470  
`\exp_last_unbraced:Nno` . . . . .  
     . . . . . 33, 2475, 8004, 11719  
`\exp_last_unbraced:NNV` . . . . . 33, 2475  
`\exp_last_unbraced:No` . . . . .  
     . . . . . 33, 2475, 28777, 28782, 28850, 28856  
`\exp_last_unbraced:Noo` . . . . .  
     . . . . . 33, 2475, 11559, 11653  
`\exp_last_unbraced:NV` . . . . . 33, 2475  
`\exp_last_unbraced:Nv` 33, 2475, 10649  
`\exp_last_unbraced:Nx` . . . . . 33, 2475  
`\exp_not:N` . . . . . 34, 34, 89, 164, 218,  
     342, 349, 354, 402, 403, 403, 404,  
     404, 583, 591, 768, 961, 970, 1043,  
     1047, 1436, 1635, 1721, 1724, 2036,  
     2037, 2100, 2101, 2243, 2289, 2442,  
     2546, 2546, 2624, 2628, 2670, 2723,  
     2743, 2753, 2766, 2870, 2872, 2873,  
     2880, 2881, 2882, 2883, 2884, 2885,  
     2891, 2917, 2926, 2981, 2982, 3042,  
     3048, 3050, 3056, 3117, 3134, 3136,  
     3164, 3912, 4347, 4365, 4391, 4398,  
     4409, 4410, 4482, 4485, 4486, 4760,  
     4761, 4962, 4963, 5794, 5795, 5822,  
     5823, 5824, 6660, 6661, 6662, 6664,  
     6665, 6667, 7620, 7622, 8020, 8067,  
     8623, 8886, 10152, 10262, 10265,  
     10273, 10274, 10524, 10598, 10602,  
     10631, 10816, 10819, 10822, 10825,  
     10828, 10831, 10834, 10900, 10904,  
     10909, 10914, 10919, 10926, 10933,  
     10938, 10943, 10948, 10953, 10958,  
     10968, 10973, 10978, 10981, 10982,  
     10985, 10995, 11000, 11015, 11034,  
     11039, 11040, 11041, 11042, 11043,  
     11044, 11046, 11048, 11049, 11050,  
     11054, 11055, 11058, 11059, 11150,  
     11153, 11154, 11156, 11157, 11161,  
     11164, 11165, 11167, 11170, 11253,  
     11259, 11290, 11293, 11300, 11301,  
     11310, 11311, 11325, 11326, 11341,  
     11346, 11351, 11360, 11361, 11608,  
     11631, 11991, 11993, 11995, 11997,  
     12002, 12003, 12008, 12010, 12012,  
     12291, 12293, 12295, 12297, 12302,  
     12303, 12308, 12310, 12312, 12778,  
     12779, 12783, 13453, 13461, 13524,  
     13527, 13528, 13530, 13531, 13532,  
     13533, 13536, 13537, 13598, 13600,  
     13918, 13919, 13920, 13921, 13922,  
     13925, 13926, 14479, 14519, 15036,  
     15038, 15052, 15054, 15108, 15109,  
     15121, 15185, 15186, 15253, 15254,  
     15425, 15426, 15427, 15428, 15429,  
     15432, 15434, 15436, 15437, 15476,  
     15477, 15478, 15479, 15482, 15484,  
     15486, 15487, 15518, 15519, 15520,  
     15521, 15524, 15526, 15528, 15529,  
     15537, 15538, 15539, 15540, 15541,  
     15544, 15546, 15548, 15549, 16361,  
     16362, 17092, 17093, 17188, 17189,  
     17190, 17191, 17297, 17337, 17341,  
     17363, 17456, 17488, 17573, 17587,  
     17604, 17615, 17625, 17662, 17665,  
     17771, 17772, 17774, 17775, 17776,  
     17777, 17778, 17779, 17782, 17784,  
     17786, 17965, 17967, 18009, 18011,  
     18132, 18147, 18758, 18915, 20924,  
     20925, 20926, 20930, 20931, 20932,  
     22772, 23439, 23442, 23594, 23595,  
     23596, 23597, 23598, 23599, 23600,  
     23601, 23602, 23603, 23604, 23605,  
     23606, 23607, 23608, 23609, 23612,  
     23613, 23614, 24110, 24112, 24114,  
     24116, 24118, 24120, 24480, 24482,  
     24675, 24677, 24688, 24692, 24859,

25313, 25972, 26186, 26308, 26321,  
 26684, 28725, 28726, 28739, 28746,  
 29257, 29259, 29413, 29414, 29417,  
 29428, 29496, 29499, 29502, 29505,  
 29508, 29511, 29514, 29631, 29632,  
 29633, 29635, 29637, 29641, 29645,  
 29646, 29798, 29799, 29804, 29805,  
 29820, 29983, 30048, 30204, 30209,  
 30211, 30212, 30213, 30215, 30216,  
 30218, 30313, 30315, 30318, 30823,  
 30829, 30831, 30832, 30836, 30837,  
 30839, 30841, 30860, 30862, 30866,  
 30910, 30912, 30926, 30928, 31154,  
 31156, 31210, 31212, 31214, 31229,  
 31231, 31233, 31352, 31354, 31715,  
 31717, 31790, 31792, 32171, 32173,  
 32177, 32179, 32184, 32186, 32290,  
 32542, 32543, 32636, 32644, 32660,  
 32663, 32664, 32677, 32680, 32897  
  
`\exp_stop_f`: . . . . . 35, 36, 101,  
 341, 408, 489, 503, 646, 730, 742,  
 808, 809, 897, 925, 957, 961, 2262,  
 2730, 2733, 2748, 2756, 2811, 4605,  
 4614, 4663, 4664, 4670, 4842, 4849,  
 4856, 5090, 5106, 5145, 5146, 5152,  
 5164, 5180, 5181, 5393, 5401, 5608,  
 5609, 5610, 5615, 5616, 5658, 6029,  
 6091, 6095, 6125, 6128, 6144, 6148,  
 6169, 6247, 6249, 6269, 6270, 6287,  
 6289, 6343, 6346, 6347, 6466, 6471,  
 6607, 6612, 7654, 8214, 8228, 8238,  
 8246, 8429, 8434, 8588, 9710, 10414,  
 10416, 10484, 10486, 10490, 10492,  
 10496, 10498, 10502, 10504, 10542,  
 10543, 10550, 10551, 10552, 10553,  
 10558, 10559, 10576, 10581, 10589,  
 10649, 10663, 10664, 10670, 11360,  
 11361, 11362, 11363, 12754, 12958,  
 13205, 13219, 13231, 13867, 14324,  
 14495, 15992, 15995, 16167, 16171,  
 16200, 16405, 16520, 16535, 16560,  
 16780, 16784, 16788, 16790, 16794,  
 16798, 16806, 16811, 16824, 16831,  
 16844, 16855, 16856, 16867, 16868,  
 16877, 16880, 16891, 16933, 16997,  
 17002, 17074, 17104, 17260, 17303,  
 17354, 17374, 17401, 17415, 17450,  
 17477, 17486, 17505, 17521, 17537,  
 17555, 17616, 17635, 17651, 17666,  
 17680, 17889, 17968, 17979, 18012,  
 18023, 18261, 18265, 18559, 18565,  
 18567, 18582, 18591, 18599, 18607,  
 18608, 18805, 18925, 18931, 18946,  
 18983, 19056, 19078, 19132, 19133,  
 19141, 19478, 19496, 19549, 19553,  
 19557, 19575, 19610, 19611, 19612,  
 19613, 19614, 19640, 19652, 19668,  
 19685, 19963, 19964, 20061, 20154,  
 20189, 20202, 20207, 20216, 20218,  
 20345, 20374, 20379, 20409, 20446,  
 20486, 20562, 20612, 20658, 20659,  
 20664, 20670, 20688, 20711, 20743,  
 20746, 20793, 20813, 20819, 20834,  
 20846, 20884, 20905, 20945, 20960,  
 20975, 20990, 21005, 21020, 21048,  
 21092, 21358, 21368, 21398, 21550,

21552, 21589, 21601, 21633, 21674,  
 21683, 21698, 21717, 21750, 21763,  
 21847, 21901, 21949, 21952, 21975,  
 22184, 22188, 22195, 22196, 22251,  
 22252, 22253, 22262, 22272, 22290,  
 22359, 22362, 22365, 22380, 22433,  
 22437, 22479, 22551, 22558, 22979,  
 23238, 23405, 23414, 23415, 23449,  
 23519, 23527, 23550, 23552, 23553,  
 23557, 23574, 23625, 23628, 23644,  
 23650, 23722, 23725, 23916, 23917,  
 23918, 23924, 23944, 24196, 24216,  
 24217, 24221, 24225, 24226, 24229,  
 24230, 24238, 24239, 24242, 24246,  
 24247, 24250, 24309, 24404, 24648,  
 24653, 24667, 24668, 24681, 24752,  
 24753, 24792, 24892, 25069, 25228,  
 25496, 25514, 25540, 25550, 25602,  
 25615, 25626, 25642, 25693, 25910,  
 26038, 26042, 26106, 26169, 26182,  
 26202, 26207, 26213, 26259, 26312,  
 26329, 26352, 26570, 26575, 26596,  
 26674, 26686, 26691, 28241, 29482,  
 30445, 30448, 30449, 30452, 30465,  
 30468, 30471, 30474, 30477, 30480,  
 30494, 30497, 30500, 30503, 30506,  
 30518, 30521, 30524, 30527, 32495,  
 32503, 32542, 32543, 32544, 32545  
**exp internal commands:**  
 \\_\_exp\_arg\_last\_unbraced:nn . . . . . 2443  
 \\_\_exp\_arg\_next:Nnn . . . . . 2243, 2250  
 \\_\_exp\_arg\_next:nnn . . . . .  
     342, 2243, 2252, 2260, 2264, 2277, 2283  
 \\_\_exp\_e:N . . . . . 2585, 2615  
 \\_\_exp\_e:nn . . . . . 344, 351, 2332,  
     2463, 2581, 2601, 2606, 2614, 2642,  
     2644, 2689, 2690, 2695, 2762, 2780  
 \\_\_exp\_e:Nnn . . . . . 352, 2615  
 \\_\_exp\_e\_end:nn . . . . . 351, 2581, 2714  
 \\_\_exp\_e\_expandable:Nnn . . . . . 352, 2615  
 \\_\_exp\_e\_group:n . . . . . 2588, 2602  
 \\_\_exp\_e\_if\_toks\_register:N . . . . . 2826  
 \\_\_exp\_e\_if\_toks\_register:NTF . . . . .  
     2777, 2826  
 \\_\_exp\_e\_noexpand:Nnn . . . . . 2635, 2670, 2692  
 \\_\_exp\_e\_primitive:Nnn . . . . . 2637, 2645  
 \\_\_exp\_e\_primitive\_aux:NNnn . . . . . 2645  
 \\_\_exp\_e\_primitive\_aux:NNw . . . . . 2645  
 \\_\_exp\_e\_primitive\_other:NNnn . . . . . 2645  
 \\_\_exp\_e\_primitive\_other\_-  
     aux:nNNnn . . . . . 2645  
 \\_\_exp\_e\_protected:Nnn . . . . . 352, 2615  
 \\_\_exp\_e\_put:nn . . . . .  
     351, 353, 355, 2602, 2695, 2707, 2794  
 \\_\_exp\_e\_put:nnn . . . . . 356, 2602, 2800  
 \\_\_exp\_e\_space:nn . . . . . 2592, 2600  
 \\_\_exp\_e\_the:N . . . . . 2758  
 \\_\_exp\_e\_the:Nnn . . . . . 2636, 2671, 2758  
 \\_\_exp\_e\_the\_errhelp: . . . . . 2826  
 \\_\_exp\_e\_the\_everycr: . . . . . 2826  
 \\_\_exp\_e\_the\_everydisplay: . . . . . 2826  
 \\_\_exp\_e\_the\_everyeof: . . . . . 2826  
 \\_\_exp\_e\_the\_everyhbox: . . . . . 2826  
 \\_\_exp\_e\_the\_everyjob: . . . . . 2826  
 \\_\_exp\_e\_the\_everymath: . . . . . 2826  
 \\_\_exp\_e\_the\_everypar: . . . . . 2826  
 \\_\_exp\_e\_the\_everyvbox: . . . . . 2826  
 \\_\_exp\_e\_the\_output: . . . . . 2826  
 \\_\_exp\_e\_the\_pdfpageattr: . . . . . 2826  
 \\_\_exp\_e\_the\_pdffpageresources: . . . . . 2826  
 \\_\_exp\_e\_the\_pdffpagesattr: . . . . . 2826  
 \\_\_exp\_e\_the\_pdfpkmode: . . . . . 2826  
 \\_\_exp\_e\_the\_toks:N . . . . . 356, 2798  
 \\_\_exp\_e\_the\_toks:n . . . . . 356, 2774, 2798  
 \\_\_exp\_e\_the\_toks:wnn . . . . . 356, 2773, 2798  
 \\_\_exp\_e\_the\_toks\_reg:N . . . . . 2758  
 \\_\_exp\_e\_the\_XeTeXinterchartoks:  
     . . . . . 2826  
 \\_\_exp\_e\_unexpanded:N . . . . . 2697  
 \\_\_exp\_e\_unexpanded:nN . . . . . 354, 2697  
 \\_\_exp\_e\_unexpanded:nn . . . . . 2697  
 \\_\_exp\_e\_unexpanded:Nnn . . . . .  
     . . . . . 2634, 2669, 2697  
 \\_\_exp\_eval\_error\_msg:w . . . . . 2287  
 \\_\_exp\_eval\_register:N . . . . .  
     . . . . . 2278, 2284, 2287, 2340,  
     2345, 2351, 2357, 2385, 2391, 2403,  
     2404, 2411, 2418, 2449, 2454, 2477,  
     2479, 2494, 2508, 2517, 2562, 2567  
 \l\_\_exp\_internal\_t1 . . . . .  
     . . . . . 316, 1508, 1512, 1513,  
     2243, 2243, 2271, 2273, 2472, 2473  
 \\_\_exp\_last\_two\_unbraced:nnN . . . . . 2541  
\expandafter . . . . . 4,  
     20, 21, 24, 25, 29, 30, 34, 35, 44,  
     45, 68, 70, 71, 72, 83, 90, 111, 119, 272  
\expanded . . . . . 316, 820  
\expandglyphsinfont . . . . . 915  
\ExplFileVersion 7, 14045, 14060, 14074, 14078  
\ExplFileDescription . . . . . 7, 14044, 14057  
\ExplFileExtension . . . . . 14047, 14062, 14071  
\ExplFileName . . . . . 7, 14046, 14061, 14070  
\ExplFileVersion . . . . . 7, 14048, 14063, 14072  
\explicitdiscretionary . . . . . 821  
\explicithyphenpenalty . . . . . 819  
\ExplLoaderFileDialog . . . . . 32586, 32592  
\ExplSyntaxOff . . . . . 4,  
     7, 7, 119, 121, 151, 165, 278, 278, 305

- \ExplSyntaxOn . . . . . 4,  
     7, 7, 119, 147, 278, 278, 305, 384, 570
- F**
- fact . . . . . 213  
 false . . . . . 218  
 \fam . . . . . 273  
 \fi . . . . . 18, 23, 32, 48, 49, 50, 75, 78,  
     80, 81, 82, 85, 86, 97, 105, 117, 118, 274  
 fi commands:  
     \fif: . . . . . 23, 101,  
         101, 102, 106, 113, 113, 165, 184,  
         249, 249, 249, 319, 320, 322, 325,  
         326, 351, 385, 389, 423, 424, 427,  
         509, 518, 538, 574, 655, 742, 767,  
         783, 814, 961, 1417, 1464, 1632,  
         1640, 1648, 1656, 1676, 1681, 1694,  
         1702, 1710, 1712, 1735, 1740, 1747,  
         1774, 1779, 1805, 1806, 1814, 1820,  
         1833, 1834, 1842, 1848, 1968, 1989,  
         1999, 2013, 2070, 2131, 2228, 2238,  
         2292, 2295, 2302, 2303, 2576, 2583,  
         2593, 2606, 2619, 2624, 2627, 2628,  
         2629, 2630, 2638, 2647, 2663, 2727,  
         2755, 2761, 2770, 2775, 2781, 2784,  
         2788, 2796, 2806, 2815, 2819, 2823,  
         2891, 2907, 2914, 2923, 2937, 2938,  
         2943, 2944, 2945, 2963, 2964, 2965,  
         2966, 2967, 2968, 2969, 2970, 2971,  
         2979, 2998, 3000, 3030, 3031, 3032,  
         3079, 3109, 3181, 3192, 3202, 3300,  
         3308, 3332, 3346, 3354, 3365, 3375,  
         3395, 3425, 3426, 3498, 3521, 3540,  
         3543, 3782, 3789, 3923, 3924, 3951,  
         3961, 3976, 3985, 3995, 4011, 4025,  
         4035, 4039, 4069, 4084, 4298, 4303,  
         4310, 4315, 4322, 4325, 4351, 4369,  
         4387, 4395, 4405, 4415, 4421, 4428,  
         4439, 4444, 4446, 4450, 4455, 4459,  
         4460, 4607, 4619, 4668, 4674, 4675,  
         4843, 4850, 4856, 4951, 5019, 5023,  
         5024, 5042, 5095, 5108, 5150, 5156,  
         5157, 5168, 5181, 5182, 5203, 5241,  
         5267, 5370, 5378, 5386, 5397, 5414,  
         5416, 5560, 5612, 5613, 5618, 5621,  
         5622, 5662, 5715, 5815, 6031, 6064,  
         6100, 6101, 6132, 6133, 6153, 6154,  
         6172, 6255, 6259, 6269, 6279, 6295,  
         6299, 6302, 6307, 6309, 6352, 6356,  
         6357, 6448, 6453, 6464, 6470, 6482,  
         6485, 6487, 6491, 6605, 6619, 6620,  
         7695, 7698, 7735, 7796, 7813, 7823,  
         7877, 7882, 8221, 8222, 8231, 8254,  
         8271, 8272, 8274, 8291, 8292, 8333,
- 8387, 8395, 8422, 8430, 8436, 8459,  
 8497, 8505, 8590, 8805, 8838, 8886,  
 8891, 9007, 9032, 9058, 9067, 9127,  
 9159, 9177, 9199, 9219, 9235, 9245,  
 9261, 9271, 9361, 9363, 9365, 9367,  
 9369, 9371, 9571, 9579, 9971, 9986,  
 10009, 10023, 10547, 10550, 10551,  
 10552, 10553, 10558, 10559, 10564,  
 10565, 10566, 10593, 10602, 10644,  
 10652, 10654, 10698, 10699, 10702,  
 10838, 10839, 10840, 10841, 10842,  
 10843, 10844, 10905, 10910, 10915,  
 10920, 10927, 10934, 10939, 10944,  
 10949, 10954, 10959, 10964, 10969,  
 10974, 10996, 11007, 11008, 11058,  
 11059, 11180, 11189, 11198, 11206,  
 11244, 11282, 11296, 11305, 11315,  
 11360, 11361, 11362, 11372, 11379,  
 11381, 11670, 12820, 12901, 12919,  
 13157, 13198, 13207, 13228, 13238,  
 13242, 13249, 13257, 13524, 13537,  
 13851, 13860, 13871, 14274, 14297,  
 14306, 14323, 14327, 14341, 14344,  
 14506, 14507, 15997, 15998, 16033,  
 16041, 16107, 16254, 16270, 16274,  
 16286, 16296, 16391, 16444, 16447,  
 16448, 16453, 16467, 16505, 16506,  
 16507, 16508, 16509, 16510, 16511,  
 16512, 16513, 16514, 16515, 16516,  
 16529, 16531, 16542, 16545, 16559,  
 16564, 16568, 16695, 16786, 16787,  
 16796, 16797, 16808, 16809, 16810,  
 16821, 16822, 16823, 16830, 16841,  
 16842, 16843, 16853, 16854, 16858,  
 16859, 16867, 16870, 16871, 16879,  
 16890, 16910, 16933, 16968, 16985,  
 17004, 17005, 17014, 17020, 17040,  
 17041, 17069, 17078, 17095, 17102,  
 17110, 17111, 17212, 17213, 17214,  
 17217, 17220, 17259, 17275, 17301,  
 17302, 17309, 17317, 17346, 17347,  
 17350, 17352, 17353, 17358, 17368,  
 17371, 17373, 17378, 17409, 17422,  
 17427, 17433, 17436, 17437, 17471,  
 17472, 17499, 17500, 17513, 17516,  
 17527, 17550, 17569, 17579, 17595,  
 17604, 17610, 17616, 17620, 17625,  
 17631, 17646, 17657, 17676, 17686,  
 17688, 17694, 17715, 17743, 17766,  
 17799, 17801, 17924, 17974, 17978,  
 17988, 17989, 18005, 18018, 18022,  
 18032, 18033, 18053, 18074, 18077,  
 18107, 18139, 18155, 18175, 18216,  
 18228, 18241, 18243, 18263, 18264,

18271, 18289, 18328, 18338, 18489,  
 18505, 18516, 18545, 18546, 18547,  
 18554, 18556, 18557, 18563, 18564,  
 18567, 18594, 18602, 18603, 18611,  
 18612, 18614, 18615, 18786, 18799,  
 18809, 18810, 18815, 18816, 18817,  
 18818, 18819, 18820, 18827, 18837,  
 18844, 18855, 18856, 18867, 18884,  
 18929, 18930, 18937, 18950, 18965,  
 18975, 18989, 19019, 19028, 19062,  
 19084, 19102, 19119, 19136, 19137,  
 19139, 19140, 19145, 19160, 19193,  
 19222, 19223, 19224, 19225, 19226,  
 19239, 19283, 19356, 19423, 19425,  
 19426, 19436, 19465, 19468, 19469,  
 19480, 19500, 19563, 19564, 19565,  
 19577, 19616, 19617, 19618, 19619,  
 19625, 19628, 19630, 19640, 19658,  
 19673, 19685, 19692, 19939, 19943,  
 19945, 19949, 19956, 19957, 19967,  
 19968, 19971, 20063, 20133, 20144,  
 20156, 20188, 20195, 20206, 20222,  
 20229, 20290, 20347, 20357, 20359,  
 20369, 20387, 20388, 20420, 20423,  
 20432, 20434, 20436, 20450, 20464,  
 20488, 20572, 20580, 20611, 20619,  
 20625, 20636, 20639, 20642, 20651,  
 20661, 20663, 20669, 20676, 20679,  
 20688, 20696, 20717, 20750, 20751,  
 20778, 20780, 20798, 20799, 20818,  
 20829, 20838, 20841, 20852, 20855,  
 20858, 20876, 20886, 20897, 20899,  
 20908, 20955, 20970, 20985, 21000,  
 21015, 21030, 21033, 21035, 21052,  
 21097, 21404, 21440, 21441, 21451,  
 21492, 21493, 21517, 21544, 21545,  
 21548, 21550, 21551, 21556, 21568,  
 21587, 21592, 21600, 21603, 21635,  
 21645, 21646, 21656, 21678, 21693,  
 21711, 21719, 21722, 21750, 21758,  
 21774, 21846, 21864, 21900, 21918,  
 21951, 21974, 21980, 22053, 22054,  
 22185, 22186, 22195, 22202, 22207,  
 22217, 22227, 22252, 22255, 22268,  
 22300, 22308, 22309, 22337, 22359,  
 22360, 22361, 22364, 22369, 22389,  
 22390, 22442, 22443, 22484, 22492,  
 22545, 22551, 22564, 22981, 22992,  
 22993, 23037, 23068, 23110, 23121,  
 23130, 23139, 23194, 23204, 23214,  
 23326, 23328, 23382, 23386, 23390,  
 23417, 23428, 23446, 23447, 23448,  
 23455, 23471, 23477, 23480, 23488,  
 23498, 23513, 23521, 23529, 23540,  
 23556, 23576, 23589, 23611, 23629,  
 23637, 23639, 23642, 23649, 23654,  
 23731, 23732, 23875, 23882, 23883,  
 23889, 23892, 23895, 23902, 23903,  
 23906, 23910, 23911, 23921, 23922,  
 23927, 23928, 23940, 23948, 23957,  
 23958, 23986, 24147, 24158, 24210,  
 24212, 24219, 24222, 24223, 24227,  
 24231, 24232, 24233, 24234, 24243,  
 24244, 24248, 24251, 24252, 24253,  
 24289, 24292, 24313, 24316, 24324,  
 24335, 24336, 24347, 24348, 24356,  
 24368, 24369, 24379, 24380, 24393,  
 24412, 24413, 24421, 24422, 24473,  
 24483, 24509, 24523, 24527, 24590,  
 24638, 24641, 24642, 24657, 24660,  
 24683, 24750, 24751, 24756, 24783,  
 24784, 24795, 24799, 24833, 24838,  
 24846, 24881, 24888, 24893, 24903,  
 24915, 24941, 25004, 25033, 25072,  
 25079, 25090, 25164, 25181, 25185,  
 25199, 25233, 25445, 25484, 25500,  
 25524, 25545, 25554, 25611, 25618,  
 25638, 25656, 25667, 25669, 25699,  
 25702, 25727, 25839, 25868, 25891,  
 25892, 25913, 25942, 25968, 26041,  
 26099, 26111, 26174, 26188, 26189,  
 26210, 26221, 26247, 26264, 26314,  
 26316, 26331, 26333, 26354, 26456,  
 26515, 26532, 26533, 26572, 26573,  
 26577, 26578, 26600, 26605, 26642,  
 26680, 26688, 26689, 26693, 26694,  
 27093, 27095, 27101, 29295, 29296,  
 29304, 29318, 29322, 29323, 29339,  
 29418, 29422, 29433, 29454, 29458,  
 29474, 29486, 29518, 29519, 29520,  
 29521, 29522, 29523, 29524, 29672,  
 29678, 30456, 30457, 30460, 30461,  
 30484, 30485, 30486, 30487, 30488,  
 30489, 30510, 30511, 30512, 30513,  
 30514, 30531, 30532, 30533, 30534,  
 32042, 32162, 32414, 32421, 32426,  
 32452, 32458, 32462, 32477, 32498,  
 32506, 32542, 32543, 32544, 32550  
 file commands:  
 \file\_add\_path:nN ..... 32763  
 \file\_compare\_timestamp:nNn ... 169  
 \file\_compare\_timestamp:nNnTF ...  
 ..... 169, 13835  
 \file\_compare\_timestamp\_p:nNn ...  
 ..... 169, 13835  
 \g\_file\_curr\_dir\_str .....  
 .... 165, 13363, 13933, 13939, 13952  
 \g\_file\_curr\_ext\_str .....

```

 165, 13363, 13935, 13941, 13954
\g_file_curr_name_str
 165, 9741, 11806,
 13363, 13934, 13940, 13953, 32776
\g_file_current_name_t1
\file_full_name:n
 166, 13547, 13640, 13700,
 13708, 13714, 13760, 13839, 13840
\file_get:nnN
 . 166, 13506, 32943, 32945, 32947,
 32951, 32956, 32960, 32967, 32971
\file_get:nnNTF ... 166, 13506, 13508
\file_get_full_name:nN
 166, 306, 13631, 32764
\file_get_full_name:nNNTF ... 166,
 12745, 13513, 13631, 13633, 13645,
 13646, 13889, 13895, 13900, 13912
\file_get_hex_dump:nN ... 167, 13776
\file_get_hex_dump:nnN ... 167, 13820
\file_get_hex_dump:nnnNTF
 167, 13820, 13822
\file_get_hex_dump:nNNTF
 167, 13776, 13777
\file_get_mdfive_hash:nN
 168, 13778, 13786
\file_get_mdfive_hash:nN\file_-
 get_size:nN 13776
\file_get_mdfive_hash:nN\file_-
 get_size:nNNTF 13776
\file_get_mdfive_hash:nNNTF 168, 13779
\file_get_size:nN 168
\file_get_size:nNNTF 168, 13781
\file_get_timestamp:nN ... 168, 13776
\file_get_timestamp:nNNTF
 168, 13776, 13783
\file_hex_dump:n ... 167, 167, 13711
\file_hex_dump:nnn
 167, 167, 13711, 13829
\file_if_exist:nTF
 . 166, 166, 166, 169, 5579, 13887,
 14185, 14187, 14191, 32766, 32768
\file_if_exist_input:n ... 169, 13893
\file_if_exist_input:nTF
 169, 13893, 32765, 32767
\file_input:n ... 169, 169, 169,
 169, 5583, 13910, 32766, 32768, 32901
\file_input_stop: 169, 13904
\file_list: 32769
\file_log_list: ... 169, 14013, 32770
\file_mdfive_hash:n .. 168, 168, 13693
\file_parse_full_name:n
 167, 665, 13956
\file_parse_full_name:nNNN
 .. 167, 167, 167, 13670, 13937, 14001
\file_parse_full_name_apply:nN ..
 167, 665, 13956, 14003
\file_path_include:n
 . 169, 32771
\file_path_remove:n
 32773
\l_file_search_path_seq
 166, 166, 167, 168, 168,
 168, 13397, 13558, 13655, 32772, 32774
\file_show_list:
 . 169, 14013
\file_size:n 168, 168, 13693
\file_timestamp:n ... 168, 168, 13693
file internal commands:
\l__file_base_name_t1
 13392, 13652, 13688
__file_compare_timestamp:nnN .. 13835
__file_const:nn
 14199
__file_details:nn
 13693
__file_details_aux:nn .. 13693, 13728
\l__file_dir_str
 13394, 13671, 13938, 13939
__file_ext_check:n
 13567, 13578, 13585
__file_ext_check:nn .. 13600, 13605
__file_ext_check:nnw .. 13591, 13596
__file_ext_check:nw
 13586, 13587, 13594
\l__file_ext_str
 .. 13394, 13671, 13672, 13938, 13941
__file_full_name:n 13547
__file_full_name_assign:nnnNNN .
 14004, 14006
__file_full_name_aux:nN 13547
__file_full_name_aux:Nnn 13547
\l__file_full_name_t1
 13392, 13513, 13516,
 13662, 13664, 13670, 13675, 13677,
 13680, 13687, 13689, 13889, 13895,
 13896, 13900, 13901, 13912, 13913
__file_get_aux:nnN
 13506
__file_get_details:nnN 13776
__file_get_do:Nw 13506
__file_get_full_name_search:nN .
 13631
__file_hex_dump:n 13711
__file_hex_dump_auxi:nnn 13711
__file_hex_dump_auxii:nnnn .. 13711
__file_hex_dump_auxiii:nnnn .. 13711
__file_hex_dump_auxiiv:nnn .. 13711
__file_hex_dump_auxiv:nnn
 13745, 13747, 13752
__file_id_info_auxi:w 14042
__file_id_info_auxii:w .. 668, 14042
__file_id_info_auxiii:w 14042
__file_if_recursion_tail_-
 break:NN 13404

```

```

__file_if_recursion_tail_stop:N 13404, 13430
__file_if_recursion_tail_stop_- do:Nn 13404
__file_if_recursion_tail_stop_- do:nn 13405, 13470
__file_input:n . 13896, 13901, 13910
__file_input_pop: 13910
__file_input_pop:nnn 13910
__file_input_push:n 13910
\g__file_internal_ior 13666, 13674, 13676, 13679, 13689, 13690, 13692, 14100, 14111, 14113
\l__file_internal_tl 13362, 13946, 13947
__file_kernel_dependency_- compare:nnm 14079
__file_list:N 14013
__file_list_aux:n 14013
\c__file_marker_tl 655, 13505, 13528, 13541
__file_mdfive_hash:n 13693
__file_mismatched_dependency_- error:nn 14095, 14098
__file_name_cleanup:w 13547
__file_name_end: 13547
__file_name_ext_check:n 13547
__file_name_ext_check:nn 13547
__file_name_ext_check:nnw 13547
__file_name_ext_check:nw 13547
\l__file_name_str 13394, 13671, 13938, 13940
__file_parse_full_name_area:nw 665, 13966
__file_parse_full_name_auxi:n . 13963, 13966
__file_parse_full_name_base:nw 666, 13974, 13977
__file_parse_full_name_tidy:nnnN 666, 13984, 13985, 13987, 13991
__file_parse_version:w 14079
__file_quark_if_nil:nTF .. 13401, 13487, 13501, 13589, 13598
__file_quark_if_nil_p:n 13401
\g__file_record_seq ... 664, 667, 667, 13391, 13920, 14023, 14037, 14038
__file_size:n 13546, 13556, 13574, 13607, 13611
\g__file_stack_seq 664, 13366, 13931, 13946
__file_str_cmp:nn 13834, 13864
__file_timestamp:n 13835
__file_tmp:w 13368, 13372, 13376, 13382, 13388
\l__file_tmp_seq ... 13398, 14017, 14020, 14023, 14024, 14026, 14034, 14039, 14110, 14112, 14135, 14139
\filedump 779
\filemoddate 780
\filesize 781
\finalhyphendemerits 275
\firstmark 276
\firstmarks 533
\firstvalidlanguage 822
flag commands:
\flag_clear:n 103, 103, 5675, 5703, 5797, 5826, 5895, 5943, 5944, 5996, 5997, 6231, 6232, 6233, 6234, 6235, 6336, 6431, 6432, 6433, 6434, 6589, 6590, 6591, 9023, 9036, 25102, 26538, 26539
\flag_clear_new:n 103, 454, 6178, 6179, 6180, 6181, 6359, 6360, 6361, 6539, 6540, 9035
\flag_height:n .. 104, 5503, 9044, 9060, 9074, 26548, 26549, 26555, 26556
\flag_if_exist:n 104
\flag_if_exist:nTF .. 104, 9036, 9047
\flag_if_exist_p:n 104, 9047
\flag_if_raised:n 104
\flag_if_raised:nTF 104, 5496, 5501, 5503, 6205, 6211, 6216, 6223, 6393, 6398, 6403, 6554, 6561, 9052, 25110
\flag_if_raised_p:n 104, 9052
\flag_log:n 103, 9037
\flag_new:n 103, 103, 454, 525, 5365, 5366, 9018, 9036, 16591, 16592, 16593, 16594, 25092, 26442, 26443
\flag_raise:n 104, 5661, 5711, 5811, 5844, 5915, 5928, 5966, 5971, 6052, 6252, 6253, 6276, 6277, 6290, 6291, 6310, 6311, 6317, 6318, 6348, 6498, 6499, 6608, 6609, 6613, 6614, 6628, 6629, 9071, 26571, 26576
\flag_raise_if_clear:n 267, 16625, 16634, 16642, 16659, 16668, 16699, 25129, 25151, 32157
\flag_show:n 103, 9037
flag fp commands:
flag_fp_division_by_zero . 209, 16591
flag_fp_invalid_operation 209, 16591
flag_fp_overflow 209, 16591
flag_fp_underflow 209, 16591
flag internal commands:
__flag_clear:wn 9023
__flag_height_end:wn 9060
__flag_height_loop:wn 9060
__flag_show:Nn 9037

```

\floatingpenalty ..... 277  
\floor ..... 214  
\fmtname ..... 110, 9432,  
9435, 9436, 9448, 9458, 29802, 29803  
\font ..... 278  
\fontchardp ..... 534  
\fontcharht ..... 535  
\fontcharic ..... 536  
\fontcharwd ..... 537  
\fontdimen ..... 279  
\fontencoding ..... 31616  
\fontfamily ..... 31617  
\fontid ..... 823  
\fontname ..... 280  
\fontseries ..... 31618  
\fontshape ..... 31619  
\fontsize ..... 31622  
\footnotesize ..... 31658  
\forcecjktoken ..... 1173  
\formatname ..... 824  
fp commands:  
  \c\_e\_fp ..... 208, 210, 18466  
  \fp\_abs:Nn ..... 213, 218, 920, 22075, 27506,  
27608, 27610, 27612, 28435, 28437  
  \fp\_add:Nn ..... 202, 920, 920, 18443  
  \fp\_compare:nNnTF .....  
   ... 204, 205, 205, 205, 206, 206,  
18507, 18648, 18654, 18659, 18667,  
18728, 18734, 27363, 27365, 27370,  
27639, 27654, 27663, 28175, 28409  
  \fp\_compare:nTF .....  
   ... 204, 205, 206, 206, 206,  
212, 18491, 18620, 18626, 18631, 18639  
  \fp\_compare\_p:n ..... 205, 18491  
  \fp\_compare\_p:nNn ..... 204, 18507  
  \fp\_const:Nn .....  
   ... 201, 18420, 18466, 18467, 18468, 18469  
  \fp\_do\_until:nNn ..... 206, 18617  
  \fp\_do\_until:nNnn ..... 205, 18645  
  \fp\_do\_while:nNn ..... 206, 18617  
  \fp\_do\_while:nNnn ..... 205, 18645  
  \fp\_eval:n .....  
   ... 202, 203, 205, 212, 212, 212,  
212, 212, 213, 213, 213, 213, 213,  
213, 213, 214, 214, 214, 214, 214,  
215, 215, 215, 216, 216, 216, 216, 217,  
217, 218, 218, 804, 22070, 32305, 32324  
  \fp\_format:nn ..... 219  
  \fp\_gadd:Nn ..... 202, 18443  
  .fp\_gset:N ..... 188, 15333  
  \fp\_gset:Nn ..... 202, 18420, 18444, 18446  
  \fp\_gset\_eq:NN ..... 202, 18429, 18434  
  \fp\_gsub:Nn ..... 202, 18443  
  \fp\_gzero:N ..... 201, 18433, 18440  
  \fp\_gzero\_new:N ..... 202, 18437  
  \fp\_if\_exist:NTF .....  
   ... 204, 18438, 18440, 18481  
  \fp\_if\_exist\_p:N ..... 204, 18481  
  \fp\_if\_nan:n ..... 266  
  \fp\_if\_nan:nTF ..... 219, 266, 18483  
  \fp\_if\_nan\_p:n ..... 266, 18483  
  \fp\_log:N ..... 209, 18453  
  \fp\_log:n ..... 209, 18462  
  \fp\_max:nn ..... 218, 22077  
  \fp\_min:nn ..... 218, 22077  
  \fp\_new:N .....  
   ... 201, 202, 18417, 18438, 18440,  
18470, 18471, 18472, 18473, 27329,  
27330, 27331, 27457, 27458, 27707,  
27708, 28202, 28203, 28369, 28370  
  .fp\_set:N ..... 188, 15333  
  \fp\_set:Nn ..... 202, 18420, 18443, 18445,  
27351, 27352, 27353, 27476, 27478,  
27519, 27539, 27559, 27576, 27578,  
27596, 27597, 27637, 27638, 28220,  
28221, 28389, 28391, 28429, 28430  
  \fp\_set\_eq:NN ..... 202, 18429, 18433,  
27524, 27544, 27561, 27640, 27641  
  \fp\_show:N ..... 209, 18453  
  \fp\_show:n ..... 209, 18462  
  \fp\_sign:n ..... 203, 22073  
  \fp\_step\_function:nnnN .....  
   ... 207, 18673, 18765  
  \fp\_step\_inline:nnnn ..... 207, 18743  
  \fp\_step\_variable:nnnNn ..... 207, 18743  
  \fp\_sub:Nn ..... 202, 18443  
  \fp\_to\_decimal:N .....  
   ... 203, 204, 16584, 21877, 21908, 22070  
  \fp\_to\_decimal:n .....  
   ... 202, 203, 203, 204, 204, 21877,  
22072, 22074, 22076, 22078, 22080  
  \fp\_to\_dim:N ..... 203, 918, 22000  
  \fp\_to\_dim:n ..... 203, 208, 22000, 27395,  
27406, 27506, 28130, 28152, 28180,  
28194, 28306, 28314, 28445, 28447  
  \fp\_to\_int:N ..... 203, 22016  
  \fp\_to\_int:n ..... 203, 22016  
  \fp\_to\_scientific:N .....  
   ... 203, 21823, 21854, 21861  
  \fp\_to\_scientific:n ..... 203, 204, 21823  
  \fp\_to\_tl:N .....  
   ... 204, 221, 16585, 18460, 21956  
  \fp\_to\_tl:n ..... 204,  
16211, 16624, 16633, 16658, 16667,  
16696, 18301, 18316, 18463, 18465,  
18700, 18701, 18720, 18731, 21956  
  \fp\_trap:nn ..... 209, 209,  
746, 16595, 16710, 16711, 16712, 16713

\fp\_until\_do:nn ..... 206, [18617](#)  
\fp\_until\_do:nNnn ..... 206, [18645](#)  
\fp\_use:N ..... 204, [221](#), [22070](#)  
\fp\_while\_do:nn ..... 206, [18617](#)  
\fp\_while\_do:nNnn ..... 206, [18645](#)  
\fp\_zero:N ..... 201, [202](#), [18433](#), [18438](#)  
\fp\_zero\_new:N ..... 202, [18437](#)  
\c\_inf\_fp ..... 208,  
  217, [16225](#), [17809](#), [19236](#), [19318](#),  
  19656, [20416](#), [20439](#), [20641](#), [20644](#),  
  20648, [20672](#), [20874](#), [21037](#), [22562](#)  
\c\_nan\_fp .... 217, [749](#), [773](#), [16225](#),  
  16635, [16643](#), [16715](#), [16921](#), [16940](#),  
  16946, [16969](#), [17136](#), [17144](#), [17152](#),  
  17230, [17287](#), [17326](#), [17721](#), [17798](#),  
  17810, [18303](#), [18318](#), [18724](#), [20615](#),  
  22114, [22160](#), [22475](#), [22534](#), [22560](#)  
\c\_one\_fp ..... 207, [801](#),  
  904, [17813](#), [18246](#), [18267](#), [18466](#),  
  18824, [19677](#), [20410](#), [20610](#), [20662](#),  
  20847, [20961](#), [20991](#), [21540](#), [22176](#)  
\c\_pi\_fp .. 208, [217](#), [783](#), [17811](#), [18468](#)  
\g\_tmpa\_fp ..... 208, [18470](#)  
\l\_tmpa\_fp ..... 208, [18470](#)  
\g\_tmpb\_fp ..... 208, [18470](#)  
\l\_tmpb\_fp ..... 208, [18470](#)  
\c\_zero\_fp 207, [804](#), [819](#), [931](#), [16225](#),  
  16279, [17814](#), [18258](#), [18270](#), [18418](#),  
  18433, [18434](#), [18826](#), [18829](#), [19065](#),  
  19232, [20419](#), [20440](#), [20638](#), [20675](#),  
  21755, [21861](#), [22045](#), [22559](#), [27363](#),  
  27365, [27370](#), [27654](#), [27663](#), [28409](#)  
fp internal commands:  
  \fp\_&\_o:ww ..... 806, [815](#), [18830](#)  
  \fp\_&\_tuple\_o:ww ..... 18830  
  \fp\_\*\_o:ww ..... 19197  
  \fp\_\*\_tuple\_o:ww ..... 19703  
  \fp+\_o:ww .... 817, [818](#), [846](#), [18918](#)  
  \fp-\_o:ww ..... 817, [818](#), [18913](#)  
  \fp/\_o:ww ..... 826, [868](#), [19309](#)  
  \fp^\_o:ww ..... 20606  
  \fp\_acos\_o:w ..... 909, [911](#), [21696](#)  
  \fp\_acot\_o:Nw .. 20936, [20938](#), [21528](#)  
  \fp\_acotii\_o:Nww .... 21538, [21541](#)  
  \fp\_acotii\_o:ww ..... 904  
  \fp\_acsc\_normal\_o:NnwNnw .....  
    911, [21754](#), [21769](#), [21777](#)  
  \fp\_acsc\_o:w ..... 21748  
  \fp\_add:NNNn ..... 18443  
  \fp\_add\_big\_i:wNww ..... 820  
  \fp\_add\_big\_i\_o:wNww .....  
    817, [820](#), [18985](#), [18992](#)  
  \fp\_add\_big\_ii:wNww ..... 820  
  \fp\_add\_big\_ii\_o:wNww 18988, [18992](#)  
  \fp\_array\_bounds:NNnTF .....  
    22431, [22462](#), [22532](#)  
\fp\_array\_bounds\_error:NNn .. [22431](#)  
\fp\_array\_count:n ..... 16328,  
  16905, [18574](#), [18575](#), [19716](#), [21796](#)  
\fp\_array\_gset:NNNNww ..... 22450  
\fp\_array\_gset:w ..... 22450  
\fp\_array\_gset\_normal:w ..... 22450  
\fp\_array\_gset\_recover:Nw .. 22450  
\fp\_array\_gset\_special:nnNNN ..  
  22450, [22507](#)  
\fp\_array\_gzero:N ..... 930  
\fp\_array\_if\_all\_fp:nTF .....  
  16340, [18296](#)  
\fp\_array\_if\_all\_fp\_loop:w .. [16340](#)  
\g\_fp\_array\_int .....  
  22396, [22403](#), [22405](#), [22417](#)  
\fp\_array\_item:N ..... 22514  
\fp\_array\_item:NNNnN ..... 22514  
\fp\_array\_item:NwN ..... 22514  
\fp\_array\_item:w ..... 22514  
\fp\_array\_item\_normal:w ..... 22514  
\fp\_array\_item\_special:w ... 22514  
\l\_fp\_array\_loop\_int .....  
  22397, [22503](#), [22506](#), [22509](#)  
\fp\_array\_new:nNNN ..... 22398  
\fp\_array\_new:nNNNN .. 22407, [22411](#)  
\fp\_array\_to\_clist:n .....  
  16973, [22081](#), [22200](#)  
\fp\_array\_to\_clist\_loop:Nw .. [22081](#)  
\fp\_asec\_o:w ..... 21761  
\fp\_asin\_auxi\_o:NnNww .....  
  909, [910](#), [911](#), [21726](#), [21729](#), [21788](#)  
\fp\_asin\_isqrt:wn ..... 21729  
\fp\_asin\_normal\_o:NnwNnnnnw ..  
  21687, [21703](#), [21714](#)

```

__fp_asin_o:w 21681
__fp_atan_auxi:ww . 906, 21606, 21620
__fp_atan_auxii:w 21620
__fp_atan_combine_aux:ww 21647
__fp_atan_combine_o:NwwwwN ...
..... 905, 906, 21565, 21582, 21647
__fp_atan_default:w 801, 904, 21528
__fp_atan_div:wnwwnw
..... 906, 21593, 21595
__fp_atan_inf_o:NNNw 904, 21553,
21554, 21555, 21563, 21699, 21772
__fp_atan_near:wwn .. 21595
__fp_atan_near_aux:wwn .. 21595
__fp_atan_normal_o:NNnwNnw ...
..... 904, 21573, 21573
__fp_atan_o:Nw . 20940, 20942, 21528
__fp_atan_Taylor_break:w ... 21631
__fp_atan_Taylor_loop:www ...
..... 907, 21626, 21631
__fp_atan_test_o:NwwNwwN ...
..... 910, 21576, 21580, 21736
__fp_atanii_o:Nww 21532, 21541
__fp_basics_pack_high:NNNNNw ...
... 821, 838, 16438, 19033, 19185,
19288, 19300, 19442, 19635, 20161
__fp_basics_pack_high_carry:w ...
..... 739, 16438
__fp_basics_pack_low:NNNNNw ...
..... 828, 838,
16438, 19035, 19187, 19290, 19302,
19444, 19584, 19586, 19637, 20163
__fp_basics_pack_weird_high:NNNNNNNNw ...
..... 220, 16449, 19044, 19453
__fp_basics_pack_weird_low:NNNNw ...
..... 220, 16449, 19046, 19455
\c__fp_big_leading_shift_int ...
.. 16424, 19514, 19849, 19859, 19869
\c__fp_big_middle_shift_int ...
.... 16424, 19517, 19520, 19523,
19526, 19529, 19532, 19536, 19851,
19861, 19871, 19881, 19884, 19887
\c__fp_big_trailing_shift_int ...
..... 16424, 19540, 19894
\c__fp_Bigg_leading_shift_int ...
..... 16429, 19363, 19381
\c__fp_Bigg_middle_shift_int ...
.. 16429, 19366, 19369, 19384, 19387
\c__fp_Bigg_trailing_shift_int ...
..... 16429, 19372, 19390
__fp_binary_rev_type_o:Nww ...
..... 17932, 19706, 19708
__fp_binary_type_o:Nww ...
..... 17932, 19704, 19717
\c__fp_block_int 16230, 20113
__fp_case_return:nw
.. 742, 16506, 16536, 16539, 16544,
17034, 20375, 20871, 21553, 21554,
21555, 21848, 21902, 21976, 21978,
21979, 22045, 22480, 22482, 22483
__fp_case_return_i_o:ww . 16513,
18935, 18949, 18958, 19230, 21544
__fp_case_return_ii_o:ww
.. 16513, 19231, 20660, 20678, 21545
__fp_case_return_o:Nw . 742, 743,
16507, 19656, 20410, 20415, 20418,
20610, 20615, 20638, 20641, 20644,
20847, 20961, 20991, 21755, 21757
__fp_case_return_o:Nww
..... 16511, 19232, 19233,
19236, 19237, 20662, 20671, 20674
__fp_case_return_same_o:w . 742,
743, 16509, 19465, 19469, 19657,
19669, 19672, 20194, 20422, 20635,
20851, 20854, 20946, 20954, 20969,
20984, 20999, 21006, 21014, 21029,
21684, 21692, 21710, 21756, 21773
__fp_case_use:nw
.. 16505, 18960, 19228, 19229, 19234,
19235, 19317, 19320, 19467, 19653,
20187, 20190, 20646, 20857, 20947,
20952, 20962, 20967, 20977, 20982,
20992, 20997, 21007, 21012, 21022,
21027, 21686, 21689, 21699, 21701,
21707, 21751, 21753, 21764, 21767,
21772, 21851, 21858, 21905, 21912
__fp_change_func_type:NNN ...
... 16368, 17725, 19699, 21833,
21887, 21964, 22010, 22025, 22464
__fp_change_func_type_aux:w . 16368
__fp_change_func_type_chk:NNN 16368
__fp_chk:w
.. 729,
731, 783, 818, 819, 820, 822, 828,
830, 16212, 16225, 16226, 16227,
16228, 16229, 16239, 16244, 16246,
16247, 16275, 16278, 16280, 16290,
16303, 16322, 16517, 16533, 16691,
16696, 16923, 16977, 16986, 16988,
17823, 18541, 18542, 18704, 18720,
18724, 18788, 18789, 18792, 18803,
18804, 18812, 18813, 18821, 18833,
18836, 18840, 18843, 18919, 18939,
18940, 18942, 18943, 18944, 18952,
18955, 18966, 18967, 18969, 18978,
19054, 19206, 19240, 19241, 19244,
19325, 19463, 19471, 19473, 19650,
19659, 19661, 19666, 19674, 19676,
19678, 19682, 20184, 20196, 20198,
20407, 20424, 20426, 20607, 20626,

```

20628, 20629, 20632, 20649, 20652,  
 20655, 20680, 20681, 20683, 20699,  
 20788, 20801, 20803, 20807, 20811,  
 20844, 20860, 20943, 20956, 20958,  
 20971, 20973, 20986, 20988, 21001,  
 21003, 21016, 21018, 21031, 21041,  
 21542, 21558, 21559, 21563, 21574,  
 21681, 21694, 21696, 21712, 21715,  
 21725, 21748, 21759, 21761, 21775,  
 21777, 21782, 21844, 21865, 21868,  
 21898, 21919, 21922, 21972, 21988,  
 21991, 22066, 22067, 22177, 22179,  
 22211, 22477, 22485, 22488, 22567  
 $\backslash\_\_fp\_compare:wNNNNw \dots . 18186$   
 $\backslash\_\_fp\_compare\_aux:wn \dots . 18507$   
 $\backslash\_\_fp\_compare\_back:ww \dots .$   
 $\quad . . . . . 925, 18523, 18802, 22195$   
 $\backslash\_\_fp\_compare\_back\_any:ww \dots . 807,$   
 $\quad . . . . . 807, 808, 18261, 18520, 18523, 18591$   
 $\backslash\_\_fp\_compare\_back\_tuple:ww \dots . 18568$   
 $\backslash\_\_fp\_compare\_nan:w \dots . 807, 18523$   
 $\backslash\_\_fp\_compare\_npos:nwnw \dots . 806,$   
 $\quad . . . . . 807, 809, 18551, 18597, 19056, 19963$   
 $\backslash\_\_fp\_compare\_return:w \dots . 18491$   
 $\backslash\_\_fp\_compare\_significand:nnnnnnnn$   
 $\quad . . . . . 18597$   
 $\backslash\_\_fp\_cos\_o:w \dots . 20958$   
 $\backslash\_\_fp\_cot\_o:w \dots . . 889, 21018$   
 $\backslash\_\_fp\_cot\_zero\_o:Nnw \dots .$   
 $\quad . . . . . 888, 890, 20976, 21018$   
 $\backslash\_\_fp\_csc\_o:w \dots . . 20973$   
 $\backslash\_\_fp\_decimate:nNnnn \dots .$   
 $\quad . . . . . 740, 744, 884, 16459,$   
 $\quad . . . . . 16524, 16551, 16990, 18994, 19002,$   
 $\quad . . . . . 19081, 20453, 20457, 20826, 21928$   
 $\backslash\_\_fp\_decimate:_:Nnnnn \dots . 16471$   
 $\backslash\_\_fp\_decimate\_auxi:Nnnnn \dots . 741, 16475$   
 $\backslash\_\_fp\_decimate\_auxii:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxiii:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxiv:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxix:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxv:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxvi:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxvii:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxviii:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxxx:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxxi:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxxii:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxxiii:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxxiv:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxxxv:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_auxxxvi:Nnnnn \dots . 16475$   
 $\backslash\_\_fp\_decimate\_pack:nnnnnnnnnw \dots .$   
 $\quad . . . . . 741, 16482, 16501$   
 $\backslash\_\_fp\_decimate\_pack:nnnnnnnw \dots .$   
 $\quad . . . . . 16502, 16503$   
 $\backslash\_\_fp\_decimate\_tiny:Nnnnn \dots . . 16471$   
 $\backslash\_\_fp\_div\_npos\_o:Nww \dots .$   
 $\quad . . . . . 830, 830, 19314, 19324$   
 $\backslash\_\_fp\_div\_significand\_calc:wwnnnnnnn$   
 $\quad . . . . . 833,$   
 $\quad . . . . . 834, 19341, 19350, 19398, 20267, 20274$   
 $\backslash\_\_fp\_div\_significand\_calc\_i:wwnnnnnnn \dots . . 19350$   
 $\backslash\_\_fp\_div\_significand\_calc\_ii:wwnnnnnnn \dots . . 19350$   
 $\backslash\_\_fp\_div\_significand\_i\_o:wnnw \dots .$   
 $\quad . . . . . 830, 833, 19331, 19337$   
 $\backslash\_\_fp\_div\_significand\_ii:wwn \dots .$   
 $\quad . . . . . 835, 19345, 19346, 19347, 19394$   
 $\backslash\_\_fp\_div\_significand\_iii:wwnnnnn$   
 $\quad . . . . . 836, 19348, 19401$   
 $\backslash\_\_fp\_div\_significand\_iv:wwnnnnnnn$   
 $\quad . . . . . 836, 19404, 19409$   
 $\backslash\_\_fp\_div\_significand\_large\_o:wwwNNNNwN \dots . 838, 19435, 19449$   
 $\backslash\_\_fp\_div\_significand\_pack:NNN \dots .$   
 $\quad . . . . . 837,$   
 $\quad . . . . . 870, 19396, 19429, 20254, 20272, 20280$   
 $\backslash\_\_fp\_div\_significand\_small\_o:wwwNNNNwN \dots . 838, 19433, 19439$   
 $\backslash\_\_fp\_div\_significand\_test\_o:w \dots .$   
 $\quad . . . . . 837, 837, 19339, 19430$   
 $\backslash\_\_fp\_div\_significand\_v:NN \dots .$   
 $\quad . . . . . 19414, 19416, 19419$   
 $\backslash\_\_fp\_div\_significand\_v:NNw \dots . . 19409$   
 $\backslash\_\_fp\_div\_significand\_vi:Nw \dots .$   
 $\quad . . . . . 836, 19409$   
 $\backslash\_\_fp\_division\_by\_zero\_o:Nnw \dots .$   
 $\quad . . . . . 746, 16655,$   
 $\quad . . . . . 16703, 19654, 20191, 21037, 21038$   
 $\backslash\_\_fp\_division\_by\_zero\_o:NNww \dots .$   
 $\quad . . . . . 746, 16663, 16703, 19318, 19321, 20648$   
 $\backslash\_\_fp\_empty\_tuple\_fp \dots .$   
 $\quad . . . . . 16323, 17130, 17784, 17794$   
 $\backslash\_\_fp\_ep\_compare:www \dots . . 19958, 21589$   
 $\backslash\_\_fp\_ep\_compare\_aux:www \dots . . 19958$   
 $\backslash\_\_fp\_ep\_div:wwwn \dots .$   
 $\quad . . . . . 902, 19988, 20099,$   
 $\quad . . . . . 21518, 21605, 21609, 21618, 21785$   
 $\backslash\_\_fp\_ep\_div\_eps\_pack:NNNNNw \dots . . 20018$   
 $\backslash\_\_fp\_ep\_div\_epsi:wnNNNNn \dots . . 860$   
 $\backslash\_\_fp\_ep\_div\_epsi:wnNNNNNn \dots .$   
 $\quad . . . . . 20015, 20018$   
 $\backslash\_\_fp\_ep\_div\_epsii:wwnNNNNNn \dots . . 20018$   
 $\backslash\_\_fp\_ep\_div\_estii:wwwn \dots .$   
 $\quad . . . . . 859, 19994, 19997$   
 $\backslash\_\_fp\_ep\_div\_estii:wwnnwwn \dots . . 19997$

`\_fp_ep_div_estiii:NNNNNwwnn . 19997`  
`\_fp_ep_inv_to_float_o:wN . . . . . 890`  
`\_fp_ep_inv_to_float_o:wwN . . . . .`  
`900, 20095, 20103, 20980, 20995`  
`\_fp_ep_isqrt:wwn . . . . . 20041, 21746`  
`\_fp_ep_isqrt_aux:wwn . . . . . 20041`  
`\_fp_ep_isqrt_auxi:wwn 20044, 20046`  
`\_fp_ep_isqrt_auxii:wwnnwn . 20041`  
`\_fp_ep_isqrt_epsi:wN . . . . .`  
`862, 20078, 20081`  
`\_fp_ep_isqrt_epsii:wwN . . . . . 20081`  
`\_fp_ep_isqrt_esti:wwnnwn . . . . .`  
`20056, 20059`  
`\_fp_ep_isqrt_estii:wwnnwn . 20059`  
`\_fp_ep_isqrt_estiii:NNNNNwwnn . . . . .`  
`20059`  
`\_fp_ep_mul:wwnn . . . . .`  
`886, 19973, 20887,`  
`20900, 21475, 21505, 21733, 21744`  
`\_fp_ep_mul_raw:wwwN . . . . .`  
`19973, 21059, 21425`  
`\_fp_ep_to_ep:wwN . 19924, 19975,`  
`19978, 19990, 19993, 20043, 21734`  
`\_fp_ep_to_ep_end:www . . . . . 19924`  
`\_fp_ep_to_ep_loop:N . . . . .`  
`899, 19924, 21426`  
`\_fp_ep_to_ep_zero:ww . . . . . 19924`  
`\_fp_ep_to_fixed:wwn . . . . . 19906,`  
`21056, 21612, 21621, 21731, 22220`  
`\_fp_ep_to_fixed_auxi:www . . . . . 19906`  
`\_fp_ep_to_fixed_auxii:nnnnnnwn . . . . .`  
`19906`  
`\_fp_ep_to_float_o:wN . . . . . 890`  
`\_fp_ep_to_float_o:wwN . . . . .`  
`888, 900, 20095,`  
`20107, 20911, 20950, 20965, 21524`  
`\_fp_error:nnnn . . . . . 16624,`  
`16632, 16641, 16658, 16666, 16694,`  
`16717, 16916, 16918, 16939, 16944,`  
`17720, 18299, 18314, 18700, 18719,`  
`18730, 21839, 21893, 21967, 22474`  
`\_fp_exp_after_?_f:nw 737, 769, 17114`  
`\_fp_exp_after_any_f:Nnw . . . . . 16393`  
`\_fp_exp_after_any_f:nw . . . . .`  
`737, 16393, 16419, 17116, 17889`  
`\_fp_exp_after_array_f:w . . . . .`  
`737, 16404,`  
`17774, 18870, 18881, 18891, 18899`  
`\_fp_exp_after_expr_mark_f:nw . . . . .`  
`769, 17114`  
`\_fp_exp_after_expr_stop_f:nw 16393`  
`\_fp_exp_after_f:nw . . . . .`  
`733, 769, 16280, 16398, 17822, 17960`  
`\_fp_exp_after_normal:nNNw . . . . .`  
`16283, 16293, 16310`  
`\_fp_exp_after_normal:Nwwww . . . . .`  
`16312, 16320`  
`\_fp_exp_after_o:w . . . . . 733, 16280,`  
`16510, 16514, 16516, 16984, 17028,`  
`17046, 18281, 18820, 18838, 18847,`  
`18856, 18943, 19680, 20800, 20805`  
`\_fp_exp_after_special:nNNw . . . . .`  
`734, 16285, 16295, 16300`  
`\_fp_exp_after_tuple_f:nw . . . . .`  
`16404, 18088`  
`\_fp_exp_after_tuple_o:w . . . . .`  
`16404, 18845, 18848, 18851, 18853`  
`\c_fp_exp_intarray . . . . .`  
`20500, 20586, 20593, 20596, 20598`  
`\_fp_exp_intarray:w . . . . . 20557`  
`\_fp_exp_intarray_aux:w . . . . . 20557`  
`\_fp_exp_large:NwN 877, 20557, 20784`  
`\_fp_exp_large_after:wwn 877, 20557`  
`\_fp_exp_normal_o:w . . . . . 20412, 20426`  
`\_fp_exp_o:w . . . . . 20170, 20407`  
`\_fp_exp_overflow>NN . . . . . 20426`  
`\_fp_exp_pos_large:NnnNwn . . . . .`  
`20458, 20557`  
`\_fp_exp_pos_o:NNwnw . . . . .`  
`20429, 20431, 20434`  
`\_fp_exp_pos_o:NnwNw . . . . . 20426`  
`\_fp_exp_Taylor:NnnNwn . . . . .`  
`20454, 20473, 20603`  
`\_fp_exp_Taylor_break:Nww . . . . . 20473`  
`\_fp_exp_Taylor_ii:ww . . . . . 20479, 20482`  
`\_fp_exp_Taylor_loop:www . . . . . 20473`  
`\_fp_expand:n . . . . . 920`  
`\_fp_exponent:w . . . . . 16247`  
`\_fp_factorial_int_o:n . . . . . 885`  
`\_fp_fact_int_o:n . . . . . 20865, 20868`  
`\_fp_fact_int_o:w . . . . . 20862`  
`\_fp_fact_loop_o:w . . . . . 20880, 20882`  
`\c_fp_fact_max_arg_int 20843, 20870`  
`\_fp_fact_o:w . . . . . 20174, 20844`  
`\_fp_fact_pos_o:w . . . . . 20859, 20862`  
`\_fp_fact_small_o:w . . . . . 20885, 20897`  
`\c_fp_five_int . . . . . 16778,`  
`16802, 16815, 16828, 16835, 16888`  
`\_fp_fixed_(calculation):wwn . . . . . 848`  
`\_fp_fixed_add:nnNnnwn . . . . . 19799`  
`\_fp_fixed_add:Nnnnnwnn . . . . . 19799`  
`\_fp_fixed_add:wwn . . . . . 848,`  
`851, 19799, 20039, 20349, 20357,`  
`20368, 20386, 21617, 21677, 22235`  
`\_fp_fixed_add_after:NNNNNwn . 19799`  
`\_fp_fixed_add_one:wN 849, 19731,`  
`20032, 20490, 20499, 21743, 22226`

`\_fp_fixed_add_pack:NNNNNwn . 19799`  
`\_fp_fixed_continue:wn . . . . .`  
`..... 19730, 19976,`  
`19981, 19991, 20568, 20759, 21094,`  
`21463, 21735, 21744, 22218, 22230`  
`\_fp_fixed_div_int:wnN . . . . . 19768`  
`\_fp_fixed_div_int:wwN . . . . .`  
`.... 850, 19768, 20348, 20489, 21636`  
`\_fp_fixed_div_int_after:Nw . . . . .`  
`..... 850, 19768`  
`\_fp_fixed_div_int_auxi:wnn . 19768`  
`\_fp_fixed_div_int_auxii:wnn . . . . .`  
`..... 850, 19768`  
`\_fp_fixed_div_int_pack:Nw . . . . .`  
`..... 850, 19768`  
`\_fp_fixed_div_myriad:wn . . . . .`  
`..... 19736, 20036`  
`\_fp_fixed_inv_to_float_o:wn . . . . .`  
`..... 20102, 20431, 20695`  
`\_fp_fixed_mul:nnnnnnnw . . . . . 19819`  
`\_fp_fixed_mul:wwn . . . . .`  
`... 848, 849, 852, 898, 900, 19819,`  
`19985, 20016, 20031, 20033, 20037,`  
`20090, 20093, 20106, 20350, 20360,`  
`20400, 20491, 20589, 20604, 20705,`  
`21432, 21486, 21624, 21657, 21659`  
`\_fp_fixed_mul_add:nnnnwnnn . . . . .`  
`..... 855, 19888, 19890`  
`\_fp_fixed_mul_add:nnnnwnnnw . . . . .`  
`..... 855, 19895, 19901`  
`\_fp_fixed_mul_add:Nwnnnwnnn . . . . .`  
`.... 854, 19852, 19862, 19873, 19877`  
`\_fp_fixed_mul_add:wwn . . . . .`  
`..... 853, 19846, 22240`  
`\_fp_fixed_mul_after:wwn . . . . .`  
`.... 852, 19738, 19744, 19747,`  
`19821, 19848, 19858, 19868, 20722`  
`\_fp_fixed_mul_one_minus_-`  
`mul:wwn . . . . . 19846`  
`\_fp_fixed_mul_short:wwn . . . . .`  
`..... 849, 19745,`  
`20014, 20035, 20077, 20079, 21670`  
`\_fp_fixed_mul_sub_back:wwn . . . . .`  
`..... 853, 19846,`  
`20091, 21453, 21455, 21456, 21457,`  
`21458, 21459, 21460, 21461, 21462,`  
`21466, 21468, 21469, 21470, 21471,`  
`21472, 21473, 21474, 21499, 21501,`  
`21502, 21503, 21504, 21507, 21509,`  
`21510, 21511, 21512, 21637, 21645`  
`\_fp_fixed_one_minus_mul:wwn . . . . .`  
`..... 853, 854, 19866`  
`\_fp_fixed_sub:wwn 19799, 20083,`  
`20366, 20382, 20394, 21098, 21618,`  
`21675, 21741, 22228, 22237, 22269`  
`\_fp_fixed_to_float_o:Nw . . . . .`  
`..... 20109, 20375`  
`\_fp_fixed_to_float_o:wN . . . . .`  
`..... 848, 864,`  
`908, 20096, 20109, 20395, 20405,`  
`20429, 20691, 21665, 22168, 22274`  
`\_fp_fixed_to_float_pack:ww . . . . .`  
`..... 20142, 20152`  
`\_fp_fixed_to_float_rad_o:wN . . . . .`  
`..... 20104, 21665`  
`\_fp_fixed_to_float_round_-`  
`up:wnnnnw . . . . . 20155, 20159`  
`\_fp_fixed_to_float_zero:w . . . . .`  
`..... 20138, 20147`  
`\_fp_fixed_to_loop:N . . . . .`  
`..... 20115, 20125, 20129`  
`\_fp_fixed_to_loop_end:w . . . . .`  
`..... 20131, 20135`  
`\_fp_from_dim:wNNnnnnnn . . . . . 22035`  
`\_fp_from_dim:wnnnnnwNn 22062, 22063`  
`\_fp_from_dim:wnnnnnwNw . . . . . 22035`  
`\_fp_from_dim:wNw . . . . . 22035`  
`\_fp_from_dim_test:ww . . . . .`  
`.... 919, 17206, 17243, 17841, 22035`  
`\_fp_func_to_name:N . . . . .`  
`..... 16571, 17720, 17729`  
`\_fp_func_to_name_aux:w . . . . . 16571`  
`\c_fp_half_prec_int . . . . .`  
`..... 16230, 17447, 17479`  
`\_fp_if_type_fp:NTwFw . 735, 801,`  
`16260, 16339, 16347, 16354, 16370,`  
`16397, 18308, 18322, 18499, 18525,`  
`18526, 18693, 18694, 18695, 18861`  
`\_fp_inf_fp:N . . . . . 16243, 16679`  
`\_fp_int:wTF . . . . . 16517, 22179`  
`\_fp_int_eval:w . . . . .`  
`.... 738, 752, 754, 754, 767, 783,`  
`820, 828, 828, 831, 835, 864, 16197,`  
`16257, 16332, 16463, 16466, 16852,`  
`16856, 16868, 16869, 16905, 16996,`  
`17000, 17039, 17253, 17258, 17300,`  
`17389, 17400, 17449, 17480, 17486,`  
`17487, 17533, 17543, 17545, 17561,`  
`17563, 17586, 17588, 17755, 17977,`  
`18021, 18221, 18512, 18982, 18990,`  
`19011, 19013, 19034, 19036, 19045,`  
`19047, 19076, 19082, 19092, 19094,`  
`19168, 19170, 19186, 19188, 19192,`  
`19208, 19248, 19256, 19258, 19260,`  
`19262, 19265, 19268, 19270, 19289,`  
`19291, 19301, 19303, 19329, 19332,`  
`19340, 19342, 19363, 19366, 19369,`  
`19372, 19381, 19384, 19387, 19390,`

19397, 19399, 19405, 19413, 19415,  
 19417, 19423, 19443, 19445, 19454,  
 19456, 19477, 19498, 19502, 19514,  
 19517, 19520, 19523, 19526, 19529,  
 19532, 19535, 19539, 19551, 19555,  
 19559, 19562, 19583, 19585, 19587,  
 19597, 19636, 19638, 19647, 19734,  
 19739, 19741, 19748, 19751, 19754,  
 19757, 19760, 19763, 19772, 19784,  
 19792, 19794, 19804, 19806, 19813,  
 19822, 19824, 19827, 19830, 19833,  
 19836, 19849, 19851, 19859, 19861,  
 19869, 19871, 19881, 19884, 19887,  
 19894, 19909, 19927, 19930, 19986,  
 20000, 20002, 20008, 20021, 20023,  
 20025, 20049, 20065, 20072, 20073,  
 20096, 20113, 20117, 20162, 20164,  
 20208, 20219, 20238, 20240, 20242,  
 20255, 20268, 20273, 20275, 20281,  
 20298, 20299, 20300, 20301, 20302,  
 20303, 20308, 20310, 20312, 20314,  
 20316, 20321, 20323, 20325, 20327,  
 20329, 20331, 20353, 20361, 20445,  
 20494, 20571, 20579, 20587, 20593,  
 20596, 20702, 20723, 20725, 20728,  
 20731, 20734, 20737, 20753, 20779,  
 20793, 20809, 20879, 20889, 20894,  
 21046, 21078, 21087, 21319, 21333,  
 21336, 21339, 21342, 21345, 21348,  
 21351, 21354, 21357, 21373, 21383,  
 21392, 21410, 21419, 21426, 21437,  
 21447, 21480, 21490, 21515, 21524,  
 21567, 21584, 21586, 21598, 21599,  
 21640, 21651, 21662, 21720, 21872,  
 21995, 22048, 22144, 22167, 22221,  
 22273, 22295, 22297, 22299, 22304,  
 22323, 22335, 22343, 22348, 22353  
 $\backslash\_{\text{fp\_int\_eval\_end}}:$  . . . . .  
 $\quad \underline{16197}, \underline{16257}, \underline{16335}, \underline{16454}, \underline{16905},$   
 $\quad \underline{17010}, \underline{17014}, \underline{18222}, \underline{18512}, \underline{19192},$   
 $\quad \underline{19227}, \underline{19419}, \underline{19794}, \underline{19930}, \underline{20753},$   
 $\quad \underline{20809}, \underline{21079}, \underline{21088}, \underline{21437}, \underline{21447},$   
 $\quad \underline{21490}, \underline{21515}, \underline{21599}, \underline{22302}, \underline{22304}$   
 $\backslash\_{\text{fp\_int\_p:w}}$  . . . . .  
 $\quad \underline{16517}$   
 $\backslash\_{\text{fp\_int\_to\_roman:w}}$  . . . . .  
 $\quad \underline{16197},$   
 $\quad \underline{16466}, \underline{17461}, \underline{17493}, \underline{20235}, \underline{22405}$   
 $\backslash\_{\text{fp\_invalid\_operation:nnw}}$  . . . . .  
 $\quad . \quad \underline{746}, \underline{16621}, \underline{16703}, \underline{16715}, \underline{21853},$   
 $\quad . \quad \underline{21860}, \underline{21907}, \underline{21914}, \underline{22014}, \underline{22029}$   
 $\backslash\_{\text{fp\_invalid\_operation\_o:nw}}$  . . . . .  
 $\quad . \quad \underline{746}, \underline{16714}, \underline{17729}, \underline{19467}, \underline{19693},$   
 $\quad . \quad \underline{20187}, \underline{20857}, \underline{20866}, \underline{20953}, \underline{20968},$   
 $\quad . \quad \underline{20983}, \underline{20998}, \underline{21013}, \underline{21028}, \underline{21690},$   
 $\quad . \quad \underline{21708}, \underline{21724}, \underline{21752}, \underline{21765}, \underline{21781}$   
 $\backslash\_{\text{fp\_invalid\_operation\_o:Nww}}$  . . . . .  
 $\quad . \quad \underline{746}, \underline{16629}, \underline{16703},$   
 $\quad . \quad \underline{17930}, \underline{18962}, \underline{19234}, \underline{19235}, \underline{20794}$   
 $\backslash\_{\text{fp\_invalid\_operation\_o:nww}}$  . . . . .  
 $\backslash\_{\text{fp\_invalid\_operation\_tl\_o:nn}}$  . . . . .  
 $\quad . \quad \underline{746}, \underline{16638}, \underline{16703}, \underline{16971}, \underline{22199}$   
 $\backslash\_{\text{fp\_kind:w}}$  . . . . .  
 $\quad \underline{16258}, \underline{16964}, \underline{18485}$   
 $\backslash\_{\text{c\_fp\_leading\_shift\_int}}$  . . . . .  
 $\quad . \quad \underline{16420}, \underline{19739},$   
 $\quad . \quad \underline{19748}, \underline{19822}, \underline{20723}, \underline{21373}, \underline{21410}$   
 $\backslash\_{\text{fp\_ln\_c:NwNw}}$   $\underline{871}, \underline{872}, \underline{20332}, \underline{20363}$   
 $\backslash\_{\text{fp\_ln\_div\_after:Nw}}$  . . . . .  
 $\quad . \quad \underline{870}, \underline{20234}, \underline{20283}$   
 $\backslash\_{\text{fp\_ln\_div\_i:w}}$  . . . . .  
 $\backslash\_{\text{fp\_ln\_div\_ii:wwn}}$  . . . . .  
 $\quad .. \quad \underline{20259}, \underline{20260}, \underline{20261}, \underline{20262}, \underline{20270}$   
 $\backslash\_{\text{fp\_ln\_div\_vi:wwn}}$  . . . . .  
 $\backslash\_{\text{fp\_ln\_exponent:wn}}$   $\underline{873}, \underline{20210}, \underline{20372}$   
 $\backslash\_{\text{fp\_ln\_exponent\_one:ww}}$   $\underline{20377}, \underline{20391}$   
 $\backslash\_{\text{fp\_ln\_exponent\_small:NNww}}$  . . . . .  
 $\quad .. \quad \underline{20380}, \underline{20384}, \underline{20397}$   
 $\backslash\_{\text{c\_fp\_ln\_i\_fixed\_tl}}$  . . . . .  
 $\backslash\_{\text{c\_fp\_ln\_ii\_fixed\_tl}}$  . . . . .  
 $\backslash\_{\text{c\_fp\_ln\_iii\_fixed\_tl}}$  . . . . .  
 $\backslash\_{\text{c\_fp\_ln\_iv\_fixed\_tl}}$  . . . . .  
 $\backslash\_{\text{c\_fp\_ln\_ix\_fixed\_tl}}$  . . . . .  
 $\backslash\_{\text{fp\_ln\_npos\_o:w}}$  . . . . .  
 $\quad .. \quad \underline{865}, \underline{866}, \underline{20196}, \underline{20198}$   
 $\backslash\_{\text{fp\_ln\_o:w}}$  . . . . .  
 $\quad \underline{865}, \underline{881}, \underline{20172}, \underline{20184}$   
 $\backslash\_{\text{fp\_ln\_significand:NNNNnnnnN}}$  . . . . .  
 $\quad .. \quad \underline{867}, \underline{20209}, \underline{20212}, \underline{20703}$   
 $\backslash\_{\text{fp\_ln\_square\_t\_after:w}}$  . . . . .  
 $\quad .. \quad \underline{20307}, \underline{20339}$   
 $\backslash\_{\text{fp\_ln\_square\_t\_pack:NNNNNw}}$  . . . . .  
 $\quad .. \quad \underline{20309}, \underline{20311}, \underline{20313}, \underline{20315}, \underline{20337}$   
 $\backslash\_{\text{fp\_ln\_t\_large:NNw}}$  . . . . .  
 $\quad .. \quad \underline{870}, \underline{20288}, \underline{20295}, \underline{20305}$   
 $\backslash\_{\text{fp\_ln\_t\_small:Nw}}$  . . . . .  
 $\quad .. \quad \underline{20286}, \underline{20293}$   
 $\backslash\_{\text{fp\_ln\_t\_small:w}}$  . . . . .  
 $\quad .. \quad \underline{870}$   
 $\backslash\_{\text{fp\_ln\_Taylor:wwNw}}$   $\underline{871}, \underline{20340}, \underline{20341}$   
 $\backslash\_{\text{fp\_ln\_Taylor\_break:w}}$   $\underline{20346}, \underline{20357}$   
 $\backslash\_{\text{fp\_ln\_Taylor\_loop:www}}$  . . . . .  
 $\quad .. \quad \underline{20342}, \underline{20343}, \underline{20352}$   
 $\backslash\_{\text{fp\_ln\_twice\_t\_after:w}}$   $\underline{20320}, \underline{20336}$   
 $\backslash\_{\text{fp\_ln\_twice\_t\_pack:Nw}}$  . . . . .  
 $\quad .. \quad \underline{20322},$   
 $\quad .. \quad \underline{20324}, \underline{20326}, \underline{20328}, \underline{20330}, \underline{20335}$   
 $\backslash\_{\text{c\_fp\_ln\_vi\_fixed\_tl}}$  . . . . .  
 $\backslash\_{\text{c\_fp\_ln\_vii\_fixed\_tl}}$  . . . . .  
 $\backslash\_{\text{c\_fp\_ln\_viii\_fixed\_tl}}$  . . . . .  
 $\backslash\_{\text{c\_fp\_ln\_x\_fixed\_tl}}$  . . . . .  
 $\quad .. \quad \underline{20175}, \underline{20394}, \underline{20401}$   
 $\backslash\_{\text{fp\_ln\_x\_ii:wnnnn}}$  . . . . .  
 $\quad .. \quad \underline{20214}, \underline{20232}$   
 $\backslash\_{\text{fp\_ln\_x\_iii:NNNNNNw}}$  . . . . .  
 $\quad .. \quad \underline{20241}, \underline{20245}$

`\__fp_ln_x_iii_var:NNNNNw . . . . .` 20239, 20247  
`\__fp_ln_x_iv:wnnnnnnn . . . . .` 869, 20237, 20252  
`\__fp_logb_aux_o:w . . . . .` 19650  
`\__fp_logb_o:w . . . . .` 18908, 19650  
`\c_fp_max_exp_exponent_int . . . . .` 16236, 20437  
`\c_fp_max_exponent_int . . . . .` 16234, 16240, 16268, 19947, 20149, 20758  
`\c_fp_middle_shift_int . . . . .` 16420, 19751, 19754, 19757, 19760, 19824, 19827, 19830, 19833, 20725, 20728, 20731, 20734, 21376, 21383, 21413, 21419  
`\__fp_minmax_aux_o:Nw . . . . .` 18774  
`\__fp_minmax_auxi:ww . . . . .` 18796, 18808, 18815  
`\__fp_minmax_auxii:ww . . . . .` 18798, 18806, 18815  
`\__fp_minmax_break_o:w . .` 18789, 18819  
`\__fp_minmax_loop:Nww . . . . .` 813, 18783, 18785, 18791  
`\__fp_minmax_o:Nw . . . . .` 806, 18478, 18480, 18774  
`\c_fp_minus_min_exponent_int . . . . .` 16234, 16269  
`\__fp_misused:n . .` 16210, 16214, 16325  
`\__fp_mul_cases_o:NnNww . . . . .` 830, 19199, 19205, 19311  
`\__fp_mul_cases_o:nNnnww . . . . .` 19205  
`\__fp_mul_npos_o:Nww . . . . .` 826, 828, 830, 919, 19202, 19243, 22065  
`\__fp_mul_significand_drop:NNNNNw . . . . .` 828, 19252  
`\__fp_mul_significand_keep:NNNNNw . . . . .` 19252  
`\__fp_mul_significand_large_-f:NwwNNNN . . . . .` 19282, 19286  
`\__fp_mul_significand_o:nnnnNnnn . . . . .` 828, 828, 19250, 19252  
`\__fp_mul_significand_small_-f:NNwwN . . . . .` 19280, 19297  
`\__fp_mul_significand_test_f:NNN . . . . .` 829, 19254, 19277  
`\c_fp_myriad_int . . . . .` 16233, 19734, 19765, 19766, 19843, 19904  
`\__fp_neg_sign:N . . . . .` 818, 16256, 18916, 19069  
`\__fp_not_o:w . . . . .` 806, 17748, 18821  
`\c_fp_one_fixed_t1 . . . . .` 19728, 20348, 20561, 20759, 20786, 21569, 21636, 21741, 22218, 22228, 22269  
`\__fp_overflow:w . . . . .` 733, 746, 748, 16271, 16703, 20439, 20873  
`\c_fp_overflowing_fp . . . . .` 16237, 21854, 21908  
`\__fp_pack:NNNNNw . . . . .` 16420, 19740, 19750, 19753, 19756, 19759, 19762, 19823, 19826, 19829, 19832, 19835, 20724, 20727, 20730, 20733, 20736  
`\__fp_pack_big:NNNNNNw . . . . .` 16424, 19516, 19519, 19522, 19525, 19528, 19531, 19534, 19538, 19850, 19860, 19870, 19880, 19883, 19886, 19893  
`\__fp_pack_Bigg:NNNNNNw . . . . .` 16429, 19365, 19368, 19371, 19383, 19386, 19389  
`\__fp_pack_eight:wNNNNNNNN . . . . .` 739, 824, 16436, 19178, 19487, 19915, 21065, 21066  
`\__fp_pack_twice_four:wNNNNNNNN . . . . .` 739, 16434, 17021, 17022, 19120, 19121, 19916, 19917, 19918, 19950, 19951, 19952, 20140, 20141, 20476, 20477, 20478, 21067, 21068, 21362, 21363, 21364, 21365, 22058  
`\__fp_parse:n . . . . .` 759, 770, 782, 790, 803, 804, 811, 920, 920, 930, 17052, 17203, 17865, 18421, 18423, 18425, 18448, 18485, 18494, 18511, 18521, 18678, 18738, 19663, 21829, 21883, 21961, 22006, 22021, 22074, 22076, 22078, 22080, 22457  
`\__fp_parse_after:ww . . . . .` 17865  
`\__fp_parse_apply_binary:NwNwN . . . . .` 763, 767, 767, 795, 17903, 18098  
`\__fp_parse_apply_binary_chk:NN . . . . .` 17903, 17934, 17947  
`\__fp_parse_apply_binary_error:NNN . . . . .` 17903  
`\__fp_parse_apply_comma:NwNwN . . . . .` 795, 18057  
`\__fp_parse_apply_compare:NwNNNNNwN . . . . .` 18245, 18254  
`\__fp_parse_apply_compare_aux:NNwN . . . . .` 18266, 18269, 18274  
`\__fp_parse_apply_function:NNNwN . . . . .` 786, 17697, 17858  
`\__fp_parse_apply_unary:NNNwN . . . . .` 17702, 17734, 17849  
`\__fp_parse_apply_unary_chk:nNNNNw . . . . .` 17713, 17714, 17717  
`\__fp_parse_apply_unary_chk:nNNNw . . . . .` 17702  
`\__fp_parse_apply_unary_chk:NwNw . . . . .` 17702

`\_fp_parse_apply_unary_error>NNw` . . . . . [17702](#), [19700](#)  
`\_fp_parse_apply_unary_type>NNN` . . . . . [17702](#)  
`\_fp_parse_caseless_inf:N` . . . [17815](#)  
`\_fp_parse_caseless_infinity:N` . . . . . [17815](#)  
`\_fp_parse_caseless_nan:N` . . . [17815](#)  
`\_fp_parse_compare>NNNNNNN` . . [18186](#)  
`\_fp_parse_compare_auxi>NNNNNNN` . . . . . [18186](#)  
`\_fp_parse_compare_auxii>NNNNN` . . . . . [18186](#)  
`\_fp_parse_compare_end>NNNNw` . [18186](#)  
`\_fp_parse_continue>NwN` . . . . .  
 . . . . . [763](#), [764](#), [791](#), [17892](#), [17905](#),  
[18085](#), [18284](#), [18878](#), [18888](#), [18896](#)  
`\_fp_parse_continue_compare>NNwNN` . . . . . [18277](#), [18292](#)  
`\_fp_parse_digits:N` . . . . . [17070](#)  
`\_fp_parse_digits_i:N` . . . . . [17070](#)  
`\_fp_parse_digits_ii:N` . . . . . [17070](#)  
`\_fp_parse_digits_iii:N` . . . . . [17070](#)  
`\_fp_parse_digits_iv:N` . . . . . [17070](#)  
`\_fp_parse_digits_v:N` . . . . . [17070](#)  
`\_fp_parse_digits_vi:N` . . . . .  
 . . . . . [17070](#), [17405](#), [17453](#)  
`\_fp_parse_digits_vii:N` . . . . .  
 . . . . . [776](#), [17070](#), [17392](#), [17442](#)  
`\_fp_parse_excl_error:` . . . . . [18186](#)  
`\_fp_parse_expand:w` . . . . .  
 . . . . . [766](#), [766](#), [767](#), [767](#), [17067](#), [17069](#),  
[17079](#), [17119](#), [17179](#), [17223](#), [17232](#),  
[17235](#), [17239](#), [17276](#), [17310](#), [17348](#),  
[17350](#), [17369](#), [17371](#), [17393](#), [17410](#),  
[17423](#), [17443](#), [17473](#), [17501](#), [17517](#),  
[17528](#), [17551](#), [17580](#), [17590](#), [17597](#),  
[17611](#), [17627](#), [17647](#), [17658](#), [17744](#),  
[17767](#), [17779](#), [17854](#), [17863](#), [17871](#),  
[17884](#), [18004](#), [18052](#), [18076](#), [18102](#),  
[18150](#), [18170](#), [18239](#), [18252](#), [18874](#)  
`\_fp_parse_exponent:N` . . . . .  
 . . . . . [780](#), [17178](#), [17384](#), [17533](#), [17600](#), [17602](#)  
`\_fp_parse_exponent:Nw` . . . . .  
 . . . . . [17408](#), [17421](#),  
[17470](#), [17498](#), [17549](#), [17578](#), [17597](#)  
`\_fp_parse_exponent_aux>NN` . . [17602](#)  
`\_fp_parse_exponent_body:N` . . . . .  
 . . . . . [17629](#), [17633](#)  
`\_fp_parse_exponent_digits:N` . . . . .  
 . . . . . [17637](#), [17649](#)  
`\_fp_parse_exponent_keep:N` . . [17660](#)  
`\_fp_parse_exponent_keep:NTF` . . . . .  
 . . . . . [17640](#), [17660](#)  
`\_fp_parse_exponent_sign:N` . . . . .  
 . . . . . [17619](#), [17623](#)  
`\_fp_parse_function>NNN` . . . . .  
 . . . . . [16764](#), [16766](#), [16768](#),  
[16771](#), [17847](#), [18478](#), [18480](#), [20936](#),  
[20938](#), [20940](#), [20942](#), [22103](#), [22105](#)  
`\_fp_parse_function_all_fp_o:nw` . . . . . [16898](#), [18294](#), [18776](#)  
`\_fp_parse_function_one_two:nw` . . . . .  
 . . . . . [904](#), [18306](#), [21530](#), [21536](#), [22172](#)  
`\_fp_parse_function_one_two_-aux:nw` . . . . . [18306](#)  
`\_fp_parse_function_one_two_-auxii:nw` . . . . . [18306](#)  
`\_fp_parse_function_one_two_-error_o:w` . . . . . [18306](#)  
`\_fp_parse_infix>NN` . . . . .  
 . . . . . [769](#), [772](#), [789](#), [793](#),  
[794](#), [17118](#), [17288](#), [17327](#), [17807](#),  
[17822](#), [17844](#), [17960](#), [17963](#), [18050](#)  
`\_fp_parse_infix_!N` . . . . . [18186](#)  
`\_fp_parse_infix_&Nw` . . . . . [18143](#)  
`\_fp_parse_infix_(N` . . . . . [18126](#)  
`\_fp_parse_infix_):N` . . . . . [18040](#)  
`\_fp_parse_infix_*:N` . . . . . [18128](#)  
`\_fp_parse_infix_+:N` . . . . .  
 . . . . . [767](#), [17067](#), [18092](#)  
`\_fp_parse_infix_,:N` . . . . . [18057](#)  
`\_fp_parse_infix_-:N` . . . . . [18092](#)  
`\_fp_parse_infix_/:N` . . . . . [18092](#)  
`\_fp_parse_infix_::N` . . . . . [18160](#), [18859](#)  
`\_fp_parse_infix_<:N` . . . . . [18186](#)  
`\_fp_parse_infix_=:N` . . . . . [18186](#)  
`\_fp_parse_infix_>:N` . . . . . [18186](#)  
`\_fp_parse_infix_?:N` . . . . . [18160](#)  
`\_fp_parse_infix_{operation}:N` . . . . . [767](#)  
`\_fp_parse_infix_^:N` . . . . . [18092](#)  
`\_fp_parse_infix_after_operand:NwN` . . . . .  
 . . . . . [772](#), [17171](#), [17249](#), [17751](#), [17958](#)  
`\_fp_parse_infix_after_paren>NN` . . . . .  
 . . . . . [17776](#), [17802](#), [18007](#)  
`\_fp_parse_infix_and:N` . . . . . [18092](#), [18159](#)  
`\_fp_parse_infix_check>NNN` . . . . .  
 . . . . . [17983](#), [17993](#), [18027](#)  
`\_fp_parse_infix_comma:w` . . . . . [795](#), [18057](#)  
`\_fp_parse_infix_end:N` . . . . .  
 . . . . . [790](#), [794](#), [17872](#), [17877](#), [17885](#), [18038](#)  
`\_fp_parse_infix_juxt:N` . . . . .  
 . . . . . [793](#), [17973](#), [17981](#), [18092](#)  
`\_fp_parse_infix_mark>NNN` . . . . .  
 . . . . . [17970](#), [18014](#), [18037](#)  
`\_fp_parse_infix_mul:N` . . . . .  
 . . . . . [793](#), [797](#), [17998](#),  
[18017](#), [18025](#), [18092](#), [18127](#), [18136](#)

`\_fp_parse_infix_or:N . 18092, 18158`  
`\_fp_parse_infix_|:Nw . . . . . 18143`  
`\_fp_parse_large:N 775, 17355, 17438`  
`\_fp_parse_large_leading:wwNN . . . . . 778, 17440, 17445`  
`\_fp_parse_large_round:NN . . . . . 779, 17481, 17553`  
`\_fp_parse_large_round_aux:wNN . . . . . 17553`  
`\_fp_parse_large_round_test:NN . . . . . 17553`  
`\_fp_parse_large_trailing:wnNN . . . . . 779, 17451, 17475`  
`\_fp_parse_letters:N . . . . . 772, 773, 17264, 17278`  
`\_fp_parse_lparen_after:NwN . 17757`  
`\_fp_parse_o:n . . . . . 759, 17865, 18676, 18677`  
`\_fp_parse_one:Nw . . . . . 762–765, 767, 774, 789, 791, 17067, 17090, 17332, 17696, 17898`  
`\_fp_parse_one_digit>NN . . . . . 787, 17106, 17247`  
`\_fp_parse_one_fp>NN . . . . . 768, 17098, 17114`  
`\_fp_parse_one_other>NN 17109, 17255`  
`\_fp_parse_one_register>NN . . . . . 17101, 17169`  
`\_fp_parse_one_register_aux:Nw . . . . . 17169`  
`\_fp_parse_one_register_- auxii:wwwNw . . . . . 17169`  
`\_fp_parse_one_register_dim:ww . . . . . 17169`  
`\_fp_parse_one_register_int:www . . . . . 17169`  
`\_fp_parse_one_register_- math:NNw . . . . . 17210`  
`\_fp_parse_one_register_mu:www . . . . . 17169`  
`\_fp_parse_one_register_- special:N . . . . . 17174, 17210`  
`\_fp_parse_one_register_wd:Nw 17210`  
`\_fp_parse_one_register_wd:w . 17210`  
`\_fp_parse_operand:Nw . . . . . 762–765, 766, 790, 795, 17067, 17740, 17742, 17763, 17765, 17854, 17863, 17870, 17883, 17892, 18075, 18101, 18169, 18252, 18873`  
`\_fp_parse_pack_carry:w . 777, 17425`  
`\_fp_parse_pack_leading:NNNNNww . . . . . 17388, 17425, 17448`  
`\_fp_parse_pack_trailing:NNNNNNww . . . . . 17398, 17425, 17467, 17478, 17485`  
`\_fp_parse_prefix:NNN . 17267, 17312`  
`\_fp_parse_prefix_!Nw . . . . . 17730`  
`\_fp_parse_prefix_(:Nw . . . . . 17757`  
`\_fp_parse_prefix_):Nw . . . . . 17789`  
`\_fp_parse_prefix_+:Nw . . . . . 17696`  
`\_fp_parse_prefix_-=Nw . . . . . 17730`  
`\_fp_parse_prefix_.:Nw . . . . . 17749`  
`\_fp_parse_prefix_unknown:NNN 17312`  
`\_fp_parse_return_semicolon:w . . . . . 17068, 17077, 17308, 17515, 17526, 17609, 17641, 17656`  
`\_fp_parse_round:Nw . . . . . 16769`  
`\_fp_parse_round_after:wN . . . . . 781, 17530, 17535, 17585`  
`\_fp_parse_round_loop:N . . . . . 780, 781, 17503, 17546, 17564, 17589`  
`\_fp_parse_round_up:N . . . . . 17503`  
`\_fp_parse_small:N 775, 17375, 17386`  
`\_fp_parse_small_leading:wwNN . . . . . 776, 17390, 17395, 17457`  
`\_fp_parse_small_round>NN . . . . . 17417, 17535, 17574`  
`\_fp_parse_small_trailing:wwNN . . . . . 777, 17403, 17412, 17489`  
`\_fp_parse_strim_end:w . . . . . 17361`  
`\_fp_parse_strim_zeros:N . . . . . 775, 787, 17342, 17361, 17755`  
`\_fp_parse_trim_end:w . . . . . 17335`  
`\_fp_parse_trim_zeros:N 17253, 17335`  
`\_fp_parse_unary_function:NNN . . . . . 17847, 18906, 18908, 18910, 18912, 20170, 20172, 20174, 20924, 20930`  
`\_fp_parse_word:Nw 772, 17261, 17278`  
`\_fp_parse_word_abs:N . . . . . 18905`  
`\_fp_parse_word_acos:N . . . . . 20916`  
`\_fp_parse_word_acosd:N . . . . . 20916`  
`\_fp_parse_word_acot:N . . . . . 20935`  
`\_fp_parse_word_acotd:N . . . . . 20935`  
`\_fp_parse_word_acsc:N . . . . . 20916`  
`\_fp_parse_word_acscd:N . . . . . 20916`  
`\_fp_parse_word_asec:N . . . . . 20916`  
`\_fp_parse_word_asecd:N . . . . . 20916`  
`\_fp_parse_word_asin:N . . . . . 20916`  
`\_fp_parse_word_asind:N . . . . . 20916`  
`\_fp_parse_word_atan:N . . . . . 20935`  
`\_fp_parse_word_atand:N . . . . . 20935`  
`\_fp_parse_word_bp:N . . . . . 17818`  
`\_fp_parse_word_cc:N . . . . . 17818`  
`\_fp_parse_word_ceil:N . . . . . 16763`  
`\_fp_parse_word_cm:N . . . . . 17818`  
`\_fp_parse_word_cos:N . . . . . 20916`  
`\_fp_parse_word_cosd:N . . . . . 20916`  
`\_fp_parse_word_cot:N . . . . . 20916`  
`\_fp_parse_word_cotd:N . . . . . 20916`

`\_fp_parse_word_csc:N` ..... 20916  
`\_fp_parse_word_cscd:N` ..... 20916  
`\_fp_parse_word_dd:N` ..... 17818  
`\_fp_parse_word_deg:N` ..... 17804  
`\_fp_parse_word_em:N` ..... 17837  
`\_fp_parse_word_ex:N` ..... 17837  
`\_fp_parse_word_exp:N` ..... 20169  
`\_fp_parse_word_fact:N` ..... 20169  
`\_fp_parse_word_false:N` ..... 17804  
`\_fp_parse_word_floor:N` ..... 16763  
`\_fp_parse_word_in:N` ..... 17818  
`\_fp_parse_word_inf:N` .....  
..... 17804, 17815, 17816  
`\_fp_parse_word_ln:N` ..... 20169  
`\_fp_parse_word_logb:N` ..... 18905  
`\_fp_parse_word_max:N` ..... 18477  
`\_fp_parse_word_min:N` ..... 18477  
`\_fp_parse_word_mm:N` ..... 17818  
`\_fp_parse_word_nan:N` ..... 17804, 17817  
`\_fp_parse_word_nc:N` ..... 17818  
`\_fp_parse_word_nd:N` ..... 17818  
`\_fp_parse_word_pc:N` ..... 17818  
`\_fp_parse_word_pi:N` ..... 17804  
`\_fp_parse_word_pt:N` ..... 17818  
`\_fp_parse_word_rand:N` ..... 22102  
`\_fp_parse_word_randint:N` ..... 22102  
`\_fp_parse_word_round:N` ..... 16769  
`\_fp_parse_word_sec:N` ..... 20916  
`\_fp_parse_word_secd:N` ..... 20916  
`\_fp_parse_word_sign:N` ..... 18905  
`\_fp_parse_word_sin:N` ..... 20916  
`\_fp_parse_word_sind:N` ..... 20916  
`\_fp_parse_word_sp:N` ..... 17818  
`\_fp_parse_word_sqrt:N` ..... 18905  
`\_fp_parse_word_tan:N` ..... 20916  
`\_fp_parse_word_tand:N` ..... 20916  
`\_fp_parse_word_true:N` ..... 17804  
`\_fp_parse_word_trunc:N` ..... 16763  
`\_fp_parse_zero:` .....  
..... 775, 17357, 17377, 17381  
`\_fp_pow_B:wwN` ..... 20706, 20741  
`\_fp_pow_C_neg:w` ..... 20744, 20761  
`\_fp_pow_C_overflow:w` .....  
..... 20749, 20756, 20777  
`\_fp_pow_C_pack:w` ..... 20763, 20771, 20782  
`\_fp_pow_C_pos:w` ..... 20747, 20766  
`\_fp_pow_C_pos_loop:wN` .....  
..... 20767, 20768, 20775  
`\_fp_pow_exponent:Nwnnnnnw` .....  
..... 20712, 20715, 20720  
`\_fp_pow_exponent:wnN` ..... 20704, 20709  
`\_fp_pow_neg:www` ..... 883, 20617, 20788  
`\_fp_pow_neg_aux:wNN` ..... 883, 20788  
`\_fp_pow_neg_case:w` ..... 20790, 20811  
`\_fp_pow_neg_case_aux:nnnnn` ..... 20811  
`\_fp_pow_neg_case_aux:Nnnw` .....  
..... 884, 20811  
`\_fp_pow_normal_o:ww` .....  
..... 879, 20622, 20654  
`\_fp_pow_npos_aux:NNnww` .....  
..... 20689, 20693, 20699  
`\_fp_pow_npos_o:Nww` ..... 880, 20666, 20683  
`\_fp_pow_zero_or_inf:ww` .....  
..... 879, 20624, 20631  
`\c_fp_prec_and_int` ..... 17052, 18123  
`\c_fp_prec_colon_int` .....  
..... 17052, 18181, 18873  
`\c_fp_prec_comma_int` .....  
..... 787, 17052, 17126,  
17763, 17791, 18061, 18066, 18075  
`\c_fp_prec_comp_int` .....  
..... 17052, 18209, 18252  
`\c_fp_prec_end_int` ..... 790,  
794, 17052, 17128, 17870, 17883, 18044  
`\c_fp_prec_func_int` .....  
..... 787, 17052, 17762, 17854, 17863  
`\c_fp_prec_hat_int` ..... 17052, 18111  
`\c_fp_prec_hatii_int` ..... 17052, 18111  
`\c_fp_prec_int` .....  
..... 16230, 16463, 16524, 16551, 16990,  
20457, 20823, 20826, 21926, 21928,  
21934, 21985, 22183, 22222, 22273  
`\c_fp_prec_juxtap_int` ..... 17052, 18113  
`\c_fp_prec_not_int` .....  
..... 787, 17052, 17747, 17748  
`\c_fp_prec_or_int` ..... 17052, 18125  
`\c_fp_prec_plus_int` .....  
..... 761, 17052, 18119, 18121  
`\c_fp_prec_quest_int` .....  
..... 17052, 18164, 18179  
`\c_fp_prec_times_int` .....  
..... 17052, 18115, 18117  
`\c_fp_prec_tuple_int` .....  
..... 787, 17052, 17127, 17765, 17793  
`\_fp_rand_myriads:n` .....  
..... 924, 925, 22138, 22155, 22241  
`\_fp_rand_myriads_get:w` ..... 22138  
`\_fp_rand_myriads_loop:w` ..... 22138  
`\_fp_rand_o:Nw` .....  
..... 22103, 22110, 22116, 22149  
`\_fp_rand_o:w` ..... 22149  
`\_fp_randinat_wide_aux:w` ..... 22311  
`\_fp_randinat_wide_auxii:w` ..... 22311  
`\_fp_randint:n` ..... 22373  
`\_fp_randint:ww` ..... 22279, 22383  
`\_fp_randint_auxi_o:ww` ..... 22170  
`\_fp_randint_auxii:wn` ..... 22170  
`\_fp_randint_auxii_o:ww` ..... 22170

```

__fp_randint_auxiv_o:ww 22170
__fp_randint_auxv_o:w 22170
__fp_randint_badarg:w ... 925, 22170
__fp_randint_default:w 22170
__fp_randint_o:Nw 22105, 22116, 22170
__fp_randint_o:w 22170
__fp_randint_split_aux:w ... 22311
__fp_randint_split_o:Nw . 928, 22311
__fp_randint_wide_aux:w
..... 928, 22314, 22345
__fp_randint_wide_auxii:w
..... 22347, 22356
__fp_reverse_args:Nww
..... 910, 911, 16206,
21516, 21591, 21704, 21770, 22267
__fp_round:NNN 752, 752, 754, 829,
845, 16779, 16849, 19037, 19048,
19292, 19304, 19446, 19457, 19641
__fp_round:Nwn . 16907, 16960, 22033
__fp_round:Nww . 16908, 16929, 16960
__fp_round:Nwww 16909, 16923
__fp_round_aux_o:Nw 16896
__fp_round_digit:Nw .. 741, 754,
828, 829, 845, 16481, 16863, 19051,
19194, 19295, 19307, 19460, 19646
__fp_round_name_from_cs:N
.. 16899, 16919, 16945, 16949, 16972
__fp_round_neg:NNN 752,
754, 825, 16874, 19156, 19171, 19189
__fp_round_no_arg_o:Nw 16906, 16913
__fp_round_normal:NnnwNNnn .. 16960
__fp_round_normal:NNwNnn .. 16960
__fp_round_normal:NwNNnw .. 16960
__fp_round_normal_end:wwNnn .. 16960
__fp_round_o:Nw
.. 16764, 16766, 16768, 16772, 16896
__fp_round_pack:Nw 16960
__fp_round_return_one:
..... 752, 16779, 16785,
16795, 16803, 16807, 16816, 16820,
16829, 16836, 16840, 16878, 16889
__fp_round_s>NNNw
.. 752, 754, 781, 16847, 17539, 17557
__fp_round_special:NwwNnn .. 16960
__fp_round_special_aux:Nw .. 16960
__fp_round_to_nearest:NNN
..... 755, 756, 16772, 16775,
16779, 16883, 16915, 16925, 22033
__fp_round_to_nearest_neg:NNN 16874
__fp_round_to_nearest_ninf:NNN ..
..... 756, 16779, 16894
__fp_round_to_nearest_ninf_-
neg:NNN 16874
__fp_round_to_nearest_pinf:NNN .
..... 756, 16779, 16885
__fp_round_to_nearest_pinf_-
neg:NNN 16874
__fp_round_to_nearest_zero:NNN ..
..... 756, 16779
__fp_round_to_nearest_zero_-
neg:NNN 16874
__fp_round_to_ninf:NNN
..... 16766, 16779, 16882, 16953
__fp_round_to_ninf_neg:NNN .. 16874
__fp_round_to_pinf:NNN
..... 16768, 16779, 16874, 16955
__fp_round_to_pinf_neg:NNN .. 16874
__fp_round_to_zero:NNN
..... 16764, 16779, 16951
__fp_round_to_zero_neg:NNN .. 16874
__fp_rrrot:www 16207, 21637
__fp_SANITIZE:Nw
..... 820, 822, 828, 830, 839,
885, 901, 908, 925, 16265, 17029,
17047, 18980, 19074, 19246, 19327,
19475, 20200, 20443, 20685, 20877,
21478, 21522, 21649, 22165, 22260
__fp_SANITIZE:wN
..... 772, 776, 16265, 17252, 17754
__fp_SANITIZE_zero:w 16265
__fp_sec_o:w 20988
__fp_set_sign_o:w
.. 17747, 18906, 19677, 19678, 19699
__fp_show>NN 18453
__fp_sign_aux_o:w 19666
__fp_sign_o:w 18910, 19666
__fp_sin_o:w 745, 786, 786, 909, 20943
__fp_sin_series_aux_o>NNnwww .. 21430
__fp_sin_series_o>NNwww .. 888,
902, 20949, 20964, 20979, 20994, 21430
__fp_small_int:wTF
..... 884, 16533, 16962, 20864
__fp_small_int_normal:NnwTF .. 16533
__fp_small_int_test:NnnwNTF .. 16533
__fp_small_int_test:NnnwNw
..... 16552, 16555
__fp_small_int_true:wTF .. 16533
__fp_sqrt_auxi_o>NNNNwnnN
..... 19497, 19505
__fp_sqrt_auxii_o:NnnnnnnnN
840, 842, 19507, 19511, 19591, 19603
__fp_sqrt_auxiii_o:wnnnnnnnn .. .
..... 19508, 19546, 19592
__fp_sqrt_auxiv_o>NNNNNw 19546
__fp_sqrt_auxix_o:wnwnw 19580
__fp_sqrt_auxv_o>NNNNNw 19546
__fp_sqrt_auxvi_o>NNNNNw 19546

```

`\_fp_sqrt_auxvii_o:NNNNNw . . . . .` [19546](#)  
`\_fp_sqrt_auxviii_o:nnnnnnn . . . . .` [19568](#), [19570](#), [19572](#), [19578](#), [19580](#)  
`\_fp_sqrt_auxx_o:Nnnnnnnn . . . . .` [19576](#), [19594](#)  
`\_fp_sqrt_auxxi_o:wwnnN . . . . .` [19594](#)  
`\_fp_sqrt_auxxii_o:nnnnnnnnw . . . . .` [19604](#), [19608](#)  
`\_fp_sqrt_auxxiii_o:w . . . . .` [19608](#)  
`\_fp_sqrt_auxxiv_o:wnnnnnnnN . . . . .` [19620](#), [19623](#), [19631](#), [19633](#)  
`\_fp_sqrt_Newton_o:wwn . . . . .` [840](#), [19482](#), [19493](#), [19494](#)  
`\_fp_sqrt_npos_auxi_o:wwnnN . . . . .` [19473](#)  
`\_fp_sqrt_npos_auxii_o:wNNNNNNNN . . . . .` [19473](#)  
`\_fp_sqrt_npos_o:w . . . . .` [19470](#), [19473](#)  
`\_fp_sqrt_o:w . . . . .` [18912](#), [19463](#)  
`\_fp_step:NNnnnn . . . . .` [18743](#)  
`\_fp_step:NnnnnN . . . . .` [18673](#)  
`\_fp_step:wwwN . . . . .` [18673](#)  
`\_fp_step_fp:wwwN . . . . .` [18673](#)  
`\_fp_str_if_eq:nn . . . . .` [16570](#),  
[17665](#), [17679](#), [17967](#), [18011](#), [20657](#)  
`\_fp_sub_back_far_o:NnnnnnnnN . . . . .` [824](#), [19083](#), [19129](#)  
`\_fp_sub_back_near_after:wnnnnn . . . . .` [19089](#), [19167](#)  
`\_fp_sub_back_near_o:nnnnnnnnN . . . . .` [823](#), [19079](#), [19089](#)  
`\_fp_sub_back_near_pack:NNNNNNw . . . . .` [19089](#), [19169](#)  
`\_fp_sub_back_not_far_o:wwwNN . . . . .` [19144](#), [19164](#)  
`\_fp_sub_back_quite_far_ii>NN . . . . .` [19148](#)  
`\_fp_sub_back_quite_far_o:wwNN . . . . .` [19142](#), [19148](#)  
`\_fp_sub_back_shift:wnnnn . . . . .` [823](#), [19101](#), [19105](#)  
`\_fp_sub_back_shift_ii:ww . . . . .` [19105](#)  
`\_fp_sub_back_shift_iii:NNNNNNNNw . . . . .` [19105](#)  
`\_fp_sub_back_shift_iv:nnnnw . . . . .` [19105](#)  
`\_fp_sub_back_very_far_ii_-o:nnNwwNN . . . . .` [19176](#)  
`\_fp_sub_back_very_far_o:wwwNN . . . . .` [19143](#), [19176](#)  
`\_fp_sub_eq_o:NnwNw . . . . .` [19054](#)  
`\_fp_sub_npos_i_o:NnwNw . . . . .` [822](#), [19059](#), [19068](#), [19072](#)  
`\_fp_sub_npos_ii_o:NnwNw . . . . .` [19054](#)  
`\_fp_sub_npos_o:NnwNw . . . . .` [822](#), [18974](#), [19054](#)  
`\_fp_tan_o:w . . . . .` [21003](#)  
`\_fp_tan_series_aux_o:Nnwww . . . . .` [21484](#)  
`\_fp_tan_series_o:NNwwww . . . . .` [889](#), [890](#), [21010](#), [21025](#), [21484](#)  
`\_fp_ternary:NwwN . . . . .` [806](#), [18179](#), [18857](#)  
`\_fp_ternary_auxi:NwwN . . . . .` [806](#), [816](#), [18857](#)  
`\_fp_ternary_auxii:NwwN . . . . .` [806](#), [816](#), [18181](#), [18857](#)  
`\_fp_tmp:w . . . . .` [741](#), [796](#),  
[16475](#), [16485](#), [16486](#), [16487](#), [16488](#),  
[16489](#), [16490](#), [16491](#), [16492](#), [16493](#),  
[16494](#), [16495](#), [16496](#), [16497](#), [16498](#),  
[16499](#), [16500](#), [16576](#), [16578](#), [17070](#),  
[17082](#), [17083](#), [17084](#), [17085](#), [17086](#),  
[17087](#), [17088](#), [17146](#), [17168](#), [17730](#),  
[17747](#), [17748](#), [17804](#), [17809](#), [17810](#),  
[17811](#), [17812](#), [17813](#), [17814](#), [17818](#),  
[17826](#), [17827](#), [17828](#), [17829](#), [17830](#),  
[17831](#), [17832](#), [17833](#), [17834](#), [17835](#),  
[17836](#), [18040](#), [18056](#), [18057](#), [18080](#),  
[18092](#), [18110](#), [18112](#), [18114](#), [18116](#),  
[18118](#), [18120](#), [18122](#), [18124](#), [18128](#),  
[18142](#), [18143](#), [18158](#), [18159](#), [18160](#),  
[18178](#), [18180](#), [19709](#), [19723](#), [19724](#)  
`\_fp_to_decimal:w . . . . .` [21888](#), [21898](#), [22015](#), [22032](#), [22519](#)  
`\_fp_to_decimal_dispatch:w . . . . .` [914](#),  
[917](#), [918](#), [18736](#), [21878](#), [21882](#), [21885](#)  
`\_fp_to_decimal_huge:wnnnn . . . . .` [21898](#)  
`\_fp_to_decimal_large:Nnnw . . . . .` [21898](#)  
`\_fp_to_decimal_normal:wnnnnn . . . . .` [21898](#), [21986](#)  
`\_fp_to_decimal_recover:w . . . . .` [21885](#)  
`\_fp_to_dim:w . . . . .` [22000](#)  
`\_fp_to_dim_dispatch:w . . . . .` [918](#), [22000](#)  
`\_fp_to_dim_recover:w . . . . .` [22000](#)  
`\_fp_to_int:w . . . . .` [918](#), [22025](#), [22030](#)  
`\_fp_to_int_dispatch:w . . . . .` [22016](#)  
`\_fp_to_int_recover:w . . . . .` [22016](#)  
`\_fp_to_scientific:w . . . . .` [915](#), [21834](#), [21844](#)  
`\_fp_to_scientific_dispatch:w . . . . .` [913](#), [917](#), [21824](#), [21828](#), [21831](#)  
`\_fp_to_scientific_normal:wnnnnn . . . . .` [21844](#)  
`\_fp_to_scientific_normal:wNw . . . . .` [21844](#)  
`\_fp_to_scientific_recover:w . . . . .` [21831](#)  
`\_fp_to_tl:w . . . . .` [21964](#), [21972](#), [22527](#)  
`\_fp_to_tl_dispatch:w . . . . .` [912](#), [916](#), [21956](#), [21960](#), [21963](#), [22096](#)  
`\_fp_to_tl_normal:nnnnn . . . . .` [21972](#)  
`\_fp_to_tl_recover:w . . . . .` [21963](#)  
`\_fp_to_tl_scientific:wnnnnn . . . . .` [21972](#)  
`\_fp_to_tl_scientific:wNw . . . . .` [21972](#)

```

\c__fp_trailing_shift_int 16420, 19741,
..... 19763, 19836, 20737, 21376, 21413
__fp_trap_division_by_zero_-
set:N 16646
__fp_trap_division_by_zero_set_-
error: 16646
__fp_trap_division_by_zero_set_-
flag: 16646
__fp_trap_division_by_zero_set_-
none: 16646
__fp_trap_invalid_operation_-
set:N 16612
__fp_trap_invalid_operation_-
set_error: 16612
__fp_trap_invalid_operation_-
set_flag: 16612
__fp_trap_invalid_operation_-
set_none: 16612
__fp_trap_overflow_set:N 16672
__fp_trap_overflow_set:NnNn 16672
__fp_trap_overflow_set_error: 16672
__fp_trap_overflow_set_flag: 16672
__fp_trap_overflow_set_none: 16672
__fp_trap_underflow_set:N 16672
__fp_trap_underflow_set_error: 16672
__fp_trap_underflow_set_flag: 16672
__fp_trap_underflow_set_none: 16672
__fp_trig:NNNNNwn 20949, 20964,
20979, 20994, 21009, 21024, 21041
\c__fp_trig_intarray 897,
21102, 21332, 21335, 21338, 21341,
21344, 21347, 21350, 21353, 21356
__fp_trig_large:ww 21049, 21316
__fp_trig_large_auxi:w 21316
__fp_trig_large_auxii:w 897, 21316
__fp_trig_large_auxiii:w 897, 21316
__fp_trig_large_auxix:Nw 21389
__fp_trig_large_auxv:www 21366, 21369
__fp_trig_large_auxvi:wnnnnnnnn 21369
__fp_trig_large_auxvii:w 21372, 21389
__fp_trig_large_auxviii:w 21389
__fp_trig_large_auxviii:ww 21391, 21395
__fp_trig_large_auxxx:wNNNNN 21389
__fp_trig_large_auxxi:w 21389
__fp_trig_large_pack:NNNNNw 21369, 21418
__fp_trig_small:ww 891, 899, 21051, 21055, 21061, 21428
__fp_trigd_large:ww 21049, 21063
__fp_trigd_large_auxi:nnnnwNNNN
..... 21063
__fp_trigd_large_auxii:wNw 21063
__fp_trigd_large_auxiii:www 21063
__fp_trigd_small:ww 892, 21051, 21057, 21100
__fp_trim_zeros:w 21815, 21939, 21948, 21999
__fp_trim_zeros_dot:w 21815
__fp_trim_zeros_end:w 21815
__fp_trim_zeros_loop:w 21815
__fp_tuple_18847, 18848, 18851, 18852
__fp_tuple_&_o:ww 18830
__fp_tuple_&_tuple_o:ww 18830
__fp_tuple_*_o:ww 19703
__fp_tuple_+_tuple_o:ww 19709
__fp_tuple_--tuple_o:ww 19709
__fp_tuple_/_o:ww 19703
__fp_tuple_chk:w 735,
16323, 16329, 16330, 16407, 16410,
18089, 18301, 18316, 18341, 18344,
18360, 18361, 18364, 18571, 18572,
19712, 19713, 19719, 19720, 21794
__fp_tuple_compare_back:ww 18568
__fp_tuple_compare_back_loop:w
..... 18568
__fp_tuple_compare_back_-
tuple:ww 18568
__fp_tuple_convert:Nw 21794, 21843, 21897, 21971
__fp_tuple_convert_end:w 21794
__fp_tuple_convert_loop:nNw 21794
__fp_tuple_count:w 16328
__fp_tuple_count_loop:Nw 16328
__fp_tuple_map_loop_o:nw 18341
__fp_tuple_map_o:nw 18341,
19696, 19704, 19706, 19708
__fp_tuple_mapthread_loop_o:nw
..... 18359
__fp_tuple_mapthread_o:nww
..... 18359, 19717
__fp_tuple_not_o:w 18821
__fp_tuple_set_sign_aux_o:Nnw 19688
__fp_tuple_set_sign_aux_o:w 19688
__fp_tuple_set_sign_o:w 19688
__fp_tuple_to_decimal:w 21885
__fp_tuple_to_scientific:w 21831
__fp_tuple_to_tl:w 21963
__fp_tuple_|_o:ww 18830
__fp_tuple_|_tuple_o:ww 18830
__fp_type_from_scan:N
..... 736, 16352, 17911, 17913, 17937,
17939, 17950, 17952, 18532, 18534

```

```

__fp_type_from_scan:w 16352
__fp_type_from_scan_other:N 16352, 16376, 16394
__fp_underflow:w 733, 746, 748, 16272, 16703, 20440
__fp_use_i:ww 856, 910, 16208, 19953, 21723
__fp_use_i:www 16208
__fp_use_i_delimit_by_s_stop:nw 16219, 18500, 18862
__fp_use_i_until_s:nw 899, 16203, 16252, 16262, 16525, 21093, 21371, 21377, 21408, 22183, 22254, 22465
__fp_use_ii_until_s:nnw 16203, 16250, 16261
__fp_use_none_stop_f:n 16200, 20118, 20119, 20120
__fp_use_none_until_s:w 16203, 19499, 20797, 21718, 21721
__fp_use_s:n 16201
__fp_use_s:nn 16201
__fp_zero_fp:N . 16243, 16687, 17035
__fp_l_o:ww 806, 18830
__fp_l_tuple_o:ww 18830
__fp 18833, 18840, 18849, 18850
fpararray commands:
\fparray_count:N 221, 221, 22425, 22437, 22448, 22504
\fparray_gset:Nnn ... 221, 931, 22450
\fparray_gzero:N 221, 22501
\fparray_item:Nn 221, 931, 22514
\fparray_item_to_tl:Nn ... 221, 22514
\fparray_new:Nn 221, 22398
\futurelet 281

G
\gdef 282
\GetIdInfo 7, 14042
\leaders 830
\global 183, 283
\globaldefs 284
\glueexpr 538
\glueshrink 539
\glueshrinkorder 540
\gluestretch 541
\gluestretchorder 542
\gluetomu 543
group commands:
\group_align_safe_begin/end: 538, 964
\group_align_safe_begin: 114, 389, 393, 530, 3907, 4516, 9165, 9368, 11233, 11248, 23395, 29585, 29910, 31443, 32524
\group_align_safe_end: 114, 389, 393, 3928, 4499, 9167, 9368, 11242, 11253, 11259, 23398, 29597, 29923, 31454, 32532
\group_begin: 9, 385, 1120, 1446, 2175, 2178, 2181, 2569, 3040, 3231, 3641, 3781, 4003, 4016, 4726, 4753, 5008, 5031, 5418, 5528, 5581, 5894, 5942, 5988, 5995, 6322, 6504, 7752, 7789, 9410, 9570, 9661, 10513, 10519, 10570, 10636, 10880, 10898, 10922, 11010, 11029, 11354, 11869, 11893, 11909, 11975, 12276, 12629, 12866, 13059, 13105, 13367, 13525, 14049, 14722, 14868, 15840, 18830, 22647, 22853, 23053, 23090, 23394, 23401, 23474, 23634, 23976, 24069, 24384, 24883, 25247, 25340, 25731, 26089, 26300, 26460, 26469, 26481, 26490, 26498, 26616, 26646, 27137, 28943, 29252, 29298, 29382, 29409, 30847, 30854, 30883, 31142, 31179, 31339, 31368, 31399, 31675, 31762, 32509, 32887, 32950, 32959, 32970
\c_group_begin_token 55, 134, 274, 403, 580, 4362, 4400, 10880, 10904, 23439, 27180, 27186, 27200, 27206, 27284, 27290, 27305, 27311, 29465, 29670, 32547
\group_end: 9, 9, 491, 938, 941, 1120, 1446, 2175, 2178, 2184, 2578, 3043, 3234, 3647, 3803, 3853, 4006, 4020, 4744, 4776, 5013, 5036, 5428, 5541, 5584, 5907, 5954, 6013, 6075, 6503, 6635, 7761, 7799, 7804, 9429, 9587, 9689, 10521, 10528, 10639, 10655, 10897, 10901, 10929, 11028, 11077, 11378, 11888, 11901, 11920, 12133, 12322, 12645, 12872, 13063, 13134, 13390, 13543, 14052, 14842, 14906, 15854, 18854, 22651, 22886, 23057, 23098, 23307, 23399, 23420, 23481, 23658, 23989, 24083, 24417, 24425, 24896, 25305, 25347, 25354, 25362, 25735, 25736, 26126, 26364, 26465, 26476, 26560, 26622, 26683, 27143, 28944, 29307, 29379, 29396, 29425, 30872, 30882, 31165, 31193, 31363, 31367, 31394, 31425, 31679, 32032, 32513, 32895, 32953, 32963, 32974
\c_group_end_token 134, 274, 580, 10880, 10909, 23442, 27194, 27299, 29466, 29676, 32548
\group_insert_after:N 9, 1452, 23985

```

|                              |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| groups commands:             |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| .groups:n .....              | 188, <a href="#">15341</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>H</b>                     |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| \H .....                     | 29543, 31805, 31952, 31953, 31980, 31981                                                                                                                                                                                                                                                                                                                                                                                                |
| \halign .....                | 285                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hangafter .....             | 286                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hangindent .....            | 287                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hbadness .....              | 288                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hbox .....                  | 289                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| hbox commands:               |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| \hbox:n .....                | 242, <a href="#">27151</a> , 27378, 27674, 28747                                                                                                                                                                                                                                                                                                                                                                                        |
| \hbox_gset:Nn .....          | 243, <a href="#">27153</a> , 27345, 27468, 27512, 27532, 27552, 27569, 27590, 27619, 27630, 27788, 28216, 32051                                                                                                                                                                                                                                                                                                                         |
| \hbox_gset:Nw .....          | 243, <a href="#">27177</a> , 27858                                                                                                                                                                                                                                                                                                                                                                                                      |
| \hbox_gset_end: .....        | 243, <a href="#">27177</a> , 27861                                                                                                                                                                                                                                                                                                                                                                                                      |
| \hbox_gset_to_wd:Nnn .....   | 243, <a href="#">27165</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| \hbox_gset_to_wd:Nnw .....   | 243, <a href="#">27197</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| \hbox_overlap_center:n ..... | 243, <a href="#">27221</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| \hbox_overlap_left:n .....   | 243, <a href="#">27221</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| \hbox_overlap_right:n .....  | 243, <a href="#">27221</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| \hbox_set:Nn .....           | 243, 243, 255, <a href="#">27153</a> , 27342, 27374, 27375, 27462, 27509, 27529, 27536, 27549, 27566, 27587, 27616, 27624, 27647, 27775, 28213, 28236, 28495, 28582, 28879, 32048, 32061, 32069, 32077, 32086, 32095, 32112, 32120, 32128, 32134, 32147                                                                                                                                                                                 |
| \hbox_set:Nw .....           | 243, <a href="#">27177</a> , 27845                                                                                                                                                                                                                                                                                                                                                                                                      |
| \hbox_set_end: .....         | 243, 243, <a href="#">27177</a> , 27848                                                                                                                                                                                                                                                                                                                                                                                                 |
| \hbox_set_to_wd:Nnn .....    | 243, 243, <a href="#">27165</a>                                                                                                                                                                                                                                                                                                                                                                                                         |
| \hbox_set_to_wd:Nnw .....    | 243, <a href="#">27197</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| \hbox_to_wd:nn .....         | 243, <a href="#">27211</a> , 27665                                                                                                                                                                                                                                                                                                                                                                                                      |
| \hbox_to_zero:n .....        | 243, <a href="#">27211</a> , 27222, 27224, 27226                                                                                                                                                                                                                                                                                                                                                                                        |
| \hbox_unpack:N .....         | 244, <a href="#">27227</a> , 28499                                                                                                                                                                                                                                                                                                                                                                                                      |
| \hbox_unpack_clear:N .....   | <a href="#">32914</a>                                                                                                                                                                                                                                                                                                                                                                                                                   |
| \hbox_unpack_drop:N .....    | 246, <a href="#">27227</a> , 32914, 32916                                                                                                                                                                                                                                                                                                                                                                                               |
| hcoffin commands:            |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| \hcoffin_gset:Nn .....       | 250, <a href="#">27771</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| \hcoffin_gset:Nw .....       | 251, <a href="#">27841</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| \hcoffin_gset_end: .....     | 251, <a href="#">27841</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| \hcoffin_set:Nn .....        | 250, <a href="#">27771</a> , 28751, 28758, 28796, 28831                                                                                                                                                                                                                                                                                                                                                                                 |
| \hcoffin_set:Nw .....        | 251, <a href="#">27841</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| \hcoffin_set_end: .....      | 251, <a href="#">27841</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| \hfi .....                   | 1121                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| \hfil .....                  | 290                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hfill .....                 | 291                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hfilneg .....               | 292                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hfuzz .....                 | 293                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hjcode .....                | 825                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hoffset .....               | 294                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \holdinginserts .....        | 295                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| hook commands:               |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| \hook_gput_code:nnn .....    | 22776, 22778                                                                                                                                                                                                                                                                                                                                                                                                                            |
| \hpack .....                 | 826                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hrule .....                 | 296                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hsize .....                 | 297                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hskip .....                 | 298                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hss .....                   | 299                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \ht .....                    | 300                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \Huge .....                  | 31655                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| \huge .....                  | 31659                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| hundred commands:            |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| \c_one_hundred .....         | 32751                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| hyphenation commands:        |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| \hyphenation .....           | 301                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hyphenationbounds .....     | 827                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hyphenationmin .....        | 828                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hyphenchar .....            | 302                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hyphenpenalty .....         | 303                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| \hyphenpenaltymode .....     | 829                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>I</b>                     |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| \i .....                     | 31392, 31735, 31861, 31863, 31865, 31867, 31918, 31921, 31924, 31927, 31998                                                                                                                                                                                                                                                                                                                                                             |
| if commands:                 |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| \if:w .....                  | 23, 129, 324, 325, 361, 391, 392, 394, 405, 406, 993, <a href="#">1417</a> , 1774, 2069, 2070, 2932, 2935, 2936, 2937, 2938, 2953, 2954, 2955, 2956, 2957, 2958, 2959, 2960, 2961, 3025, 3026, 3028, 3957, 3967, 4063, 4414, 4434, 4449, 8886, 11193, 16964, 17337, 17341, 17363, 17456, 17488, 17507, 17573, 17587, 17604, 17625, 18132, 18147, 18485, 20688, 22195, 24285, 29312, 29320, 29337, 29449                                 |
| \if_bool:N .....             | 113, 113, 527, 1427, <a href="#">9078</a> , 9123                                                                                                                                                                                                                                                                                                                                                                                        |
| \if_box_empty:N .....        | 249, <a href="#">27089</a> , 27101                                                                                                                                                                                                                                                                                                                                                                                                      |
| \if_case:w .....             | 101, 423, 424, 461, 518, 742, 830, 883, 925, 1977, 2623, 5106, 5180, 5404, 6449, <a href="#">8195</a> , 8777, 8810, 10648, 13219, 16267, 16520, 16535, 16904, 16933, 18220, 18261, 18921, 19056, 19131, 19156, 19208, 19652, 19668, 19685, 19962, 20189, 20216, 20374, 20409, 20567, 20612, 20664, 20790, 20813, 20846, 20905, 20945, 20960, 20975, 20990, 21005, 21020, 21546, 21599, 21683, 21698, 21750, 21763, 21847, 21901, 21975, |

22192, 22479, 22558, 23450, 23644,  
23935, 24199, 25000, 25029, 25086,  
25496, 25550, 25910, 26241, 32541  
\if\_catcode:w ..... 23, 403,  
403, 591, 1417, 3072, 4357, 4398,  
10816, 10819, 10822, 10825, 10828,  
10831, 10834, 10904, 10909, 10914,  
10919, 10926, 10933, 10938, 10943,  
10948, 10953, 10958, 10968, 10995,  
11285, 11290, 11360, 11361, 17092,  
17297, 17615, 17662, 17965, 18009,  
23439, 23442, 23596, 23598, 23600,  
23602, 23604, 23606, 23608, 29413,  
29414, 29450, 29465, 29466, 29467,  
29468, 29469, 29470, 29471, 29472,  
29473, 29496, 29499, 29502, 29505,  
29508, 29511, 29514, 32542, 32543  
\if\_charcode:w .....  
..... 23, 129, 403, 403, 427, 591,  
969, 1417, 4343, 4391, 5264, 5384,  
6064, 10973, 11287, 13847, 13856,  
16523, 18498, 18860, 23509, 23533,  
23582, 24142, 24152, 24634, 26095  
\if\_cs\_exist:N ..... 23,  
1432, 1801, 1829, 2572, 11003, 11202  
\if\_cs\_exist:w 23, 1432, 1460, 1810,  
1838, 1964, 9026, 9054, 9063, 32159  
\if\_dim:w .....  
184, 14205, 14293, 14305, 14328, 14499  
\if\_eof:w .....  
165, 636, 12809, 12816, 12899, 12917  
\if\_false: ..... 23, 107, 351,  
385, 389, 393, 401, 493, 509, 538,  
574, 655, 961, 1050, 1417, 2583,  
2593, 2606, 2619, 2647, 2663, 2761,  
2775, 2781, 2788, 2796, 2806, 2819,  
2823, 3782, 3789, 3923, 3924, 4035,  
4039, 4078, 4298, 4303, 4310, 4315,  
4325, 4415, 4428, 4446, 4450, 4460,  
7695, 7698, 7877, 7882, 8407, 9369,  
9571, 9579, 10600, 10644, 13198,  
13238, 13242, 13249, 13257, 13524,  
13537, 14315, 23386, 23428, 23477,  
23480, 24393, 24412, 24413, 24422,  
24473, 24509, 24523, 24527, 24750,  
24783, 24795, 24799, 24833, 24838,  
24846, 24881, 24888, 24893, 24941,  
25164, 25181, 25185, 26316, 26333,  
26572, 26577, 26688, 26693, 29672,  
29678, 32414, 32426, 32452, 32462  
\if\_hbox:N ..... 249, 27089, 27093  
\if\_int\_compare:w .....  
... 22, 101, 509, 510, 1450, 2811,  
4605, 4614, 4663, 4664, 4670, 4840,  
4849, 4854, 5090, 5145, 5146, 5152,  
5164, 5180, 5393, 5401, 5608, 5609,  
5610, 5615, 5616, 5658, 5710, 5810,  
6029, 6091, 6095, 6125, 6128, 6144,  
6148, 6169, 6247, 6249, 6268, 6269,  
6287, 6289, 6343, 6346, 6347, 6465,  
6466, 6607, 6612, 8195, 8250, 8291,  
8292, 8387, 8440, 8442, 8444, 8446,  
8448, 8450, 8452, 8455, 8588, 9369,  
9371, 10542, 10543, 10550, 10551,  
10552, 10553, 10558, 10559, 10589,  
10663, 10664, 10670, 11184, 13205,  
13863, 14344, 15992, 15995, 16039,  
16105, 16268, 16269, 16463, 16560,  
16784, 16794, 16802, 16815, 16828,  
16835, 16856, 16868, 16877, 16888,  
16997, 17002, 17074, 17104, 17257,  
17259, 17296, 17301, 17354, 17374,  
17401, 17415, 17450, 17477, 17505,  
17521, 17537, 17555, 17615, 17635,  
17651, 17664, 17678, 17739, 17762,  
17791, 17793, 17966, 17976, 17978,  
18010, 18020, 18022, 18044, 18061,  
18066, 18096, 18164, 18209, 18509,  
18556, 18559, 18590, 18599, 18602,  
18607, 18608, 18611, 18614, 18801,  
18925, 18946, 18983, 19078, 19132,  
19133, 19136, 19139, 19209, 19218,  
19423, 19496, 19549, 19553, 19557,  
19575, 19610, 19611, 19612, 19613,  
19614, 19640, 19964, 19967, 20061,  
20154, 20202, 20218, 20345, 20379,  
20437, 20446, 20486, 20657, 20659,  
20670, 20688, 20711, 20743, 20746,  
20793, 20823, 20870, 20884, 21048,  
21092, 21550, 21588, 21597, 21633,  
21717, 21720, 21949, 22182, 22250,  
22251, 22252, 22262, 22290, 22295,  
22296, 22359, 22360, 22361, 22365,  
22380, 22385, 22433, 22437, 22978,  
23035, 23064, 23105, 23116, 23119,  
23137, 23192, 23202, 23212, 23414,  
23486, 23519, 23527, 23550, 23574,  
23625, 23640, 23722, 23725, 23873,  
23879, 23880, 23887, 23890, 23893,  
23899, 23900, 23904, 23907, 23908,  
23916, 23917, 23918, 23924, 23954,  
23955, 24196, 24216, 24217, 24218,  
24221, 24225, 24226, 24229, 24230,  
24238, 24239, 24242, 24246, 24247,  
24250, 24309, 24331, 24343, 24352,  
24360, 24363, 24373, 24376, 24404,  
24477, 24582, 24648, 24653, 24681,  
24748, 24781, 24892, 24909, 25195,

|                                     |                                               |
|-------------------------------------|-----------------------------------------------|
| 25228, 25443, 25514, 25540, 25602,  | 20669, 20770, 20774, 20833, 20850,            |
| 25615, 25626, 25642, 25693, 25725,  | 20856, 21440, 21493, 21544, 21545,            |
| 25888, 25889, 25936, 25963, 26038,  | 21547, 21548, 21568, 21585, 21652,            |
| 26106, 26169, 26179, 26182, 26202,  | 21750, 21846, 21900, 21974, 22044,            |
| 26259, 26312, 26329, 26352, 26501,  | 22049, 22181, 22213, 22224, 22331,            |
| 26530, 26570, 26575, 26596, 26674,  | 22492, 22545, 22551, 22988, 22989,            |
| 26686, 26691, 29313, 29482, 30445,  | 23436, 23466, 23494, 23594, 23984,            |
| 30448, 30449, 30452, 30465, 30468,  | 24284, 24308, 24630, 24633, 25076,            |
| 30471, 30474, 30477, 30480, 30494,  | 25478, 25650, 25661, 25676, 25831,            |
| 30497, 30500, 30503, 30506, 30518,  | 25864, 26214, 26452, 26529, 26582,            |
| 30521, 30524, 30527, 32495, 32503   | 26621, 29289, 29291, 29302, 29428,            |
| \if_int_odd:w . . . . . 102,        | 32038, 32419, 32456, 32475, 32544             |
| 902, 8195, 8322, 8493, 8501, 9001,  | \if_mode_horizontal: . 23, 1428, 9363         |
| 10549, 10557, 10574, 11359, 16806,  | \if_mode_inner: . . . . . 23, 1428, 9365      |
| 16853, 16865, 18257, 19192, 19478,  | \if_mode_math: . . . . . 23, 1428, 9367       |
| 20834, 21398, 21437, 21447, 21490,  | \if_mode_vertical: 23, 1428, 2236, 9361       |
| 21514, 21674, 22358, 23650, 23944,  | \if_predicate:w 105, 107, 113, 9078,          |
| 24320, 24328, 24340, 24754, 25069   | 9155, 9215, 9230, 9241, 9256, 9267            |
| \if_meaning:w . . . . . 23, 373,    | \if_true: . . . . . 23, 107, 1417, 4440, 4446 |
| 403, 814, 1132, 1417, 1628, 1654,   | \if_vbox:N . . . . . 249, 27089, 27095        |
| 1672, 1731, 1736, 1745, 1798, 1816, | \ifabsdim . . . . . 916                       |
| 1826, 1844, 1995, 2009, 2130, 2226, | \ifabsnum . . . . . 917                       |
| 2289, 2290, 2624, 2627, 2628, 2629, | \ifcase . . . . . 305                         |
| 2630, 2723, 2753, 2766, 2772, 2880, | \ifcat . . . . . 306                          |
| 2903, 2912, 3104, 3177, 3189, 3190, | \ifcondition . . . . . 831                    |
| 3298, 3304, 3330, 3342, 3350, 3382, | \ifcsname . . . . . 544                       |
| 3389, 3413, 3417, 3495, 3520, 3535, | \ifdbox . . . . . 1122                        |
| 3947, 3991, 4007, 4021, 4382, 4949, | \ifddir . . . . . 1123                        |
| 5017, 5040, 5201, 5239, 5554, 6297, | \ifdefined . . . . . 545                      |
| 6446, 6461, 6488, 6603, 7731, 7794, | \ifdim . . . . . 307                          |
| 7809, 7817, 8218, 8221, 8231, 8266, | \ifeof . . . . . 308                          |
| 8271, 8272, 8422, 9177, 9199, 9967, | \iffalse . . . . . 309                        |
| 9982, 10004, 10018, 10963, 11000,   | \iffontchar . . . . . 546                     |
| 11039, 11042, 11176, 11239, 11278,  | \ifhbox . . . . . 310                         |
| 11362, 11666, 13155, 14274, 14321,  | \ifhmode . . . . . 311                        |
| 14502, 16249, 16270, 16282, 16292,  | \ifincsname . . . . . 690                     |
| 16387, 16442, 16451, 16542, 16557,  | \ifinner . . . . . 312                        |
| 16559, 16695, 16783, 16793, 16805,  | \ifjfont . . . . . 1124                       |
| 16818, 16819, 16838, 16839, 16853,  | \ifmbox . . . . . 1125                        |
| 16854, 16865, 16866, 16932, 16979,  | \ifmdir . . . . . 1126                        |
| 17014, 17017, 17033, 17040, 17093,  | \ifmmode . . . . . 313                        |
| 17096, 17212, 17213, 17214, 17215,  | \ifnum . . . . . 27, 38, 67, 80, 85, 314      |
| 17218, 17314, 17427, 17433, 17663,  | \ifodd . . . . . 315                          |
| 17711, 17922, 17995, 18258, 18276,  | \ifpdfabsdim . . . . . 651                    |
| 18326, 18336, 18545, 18546, 18547,  | \ifpdfabsnum . . . . . 652                    |
| 18548, 18549, 18550, 18782, 18794,  | \ifpdfprimitive . . . . . 653                 |
| 18795, 18823, 18835, 18842, 18859,  | \ifprimitive . . . . . 783                    |
| 18922, 18957, 18971, 19017, 19024,  | \iftbox . . . . . 1127                        |
| 19100, 19112, 19212, 19215, 19226,  | \iftdir . . . . . 1129                        |
| 19279, 19352, 19422, 19425, 19432,  | \iftfont . . . . . 1128                       |
| 19465, 19466, 19469, 19690, 19935,  | \iftrue . . . . . 316, 32038                  |
| 19946, 20127, 20137, 20186, 20285,  | \ifvbox . . . . . 317                         |
| 20365, 20414, 20428, 20575, 20609,  | \ifvmode . . . . . 318                        |
| 20621, 20634, 20637, 20640, 20643,  | \ifvoid . . . . . 319                         |

\ifx ..... 4, 21, 25,  
   30, 68, 70, 71, 72, 83, 90, 110, 111, 320  
\ifybox ..... 1130  
\ifydir ..... 1131  
\ignored ..... 33  
\ignoreligaturesinfont ..... 918  
\ignorespaces ..... 321  
\IJ ..... 29551, 31383, 31725  
\ij ..... 29551, 31383, 31737  
\immediate ..... 322  
\immediateassigned ..... 832  
\immediateassignment ..... 833  
in ..... 218  
\indent ..... 323  
inf ..... 217  
\infty ..... 17215, 17216  
inherit commands:  
  .inherit:n ..... 188, [15343](#)  
\inhibitglue ..... 1132  
\inhibitxspcode ..... 1133  
\initcatcodetable ..... 834  
initial commands:  
  .initial:n ..... 189, [15345](#)  
\input ..... 31, 324  
\inputlineno ..... 325  
\insert ..... 326  
\insertrht ..... 919  
\insertpenalties ..... 327  
int commands:  
  \c\_eight ..... 32731  
  \c\_eleven ..... 32737  
  \c\_fifteen ..... 32745  
  \c\_five ..... 32725  
  \l\_foo\_int ..... 234  
  \c\_four ..... 32723  
  \c\_fourteen ..... 32743  
  \int\_abs:n ..... 90, 503, [8224](#), 16039  
  \int\_add:Nn ..... 91, [8352](#), 13314, 23925,  
   25071, 25632, 25633, 25873, 25960  
  \int\_case:nn ..... 94, 518, [8461](#),  
   8640, 8646, 22926, 30599, 30614, 30677  
  \int\_case:nnm ..... 32777  
  \int\_case:nnTF .....  
   ..... 94, 8109, [8461](#), 8466, 8471,  
   10280, 17124, 21796, 26972, 32778  
\int\_compare:nNnTF .....  
   ..... 92, 92, 93, 94, 94, 95, 95,  
   204, 3784, 3812, 3827, 3835, 4560,  
   4567, 4634, 5069, 5071, 5080, 5766,  
   5842, 6647, 7746, 7935, 7942, 8303,  
   8309, [8453](#), 8485, 8537, 8545, 8554,  
   8560, 8572, 8575, 8636, 8724, 8730,  
   8736, 8756, 8910, 8929, 8931, 8973,  
   9439, 9445, 9756, 10319, 10321,  
   10326, 10335, 10355, 10372, 10782,  
   10866, 13014, 13127, 13610, 13734,  
   13744, 14092, 14135, 14523, 15977,  
   15982, 15989, 16097, 16139, 16165,  
   18574, 19715, 21779, 21924, 21926,  
   22413, 22623, 22811, 22965, 23769,  
   23962, 23974, 24128, 24446, 24448,  
   25241, 25834, 26056, 26071, 26271,  
   26546, 26989, 30138, 30167, 30174,  
   30254, 30256, 30259, 30300, 30313,  
   30322, 30363, 30753, 30756, 30796,  
   30802, 30809, 30823, 32246, 32574,  
   32576, 32577, 32578, 32580, 32603  
\int\_compare:nTF .....  
   ..... 92, 93, 95, 95, 95, 95, 205, 675,  
   [8400](#), 8509, 8517, 8526, 8532, 12787,  
   12814, 12984, 21984, 22817, 25345,  
   26729, 26951, 26952, 26957, 26959  
\int\_compare\_p:n ..... 93, [8400](#), 25352  
\int\_compare\_p:nNn ..... 22, 92,  
  [8453](#), 9453, 9672, 9735, 9737, 9739,  
  12929, 13723, 13724, 22749, 25140,  
  25141, 30195, 30394, 30395, 30554,  
  30555, 30657, 30658, 30659, 30707,  
  30715, 30716, 30737, 30738, 30780  
\int\_const:Nn .....  
   ..... 91, 5344, 5345, [8301](#), 8939,  
   8940, 8941, 8942, 8943, 8944, 8945,  
   8946, 8947, 8948, 8949, 8950, 8951,  
   8952, 8997, 8998, 8999, 9591, 9682,  
   9684, 9686, 9687, 9688, 9722, 12718,  
   12859, 12924, 12925, 16230, 16231,  
   16232, 16233, 16234, 16235, 16236,  
   16420, 16421, 16422, 16424, 16425,  
   16426, 16429, 16430, 16431, 16778,  
   17052, 17053, 17054, 17055, 17056,  
   17057, 17058, 17059, 17060, 17061,  
   17062, 17063, 17064, 17065, 17066,  
   20843, 22132, 23852, 23853, 23854,  
   23855, 24257, 24258, 24259, 24260,  
   24261, 24262, 24266, 24267, 24268,  
   24269, 24270, 24271, 24272, 24273,  
   24274, 24275, 24276, 24277, 24278  
\int\_decr:N ..... 91,  
  [8364](#), 23125, 23126, 23127, 23190,  
  23191, 23200, 23201, 23210, 23211,  
  23429, 25912, 26261, 26330, 26531  
\int\_div\_round:nn ..... 90, [8256](#)  
\int\_div\_truncate:nn ..... 90,  
  90, 5449, 5454, 6118, 6119, 6174,  
  6354, 6520, 6531, [8256](#), 8651, 8749,  
  8769, 9685, 10676, 10689, 10694, 10706  
\int\_do\_until:nn ..... 95, [8507](#)  
\int\_do\_until:nNnn ..... 94, [8535](#)

```

\int_do_while:nn 95, 8507
\int_do_while:nNnn 94, 8535
\int_eval:n 14,
 28, 89, 90, 90, 92, 92, 93, 94,
 101, 101, 268, 305, 332, 399, 505,
 521, 724, 725, 728, 760, 807, 831–
 833, 977, 1191, 1977, 2006, 2022,
 3838, 4196, 4201, 4209, 4553, 4561,
 4569, 4596, 4600, 4609, 4616, 4651,
 4661, 5063, 5076, 5101, 5125, 5126,
 5138, 5143, 5174, 5191, 5228, 5404,
 5424, 5442, 5735, 6258, 6273, 6301,
 6450, 6455, 6473, 6617, 7928, 7936,
 7944, 8083, 8207, 8464, 8469, 8474,
 8479, 8633, 8719, 8721, 8851, 8861,
 8896, 8907, 8913, 8924, 8955, 8992,
 8996, 9316, 9710, 10253, 10262,
 10313, 10323, 10337, 10344, 10359,
 10414, 10416, 10484, 10486, 10490,
 10492, 10496, 10498, 10502, 10504,
 10537, 10538, 10659, 10711, 10714,
 10719, 10725, 11572, 12506, 12771,
 12969, 13250, 13308, 13715, 13716,
 13738, 13748, 13755, 13756, 15976,
 16014, 16015, 16066, 16083, 16135,
 16159, 16168, 16172, 16240, 22121,
 22281, 22284, 22285, 22375, 22376,
 22408, 22456, 22518, 22526, 22595,
 22619, 23067, 23331, 23332, 23552,
 23628, 23632, 23655, 23944, 24200,
 25069, 25275, 25279, 25282, 25286,
 25463, 25465, 25479, 25480, 25482,
 25483, 25626, 25716, 25747, 25931,
 25979, 26054, 26181, 26186, 26733,
 26778, 26779, 26978, 27119, 27129,
 32249, 32497, 32505, 32604, 32605
\int_eval:w 90, 306, 310,
 310, 5094, 5436, 8059, 8207, 9029,
 9064, 13201, 13210, 13235, 13247,
 16110, 19664, 23379, 23614, 23624
\int_from_alpha:n 98, 8894
\int_from_base:nn 99, 8911, 8934, 8936, 8938
\int_from_bin:n 99, 8933, 32780
\int_from_binary:n 32779
\int_from_hex:n 99, 8933, 32782
\int_from_hexadecimal:n 32781
\int_from_oct:n 99, 8933, 32784
\int_from_octal:n 32783
\int_from_roman:n 99, 8953
\int_gadd:Nn 91, 8352
\int_gdecr:N 91, 4142,
 4968, 7993, 8041, 8364, 8631, 10214,
 11711, 12894, 14487, 18766, 23694
\int_gincr:N 91, 4135, 4957,
 7985, 8035, 8364, 8606, 8617, 10207,
 11706, 12885, 14466, 14473, 15967,
 18745, 18752, 22403, 22706, 23672
.int_gset:N 189, 15353
\int_gset:Nn 92, 505, 8376, 11926
\int_gset_eq:NN 91, 8344
\int_gsub:Nn 92, 8352, 22417
\int_gzero:N 91, 8334, 8341
\int_gzero_new:N 91, 8338
\int_if_even:nTF 94, 8491, 13476
\int_if_even_p:n 94, 8491
\int_if_exist:NTF 91, 8339,
 8341, 8348, 8967, 8971, 24995, 25050
\int_if_exist_p:N 91, 8348
\int_if_odd:nTF 94, 8491, 20050
\int_if_odd_p:n 94, 8491, 25389
\int_incr:N 91, 7768, 8364, 15104, 16055,
 16089, 16181, 22506, 23039, 23135,
 23136, 23470, 23512, 23525, 23543,
 23809, 23810, 24886, 25310, 25471,
 25561, 25874, 25909, 25911, 25986,
 26248, 26313, 26472, 26528, 26592
\int_log:N 100, 8993
\int_log:n 100, 8995
\int_max:nn 90, 920, 8224, 19911,
 21072, 25585, 25795, 26678, 26679
\int_min:nn 90, 923, 8224
\int_mod:nn 90, 5771, 5835, 6119, 6120, 6355,
 8256, 8641, 8740, 8760, 9683, 10708
\int_new:N 90,
 91, 5817, 8295, 8305, 8311, 8339,
 8341, 9009, 9010, 9011, 9012, 9013,
 9014, 9372, 13036, 13039, 13041,
 13054, 14915, 15959, 15961, 22396,
 22397, 22574, 22944, 22945, 22946,
 22947, 22948, 22949, 22950, 22951,
 22952, 22953, 22954, 23365, 23366,
 23367, 23368, 23835, 23836, 23837,
 23850, 24255, 24256, 24263, 24264,
 24281, 25406, 25408, 25409, 25410,
 25413, 25753, 25754, 25755, 25756,
 25757, 25758, 25759, 25760, 25761,
 25762, 25765, 25766, 25767, 26016,
 26441, 26444, 26445, 26446, 26993
\int_rand:n 99, 267, 16076, 22123, 22126, 22373
\int_rand:nn 99, 117, 267, 922, 923,
 930, 1193, 4576, 7950, 8997, 10373,
 10378, 22117, 22120, 22279, 32239
\int_range:nn 924
.int_set:N 189, 15353

```

```

\int_set:Nn 92, 305, 2182, 3785, 3820,
 5641, 5896, 5945, 5998, 7769, 8376,
 12845, 12847, 13020, 13022, 13037,
 13047, 13060, 13107, 13113, 13125,
 13130, 15109, 22674, 22675, 22690,
 22839, 22854, 22890, 22959, 22961,
 22963, 22985, 22986, 23001, 23009,
 23010, 23022, 23023, 23041, 23044,
 23424, 23487, 23797, 25077, 25407,
 25458, 25460, 25529, 25581, 25582,
 25592, 25603, 25627, 25645, 25694,
 25780, 25793, 25824, 25845, 25935,
 25970, 26477, 26625, 26651, 27128,
 27130, 27138, 27139, 27140, 27141
\int_set_eq:NN 91, 3783, 3786,
 8344, 9572, 13526, 23002, 23032,
 23915, 24374, 24378, 24387, 24389,
 24432, 24485, 24785, 24885, 24898,
 24997, 25423, 25434, 25435, 25469,
 25470, 25520, 25624, 25625, 25677,
 25732, 25782, 25785, 25800, 25807,
 25821, 25823, 25826, 25840, 25843,
 25875, 25876, 25880, 26010, 26583
\int_show:N 100, 8989
\int_show:n 100, 523, 1191, 8991
\int_sign:n 90, 680, 8210, 22756
\int_step_... 236
\int_step_function:nN
 96, 7759, 8563, 22858, 22859
\int_step_function:nnN 96, 8563,
 10635, 10640, 22860, 22861, 22862,
 22863, 22884, 23094, 25881, 26542
\int_step_function:nnnN 96, 270,
 270, 514, 811, 8563, 8630, 26654, 26662
\int_step_inline:nn 96,
 725, 8600, 15970, 22600, 22634, 22685
\int_step_inline:nnn
 96, 5755, 5764, 8600, 12722, 12936,
 22607, 22610, 22840, 25815, 26996
\int_step_inline:nnnn 96, 812, 8600
\int_step_variable:nNn 96, 8600
\int_step_variable:nnNn 96, 8600
\int_step_variable:nnnNn 96, 8600
\int_sub:Nn 92, 8352, 11870, 13322,
 23919, 24589, 25680, 25688, 25697
\int_to_Alph:n 97, 98, 8654
\int_to_alpha:n 97, 97, 98, 8654
\int_to_arabic:n 97, 8633
\int_to_Base:n 98
\int_to_base:n 98
\int_to_Base:nn 98, 99, 8718, 8845
\int_to_base:nn
 98, 99, 8718, 8841, 8843, 8847
\int_to_bin:n 98, 98, 99, 8840, 32786
\int_to_binary:n 32785
\int_to_Hex:n 98, 99, 8840, 24131
\int_to_hex:n 98, 99, 8840, 32788
\int_to_hexadecimal:n 32787
\int_to_oct:n 98, 99, 8840, 32790
\int_to_octal:n 32789
\int_to_Roman:n 98, 99, 8848
\int_to_roman:n 98, 99, 8848
\int_to_symbols:nnn
 97, 97, 8634, 8656, 8688
\int_until_do:nn 95, 8507
\int_until_do:nNn 95, 8535
\int_use:N
 89, 92, 754, 759, 4137, 4139,
 4959, 4963, 6254, 6278, 6292, 6312,
 6319, 6501, 6610, 6615, 6631, 7986,
 7992, 8037, 8039, 8382, 8609, 8620,
 10209, 10211, 11705, 11713, 11830,
 12401, 12508, 12848, 12887, 13016,
 14469, 14476, 15110, 18748, 18755,
 22708, 22710, 22741, 23674, 24406,
 24479, 24550, 24561, 24570, 24574,
 24585, 24586, 24592, 24593, 24599,
 24600, 24768, 25389, 25451, 25453,
 25559, 25572, 25573, 25971, 26001,
 26108, 26120, 26216, 26477, 26990,
 32305, 32314, 32316, 32319, 32324,
 32333, 32335, 32339, 32342, 32347
\int_value:w 101, 310, 310,
 356, 503, 509, 532, 675, 724, 725,
 733, 738, 742, 754, 760, 761, 767,
 770, 776, 783, 808, 809, 818, 826,
 834, 897, 902, 915, 961, 1193, 1779,
 2773, 2775, 4609, 4616, 5063, 5064,
 5076, 5094, 5101, 5124, 5125, 5126,
 5138, 5174, 5688, 5812, 6174, 6250,
 6258, 6273, 6301, 8047, 8059, 8195,
 8208, 8209, 8212, 8213, 8226, 8227,
 8234, 8235, 8236, 8242, 8243, 8244,
 8258, 8260, 8261, 8278, 8281, 8282,
 8283, 8290, 8403, 8407, 8437, 8566,
 8567, 8568, 8594, 8804, 8837, 9029,
 9064, 9074, 9190, 9193, 9316, 9698,
 10537, 10538, 13201, 13210, 14302,
 14493, 14530, 15986, 15989, 16014,
 16015, 16061, 16066, 16110, 16314,
 16315, 16316, 16317, 16318, 16332,
 16480, 16541, 16559, 16852, 16982,
 16996, 16998, 17000, 17003, 17039,
 17177, 17207, 17208, 17245, 17253,
 17384, 17389, 17391, 17400, 17404,
 17441, 17449, 17452, 17458, 17469,
 17480, 17486, 17487, 17490, 17533,
 17543, 17545, 17561, 17563, 17586,
```

17600, 17679, 17681, 17755, 17843,  
 18544, 18577, 18932, 18933, 18934,  
 18936, 18982, 18985, 18988, 19011,  
 19013, 19034, 19036, 19045, 19047,  
 19051, 19069, 19076, 19082, 19092,  
 19094, 19108, 19116, 19124, 19168,  
 19170, 19186, 19188, 19191, 19194,  
 19248, 19256, 19258, 19260, 19262,  
 19265, 19268, 19270, 19289, 19291,  
 19295, 19301, 19303, 19307, 19329,  
 19332, 19340, 19342, 19345, 19346,  
 19347, 19348, 19363, 19366, 19369,  
 19372, 19381, 19384, 19387, 19390,  
 19397, 19399, 19405, 19413, 19415,  
 19417, 19443, 19445, 19454, 19456,  
 19460, 19477, 19498, 19502, 19514,  
 19517, 19520, 19523, 19526, 19529,  
 19532, 19535, 19539, 19551, 19555,  
 19559, 19562, 19583, 19585, 19587,  
 19597, 19621, 19624, 19636, 19638,  
 19644, 19647, 19664, 19684, 19734,  
 19739, 19741, 19748, 19751, 19754,  
 19757, 19760, 19763, 19772, 19784,  
 19792, 19794, 19804, 19806, 19813,  
 19822, 19824, 19827, 19830, 19833,  
 19836, 19849, 19851, 19859, 19861,  
 19869, 19871, 19881, 19884, 19887,  
 19894, 19909, 19927, 19930, 19986,  
 20000, 20002, 20008, 20021, 20023,  
 20025, 20049, 20065, 20072, 20073,  
 20117, 20119, 20120, 20121, 20162,  
 20164, 20201, 20208, 20215, 20236,  
 20238, 20240, 20242, 20255, 20259,  
 20260, 20261, 20262, 20263, 20268,  
 20273, 20275, 20281, 20298, 20299,  
 20300, 20301, 20302, 20303, 20308,  
 20310, 20312, 20314, 20316, 20321,  
 20323, 20325, 20327, 20329, 20331,  
 20353, 20361, 20377, 20382, 20386,  
 20445, 20494, 20562, 20571, 20579,  
 20590, 20592, 20595, 20598, 20687,  
 20723, 20725, 20728, 20731, 20734,  
 20737, 20744, 20747, 20749, 20753,  
 20775, 20777, 20809, 20879, 20889,  
 20894, 20904, 21046, 21078, 21087,  
 21319, 21320, 21331, 21334, 21337,  
 21340, 21343, 21346, 21349, 21352,  
 21355, 21373, 21383, 21392, 21410,  
 21419, 21426, 21436, 21480, 21489,  
 21524, 21567, 21584, 21640, 21651,  
 21662, 21872, 21948, 21995, 22040,  
 22048, 22050, 22052, 22144, 22167,  
 22221, 22261, 22273, 22284, 22285,  
 22315, 22318, 22321, 22323, 22325,  
 22332, 22335, 22343, 22348, 22353,  
 22456, 22518, 22526, 22539, 22540,  
 22541, 22551, 23067, 23379, 23391,  
 23570, 23612, 23614, 23624, 23632,  
 23651, 23653, 23661, 24124, 24618,  
 24624, 24656, 24658, 24667, 24668,  
 24792, 25217, 25232, 26032, 26033,  
 26044, 26566, 26597, 26599, 29332,  
 29464, 32239, 32249, 32497, 32505  
 \int\_while\_do:nn ..... 95, 8507  
 \int\_while\_do:nNn ..... 95, 8535  
 \int\_zero:N 91, 91, 7753, 8334, 8339,  
     13167, 15101, 16052, 16081, 16178,  
     22503, 23425, 23426, 23427, 23526,  
     24386, 24587, 25034, 25342, 25422,  
     25779, 25792, 25822, 26091, 26471  
 \int\_zero\_new:N ..... 91, 8338  
 \c\_max\_int ..... 100, 198,  
     727, 923, 982, 1033, 8998, 22326,  
     23890, 23904, 25876, 27116, 27122  
 \c\_nine ..... 32733  
 \c\_one ..... 32717  
 \c\_one\_int ..... 100,  
     8365, 8367, 8369, 8371, 8997, 16110  
 \c\_seven ..... 32729  
 \c\_six ..... 32727  
 \c\_sixteen ..... 32747  
 \c\_ten ..... 32735  
 \c\_thirteen ..... 32741  
 \c\_three ..... 32721  
 \g\_tmpa\_int ..... 100, 9009  
 \l\_tmpa\_int ..... 2, 100, 229, 9009  
 \g\_tmpb\_int ..... 100, 9009  
 \l\_tmpb\_int ..... 2, 100, 9009  
 \c\_twelve ..... 32739  
 \c\_two ..... 32719  
 \c\_zero ..... 32715  
 \c\_zero\_int ..... 100, 314, 325,  
     325, 399, 1469, 1777, 1779, 3783,  
     5810, 8291, 8292, 8303, 8334, 8335,  
     8387, 8395, 8572, 8575, 8997, 9369,  
     9371, 9572, 9701, 13526, 14344,  
     15971, 16097, 22167, 22296, 22360  
 int internal commands:  
 \\_\_int\_abs:N ..... 8224  
 \\_\_int\_case:nnTF ..... 8461  
 \\_\_int\_case:nw ..... 8461  
 \\_\_int\_case\_end:nw ..... 8461  
 \\_\_int\_compare:nnN ..... 510, 8400  
 \\_\_int\_compare:NNw ..... 509, 510, 8400  
 \\_\_int\_compare:Nw ..... 508, 509, 510, 8400  
 \\_\_int\_compare:w ..... 509, 8400  
 \\_\_int\_compare\_!=:NNw ..... 8400  
 \\_\_int\_compare\_<:NNw ..... 8400

```

__int_compare_<=:NNw 8400
__int_compare_=:NNw 8400
__int_compare_==:NNw 8400
__int_compare_>:NNw 8400
__int_compare_>=:NNw 8400
__int_compare_end_=:NNw ... 510, 8400
__int_compare_error:
..... 508, 509, 8385, 8403, 8405
__int_compare_error:Nw
..... 508, 509, 510, 8385, 8425
__int_constdef:Nw 8301
__int_div_truncate:NwNw 8256
__int_eval:w
..... 305, 503, 504, 509, 8195,
8208, 8209, 8213, 8227, 8235, 8236,
8243, 8244, 8258, 8260, 8261, 8278,
8281, 8282, 8283, 8290, 8319, 8353,
8355, 8357, 8359, 8377, 8379, 8403,
8437, 8455, 8493, 8501, 8566, 8567,
8568, 8594, 8777, 8804, 8810, 8837
__int_eval_end:
..... 8195, 8208, 8213, 8227,
8262, 8278, 8284, 8293, 8319, 8353,
8355, 8357, 8359, 8377, 8379, 8455,
8493, 8501, 8777, 8804, 8810, 8837
__int_from_alpha:N 521, 8894
__int_from_alpha:nN 521, 8894
__int_from_base:N 521, 8911
__int_from_base:nnN 521, 8911
__int_from_roman>NN 8953
\c__int_from_roman_C_int 8939
\c__int_from_roman_c_int 8939
\c__int_from_roman_D_int 8939
\c__int_from_roman_d_int 8939
__int_from_roman_error:w 8953
\c__int_from_roman_I_int 8939
\c__int_from_roman_i_int 8939
\c__int_from_roman_L_int 8939
\c__int_from_roman_l_int 8939
\c__int_from_roman_M_int 8939
\c__int_from_roman_m_int 8939
\c__int_from_roman_V_int 8939
\c__int_from_roman_v_int 8939
\c__int_from_roman_X_int 8939
\c__int_from_roman_x_int 8939
__int_if_recursion_tail_stop:N ..
..... 8205, 8966
__int_if_recursion_tail_stop_-
do:Nn 8205, 8905, 8922, 8969
\l__int_internal_a_int 9013
\l__int_internal_b_int 9013
\c__int_max_constdef_int 8301
__int_maxmin:wwN 8224
__int_mod:ww 8256
__int_pass_signs:wn
..... 520, 8884, 8898, 8915
__int_pass_signs_end:wn 8884
__int_show:nn 8989
__int_sign:Nw 8210
__int_step:NNnnnn 8600
__int_step:NwnnnN 8563
__int_step:wwwN 8563
__int_to_Base:nn 8718
__int_to_base:nn 8718
__int_to_Base:nnN 8718
__int_to_base:nnN 8718
__int_to_Base:nnnN 8718
__int_to_base:nnnN 8718
__int_to_Letter:n 8718
__int_to_letter:n 8718
__int_to_roman:N 8848
__int_to_roman:w
..... 509, 519, 1450, 8195, 8413, 8851, 8861
__int_to_Roman_aux:N 8860, 8863, 8866
__int_to_Roman_c:w 8848
__int_to_roman_c:w 8848
__int_to_Roman_d:w 8848
__int_to_roman_d:w 8848
__int_to_Roman_i:w 8848
__int_to_roman_i:w 8848
__int_to_Roman_l:w 8848
__int_to_roman_l:w 8848
__int_to_Roman_m:w 8848
__int_to_roman_m:w 8848
__int_to_Roman_Q:w 8848
__int_to_roman_Q:w 8848
__int_to_Roman_v:w 8848
__int_to_roman_v:w 8848
__int_to_Roman_x:w 8848
__int_to_roman_x:w 8848
__int_to_symbols:nnnn 8634
__int_use_none_delimit_by_s_-
stop:w 8202, 8435
intarray commands:
\intarray_const_from_clist:Nn ...
..... 198, 16078, 20500, 21102
\intarray_count:N
..... 198, 198, 199, 306, 306,
5772, 5835, 15977, 15980, 15982,
15983, 15986, 15995, 16005, 16053,
16076, 16097, 16122, 16179, 22428
\intarray_gset:Nnn
..... 198, 306, 723, 725,
5756, 5768, 5781, 5787, 16008, 22595
\intarray_gset_rand:Nn ... 267, 16127
\intarray_gset_rand:Nnn .. 267, 16127
\intarray_gzero:N 198, 16050

```

```

\intarray_item:Nn 199, 306, 723, 725, 5766,
5771, 5805, 5834, 5842, 16060, 16076
\intarray_log:N 199, 16112
\intarray_new:Nn ... 198, 722, 725,
5754, 5763, 15964, 22420, 22421,
22422, 22593, 23847, 23848, 23849,
25768, 25769, 26447, 26448, 26449
\intarray_rand_item:N ... 199, 16075
\intarray_show:N 199, 726, 16112
\intarray_to_clist:N 267, 16093
intarray internal commands:
 __intarray_bounds:NNnTF
 15990, 16020, 16071
 __intarray_bounds_error:NNnw . 15990
 __intarray_const_from_clist:nN .
 16078
 __intarray_count:w
 .. 15957, 15976, 15986, 16084, 16105
 __intarray_entry:w
 15957, 16009, 16056, 16061
 \g_intarray_font_int
 15961, 15967, 15969
 __intarray_gset:Nnn
 16008
 __intarray_gset:Nww ... 16012, 16018
 __intarray_gset_all_same:Nn . 16127
 __intarray_gset_overflow:Nnn .. 16008
 __intarray_gset_overflow:NNnn ..
 16032, 16040, 16044
 __intarray_gset_overflow_-
 test:nw 725, 727, 16022,
 16029, 16037, 16090, 16146, 16153
 __intarray_gset_rand:Nnn 16127
 __intarray_gset_rand_auxi:Nnnn .
 16127
 __intarray_gset_rand_auxii:Nnnn .
 16127
 __intarray_gset_rand_auxiii:Nnnn
 16127
 __intarray_item:Nn 16060
 __intarray_item:Nw ... 16064, 16069
 \I_intarray_loop_int 15959,
 16052, 16055, 16056, 16081, 16084,
 16089, 16091, 16178, 16181, 16182
 __intarray_new:N 15964, 16080
 __intarray_show>NN 16112, 16114, 16116
 __intarray_signed_max_dim:n ...
 15988, 16047, 16048
 \c_intarray_sp_dim
 15960, 15969, 16009
 __intarray_to_clist:Nn 16093, 16123
 __intarray_to_clist:w 16093
\interactionmode 547
\interlinepenalties 548
\interlinepenalty 328
ior commands:
 \ior_close:N 158, 159, 159,
 267, 12766, 12785, 13666, 13679,
 14113, 29308, 29341, 29378, 29395
 \ior_get:NN 159,
 160, 160, 161, 161, 267, 12826, 12906
 \ior_get:NNTF 160, 12826, 12827
 \ior_get_str:NN 32791
 \ior_get_term:nN 267, 12860
 \ior_if_eof:N 636
 \ior_if_eof:NTF 162, 12810, 12832,
 12852, 12892, 12911, 13676, 13690
 \ior_if_eof_p:N 162, 12810
 \ior_list_streams: 32793
 \ior_log_list: 159, 12797, 32796
 \ior_log_streams: 32795
 \ior_map_break: 161, 12875, 12893,
 12900, 12912, 12918, 29303, 29374
 \ior_map_break:n 162, 12875
 \ior_map_inline:Nn 161, 161, 12879, 14111
 \ior_map_variable:NNn 161, 12905, 29300
 \ior_new:N 158, 12735, 12737, 12738, 13692, 29249
 \ior_open:NnTF 159, 12740, 12743
 \ior_shell_open:Nn . 267, 14100, 32353
 \ior_show_list: ... 159, 12797, 32794
 \ior_str_get>NN 159, 160, 267, 12839, 12908, 32792
 \ior_str_get:NNTF .. 160, 12839, 12840
 \ior_str_get_term:nN 267, 12860
 \ior_str_map_inline:Nn 161, 161, 12879, 29335, 29365
 \ior_str_map_variable:NNn 161, 12905
 \c_term_ior 33017
 \g_tmtpa_ior 165, 12737
 \g_tmtpb_ior 165, 12737
ior internal commands:
 \l_ior_file_name_t1 12742, 12745, 12747
 __ior_get:NN ... 12826, 12861, 12880
 __ior_get_term:NnN 12860
 \l_ior_internal_t1 12717, 12898, 12902
 __ior_list:N 12797
 __ior_map_inline>NNn 12879
 __ior_map_inline>NNNN 12879
 __ior_map_inline_loop>NNNN .. 12879
 __ior_map_variable>NNNN 12905

```

\\_\_ior\_map\_variable\_loop:NNNn . 12905  
 \\_\_ior\_new:N ..... 632, 12753, 12770  
 \\_\_ior\_new\_aux:N ..... 12757, 12761  
 \\_\_ior\_open\_stream:Nn ..... 12764  
 \\_\_ior\_shell\_open:nN ..... 32353  
 \\_\_ior\_str\_get:NN 12839, 12863, 12882  
 \l\_\_ior\_stream\_tl .....  
     ..... 12720, 12767, 12771, 12778  
 \g\_\_ior\_streams\_prop .....  
     ..... 12721, 12779, 12790, 12804  
 \g\_\_ior\_streams\_seq .....  
     ..... 12719, 12767, 12791, 12792  
 \c\_\_ior\_term\_ior ..... 12718,  
     12735, 12787, 12793, 12814, 12870  
 \c\_\_ior\_term\_noprompt\_ior .....  
     ..... 12859, 12869  
 iow commands:  
 \iow\_allow\_break: .....  
     ..... 164, 266, 13074, 13116, 13121  
 \iow\_allow\_break:n ..... 643  
 \iow\_char:N ..... 163,  
     5874, 12409, 12411, 12412, 12444,  
     12538, 12571, 13035, 20606, 22905,  
     22907, 22908, 22911, 22913, 22914,  
     22917, 22919, 22920, 22921, 22925,  
     22932, 23319, 23322, 23323, 23347,  
     23348, 23355, 23356, 24110, 24112,  
     24114, 24116, 24118, 24120, 24724,  
     24725, 25266, 25379, 25380, 25381,  
     25402, 26701, 26704, 26705, 26710,  
     26744, 26753, 26757, 26762, 26782,  
     26784, 26785, 26787, 26790, 26792,  
     26799, 26803, 26806, 26807, 26810,  
     26812, 26816, 26818, 26824, 26826,  
     26830, 26832, 26836, 26841, 26843,  
     26885, 26887, 26892, 26894, 26900,  
     26905, 26910, 26914, 26924, 26927,  
     26931, 26932, 26936, 26944, 27001  
 \iow\_close:N .. 159, 159, 12964, 12982  
 \iow\_indent:n .. 164, 164, 643, 644,  
     5872, 6197, 6383, 12355, 12458,  
     13085, 13117, 13122, 16727, 16739  
 \l\_\_iow\_line\_count\_int .....  
     . 164, 164, 644, 975, 11870, 13036,  
     13126, 13131, 13169, 23771, 23775  
 \iow\_list\_streams: ..... 32797  
 \iow\_log:n .....  
     ... 162, 1887, 4706, 12068, 12083,  
     12084, 12090, 13030, 32818, 32898  
 \iow\_log\_list: .... 159, 12994, 32800  
 \iow\_log\_streams: ..... 32799  
 \iow\_new:N ... 158, 12953, 12955, 12956  
 \iow\_newline: ..... 163,  
     163, 163, 164, 307, 412, 611, 640,  
     11892, 13034, 13114, 13123, 13129,  
     14029, 23721, 25313, 28907, 28908,  
     28909, 32171, 32173, 32176, 32183  
 \iow\_now:Nn .....  
     ... 162, 162, 162, 163, 163, 9619,  
     13024, 13030, 13031, 13032, 13033  
 \iow\_open:Nn ..... 159, 12960  
 \iow\_shipout:Nn .....  
     ... 163, 163, 163, 640, 9646, 13009  
 \iow\_shipout\_x:Nn .....  
     ... 163, 163, 163, 640, 13006  
 \iow\_show\_list: ... 159, 12994, 32798  
 \iow\_term:n ..... 162, 267,  
     1887, 11904, 12046, 12061, 12062,  
     12096, 12122, 13030, 26991, 32820  
 \iow\_wrap:nnnn ..... 163, 163,  
     164, 164, 164, 266, 412, 644, 1191,  
     4691, 4706, 11868, 11871, 11883,  
     12047, 12069, 12088, 12094, 12101,  
     13077, 13083, 13088, 13100, 13103  
 \c\_log\_iow .....  
     165, 637, 12924, 12984, 13030, 13031  
 \c\_term\_iow ..... 165, 637, 12924,  
     12953, 12984, 12990, 13032, 13033  
 \g\_tmqa\_iow ..... 165, 12955  
 \g\_tmrb\_iow ..... 165, 12955  
 iow internal commands:  
 \\_\_iow\_allow\_break: 643, 13074, 13116  
 \\_\_iow\_allow\_break\_error: .....  
     ..... 643, 13074, 13121  
 \l\_\_iow\_file\_name\_tl .....  
     ... 12959, 12962, 12966, 12970  
 \\_\_iow\_indent:n ... 643, 13085, 13117  
 \\_\_iow\_indent\_error:n .....  
     ..... 643, 13085, 13122  
 \l\_\_iow\_indent\_int ..... 13053,  
     13167, 13185, 13297, 13314, 13322  
 \l\_\_iow\_indent\_tl .. 13053, 13168,  
     13184, 13296, 13315, 13323, 13324  
 \l\_\_iow\_line\_break\_bool .....  
     13057, 13163, 13291, 13305, 13313,  
     13321, 13329, 13331, 13336, 13338  
 \l\_\_iow\_line\_part\_tl .....  
     ... 646, 647, 649, 13055, 13165,  
     13177, 13198, 13256, 13259, 13290,  
     13304, 13306, 13312, 13320, 13343  
 \l\_\_iow\_line\_target\_int .....  
     ... 649, 13039, 13125,  
     13127, 13130, 13292, 13297, 13332  
 \l\_\_iow\_line\_tl 13055, 13164, 13181,  
     13271, 13287, 13303, 13304, 13312,  
     13320, 13342, 13343, 13348, 13350  
 \\_\_iow\_list:N ..... 12994  
 \\_\_iow\_new:N ..... 12957, 12968

```

\l__iow_newline_tl 13038,
 13123, 13124, 13126, 13129, 13347
\l__iow_one_indent_int 13040, 13314, 13322
\l__iow_one_indent_tl 642, 13040, 13315
\l__iow_open_stream:Nn 12960
\l__iow_set_indent:n 641, 13040
\l__iow_stream_tl 12934, 12965, 12969, 12976
\g__iow_streams_prop 12935, 12977, 12987, 13001
\g__iow_streams_seq 12933, 12965, 12988, 12989
\l__iow_tmp:w 647, 13171,
 13195, 13252, 13284, 13352, 13360
\l__iow_unindent:w .. 641, 13040, 13324
\l__iow_use_i_delimit_by_s-
 stop:nw 12951, 13156
\l__iow_with:nNnn 13012
\l__iow_wrap_allow_break:n 13301
\c__iow_wrap_allow_break_marker-
 tl 13059, 13079
\l__iow_wrap_break:w 13238, 13252
\l__iow_wrap_break_end:w .. 647, 13252
\l__iow_wrap_break_first:w 13252
\l__iow_wrap_break_loop:w 13252
\l__iow_wrap_break_none:w 13252
\l__iow_wrap_chunk:nw 13169, 13171,
 13307, 13308, 13316, 13325, 13332
\l__iow_wrap_do: 13133, 13138
\l__iow_wrap_end:n 13327
\l__iow_wrap_end_chunk:w
 645, 13189, 13196, 13288
\c__iow_wrap_end_marker_tl
 13059, 13143
\l__iow_wrap_fix_newline:w 13138
\l__iow_wrap_indent:n 13310
\c__iow_wrap_indent_marker_tl ...
 13059, 13093
\l__iow_wrap_line:nw
 645, 648, 13183, 13187, 13196, 13295
\l__iow_wrap_line_aux:Nw 13196
\l__iow_wrap_line_end:NnnnnnnnnN 13196
\l__iow_wrap_line_end:nw
 647, 13196, 13272, 13273, 13282
\l__iow_wrap_line_loop:w 13196
\l__iow_wrap_line_seven:nnnnnnn 13196
\c__iow_wrap_marker_tl
 642, 645, 13059, 13195
\l__iow_wrap_newline:n 13327
\c__iow_wrap_newline_marker_tl ...
 644, 13059, 13156
\l__iow_wrap_next:nw
 13171, 13250, 13292
\l__iow_wrap_next_line:w 13244, 13285
\l__iow_wrap_start:w 13138
\l__iow_wrap_store_do:n
 13243, 13330, 13337, 13340
\l__iow_wrap_tl 644, 644, 649, 650, 13058,
 13120, 13135, 13140, 13142, 13145,
 13147, 13150, 13166, 13344, 13346
\l__iow_wrap_trim:N 650, 13273, 13304, 13330, 13337, 13352
\l__iow_wrap_trim:w 13352
\l__iow_wrap_trim_aux:w 13352
\l__iow_wrap_unindent:n 13310
\c__iow_wrap_unindent_marker_tl .
 13059, 13095
\itshape 31650

J
\j 31393, 31736, 31931, 32010
\jcharwidowpenalty 1134
\jfam 1135
\jfont 1136
\jis 1137
job commands:
 \c_job_name_tl 32711
\jobname 329

K
\k 29543, 31809, 31884,
 31885, 31902, 31903, 31925, 31926,
 31927, 31982, 31983, 32008, 32009
\kanjiskip 1138
\kansuji 1139
\kansujichar 1140
\kcatcode 1141
\kchar 1174
\kchardef 1175
\kern 330
kernel internal commands:
 \l_kernel_backend_align_begin: . 312
 \l_kernel_backend_align_end: .. 312
 \g_kernel_backend_header_bool . 312
 \l_kernel_backend_literal:n ... 311
 \l_kernel_backend_literal_pdf:n 311
 \l_kernel_backend_literal_-
 postscript:n 311
 \l_kernel_backend_literal_svg:n 311
 \l_kernel_backend_matrix:n 312
 \l_kernel_backend_postscript:n . 312
 \l_kernel_backend_scope_begin: . 312
 \l_kernel_backend_scope_end: .. 312
 \l_kernel_chk_cs_exist:N 305

```

```

__kernel_chk_defined:NTF
 305, 567, 606,
 2146, 2165, 4684, 8180, 9041, 9142,
 10387, 11741, 16118, 18459, 26401
__kernel_chk_expr:nNnN 305
__kernel_chk_if_free_cs:N . . .
 579, 607,
 1891, 1906, 1954, 3285, 3588, 3594,
 3599, 7535, 8297, 8315, 9084, 10881,
 10883, 10893, 11412, 14213, 14557,
 14649, 15966, 22582, 22598, 27012
\l__kernel_color_stack_int 312
__kernel_cs_parm_from_arg_-
 count:nnTF . . . 305, 1616, 1972, 2019
__kernel_dependency_version_-
 check:Nn 305, 14079
__kernel_dependency_version_-
 check:nn 305, 14079
__kernel_deprecation_code:nn . .
 305,
 1204, 1566, 32611, 32646, 32653, 32654
__kernel_deprecation_error:Nnn . .
 1204, 32614, 32673
\g__kernel_deprecation_undo_-
 recent_bool . . . 32582, 32596, 32623
__kernel_exp_not:w
 305, 349, 399, 2545,
 2547, 2551, 2555, 2558, 2561, 2566,
 3595, 3624, 3625, 3632, 3633, 3652,
 3654, 3658, 3660, 3672, 3677, 3683,
 3684, 3688, 3692, 3697, 3703, 3704,
 3708, 3718, 3722, 3728, 3729, 3733,
 3735, 3739, 3745, 3746, 3750, 3909,
 4236, 4241, 4297, 4302, 4309, 4334,
 4527, 4687, 8418, 29577, 29900, 31434
\l__kernel_expl_bool
 146, 149, 164, 178, 1416
\c__kernel_expl_date_t1
 670, 1416, 14083, 14086, 14126, 14130
__kernel_file_input_pop: 306, 13910
__kernel_file_input_push:n . .
 306, 13910
__kernel_file_missing:n . .
 306, 12740, 13905, 13914
__kernel_file_name_expand_-
 group:nw 13406
__kernel_file_name_expand_-
 loop:w 13406
__kernel_file_name_expand_N_-
 type:Nw 13406
__kernel_file_name_expand_-
 space:w 13406
__kernel_file_name_quote:n 1196,
 12783, 12979, 13497, 13536, 13925
__kernel_file_name_quote:nw . 13497
__kernel_file_name_sanitize:n .
 665, 12963,
 13406, 13550, 13653, 13908, 13964
__kernel_file_name_sanitize:nN .
 306, 306
__kernel_file_name_strip_-
 quotes:n 13406
__kernel_file_name_strip_-
 quotes:nnn 13406
__kernel_file_name_strip_-
 quotes:nnnw 13406
__kernel_file_name_trim_-
 spaces:n 13406
__kernel_file_name_trim_-
 spaces:nw 13406
__kernel_file_name_trim_spaces_-
 aux:n 13406
__kernel_file_name_trim_spaces_-
 aux:w 13406
__kernel_if_debug:TF
 1553, 32622, 32634
__kernel_int_add:nnn 306, 8288, 22326
__kernel_intarray_gset:Nnn 306,
 724, 15971, 15983, 16008, 16091,
 16182, 22490, 22491, 22493, 22497,
 22498, 22499, 22601, 22602, 22636,
 22639, 25818, 25902, 25904, 25906,
 25926, 25929, 25984, 26504, 26506,
 26512, 26520, 26522, 26525, 26586,
 26588, 26590, 26603, 26609, 26613
__kernel_intarray_item:Nn . .
 306, 725,
 897, 16060, 16108, 20586, 20592,
 20595, 20598, 21332, 21335, 21338,
 21341, 21344, 21347, 21350, 21353,
 21356, 22539, 22540, 22541, 22688,
 22691, 25897, 25918, 25921, 25937,
 25964, 26025, 26026, 26049, 26050,
 26058, 26064, 26066, 26073, 26079,
 26081, 26135, 26139, 26509, 26636
__kernel_ior_open:Nn . 306, 1196,
 12747, 12764, 13674, 13689, 32366
__kernel_iow_with:Nnn 307,
 412, 610, 640, 4695, 4697, 11905,
 11907, 12124, 12126, 13012, 13026
__kernel_msg_critical:nn 307, 12314
__kernel_msg_critical:nnn 307, 12314
__kernel_msg_critical:nnnn . .
 307, 12314
__kernel_msg_critical:nnnnn . .
 307, 12314
__kernel_msg_critical:nnnnnn . .
 307, 12314
__kernel_msg_critical:nnnnnnn . .
 307, 12314

```

```

__kernel_msg_error:nn 308,
1868, 9470, 12314, 22736, 22781,
24366, 24399, 24447, 24450, 24911,
25168, 26269, 26353, 28111, 32357
__kernel_msg_error:nnn
. 308, 1556, 1561, 1629,
1684, 1732, 1737, 1868, 2056, 2063,
2151, 2904, 3178, 3390, 3414, 3418,
3563, 3881, 4793, 5497, 5556, 7748,
7779, 9565, 9715, 11495, 12137,
12314, 13805, 13907, 15012, 15041,
15057, 15225, 15243, 15979, 16189,
16211, 16607, 22415, 22754, 22796,
22802, 23313, 24405, 24607, 25010,
25023, 25062, 25186, 26107, 26114,
26367, 27146, 27736, 28992, 32363
__kernel_msg_error:nnnn
. 308, 1620, 1660, 1751,
1868, 1895, 2021, 2989, 3198, 3221,
3433, 5588, 9492, 9511, 9527, 11769,
12163, 12314, 13076, 14114, 14969,
15022, 15076, 15090, 15234, 15731,
15784, 16603, 23176, 23183, 24584,
24649, 24873, 26273, 26289, 27953
__kernel_msg_error:nnnnn
. 308, 12314, 13087, 16020,
22462, 23329, 26553, 26676, 32680
__kernel_msg_error:nnnnnn
. 308, 3004, 3018, 12314, 16046, 32663
__kernel_msg_expandable_-
error:nn 308,
2575, 7526, 9347, 10544, 10546,
10554, 10560, 10590, 11401, 12646,
14814, 14821, 14861, 17135, 24105
__kernel_msg_expandable_-
error:nnn 308,
2304, 2651, 2711, 2735, 4189, 8123,
8396, 8577, 9699, 10295, 12646,
13478, 13620, 13881, 14448, 17142,
17157, 17162, 17228, 17285, 17324,
17330, 17668, 17673, 17684, 17691,
17782, 17796, 17996, 18049, 18715,
22112, 22119, 22125, 24191, 28983
__kernel_msg_expandable_-
error:nnnn
. 308, 12646, 13082, 16141, 18183,
18204, 18876, 22291, 22381, 24130
__kernel_msg_expandable_-
error:nnnnn 308, 12646, 13099,
16071, 16718, 22158, 22532, 32677
__kernel_msg_expandable_-
error:nnnnnn 308, 12646, 32664
__kernel_msg_fatal:nn 307, 12314
__kernel_msg_fatal:nnn 307, 12314
__kernel_msg_fatal:nnnn 307, 12314
__kernel_msg_info:nn 308, 12320
__kernel_msg_info:nnn 308, 12320
__kernel_msg_info:nnnn 308, 12320
__kernel_msg_info:nnnnn 308, 12320
__kernel_msg_info:nnnnnn 308, 12320
__kernel_msg_new:nnn
. 307, 5846, 5848, 5857, 12268,
12365, 12367, 12369, 12371, 12373,
12443, 12494, 12542, 12562, 12564,
12566, 12568, 12570, 12572, 12574,
12576, 12580, 12583, 12590, 12592,
12599, 12606, 14161, 14864, 14866,
15947, 15962, 16746, 16748, 16750,
16752, 16754, 16756, 16758, 18381,
18383, 18385, 18387, 18389, 18391,
18393, 18395, 18397, 18399, 18401,
18403, 18405, 18407, 18411, 18768,
18770, 18772, 22108, 23785, 26700,
26702, 26707, 26961, 28935, 32685
__kernel_msg_new:nnn 307, 5717,
5850, 5865, 5879, 5885, 5932, 5977,
6067, 6182, 6362, 6369, 6541, 12268,
12323, 12331, 12339, 12346, 12357,
12375, 12384, 12391, 12398, 12405,
12414, 12423, 12430, 12436, 12445,
12452, 12461, 12467, 12474, 12481,
12484, 12496, 12503, 12511, 12518,
12526, 12534, 12553, 12617, 12623,
13625, 14122, 14155, 14167, 14172,
15911, 15914, 15917, 15923, 15929,
15935, 15941, 16580, 16720, 16735,
22898, 22904, 22910, 22916, 22923,
23318, 23335, 23342, 23351, 26713,
26720, 26726, 26736, 26742, 26766,
26773, 26781, 26789, 26796, 26802,
26809, 26815, 26823, 26829, 26835,
26845, 26852, 26861, 26864, 26872,
26878, 26884, 26891, 26898, 26908,
26919, 26929, 26939, 26948, 26954,
28919, 28926, 28929, 28997, 32368
__kernel_msg_set:nnn 307, 12268
__kernel_msg_set:nnnn 307, 12268
__kernel_msg_warning:nn
. 308, 12320, 24901
__kernel_msg_warning:nnn
. 308, 12320, 24817,
24821, 24863, 24925, 24963, 24982
__kernel_msg_warning:nnnn
. 308, 12320, 24514, 24663
__kernel_msg_warning:nnnnn . . .
. 308, 12320, 32636

```

\\_\\_kernel\\_msg\\_warning:nnnnnn . . .  
     . . . . . 308, 12251, 12320  
 \\_\\_kernel\\_patch\_deprecation:nnNNpn  
     . . . . . 1203, 32607, 32699, 32704,  
     32906, 32909, 32914, 32918, 32923,  
     32925, 32927, 32929, 32931, 32933,  
     32935, 32938, 32940, 32943, 32947,  
     32956, 32967, 32977, 32980, 32983,  
     32986, 32989, 32992, 32995, 32997,  
     32999, 33001, 33003, 33005, 33007,  
     33009, 33011, 33013, 33015, 33017  
 \\_\\_kernel\\_prefix\\_arg\\_replacement:wN  
     . . . . . 2187  
 \g\\_kernel\\_prg\\_map\\_int . . . . .  
     . . . . . 308, 396, 514,  
     679, 972, 1416, 4135, 4137, 4139,  
     4142, 4957, 4959, 4963, 4968, 7985,  
     7986, 7992, 7993, 8035, 8037, 8039,  
     8041, 8606, 8609, 8617, 8620, 8631,  
     9372, 10207, 10209, 10211, 10214,  
     11705, 11706, 11711, 11713, 12885,  
     12887, 12894, 14466, 14469, 14473,  
     14476, 14487, 18745, 18748, 18752,  
     18755, 18766, 23672, 23674, 23694  
 \\_\\_kernel\\_primitive:NN . . . . .  
     . 186, 191, 192, 193, 194, 195, 196,  
     197, 198, 199, 200, 201, 202, 203,  
     204, 205, 206, 207, 208, 209, 210,  
     211, 212, 213, 214, 215, 216, 217,  
     218, 219, 220, 221, 222, 223, 224,  
     225, 226, 227, 228, 229, 230, 231,  
     232, 233, 234, 235, 236, 237, 238,  
     239, 240, 241, 242, 243, 244, 245,  
     246, 247, 248, 249, 250, 251, 252,  
     253, 254, 255, 256, 257, 258, 259,  
     260, 261, 262, 263, 264, 265, 266,  
     267, 268, 269, 270, 271, 272, 273,  
     274, 275, 276, 277, 278, 279, 279,  
     280, 281, 282, 283, 284, 285, 286,  
     287, 288, 289, 290, 291, 292, 293,  
     294, 295, 296, 297, 298, 299, 300,  
     301, 302, 303, 304, 305, 306, 307,  
     308, 309, 310, 311, 312, 313, 314,  
     315, 316, 317, 318, 319, 320, 321,  
     322, 323, 324, 325, 326, 327, 328,  
     329, 330, 331, 332, 333, 334, 335,  
     336, 337, 338, 339, 340, 341, 342,  
     343, 344, 345, 346, 347, 348, 349,  
     350, 351, 352, 353, 354, 355, 356,  
     357, 358, 359, 360, 361, 362, 363,  
     364, 365, 366, 367, 368, 369, 370,  
     371, 372, 373, 374, 375, 376, 377,  
     378, 379, 380, 381, 382, 383, 384,  
     385, 386, 387, 388, 389, 390, 391,

794, 796, 797, 798, 799, 800, 801,  
802, 803, 804, 805, 807, 808, 809,  
810, 811, 812, 813, 814, 815, 816,  
817, 818, 819, 820, 821, 822, 823,  
824, 825, 826, 827, 828, 829, 830,  
831, 832, 833, 834, 835, 836, 837,  
838, 839, 840, 841, 842, 843, 844,  
845, 846, 847, 848, 849, 850, 851,  
852, 853, 854, 855, 856, 857, 858,  
859, 860, 861, 862, 863, 864, 865,  
866, 867, 868, 869, 870, 871, 872,  
873, 874, 875, 876, 877, 878, 879,  
880, 881, 882, 883, 884, 885, 886,  
887, 888, 889, 890, 891, 892, 893,  
894, 895, 896, 897, 898, 899, 900,  
901, 902, 903, 904, 906, 907, 908,  
909, 910, 911, 912, 913, 914, 915,  
916, 917, 918, 919, 920, 922, 924,  
926, 927, 928, 929, 930, 931, 932,  
933, 934, 935, 936, 937, 938, 939,  
940, 941, 942, 943, 944, 945, 946,  
947, 948, 949, 950, 951, 952, 953,  
954, 955, 956, 957, 958, 959, 960,  
961, 962, 963, 964, 965, 966, 967,  
968, 969, 971, 973, 974, 975, 976,  
978, 979, 980, 981, 983, 984, 986,  
988, 989, 990, 991, 992, 994, 996,  
997, 998, 999, 1001, 1002, 1003,  
1004, 1005, 1006, 1007, 1008, 1009,  
1010, 1011, 1012, 1013, 1014, 1015,  
1016, 1017, 1018, 1019, 1020, 1021,  
1022, 1023, 1024, 1025, 1026, 1027,  
1028, 1029, 1030, 1031, 1032, 1033,  
1034, 1035, 1036, 1037, 1038, 1040,  
1042, 1043, 1045, 1047, 1048, 1049,  
1050, 1052, 1053, 1054, 1056, 1058,  
1060, 1061, 1062, 1063, 1064, 1065,  
1066, 1067, 1068, 1069, 1070, 1071,  
1073, 1075, 1076, 1077, 1078, 1079,  
1080, 1081, 1082, 1083, 1084, 1085,  
1086, 1087, 1088, 1089, 1090, 1091,  
1093, 1095, 1096, 1097, 1098, 1099,  
1100, 1101, 1102, 1103, 1104, 1105,  
1106, 1107, 1108, 1109, 1110, 1111,  
1112, 1113, 1114, 1115, 1116, 1117,  
1118, 1119, 1120, 1121, 1122, 1123,  
1124, 1125, 1126, 1127, 1128, 1129,  
1130, 1131, 1132, 1133, 1134, 1135,  
1136, 1137, 1138, 1139, 1140, 1141,  
1142, 1143, 1144, 1145, 1146, 1147,  
1148, 1149, 1150, 1151, 1152, 1153,  
1154, 1156, 1158, 1159, 1160, 1161,  
1162, 1164, 1165, 1166, 1167, 1168,  
1169, 1170, 1171, 1172, 1173, 1174,  
1175, 1176, 1177, 1178, 1179, 1180,  
1181, 1182, 1183, 1184, 1185, 1186  
\\_\\_kernel\_quark\_new\_conditional:Nn  
..... 310, 3385, 3765,  
10412, 13401, 14941, 23862, 29403  
\\_\\_kernel\_quark\_new\_test:N ..  
..... 309, 372,  
373, 375, 377, 3385, 3764, 4724,  
4725, 8205, 8206, 9120, 9818, 9819,  
11408, 13404, 13405, 29408, 31431  
\\_\\_kernel\_randint:n ..  
..... 310, 310, 310, 727, 924,  
927, 16172, 22133, 22145, 22303, 22388  
\\_\\_kernel\_randint:nn ..  
..... 310, 727, 16168, 22307, 22311, 22386  
\c\_\\_kernel\_randint\_max\_int ..  
927, 1416, 16165, 22132, 22301, 22385  
\\_\\_kernel\_register\_log:N ..  
..... 310, 2155, 8993, 14543,  
14544, 14637, 14638, 14705, 14706  
\\_\\_kernel\_register\_show:N ..  
..... 310, 310,  
412, 2155, 8989, 14539, 14633, 14701  
\\_\\_kernel\_register\_show\_aux:NN 2155  
\\_\\_kernel\_register\_show\_aux:nNN 2155  
\\_\\_kernel\_show:NN ..  
..... 2173  
\\_\\_kernel\_str\_to\_other:n ... 311,  
311, 419, 421, 426, 5003, 5055, 5116  
\\_\\_kernel\_str\_to\_other\_fast:n ...  
..... 311, 4964, 4984,  
5026, 5530, 13046, 13142, 24076, 25205  
\\_\\_kernel\_str\_to\_other\_fast\_-  
loop:w ..  
..... 5026  
\\_\\_kernel\_sys\_configuration\_-  
load:n 9477, 9539, 9545, 32900, 32902  
\\_\\_kernel\_tl\_gset:Nn ..  
..... 311, 379, 3584, 3630, 3658, 3660,  
3662, 3691, 3696, 3701, 3708, 3735,  
3738, 3743, 3750, 3868, 3872, 4250,  
4547, 4780, 4784, 5464, 5480, 5530,  
5676, 5728, 5739, 5897, 5946, 5999,  
6005, 6236, 6436, 6593, 7571, 7576,  
7594, 7648, 7688, 7714, 7871, 7914,  
8064, 8074, 9769, 9890, 9917, 9936,  
9979, 10015, 10064, 10103, 11603,  
11626, 18423, 23055, 23561, 24075,  
25203, 32277, 32287, 32470, 32973  
\\_\\_kernel\_tl\_set:Nn ..  
..... 311,  
3584, 3622, 3652, 3654, 3656, 3671,  
3676, 3681, 3688, 3718, 3721, 3726,  
3733, 3866, 3870, 4248, 4545, 4778,  
4782, 5419, 7561, 7566, 7592, 7614,  
7640, 7686, 7712, 7827, 7852, 7869,  
7883, 7911, 8062, 8072, 9888, 9915,

9934, 9977, 10013, 10062, 10101,  
 10603, 11602, 11624, 12771, 12962,  
 12969, 13045, 13120, 13123, 13124,  
 13140, 13145, 13303, 13323, 13342,  
 13344, 13639, 13652, 13687, 13794,  
 13828, 15466, 15508, 15574, 15772,  
 18421, 23964, 23977, 24833, 24838,  
 25103, 25164, 26630, 26660, 28723,  
 32275, 32285, 32450, 32465, 32962  
`\_kernel_tl_to_str:w` . . . . . 311,  
 394, 1443, 3383, 3967, 4064, 4183, 4939  
**keys commands:**  
`\_keys_choice_int` . . . . .  
 . . . . . 187, 189, 191, 191,  
 193, 14915, 15101, 15104, 15109, 15110  
`\_keys_choice_tl` . . . . .  
 . . . . . 187, 189, 191, 193, 14915, 15108  
`\keys_define:nn` . . . . . 186, 12456, 14942  
`\keys_if_choice_exist:nnnTF` . . . . .  
 . . . . . 196, 15880  
`\keys_if_choice_exist_p:nnn` . . . . .  
 . . . . . 196, 15880  
`\keys_if_exist:nnTF` . . . . .  
 . . . . . 195, 720, 15873, 15897  
`\keys_if_exist_p:nn` . . . . . 195, 15873  
`\_keys_key_str` 193, 14918, 15042,  
 15058, 15583, 15584, 15683, 15687,  
 15712, 15715, 15716, 15752, 15809  
`\_keys_key_tl` . . . . . 14918, 15584  
`\keys_log:nn` . . . . . 196, 15888  
`\_keys_path_str` . . . . . 193, 14923,  
 14970, 14989, 14996, 15004, 15005,  
 15006, 15023, 15035, 15037, 15039,  
 15051, 15053, 15055, 15070, 15073,  
 15077, 15085, 15087, 15088, 15091,  
 15106, 15120, 15130, 15135, 15145,  
 15149, 15156, 15161, 15165, 15166,  
 15171, 15177, 15181, 15183, 15198,  
 15209, 15215, 15219, 15235, 15244,  
 15251, 15292, 15574, 15582, 15620,  
 15623, 15662, 15666, 15671, 15680,  
 15694, 15696, 15697, 15701, 15709,  
 15732, 15762, 15785, 15797, 15806  
`\_keys_path_tl` . . 14923, 15006, 15077  
`\keys_set:nn` . . . . . 185,  
 189, 193, 193, 194, 15173, 15177, 15421  
`\keys_set_filter:nnn` . . . . . 195, 15491  
`\keys_set_filter:nnnN` . . . . . 195, 15491  
`\keys_set_filter:nnnnN` . . . . . 195, 15491  
`\keys_set_groups:nnn` . . . . . 195, 15491  
`\keys_set_known:nn` . . . . . 194, 15450  
`\keys_set_known:nnN` . . . . . 194, 711, 15450  
`\keys_set_known:nnnN` . . . . . 194, 15450  
`\keys_show:nn` . . . . . 196, 196, 15888  
`\_keys_value_tl` 193, 14933, 15235,  
 15665, 15669, 15675, 15686, 15697,  
 15716, 15729, 15754, 15764, 15792  
**keys internal commands:**  
`\_keys_bool_set:Nn` . . . . .  
 . . . . . 15031, 15266, 15268, 15270, 15272  
`\_keys_bool_set_inverse:Nn` . . . . .  
 . . . . . 15047, 15274, 15276, 15278, 15280  
`\_keys_check_groups:` . . 15624, 15632  
`\_keys_choice_find:n` . . 15064, 15803  
`\_keys_choice_find:nn` . . . . . 15803  
`\_keys_choice_make:` . . . . .  
 . . . . . 15034, 15050, 15063, 15095, 15282  
`\_keys_choice_make:N` . . . . . 15063  
`\_keys_choice_make_aux:N` . . . . . 15063  
`\_keys_choices_make:nn` . . . . .  
 . . . . . 15094, 15284, 15286, 15288, 15290  
`\_keys_choices_make:Nnn` . . . . . 15094  
`\_keys_cmd_set:nn` . . . . .  
 15035, 15037, 15039, 15051, 15053,  
 15055, 15087, 15088, 15105, 15115,  
 15171, 15177, 15183, 15251, 15292  
`\c_keys_code_root_str` . . . . .  
 . . . . . 717, 14908, 15116,  
 15120, 15165, 15694, 15712, 15728,  
 15742, 15814, 15876, 15884, 15903  
`\_keys_cs_set>NNpn` . . . . .  
 . . . . . 15118, 15302, 15304, 15306,  
 15308, 15310, 15312, 15314, 15316  
`\_keys_default_inherit:` . . . . . 15658  
`\c_keys_default_root_str` . . . . .  
 . . . . . 14908, 15130,  
 15135, 15662, 15666, 15683, 15687  
`\_keys_default_set:n` 15044, 15060,  
 15125, 15318, 15320, 15322, 15324  
`\_keys_define:n` . . . . . 14947, 14951  
`\_keys_define:nn` . . . . . 14947, 14951  
`\_keys_define:nnn` . . . . . 14942  
`\_keys_define_aux:nn` . . . . . 14951  
`\_keys_define_code:n` . . 14965, 15014  
`\_keys_define_code:w` . . . . . 15014  
`\_keys_execute:` . . . . .  
 . . . . . 15588, 15628, 15650, 15654, 15692  
`\_keys_execute:nn` . . . . .  
 . . . . . 15166, 15692, 15815, 15816  
`\_keys_execute_inherit:` 15162, 15692  
`\_keys_execute_unknown:` . . 716, 15692  
`\_keys_filtered_bool` . . . . . 14929,  
 15426, 15433, 15434, 15477, 15483,  
 15484, 15519, 15525, 15526, 15538,  
 15545, 15546, 15627, 15648, 15653  
`\_keys_find_key_module:wNN` . . . . .  
 . . . . . 15181, 15562

\\_\_keys\_find\_key\_module\_auxi:Nw .  
..... 15562  
\\_\_keys\_find\_key\_module\_auxii:Nw .  
..... 15562  
\\_\_keys\_find\_key\_module\_auxiii:Nn .  
..... 15562  
\\_\_keys\_find\_key\_module\_auxiii:Nw .  
..... 15605, 15607  
\\_\_keys\_find\_key\_module\_auxiv:Nw .  
..... 15562  
\l\_\_keys\_groups\_clist ... 14917,  
15142, 15143, 15150, 15622, 15637  
\c\_\_keys\_groups\_root\_str .....  
... 14908, 15145, 15149, 15620, 15623  
\\_\_keys\_groups\_set:n ... 15140, 15342  
\\_\_keys\_inherit:n .... 15153, 15344  
\c\_\_keys\_inherit\_root\_str .....  
..... 14908, 15156,  
15161, 15671, 15680, 15701, 15709  
\l\_\_keys\_inherit\_str .... 14925,  
15164, 15581, 15714, 15805, 15809  
\\_\_keys\_initialise:n .....  
... 15158, 15346, 15348, 15350, 15352  
\\_\_keys\_meta\_make:n ... 15169, 15362  
\\_\_keys\_meta\_make:nn ... 15169, 15364  
\l\_\_keys\_module\_str 14920, 14943,  
14946, 14948, 14990, 15173, 15443,  
15446, 15448, 15565, 15570, 15580,  
15583, 15589, 15728, 15729, 15732  
\\_\_keys\_multichoice\_find:n .....  
..... 15066, 15803  
\\_\_keys\_multichoice\_make: .....  
..... 15063, 15097, 15366  
\\_\_keys\_multichoice\_make:nn ...  
... 15094, 15368, 15370, 15372, 15374  
\l\_\_keys\_no\_value\_bool .....  
..... 14921, 14953,  
14958, 15016, 15232, 15241, 15564,  
15569, 15660, 15753, 15763, 15791  
\l\_\_keys\_only\_known\_bool .....  
... 14922, 15425, 15431, 15432,  
15476, 15481, 15482, 15518, 15523,  
15524, 15537, 15543, 15544, 15724  
\\_\_keys\_parent:n .....  
... 15070, 15073, 15077, 15161,  
15671, 15680, 15701, 15709, 15820  
\\_\_keys\_parent\_auxi:w ..... 15820  
\\_\_keys\_parent\_auxii:w ..... 15820  
\\_\_keys\_parent\_auxiii:n ..... 15820  
\\_\_keys\_parent\_auxiv:w ..... 15820  
\\_\_keys\_prop\_put:Nn .....  
... 15178, 15384, 15386, 15388, 15390  
\\_\_keys\_property\_find:n 14963, 14974  
\\_\_keys\_property\_find\_auxi:w . 14974  
\\_\_keys\_property\_find\_auxii:w . 14974  
\\_\_keys\_property\_find\_auxiii:w . 14974  
\\_\_keys\_property\_find\_auxiv:w . 14974  
\\_\_keys\_property\_find\_err:w ....  
..... 14979, 14987, 15008, 15009  
\l\_\_keys\_property\_str 14928, 14964,  
14967, 14970, 14976, 14977, 15003,  
15011, 15019, 15020, 15023, 15026  
\c\_\_keys\_props\_root\_str .....  
..... 14914, 14964,  
15020, 15026, 15265, 15267, 15269,  
15271, 15273, 15275, 15277, 15279,  
15281, 15283, 15285, 15287, 15289,  
15291, 15293, 15295, 15297, 15299,  
15301, 15303, 15305, 15307, 15309,  
15311, 15313, 15315, 15317, 15319,  
15321, 15323, 15325, 15327, 15329,  
15331, 15333, 15335, 15337, 15339,  
15341, 15343, 15345, 15347, 15349,  
15351, 15353, 15355, 15357, 15359,  
15361, 15363, 15365, 15367, 15369,  
15371, 15373, 15375, 15377, 15379,  
15381, 15383, 15385, 15387, 15389,  
15391, 15393, 15395, 15397, 15399,  
15401, 15403, 15405, 15407, 15409,  
15411, 15413, 15415, 15417, 15419  
\\_\_keys\_quark\_if\_no\_value:NTF ...  
..... 14941, 15748  
\\_\_keys\_quark\_if\_no\_value\_p:N . 14941  
\l\_\_keys\_relative\_t1 .... 14926,  
15428, 15437, 15438, 15479, 15487,  
15488, 15521, 15529, 15530, 15540,  
15549, 15550, 15748, 15758, 15772,  
15773, 15777, 15778, 15786, 15798  
\l\_\_keys\_selective\_bool .....  
... 14929, 15427, 15435, 15436,  
15478, 15485, 15486, 15520, 15527,  
15528, 15539, 15547, 15548, 15586  
\l\_\_keys\_selective\_seq .....  
... 14931, 15555, 15558, 15560, 15635  
\\_\_keys\_set:nn ... 15421, 15480, 15559  
\\_\_keys\_set:nnn ..... 15421  
\\_\_keys\_set\_filter:nnnn ..... 15491  
\\_\_keys\_set\_filter:nnnnnN ..... 15491  
\\_\_keys\_set\_keyval:n .. 15447, 15562  
\\_\_keys\_set\_keyval:nn .. 15447, 15562  
\\_\_keys\_set\_keyval:nnn ..... 15562  
\\_\_keys\_set\_known:nnn ..... 15450  
\\_\_keys\_set\_known:nnnnN ..... 15450  
\\_\_keys\_set\_selective: ..... 15562  
\\_\_keys\_set\_selective:nnn .... 15491  
\\_\_keys\_set\_selective:nnnn ... 15491  
\\_\_keys\_set\_show:Nnn ..... 15888

\\_\_keys\_store\_unused: .....  
     ..... 15629, 15649, 15655, 15692  
 \\_\_keys\_store\_unused:w .....  
     ..... 15776, 15797, 15802  
 \\_\_keys\_store\_unused\_aux: .... 15692  
 \\_\_keys\_tmp:n ..... 15841, 15853  
 \l\_\_keys\_tmp\_bool .....  
     ..... 14934, 15634, 15641, 15646  
 \l\_\_keys\_tmpa\_tl ..... 14934, 15182  
 \l\_\_keys\_tmpb\_tl . 14934, 15182, 15187  
 \\_\_keys\_trim\_spaces:n .....  
     .... 14946, 14976, 15005, 15106,  
         15446, 15578, 15773, 15814, 15815,  
         15840, 15876, 15884, 15895, 15904  
 \\_\_keys\_trim\_spaces\_auxi:w ... 15840  
 \\_\_keys\_trim\_spaces\_auxii:w .. 15840  
 \\_\_keys\_trim\_spaces\_auxiii:w . 15840  
 \c\_\_keys\_type\_root\_str .....  
     ..... 14908, 15070, 15073, 15085  
 \\_\_keys\_undefine: 15155, 15192, 15416  
 \l\_\_keys\_unused\_clist .....  
     . 711, 14932, 15453, 15459, 15464,  
         15466, 15467, 15494, 15501, 15506,  
         15508, 15509, 15750, 15760, 15788  
 \\_\_keys\_validate\_forbidden: .. 15202  
 \\_\_keys\_validate\_required: ... 15202  
 \c\_\_keys\_validate\_root\_str 14908,  
     15209, 15215, 15219, 15696, 15715  
 \\_\_keys\_value\_or\_default:n .....  
     ..... 15585, 15658  
 \\_\_keys\_value\_requirement:nn ...  
     .. 15137, 15202, 15262, 15418, 15420  
 \\_\_keys\_variable\_set:NnnN .....  
     ..... 15248, 15294, 15296,  
         15298, 15300, 15400, 15402, 15404,  
         15406, 15408, 15410, 15412, 15414  
 \\_\_keys\_variable\_set\_required:NnnN .....  
     ..... 15248,  
         15326, 15328, 15330, 15332, 15334,  
         15336, 15338, 15340, 15354, 15356,  
         15358, 15360, 15376, 15378, 15380,  
         15382, 15392, 15394, 15396, 15398  
 keyval commands:  
     \keyval\_parse:NNn .....  
         .. 197, 693, 694, 14725, 14947, 15447  
 keyval internal commands:  
     \\_\_keyval\_blank\_key\_error:w .....  
         ..... 14845, 14851, 14858  
     \\_\_keyval\_blank\_true:w . 14806, 14858  
     \\_\_keyval\_clean\_up\_active:w ....  
         ..... 691, 14744, 14781, 14828  
     \\_\_keyval\_clean\_up\_other:w ....  
         ..... 692, 14786, 14803  
     \\_\_keyval\_end\_loop\_active:w ....  
         ..... 14730, 14824  
     \\_\_keyval\_end\_loop\_other:w ....  
         ..... 14741, 14824  
     \\_\_keyval\_if\_blank:w .....  
         ..... 14806, 14845, 14851, 14855  
     \\_\_keyval\_if\_empty:w ..... 14855  
     \\_\_keyval\_if\_recursion\_tail:w ...  
         ..... 14729, 14740, 14855  
     \\_\_keyval\_key:nNN .. 692, 14808, 14843  
     \\_\_keyval\_loop\_active>NNw .....  
         ..... 14726, 14727, 14836  
     \\_\_keyval\_loop\_other>NNw .....  
         ..... 14731, 14738, 14831, 14835  
     \\_\_keyval\_misplaced\_equal\_after\_-  
         active\_error:w .....  
         ..... 690, 14752, 14757, 14810  
     \\_\_keyval\_misplaced\_equal\_in\_-  
         split\_error:w ... 14763, 14769,  
             14773, 14778, 14794, 14800, 14810  
     \\_\_keyval\_pair:nnNN .....  
         .... 691, 692, 14780, 14802, 14843  
     \\_\_keyval\_split\_active:w .....  
         690, 690, 14734, 14742, 14762, 14826  
     \\_\_keyval\_split\_active\_auxi:w ...  
         ..... 14743, 14749, 14782, 14827  
     \\_\_keyval\_split\_active\_auxii:w ..  
         ..... 690, 690, 14749  
     \\_\_keyval\_split\_active\_auxiii:w .  
         ..... 690, 14749  
     \\_\_keyval\_split\_active\_auxiv:w ..  
         ..... 691, 14749  
     \\_\_keyval\_split\_active\_auxv:w . 14749  
     \\_\_keyval\_split\_other:w .....  
         .. 14734, 14751, 14772, 14784, 14793  
     \\_\_keyval\_split\_other\_auxi:w ...  
         ..... 691, 14785, 14789, 14804  
     \\_\_keyval\_split\_other\_auxii:w . 14789  
     \\_\_keyval\_split\_other\_auxiii:w ..  
         ..... 692, 14789  
     \\_\_keyval\_tmp:n ..... 14869, 14905  
     \\_\_keyval\_tmp:NN ... 693, 14723, 14841  
     \\_\_keyval\_trim:nN ..... 14759,  
         14780, 14790, 14802, 14808, 14868  
     \\_\_keyval\_trim\_auxi:w ..... 14868  
     \\_\_keyval\_trim\_auxii:w ..... 14868  
     \\_\_keyval\_trim\_auxiii:w ..... 14868  
     \\_\_keyval\_trim\_auxiv:w ..... 14868  
 \kuten ..... 1142, 1176

## L

\L ..... 29552, 31384, 31726  
 \l ..... 29552, 31384, 31738  
 \3kernel ..... 257, 29005

|                                    |                                     |                                          |
|------------------------------------|-------------------------------------|------------------------------------------|
| 13kernel.charcat .....             | 257, 29058                          | \LineBreak .....                         |
| 13kernel.elapsedtime .....         | 257, 29061                          | . 58, 59, 60, 61, 62, 63, 64, 65, 89, 91 |
| 13kernel.fileref .....             | 257, 29075                          | \linedir .....                           |
| 13kernel.filemd5sum .....          | 257, 29102                          | \linedirection .....                     |
| 13kernel.filemoddate .....         | 257, 29117                          | \linepenalty .....                       |
| 13kernel.filesize .....            | 257, 29176                          | \lineskip .....                          |
| 13kernel.resettimer .....          | 258, 29061                          | \lineskiplimit .....                     |
| 13kernel.shellescape .....         | 258, 29201                          | \linewidth .....                         |
| 13kernel.strcmp .....              | 258, 29192                          | \ln .....                                |
| \label .....                       | 29560, 29567, 31674                 | 20712, 20715                             |
| \language .....                    | 331                                 | \ln .....                                |
| \LARGE .....                       | 31656                               | \localbrokenpenalty .....                |
| \Large .....                       | 31657                               | \localinterlinepenalty .....             |
| \large .....                       | 31660                               | \localleftbox .....                      |
| \lastallocatedtoks .....           | 23016                               | \localrightbox .....                     |
| \lastbox .....                     | 332                                 | \locount .....                           |
| \lastkern .....                    | 333                                 | 12729, 12943                             |
| \lastlinefit .....                 | 549                                 | \loctoks .....                           |
| \lastnamedcs .....                 | 835                                 | 22988, 22989, 23015                      |
| \lastnodechar .....                | 1143                                | \logb .....                              |
| \lastnodesubtype .....             | 1144                                | 214                                      |
| \lastnodedtype .....               | 550                                 | \long .....                              |
| \lastpenalty .....                 | 334                                 | 186, 347, 11066, 11070                   |
| \lastsavedboxresourceindex .....   | 920                                 | \LongText .....                          |
| \lastsavedimageresourceindex ..... | 922                                 | 54, 101                                  |
| \lastsavedimageresourcepages ..... | 924                                 | \looseness .....                         |
| \lastskip .....                    | 335                                 | \lower .....                             |
| \lastxpos .....                    | 926                                 | \lowercase .....                         |
| \lastypos .....                    | 927                                 | 348                                      |
| \latelua .....                     | 836                                 | \lower .....                             |
| \lateluafunction .....             | 837                                 | 349                                      |
| LaTeX3 error commands:             |                                     | \lower .....                             |
| \LaTeX3_error: .....               | 628                                 | 350                                      |
| \lccode .....                      | 336                                 | \lpcode .....                            |
| \leaders .....                     | 337                                 | 693                                      |
| \left .....                        | 338                                 | \ltx.utils .....                         |
| left commands:                     |                                     | 257, 29005                               |
| \c_left_brace_str .....            | 71, 993, 5316, 13452, 24152, 24537, | \ltx.utils .....                         |
| 24541, 24561, 24574, 24598, 25081, | 257, 29075                          |                                          |
| 25162, 26194, 26229, 26253, 29654  | \ltx.utils .....                    |                                          |
| \leftghost .....                   | 838                                 | 257, 29102                               |
| \lefthyphenmin .....               | 339                                 | \ltx.utils .....                         |
| \leftmarginkern .....              | 691                                 | 257, 29117                               |
| \leftskip .....                    | 340                                 | \ltx.utils .....                         |
| legacy commands:                   |                                     | 257, 29176                               |
| \legacy_if:nTF .....               | 264, 32036                          | lua commands:                            |
| \legacy_if_p:n .....               | 264, 32036                          | \lua_escape:n .....                      |
| \leqno .....                       | 341                                 | \lua_now:n .....                         |
| \let .....                         | 2, 22, 183, 184, 342                | \lua_now_x:n .....                       |
| \letcharcode .....                 | 839                                 | \lua_shipout:n .....                     |
| \letterspacefont .....             | 692                                 | \lua_shipout_e:n .....                   |
| \limits .....                      | 343                                 | \lua_shipout_x:n .....                   |
| luatex internal commands:          |                                     | 256, 9411, 9420, 28965, 28966, 32804     |
| \__lua_escape:n .....              | 28961, 28971                        | \lua internal commands:                  |
| \__lua_now:n .....                 | 28961, 28966                        | \__lua_now:n .....                       |
| \__lua_shipout:n .....             | 28961, 28968                        | \__lua_shipout:n .....                   |
| \luabytecode .....                 | 844                                 | \luabYTECODE .....                       |
| \luabytecodecall .....             | 845                                 | \luacopyinputnodes .....                 |
| \luacopyinputnodes .....           | 846                                 | \luadef .....                            |
| \luadef .....                      | 847                                 | \luadef .....                            |
| \luadef .....                      | 29212                               | \luadef .....                            |
| \luaescapestring .....             | 850                                 | \luadef .....                            |
| \luafunction .....                 | 851                                 | \luafunction .....                       |
| \luafunctioncall .....             | 852                                 | \luafunctioncall .....                   |
| luatex commands:                   |                                     | \luafunctioncall .....                   |
| \luatex_if_engine:TF .....         | 32809, 32811, 32813                 | \luatex_alignmark .....                  |
| \luatex_if_engine_p: .....         | 32807                               | \luatex_alignmark .....                  |
| \luatex_alignmark .....            | 1243                                |                                          |

|                                                                                                               | M                                              |
|---------------------------------------------------------------------------------------------------------------|------------------------------------------------|
| \luatexaligntab . . . . .                                                                                     | 1244                                           |
| \luatexattribute . . . . .                                                                                    | 1245                                           |
| \luatexattributedef . . . . .                                                                                 | 1246                                           |
| \luatexbanner . . . . .                                                                                       | 853                                            |
| \luatexbodydir . . . . .                                                                                      | 1282                                           |
| \luatexboxdir . . . . .                                                                                       | 1283                                           |
| \luatexcatcodetable . . . . .                                                                                 | 1247                                           |
| \luatexclearmarks . . . . .                                                                                   | 1248                                           |
| \luatexcrampeddisplaystyle . . . . .                                                                          | 1249                                           |
| \luatexcrampedscriptstyle . . . . .                                                                           | 1251                                           |
| \luatexcrampedscriptstyle . . . . .                                                                           | 1252                                           |
| \luatexcrampedtextstyle . . . . .                                                                             | 1253                                           |
| \luatexfontid . . . . .                                                                                       | 1254                                           |
| \luatexformatname . . . . .                                                                                   | 1255                                           |
| \luatexleaders . . . . .                                                                                      | 1256                                           |
| \luatexinitcatcodetable . . . . .                                                                             | 1257                                           |
| \luatexlatelua . . . . .                                                                                      | 1258                                           |
| \luatexleftghost . . . . .                                                                                    | 1284                                           |
| \luatexlocalbrokenpenalty . . . . .                                                                           | 1285                                           |
| \luatexlocalinterlinepenalty . . . . .                                                                        | 1287                                           |
| \luatexlocalleftbox . . . . .                                                                                 | 1288                                           |
| \luatexlocalrightbox . . . . .                                                                                | 1289                                           |
| \luatexluaescapestring . . . . .                                                                              | 1259                                           |
| \luatexluafunction . . . . .                                                                                  | 1260                                           |
| \luatexmathdir . . . . .                                                                                      | 1290                                           |
| \luatexmathstyle . . . . .                                                                                    | 1261                                           |
| \luatexnokerns . . . . .                                                                                      | 1262                                           |
| \luatexnoligs . . . . .                                                                                       | 1263                                           |
| \luatexoutputbox . . . . .                                                                                    | 1264                                           |
| \luatexpagebottomoffset . . . . .                                                                             | 1291                                           |
| \luatexpagedir . . . . .                                                                                      | 1292                                           |
| \luatexpageheight . . . . .                                                                                   | 1293                                           |
| \luatexpageleftoffset . . . . .                                                                               | 1265                                           |
| \luatexpagerightoffset . . . . .                                                                              | 1294                                           |
| \luatexpagetopoffset . . . . .                                                                                | 1266                                           |
| \luatexpagewidth . . . . .                                                                                    | 1295                                           |
| \luatexpardir . . . . .                                                                                       | 1296                                           |
| \luatexposttexhyphenchar . . . . .                                                                            | 1267                                           |
| \luatexposthyphenchar . . . . .                                                                               | 1268                                           |
| \luatexpreexhyphenchar . . . . .                                                                              | 1269                                           |
| \luatexprehyphenchar . . . . .                                                                                | 1270                                           |
| \luatexrevision . . . . .                                                                                     | 854                                            |
| \luatexrightghost . . . . .                                                                                   | 1297                                           |
| \luatexsavecatcodetable . . . . .                                                                             | 1271                                           |
| \luatexscantextokens . . . . .                                                                                | 1272                                           |
| \luatexsuppressfontnotfounderror . . . . .                                                                    | 1242, 1281                                     |
| \luatexsuppressifcsnameerror . . . . .                                                                        | 1274                                           |
| \luatexsuppresslongerror . . . . .                                                                            | 1275                                           |
| \luatexsuppressmathparerror . . . . .                                                                         | 1277                                           |
| \luatexsuppressoutererror . . . . .                                                                           | 1278                                           |
| \luatextextdir . . . . .                                                                                      | 1298                                           |
| \luatexUchar . . . . .                                                                                        | 1279                                           |
| \luatexversion . . . . .                                                                                      | 27, 85, 855                                    |
| \mag . . . . .                                                                                                | 351                                            |
| \mark . . . . .                                                                                               | 352                                            |
| \marks . . . . .                                                                                              | 551                                            |
| math commands:                                                                                                |                                                |
| \c_math_subscript_token . . . . .                                                                             |                                                |
| . . . . . 134, 581, 10825,<br>10880, 10938, 23606, 29470, 29505                                               |                                                |
| \c_math_superscript_token . . . . .                                                                           | 134,<br>581, 10822, 10880, 10933, 23604, 29502 |
| \c_math_toggle_token . . . . .                                                                                |                                                |
| . . . . . 134, 580, 10816,<br>10880, 10914, 23600, 29467, 29496                                               |                                                |
| \mathaccent . . . . .                                                                                         | 353                                            |
| \mathbin . . . . .                                                                                            | 354                                            |
| \mathchar . . . . .                                                                                           | 355, 11065                                     |
| \mathchardef . . . . .                                                                                        | 356                                            |
| \mathchoice . . . . .                                                                                         | 357                                            |
| \mathclose . . . . .                                                                                          | 358                                            |
| \mathcode . . . . .                                                                                           | 359                                            |
| \mathdelimitersmode . . . . .                                                                                 | 856                                            |
| \mathdir . . . . .                                                                                            | 857                                            |
| \mathdirection . . . . .                                                                                      | 858                                            |
| \mathdisplayskipmode . . . . .                                                                                | 859                                            |
| \matheqno gapstep . . . . .                                                                                   | 860                                            |
| \mathinner . . . . .                                                                                          | 360                                            |
| \mathnolimits mode . . . . .                                                                                  | 861                                            |
| \mathop . . . . .                                                                                             | 361                                            |
| \mathopen . . . . .                                                                                           | 362                                            |
| \mathoption . . . . .                                                                                         | 862                                            |
| \mathord . . . . .                                                                                            | 363                                            |
| \mathpenalties mode . . . . .                                                                                 | 863                                            |
| \mathpunct . . . . .                                                                                          | 364                                            |
| \mathrel . . . . .                                                                                            | 365                                            |
| \mathrules fam . . . . .                                                                                      | 864                                            |
| \mathscriptbox mode . . . . .                                                                                 | 866                                            |
| \mathscriptchar mode . . . . .                                                                                | 867                                            |
| \mathscripts mode . . . . .                                                                                   | 865                                            |
| \mathstyle . . . . .                                                                                          | 868                                            |
| \mathsurround . . . . .                                                                                       | 366                                            |
| \mathsurround mode . . . . .                                                                                  | 869                                            |
| \mathsurround skip . . . . .                                                                                  | 870                                            |
| max . . . . .                                                                                                 | 214                                            |
| max commands:                                                                                                 |                                                |
| \c_max_char_int 100, 8999, 10559, 24128                                                                       |                                                |
| \c_max_register_int . . . . . 100,<br>236, 948, 1470, 7746, 8195, 12401,<br>12506, 12508, 22961, 22986, 23023 |                                                |
| \maxdeadcycles . . . . .                                                                                      | 367                                            |
| \maxdepth . . . . .                                                                                           | 368                                            |
| md5.HEX . . . . .                                                                                             | 29094                                          |
| \mdfivesum . . . . .                                                                                          | 782                                            |
| \mdseries . . . . .                                                                                           | 31649                                          |
| \meaning . . . . .                                                                                            | 369                                            |

\medmuskip ..... 370  
 \message ..... 371  
 \MessageBreak ..... 89  
 meta commands:  
   .meta:n ..... 189, [15361](#)  
   .meta:nn ..... 189, [15363](#)  
 \middle ..... 552  
 min ..... 214  
 minus commands:  
   \c\_minus\_inf\_fp .....  
     .... 208, 217, [16225](#), [19237](#),  
       19321, 19654, 19019, 21038, 22563  
   \c\_minus\_zero\_fp .....  
     .... 207, [16225](#), [19233](#), [21757](#), [22561](#)  
 \mkern ..... 372  
 mm ..... 218  
 mode commands:  
   \mode\_if\_horizontal:TF .... 112, [9362](#)  
   \mode\_if\_horizontal\_p: .... 112, [9362](#)  
   \mode\_if\_inner:TF ..... 112, [9364](#)  
   \mode\_if\_inner\_p: ..... 112, [9364](#)  
   \mode\_if\_math:TF ..... 112, [9366](#)  
   \mode\_if\_math\_p: ..... 112, [9366](#)  
   \mode\_if\_vertical:TF .... 113, [9360](#)  
   \mode\_if\_vertical\_p: ..... 113, [9360](#)  
   \mode\_leave\_vertical: 24, [2234](#), [28667](#)  
 \month ..... 373, 1321, 9687  
 \moveleft ..... 374  
 \moveright ..... 375  
 msg commands:  
   \msg\_critical:nn ..... 154, [169](#), [12027](#)  
   \msg\_critical:nnn ..... 154, [12027](#)  
   \msg\_critical:nnnn ..... 154, [12027](#)  
   \msg\_critical:nnnnn ..... 154, [12027](#)  
   \msg\_critical:nnnnnn ..... 154, [12027](#)  
   \msg\_critical:nnnnnnn ..... 154, [12027](#)  
   \msg\_critical\_text:n ..... 152, [11927](#), [12030](#)  
   \msg\_error:nn ..... 154, [12035](#)  
   \msg\_error:nnn ..... 154, [12035](#)  
   \msg\_error:nnnn ..... 154, [12035](#)  
   \msg\_error:nnnnn ..... 154, [12035](#)  
   \msg\_error:nnnnnn ..... 154, 155, [12035](#)  
   \msg\_error\_text:n ..... 152, [11927](#), [12038](#)  
   \msg\_expandable\_error:nn ..... 156, [12683](#)  
   \msg\_expandable\_error:nnn ..... 156, [12683](#)  
   \msg\_expandable\_error:nnnn ..... 156, [12683](#)  
   \msg\_expandable\_error:nnnnn ..... 156, [12683](#)  
   \msg\_expandable\_error:nnnnnn ..... 156, [12683](#)  
   \msg\_fatal:nn ..... 154, [12014](#)  
   \msg\_fatal:nnn ..... 154, [12014](#)  
   \msg\_fatal:nnnn ..... 154, [12014](#)  
   \msg\_fatal:nnnnn ..... 154, [12014](#)  
   \msg\_fatal:nnnnnn ..... 154, [12014](#)  
 \msg\_fatal:n ..... 152, [11927](#), [12017](#)  
 \msg\_gset:nnn ..... 151, [11773](#)  
 \msg\_gset:nnnn ..... 151, [11773](#)  
 \msg\_if\_exist:nnTF .....  
   .... 152, [11760](#), [11767](#), [12147](#)  
 \msg\_if\_exist\_p:nn ..... 152, [11760](#)  
 \msg\_info:nn ..... 155, [12064](#)  
 \msg\_info:nnn ..... 155, [12064](#)  
 \msg\_info:nnnn ..... 155, [12064](#)  
 \msg\_info:nnnnn ..... 155, [12064](#)  
 \msg\_info:nnnnnn ..... 155, [12064](#), [12321](#)  
 \msg\_info\_text:n ..... 153, [11927](#), [12066](#)  
 \msg\_interrupt:nnn ..... 32815  
 \msg\_line\_context: ..... 152,  
   609, [1885](#), [11830](#), [14865](#), [14867](#), [22905](#)  
 \msg\_line\_number: ..... 152, [11830](#)  
 \msg\_log:n ..... 32817  
 \msg\_log:nn ..... 155, [12086](#)  
 \msg\_log:nnn ..... 155, [12086](#)  
 \msg\_log:nnnn ..... 155, [12086](#)  
 \msg\_log:nnnnn ..... 155, [12086](#)  
 \msg\_log:nnnnnn ..... 155, 155, 8176,  
   10383, 10396, 11737, [12086](#), [12798](#),  
   12995, 14014, 15891, 16114, 28898  
 \msg\_log\_eval:Nn ..... 268, 8996, 9133,  
   14546, 14640, 14708, 18465, [32165](#)  
 \g\_msg\_module\_documentation\_prop ..... 153  
 \msg\_module\_name:n .....  
   153, [11840](#), [11946](#), [11964](#), [12045](#), [12067](#)  
 \g\_msg\_module\_name\_prop .....  
   153, 153, [11954](#), [11966](#), [11967](#)  
 \msg\_module\_type:n .....  
   152, 153, 153, [11945](#), [11958](#)  
 \g\_msg\_module\_type\_prop .....  
   153, 153, [11954](#), [11960](#), [11961](#)  
 \msg\_new:nnn ..... 151, [11773](#), [12271](#)  
 \msg\_new:nnnn ..... 151, 607, [11773](#), [12269](#)  
 \msg\_none:nn ..... 155, [12098](#)  
 \msg\_none:nnn ..... 155, [12098](#)  
 \msg\_none:nnnn ..... 155, [12098](#)  
 \msg\_none:nnnnn ..... 155, [12098](#)  
 \msg\_none:nnnnnn ..... 155, [12098](#)  
 \msg\_redirect\_class:nn ..... 157, [12220](#)  
 \msg\_redirect\_module:nnn ..... 157, [12220](#)  
 \msg\_redirect\_name:nnn ..... 157, [12211](#)  
 \msg\_see\_documentation\_text:n .....  
   .... 153, [11964](#)  
 \msg\_set:nnn ..... 151, [11773](#), [12275](#)  
 \msg\_set:nnnn ..... 151, [11773](#), [12273](#)  
 \msg\_show:nn ..... 268, [12099](#)  
 \msg\_show:nnn ..... 268, [12099](#)  
 \msg\_show:nnnn ..... 268, [12099](#)  
 \msg\_show:nnnnn ..... 268, [12099](#)

\msg\_show:nnnnnn . . . . .  
     . . . . . 268, 268, 501, 567, 606,  
     8174, 10381, 10395, 11735, 12099,  
     12797, 12994, 14013, 15889, 16112,  
     23701, 23709, 26395, 26404, 28895  
\msg\_show\_eval:Nn 268, 8992, 9131,  
     14542, 14636, 14704, 18463, 32165  
\msg\_show\_item:n . . . . .  
     . 268, 268, 8184, 10391, 10400, 32170  
\msg\_show\_item:nn . . . . .  
     . . . . . 268, 606, 11745, 32170  
\msg\_show\_item\_unbraced:n 268, 32170  
\msg\_show\_item\_unbraced:nn . 268,  
     633, 12805, 13002, 15899, 28914, 32170  
\msg\_term:n . . . . . 32819  
\msg\_term:nn . . . . . 155, 12092  
\msg\_term:nnn . . . . . 155, 12092  
\msg\_term:nnnn . . . . . 155, 12092  
\msg\_term:nnnnn . . . . . 155, 12092  
\msg\_term:nnnnnn . . . . . 155, 12092  
\msg\_warning:nn . . . . . 154, 12042  
\msg\_warning:nnn . . . . . 154, 12042  
\msg\_warning:nnnn . . . . . 154, 12042  
\msg\_warning:nnnnn . . . . . 154, 12042  
\msg\_warning:nnnnnn . . . . . 154, 12042, 12320  
\msg\_warning\_text:n 152, 11927, 12044  
msg internal commands:  
\msg\_chk\_free:nn . . . . . 11765, 11775  
\msg\_chk\_if\_free:nn . . . . . 11765  
\msg\_class\_chk\_exist:nTF . . . . .  
     . . . . . 12134, 12149, 12216, 12226, 12231  
\msg\_class\_loop\_seq . . . . .  
     . . . . . 620, 12143, 12235,  
     12243, 12253, 12254, 12257, 12259  
\msg\_class\_new:nn . . . . . 616, 621,  
     11975, 12014, 12027, 12035, 12042,  
     12064, 12086, 12092, 12098, 12099  
\msg\_class\_tl . . . . . 617, 620, 12139,  
     12156, 12169, 12190, 12194, 12197,  
     12205, 12244, 12246, 12248, 12262  
\msg\_coding\_error\_text\_tl . . . . .  
     . . . . . 11798,  
     12326, 12334, 12360, 12378, 12387,  
     12394, 12408, 12417, 12439, 12448,  
     12455, 12464, 12470, 12477, 12487,  
     12499, 12514, 12521, 12529, 12537  
\msg\_continue\_text\_tl 11798, 11847  
\msg\_critical\_code:nnnnnn . . . . . 12315  
\msg\_critical\_text\_tl 11798, 12032  
\msg\_current\_class\_tl . . . . .  
     . . . . . 619, 12139, 12151, 12189,  
     12194, 12197, 12205, 12234, 12248  
\msg\_error\_code:nnnnnn . . . . . 12319  
\msg\_expandable\_error:n . . . . .  
     . . . . . 628, 12629, 12649  
\msg\_expandable\_error:w 628, 12629  
\msg\_expandable\_error\_module:nn . . . . .  
     . . . . . 12683  
\msg\_fatal\_code:nnnnnn . . . . . 12314  
\msg\_fatal\_exit: . . . . . 12014  
\c\_msg\_fatal\_text\_tl . . . . . 11798, 12019  
\c\_msg\_help\_text\_tl . . . . . 11798, 11857  
\l\_msg\_hierarchy\_seq . . . . .  
     . . . . . 618, 618, 12142, 12172, 12182, 12187  
\l\_msg\_internal\_tl . . . . . 11752,  
     11883, 11889, 12025, 12123, 12129  
\msg\_interrupt:n . . . . . 11884, 11893  
\msg\_interrupt:Nnnn . . . . . 11837  
\msg\_interrupt:NnnnN . . . . .  
     . . . . . 11837, 12016, 12029, 12037  
\msg\_interrupt\_more\_text:n . . . . .  
     . . . . . 610, 11866  
\msg\_interrupt\_text:n . . . . . 11866  
\msg\_interrupt\_wrap:nnn . . . . .  
     . . . . . 11845, 11855, 11866  
\msg\_kernel\_class\_new:nN . . . . .  
     . . . . . 622, 12276,  
     12314, 12315, 12319, 12320, 12321  
\msg\_kernel\_class\_new\_aux:nN 12276  
\c\_msg\_more\_text\_prefix\_tl . . . . .  
     . . . . . 11758, 11784, 11793, 11842, 11859  
\l\_msg\_name\_str . . . . .  
     . . . . . 11753, 11840, 11873, 11877, 12045,  
     12053, 12057, 12067, 12075, 12079  
\c\_msg\_no\_info\_text\_tl 11798, 11849  
\msg\_no\_more\_text:nnnn . . . . . 11837  
\c\_msg\_on\_line\_text\_tl 11798, 11833  
\msg\_redirect:nnn . . . . . 12220  
\msg\_redirect\_loop\_chk:nnn . . . . .  
     . . . . . 12220, 12262  
\msg\_redirect\_loop\_list:n . . . . . 12220  
\l\_msg\_redirect\_prop . . . . .  
     . . . . . 12141, 12169, 12214, 12217  
\c\_msg\_return\_text\_tl . . . . .  
     . . . . . 11798, 12329, 12337, 12344  
\msg\_show:n . . . . . 616, 12099  
\msg\_show:nn . . . . . 12099  
\msg\_show:w . . . . . 12099  
\msg\_show\_dot:w . . . . . 12099  
\msg\_show\_eval:nnN . . . . . 32165  
\msg\_text:n . . . . . 11927  
\msg\_text:nn . . . . . 11927  
\c\_msg\_text\_prefix\_tl . . . . .  
     . . . . . 628, 11758, 11762, 11782,  
     11791, 11846, 11856, 12050, 12072,  
     12089, 12095, 12102, 12652, 12688

|                                                 |                                                                      |                                   |                                       |
|-------------------------------------------------|----------------------------------------------------------------------|-----------------------------------|---------------------------------------|
| \l__msg_text_str . . . . .                      | 11753, 11839, 11871, 11876, 12044, 12049, 12056, 12066, 12071, 12078 | \g_tmpb_muskip . . . . .          | 184, 14711                            |
| \__msg_tmp:w . . . . .                          | 12630, 12643                                                         | \l_tmpb_muskip . . . . .          | 184, 14711                            |
| \c__msg_trouble_text_t1 . . . . .               | 11798                                                                | \c_zero_muskip                    | 184, 14660, 14662, 14709              |
| \__msg_use:nnnnnnn . . . . .                    | 11985, 12144                                                         | \muskipdef . . . . .              | 379                                   |
| \__msg_use_code: . . . . .                      | 617, 12144                                                           | \mutoglu . . . . .                | 554                                   |
| \__msg_use_hierarchy:nwwN . . . . .             | 12144                                                                |                                   |                                       |
| \__msg_use_none_delimit_by_s_- stop:w . . . . . | 11757, 12175, 12711                                                  |                                   |                                       |
| \__msg_use_redirect_module:n . . . . .          | 618, 12144                                                           |                                   |                                       |
| \__msg_use_redirect_name:n . . . . .            | 12144                                                                |                                   |                                       |
| \mskip . . . . .                                | 376                                                                  |                                   |                                       |
| \muexpr . . . . .                               | 553                                                                  |                                   |                                       |
| multichoice commands:                           |                                                                      |                                   |                                       |
| .multichoice: . . . . .                         | 189, 15365                                                           |                                   |                                       |
| multicoices commands:                           |                                                                      |                                   |                                       |
| .multicoices:nn . . . . .                       | 189, 15365                                                           |                                   |                                       |
| \multiply . . . . .                             | 377                                                                  |                                   |                                       |
| \muskip . . . . .                               | 378, 11074                                                           |                                   |                                       |
| muskip commands:                                |                                                                      |                                   |                                       |
| \c_max_muskip . . . . .                         | 184, 14709                                                           |                                   |                                       |
| \muskip_add:Nn . . . . .                        | 182, 14685                                                           |                                   |                                       |
| \muskip_const:Nn . . . . .                      | 182, 14653, 14709, 14710                                             |                                   |                                       |
| \muskip_eval:n . . . . .                        | 183, 183, 14656, 14697, 14704, 14708                                 |                                   |                                       |
| \muskip_gadd:Nn . . . . .                       | 182, 14685                                                           |                                   |                                       |
| .muskip_gset:N . . . . .                        | 189, 15375                                                           |                                   |                                       |
| \muskip_gset:Nn . . . . .                       | 183, 14675                                                           |                                   |                                       |
| \muskip_gset_eq:NN . . . . .                    | 183, 14681                                                           |                                   |                                       |
| \muskip_gsub:Nn . . . . .                       | 183, 14685                                                           |                                   |                                       |
| \muskip_gzero:N . . . . .                       | 182, 14659, 14668                                                    |                                   |                                       |
| \muskip_gzero_new:N . . . . .                   | 182, 14665                                                           |                                   |                                       |
| \muskip_if_exist:NTF . . . . .                  | 182, 14666, 14668, 14671                                             |                                   |                                       |
| \muskip_if_exist_p:N . . . . .                  | 182, 14671                                                           |                                   |                                       |
| \muskip_log:N . . . . .                         | 184, 14705                                                           |                                   |                                       |
| \muskip_log:n . . . . .                         | 184, 14705                                                           |                                   |                                       |
| \muskip_new:N . . . . .                         | 182, 182, 14647, 14655, 14666, 14668, 14711, 14712, 14713, 14714     |                                   |                                       |
| .muskip_set:N . . . . .                         | 189, 15375                                                           |                                   |                                       |
| \muskip_set:Nn . . . . .                        | 183, 14675                                                           |                                   |                                       |
| \muskip_set_eq:NN . . . . .                     | 183, 14681                                                           |                                   |                                       |
| \muskip_show:N . . . . .                        | 183, 14701                                                           |                                   |                                       |
| \muskip_show:n . . . . .                        | 184, 688, 14703                                                      |                                   |                                       |
| \muskip_sub:Nn . . . . .                        | 183, 14685                                                           |                                   |                                       |
| \muskip_use:N . . . . .                         | 183, 183, 14698, 14699                                               |                                   |                                       |
| \muskip_zero:N . . . . .                        | 182, 182, 14659, 14666                                               |                                   |                                       |
| \muskip_zero_new:N . . . . .                    | 182, 14665                                                           |                                   |                                       |
| \g_tmpa_muskip . . . . .                        | 184, 14711                                                           |                                   |                                       |
| \l_tmpa_muskip . . . . .                        | 184, 14711                                                           |                                   |                                       |
|                                                 |                                                                      | N                                 |                                       |
|                                                 |                                                                      | \n . . . . .                      | 9600, 9602, 9604, 29205, 29207, 29209 |
|                                                 |                                                                      | nan . . . . .                     | 217                                   |
|                                                 |                                                                      | nc . . . . .                      | 218                                   |
|                                                 |                                                                      | nd . . . . .                      | 218                                   |
|                                                 |                                                                      | \newbox . . . . .                 | 505                                   |
|                                                 |                                                                      | \newcatcodetable . . . . .        | 22587                                 |
|                                                 |                                                                      | \newcount . . . . .               | 505                                   |
|                                                 |                                                                      | \newdimen . . . . .               | 505                                   |
|                                                 |                                                                      | \newlinechar . . . . .            | 88, 380                               |
|                                                 |                                                                      | \next . . . . .                   | 52, 98, 106, 109, 112, 120            |
|                                                 |                                                                      | \NG . . . . .                     | 29553, 31385, 31727                   |
|                                                 |                                                                      | \ng . . . . .                     | 29553, 31385, 31739                   |
|                                                 |                                                                      | \noalign . . . . .                | 381                                   |
|                                                 |                                                                      | \noautospacing . . . . .          | 1145                                  |
|                                                 |                                                                      | \noautoxspacing . . . . .         | 1146                                  |
|                                                 |                                                                      | \noboundary . . . . .             | 382                                   |
|                                                 |                                                                      | \nobreakspace . . . . .           | 31682                                 |
|                                                 |                                                                      | \noexpand . . . . .               | 34, 35, 89, 100, 103, 123, 383        |
|                                                 |                                                                      | \nohrule . . . . .                | 871                                   |
|                                                 |                                                                      | \noindent . . . . .               | 384                                   |
|                                                 |                                                                      | \nokerns . . . . .                | 872                                   |
|                                                 |                                                                      | \noligs . . . . .                 | 873                                   |
|                                                 |                                                                      | \nolimits . . . . .               | 385                                   |
|                                                 |                                                                      | \nonscript . . . . .              | 386                                   |
|                                                 |                                                                      | \nonstopmode . . . . .            | 387                                   |
|                                                 |                                                                      | \normaldeviate . . . . .          | 928                                   |
|                                                 |                                                                      | \normalend . . . . .              | 1344, 1345, 12725, 12755, 12939       |
|                                                 |                                                                      | \normaleveryjob . . . . .         | 1346                                  |
|                                                 |                                                                      | \normalexpanded . . . . .         | 1355                                  |
|                                                 |                                                                      | \normalfont . . . . .             | 31644                                 |
|                                                 |                                                                      | \normalhoffset . . . . .          | 1358                                  |
|                                                 |                                                                      | \normalinput . . . . .            | 1347                                  |
|                                                 |                                                                      | \normalitaliccorrection . . . . . | 1357, 1359                            |
|                                                 |                                                                      | \normallanguage . . . . .         | 1348                                  |
|                                                 |                                                                      | \normalleft . . . . .             | 1365, 1366                            |
|                                                 |                                                                      | \normalmathop . . . . .           | 1349                                  |
|                                                 |                                                                      | \normalmiddle . . . . .           | 1367                                  |
|                                                 |                                                                      | \normalmonth . . . . .            | 1350                                  |
|                                                 |                                                                      | \normalouter . . . . .            | 1351                                  |
|                                                 |                                                                      | \normalover . . . . .             | 1352                                  |
|                                                 |                                                                      | \normalright . . . . .            | 1368                                  |
|                                                 |                                                                      | \normalshowtokens . . . . .       | 1361                                  |
|                                                 |                                                                      | \normalsize . . . . .             | 31661                                 |
|                                                 |                                                                      | \normalunexpanded . . . . .       | 1354                                  |
|                                                 |                                                                      | \normalvcenter . . . . .          | 1353                                  |
|                                                 |                                                                      | \normalvoffset . . . . .          | 1360                                  |
|                                                 |                                                                      | \nospaces . . . . .               | 874                                   |

notexpanded commands:

|                             |     |
|-----------------------------|-----|
| \notexpanded: <token> ..... | 141 |
| \novrule .....              | 875 |
| \nulldelimiterspace .....   | 388 |
| \nullfont .....             | 389 |
| \num .....                  | 201 |
| \number .....               | 390 |
| \numexpr .....              | 555 |

## O

|                     |                            |
|---------------------|----------------------------|
| \O .....            | 29554, 31386, 31728, 32018 |
| \o .....            | 29554, 31386, 31740, 32019 |
| \odelcode .....     | 1180                       |
| \odelimiter .....   | 1181                       |
| \OE .....           | 29555, 31387, 31729        |
| \oe .....           | 29555, 31387, 31741        |
| \omathaccent .....  | 1182                       |
| \omathchar .....    | 1183                       |
| \omathchardef ..... | 1184                       |
| \omathcode .....    | 1185                       |
| \omit .....         | 391                        |

one commands:

|                    |                        |
|--------------------|------------------------|
| \c_minus_one ..... | 32713                  |
| \c_one_degree_fp   | 208, 217, 17812, 18468 |
| \openin .....      | 392                    |
| \openout .....     | 393                    |
| \or .....          | 394                    |

or commands:

|            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \or: ..... | 101, 423, 424, 742, 1417,<br>1979, 1980, 1981, 1982, 1983, 1984,<br>1985, 1986, 1987, 2633, 2634, 2635,<br>2636, 2637, 5106, 5182, 5406, 5407,<br>5408, 5409, 5410, 6451, 6452, 8195,<br>8779, 8780, 8781, 8782, 8783, 8784,<br>8785, 8786, 8787, 8788, 8789, 8790,<br>8791, 8792, 8793, 8794, 8795, 8796,<br>8797, 8798, 8799, 8800, 8801, 8802,<br>8803, 8812, 8813, 8814, 8815, 8816,<br>8817, 8818, 8819, 8820, 8821, 8822,<br>8823, 8824, 8825, 8826, 8827, 8828,<br>8829, 8830, 8831, 8832, 8833, 8834,<br>8835, 8836, 10598, 10602, 10605,<br>10607, 10608, 10610, 10612, 10614,<br>10615, 10617, 10619, 10621, 10623,<br>10651, 13221, 13222, 13223, 13224,<br>13225, 13226, 13227, 16271, 16272,<br>16273, 16522, 16537, 16538, 16907,<br>16908, 16933, 18224, 18225, 18226,<br>18262, 18933, 18934, 18935, 19058,<br>19143, 19229, 19230, 19231, 19232,<br>19233, 19234, 19235, 19236, 19237,<br>19316, 19319, 19655, 19656, 19670,<br>19671, 19685, 19969, 20192, 20217,<br>20223, 20224, 20225, 20226, 20227, |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 20376, 20411, 20413, 20421, 20614,<br>20665, 20668, 20677, 20792, 20815,<br>20816, 20848, 20849, 20853, 20906,<br>20907, 20947, 20952, 20962, 20967,<br>20977, 20982, 20992, 20997, 21007,<br>21012, 21022, 21027, 21554, 21555,<br>21600, 21685, 21688, 21700, 21706,<br>21753, 21755, 21756, 21766, 21772,<br>21849, 21850, 21857, 21903, 21904,<br>21911, 21977, 21978, 22198, 22481,<br>22482, 22483, 22560, 22561, 22562,<br>23452, 23453, 23646, 23647, 23936,<br>23937, 23938, 23939, 24202, 24203,<br>24204, 24205, 24206, 25498, 25552,<br>25911, 25912, 32547, 32548, 32549 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                       |             |
|-----------------------|-------------|
| \radical .....        | 1186        |
| \orieveryjob .....    | 1338, 1339  |
| \oripdfoutput .....   | 1341, 1342  |
| \outer .....          | 6, 395, 505 |
| \output .....         | 396         |
| \outputbox .....      | 876         |
| \outputmode .....     | 929         |
| \outputpenalty .....  | 397         |
| \over .....           | 398         |
| \overfullrule .....   | 399         |
| \overline .....       | 400         |
| \overwithdelims ..... | 401         |

## P

|                         |                                                                                                                                                          |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| \PackageError .....     | 92, 100                                                                                                                                                  |
| \pagebottomoffset ..... | 877                                                                                                                                                      |
| \pagedepth .....        | 402                                                                                                                                                      |
| \pagedir .....          | 878                                                                                                                                                      |
| \pagedirection .....    | 879                                                                                                                                                      |
| \pagediscards .....     | 556                                                                                                                                                      |
| \pagefilllstretch ..... | 403                                                                                                                                                      |
| \pagefillstretch .....  | 404                                                                                                                                                      |
| \pagefilstretch .....   | 405                                                                                                                                                      |
| \pagefistretch .....    | 1147                                                                                                                                                     |
| \pagegoal .....         | 406                                                                                                                                                      |
| \pageheight .....       | 930                                                                                                                                                      |
| \pageleftoffset .....   | 880                                                                                                                                                      |
| \pagerightoffset .....  | 881                                                                                                                                                      |
| \pageshrink .....       | 407                                                                                                                                                      |
| \pagestretch .....      | 408                                                                                                                                                      |
| \pagetopoffset .....    | 882                                                                                                                                                      |
| \pagetotal .....        | 409                                                                                                                                                      |
| \pagewidth .....        | 931                                                                                                                                                      |
| \par .....              | 10, 11,<br>11, 12, 12, 12, 13, 13, 13,<br>14, 14, 160, 331, 410, 1075, 27232,<br>27234, 27238, 27243, 27248, 27253,<br>27260, 27265, 27272, 27277, 27297 |
| \pardir .....           | 883                                                                                                                                                      |

|                                   |     |                                    |              |
|-----------------------------------|-----|------------------------------------|--------------|
| \pardirection . . . . .           | 884 | \pdflastlinedepth . . . . .        | 665          |
| \parfillskip . . . . .            | 411 | \pdflastlink . . . . .             | 610          |
| \parindent . . . . .              | 412 | \pdflastobj . . . . .              | 611          |
| \parshape . . . . .               | 413 | \pdflastxform . . . . .            | 612          |
| \parshapedimen . . . . .          | 557 | \pdflastximage . . . . .           | 613          |
| \parshapeindent . . . . .         | 558 | \pdflastximagecolordepth . . . . . | 614          |
| \parshapelenngth . . . . .        | 559 | \pdflastximagepages . . . . .      | 616          |
| \parskip . . . . .                | 414 | \pdflastxpos . . . . .             | 666          |
| \patterns . . . . .               | 415 | \pdflastypos . . . . .             | 667          |
| \pausing . . . . .                | 416 | \pdflinkmargin . . . . .           | 617          |
| pc . . . . .                      | 218 | \pdfliteral . . . . .              | 618          |
| \pdfadjustspacing . . . . .       | 654 | \pdfmajorversion . . . . .         | 619          |
| \pdfannot . . . . .               | 582 | \pdfmapfile . . . . .              | 668          |
| \pdfcatalog . . . . .             | 583 | \pdfmapline . . . . .              | 669          |
| \pdfcolorstack . . . . .          | 585 | \pdfmdfivesum . . . . .            | 714          |
| \pdfcolorstackinit . . . . .      | 586 | \pdfminorversion . . . . .         | 620          |
| \pdfcompresslevel . . . . .       | 584 | \pdfnames . . . . .                | 621          |
| \pdfcopyfont . . . . .            | 655 | \pdfnoligatures . . . . .          | 670          |
| \pdfcreationdate . . . . .        | 587 | \pdfnormaldeviate . . . . .        | 671          |
| \pdfdecimaldigits . . . . .       | 588 | \pdfobj . . . . .                  | 622          |
| \pdfdest . . . . .                | 589 | \pdfobjcompresslevel . . . . .     | 623          |
| \pdfdestmargin . . . . .          | 590 | \pdfoutline . . . . .              | 624          |
| \pdfdraftmode . . . . .           | 656 | \pdfoutput . . . . .               | 625          |
| \pdfeachlinedepth . . . . .       | 657 | \pdfpageattr . . . . .             | 626          |
| \pdfeachlineheight . . . . .      | 658 | \pdfpagebox . . . . .              | 628          |
| \pdfelapsetime . . . . .          | 659 | \pdfpageheight . . . . .           | 672          |
| \pdfendlink . . . . .             | 591 | \pdfpageref . . . . .              | 629          |
| \pdfendthread . . . . .           | 592 | \pdfpageresources . . . . .        | 630          |
| \pdfextension . . . . .           | 885 | \pdfpagesattr . . . . .            | 627, 631     |
| \pdffeedback . . . . .            | 886 | \pdfpagewidth . . . . .            | 673          |
| \pdffiledump . . . . .            | 716 | \pdfpkmode . . . . .               | 674          |
| \pdffilemoddate . . . . .         | 715 | \pdfpkresolution . . . . .         | 675          |
| \pdffilesize . . . . .            | 713 | \pdfprimitive . . . . .            | 676          |
| \pdffirstlineheight . . . . .     | 660 | \pdfprotrudechars . . . . .        | 677          |
| \pdffontattr . . . . .            | 593 | \pdfpxdimen . . . . .              | 678          |
| \pdffontexpand . . . . .          | 661 | \pdfrandomseed . . . . .           | 679          |
| \pdffontname . . . . .            | 594 | \pdfrefobj . . . . .               | 632          |
| \pdffontobjnum . . . . .          | 595 | \pdfrefxform . . . . .             | 633          |
| \pdffontsize . . . . .            | 662 | \pdfrefximage . . . . .            | 634          |
| \pdffgamma . . . . .              | 596 | \pdfresettimer . . . . .           | 680          |
| \pdffgentounicode . . . . .       | 599 | \pdfrestore . . . . .              | 635          |
| \pdffglyptounicode . . . . .      | 600 | \pdfretval . . . . .               | 636          |
| \pdfhorigin . . . . .             | 601 | \pdfsave . . . . .                 | 637          |
| \pdfignoreddimen . . . . .        | 663 | \pdfsavepos . . . . .              | 681          |
| \pdfimageapplygamma . . . . .     | 597 | \pdfsetmatrix . . . . .            | 638          |
| \pdfimagegamma . . . . .          | 598 | \pdfsetrandomseed . . . . .        | 682          |
| \pdfimagehicolor . . . . .        | 602 | \pdfshellescape . . . . .          | 683          |
| \pdfimageresolution . . . . .     | 603 | \pdfstartlink . . . . .            | 639          |
| \pdfincludechars . . . . .        | 604 | \pdfstartthread . . . . .          | 640          |
| \pdfinclusioncopyfonts . . . . .  | 605 | \pdfstrcmp . . . . .               | 22, 416, 712 |
| \pdfinclusionerrorlevel . . . . . | 606 | \pdfsuppressptexinfo . . . . .     | 641          |
| \pdfinfo . . . . .                | 608 | pdftex commands:                   |              |
| \pdfinsertht . . . . .            | 664 | \pdf_if_engine:TF . . . . .        |              |
| \pdflastannot . . . . .           | 609 | 32823, 32825, 32827                |              |

\pdftex\_if\_engine\_p: ..... 32821  
 \pdftexbanner ..... 686  
 \pdftexrevision ..... 687  
 \pdftexversion ..... 80, 688  
 \pdfthread ..... 642  
 \pdfthreadmargin ..... 643  
 \pdftracingfonts ..... 684, 1233, 1234  
 \pdftrailer ..... 644  
 \pdfuniformdeviate ..... 685  
 \pdfuniquepagesname ..... 645  
 \pdfvariable ..... 887  
 \pdfvorigin ..... 646  
 \pdfxform ..... 647  
 \pdfxformname ..... 648  
 \pdfximage ..... 649  
 \pdfximagebbox ..... 650  
 peek commands:  
   \peek\_after:Nw ..... 114, 138,  
     138, 138, 11221, 11234, 11262, 32536  
   \peek\_catcode:NTF ..... 139, 11317  
   \peek\_catcode\_collect\_inline:Nn .  
     ..... 274, 32516  
   \peek\_catcode\_ignore\_spaces:NTF .  
     ..... 139, 11331  
   \peek\_catcode\_remove:NTF .. 139, 11317  
   \peek\_catcode\_remove\_ignore\_-  
     spaces:NTF ..... 139, 11331  
   \peek\_charcode:NTF ..... 139, 11317  
   \peek\_charcode\_collect\_inline:Nn  
     ..... 274, 32516  
   \peek\_charcode\_ignore\_spaces:NTF  
     ..... 139, 11331  
   \peek\_charcode\_remove:NTF .. 140, 11317  
   \peek\_charcode\_remove\_ignore\_-  
     spaces:NTF ..... 140, 11331  
   \peek\_gafter:Nw ..... 138, 138, 11221  
   \peek\_meaning:NTF ..... 140, 11317  
   \peek\_meaning\_collect\_inline:Nn .  
     ..... 274, 32516  
   \peek\_meaning\_ignore\_spaces:NTF .  
     ..... 140, 11331  
   \peek\_meaning\_remove:NTF .. 140, 11317  
   \peek\_meaning\_remove\_ignore\_-  
     spaces:NTF ..... 140, 11331  
   \peek\_N\_type:TF .....  
     ..... 141, 11354, 11391, 11393  
   \peek\_remove\_spaces:n .....  
     ..... 274, 592, 11230, 11340, 11345, 11350  
 peek internal commands:  
   \\_\_peek\_collect:N ..... 1201, 32516  
   \\_\_peek\_collect>NNn ..... 32516  
   \\_\_peek\_collect\_remove:nw .... 32516  
   \l\_peek\_collect\_t1 ..... 1201,  
     32515, 32527, 32529, 32554, 32559  
   \\_\_peek\_collect\_true:w .. 1201, 32516  
   \\_\_peek\_execute\_branches\_...: . 1201  
   \\_\_peek\_execute\_branches\_-  
     catcode: ..... 592, 11284, 32517  
   \\_\_peek\_execute\_branches\_-  
     catcode\_aux: ..... 11284  
   \\_\_peek\_execute\_branches\_-  
     catcode\_auxii:N ..... 11284  
   \\_\_peek\_execute\_branches\_-  
     catcode\_auxiii: ..... 11284  
   \\_\_peek\_execute\_branches\_-  
     charcode: ..... 592, 11284, 32519  
   \\_\_peek\_execute\_branches\_-  
     meaning: ..... 592, 11276, 32521  
   \\_\_peek\_execute\_branches\_N\_type:  
     ..... 11354  
   \\_\_peek\_false:w ..... 593, 1201,  
     11214, 11232, 11243, 11257, 11281,  
     11304, 11314, 11371, 11384, 32528  
   \\_\_peek\_false\_aux:n 1201, 32529, 32530  
   \\_\_peek\_N\_type:w ..... 11354  
   \\_\_peek\_N\_type\_aux:nnw ..... 11354  
   \\_\_peek\_remove\_spaces: ..... 11230  
   \l\_peek\_search\_t1 589, 591, 1201,  
     11213, 11250, 11301, 11311, 32526  
   \l\_peek\_search\_token .....  
     589, 1201, 11212, 11249, 11278, 32525  
   \\_\_peek\_tmp:w .....  
     ..... 11214, 11228, 11355, 11377  
   \\_\_peek\_token\_generic:NNTF .....  
     ..... 592, 593, 11264,  
     11266, 11268, 11388, 11392, 11394  
   \\_\_peek\_token\_generic\_aux:NNNTF .  
     ..... 11246, 11265, 11271  
   \\_\_peek\_token\_remove\_generic:NNTF  
     ..... 592, 11264, 11272, 11274  
   \\_\_peek\_true:w ..... 593,  
     1201, 11214, 11256, 11279, 11302,  
     11312, 11369, 11383, 11384, 32535  
   \\_\_peek\_true\_aux:w ..... 590,  
     590, 11214, 11227, 11234, 11235,  
     11251, 11265, 32536, 32537, 32555  
   \\_\_peek\_true\_remove:w .....  
     590, 590, 11225, 11240, 11271, 32560  
   \\_\_peek\_use\_none\_delimit\_by\_s\_-  
     stop:w ..... 593, 11220, 11367  
   \penalty ..... 417  
   \pi ..... 17218, 17219  
   pi ..... 217  
   \pm ..... 18683, 18684  
   \postbreakpenalty ..... 1148  
   \postdisplaypenalty ..... 418  
   \postexhyphenchar ..... 888  
   \posthyphenchar ..... 889

\prebinoppenalty ..... 890  
 \prebreakpenalty ..... 1149  
 \predisplaydirection ..... 560  
 \predisplaygapfactor ..... 891  
 \predisplaypenalty ..... 419  
 \predisplaysize ..... 420  
 \preehyphenchar ..... 892  
 \prehyphenchar ..... 893  
 \prerelpenalty ..... 894  
 \pretolerance ..... 421  
 \prevdepth ..... 422  
 \prevgraf ..... 423  
 prg commands:  
   \prg\_break: ... 114, 456, 495, 496,  
     604, 605, 1041, 2231, 4566, 5468,  
     5484, 5602, 5652, 5758, 5761, 5902,  
     5949, 6002, 6008, 6239, 6320, 6492,  
     6633, 7930, 7963, 8006, 8051, 8579,  
     9373, 11671, 11692, 11721, 13690,  
     16333, 16342, 18348, 18368, 18369,  
     18579, 18580, 18593, 18693, 18694,  
     18695, 22087, 22139, 22363, 23232,  
     23307, 23641, 23715, 23745, 23746,  
     23747, 23748, 23749, 23750, 24102,  
     24106, 26013, 26040, 32257, 32263  
   \prg\_break:n ... 114, 114, 2231,  
     4568, 5370, 5378, 5390, 7804, 7943,  
     8589, 9373, 11566, 16106, 16349, 25575  
   \prg\_break\_point: ...  
     ... 114, 114, 658, 949, 950, 956,  
     1194, 2231, 4556, 5371, 5379, 5469,  
     5485, 5603, 5653, 5759, 5762, 5903,  
     5950, 6003, 6009, 6240, 6440, 6597,  
     7801, 7931, 7963, 8006, 8051, 8096,  
     8103, 8584, 9373, 11561, 11656,  
     11692, 11721, 13663, 16100, 16334,  
     16343, 18349, 18370, 18581, 18697,  
     22088, 22139, 22371, 23060, 23101,  
     23225, 23232, 23564, 23716, 23752,  
     24080, 25576, 25886, 26034, 32258  
   \prg\_break\_point:Nn ... 40,  
     113, 113, 339, 495, 514, 679, 2222,  
     4123, 4141, 4151, 4166, 4941, 4967,  
     4987, 7964, 7999, 8007, 8024, 8031,  
     8040, 8631, 9373, 10179, 10193,  
     10213, 10231, 11693, 11709, 11722,  
     12893, 12912, 14487, 18766, 23693  
   \prg\_do\_nothing: ...  
     ... 9, 114, 384, 436, 485, 551,  
     566, 597, 665, 748, 919, 997, 1044,  
     2220, 2231, 2555, 2947, 2974, 3073,  
     3074, 3075, 3535, 3536, 3797, 4415,  
     4450, 4778, 4780, 5534, 6490, 7605,  
     7612, 7896, 7898, 9577, 9824, 9830,  
     9838, 9993, 10192, 10200, 10349,  
     10353, 10360, 11467, 11475, 11484,  
     13220, 13532, 13959, 13996, 13998,  
     15737, 16613, 16647, 16673, 16681,  
     18233, 22068, 22742, 23147, 23305,  
     23306, 23535, 23584, 24401, 24444,  
     24445, 24452, 24453, 26105, 26268  
   \prg\_generate\_conditional\_=  
     variant:Nnn ... 107, 3167, 3356,  
     3378, 3953, 3963, 3987, 3997, 4013,  
     4030, 4041, 4115, 4353, 4371, 4845,  
     4858, 4866, 4897, 7737, 7805, 7899,  
     7901, 7915, 7917, 7919, 7921, 9129,  
     10011, 10025, 10026, 10169, 10171,  
     11600, 11601, 11649, 11673, 11684,  
     12752, 27096, 27098, 27102, 27729  
   \prg\_map\_break:Nn ... 113, 113, 339,  
     398, 563, 606, 2222, 4179, 4181,  
     5000, 5002, 7954, 7956, 9373, 10248,  
     10250, 11732, 11734, 12876, 12878  
   \prg\_new\_conditional:Nnn ...  
     ... 105, 1597, 9079  
   \prg\_new\_conditional:Npnn ... 105,  
     105, 107, 310, 403, 580, 592, 1580,  
     2128, 2826, 3340, 3348, 3358, 3368,  
     3517, 3945, 3955, 3970, 3979, 3989,  
     4045, 4061, 4072, 4341, 4355, 4373,  
     4412, 4432, 4447, 4838, 4847, 4852,  
     5367, 5376, 5391, 5399, 6089, 6123,  
     6142, 7729, 8400, 8453, 8491, 8499,  
     9047, 9052, 9079, 9121, 9153, 9213,  
     9228, 9239, 9254, 9264, 9360, 9362,  
     9364, 9366, 9840, 10122, 10902,  
     10907, 10912, 10917, 10924, 10930,  
     10936, 10941, 10946, 10951, 10956,  
     10961, 10966, 10971, 10978, 10993,  
     10998, 11034, 11141, 11150, 11644,  
     11651, 11760, 12810, 13835, 14303,  
     14308, 14603, 14611, 15873, 15880,  
     16517, 17660, 18483, 18491, 18507,  
     24194, 24214, 24236, 24282, 24306,  
     27092, 27094, 27100, 27719, 29526,  
     30443, 30463, 30491, 30516, 32036  
   \prg\_new\_eq\_conditional:NNn ...  
     106, 1713, 3638, 3639, 4829, 4831,  
     4833, 4835, 7634, 7636, 8168, 8169,  
     8170, 8171, 8172, 8173, 8348, 8350,  
     9079, 9149, 9151, 9929, 9931, 10118,  
     10120, 11640, 11642, 14234, 14236,  
     14577, 14579, 14671, 14673, 18481,  
     18482, 22784, 22786, 27040, 27042  
   \prg\_new\_protected\_conditional:Nnn  
     ... 105, 1597, 9079  
   \prg\_new\_protected\_conditional:Npnn

105, 1580, 4001, 4014, 4032, 4860,  
 4868, 5508, 5517, 7786, 7895, 7897,  
 7903, 7906, 7909, 7912, 9079, 9555,  
 10002, 10012, 10014, 10136, 10140,  
 11580, 11590, 11675, 12743, 12830,  
 12850, 13511, 13637, 13784, 13786,  
 13788, 13790, 13825, 13887, 22788,  
 24628, 26409, 26414, 26427, 26429  
`\prg_replicate:nn` .....  
 ..... 49, 81, 112, 536, 725,  
 9312, 11874, 12054, 12076, 13052,  
 16053, 16179, 19911, 20764, 21072,  
 21328, 21374, 21411, 21934, 21942,  
 22401, 22504, 23806, 24407, 25144,  
 25505, 25531, 25678, 25686, 26110,  
 26572, 26577, 26584, 26687, 26692  
`\prg_return_false:` .....  
 ..... 106, 107, 275, 320, 491,  
 509, 560, 560, 1056, 1574, 1640,  
 1648, 1799, 1804, 1817, 1822, 1830,  
 1847, 2131, 2836, 3345, 3353, 3364,  
 3374, 3521, 3950, 3960, 3975, 3984,  
 3994, 4010, 4024, 4038, 4052, 4068,  
 4083, 4350, 4368, 4386, 4394, 4404,  
 4420, 4443, 4454, 4843, 4850, 4856,  
 4864, 4872, 5372, 5380, 5396, 5412,  
 5515, 5524, 6093, 6096, 6099, 6126,  
 6129, 6146, 6149, 6152, 7734, 7800,  
 7819, 8398, 8430, 8435, 8458, 8496,  
 8504, 9050, 9057, 9079, 9126, 9158,  
 9218, 9234, 9244, 9260, 9270, 9361,  
 9363, 9365, 9367, 9559, 9567, 9856,  
 9859, 10005, 10019, 10125, 10160,  
 10166, 10905, 10910, 10915, 10920,  
 10927, 10934, 10939, 10944, 10949,  
 10954, 10959, 10964, 10969, 10974,  
 10991, 10996, 11001, 11006, 11040,  
 11043, 11055, 11154, 11179, 11196,  
 11205, 11588, 11598, 11647, 11667,  
 11682, 11763, 12750, 12819, 12833,  
 12853, 13520, 13642, 13667, 13797,  
 13817, 13831, 13850, 13859, 13870,  
 13884, 13891, 14306, 14325, 14340,  
 14341, 14607, 14614, 15878, 15886,  
 16528, 16530, 17675, 17687, 18488,  
 18502, 18515, 22798, 22804, 24208,  
 24219, 24222, 24227, 24231, 24232,  
 24240, 24243, 24248, 24251, 24288,  
 24291, 24312, 24315, 24635, 24640,  
 26455, 27093, 27095, 27101, 27725,  
 27727, 29536, 29539, 30446, 30450,  
 30453, 30483, 30509, 30530, 32041  
`\prg_return_true:` .....  
 ..... 106, 107, 275, 320, 391, 403  
  
`\prg_set_conditional:Nnn` .....  
 ..... 105, 1597, 9079  
`\prg_set_conditional:Npnn` .....  
 ..... 105, 106, 107, 1580,  
 1796, 1808, 1824, 1836, 9079, 13878  
`\prg_set_eq_conditional:NNn` .....  
 ..... 106, 1713, 9079  
`\prg_set_protected_conditional:Nnn` .....  
 ..... 105, 1597, 9079  
`\prg_set_protected_conditional:Npnn` .....  
 ..... 105, 1580, 9079, 13650  
`\prg internal commands:`  
`\_\_prg_break_point:Nn` ..... 339  
`\_\_prg_generate_conditional:nnNNNnnn`  
 ..... 1592, 1617, 1626  
`\_\_prg_generate_conditional:NNnnnnNw`  
 ..... 1626  
`\_\_prg_generate_conditional_-count:NNNnn` ..... 1597  
`\_\_prg_generate_conditional_-count:nnNNNnn` ..... 1597

```

__prg_generate_conditional_-
 fast:nw 320, 321, 1626
__prg_generate_conditional_-
 parm:NNNpnn 1580
__prg_generate_conditional_-
 test:w 1626
__prg_generate_F_form:wNNnnnnN 1669
__prg_generate_p_form:wNNnnnnN .
 320, 1669
__prg_generate_T_form:wNNnnnnN 1669
__prg_generate_TF_form:wNNnnnnN
 1669
__prg_p_true:w 322, 1669
__prg_replicate:N 9312
__prg_replicate_ 9312
__prg_replicate_0:n 9312
__prg_replicate_1:n 9312
__prg_replicate_2:n 9312
__prg_replicate_3:n 9312
__prg_replicate_4:n 9312
__prg_replicate_5:n 9312
__prg_replicate_6:n 9312
__prg_replicate_7:n 9312
__prg_replicate_8:n 9312
__prg_replicate_9:n 9312
__prg_replicate_first:N 9312
__prg_replicate_first_-:n 9312
__prg_replicate_first_0:n 9312
__prg_replicate_first_1:n 9312
__prg_replicate_first_2:n 9312
__prg_replicate_first_3:n 9312
__prg_replicate_first_4:n 9312
__prg_replicate_first_5:n 9312
__prg_replicate_first_6:n 9312
__prg_replicate_first_7:n 9312
__prg_replicate_first_8:n 9312
__prg_replicate_first_9:n 9312
__prg_set_eq_conditional>NNNn 1713
__prg_set_eq_conditional:nnNnnNNw
 1721, 1729
__prg_set_eq_conditional_F_-
 form:nnn 1729
__prg_set_eq_conditional_F_-
 form:wNNnnnn 1766
__prg_set_eq_conditional_-
 loop:nnnnNw 1729
__prg_set_eq_conditional_p_-
 form:nnn 1729
__prg_set_eq_conditional_p_-
 form:wNNnnnn 1760
__prg_set_eq_conditional_T_-
 form:nnn 1729
__prg_set_eq_conditional_T_-
 form:wNNnnnn 1764
__prg_set_eq_conditional_TF_-
 form:nnn 1729
__prg_set_eq_conditional_TF_-
 form:wNnnnn 1762
__prg_use_none_delimit_by_q_-
 recursion_stop:w 1578, 1655, 1734, 1739, 1746
\primitive 784
prop commands:
 \c_empty_prop 150,
 596, 11403, 11413, 11417, 11420, 11646
 \prop_clear:N 144, 144,
 11416, 11423, 11443, 11446, 11451,
 11454, 11459, 11462, 25842, 28506
 \prop_clear_new:N 144, 11422
 \prop_const_from_keyval:Nn
 145, 11441, 27688, 27695
 \prop_count:N 146, 11570, 32239
 \prop_gclear:N 144, 11416, 11426
 \prop_gclear_new:N 144,
 1090, 11422, 27762, 27763
 \prop_get:Nn 114, 32829, 32831
 \prop_get:NnN 38, 39, 145,
 146, 11527, 28764, 28768, 28837, 28841
 \prop_get:NnNTF 145,
 147, 147, 5575, 11675,
 12169, 12189, 12244, 22673, 27950
 \prop_gpop:Nnn 146, 11535
 \prop_gpop:NnNTF 146, 147, 11580
 .prop_gput:N 189, 15383
 \prop_gput:Nnn 145,
 5349, 5350, 5351, 5352, 5353, 5354,
 5355, 5356, 5357, 5358, 5359, 5360,
 5361, 5362, 5363, 11602, 11955,
 11957, 12733, 12779, 12947, 12977,
 22626, 27976, 27994, 28029, 28060
 \prop_gput_if_new:Nnn 145, 11623
 \prop_gremove:Nn 146, 11511, 12790, 12987, 22624
 \prop_gset_eq:NN 144, 11420, 11428,
 11453, 27764, 27766, 27928, 27930,
 27967, 27969, 28216, 28384, 28425
 \prop_gset_from_keyval:Nn 144, 11441
 \prop_hput:Nnn 11602
 \prop_if_empty:NTF .. 146, 11644, 32236
 \prop_if_empty_p:N 146, 11644
 \prop_if_exist:NTF 146, 11423, 11426, 11640, 15180
 \prop_if_exist_p:N 146, 11640
 \prop_if_in:NnTF 147, 11651, 11960, 11966
 \prop_if_in_p:Nn 147, 11651
 \prop_item:Nn 146,
 148, 11557, 11961, 11967, 32830, 32832

```

```

\prop_log:N 149, 11735
\prop_map_break: 148, 605, 11693, 11709, 11722, 11731
\prop_map_break:n 149, 11731
\prop_map_function:NN .. 148, 148, 268, 603, 605, 11575, 11686, 11745, 12804, 13001, 28912
\prop_map_inline:Nn 148, 11702, 26594, 28226, 28228, 28231, 28251, 28253, 28327, 28344, 28405, 28407, 28411, 28413, 28593, 28612, 28811, 28820
\prop_map_tokens:Nn 148, 148, 497, 11717
\prop_new:N 144, 144, 5348, 11410, 11423, 11426, 11436, 11437, 11438, 11439, 11440, 11954, 11956, 11978, 12141, 12721, 12935, 15180, 22577, 25763, 25764, 28204, 28205, 28206, 28676, 28717
\prop_pop:NnN 145, 11535
\prop_pop:NnNTF 145, 147, 11580
\prop_put:N 189, 15383
\prop_put:Nnn 145, 360, 595, 595, 11493, 11602, 12217, 12233, 12250, 26000, 27973, 27991, 28010, 28027, 28058, 28262, 28264, 28270, 28272, 28281, 28287, 28295, 28354, 28362, 28452, 28458, 28466, 28473, 28617, 28677, 28679, 28681, 28683, 28685, 28687, 28689, 28691, 28693, 28695, 28697, 28699, 28701, 28703, 28705, 28707, 28709, 28711
\prop_put_if_new:Nnn 145, 11623
\prop_rand_key_value:N ... 269, 32234
\prop_remove:Nn 146, 11511, 12214, 12229, 28806, 28809, 28813
\prop_set_eq:NN 144, 11417, 11428, 11445, 26011, 27916, 27918, 27960, 27962, 28213, 28222, 28224, 28377, 28401, 28403, 28422, 28550, 28801
\prop_set_from_keyval:Nn 144, 597, 11441
\prop_show:N 149, 11735
\g_tmpa_prop 149, 11436
\l_tmpa_prop 149, 11436
\g_tmpb_prop 149, 11436
\l_tmpb_prop 149, 11436
prop internal commands:
__prop_count:nn 11570
__prop_from_keyval:n 11441
__prop_from_keyval_key:n 11441
__prop_from_keyval_key:w 597, 11441
__prop_from_keyval_loop:w ... 11441
__prop_from_keyval_split:Nw . 11441
__prop_from_keyval_value:n .. 11441
__prop_from_keyval_value:w 597, 11441
__prop_if_in:N 603, 11651
__prop_if_in:nwwn 603, 11651
__prop_if_recursion_tail_stop:n 11408, 11472
\l__prop_internal_prop ... 11440, 11443, 11445, 11446, 11451, 11453, 11454, 11459, 11461, 11462, 11493
\l__prop_internal_tl 602, 11399, 11402, 11606, 11612, 11613, 11629, 11636
__prop_item_Nn:nwn 600
__prop_item_Nn:nwwn 11557
__prop_map_function:Nwwn 11686
__prop_map_tokens:nwwn 11717
__prop_pair:wn 594, 594, 595, 599, 603, 604, 605, 605, 11399, 11400, 11505, 11508, 11560, 11563, 11608, 11631, 11654, 11658, 11692, 11695, 11705, 11707, 11712, 11721, 11724, 32244
__prop_put:NNnn 11602
__prop_put_if_new:NNnn 11623
__prop_rand_item:w 32234
__prop_show>NN . 11735, 11737, 11739
__prop_split:NnTF 595, 602, 602, 603, 11500, 11513, 11519, 11529, 11537, 11546, 11582, 11592, 11611, 11634, 11677
__prop_split_aux:NnTF 11500
__prop_split_aux:w 599, 11500
__prop_use_i_delimit_by_s_- stop:nw 32233, 32247
\protect 1141, 13119, 17153, 29798, 29821, 29823, 31595
\protected 121, 123, 147, 561, 11068, 11070
\protrudechars 932
\ProvidesExplClass 7
\ProvidesExplFile 7, 32888, 32905
\ProvidesExplFileAux 32891, 32893
\ProvidesExplPackage 7
\ProvidesFile 32896, 32897
pt 218
\ptexminorversion 1150
\ptexrevision 1151
\ptexversion 1152
\pxdimen 933

```

## Q

quark commands:

\q\_mark ..... 39, [961](#), [3288](#)

\q\_nil . . . . . 21, 21, 39, 39, 39,  
 58, 317, 369, 372, 373, 1536, 1539,  
 3288, 3342, 3361, 3367, 3382, 3383,  
 3389, 3413, 3417, 15822, 15823,  
 15825, 15827, 15829, 15831, 15837  
\q\_no\_value 38, 39, 39, 39, 77, 77, 77,  
 77, 77, 77, 84, 84, 84, 117, 126, 145,  
 145, 146, 160, 160, 166, 166, 167,  
 168, 168, 168, 369, 371, 492, 493,  
 555, 600, 600, 3288, 3350, 3371,  
 3377, 7810, 7818, 7830, 7856, 9553,  
 9968, 9983, 11531, 11542, 11551,  
 12827, 12840, 13509, 13634, 13662,  
 13777, 13779, 13781, 13783, 13823  
\quark\_if\_nil:n . . . . . 372  
\quark\_if\_nil:NTF . . . . . 39, 3340  
\quark\_if\_nil:nTF . . . . . 39, 370, 373, 3358  
\quark\_if\_nil\_p:N . . . . . 39, 3340  
\quark\_if\_nil\_p:n . . . . . 39, 3358  
\quark\_if\_no\_value:NTF . . . . . 39, 3340,  
 13664, 28766, 28770, 28839, 28843  
\quark\_if\_no\_value:nTF . . . . . 39, 3358  
\quark\_if\_no\_value\_p:N . . . . . 39, 3340  
\quark\_if\_no\_value\_p:n . . . . . 39, 3358  
\quark\_if\_recursion\_tail\_break:N  
 . . . . . 32833  
\quark\_if\_recursion\_tail\_break:n  
 . . . . . 32835  
\quark\_if\_recursion\_tail\_-  
 break:NN . . . . . 40, 373, 3328  
\quark\_if\_recursion\_tail\_-  
 break:nN . . . . . 40, 373, 3328  
\quark\_if\_recursion\_tail\_stop:N .  
 . . . . . 40,  
 309, 373, 1138, 3296, 29385, 31371,  
 31402, 31691, 31703, 31765, 31816  
\quark\_if\_recursion\_tail\_stop:n .  
 . . . . . 40, 309, 371, 373, 3310, 30890  
\quark\_if\_recursion\_tail\_stop\_-  
 do:Nn . . . . . 40, 309, 373, 3296  
\quark\_if\_recursion\_tail\_stop\_-  
 do:nn . . . . . 40, 309, 373, 3310  
\quark\_new:N . . . . .  
 . . . . . 38, 309, 310, 375, 3283, 3288,  
 3289, 3290, 3291, 3292, 3293, 3295,  
 3759, 3760, 3761, 3762, 3763, 4722,  
 4723, 5347, 8203, 8204, 9116, 9117,  
 9816, 9817, 10411, 11406, 11407,  
 12952, 13400, 13402, 13403, 14940,  
 23856, 23861, 29402, 29404, 29405  
\q\_recursion\_stop . . . . . 21, 21, 40,  
 40, 40, 40, 40, 41, 317, 370, 1538,  
 1542, 3292, 29393, 31139, 31336,  
 31391, 31424, 31745, 31813, 32031  
\q\_recursion\_tail . . . . . 39,  
 40, 40, 40, 40, 40, 41, 370, 370,  
 3292, 3298, 3304, 3313, 3320, 3325,  
 3330, 3337, 29393, 31138, 31335,  
 31390, 31423, 31744, 31812, 32030  
\q\_stop . . . . . 21,  
 21, 33, 38, 38, 39, 39, 54, 317,  
 369, 1537, 1540, 3288, 4329, 29271,  
 29275, 29286, 29305, 29310, 29321,  
 29325, 29337, 29338, 29344, 29346,  
 29347, 29349, 29352, 29368, 29376  
\s\_stop . . . . . 42, 42, 378, 3571, 3580  
quark internal commands:  
\s\_\_bool\_mark . . . . . 32195, 32216, 32230  
\q\_\_bool\_recursion\_stop . . . . .  
 . . . . . 9116, 9119, 9212, 9238  
\q\_\_bool\_recursion\_tail . . . . .  
 . . . . . 9116, 9212, 9238  
\s\_\_bool\_stop . . . . . 32195, 32216, 32230  
\q\_\_char\_no\_value . . . . . 10411, 10787  
\s\_\_char\_stop . . . . .  
 . . . . . 10410, 10741, 10746, 10787, 10789  
\q\_\_clist\_mark . . . . . 564  
\s\_\_clist\_mark . . . . . 551, 556, 558,  
 559, 560, 9812, 9845, 9846, 9863,  
 9985, 9995, 9999, 10021, 10077,  
 10083, 10097, 10109, 10110, 10111,  
 10114, 10115, 10116, 10125, 10126,  
 10135, 10289, 10290, 10302, 10303  
\q\_\_clist\_recursion\_stop . . . . .  
 . . . . . 563, 9816, 9831, 10230, 10266  
\q\_\_clist\_recursion\_tail . . . . .  
 . . . . . 561, 9816, 9831,  
 10178, 10192, 10212, 10230, 10266  
\q\_\_clist\_stop . . . . . 564  
\s\_\_clist\_stop . . . . . 559, 9812, 9814,  
 9815, 9970, 9973, 9985, 9988, 9996,  
 9999, 10007, 10021, 10083, 10111,  
 10114, 10115, 10127, 10135, 10291,  
 10302, 10303, 10304, 10330, 10364  
\s\_\_color\_stop . . . . . 28950  
\s\_\_cs\_mark . . . . . 326,  
 359, 360, 1786, 1787, 1790, 1791,  
 1792, 2856, 2886, 2887, 2889, 2895,  
 2899, 2921, 2930, 2949, 2977, 2980,  
 2988, 3003, 3035, 3049, 3053, 3062,  
 3081, 3090, 3095, 3184, 3187, 3203  
\q\_\_cs\_nil . . . . . 3190  
\q\_\_cs\_recursion\_stop . . . . .  
 . . . . . 2858, 2862, 2873, 3183  
\s\_\_cs\_stop . . . . . 326, 360,  
 1787, 1790, 1791, 1792, 2856, 2859,  
 2860, 2890, 2899, 2925, 2977, 2980,

2984, 2992, 2998, 3007, 3013, 3015,  
 3035, 3057, 3062, 3092, 3095, 3184  
`\s__deprecation_mark` .....  
     ..... [32567](#), 32570, 32572  
`\s__deprecation_stop` .....  
     .. [32567](#), 32570, 32572, 32592, 32601  
`\s__dim_mark` .... [14208](#), 14369, 14376  
`\s__dim_stop` ..... [14208](#),  
     14210, 14316, 14340, 14369, 14376  
`\q__file_nil` .....  
     .. [13400](#), 13484, 13498, 13586, 13592  
`\q__file_recursion_stop` .....  
     .. [13402](#), 13413, 13417, 13426, 13466  
`\q__file_recursion_tail` .....  
     ..... [13402](#), 13413, 13466  
`\s__file_stop` .. [665](#), 13372, 13377,  
     [13399](#), 13484, 13485, 13490, 13496,  
     13498, 13499, 13586, 13587, 13592,  
     13594, 13596, 13969, 13971, 13974,  
     13975, 13977, 13989, 14065, 14068,  
     14075, 14077, 14093, 14094, 14097  
`\s__fp` [729](#), [731](#), [735](#), [736](#), [759](#), [765](#),  
     [766](#), [769](#), [783](#), [784](#), [786](#), [815](#), [818](#),  
     [819](#), [820](#), [822](#), [828](#), [830](#), [919](#), [16212](#),  
     16225, 16226, 16227, 16228, 16229,  
     16239, 16244, 16246, 16247, 16262,  
     16275, 16278, 16280, 16290, 16302,  
     16322, 16339, 16342, 16349, 16356,  
     16372, 16399, 16505, 16507, 16509,  
     16510, 16511, 16513, 16514, 16515,  
     16517, 16533, 16691, 16696, 16923,  
     16977, 16986, 16988, 17665, 17823,  
     18309, 18324, 18348, 18368, 18369,  
     18501, 18526, 18527, 18541, 18542,  
     18579, 18580, 18693, 18694, 18695,  
     18704, 18720, 18724, 18788, 18789,  
     18792, 18803, 18804, 18812, 18813,  
     18815, 18816, 18817, 18819, 18820,  
     18821, 18833, 18836, 18840, 18843,  
     18863, 18913, 18916, 18919, 18939,  
     18940, 18942, 18943, 18944, 18952,  
     18955, 18966, 18967, 18969, 18978,  
     19054, 19206, 19240, 19241, 19244,  
     19325, 19463, 19471, 19473, 19650,  
     19659, 19661, 19666, 19674, 19676,  
     19678, 19681, 20184, 20196, 20198,  
     20407, 20424, 20426, 20607, 20626,  
     20628, 20629, 20632, 20649, 20652,  
     20655, 20680, 20681, 20683, 20699,  
     20788, 20801, 20803, 20806, 20811,  
     20844, 20860, 20943, 20956, 20958,  
     20971, 20973, 20986, 20988, 21001,  
     21003, 21016, 21018, 21031, 21041,  
     21542, 21558, 21559, 21563, 21574,  
     21681, 21694, 21696, 21712, 21715,  
     21725, 21748, 21759, 21761, 21775,  
     21777, 21782, 21844, 21865, 21868,  
     21898, 21919, 21922, 21972, 21988,  
     21991, 22066, 22067, 22177, 22179,  
     22211, 22477, 22485, 22488, 22567  
`\s__fp_(type)` ..... [759](#)  
`\s__fp_division` ..... [16220](#)  
`\s__fp_exact` ..... [16220](#), 16225,  
     16226, 16227, 16228, 16229, 18788  
`\s__fp_expr_mark` ... [765](#), [769](#), [790](#),  
     [794](#), [16215](#), 17872, 17885, 17967, 18011  
`\s__fp_expr_stop` ..... [737](#),  
     [16215](#), 16413, 17774, 17873, 17877,  
     17886, 18870, 18881, 18891, 18899  
`\s__fp_invalid` ..... [16220](#)  
`\s__fp_mark` [16217](#), 16362, 16363, 16367  
`\s__fp_overflow` ..... [16220](#), 16246  
`\s__fp_stop` .....  
     .. [735](#), [16217](#), 16219, 16263, 16339,  
     16350, 16357, 16363, 16367, 16381,  
     16400, 17192, 17196, 17704, 17709,  
     18309, 18331, 18500, 18501, 18526,  
     18527, 18693, 18694, 18695, 18862,  
     18863, 20483, 20498, 21818, 21822  
`\s__fp_tuple` ..... [735](#),  
     [16323](#), 16329, 16330, 16407, 16409,  
     18089, 18301, 18316, 18341, 18343,  
     18360, 18361, 18363, 18571, 18572,  
     19712, 19713, 19719, 19720, 21794  
`\s__fp_underflow` ..... [16220](#), 16244  
`\s__int_mark` .....  
     .. [8200](#), 8413, 8416, 8482, 8489  
`\q__int_recursion_stop` .....  
     .. [8203](#), 8900, 8917, 8960, 8987  
`\q__int_recursion_tail` .....  
     ..... [8203](#), 8900, 8917, 8960  
`\s__int_stop` ..... [509](#), [520](#),  
     [8200](#), 8202, 8392, 8408, 8410, 8414,  
     8427, 8482, 8489, 8893, 8899, 8916  
`\s__iow_mark` . [12949](#), 13262, 13269,  
     13281, 13355, 13356, 13357, 13358  
`\q__iow_nil` ..... [12952](#), 13148, 13155  
`\s__iow_stop` .....  
     .... [12949](#), 12951, 13148, 13189,  
     13247, 13285, 13298, 13355, 13358  
`\s__kernel_stop` [2190](#), 2198, 2207, 2216  
`\s__keys_mark` .....  
     [14937](#), 14983, 14986, 14994, 15001,  
     15599, 15602, 15607, 15613, 15847,  
     15850, 15859, 15861, 15866, 15869  
`\s__keys_nil` .....  
     .. [14937](#), 14978, 14979, 14981,  
     14983, 14986, 14991, 15000, 15001,

15009, 15594, 15595, 15597, 15599,  
 15602, 15605, 15613, 15614, 15846,  
 15849, 15855, 15857, 15865, 15868  
`\q__keys_no_value` .....  
     ..... 717, 14927, 14940, 15429,  
         15453, 15470, 15495, 15512, 15541  
`\s__keys_stop` 14937, 15019, 15029,  
     15181, 15582, 15592, 15779, 15799  
`\s__keyval_mark` 689, 689, 690, 690,  
     693, 14718, 14726, 14732, 14734,  
     14735, 14736, 14737, 14743, 14744,  
     14747, 14752, 14753, 14757, 14758,  
     14763, 14764, 14769, 14770, 14773,  
     14774, 14778, 14779, 14782, 14785,  
     14786, 14794, 14795, 14800, 14801,  
     14804, 14807, 14811, 14812, 14818,  
     14827, 14828, 14831, 14836, 14845,  
     14846, 14851, 14852, 14855, 14856,  
     14857, 14858, 14859, 14876, 14877,  
     14883, 14887, 14889, 14891, 14902  
`\s__keyval_nil` . 690, 14718, 14742,  
     14751, 14757, 14760, 14762, 14769,  
     14772, 14778, 14782, 14784, 14791,  
     14793, 14800, 14804, 14806, 14811,  
     14812, 14818, 14826, 14845, 14851,  
     14875, 14879, 14896, 14899, 14902  
`\s__keyval_stop` 14718, 14734, 14735,  
     14736, 14737, 14745, 14749, 14754,  
     14758, 14765, 14770, 14775, 14779,  
     14782, 14787, 14789, 14796, 14801,  
     14804, 14806, 14807, 14811, 14818,  
     14829, 14845, 14846, 14851, 14852,  
     14855, 14858, 14859, 14881, 14902  
`\s__keyval_tail` .....  
     . 689, 14718, 14726, 14730, 14731,  
         14741, 14825, 14834, 14835, 14857  
`\s__msg_mark` ..... 11755,  
     12109, 12174, 12175, 12180, 12183  
`\s__msg_stop` .. 628, 11755, 11757,  
     12111, 12115, 12117, 12176, 12712  
`\s__peek_mark` .....  
     ..... 11218, 11366, 11367, 11374  
`\s__peek_stop` .....  
     .. 11218, 11220, 11355, 11368, 11377  
`\s__prg_mark` ..... 1638, 1640, 1648  
`\q__prg_recursion_stop` .....  
     ..... 323, 1579, 1644, 1726  
`\q__prg_recursion_tail` .....  
     ..... 323, 1644, 1654, 1726, 1745  
`\s__prg_stop` 1665, 1670, 1689, 1697,  
     1705, 1756, 1760, 1762, 1764, 1766  
`\s__prop` .. 594, 595, 599, 605, 605,  
     1193, 11399, 11399, 11400, 11403,  
     11505, 11508, 11560, 11563, 11609,  
     11632, 11654, 11658, 11692, 11695,  
     11707, 11721, 11724, 32244, 32249  
`\s__prop_mark` .....  
     .. 597, 599, 11404, 11480, 11481,  
         11487, 11488, 11505, 11507, 11508  
`\q__prop_recursion_stop` 11406, 11468  
`\q__prop_recursion_tail` .....  
     .... 603, 11406, 11468, 11655, 11666  
`\s__prop_stop` ..... 597,  
     599, 11404, 11474, 11481, 11484,  
     11488, 11505, 11508, 32233, 32240  
`\s__quark` 378, 3294, 3529, 3531, 3532,  
     3543, 3546, 3551, 3554, 3556, 3577  
`\_\_quark_if_empty_if:n` ... 3358, 3519  
`\_\_quark_if_nil:w` ..... 372, 3358  
`\_\_quark_if_no_value:w` ..... 3358  
`\_\_quark_if_recursion_tail:w` ...  
     ..... 370, 375, 3310, 3337  
`\_\_quark_module_name:N` .....  
     ..... 376, 3386, 3409, 3524  
`\_\_quark_module_name:w` ..... 3524  
`\_\_quark_module_name_end:w` ... 3524  
`\_\_quark_module_name_loop:w` .. 3524  
`\_\_quark_new_conditional:Nnnn` . 3385  
`\_\_quark_new_conditional_aux_-`  
     do:NNnnn .... 376, 3506, 3508, 3509  
`\_\_quark_new_conditional_-`  
     define:NNNNn ..... 376, 3509  
`\_\_quark_new_conditional_N:Nnnn` 3505  
`\_\_quark_new_conditional_n:Nnnn` 3505  
`\_\_quark_new_test:NNNn` ..... 3385  
`\_\_quark_new_test_aux:Nn` .....  
     ..... 3386, 3387, 3397  
`\_\_quark_new_test_aux:nnNNnnnn` 3385  
`\_\_quark_new_test_aux_do:nNNnnnnNNn`  
     ..... 374, 375, 3440,  
     3445, 3450, 3455, 3460, 3466, 3469  
`\_\_quark_new_test_define_break_-`  
     ifx:nNNNNn ..... 3467, 3482  
`\_\_quark_new_test_define_break_-`  
     tl:nNNNNn ..... 3451, 3482  
`\_\_quark_new_test_define_-`  
     ifx:nNnNNn 374, 375, 3456, 3461, 3482  
`\_\_quark_new_test_define_-`  
     tl:nNnNNn 374, 375, 3441, 3446, 3482  
`\_\_quark_new_test_N:Nnnn` ..... 3438  
`\_\_quark_new_test_n:Nnnn` ..... 3438  
`\_\_quark_new_test_NN:Nnnn` ..... 3438  
`\_\_quark_new_test_Nn:Nnnn` ..... 3438  
`\_\_quark_new_test_nN:Nnnn` ..... 3448  
`\_\_quark_new_test_nn:Nnnn` ..... 3438  
`\q__quark_nil` .....  
`\_\_quark_quark_conditional_-`  
     name:N ..... 377, 3408, 3546

\\_\_quark\_quark\_conditional\_-  
     name:w ..... 377, 3546  
 \\_\_quark\_test\_define\_aux:NNNNnnNNn  
     ..... 375, 3469  
 \\_\_quark\_tmp:w .....  
     ..... 377, 3524, 3545, 3546, 3556  
 \q\_\_regex\_nil .....  
     ..... 23861, 25106, 25124, 25125  
 \q\_\_regex\_recursion\_stop .....  
     .. 23856, 23858, 23860, 25106, 25125  
 \s\_\_seq 483, 486, 487, 493, 497, 499,  
     1193, 7521, 7532, 7562, 7567, 7572,  
     7577, 7588, 7616, 7642, 7650, 7654,  
     7877, 7925, 8129, 32254, 32260, 32291  
 \s\_\_seq\_mark .....  
     ..... 7522, 8117, 8118, 8132, 8135  
 \s\_\_seq\_stop ..... 7522, 7830,  
     7833, 7841, 7843, 7924, 7925, 8119,  
     8132, 8135, 8137, 32253, 32254,  
     32256, 32260, 32264, 32266, 32271  
 \s\_\_skip\_stop ... 14554, 14615, 14617  
 \s\_\_sort\_mark ..... 952,  
     956, 957, 958, 22955, 23150, 23154,  
     23160, 23164, 23170, 23173, 23238,  
     23239, 23241, 23278, 23280, 23283,  
     23287, 23290, 23293, 23295, 23298  
 \s\_\_sort\_stop ..... 955,  
     956, 957, 958, 22955, 23226, 23235,  
     23239, 23241, 23278, 23279, 23280,  
     23285, 23287, 23291, 23293, 23301  
 \s\_\_str ..... 435, 443,  
     461, 464, 5346, 5484, 5488, 5688,  
     5735, 5803, 5806, 6248, 6260, 6265,  
     6275, 6280, 6285, 6288, 6303, 6316,  
     6319, 6454, 6455, 6472, 6478, 6494,  
     6500, 6501, 6606, 6621, 6630, 6631  
 \s\_\_str\_mark .....  
     ..... 415, 420, 427, 4718, 4895,  
     4926, 4933, 5006, 5023, 5271, 5273  
 \q\_\_str\_nil .....  
     ... 461, 5347, 6439, 6446, 6461, 6488  
 \q\_\_str\_recursion\_stop .....  
     ..... 4722, 5287, 5295, 5300  
 \q\_\_str\_recursion\_tail .....  
     419, 4722, 4940, 4949, 4966, 4986, 5287  
 \s\_\_str\_stop .....  
     ..... 420, 424, 460, 464, 4718,  
     4720, 4721, 4813, 4895, 4926, 4933,  
     5006, 5015, 5021, 5023, 5029, 5046,  
     5065, 5127, 5184, 5196, 5234, 5250,  
     5257, 5265, 5267, 5271, 5273, 5484,  
     5490, 5532, 5537, 5545, 5758, 5761,  
     5780, 5786, 6165, 6167, 6175, 6261,  
     6297, 6411, 6413, 6417, 6429, 6569,  
     6571, 6575, 6587, 6596, 6603, 6624  
 \q\_\_text\_nil .... 29402, 29814, 29815  
 \q\_\_text\_recursion\_stop .....  
     ... 29404, 29407, 29587, 29601,  
     29610, 29689, 29705, 29714, 29758,  
     29780, 29913, 29927, 29936, 29938,  
     30005, 30021, 30030, 30074, 30142,  
     30151, 30178, 30186, 30345, 30353,  
     30401, 30407, 30423, 30428, 30541,  
     30546, 30562, 30571, 30643, 30648,  
     30696, 30701, 30724, 30729, 30765,  
     30774, 31445, 31458, 31467, 31485,  
     31498, 31500, 31517, 31526, 31554  
 \q\_\_text\_recursion\_tail .....  
     ... 29404, 29534, 29587,  
     29688, 29758, 29780, 29913, 29938,  
     30004, 30074, 31445, 31484, 31554  
 \s\_\_text\_stop .....  
     .. 29401, 29476, 29478, 29814, 29815  
 \s\_\_tl ..... 960, 961, 961, 962,  
     963, 970, 970, 971, 23361, 23362,  
     23581, 23612, 23618, 23643, 23661,  
     23666, 23680, 23692, 23715, 23718  
 \s\_\_tl\_act\_stop ..... 406,  
     406, 407, 4462, 4468, 4469, 4472,  
     4475, 4479, 4488, 4491, 4494, 4497,  
     4500, 4502, 4504, 4508, 4511, 4517  
 \q\_\_tl\_mark .....  
     ... 388, 3759, 3866, 3868, 3870, 3872  
 \s\_\_tl\_mark ..... 396, 4107, 4117,  
     4224, 4225, 4228, 4231, 4232, 4707  
 \q\_\_tl\_nil ..... 388, 3759, 3892  
 \s\_\_tl\_nil .....  
     400, 4259, 4263, 4282, 4285, 4288, 4707  
 \q\_\_tl\_recursion\_stop ..... 3762  
 \q\_\_tl\_recursion\_tail .....  
     ... 3762, 4122, 4140, 4150, 4165, 4555  
 \q\_\_tl\_stop ..... 388, 3759, 3891  
 \s\_\_tl\_stop .....  
     ... 386, 399, 3849, 3851, 4065, 4071,  
     4107, 4117, 4226, 4228, 4233, 4235,  
     4265, 4288, 4310, 4312, 4330, 4345,  
     4359, 4383, 4408, 4694, 4704, 4707  
 \s\_\_token\_stop ..... 583,  
     585, 10879, 10983, 10986, 11016,  
     11051, 11158, 11162, 11168, 11191  
 \quitvmode ..... 694

**R**

\r 29543, 31804, 31828, 31854, 31978, 31979  
 \radical ..... 424  
 \raise ..... 425  
 rand ..... 217

**randint** ..... 217  
**\randomseed** ..... 934  
**\read** ..... 426  
**\readline** ..... 562  
**\readpapersizespecial** ..... 1153  
**\ref** ..... 29560, 29567  
**regex commands:**  
    **\c\_foo\_regex** ..... 226  
    **\regex\_(g)set:Nn** ..... 233  
    **\regex\_const:Nn** ..... 226, 233, 26376  
    **\regex\_count:NnN** ..... 234, 26419  
    **\regex\_count:nnN** ..... 234, 1055, 26419  
    **\regex\_extract\_all:NnN** ..... 234, 26423  
    **\regex\_extract\_all:nnN** ..... 227, 234, 977, 26423  
    **\regex\_extract\_all:NnNTF** ..... 234, 26423  
    **\regex\_extract\_all:nnNTF** ..... 234, 26423  
    **\regex\_extract\_once:NnN** ..... 234, 26423  
    **\regex\_extract\_once:nnN** ..... 234, 234, 26423  
    **\regex\_extract\_once:NnNTF** ..... 234, 26423  
    **\regex\_extract\_once:nnNTF** ..... 231, 234, 26423  
    **\regex\_gset:Nn** ..... 233, 26376  
    **\regex\_match:NnTF** ..... 233, 26409  
    **\regex\_match:nnTF** ..... 233, 26409  
    **\regex\_new:N** ..... 233,  
        979, 26370, 26372, 26373, 26374, 26375  
    **\regex\_replace\_all:NnN** ..... 235, 26423  
    **\regex\_replace\_all:nnN** ..... 227, 235, 26423  
    **\regex\_replace\_all:NnNTF** ..... 235, 26423  
    **\regex\_replace\_all:nnNTF** ..... 235, 26423  
    **\regex\_replace\_once:NnN** ..... 235, 26423  
    **\regex\_replace\_once:nnN** ..... 234, 235, 26423  
    **\regex\_replace\_once:NnNTF** ..... 235, 26423  
    **\regex\_replace\_once:nnNTF** ..... 235, 26423  
    **\regex\_set:Nn** ..... 233, 26376  
    **\regex\_show:N** ..... 233, 26391  
    **\regex\_show:n** ..... 226, 233, 26391  
    **\regex\_split:NnN** ..... 235, 26423  
    **\regex\_split:nnN** ..... 235, 26423  
    **\regex\_split:NnNTF** ..... 235, 26423  
    **\regex\_split:nnNTF** ..... 235, 26423  
    **\g\_tmpa\_regex** ..... 235, 26372  
    **\l\_tmpa\_regex** ..... 235, 26372  
    **\g\_tmpb\_regex** ..... 235, 26372  
    **\l\_tmpb\_regex** ..... 235, 26372  
**regex internal commands:**  
    **\\_regex\_action\_cost:n** .....  
        1022, 1025, 1033,  
        25493, 25494, 25502, 25950, 25976  
    **\\_regex\_action\_free:n** .....  
        1022, 1033, 25516, 25522,  
        25523, 25534, 25593, 25597, 25622,  
        25647, 25651, 25654, 25682, 25690,  
        25700, 25714, 25745, 25948, 25952  
**\\_regex\_action\_free\_aux:nn** .. 25952  
**\\_regex\_action\_free\_group:n** ...  
    1022, 1033, 25543, 25662, 25665, 25952  
**\\_regex\_action\_start\_wildcard:** .  
    1022, 25427, 25945  
**\\_regex\_action\_submatch:n** 1022,  
    25616, 25617, 25743, 25996, 25998  
**\\_regex\_action\_success:** .....  
    1022, 25430, 25444, 26003  
**\\_regex\_action\_wildcard:** .... 1037  
**\c\_regex\_all\_catcodes\_int** .....  
    24266, 24388, 24478, 25078  
**\\_regex\_anchor:N** .....  
    991, 1031, 24707, 25273, 25705  
**\c\_regex\_ascii\_lower\_int** .....  
    23855, 23920, 23926  
**\c\_regex\_ascii\_max\_control\_int** .  
    23852, 24036  
**\c\_regex\_ascii\_max\_int** .....  
    23852, 24029, 24037, 24218  
**\c\_regex\_ascii\_min\_int** .....  
    23852, 24028, 24035  
**\\_regex\_assertion:Nn** .. 991, 1031,  
    24707, 24732, 24741, 25267, 25705  
**\\_regex\_b\_test:** .....  
    991, 1031, 24732, 24741, 25272, 25705  
**\l\_regex\_balance\_int** ..... 979,  
    1044, 23850, 25779, 25792, 25907,  
    25911, 25912, 26091, 26120, 26313,  
    26330, 26625, 26651, 26674, 26678,  
    26679, 26686, 26687, 26691, 26692  
**\g\_regex\_balance\_intarray** .....  
    977, 979, 23847, 25906, 26064, 26079  
**\l\_regex\_balance\_tl** .....  
    1044, 1047, 26019, 26092, 26121, 26183  
**\\_regex\_branch:n** .....  
    991, 1008, 1027, 23844, 24393,  
    24888, 24941, 25120, 25249, 25588  
**\\_regex\_break\_point:TF** .....  
    980, 1003, 1025, 23863,  
    23869, 25493, 25494, 25711, 25734  
**\\_regex\_break\_true:w** .....  
    980, 981, 23863, 23869, 23874,  
    23881, 23888, 23894, 23901, 23909,  
    23956, 23968, 23985, 24682, 25726  
**\\_regex\_build:N** ..... 1054,  
    25414, 26416, 26422, 26426, 26430  
**\\_regex\_build:n** ..... 1054,  
    25414, 26411, 26420, 26425, 26428  
**\\_regex\_build\_for\_cs:n** 23980, 25432

```

__regex_build_new_state: 25424, 25425,
..... 25436, 25437, 25466, 25475, 25507,
25541, 25546, 25590, 25605, 25610,
25649, 25668, 25703, 25707, 25740
\l__regex_build_t1 1008,
23841, 24385, 24392, 24410, 24415,
24418, 24419, 24422, 24423, 24426,
24472, 24475, 24508, 24522, 24526,
24651, 24665, 24706, 24731, 24740,
24750, 24782, 24795, 24799, 24881,
24884, 24887, 24893, 24894, 24897,
24940, 25208, 25224, 25242, 25248,
25303, 25306, 25311, 25341, 25356,
25360, 25363, 25369, 26090, 26109,
26124, 26127, 26148, 26180, 26242,
26245, 26260, 26304, 26320, 26356
__regex_build_transition_-
left>NNN 25462, 25651, 25665, 25682
__regex_build_transition_-
right:nNn 25462,
25508, 25543, 25593, 25597, 25622,
25647, 25654, 25662, 25690, 25700
__regex_build_transitions_-
lazyness>NNNN 25473, 25515, 25521, 25533
\l__regex_capturing_group_int ...
..... 977,
1021, 1061, 25413, 25422, 25559,
25561, 25572, 25573, 25581, 25582,
25585, 26179, 26584, 26656, 26664
\l__regex_case_changed_char_int .
..... 982,
23890, 23893, 23904, 23907, 23908,
23915, 23919, 23925, 25758, 25876
\c__regex_catcode_A_int 24266
\c__regex_catcode_B_int 24266
\c__regex_catcode_C_int 24266
\c__regex_catcode_D_int 24266
\c__regex_catcode_E_int 24266
\c__regex_catcode_in_class_mode_-
int 24256,
24377, 24749, 24910, 25003, 25032
\g__regex_catcode_intarray
..... 977, 979, 23847, 25904, 25921
\c__regex_catcode_L_int 24266
\c__regex_catcode_M_int 24266
\c__regex_catcode_mode_int 24256,
24373, 24446, 24781, 25001, 25030
\c__regex_catcode_O_int 24266
\c__regex_catcode_P_int 24266
\c__regex_catcode_S_int 24266
\c__regex_catcode_T_int 24266
\c__regex_catcode_U_int 24266
\l__regex_catcodes_bool
..... 24263, 25037, 25041, 25076
\l__regex_catcodes_int 992,
24263, 24389, 24477, 24479, 24485,
24768, 24785, 24885, 24898, 24997,
25034, 25069, 25071, 25077, 25078
__regex_char_if_alphanumeric:N .
..... 24236
__regex_char_if_alphanumeric:NTF
..... 24214, 24439, 26287
__regex_char_if_special:N ... 24214
__regex_char_if_special:NTF ...
..... 24214, 24435
\g__regex_charcode_intarray ...
..... 977, 979, 23847, 25902, 25918
__regex_chk_c_allowed:TF ...
..... 24358, 24990
__regex_class:NnnN ...
..... 991, 998, 1000, 1006,
23845, 24473, 24776, 24777, 24783,
25137, 25216, 25226, 25264, 25487
\c__regex_class_mode_int ...
..... 24256, 24363, 24378
__regex_class_repeat:n ...
... 1025, 25497, 25503, 25519, 25528
__regex_class_repeat:nN 25498, 25512
__regex_class_repeat:nnN ...
..... 25499, 25526
__regex_command_K: ...
..... 991, 25242, 25265, 25738
__regex_compile:n ...
..... 24428,
25416, 26378, 26383, 26388, 26393
__regex_compile:w ...
..... 997, 24382, 24430, 25083
__regex_compile_$: ...
..... 24702
__regex_compile_(: ...
..... 24905
__regex_compile_): ...
..... 24944
__regex_compile_.: ...
..... 24673
__regex_compile/_A: ...
..... 24702
__regex_compile/_B: ...
..... 24726
__regex_compile/_b: ...
..... 24726
__regex_compile/_c: ...
..... 24989
__regex_compile/_D: ...
..... 24685
__regex_compile/_d: ...
..... 24685
__regex_compile/_G: ...
..... 24702
__regex_compile/_H: ...
..... 24685
__regex_compile/_h: ...
..... 24685
__regex_compile/_K: ...
..... 25239
__regex_compile/_N: ...
..... 24685
__regex_compile/_S: ...
..... 24685
__regex_compile/_s: ...
..... 24685
__regex_compile/_u: ...
..... 25157
__regex_compile/_V: ...
..... 24685
__regex_compile/_v: ...
..... 24685

```

\\_regex\_compile\_W: ..... 24685  
\\_regex\_compile\_w: ..... 24685  
\\_regex\_compile\_Z: ..... 24702  
\\_regex\_compile\_z: ..... 24702  
\\_regex\_compile[: ..... 24760  
\\_regex\_compile]: ..... 24744  
\\_regex\_compile^: ..... 24702  
\\_regex\_compile\_abort\_tokens:n .  
..... 24488, 24515, 24865, 24875  
\\_regex\_compile\_anchor:NTF .. 24702  
\\_regex\_compile\_c[:w ..... 25026  
\\_regex\_compile\_c:C>NN 25005, 25014  
\\_regex\_compile\_c\_lbrack\_add:N .  
..... 25026  
\\_regex\_compile\_c\_lbrack\_end: 25026  
\\_regex\_compile\_c\_lbrack\_-  
loop:NN ..... 25026  
\\_regex\_compile\_c\_test:NN ... 24989  
\\_regex\_compile\_class>NN ..... 24790  
\\_regex\_compile\_class:TFNN ....  
..... 1006, 24775, 24786, 24790  
\\_regex\_compile\_class\_catcode:w  
..... 24767, 24779  
\\_regex\_compile\_class\_normal:w ..  
..... 24770, 24773  
\\_regex\_compile\_class\_posix:NNNNw  
..... 24809  
\\_regex\_compile\_class\_posix\_-  
end:w ..... 24809  
\\_regex\_compile\_class\_posix\_-  
loop:w ..... 24809  
\\_regex\_compile\_class\_posix\_-  
test:w ..... 24763, 24809  
\\_regex\_compile\_cs\_aux:Nn ... 25092  
\\_regex\_compile\_cs\_aux:NNnnnN 25092  
\\_regex\_compile\_end: .....  
..... 997, 24382, 24455, 25101  
\\_regex\_compile\_end\_cs: 24451, 25092  
\\_regex\_compile\_escaped:N ....  
..... 24440, 24457  
\\_regex\_compile\_group\_begin:N ..  
.. 24879, 24927, 24932, 24950, 24952  
\\_regex\_compile\_group\_end: ....  
..... 24879, 24947  
\\_regex\_compile\_lparen:w ....  
..... 24914, 24918  
\\_regex\_compile\_one:n .....  
.... 24467, 24617, 24623, 24677,  
24688, 24691, 24701, 24856, 25108  
\\_regex\_compile\_quantifier:w ...  
..... 24486, 24497, 24755, 24899  
\\_regex\_compile\_quantifier\_\*:w ..  
..... 24531  
\\_regex\_compile\_quantifier\_+:w ..  
..... 24531  
\\_regex\_compile\_quantifier\_?:w ..  
..... 24531  
\\_regex\_compile\_quantifier\_-  
abort:nNN .....  
.. 24506, 24541, 24560, 24573, 24596  
\\_regex\_compile\_quantifier\_-  
braced\_auxi:w ..... 24537  
\\_regex\_compile\_quantifier\_-  
braced\_auxii:w ..... 24537  
\\_regex\_compile\_quantifier\_-  
braced\_auxiii:w ..... 24537  
\\_regex\_compile\_quantifier\_-  
lazyness:nnNN .....  
..... 1000, 24518, 24532,  
24534, 24536, 24549, 24569, 24591  
\\_regex\_compile\_quantifier\_-  
none: ..... 24502, 24504, 24506  
\\_regex\_compile\_range:Nw ....  
..... 24615, 24628  
\\_regex\_compile\_raw:N .....  
.... 24308, 24436, 24440, 24442,  
24460, 24465, 24493, 24608, 24610,  
24630, 24676, 24722, 24758, 24806,  
24826, 24844, 24902, 24907, 24912,  
24928, 24938, 24946, 24964, 24965,  
24966, 24972, 24983, 24984, 24985,  
24993, 25048, 25097, 25169, 25175  
\\_regex\_compile\_raw\_error:N ...  
..... 24605,  
24713, 24729, 24738, 25160, 25243  
\\_regex\_compile\_special:N ....  
..... 993, 24436, 24457,  
24499, 24520, 24547, 24552, 24567,  
24580, 24614, 24633, 24793, 24811,  
24830, 24850, 24851, 24920, 24955,  
24973, 25016, 25035, 25162, 25178  
\\_regex\_compile\_special\_group\_-  
::w ..... 24953  
\\_regex\_compile\_special\_group\_-  
::w ..... 24949  
\\_regex\_compile\_special\_group\_-  
i:w ..... 24953  
\\_regex\_compile\_special\_group\_-  
!w ..... 24949  
\\_regex\_compile\_u\_end: .....  
..... 25181, 25187, 25192  
\\_regex\_compile\_u\_in\_cs: .....  
..... 25198, 25201  
\\_regex\_compile\_u\_in\_cs\_aux:n ..  
..... 25211, 25214  
\\_regex\_compile\_u\_loop>NN ... 25157

```

__regex_compile_u_not_cs:
..... 25196, 25220
__regex_compile_|:
__regex_compute_case_changed_-
char: 23891, 23905, 23913
__regex_count:nnN 26420, 26422, 26467
\c__regex_cs_in_class_mode_int ..
..... 24256, 25089
\c__regex_cs_mode_int . 24256, 25087
\l__regex_cs_name_tl ..
..... 23851, 23977, 23983
\l__regex_curr_catcode_int 23935,
23954, 23962, 23974, 25758, 25920
\l__regex_curr_char_int ..
..... 23873, 23879,
23880, 23887, 23899, 23900, 23915,
23916, 23917, 23918, 23924, 23955,
24681, 25732, 25758, 25875, 25917
__regex_curr_cs_to_str: ..
..... 23827, 23965, 23977
\l__regex_curr_pos_int ..
..... 978, 23830, 25443, 25725, 25753,
25780, 25782, 25785, 25793, 25800,
25807, 25835, 25845, 25874, 25889,
25903, 25905, 25907, 25908, 25909,
25919, 25922, 26001, 26010, 26519
\l__regex_curr_state_int ..
..... 1033, 1040, 25762,
25927, 25928, 25930, 25935, 25938,
25960, 25965, 25970, 25971, 25979
\l__regex_curr_submatches_prop ..
..... 25763, 25842, 25940,
25972, 25973, 25991, 26000, 26012
\l__regex_default_catcodes_int ..
..... 992, 24263, 24387,
24389, 24485, 24785, 24885, 24898
__regex_disable_submatches: ..
..... 23979, 25084, 25993, 26461, 26470
\l__regex_empty_success_bool ..
..... 25771, 25827, 25831, 26008, 26529
__regex_escape_□:w .. 24102
__regex_escape_/_a:w .. 24102
__regex_escape_/_break:w .. 24102
__regex_escape_/_e:w .. 24102
__regex_escape_/_f:w .. 24102
__regex_escape_/_n:w .. 24102
__regex_escape_/_r:w .. 24102
__regex_escape_/_t:w .. 24102
__regex_escape_/_x:w .. 24121
__regex_escape_\:w .. 24086
__regex_escape_break:w .. 24102
__regex_escape_escaped:N .. .
..... 24072, 24096, 24099
__regex_escape_loop:N .. .
..... 986, 24079, 24086, 24121,
24157, 24165, 24166, 24183, 24192
__regex_escape_raw:N .. .
..... 987, 24073, 24099, 24110, 24112,
24114, 24116, 24118, 24120, 24134
__regex_escape_unescaped:N ..
..... 24071, 24089, 24099
__regex_escape_use:nnnn .. .
..... 986, 997, 24067, 24433, 26093
__regex_escape_x:N 988, 24156, 24160
__regex_escape_x_end:w .. 987, 24121
__regex_escape_x_large:n .. 24121
__regex_escape_x_loop:N ..
..... 988, 24153, 24169
__regex_escape_x_loop_error: . 24169
__regex_escape_x_loop_error:n ..
..... 24172, 24184, 24189
__regex_escape_x_test:N ..
..... 987, 24124, 24138
__regex_escape_x_testii:N .. 24138
\l__regex_every_match_tl ..
..... 25770, 25849, 25853, 25862
__regex_extract: ... 1056, 26485,
26491, 26503, 26580, 26624, 26647
__regex_extract_all:nnN 26434, 26479
__regex_extract_b:wn .. 26580
__regex_extract_e:wn .. 26580
__regex_extract_once:nnN ..
..... 26432, 26479
__regex_extract_seq_aux:n ..
..... 26545, 26563
__regex_extract_seq_aux:ww .. 26563
\l__regex_fresh_thread_bool ..
..... 1034, 1039, 25744, 25750,
25771, 25887, 25947, 25949, 26009
__regex_get_digits:NTFw .. .
..... 24294, 24539, 24554
__regex_get_digits_loop:nw ..
..... 24297, 24300, 24303
__regex_get_digits_loop:w .. 24294
__regex_group:nnN .. 991, 1008,
24927, 24932, 25258, 25428, 25556
__regex_group_aux:nnnn .. .
..... 1027, 25538, 25558, 25566, 25569
__regex_group_aux:nnnnN .. 1027
__regex_group_end_extract_seq:N ..
..... 26486, 26494, 26534, 26536
__regex_group_end_replace:N ..
..... 26641, 26670, 26672
\l__regex_group_level_int ..
..... 24255, 24386,
24404, 24406, 24408, 24886, 24892

```

\\_\\_regex\\_group\\_no\\_capture:nnN . . . . .  
 ..... 991, 24950, 25260, 25556  
 \\_\\_regex\\_group\\_repeat:nn 25551, 25600  
 \\_\\_regex\\_group\\_repeat:nnN . . . . .  
 ..... 25552, 25640  
 \\_\\_regex\\_group\\_repeat:nnnN . . . . .  
 ..... 25553, 25671  
 \\_\\_regex\\_group\\_repeat\\_aux:n . . . . .  
 1028, 1029, 25607, 25620, 25658, 25675  
 \\_\\_regex\\_group\\_resetting:nnnN . . . . .  
 ..... 991, 24952, 25262, 25567  
 \\_\\_regex\\_group\\_resetting\_-  
 loop:nnNn . . . . . 25567  
 \\_\\_regex\\_group\\_submatches:nNN . . . . .  
 .. 25608, 25613, 25643, 25659, 25673  
 \\_\\_regex\\_hexadecimal\\_use:N . . . . . 24194  
 \\_\\_regex\\_hexadecimal\\_use:NTF . . . . .  
 ..... 24155, 24164, 24174, 24194  
 \\_\\_regex\\_if\\_end\\_range:NN . . . . . 24628  
 \\_\\_regex\\_if\\_end\\_range:NNTF . . . . . 24628  
 \\_\\_regex\\_if\\_in\\_class:TF . . . . .  
 ..... 24318, 24397, 24470,  
 24486, 24612, 24675, 24746, 24762,  
 24907, 24938, 24946, 26747, 26760  
 \\_\\_regex\\_if\\_in\\_class\\_or\\_catcode:TF  
 .. 24338, 24704, 24728, 24737, 25159  
 \\_\\_regex\\_if\\_in\\_cs:TF . . . . .  
 ..... 24326, 25095, 26745, 26754  
 \\_\\_regex\\_if\\_match:nn . . . . .  
 ..... 26411, 26416, 26458  
 \\_\\_regex\\_if\\_raw\\_digit:NN . . . . . 24306  
 \\_\\_regex\\_if\\_raw\\_digit:NNTF . . . . .  
 ..... 24296, 24302, 24306  
 \\_\\_regex\\_if\\_two\\_empty\\_matches:TF  
 ... 1034, 25771, 25832, 25838, 26005  
 \\_\\_regex\\_if\\_within\\_catcode:TF . . . . .  
 ..... 24350, 24765  
 \\_\\_regex\\_int\\_eval:w . . . . .  
 .. 23795, 26032, 26033, 26044, 26601  
 \\_\\_regex\\_internal\\_a\\_int . . . . .  
 ..... 1000, 1047, 23833,  
 24539, 24550, 24561, 24570, 24574,  
 24582, 24585, 24589, 24592, 24599,  
 25520, 25523, 25529, 25534, 25609,  
 25624, 25630, 25636, 25645, 25648,  
 25652, 25655, 25660, 25663, 25666,  
 25681, 25689, 25698, 26195, 26216  
 \\_\\_regex\\_internal\\_a\\_tl . . . . .  
 .... 986, 1016, 1016, 1017, 1058,  
 1061, 23833, 23964, 23967, 24070,  
 24077, 24084, 24833, 24838, 24854,  
 24859, 24864, 24868, 24874, 24875,  
 25103, 25114, 25164, 25194, 25206,  
 25222, 25252, 25255, 25306, 25321,  
 25363, 25370, 25457, 25458, 25459,  
 25460, 25591, 25592, 25596, 25598,  
 26397, 26406, 26630, 26660, 26690  
 \\_\\_regex\\_internal\\_b\\_int . . . . .  
 .... 23833, 24554, 24583, 24586,  
 24587, 24589, 24593, 24600, 25625,  
 25630, 25635, 25681, 25689, 25698  
 \\_\\_regex\\_internal\\_b\\_tl . . . . . 23833  
 \\_\\_regex\\_internal\\_bool . . . . .  
 .. 23833, 24832, 24837, 24858, 24867  
 \\_\\_regex\\_internal\\_c\\_int . . . . .  
 .. 23833, 25627, 25632, 25633, 25637  
 \\_\\_regex\\_internal\\_regex . . . . .  
 . 996, 24279, 24426, 25105, 25111,  
 25417, 26379, 26384, 26389, 26394  
 \\_\\_regex\\_internal\\_seq . . . . . 23833,  
 25387, 25388, 25393, 25400, 25401,  
 25402, 25404, 26540, 26558, 26561  
 \\_\\_regex\\_internal\\_tl . . . . .  
 .. 23833, 24075, 24079, 25203, 25210  
 \\_\\_regex\\_item\\_caseful\\_equal:n . . . . .  
 ..... 991, 23871, 23996,  
 23997, 24001, 24002, 24003, 24004,  
 24005, 24014, 24019, 24037, 24055,  
 24390, 24977, 25139, 25217, 25274  
 \\_\\_regex\\_item\\_caseful\\_range:nn . . . . .  
 ..... 991, 23871, 23993,  
 24008, 24011, 24012, 24013, 24027,  
 24034, 24041, 24043, 24045, 24048,  
 24049, 24050, 24051, 24056, 24059,  
 24064, 24065, 24391, 24979, 25276  
 \\_\\_regex\\_item\\_caseless\\_equal:n . . . . .  
 ..... 991, 23885, 24958, 25281  
 \\_\\_regex\\_item\\_caseless\\_range:nn . . . . .  
 ..... 991, 23885, 24960, 25283  
 \\_\\_regex\\_item\\_catcode: . . . . . 23932  
 \\_\\_regex\\_item\\_catcode:nTF . . . . .  
 991, 1006, 23932, 24479, 24787, 25288  
 \\_\\_regex\\_item\\_catcode\\_reverse:nTF  
 ..... 991, 23932, 24788, 25290  
 \\_\\_regex\\_item\\_cs:n . . . . .  
 ..... 979, 991, 23972, 25111, 25297  
 \\_\\_regex\\_item\\_equal:n . . . . .  
 .... 23930, 24390, 24618, 24624,  
 24654, 24667, 24668, 24957, 24976  
 \\_\\_regex\\_item\\_exact:nn . . . . .  
 .... 991, 1016, 23952, 25232, 25294  
 \\_\\_regex\\_item\\_exact\\_cs:n . . . . .  
 991, 1013, 23952, 25113, 25229, 25296  
 \\_\\_regex\\_item\\_range:nn . . . . .  
 .. 23930, 24391, 24656, 24959, 24978  
 \\_\\_regex\\_item\\_reverse:n . . . . .  
 .... 991, 1007, 23866, 23951,  
 24018, 24692, 24858, 25292, 25735

```

\l__regex_last_char_int 25732, 25758, 25875
\l__regex_left_state_int 25409, 25426, 25451, 25458,
25469, 25476, 25479, 25480, 25482,
25483, 25509, 25517, 25520, 25544,
25592, 25594, 25604, 25624, 25644,
25646, 25674, 25677, 25680, 25683,
25695, 25708, 25717, 25741, 25748
\l__regex_left_state_seq 25409, 25450, 25457, 25591
__regex_match:n 25777, 26464, 26474,
26484, 26493, 26518, 26620, 26650
\l__regex_match_count_int 1055, 1056, 26441, 26471, 26472, 26477
__regex_match_cs:n 23983, 25777
__regex_match_init: 25777
__regex_match_loop: 1036, 1039, 25848, 25871
__regex_match_once: 1036, 1037, 25788, 25810, 25829, 25867
__regex_match_one_active:n .. 25871
\l__regex_match_success_bool 1035,
25774, 25841, 25857, 25864, 26007
\l__regex_max_active_int 1023, 25434, 25766, 25843, 25880,
25883, 25888, 25985, 25986, 25990
\l__regex_max_pos_int 1043,
24717, 24718, 24725, 25381, 25443,
25753, 25785, 25796, 25807, 25889,
26519, 26524, 26530, 26639, 26668
\l__regex_max_state_int 1021, 1023, 1068, 25406, 25423,
25435, 25468, 25470, 25471, 25530,
25542, 25603, 25623, 25625, 25633,
25677, 25683, 25691, 25701, 25780,
25795, 25816, 25821, 25825, 26998
\l__regex_min_active_int 1023, 25766,
25821, 25843, 25880, 25882, 25888
\l__regex_min_pos_int 1043, 24715, 24724,
25379, 25753, 25782, 25800, 25823
\l__regex_min_state_int 1023, 25406, 25423, 25434,
25435, 25795, 25816, 25844, 26997
\l__regex_min_submatch_int 1055, 1058, 1061, 25824,
25826, 26444, 26542, 26655, 26663
\l__regex_mode_int 24256,
24320, 24328, 24331, 24340, 24343,
24352, 24360, 24363, 24373, 24374,
24376, 24378, 24432, 24446, 24448,
24748, 24752, 24753, 24754, 24781,
24792, 24909, 24999, 25000, 25028,
25029, 25085, 25086, 25195, 25241
__regex_mode_quit_c: 24371, 24469, 24882
__regex_msg_repeated:nnN 25336, 25357, 25367, 26967
__regex_multi_match:n 1034,
25851, 26472, 26491, 26499, 26647
\c__regex_no_match_regex 23842, 24279, 26371
\c__regex_outer_mode_int 24256, 24331, 24343, 24352, 24360,
24374, 24432, 24448, 25195, 25241
__regex_pop_lr_states: 25440, 25448, 25549
__regex_posix_alnum: 24021
__regex_posix_alpha: 24021
__regex_posix_ascii: 24021
__regex_posix_blank: 24021
__regex_posix_cntrl: 24021
__regex_posix_digit: 24021
__regex_posix_graph: 24021
__regex_posix_lower: 24021
__regex_posix_print: 24021
__regex_posix_punct: 24021
__regex_posix_space: 24021
__regex_posix_upper: 24021
__regex_posix_word: 24021
__regex_posix_xdigit: 24021
__regex_prop_..: 1003, 24673
__regex_prop_d: .. 1003, 23992, 24039
__regex_prop_h: 23992, 24031
__regex_prop_N: 23992, 24701
__regex_prop_s: 23992
__regex_prop_v: 23992
__regex_prop_w: 23992, 24060, 25733, 25735, 25736
__regex_push_lr_states: 25438, 25448, 25547
__regex_quark_if_nil:N 23862
__regex_quark_if_nil:NTF 25129, 25149
__regex_quark_if_nil:nTF 23862
__regex_quark_if_nil_p:n 23862
__regex_query_get: 25847, 25877, 25915
__regex_query_range:nn 1043, 26024,
26029, 26048, 26133, 26634, 26667
__regex_query_range_loop:ww . 26029

```

\\_regex\_query\_set:nnn . . . . .  
    . . . . . 1035, 25781, 25784,  
    25786, 25799, 25803, 25808, 25900  
\\_regex\_query\_submatch:n . . . . .  
    . . . . . 26046, 26181, 26574  
\\_regex\_replace\_all:nnN 26438, 26644  
\\_regex\_replace\_once:nnN . . . . .  
    . . . . . 26436, 26614  
\\_regex\_replacement:n . . . . .  
    . . . . . 26087, 26619, 26649  
\\_regex\_replacement\_aux:n . . . 26087  
\\_regex\_replacement\_balance\_-  
    one\_match:n . . . . . 1042,  
    1042, 26020, 26118, 26627, 26658  
\\_regex\_replacement\_c:w . . . 26225  
\\_regex\_replacement\_c\_A:w . . .  
    . . . . . 1046, 26306  
\\_regex\_replacement\_c\_B:w . . . 26309  
\\_regex\_replacement\_c\_C:w . . . 26318  
\\_regex\_replacement\_c\_D:w . . . 26323  
\\_regex\_replacement\_c\_E:w . . . 26326  
\\_regex\_replacement\_c\_L:w . . . 26335  
\\_regex\_replacement\_c\_M:w . . . 26338  
\\_regex\_replacement\_c\_O:w . . . 26341  
\\_regex\_replacement\_c\_P:w . . . 26344  
\\_regex\_replacement\_c\_S:w . . . 26350  
\\_regex\_replacement\_c\_T:w . . . 26358  
\\_regex\_replacement\_c\_U:w . . . 26361  
\\_regex\_replacement\_cat:NNN . . . . .  
    . . . . . 26233, 26266  
\\_regex\_replacement\_category\_-  
    seq . . . . . 26017,  
    26112, 26115, 26116, 26152, 26280  
\\_regex\_replacement\_category\_-  
    tl . . . . . 1046, 26017,  
    26147, 26153, 26159, 26281, 26282  
\\_regex\_replacement\_char:nNN . . .  
    . . . . . 1052, 26301,  
    26308, 26315, 26325, 26332, 26337,  
    26340, 26343, 26347, 26360, 26363  
\\_regex\_replacement\_csnames\_-  
    int . . . . . 1042, 26016, 26106,  
    26108, 26110, 26182, 26241, 26248,  
    26259, 26261, 26271, 26312, 26329  
\\_regex\_replacement\_cu\_aux:Nw . . .  
    . . . . . 26230, 26239, 26254  
\\_regex\_replacement\_do\_one\_-  
    match:n . . . . . 26022, 26131, 26632, 26666  
\\_regex\_replacement\_error:NNN . . .  
    . . . . . 26196, 26208,  
    26219, 26234, 26237, 26255, 26365  
\\_regex\_replacement\_escaped:N . . .  
    . . . . . 26102, 26165, 26285  
\\_regex\_replacement\_exp\_not:N . . .  
    . . . . . 1048, 26028, 26230  
\\_regex\_replacement\_g:w . . . . . 26191  
\\_regex\_replacement\_g\_digits:NN  
    . . . . . 26191  
\\_regex\_replacement\_normal:n . . .  
    26098, 26103, 26145, 26172, 26194,  
    26200, 26227, 26253, 26263, 26278  
\\_regex\_replacement\_put\_-  
    submatch:n . . . . . 26170, 26177, 26215  
\\_regex\_replacement\_rbrace:N . . .  
    . . . . . 26096, 26214, 26257  
\\_regex\_replacement\_u:w . . . . . 26250  
\\_regex\_return: . . . . . 1054,  
    26412, 26417, 26428, 26430, 26450  
\\_regex\_right\_state\_int . . . . .  
    . . . . . 25409, 25429, 25441,  
    25453, 25460, 25469, 25470, 25509,  
    25516, 25522, 25535, 25542, 25544,  
    25594, 25598, 25609, 25623, 25632,  
    25644, 25648, 25652, 25655, 25660,  
    25663, 25666, 25674, 25688, 25691,  
    25694, 25697, 25701, 25717, 25748  
\\_regex\_right\_state\_seq . . . . .  
    . . . . . 25409, 25452, 25459, 25596  
\\_regex\_saved\_success\_bool . . . . .  
    . . . . . 1035, 23981, 23988, 25774  
\\_regex\_show:N . . . . . 25245, 26394, 26403  
\\_regex\_show\_anchor\_to\_str:N . . . . .  
    . . . . . 25273, 25374  
\\_regex\_show\_class:NnnnN . . . . .  
    . . . . . 25264, 25338  
\\_regex\_show\_group\_aux:nnnnN . . . . .  
    . . . . . 25259, 25261, 25263, 25329  
\\_regex\_show\_item\_catcode:NnTF . . .  
    . . . . . 25289, 25291, 25385  
\\_regex\_show\_item\_exact\_cs:n . . . . .  
    . . . . . 25296, 25398  
\\_regex\_show\_lines\_int . . . . .  
    . . . . . 24281, 25310, 25342, 25345, 25352  
\\_regex\_show\_one:n . . . . . 25253,  
    25266, 25269, 25275, 25278, 25282,  
    25285, 25295, 25299, 25308, 25324,  
    25331, 25335, 25348, 25364, 25403  
\\_regex\_show\_pop: . . . . . 25318, 25334  
\\_regex\_show\_prefix\_seq . . . . .  
    . . . . . 24280, 25251,  
    25254, 25300, 25314, 25319, 25321  
\\_regex\_show\_push:n . . . . .  
    . . . . . 25301, 25318, 25332, 25343  
\\_regex\_show\_scope:nn . . . . .  
    . . . . . 25293, 25298, 25318, 25390  
\\_regex\_single\_match: . . . . . 1034,  
    23978, 25851, 26462, 26482, 26617

```

__regex_split:nnN 26440, 26496
__regex_standard_escapechar: ...
 23796, 24074, 24431, 25421
\l__regex_start_pos_int
 24716, 25380,
25753, 25835, 25840, 25846, 26502,
26514, 26527, 26530, 26604, 26668
\g__regex_state_active_intarray ..
 976, 1023, 1034, 1034, 1035, 25768,
 25819, 25926, 25929, 25937, 25964
\l__regex_step_int
 976, 25765, 25822, 25873,
 25927, 25931, 25939, 25953, 25955
__regex_store_state:n
 25844, 25978, 25981
__regex_store_submatches:
 25981, 25995
__regex_submatch_balance:n ...
 .. 26021, 26052, 26122, 26185, 26566
\g__regex_submatch_begin_-
 intarray 977, 1042, 26026,
 26049, 26074, 26082, 26140, 26447,
 26509, 26512, 26525, 26586, 26610
\g__regex_submatch_end_intarray ..
 . 977, 26050, 26059, 26067, 26447,
 26506, 26522, 26588, 26613, 26636
\l__regex_submatch_int
 977, 1055, 1057, 1058, 1061,
 25826, 26444, 26521, 26523, 26526,
 26528, 26531, 26543, 26583, 26587,
 26589, 26591, 26592, 26657, 26665
\g__regex_submatch_prev_intarray
 .. 977, 1055, 1059, 26025, 26136,
 26447, 26504, 26520, 26590, 26603
\g__regex_success_bool 1035,
 23982, 23984, 23987, 25774, 25814,
 25856, 25865, 26452, 26582, 26621
\l__regex_success_pos_int
 .. 25753, 25823, 25840, 26010, 26502
\l__regex_success_submatches_-
 prop 1033, 1059, 25763, 26011, 26594
__regex_tests_action_cost:n ...
 25487, 25508, 25517, 25535
\g__regex_thread_state_intarray .
 977, 1023, 1032,
 1034, 1034, 1040, 25768, 25897, 25984
__regex_tmp:w
 23815, 23817, 23821, 23823, 23832,
 24685, 24695, 24696, 24697, 24698,
 24699, 24710, 24715, 24716, 24717,
 24718, 24719, 24724, 24725, 26423,
 26432, 26434, 26436, 26438, 26440
__regex_toks_clear:N . 23799, 25468
__regex_toks_memcpy>NNn 23804, 25634
__regex_toks_put_left:Nn
 23813, 25463, 25616, 25617
__regex_toks_put_right:Nn
 978, 23813, 25426, 25429,
 25441, 25465, 25476, 25708, 25741
__regex_toks_set:Nn
 23799, 25908, 25990
__regex_toks_use:w
 .. 23798, 25898, 25928, 26042, 27001
__regex_trace:nnn 26983, 27000
__regex_trace_pop:nnN 26983
__regex_trace_push:nnN 26983
\g__regex_trace_regex_int 26993
__regex_trace_states:n 26994
__regex_two_if_eq:NNNN 24282
__regex_two_if_eq:NNNTF
 24282, 24520, 24567,
 24580, 24614, 24793, 24830, 24850,
 24851, 24920, 24955, 24972, 24973,
 25035, 25162, 26193, 26252, 26278
__regex_use_i_delimit_by_q_-
 recursion_stop:nw .. 23857, 25152
__regex_use_none_delimit_by_q_-
 recursion_stop:w
 23857, 25130, 25154
__regex_use_state:
 25924, 25941, 25967
__regex_use_state_and_submatches:nn
 1037, 25896, 25933
\l__regex_zeroth_submatch_int ...
 1055, 1059, 26444,
 26505, 26507, 26510, 26513, 26583,
 26601, 26604, 26628, 26633, 26637
\relax 4, 21, 25,
 30, 68, 70, 71, 72, 83, 90, 111, 124,
 125, 126, 127, 128, 129, 130, 131,
 132, 133, 134, 136, 137, 138, 139,
 140, 141, 142, 143, 144, 145, 146, 427
\relpenalty 428
\RequirePackage 114
\resettimer 785
reverse commands:
 \reverse_if:N
 23, 427, 509, 510, 768, 1417,
 5264, 8291, 8440, 8442, 8444, 8446,
 8501, 9266, 14328, 14333, 14337,
 14339, 17094, 20669, 21491, 21514,
 23879, 23880, 23899, 23900, 23907,
 23908, 29289, 29291, 29313, 29337
\right 429
right commands:
 \c_right_brace_str 71, 5316,
 13455, 24182, 24547, 24567, 24580,
 25093, 25097, 25180, 26095, 29662
```

|                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-----------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \rightghost . . . . .                                                 | 895                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | 14656, 14676, 14678, 14686, 14688,                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| \righthyphenmin . . . . .                                             | 430                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | 14692, 14694, 14698, 15969, 15976,                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| \rightmarginkern . . . . .                                            | 695                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | 16198, 16387, 17092, 17096, 17297,                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| \rightskip . . . . .                                                  | 431                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | 17314, 17615, 17662, 17663, 17922,                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| \rmfamily . . . . .                                                   | 31645                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | 17965, 17995, 18009, 18766, 20579,                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| \romannumeral . . . . .                                               | 432                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | 20587, 21333, 21336, 21339, 21342,                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| round . . . . .                                                       | 214                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | 21345, 21348, 21351, 21354, 21357,                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| \rpcode . . . . .                                                     | 696                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | 22622, 22649, 22989, 23484, 23524,                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| \rule . . . . .                                                       | 28745, 28746                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | 23528, 23534, 23536, 23583, 23585,                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | 23965, 23966, 24304, 25122, 25400,                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | 25919, 25922, 25943, 26303, 26355,                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | 27009, 27152, 28747, 29414, 29415,                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | 32613, 32616, 32961, 32972, 33018                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | <b>S</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| s@ internal commands:                                                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \sa_ . . . . .                                                        | 963                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \saveboxresource . . . . .                                            | 938                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \savecatcodetable . . . . .                                           | 896                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \saveimageresource . . . . .                                          | 939                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \savepos . . . . .                                                    | 937                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \savinghyphcodes . . . . .                                            | 563                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \savingvdiscards . . . . .                                            | 564                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| scan commands:                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scan_align_safe_stop: . . . . .                                      | 32837                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scan_new:N . . . . .                                                 | 41, 378, 413,<br>3559, 4462, 4707, 4708, 4709, 4718,<br>4719, 5346, 7521, 7522, 7523, 8200,<br>8201, 9812, 9813, 10410, 10879,<br>11218, 11219, 11399, 11404, 11405,<br>11755, 11756, 12949, 12950, 13399,<br>14208, 14209, 14554, 14718, 14719,<br>14720, 14721, 14937, 14938, 14939,<br>16212, 16215, 16216, 16217, 16218,<br>16220, 16221, 16222, 16223, 16224,<br>16323, 22955, 22956, 23362, 28950,<br>29401, 32195, 32196, 32567, 32568                                                                                                                                                                                                                                                                                                                                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scan_stop: . . . . .                                                 | 9, 17, 18, 18,<br>18, 41, 41, 90, 163, 177, 306, 310,<br>310, 326, 327, 329, 338, 342, 343,<br>354, 378, 380, 384, 391, 392, 393,<br>398, 405, 427, 509, 514, 528, 583,<br>583, 591, 593, 594, 605, 628, 673,<br>674, 674, 679, 765, 768, 769, 770,<br>774, 991, 1013, 1446, 1798, 1816,<br>1826, 1844, 1870, 2200, 2209, 2218,<br>2290, 2732, 2856, 2857, 2872, 2912,<br>2938, 2962, 2979, 3183, 3189, 3294,<br>3568, 3571, 3787, 3957, 3967, 4034,<br>4063, 4065, 4417, 4437, 4451, 5265,<br>5445, 6450, 7875, 8631, 9574, 9578,<br>9784, 10923, 10995, 11290, 11389,<br>11392, 11394, 12778, 12783, 12867,<br>12976, 12979, 13529, 13536, 13925,<br>14220, 14239, 14241, 14245, 14248,<br>14251, 14255, 14260, 14264, 14487,<br>14564, 14582, 14584, 14592, 14594,<br>14598, 14600, 14621, 14627, 14630, |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | 14656, 14676, 14678, 14686, 14688,<br>14692, 14694, 14698, 15969, 15976,<br>16198, 16387, 17092, 17096, 17297,<br>17314, 17615, 17662, 17663, 17922,<br>17965, 17995, 18009, 18766, 20579,<br>20587, 21333, 21336, 21339, 21342,<br>21345, 21348, 21351, 21354, 21357,<br>22622, 22649, 22989, 23484, 23524,<br>23528, 23534, 23536, 23583, 23585,<br>23965, 23966, 24304, 25122, 25400,<br>25919, 25922, 25943, 26303, 26355,<br>27009, 27152, 28747, 29414, 29415,<br>32613, 32616, 32961, 32972, 33018 |
| scan internal commands:                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \g__scan_marks_tl . . . . .                                           | 378, 3558, 3561, 3567, 3572, 3574                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scantextokens . . . . .                                              | 897                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scantokens . . . . .                                                 | 565                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scriptbaselineshiftfactor . . . . .                                  | 1154                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scriptfont . . . . .                                                 | 433                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scriptscriptbaselineshiftfactor . . . . .                            | 1156                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scriptscriptfont . . . . .                                           | 434                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scriptscriptstyle . . . . .                                          | 435                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scriptsize . . . . .                                                 | 31662                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scriptspace . . . . .                                                | 436                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scriptstyle . . . . .                                                | 437                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scrollmode . . . . .                                                 | 438                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \scshape . . . . .                                                    | 31651                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| sec . . . . .                                                         | 215                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| secd . . . . .                                                        | 215                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \selectfont . . . . .                                                 | 31623                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| seq commands:                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \c_empty_seq . . . . .                                                | 86, 484, 7532,<br>7536, 7540, 7543, 7731, 7809, 7817                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \l_foo_seq . . . . .                                                  | 231                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_clear:N . . . . .                                                | 75,                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 75, 86, 7539, 7546, 7675, 12172,<br>12235, 14017, 14110, 25300, 26116 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_clear_new:N . . . . .                                            | 75, 7545                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_concat:NNN . . . . .                                             | 76, 86, 7628, 14023                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_const_from_clist:Nn . . . . .                                    | 76, 7585                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_count:N . . . . .                                                | 77, 83, 85, 198, 7746,<br>7936, 7950, 8081, 8109, 14135, 26115                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_elt:w . . . . .                                                  | 483                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_elt_end: . . . . .                                               | 483                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_gclear:N . . . . .                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 75, 950, 7539, 7549, 7763, 23100                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_gclear_new:N . . . . .                                           | 75, 7545                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_gconcat:NNN . . . . .                                            | 76, 7628, 14036                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_get>NN . . . . .                                                 | 84, 8162, 25591, 25596                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_get:NNTF . . . . .                                               | 84, 8168                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_get_left>NN . . . . .                                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 77, 7825, 8162, 8163, 8168, 8169                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| \seq_get_left:NNTF . . . . .                                          | 78, 7895                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

```

\seq_get_right:NN 77, 7850
\seq_get_right:NNTF 78, 7895
\seq_gpop:NN .. 84, 8162, 13946, 22746
\seq_gpop:NNTF
. 85, 8168, 12767, 12965, 22717, 22729
\seq_gpop_left:NN
. 77, 7836, 8166, 8167, 8172, 8173
\seq_gpop_left:NNTF 78, 7903
\seq_gpop_right:NN 77, 7868
\seq_gpop_right:NNTF 79, 7903
\seq_gpush:Nn .. 25, 85, 8142, 12792,
12989, 13931, 22721, 22731, 22740
\seq_gput_left:Nn
. 76, 7638, 8152, 8153, 8154, 8155,
8156, 8157, 8158, 8159, 8160, 8161
\seq_gput_right:Nn
. 76, 7659, 13374, 13381, 13920
\seq_gremove_all:Nn 79, 7685
\seq_gremove_duplicates:N .. 79, 7669
\seq_greverse:N 79, 7711
\seq_gset_eq:NN
. 75, 7543, 7551, 7672, 7743, 23074
\seq_gset_filter:NNn 270, 32274
\seq_gset_from_clist:NN .. 75, 7559
\seq_gset_from_clist:Nn .. 75, 7559
\seq_gset_from_function:NNN
. 270, 32294
\seq_gset_from_inline_x:Nnn
. 270, 7758, 23092, 32284, 32297
\seq_gset_map:NNn 82, 8071
\seq_gset_map_x:NNn 83, 8061
\seq_gset_split:NNn 76, 7591
\seq_gshuffle:N 80, 7739
\seq_gsort:Nn 79, 7729, 23070
\seq_if_empty:NTF
. 80, 7729, 7949, 9893, 22780, 26112
\seq_if_empty_p:N 80, 7729
\seq_if_exist:NTF
. 76, 7546, 7549, 7634, 8107
\seq_if_exist_p:N 76, 7634
\seq_if_in:Nn
. 560
\seq_if_in:NnTF
. 80, 85, 86, 7678, 7786, 12791, 12988
\seq_indexed_map_function:NN .. 32938
\seq_indexed_map_inline:Nn .. 32938
\seq_item:Nn
. 77, 234,
495, 7923, 7950, 12253, 12254, 12259
\seq_log:N 87, 8174
\seq_map_break:
. 82, 82, 83, 270, 7953, 7964,
7999, 8007, 8024, 8031, 8040, 15642
\seq_map_break:n
. 82, 495, 7953, 12192,
12206, 13559, 13656, 23071, 23074
\seq_map_function:NN
. 4, 80, 81, 268, 497, 7957,
8184, 9899, 12257, 14026, 25314, 25393
\seq_map_indexed_function:NN
. 81, 8028, 32940, 32941
\seq_map_indexed_inline:Nn
. 81, 8028, 32938, 32939
\seq_map_inline:Nn
. 80, 80, 81, 86, 1194, 7676, 7995,
12187, 13655, 15635, 23071, 23074
\seq_map_tokens:Nn
. 80, 81, 8002, 13558, 14139
\seq_map_variable:NNn 81, 8016
\seq_mapthread_function:NNN
. 270, 32252
\seq_new:N
. 4, 75, 75, 7533, 7546, 7549, 7668,
7741, 8188, 8189, 8190, 8191, 10044,
10507, 10510, 12142, 12143, 12719,
12933, 13366, 13391, 13397, 13398,
14931, 22571, 22572, 22573, 22942,
23839, 24280, 25411, 25412, 26018
\seq_pop:NN
. 84, 8162, 25457, 25459, 26152
\seq_pop:NNTF 85, 8168
\seq_pop_left:NN
. 77, 7836, 8164, 8165, 8170, 8171
\seq_pop_left:NNTF 78, 7903
\seq_pop_right:NN
. 77, 7868, 25251, 25321
\seq_pop_right:NNTF 79, 7903
\seq_push:Nn
. 85, 8142, 8149, 25450, 25452, 26280
\seq_put_left:Nn
. 76,
7638, 8142, 8143, 8144, 8145, 8146,
8147, 8148, 8149, 8150, 8151, 12182
\seq_put_right:Nn
. 76, 85, 86, 7659, 7679,
12243, 14112, 25254, 25319, 32772
\seq_rand_item:N 78, 7947
\seq_remove_all:Nn
. 76, 79, 85, 86, 7685, 10070, 32774
\seq_remove_duplicates:N
. 79, 85, 86, 7669, 14024
\seq_reverse:N 79, 489, 7711
\seq_set_eq:NN
. 75, 86, 7540, 7551, 7670, 7742, 23071
\seq_set_filter:NNn
. 270, 498, 25388, 32274
\seq_set_from_clist:NN .. 75, 7559, 10069
\seq_set_from_clist:Nn 75,
119, 485, 7559, 14020, 14034, 15558
\seq_set_from_function:NNN
. 270, 26540, 32294

```

```

\seq_set_from_inline_x:Nnn
..... 270, 1194, 32284, 32295
\seq_set_map:NNn 82, 8071
\seq_set_map_x:NNn
..... 83, 498, 8061, 25401, 26558
\seq_set_split:Nnn
. 76, 7591, 10508, 10511, 25387, 25400
\seq_show:N 87, 616, 8174
\seq_shuffle:N 80, 7739
\seq_sort:Nn 79, 224, 7729, 23070
\seq_use:Nn 84, 8105, 25404
\seq_use:Nnnn 83, 8105
\g_tmpa_seq 87, 8188
\l_tmpa_seq 87, 8188
\g_tmpb_seq 87, 8188
\l_tmpb_seq 87, 8188
seq internal commands:
__seq_count:w 499, 8081
__seq_count_end:w 499, 8081
__seq_get_left:wnw 7825
__seq_get_right_end:NnN 7850
__seq_get_right_loop:nw .. 493, 7850
__seq_if_in: 7786
\l__seq_internal_a_int
..... 7753, 7759, 7768, 7770, 7771
\l__seq_internal_a_t1
..... 485, 7529, 7599, 7603, 7609,
7614, 7616, 7700, 7705, 7790, 7794
\l__seq_internal_b_int
..... 7769, 7772, 7773
\l__seq_internal_b_t1
..... 7529, 7696, 7700, 7793, 7794
\g__seq_internal_seq 7739
__seq_item:n
..... 483, 483, 483, 483, 487, 491,
492, 493, 495, 496, 496, 497, 499,
499, 1193, 1194, 1194, 7524, 7642,
7650, 7660, 7662, 7667, 7717, 7718,
7720, 7725, 7755, 7791, 7830, 7833,
7843, 7858, 7861, 7874, 7875, 7886,
7930, 7939, 7963, 7966, 7976, 7981,
7987, 7991, 8006, 8010, 8051, 8053,
8067, 8077, 8088, 8089, 8090, 8091,
8092, 8093, 8094, 8095, 8100, 8101,
8116, 8131, 8134, 8137, 32290, 32291
__seq_item:nN 7923
__seq_item:nw 7923
__seq_item:wNn 7923
__seq_map_function>NNn 7957
__seq_map_function:Nw
..... 7960, 7966, 7970
__seq_map_indexed>NN 8030, 8038, 8043
__seq_map_indexed:nNN 8028
__seq_map_indexed:Nw 497, 8028
__seq_map_tokens:nw 8002
__seq_mapthread_function:Nnnwnn
..... 32252
__seq_mapthread_function:wNN . 32252
__seq_mapthread_function:wNw . 32252
__seq_pop:NNNN
..... 7807, 7837, 7839, 7869, 7871
__seq_pop_item_def:
..... 483, 483, 7707, 7757, 7973,
7999, 8024, 8069, 8079, 32282, 32292
__seq_pop_left:NNN . 7836, 7905, 7908
__seq_pop_left:wnwNNN 7836
__seq_pop_right:NNN
..... 488, 7868, 7911, 7914
__seq_pop_right_loop:nn 7868
__seq_pop_TF>NNNN 494, 7807,
7896, 7898, 7905, 7908, 7911, 7914
__seq_push_item_def: ... 7754, 7973
__seq_push_item_def:n
..... 483, 483, 7691, 7973,
7997, 8018, 8067, 8077, 32280, 32290
__seq_put_left_aux:w 487, 7638
__seq_remove_all_aux:NnN 7685
__seq_remove_duplicates>NN .. 7669
\l__seq_remove_seq
..... 7668, 7675, 7678, 7679, 7681
__seq_reverse>NN 7711
__seq_reverse_item:nw 489
__seq_reverse_item:nwn 7711
__seq_set_filter>NNNN 32274
__seq_set_from_inline_x>NNnn . 32284
__seq_set_map>NNNn 8071
__seq_set_map_x>NNNn 8061
__seq_set_split>NNnn 7591
__seq_set_split_auxi:w ... 485, 7591
__seq_set_split_auxii:w ... 485, 7591
__seq_set_split_end: ... 485, 7591
__seq_show>NN 8174
__seq_shuffle>NN 7739
__seq_shuffle_item:n 7739
__seq_tmp:w
..... 7531, 7717, 7720, 7874, 7886
__seq_use>NNnNnn 8105
__seq_use:nwwn 8105
__seq_use:nwwwnwn 8105
__seq_use_setup:w 8105
__seq_wrap_item:n
..... 485, 1194, 7562, 7567, 7572, 7577,
7588, 7600, 7625, 7667, 7703, 32280
\setbox 439
\setfontid 898
\setlanguage 440
\setrandomseed 940
\sfcode 441

```

\sffamily ..... 28725, 31646  
 \shapemode ..... 899  
 \shellescape ..... 786  
 \Shipout ..... 1219  
 \shipout ..... 442, 1206, 1207  
 \ShortText ..... 53, 100  
 \show ..... 443  
 \showbox ..... 444  
 \showboxbreadth ..... 445  
 \showboxdepth ..... 446  
 \showgroups ..... 566  
 \showifs ..... 567  
 \showlists ..... 447  
 \showmode ..... 1158  
 \showthe ..... 448  
 \ShowTokens ..... 225  
 \showtokens ..... 568  
 sign ..... 214  
 sin ..... 215  
 sind ..... 215  
 \sjis ..... 1159  
 \skewchar ..... 449  
 \skip ..... 450, 11075  
 skip commands:  
   \c\_max\_skip ..... 181, 14641  
   \skip\_add:Nn ..... 179, 14591  
   \skip\_const:Nn ..... 179, 686, 14561, 14641, 14642  
   \skip\_eval:n ..... 180, 180, 180,  
     180, 14564, 14605, 14620, 14636, 14640  
   \skip\_gadd:Nn ..... 179, 14591  
   \skip\_gset:N ..... 190, 15391  
   \skip\_gset:Nn ..... 179, 683, 14581  
   \skip\_gset\_eq:NN ..... 179, 14587  
   \skip\_gsub:Nn ..... 179, 14591  
   \skip\_gzero:N ..... 179, 14567, 14574  
   \skip\_gzero\_new:N ..... 179, 14571  
   \skip\_horizontal:N ..... 181, 14625  
   \skip\_horizontal:n ..... 181, 14625  
   \skip\_if\_eq:nnTF ..... 180, 14603  
   \skip\_if\_eq\_p:nn ..... 180, 14603  
   \skip\_if\_exist:NTF ..... 179, 14572, 14574, 14577  
   \skip\_if\_exist\_p:N ..... 179, 14577  
   \skip\_if\_finite:nTF ..... 180, 14609  
   \skip\_if\_finite\_p:n ..... 180, 14609  
   \skip\_log:N ..... 181, 14637  
   \skip\_log:n ..... 181, 14637  
   \skip\_new:N ..... 178, 179, 14555, 14563, 14572,  
     14574, 14643, 14644, 14645, 14646  
   \skip\_set:N ..... 190, 15391  
   \skip\_set:Nn ..... 179, 14581  
   \skip\_set\_eq:NN ..... 179, 14587  
   \skip\_show:N ..... 180, 14633  
   \skip\_show:n ..... 180, 685, 14635  
   \skip\_sub:Nn ..... 179, 14591  
   \skip\_use:N ..... 180, 180, 14614, 14621, 14622  
   \skip\_vertical:N ..... 182, 14625  
   \skip\_vertical:n ..... 182, 14625  
   \skip\_zero:N ..... 179, 179, 182, 14567, 14572  
   \skip\_zero\_new:N ..... 179, 14571  
   \g\_tmpa\_skip ..... 181, 14643  
   \l\_tmpa\_skip ..... 181, 14643  
   \g\_tmpb\_skip ..... 181, 14643  
   \l\_tmpb\_skip ..... 181, 14643  
   \c\_zero\_skip ..... 181,  
     673, 14223, 14225, 14567, 14568, 14641  
 skip internal commands:  
   \\_\_skip\_if\_finite:wwNw ..... 14609  
   \\_\_skip\_tmp:w ..... 14609, 14619  
 \skipdef ..... 451  
 \slshape ..... 31652  
 \small ..... 31663  
 sort commands:  
   \sort\_ordered: ..... 32839  
   \sort\_return\_same: .....  
     224, 224, 952, 23153, 32840  
   \sort\_return\_swapped: .....  
     224, 224, 952, 23153, 32842  
   \sort\_reversed: ..... 32841  
 sort internal commands:  
   \\_\_sort:nnNnn ..... 954, 955  
   \l\_\_sort\_A\_int .....  
     952, 22952, 22959, 22966, 22969,  
     22978, 23117, 23122, 23125, 23145,  
     23177, 23184, 23199, 23201, 23202  
   \l\_\_sort\_B\_int .....  
     952, 952, 22952, 23122, 23126,  
     23134, 23136, 23137, 23189, 23190,  
     23199, 23200, 23209, 23210, 23212  
   \l\_\_sort\_begin\_int .....  
     946, 952, 22950, 23114, 23202, 23212  
   \l\_\_sort\_block\_int .....  
     946, 947, 951, 22949, 22961,  
     22966, 22970, 22973, 22978, 22979,  
     23044, 23105, 23108, 23115, 23118  
   \l\_\_sort\_C\_int .....  
     952, 952, 22952, 23123, 23127,  
     23134, 23135, 23146, 23178, 23185,  
     23189, 23191, 23192, 23209, 23211  
   \\_\_sort\_compare:nn .....  
     949, 953, 23043, 23144  
   \\_\_sort\_compute\_range: .....  
     946, 947, 948, 22983, 23031  
   \\_\_sort\_copy\_block: ..... 951, 23124, 23132  
   \\_\_sort\_disable\_toksdef: ..... 23030, 23309

\\_\\_sort\\_disabled\\_toksdef:n . . . . .  
 \l\\_\\_sort\\_end\\_int 946, 951, 952, 952,  
     22950, 23106, 23114, 23115, 23116,  
     23117, 23118, 23119, 23120, 23137  
 \\_\\_sort\\_error: . . . . .  
 \\_\\_sort\\_i:nnnnNn . . . . . 956  
 \g\\_\\_sort\\_internal\\_seq . . . . .  
     949, 950, 22942, 23092, 23099, 23100  
 \g\\_\\_sort\\_internal\\_tl . . . . .  
     22942, 23055, 23058, 23059  
 \l\\_\\_sort\\_length\\_int . . . . .  
     946, 947, 22944, 23041, 23105  
 \\_\\_sort\\_level: . . . . .  
     949, 959, 23045, 23103, 23307  
 \\_\\_sort\\_loop:wNn . . . . . 955, 956  
 \\_\\_sort\\_main:NNNn . . . . .  
     949, 950, 23028, 23054, 23091  
 \l\\_\\_sort\\_max\\_int . . . . .  
     946, 947, 22944, 22963, 23035  
 \c\\_\\_sort\\_max\\_length\\_int . . . . . 22983  
 \\_\\_sort\\_merge\\_blocks: . . . . .  
     23107, 23112, 23306  
 \\_\\_sort\\_merge\\_blocks\\_aux: . . . . .  
     951, 23128, 23142, 23195, 23205, 23305  
 \\_\\_sort\\_merge\\_blocks\\_end: . . . . .  
     953, 23203, 23207  
 \l\\_\\_sort\\_min\\_int . . . . . 946, 947, 949,  
     949, 22944, 22960, 22968, 22985,  
     23001, 23009, 23022, 23032, 23042,  
     23056, 23095, 23106, 23331, 23332  
 \\_\\_sort\\_quick\\_cleanup:w . . . . . 23217  
 \\_\\_sort\\_quick\\_end:nnTFNn . . . . .  
     957, 958, 23237, 23277  
 \\_\\_sort\\_quick\\_only\\_i:NnnnnNn . 23242  
 \\_\\_sort\\_quick\\_only\\_i\\_end:nnwnw .  
     23253, 23277  
 \\_\\_sort\\_quick\\_only\\_ii:NnnnnNn . 23242  
 \\_\\_sort\\_quick\\_only\\_ii\\_end:nnwnw .  
     23260, 23277  
 \\_\\_sort\\_quick\\_prepare:Nnnn . . . . . 23217  
 \\_\\_sort\\_quick\\_prepare\\_end:NNNnw .  
     23217  
 \\_\\_sort\\_quick\\_single\\_end:nnwnw .  
     23246, 23277  
 \\_\\_sort\\_quick\\_split:NnNn . . . . .  
     956, 956, 23237,  
     23242, 23282, 23289, 23295, 23297  
 \\_\\_sort\\_quick\\_split\\_end:nnwnw .  
     23267, 23274, 23277  
 \\_\\_sort\\_quick\\_split\\_i:NnnnnNn . .  
     955, 23242  
 \\_\\_sort\\_quick\\_split\\_ii:NnnnnNn 23242  
 \\_\\_sort\\_redefine\\_compute\\_range: .  
     22983  
 \\_\\_sort\\_return\\_mark:w . . . . .  
     952, 23148, 23149, 23153  
 \\_\\_sort\\_return\\_none\\_error: . . . . .  
     952, 23151, 23153, 23187, 23197  
 \\_\\_sort\\_return\\_same:w . . . . .  
     952, 23161, 23179, 23187  
 \\_\\_sort\\_return\\_swapped:w 23171, 23197  
 \\_\\_sort\\_return\\_two\\_error: 952, 23153  
 \\_\\_sort\\_seq:NNNNn . . . . . 949, 23070  
 \\_\\_sort\\_shrink\\_range: . . . . . 947,  
     948, 22957, 22987, 23003, 23011, 23024  
 \\_\\_sort\\_shrink\\_range\\_loop: . . . . . 22957  
 \\_\\_sort\\_tl:NNn . . . . . 949, 23047  
 \\_\\_sort\\_tl\\_toks:w . . . . . 949, 23047  
 \\_\\_sort\\_too\\_long\\_error:NNw . . . . .  
     23036, 23326  
 \l\\_\\_sort\\_top\\_int 946, 949, 949, 952,  
     952, 22944, 23032, 23035, 23038,  
     23039, 23042, 23064, 23095, 23116,  
     23119, 23120, 23123, 23192, 23332  
 \\_\\_sort\\_true\\_max\\_int . . . . .  
     946, 947, 22944, 22960, 22973,  
     22986, 23002, 23010, 23023, 23331  
 sp . . . . . 218  
 spac commands:  
     \spac\\_directions\\_normal\\_body\\_dir  
         . . . . . 1362  
     \spac\\_directions\\_normal\\_page\\_dir  
         . . . . . 1363  
 \spacefactor . . . . . 452  
 \spaceskip . . . . . 453  
 \span . . . . . 454  
 \special . . . . . 455  
 \splitbotmark . . . . . 456  
 \splitbotmarks . . . . . 569  
 \splitediscards . . . . . 570  
 \splitfirstmark . . . . . 457  
 \splitfirstmarks . . . . . 571  
 \splitmaxdepth . . . . . 458  
 \splittopskip . . . . . 459  
 sqrt . . . . . 216  
 \SS . . . . . 29556, 31388, 31746  
 \ss . . . . . 29556, 31388, 31742  
 str commands:  
     \c\\_ampersand\\_str . . . . . 71, 5316  
     \c\\_at\\_sign\\_str . . . . . 71, 5316  
     \c\\_backslash\\_str . . . . .  
         . 71, 5316, 6016, 6018, 6041, 6070,  
         6072, 6104, 6113, 6117, 24092, 24664  
     \c\\_circumflex\\_str . . . . . 71, 5316  
     \c\\_colon\\_str . . . . .  
         . 71, 5316, 10986, 11162, 11168, 15029  
     \c\\_dollar\\_str . . . . . 71, 5316  
     \l\\_foo\\_str . . . . . 72

\c\_hash\_str .....  
     . 71, 5316, 5984, 6087, 6660, 6661,  
     6664, 6667, 29337, 29368, 29369, 29373  
\c\_percent\_str .. 71, 5316, 5986, 6140  
\str\_byte ..... 5365  
\str\_case:nn .....  
     . 64, 4874, 12609, 13809, 24813, 32301  
\str\_case:nnn ..... 32843, 32845  
\str\_case:nnTF ..... 64, 676,  
     4874, 4879, 4884, 9485, 9520, 9788,  
     12546, 15204, 25377, 32844, 32846  
\str\_case\_e:nn ..... 64, 4874, 32848  
\str\_case\_e:nnTF .....  
     . 64, 2667, 4874, 4910, 4915, 6039,  
     24545, 32850, 32852, 32854, 32856  
\str\_case\_x:nn ..... 32847  
\str\_case\_x:nnn ..... 32849  
\str\_case\_x:nnTF .. 32851, 32853, 32855  
\str\_clear:N ..... 61,  
     61, 4726, 15011, 15164, 15580, 15581  
\str\_clear\_new:N ..... 61, 4726  
\str\_concat:NNN ..... 61, 4726  
\str\_const:Nn ..... 60,  
     4753, 5316, 5317, 5318, 5319, 5320,  
     5321, 5322, 5323, 5324, 5325, 5326,  
     5327, 6080, 6081, 6103, 9393, 9412,  
     9430, 9476, 9754, 14181, 14188,  
     14192, 14196, 14908, 14909, 14910,  
     14911, 14912, 14913, 14914, 32299  
\str\_convert\_pdfname:n .... 74, 6636  
\str\_count:N ..... 66,  
     5206, 11876, 11877, 12056, 12057,  
     12078, 12079, 13048, 13126, 23776  
\str\_count:n ..... 66, 5206, 23770  
\str\_count\_ignore\_spaces:n .....  
     . 66, 426, 5206, 23391  
\str\_count\_spaces:N ..... 66, 5186  
\str\_count\_spaces:n 66, 426, 5186, 5212  
\str\_declare\_eight\_bit\_encoding:nnn  
     ..... 32935  
\str\_end ..... 6539  
\str\_error ..... 5365  
\str\_fold\_case:n ..... 32923  
\str\_foldcase:n .....  
     . 69, 70, 130, 262, 5274,  
     17283, 32931, 32932, 32933, 32934  
\str\_gclear:N ..... 61, 4726  
\str\_gclear\_new:N ..... 4726  
\str\_gconcat:NNN ..... 61, 4726  
\str\_gput\_left:Nn ..... 61, 4753  
\str\_gput\_right:Nn ..... 62, 4753  
\str\_gremove\_all:Nn ..... 62, 4823  
\str\_gremove\_once:Nn ..... 62, 4817  
\str\_greplace\_all:Nnn 62, 4777, 4826  
\str\_greplace\_once:Nnn 62, 4777, 4820  
\str\_gset:Nn .....  
     . 61, 4753, 13952, 13953, 13954  
\str\_gset\_convert:Nnnn ..... 72, 5504  
\str\_gset\_convert:NnnnTF ..... 74, 5504  
\str\_gset\_eq:NN .....  
     . 61, 4726, 13939, 13940, 13941  
\str\_head:N ..... 67, 427, 5244  
\str\_head:n .....  
     . 67, 403, 427, 4346, 4391, 5244  
\str\_head\_ignore\_spaces:n .. 67, 5244  
\str\_if\_empty:NTF .....  
     . 63, 4829, 13672, 14967, 14990, 15805  
\str\_if\_empty\_p:N ..... 63, 4829  
\str\_if\_eq:NN ..... 417  
\str\_if\_eq:nn ..... 144, 595, 603  
\str\_if\_eq:NNTF ..... 63, 4852  
\str\_if\_eq:nntf ..... 47, 47,  
     47, 63, 64, 147, 148, 488, 584,  
     2053, 2688, 4048, 4838, 4901, 4929,  
     6421, 6424, 6579, 6582, 7693, 9448,  
     9469, 9500, 9664, 10989, 11046,  
     11492, 11565, 11660, 12203, 12246,  
     13995, 14058, 14073, 14605, 15072,  
     15639, 17153, 17226, 24140, 24162,  
     24171, 26839, 26969, 29348, 29367,  
     29371, 29764, 29798, 30080, 30113,  
     31595, 32689, 32860, 32862, 32864  
\str\_if\_eq\_p:NN ..... 63, 4852  
\str\_if\_eq\_p:nn .....  
     . 63, 4838, 9408, 9435, 9436, 9508,  
     9509, 9762, 9764, 14200, 29803, 32858  
\str\_if\_eq\_x:nnTF 32859, 32861, 32863  
\str\_if\_eq\_x\_p:nn ..... 32857  
\str\_if\_exist:NTF ... 63, 4829, 9467  
\str\_if\_exist\_p:N ..... 63, 4829  
\str\_if\_in:NNTF ..... 63, 4860  
\str\_if\_in:nntf 63, 3219, 4860, 22824  
\str\_item:Nn ..... 67, 5048  
\str\_item:nn ..... 67, 422, 426, 5048  
\str\_item\_ignore\_spaces:nn .....  
     . 67, 422, 5048  
\str\_log:N ..... 70, 5332  
\str\_log:n ..... 70, 5332  
\str\_lower\_case:n ..... 32923  
\str\_lowercase:n ..... 69,  
     262, 5274, 32923, 32924, 32925, 32926  
\str\_map\_break: ..... 65, 4935  
\str\_map\_break:n .. 65, 66, 3223, 4935  
\str\_map\_function>NN .....  
     . 64, 64, 65, 65, 4935  
\str\_map\_function:nN .....  
     . 64, 64, 419, 4935, 6639  
\str\_map\_inline:Nn .. 65, 65, 65, 4935

```

\str_map_inline:nn 65, 3217, 4935, 25801
\str_map_variable:NnN 65, 4935
\str_map_variable:nNn 65, 4935
\str_new:N 60, 61, 4726, 5328, 5329,
5330, 5331, 11753, 11754, 13363,
13364, 13365, 13394, 13395, 13396,
14918, 14920, 14923, 14925, 14928
str_overflow 6539
\str_put_left:Nn 61, 4753
\str_put_right:Nn 62, 4753
\str_range:Nnn 68, 5109
\str_range:nmm 68, 167, 426, 5109, 23773
\str_range_ignore_spaces:nnn 68, 5109
\str_remove_all:Nn 62, 62, 4823
\str_remove_once:Nn 62, 4817
\str_replace_all:Nnn 62, 4777, 4824
\str_replace_once:Nnn 62, 4777, 4818
\str_set:Nn ... 61, 62, 4753, 4994,
11839, 11840, 12044, 12045, 12066,
12067, 14008, 14009, 14010, 14946,
14948, 15446, 15448, 15589, 15714
\str_set_convert:Nnnn 72, 74, 74, 434, 445, 5504
\str_set_convert:NnnnTF 74, 434, 5504
\str_set_eq:NN 61, 4726
\str_show:N 70, 5332
\str_show:n 70, 5332
\str_tail:N 67, 5259
\str_tail:n 67, 965, 5259, 29450
\str_tail_ignore_spaces:n .. 67, 5259
\str_upper_case:n 32923
\str_uppercase:n 69,
262, 5274, 32927, 32928, 32929, 32930
\str_use:N 66, 4726
\c_tilde_str 71, 5316
\g_tmpa_str 71, 5328
\l_tmpa_str 62, 71, 5328
\g_tmpb_str 71, 5328
\l_tmpb_str 71, 5328
\c_underscore_str 71, 5316
str internal commands:
\g_str_alias_prop .. 437, 5348, 5575
\c_str_byte_-1_tl 5418
\c_str_byte_0_tl 5418
\c_str_byte_1_tl 5418
\c_str_byte_255_tl 5418
\c_str_byte_<number>_tl 433
__str_case:nnTF 4874
__str_case:nw 4874
__str_case_e:nnTF 4874
__str_case_e:nw 4874
__str_case_end:nw 4874
__str_change_case:nn 5274
__str_change_case_char:nN 5274
__str_change_case_end:nw 5274
__str_change_case_end:wn 5293, 5311
__str_change_case_loop:nw ... 5274
__str_change_case_output:nw . 5274
__str_change_case_result:n .. 5274
__str_change_case_space:n ... 5274
__str_collect_delimit_by_q_-
stop:w 5137, 5160
__str_collect_end:nnnnnnnnw ...
..... 424, 5160
__str_collect_end:wn 5160
__str_collect_loop:wn 5160
__str_collect_loop:wnNNNNNNN . 5160
__str_convert:nnn 436, 437, 5547, 5548, 5562
__str_convert:nnnn 437, 5562
__str_convert:NNnNN 5544
__str_convert:nNNnnn 5504
__str_convert:wwnn 436, 5531, 5536, 5544
__str_convert_decode: .. 5535, 5685
__str_convert_decode_clist: . 5725
__str_convert_decode_eight_-
bit:n 5746, 5790
__str_convert_decode_utf16: . 6410
__str_convert_decode_utf16be: 6410
__str_convert_decode_utf16le: 6410
__str_convert_decode_utf32: . 6568
__str_convert_decode_utf32be: 6568
__str_convert_decode_utf32le: 6568
__str_convert_decode_utf8: .. 6229
__str_convert_encode: .. 5540, 5689
__str_convert_encode_clist: . 5736
__str_convert_encode_eight_-
bit:n 5748, 5817
__str_convert_encode_utf16: . 6325
__str_convert_encode_utf16be: 6325
__str_convert_encode_utf16le: 6325
__str_convert_encode_utf32: . 6508
__str_convert_encode_utf32be: 6508
__str_convert_encode_utf32le: 6508
__str_convert_encode_utf8: .. 6156
__str_convert_escape: 5683
__str_convert_escape_bytes: . 5683
__str_convert_escape_hex: ... 6076
__str_convert_escape_name: 452, 6080
__str_convert_escape_string: . 6103
__str_convert_escape_url: ... 6135
__str_convert_gmap:N 5462,
5686, 5798, 6077, 6083, 6106, 6136

```

```

__str_convert_gmap_internal:N .. 460, 6410
 5478, 5696, 5704, 5738,
 5827, 6157, 6338, 6510, 6514, 6516
__str_convert_gmap_internal_-_
 loop:Nw 5478
__str_convert_gmap_internal_-_
 loop:Nww 5482, 5488, 5492
__str_convert_gmap_loop>NN .. 5462
__str_convert_lowercase_-
 alphanum:n 5567, 5599
__str_convert_lowercase_-
 alphanum:N 5599
__str_convert_pdfname:n 6636
__str_convert_pdfname_bytes:n 6636
__str_convert_pdfname_bytes_-
 aux:n 6636
__str_convert_pdfname_bytes_-
 aux:nnn 6636
__str_convert_pdfname_bytes_-
 aux:nnnn 6657, 6658
__str_convert_unescape:_ 5667
__str_convert_unescape_bytes: 5667
__str_convert_unescape_hex: . 5892
__str_convert_unescape_name: ...
 447, 5938
__str_convert_unescape_string: 5988
__str_convert_unescape_url: . 5938
__str_count:n . 426, 5064, 5124, 5206
__str_count_aux:n 5206
__str_count_loop>NNNNNNNN .. 5206
__str_count_spaces_loop:w ... 5186
__str_declare_eight_bit_-
 aux:NNnnn 5742
__str_declare_eight_bit_-
 encoding:nnnn 442,
 5742, 6675, 6682, 6746, 6788, 6845,
 6946, 7033, 7119, 7193, 7206,
 7259, 7357, 7420, 7458, 7473, 32937
__str_declare_eight_bit_loop:Nn
 5742
__str_declare_eight_bit_-
 loop:Nnn 5742
__str_decode_clist_char:n ... 5725
__str_decode_eight_bit_aux:n . 5790
__str_decode_eight_bit_aux:Nn 5790
__str_decode_native_char:N .. 5685
__str_decode_utf_viii_aux:wNnnwN
 6229
__str_decode_utf_viii_continuation:wwN
 6229
__str_decode_utf_viii_end: .. 6229
__str_decode_utf_viii_overflow:w
 6229
__str_decode_utf_viii_start:N 6229
__str_decode_utf_xvi:Nw .. 460, 6410
__str_decode_utf_xvi_bom>NN . 6410
__str_decode_utf_xvi_error:nNN 6444
__str_decode_utf_xvi_extra>NNw 6444
__str_decode_utf_xvi_pair>NN ...
 460, 461, 6438, 6444
__str_decode_utf_xvi_pair_-
 end:Nw 6444
__str_decode_utf_xvi_quad>NNwNNN
 6444
__str_decode_utf_xxxii:Nw 461, 6568
__str_decode_utf_xxxii_bom>NNNN
 6568
__str_decode_utf_xxxii_end:w . 6568
__str_decode_utf_xxxii_loop>NNNN
 6568
__str_encode_clist_char:n ... 5736
__str_encode_eight_bit_aux>NNn 5817
__str_encode_eight_bit_aux:nnN 5817
__str_encode_native_char:n .. 5689
__str_encode_utf_vii_loop:wwnw 453
__str_encode_utf_viii_char:n . 6156
__str_encode_utf_viii_loop:wwnw
 6156
__str_encode_utf_xvi_aux:N .. 6325
__str_encode_utf_xvi_be:nn ... 458
__str_encode_utf_xvi_char:n .. 6325
__str_encode_utf_xxxii_be:n . 6508
__str_encode_utf_xxxii_be_-
 aux:nn 6508
__str_encode_utf_xxxii_le:n . 6508
__str_encode_utf_xxxii_le_-
 aux:nn 6508
\l__str_end_flag 6359
\g__str_error_bool ...
 ... 5364, 5501, 5511, 5515, 5520, 5524
__str_escape_hex_char:N 6076
__str_escape_name_char:n
 6080, 6649, 6672
\c__str_escape_name_not_str 451, 6080
\c__str_escape_name_str ... 451, 6080
__str_escape_string_char:N .. 6103
\c__str_escape_string_str 6103
__str_escape_url_char:n 6135
\l__str_extra_flag 6178, 6359
__str_filter_bytes:n
 5643, 5677, 5958, 6020
__str_filter_bytes_aux:N 5643
__str_head:w 427, 5244
__str_hexadecimal_use:N 5399
__str_hexadecimal_use:NTF
 ... 447, 5399, 5912, 5922, 5961, 5963
__str_if_contains_char:Nn ... 5367
__str_if_contains_char:nn ... 5376

```

```

__str_if_contains_char:NnTF . . .
 5367, 6092, 6098, 6111
__str_if_contains_char:nnTF . . .
 431, 5367, 6145, 6151
__str_if_contains_char_aux:nn 5367
__str_if_contains_char_auxi:nN 5367
__str_if_contains_char_true: . 5367
__str_if_eq:nn 4837, 4841, 4849, 4855
__str_if_escape_name:n 6089
__str_if_escape_name:nTF 6080
__str_if_escape_string:N 6123
__str_if_escape_string:NTF .. 6103
__str_if_escape_url:n 6142
__str_if_escape_url:nTF 6135
__str_if_flag_error:nnm . . .
 434, 435, 5494, 5513, 5522,
 5705, 5799, 5828, 5906, 5952, 5953,
 6011, 6012, 6242, 6339, 6442, 6599
__str_if_flag_no_error:nnm . . .
 434, 5494, 5513, 5522
__str_if_flag_times:nTF . . .
 5502, 6187, 6188, 6189,
 6190, 6374, 6375, 6376, 6546, 6547
__str_if_recursion_tail_-_
 break:NN 4724, 4975, 4993
__str_if_recursion_tail_stop_-_
 do:Nn 4724, 5310
\l__str_internal_tl 437,
 5341, 5419, 5420, 5422, 5575, 5576,
 5577, 5579, 5583, 5587, 5594, 5744
__str_item:nn 422, 5048
__str_item:w 422, 5048
__str_load_catcodes: ... 5582, 5625
__str_map_function:Nn 419, 4935
__str_map_function:w 419, 4935
__str_map_inline:NN 4935
__str_map_variable:NnN 4935
\c__str_max_byte_int 5345, 5710
\l__str_missing_flag 6178, 6359
\l__str_modulo_int 5817
__str_octal_use:N 5391
__str_octal_use:NTF . . .
 431, 432, 5391, 6023, 6025, 6027
__str_output_byte:n . . .
 463, 5430, 5459, 5460,
 5620, 5843, 6170, 6176, 6526, 6535
__str_output_byte:w . . .
 447, 5430, 5899, 5925, 5960, 6022
__str_output_byte_pair:nnN .. 5446
__str_output_byte_pair_be:n . . .
 5446, 6327, 6331, 6525
__str_output_byte_pair_le:n . . .
 5446, 6333, 6536
__str_output_end: . . .
 447, 5430, 5904, 5924, 5974, 6056, 6060
__str_output_hexadecimal:n . . .
 5430, 6079,
 6087, 6140, 6660, 6661, 6664, 6667
\l__str_overflow_flag 6178
\l__str_overlong_flag 6178
__str_range:nnn 5109
__str_range:nnw 5109
__str_range:w 5109
__str_range_normalize:nn . . .
 5132, 5133, 5141
__str_replace>NNNnn 4777
__str_replace_aux>NNNnnn 4777
__str_replace_next:w 4777
\c__str_replacement_char_int ...
 5344, 5812, 6254, 6278, 6292, 6312,
 6319, 6349, 6501, 6610, 6615, 6631
\g__str_result_tl . . .
 430, 433, 434, 435, 440, 441, 447,
 460, 463, 464, 5343, 5464, 5468,
 5480, 5484, 5530, 5542, 5676, 5677,
 5727, 5728, 5731, 5739, 5897, 5901,
 5946, 5948, 5999, 6002, 6005, 6008,
 6236, 6238, 6328, 6411, 6413, 6417,
 6436, 6511, 6569, 6571, 6574, 6593
__str_skip_end>NNNNNNNN .. 423, 5088
__str_skip_end:w 5088
__str_skip_exp_end:w . . .
 423, 424, 5075, 5084, 5088, 5139
__str_skip_loop:wNNNNNNNN .. 5088
__str_tail_auxi:w 5259
__str_tail_auxii:w 427, 5259
__str_tmp:n . . .
 4727, 4733, 4736, 4754, 4764, 4767
__str_tmp:w 443, 447,
 458, 460, 464, 5341, 5792, 5798,
 5820, 5827, 5938, 5984, 5986, 5991,
 6016, 6337, 6344, 6349, 6351, 6354,
 6355, 6435, 6450, 6455, 6466, 6469,
 6475, 6476, 6592, 6607, 6612, 6618
__str_to_other_end:w 420, 5003
__str_to_other_fast_end:w ... 5026
__str_to_other_fast_loop:w . . .
 5028, 5037, 5044
__str_to_other_loop:w 420, 5003
__str_unescape_hex_auxi:N ... 5892
__str_unescape_hex_auxii:N .. 5892
__str_unescape_name_loop:wNN .. 5938
__str_unescape_string_loop:wNNN ...
 5988
__str_unescape_string_newlines:wN
 5988

```

```

__str_unescape_string_repeat:NNNNNN
 5988
__str_unescape_url_loop:wNN . 5938
__str_use_i_delimit_by_s_-
 stop:nw 427,
 4720, 5074, 5083, 5202, 5253, 5256
__str_use_none_delimit_by_s_-
 stop:w .. 4720, 4811, 5072, 5081,
 5240, 5490, 5780, 5786, 6171, 6261
\strcmp 22
\string 460
\suppressfontnotfounderror 717
\suppressifcsnameerror 900
\suppresslongerror 901
\suppressmathparerror 902
\suppressoutererror 903
\suppressprimitiveerror 904
\synctex 697
sys commands:
 \c_sys_backend_str 118, 9464
 \c_sys_day_int 115, 9659
 \c_sys_engine_exec_str
 116, 539, 9410, 14103
 \c_sys_engine_format_str
 116, 539, 9410, 14104
 \c_sys_engine_str
 115, 539, 669, 9393, 32301
 \c_sys_engine_version_str 271, 32299
 \sys_everyjob: 9649, 9745
 \sys_finalise: 118, 9466, 9743
 \sys_get_shell:nnN 117, 9550
 \sys_get_shell:nnN(TF) 267
 \sys_get_shell:nnNTF 117, 9550, 9552
 \sys_gset_rand_seed:n 117, 217, 9705
 \c_sys_hour_int 115, 9659
 \sys_if_engine_luatex:TF
 115, 256, 2674, 9393, 9418,
 9443, 9502, 9512, 9513, 9590, 9612,
 9639, 9724, 10575, 11078, 12781,
 13534, 13761, 13923, 14179, 22578,
 22616, 22655, 22668, 22694, 22762,
 22807, 28973, 32810, 32812, 32814
 \sys_if_engine_luatex_p: ... 115,
 5645, 5669, 5691, 5861, 6642, 9393,
 12928, 22857, 22883, 29250, 30133,
 30207, 30247, 30295, 30331, 30537,
 30585, 30592, 30670, 30748, 30826,
 30849, 30885, 31686, 31771, 32808
 \sys_if_engine_pdftex:TF ... 115,
 9393, 9414, 9438, 32824, 32826, 32828
 \sys_if_engine_pdftex_p:
 115, 9393, 9452, 32822
 \sys_if_engine_ptex:TF
 115, 9393, 9416, 9441
\sys_if_engine_ptex_p:
 115, 9393, 23409
\sys_if_engine_uptex:TF
 115, 9393, 9417, 9442
\sys_if_engine_uptex_p:
 115, 9393, 23410
\sys_if_engine_xetex:TF
 .. 5, 115, 2673, 9393, 9415, 9440,
 9483, 9771, 10576, 32882, 32884, 32886
\sys_if_engine_xetex_p: 115,
 5646, 5670, 5692, 5862, 6643, 9393,
 9670, 22857, 22883, 29250, 30134,
 30208, 30248, 30296, 30332, 30538,
 30586, 30593, 30671, 30749, 30827,
 30850, 30886, 31687, 31772, 32880
\sys_if_output_dvi:TF 116, 9752
\sys_if_output_dvi_p: 116, 9752
\sys_if_output_pdf:TF
 116, 9498, 9752, 9774
\sys_if_output_pdf_p: 116, 9752
\sys_if_platform_unix:TF
 116, 9464, 14197
\sys_if_platform_unix_p:
 116, 9464, 14197
\sys_if_platform_windows:TF
 116, 9464, 14197
\sys_if_platform_windows_p:
 116, 9464, 14197
\sys_if_rand_exist:TF .. 271, 540,
 9462, 9693, 9707, 16130, 22106, 22130
\sys_if_rand_exist_p: 271, 9462
\sys_if_shell: 118
\sys_if_shell:TF 117, 9557, 9732, 32355
\sys_if_shell_p: 117, 9732
\sys_if_shell_restricted:TF 118, 9732
\sys_if_shell_restricted_p: 118, 9732
\sys_if_shell_unrestricted:TF ...
 117, 9732
\sys_if_shell_unrestricted_p: ...
 117, 9732
\c_sys_jobname_str
 115, 165, 548, 9657, 32712
\sys_load_backend:n .. 118, 118, 9464
\sys_load_debug: 118, 9536
\sys_load_deprecation: 118, 9536
\c_sys_minute_int 115, 9659
\c_sys_month_int 115, 9659
\c_sys_output_str 116, 9752
\c_sys_platform_str
 116, 9464, 14179, 14200
\sys_rand_seed: ... 80, 116, 217, 9691
\c_sys_shell_escape_int
 117, 9720, 9735, 9737, 9739
\sys_shell_now:n 118, 9592

```

```

\sys_shell_shipout:n 118, 9623
\c_sys_year_int 115, 9659
sys internal commands:
 \g__sys_backend_tl 9474, 9475, 9476, 9766
 __sys_const:nn 9377, 9407,
 9462, 9734, 9736, 9738, 9761, 9763
 \g__sys_debug_bool .. 9534, 9538, 9540
 \g__sys_deprecation_bool 1210, 9534, 9544, 9546
 __sys_everyjob:n 9649, 9657,
 9659, 9691, 9705, 9720, 9732, 9741
 \g__sys_everyjob_tl 9649
 __sys_finalise:n 9743, 9752, 9767, 9780
 \g__sys_finalise_tl 9743
 __sys_get:nnN 9550
 __sys_get_do:Nw 9550
 \l__sys_internal_tl 9548
 __sys_load_backend_check:N .. 9464
 \c_sys_marker_tl ... 9549, 9573, 9585
 __sys_shell_now:n 9592
 __sys_shell_shipout:n 9623
 \c__sys_shell_stream_int 9590, 9619, 9646
 __sys_tmp:w 9662, 9683, 9685, 9686, 9687, 9688
syst commands:
 \c_syst_catcodes_n 22813, 22817
 \c_syst_last_allocated_toks .. 23016

T
\texttt{T} 29543, 31811
\tabskip 461
>tagcode 698
\tan 215
\tand 215
\tate 1160
\tbaselineshift 1161
\TeX 29051
TeX and LATEX 2< commands:
 \@ 5317
 \@@@hyph 299
 \@@end 1192, 1193
 \@@hyph 1196, 1199
 \@@input 1194
 \@@italiccorr 1200
 \@@shipout 1202, 1203
 \@@tracingfonts 300, 1238
 \@@underline 1201
 \@addtofilelist 13919
 \@changed@cmd 29830
 \@classoptionslist .. 9782, 9784, 9786
 \@current@cmd 29829
 \@currnamestack 651, 13385, 13387, 13388
 \@expl@finalise@setup@@ 22848, 22849
 \@filelist 169, 651, 664, 667,
 667, 13918, 14018, 14021, 14030, 14035
 \@firstofone 19
 \@firstoftwo 19, 316
 \@gobbletwo 20
 \@gobble 20
 \@secondoftwo 19, 316
 \@tempa 108, 110, 1210, 1224, 1227
 \@tfor 299, 1210
 \@uclclist 1174, 31422
 \@unexpandable@protect 769
 \@unusedoptionlist 9801
 \AtBeginDocument 299
 \botmark 585
 \box 246
 \catcodetable 935, 939, 941
 \char 142
 \chardef 137, 137, 505, 528, 1132
 \color 1115
 \conditionally@traceoff 643, 12146, 13106
 \conditionally@traceon 12164
 \copy 239
 \count 142, 948
 \cr 538
 \CROP@shipout 1211
 \csname 17
 \csstring 325
 \currentgrouplevel ... 337, 938, 1191
 \currentgroupype 337, 1191
 \def 142
 \detokenize 51
 \development@branch@name 14106, 14107
 \dimen 584
 \dimendef 584
 \directlua 256
 \dp 240, 770, 771
 \dup@shipout 1212
 \e@alloc@ccodetable@count 22811
 \e@alloc@top 948, 23002
 \edef 1, 4, 379
 \end 299, 613
 \endcsname 17
 \endinput 154
 \endlinechar 46, 46,
 160, 384, 386, 585, 935, 936, 936, 938
 \endtemplate 114, 538
 \errhelp 609, 610
 \errmessage 609, 610, 610, 611
 \errorcontextlines 307, 412, 611, 1073
 \escapechar 51, 325, 337, 642, 977

```

|                                 |                                                              |
|---------------------------------|--------------------------------------------------------------|
| \everyeof .....                 | 386                                                          |
| \everyjob .....                 | 545                                                          |
| \everypar .....                 | 24, 339, 357                                                 |
| \expandafter .....              | 33, 35                                                       |
| \expanded .....                 | 4, 20, 28, 30,<br>341, 344, 350, 352, 357, 364, 385, 401     |
| \fi .....                       | 141                                                          |
| \firstmark .....                | 358, 585                                                     |
| \fmtname .....                  | 116                                                          |
| \font .....                     | 141, 583                                                     |
| \fontdimen .....                | 199, 237, 722, 724, 725, 725, 725                            |
| \frozen@everydisplay .....      | 1197                                                         |
| \frozen@everymath .....         | 1198                                                         |
| \futurelet .....                | 538, 589, 591, 961, 963, 965, 965                            |
| \global .....                   | 279                                                          |
| \GPTorg@shipout .....           | 1213                                                         |
| \halign .....                   | 114, 339, 538, 574                                           |
| \hskip .....                    | 181                                                          |
| \ht .....                       | 240, 770, 771                                                |
| \hyphen .....                   | 585                                                          |
| \hyphenchar .....               | 722                                                          |
| \ifcase .....                   | 101                                                          |
| \ifdim .....                    | 184                                                          |
| \ifeof .....                    | 165                                                          |
| \iffalse .....                  | 107                                                          |
| \ifhbox .....                   | 249                                                          |
| \ifnum .....                    | 101                                                          |
| \ifodd .....                    | 102, 593                                                     |
| \iftrue .....                   | 107                                                          |
| \ifvbox .....                   | 249                                                          |
| \ifvoid .....                   | 249                                                          |
| \ifx .....                      | 23, 277                                                      |
| \indent .....                   | 339                                                          |
| \infty .....                    | 210                                                          |
| \input .....                    | 299                                                          |
| \input@path .....               | 166, 656, 13560, 13562, 13657, 13659                         |
| \italiccorr .....               | 585                                                          |
| \jobname .....                  | 115, 545                                                     |
| \lastnamedcs .....              | 328                                                          |
| \lccode .....                   | 524, 964, 968, 1128                                          |
| \leavevmode .....               | 24                                                           |
| \let .....                      | 279                                                          |
| \letcharcode .....              | 570                                                          |
| \LL@shipout .....               | 1214                                                         |
| \loctoks .....                  | 948                                                          |
| \long .....                     | 3, 143, 352                                                  |
| \lower .....                    | 1188                                                         |
| \lowercase .....                | 1050, 1051, 1051                                             |
| \luaescapestring .....          | 256                                                          |
| \makeatletter .....             | 7                                                            |
| \MakeUppercase .....            | 1167                                                         |
| \mathchar .....                 | 142                                                          |
| \mathchardef .....              | 137, 505, 1132                                               |
| \meaning .....                  | 15,<br>134, 142, 143, 583, 584, 591, 593, 963                |
| \mem@oldshipout .....           | 1215                                                         |
| \message .....                  | 28                                                           |
| \newcatcodetable .....          | 935                                                          |
| \newif .....                    | 107, 264                                                     |
| \newlinechar .....              | 46,<br>46, 307, 328, 384, 386, 412, 611, 640                 |
| \newread .....                  | 632                                                          |
| \newtoks .....                  | 224, 959, 976                                                |
| \newwrite .....                 | 638                                                          |
| \noexpand .....                 | 34, 141, 351, 352, 352, 353, 354                             |
| \nullfont .....                 | 585, 586                                                     |
| \number .....                   | 101, 823                                                     |
| \numexpr .....                  | 306, 355                                                     |
| \opem@shipout .....             | 1216                                                         |
| \or .....                       | 101                                                          |
| \outer .....                    | 143,<br>277, 593, 632, 638, 1204, 1204, 1206                 |
| \parindent .....                | 24                                                           |
| \pdfescapehex .....             | 446                                                          |
| \pdfescapename .....            | 73, 446                                                      |
| \pdfescapestring .....          | 73, 446                                                      |
| \pdffilesize .....              | 656                                                          |
| \pdfmapfile .....               | 301                                                          |
| \pdfmapline .....               | 301                                                          |
| \pdfstrcmp .....                | xiii, 275, 276, 277, 290, 1126                               |
| \pdfuniformdeviate .....        | 217                                                          |
| \pgfpages@originalshipout ..... | 1217                                                         |
| \pi .....                       | 210                                                          |
| \pr@shipout .....               | 1218                                                         |
| \primitive .....                | 299, 351, 352, 352, 546                                      |
| \protect .....                  | 644, 768, 769, 1178                                          |
| \protected .....                | 143, 352                                                     |
| \ProvidesClass .....            | 7                                                            |
| \ProvidesFile .....             | 7                                                            |
| \ProvidesPackage .....          | 7                                                            |
| \quitvmode .....                | 339                                                          |
| \read .....                     | 160, 636                                                     |
| \readline .....                 | 160, 636                                                     |
| \relax .....                    | 22, 141, 277, 321,<br>326, 337, 525, 525, 729, 731, 754, 785 |
| \RequirePackage .....           | 7, 277, 651                                                  |
| \reserveinserts .....           | 277                                                          |
| \romannumeral .....             | 36, 728                                                      |
| \savecatcodetable .....         | 938                                                          |
| \scantokens .....               | 74, 384, 655                                                 |
| \shipout .....                  | 299                                                          |
| \show .....                     | 16, 58, 337                                                  |
| \showbox .....                  | 1072                                                         |
| \showthe .....                  | 337, 523, 682, 685, 688                                      |
| \showtokens .....               | 58, 412, 616                                                 |

|                                   |                                                                                                                                                                                                                                                                                        |  |
|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| \sin . . . . .                    | 210                                                                                                                                                                                                                                                                                    |  |
| \skip . . . . .                   | 968, 969                                                                                                                                                                                                                                                                               |  |
| \space . . . . .                  | 585                                                                                                                                                                                                                                                                                    |  |
| \splitbotmark . . . . .           | 585                                                                                                                                                                                                                                                                                    |  |
| \splitfirstmark . . . . .         | 585                                                                                                                                                                                                                                                                                    |  |
| \SS . . . . .                     | 1181                                                                                                                                                                                                                                                                                   |  |
| \strcmp . . . . .                 | 275, 290                                                                                                                                                                                                                                                                               |  |
| \string . . . . .                 | 134, 963, 965, 966                                                                                                                                                                                                                                                                     |  |
| \tenrm . . . . .                  | 141                                                                                                                                                                                                                                                                                    |  |
| \tex_lowercase:D . . . . .        | 573                                                                                                                                                                                                                                                                                    |  |
| \tex_unexpanded:D . . . . .       | 349                                                                                                                                                                                                                                                                                    |  |
| \the . . . . .                    | 92, 141,<br>176, 180, 183, 342, 351, 352, 352, 355                                                                                                                                                                                                                                     |  |
| \toks . . . . .                   | xxii, 80,<br>101, 224, 355, 356, 357, 490, 946,<br>947, 947, 948, 949, 949, 950, 951,<br>952, 953, 953, 954, 959, 963, 964,<br>966, 968, 969, 971, 976, 977, 977,<br>978, 978, 978, 979, 983, 1022, 1023,<br>1028, 1029, 1033, 1033, 1034, 1034,<br>1035, 1038, 1040, 1043, 1050, 1068 |  |
| \toks@ . . . . .                  | 357                                                                                                                                                                                                                                                                                    |  |
| \toksdef . . . . .                | 959                                                                                                                                                                                                                                                                                    |  |
| \topmark . . . . .                | 142, 585                                                                                                                                                                                                                                                                               |  |
| \tracingfonts . . . . .           | 300                                                                                                                                                                                                                                                                                    |  |
| \tracingnesting . . . . .         | 384, 655                                                                                                                                                                                                                                                                               |  |
| \tracingonline . . . . .          | 1073                                                                                                                                                                                                                                                                                   |  |
| \typeout . . . . .                | 644                                                                                                                                                                                                                                                                                    |  |
| \uccode . . . . .                 | 1128                                                                                                                                                                                                                                                                                   |  |
| \Ucharcat . . . . .               | 572                                                                                                                                                                                                                                                                                    |  |
| \unexpanded . . . . .             | 34, 52, 52,<br>52, 56, 56, 57, 77, 78, 83, 84, 122,<br>125, 126, 127, 127, 146, 269, 272,<br>351, 352, 352, 354, 379, 401, 402, 509                                                                                                                                                    |  |
| \unhbox . . . . .                 | 246                                                                                                                                                                                                                                                                                    |  |
| \unhcopy . . . . .                | 244                                                                                                                                                                                                                                                                                    |  |
| \uniformdeviate . . . . .         | 217                                                                                                                                                                                                                                                                                    |  |
| \unless . . . . .                 | 23                                                                                                                                                                                                                                                                                     |  |
| \unvbox . . . . .                 | 246                                                                                                                                                                                                                                                                                    |  |
| \unvcopy . . . . .                | 245                                                                                                                                                                                                                                                                                    |  |
| \uppercase . . . . .              | 1050                                                                                                                                                                                                                                                                                   |  |
| \usepackage . . . . .             | 651                                                                                                                                                                                                                                                                                    |  |
| \valign . . . . .                 | 538                                                                                                                                                                                                                                                                                    |  |
| \verso@orig@shipout . . . . .     | 1220                                                                                                                                                                                                                                                                                   |  |
| \vskip . . . . .                  | 182                                                                                                                                                                                                                                                                                    |  |
| \vtop . . . . .                   | 1091                                                                                                                                                                                                                                                                                   |  |
| \wd . . . . .                     | 240, 770, 771                                                                                                                                                                                                                                                                          |  |
| \write . . . . .                  | 163, 640                                                                                                                                                                                                                                                                               |  |
| tex commands:                     |                                                                                                                                                                                                                                                                                        |  |
| \tex_above:D . . . . .            | 194                                                                                                                                                                                                                                                                                    |  |
| \tex_abovedisplayshortskip:D . .  | 195                                                                                                                                                                                                                                                                                    |  |
| \tex_abovedisplayskip:D . . . .   | 196                                                                                                                                                                                                                                                                                    |  |
| \tex_abovewithdelims:D . . . . .  | 197                                                                                                                                                                                                                                                                                    |  |
| \tex_accent:D . . . . .           | 198                                                                                                                                                                                                                                                                                    |  |
| \tex_adjdemerits:D . . . . .      | 199                                                                                                                                                                                                                                                                                    |  |
| \tex_adjustspacing:D . . . . .    | 654, 912                                                                                                                                                                                                                                                                               |  |
| \tex_advance:D ..                 | 200, 8353, 8355,<br>8357, 8359, 8365, 8367, 8369, 8371,<br>14251, 14254, 14260, 14263, 14592,<br>14594, 14598, 14600, 14686, 14688,<br>14692, 14694, 23108, 23115, 23118,<br>23518, 23520, 23553, 23555, 24752                                                                         |  |
| \tex_afterassignment:D . . . . .  | 201, 11227, 23459, 23502                                                                                                                                                                                                                                                               |  |
| \tex_aftergroup:D . . . . .       | 202, 941, 1452                                                                                                                                                                                                                                                                         |  |
| \tex_alignmark:D . . . . .        | 787, 1243                                                                                                                                                                                                                                                                              |  |
| \tex_aligntab:D . . . . .         | 788, 1244                                                                                                                                                                                                                                                                              |  |
| \tex_atop:D . . . . .             | 203                                                                                                                                                                                                                                                                                    |  |
| \tex_atopwithdelims:D . . . . .   | 204                                                                                                                                                                                                                                                                                    |  |
| \tex_attribute:D . . . . .        | 789, 1245                                                                                                                                                                                                                                                                              |  |
| \tex_attributedef:D . . . . .     | 790, 1246                                                                                                                                                                                                                                                                              |  |
| \tex_automaticdiscretionary:D ..  | 792                                                                                                                                                                                                                                                                                    |  |
| \tex_automatichyphenmode:D . . .  | 793                                                                                                                                                                                                                                                                                    |  |
| \tex_automatichyphenpenalty:D ..  | 795                                                                                                                                                                                                                                                                                    |  |
| \tex_automospacing:D . . . . .    | 1111                                                                                                                                                                                                                                                                                   |  |
| \tex_autoxspacing:D . . . . .     | 1112                                                                                                                                                                                                                                                                                   |  |
| \tex_badness:D . . . . .          | 205                                                                                                                                                                                                                                                                                    |  |
| \tex_baselineskip:D . . . . .     | 206                                                                                                                                                                                                                                                                                    |  |
| \tex_batchmode:D . . . . .        | 207, 12024                                                                                                                                                                                                                                                                             |  |
| \tex_begincsname:D . . . . .      | 796                                                                                                                                                                                                                                                                                    |  |
| \tex_begingroup:D                 | 208, 1205, 1307, 1447                                                                                                                                                                                                                                                                  |  |
| \tex_beginL:D . . . . .           | 516                                                                                                                                                                                                                                                                                    |  |
| \tex_beginR:D . . . . .           | 517                                                                                                                                                                                                                                                                                    |  |
| \tex_belowdisplayshortskip:D ..   | 209                                                                                                                                                                                                                                                                                    |  |
| \tex_belowdisplayskip:D . . . . . | 210                                                                                                                                                                                                                                                                                    |  |
| \tex_binoppenalty:D . . . . .     | 211                                                                                                                                                                                                                                                                                    |  |
| \tex_bodydir:D . . . . .          | 797, 1282, 1362                                                                                                                                                                                                                                                                        |  |
| \tex_bodydirection:D . . . . .    | 798                                                                                                                                                                                                                                                                                    |  |
| \tex_botmark:D . . . . .          | 212                                                                                                                                                                                                                                                                                    |  |
| \tex_botmarks:D . . . . .         | 518                                                                                                                                                                                                                                                                                    |  |
| \tex_box:D . . . . .              | 213, 27035, 27037, 27077, 32908, 32911                                                                                                                                                                                                                                                 |  |
| \tex_boxdir:D . . . . .           | 799, 1283                                                                                                                                                                                                                                                                              |  |
| \tex_boxdirection:D . . . . .     | 800                                                                                                                                                                                                                                                                                    |  |
| \tex_boxmaxdepth:D . . . . .      | 214                                                                                                                                                                                                                                                                                    |  |
| \tex_breakafterdirmode:D . . . .  | 801                                                                                                                                                                                                                                                                                    |  |
| \tex_brokenpenalty:D . . . . .    | 215                                                                                                                                                                                                                                                                                    |  |
| \tex_catcode:D . . . . .          | 216, 2570, 10414, 10416, 29261                                                                                                                                                                                                                                                         |  |
| \tex_catcodetable:D . . . . .     | 802, 1247, 22672, 22679                                                                                                                                                                                                                                                                |  |
| \tex_char:D . . . . .             | 217                                                                                                                                                                                                                                                                                    |  |
| \tex_chardef:D . . . . .          | 218, 314, 1440, 1469, 1471, 1768,<br>1769, 8328, 9085, 9107, 9112, 11147,<br>12778, 12976, 29569, 29571, 29573                                                                                                                                                                         |  |
| \tex_cleaders:D . . . . .         | 219                                                                                                                                                                                                                                                                                    |  |
| \tex_clearmarks:D . . . . .       | 803, 1248                                                                                                                                                                                                                                                                              |  |

\tex\_closein:D ..... 220, 12789  
\tex\_closeout:D ..... 221, 12986  
\tex\_clubpenalties:D ..... 519  
\tex\_clubpenalty:D ..... 222  
\tex\_copy:D ..... 223, 27029,  
27031, 27052, 27061, 27070, 27078  
\tex\_copyfont:D ..... 655, 913  
\tex\_count:D .....  
..... 224, 12726, 12728, 12940,  
12942, 22985, 23001, 23009, 23010  
\tex\_countdef:D ..... 225  
\tex\_cr:D ..... 226  
\tex\_crampeddisplaystyle:D 804, 1249  
\tex\_crampedscriptscriptstyle:D ..  
..... 806, 1250  
\tex\_crampedscriptstyle:D . 807, 1252  
\tex\_crampedtextstyle:D ... 808, 1253  
\tex\_crcr:D ..... 227  
\tex\_creationdate:D ..... 777  
\tex\_csname:D ..... 228, 1434  
\tex\_cssstring:D ..... 809  
\tex\_currentcjktoken:D ... 1113, 1170  
\tex\_currentgrouplevel:D ..  
.. 520, 940, 22661, 22741, 22750, 22757  
\tex\_currentgroupstype:D ..... 521  
\tex\_currentifbranch:D ..... 522  
\tex\_currentiflevel:D ..... 523  
\tex\_currentiftype:D ..... 524  
\tex\_currentspacingmode:D .... 1114  
\tex\_currentxspacingmode:D ... 1115  
\tex\_day:D ..... 229, 1314, 1318  
\tex\_deadcycles:D ..... 230  
\tex\_def:D ..  
.. 231, 701, 702, 703, 1453, 1455,  
1457, 1458, 1479, 1481, 1482, 1483,  
1485, 1486, 1487, 1489, 1490, 1491  
\tex\_defaulthyphenchar:D ..... 232  
\tex\_defaultskewchar:D ..... 233  
\tex\_delcode:D ..... 234  
\tex\_delimiter:D ..... 235  
\tex\_delimiterfactor:D ..... 236  
\tex\_delimitershortfall:D ..... 237  
\tex\_detokenize:D ..  
..... 525, 1443, 1445, 29254  
\tex\_dimen:D ..... 238  
\tex\_dimendef:D ..... 239  
\tex\_dimexpr:D .... 526, 14206, 27007  
\tex\_directlua:D ..... 810,  
1236, 1237, 9411, 9726, 14182, 28962  
\tex\_disablecjktoken:D ..... 1171  
\tex\_discretionary:D ..... 240  
\tex\_disinhbitglue:D ..... 1116  
\tex\_displayindent:D ..... 241  
\tex\_displaylimits:D ..... 242  
\tex\_displaystyle:D ..... 243  
\tex\_displaywidowpenalties:D .. 527  
\tex\_displaywidowpenalty:D .... 244  
\tex\_displaywidth:D ..... 245  
\tex\_divide:D ..... 246, 22979, 24753  
\tex\_doublehyphendemerits:D ... 247  
\tex\_dp:D ..... 248, 27045  
\tex\_draftmode:D ..... 656, 914  
\tex\_dtou:D ..... 1117  
\tex\_dump:D ..... 249  
\tex\_dviextension:D ..... 811  
\tex\_dvifeedback:D ..... 812  
\tex\_dvivariable:D ..... 813  
\tex\_eachlinedepth:D ..... 657  
\tex\_eachlineheight:D ..... 658  
\tex\_edef:D ..... 250,  
1206, 1207, 1223, 1308, 1309, 1314,  
1315, 1320, 1321, 1326, 1327, 1480,  
1484, 1488, 1492, 13198, 13256, 32675  
\tex\_efcode:D ..... 689  
\tex\_elapsedtime:D ..... 659, 778  
\tex\_else:D ..  
.... 251, 1209, 1235, 1311, 1317,  
1323, 1329, 1420, 1472, 1475, 1517  
\tex\_emergencystretch:D ..... 252  
\tex\_enablecjktoken:D ... 1172, 9399  
\tex\_end:D ..... 253, 1193, 1345, 1879  
\tex\_endcsname:D ..... 254, 1435  
\tex\_endgroup:D ..  
.... 255, 1191, 1231, 1332, 1448  
\tex\_endinput:D ..  
.... 256, 12033, 13904, 14116  
\tex\_endL:D ..... 528  
\tex\_endlinechar:D ..  
.. 162, 163, 177, 257, 3784, 3785,  
3786, 3820, 5641, 12845, 12847,  
12848, 22623, 22627, 22640, 22674,  
22675, 22690, 22839, 22854, 22890  
\tex\_endR:D ..... 529  
\tex\_epTeXinputencoding:D .... 1118  
\tex\_epTeXversion:D 1119, 32319, 32342  
\tex\_eqno:D ..... 258  
\tex\_errhelp:D ..... 259, 11892  
\tex\_errmessage:D ... 260, 1871, 11912  
\tex\_errorcontextlines:D ..  
.... 261, 4697, 11907, 11926, 12126, 27141  
\tex\_errorstopmode:D ..... 262  
\tex\_escapechar:D 263, 2182, 5896,  
5945, 5998, 12867, 13060, 13107,  
13113, 23424, 23486, 23487, 23797  
\tex\_eTeXglueshrinkorder:D .... 814  
\tex\_eTeXgluestretchorder:D ... 815  
\tex\_eTeXrevision:D ..... 530  
\tex\_eTeXversion:D .....

\tex\_etoksapp:D . . . . . 816  
 \tex\_etokspre:D . . . . . 817  
 \tex\_euc:D . . . . . 1120  
 \tex\_everycr:D . . . . . 264  
 \tex\_everydisplay:D . . . . . 265, 1197  
 \tex\_everyeof:D . . . . .  
     . . . . . 532, 3793, 3845, 9573, 13527  
 \tex\_everyhbox:D . . . . . 266  
 \tex\_everyjob:D . . . . . 267, 1339, 1346  
 \tex\_everymath:D . . . . . 268, 1198  
 \tex\_everypar:D . . . . . 269  
 \tex\_everyvbox:D . . . . . 270  
 \tex\_exceptionpenalty:D . . . . . 818  
 \tex\_exhyphenpenalty:D . . . . . 271  
 \tex\_expandafter:D . . . . . 272, 706,  
     1210, 1224, 1226, 1227, 1436, 29254  
 \tex\_expanded:D . . . . . 364,  
     820, 1355, 1516, 1517, 2253, 2256,  
     2323, 2326, 2359, 2365, 2456, 2459,  
     2480, 2483, 2548, 2551, 2579, 3056,  
     3096, 3104, 4293, 4297, 10583, 12578  
 \tex\_explicitdiscretionary:D . . . . . 821  
 \tex\_explicithyphenpenalty:D . . . . . 819  
 \tex\_fam:D . . . . . 273  
 \tex\_fi:D . . . . . 274, 707, 1195,  
     1204, 1228, 1230, 1239, 1240, 1241,  
     1299, 1301, 1302, 1306, 1313, 1319,  
     1325, 1331, 1337, 1340, 1343, 1356,  
     1364, 1369, 1421, 1477, 1478, 1519  
 \tex\_filedump:D . . . . .  
     . . . . . 716, 779, 1407, 13754, 13766, 13773  
 \tex\_filemoddate:D . . . . .  
     . . . . . 715, 780, 1403, 13875  
 \tex\_filesize:D . . . . .  
     . . . . . 658, 658, 713, 781, 1390,  
     13546, 13616, 13648, 13773, 13800  
 \tex\_finalhyphendemerits:D . . . . . 275  
 \tex\_firstlineheight:D . . . . . 660  
 \tex\_firstmark:D . . . . . 276  
 \tex\_firstmarks:D . . . . . 533  
 \tex\_firstvalidlanguage:D . . . . . 822  
 \tex\_floatingpenalty:D . . . . . 277  
 \tex\_font:D . . . . . 278, 15968  
 \tex\_fontchardp:D . . . . . 534  
 \tex\_fontcharht:D . . . . . 535  
 \tex\_fontcharic:D . . . . . 536  
 \tex\_fontcharwd:D . . . . . 537  
 \tex\_fontdimen:D . . . . . 279, 15957  
 \tex\_fontexpand:D . . . . . 661, 915  
 \tex\_fontid:D . . . . . 823, 1254  
 \tex\_fontname:D . . . . . 280  
 \tex\_fontsize:D . . . . . 662  
 \tex\_forcecjktoken:D . . . . . 1173  
 \tex\_formatname:D . . . . . 824, 1255  
 \tex\_futurelet:D . . . . .  
     . . . . . 281, 11222, 11224, 23432, 23490  
 \tex\_gdef:D . . . . . 282, 1493, 1496, 1500, 1504  
 \tex\_gleaders:D . . . . . 830, 1256  
 \tex\_global:D . . . . . 183, 187, 283, 674, 708,  
     1226, 1312, 1318, 1324, 1330, 1948,  
     1955, 8306, 8312, 8316, 8335, 8346,  
     8357, 8359, 8369, 8371, 8379, 9085,  
     9112, 10882, 10884, 10894, 11224,  
     12778, 12976, 14220, 14225, 14241,  
     14248, 14254, 14263, 14564, 14568,  
     14584, 14589, 14594, 14600, 14656,  
     14662, 14678, 14683, 14688, 14694,  
     15968, 27031, 27037, 27107, 27160,  
     27172, 27185, 27205, 27252, 27264,  
     27276, 27289, 27310, 27325, 32911  
 \tex\_globaldefs:D . . . . . 284  
 \tex\_glueexpr:D . . . . . 538, 14582,  
     14584, 14592, 14594, 14598, 14600,  
     14614, 14621, 14627, 14630, 22040  
 \tex\_glueshrink:D . . . . . 539  
 \tex\_glueshrinkorder:D . . . . . 540  
 \tex\_gluestretch:D . . . . . 541, 23644, 23650  
 \tex\_gluestretchorder:D . . . . . 542  
 \tex\_gluetomu:D . . . . . 543  
 \tex\_halign:D . . . . . 285  
 \tex\_hangafter:D . . . . . 286  
 \tex\_hangindent:D . . . . . 287  
 \tex\_hbadness:D . . . . . 288  
 \tex\_hbox:D . . . . . 289, 27152, 27155,  
     27160, 27167, 27172, 27179, 27185,  
     27199, 27205, 27213, 27218, 28664  
 \tex\_hfi:D . . . . . 1121  
 \tex\_hfil:D . . . . . 290  
 \tex\_hfill:D . . . . . 291  
 \tex\_hfilneg:D . . . . . 292  
 \tex\_hfuzz:D . . . . . 293  
 \tex\_hjcode:D . . . . . 825  
 \tex\_hoffset:D . . . . . 294, 1358  
 \tex\_holdinginserts:D . . . . . 295  
 \tex\_hpack:D . . . . . 826  
 \tex\_hrule:D . . . . . 296  
 \tex\_hsize:D . . . . .  
     . . . . . 297, 27815, 27837, 27839, 27888  
 \tex\_hskip:D . . . . . 298, 14625  
 \tex\_hss:D . . . . .  
     . . . . . 299, 27222, 27224, 27226, 27669, 27678  
 \tex\_ht:D . . . . . 300, 27044  
 \tex\_hyphen:D . . . . . 193, 1199  
 \tex\_hyphenation:D . . . . . 301  
 \tex\_hyphenationbounds:D . . . . . 827  
 \tex\_hyphenationmin:D . . . . . 828  
 \tex\_hyphenchar:D . . . . . 302, 15958  
 \tex\_hyphenpenalty:D . . . . . 303

\tex\_hyphenpenalty:D . . . . . 829  
\tex\_if:D . . . . . 129, 304, 1423, 1424  
\tex\_ifabsdim:D . . . . . 651, 916  
\tex\_ifabsnum:D 652, 917, 16027, 16031  
\tex\_ifcase:D . . . . . 305, 8199  
\tex\_ifcat:D . . . . . 306, 1425  
\tex\_ifcondition:D . . . . . 831  
\tex\_ifcsname:D . . . . . 544, 1433  
\tex\_ifdbox:D . . . . . 1122  
\tex\_ifddir:D . . . . . 1123  
\tex\_ifdefined:D . . . . . 545, 705,  
1192, 1196, 1202, 1233, 1236, 1242,  
1301, 1302, 1333, 1338, 1341, 1344,  
1357, 1365, 1432, 1470, 1473, 1517  
\tex\_ifdim:D . . . . . 307, 14205  
\tex\_ifeof:D . . . . . 308, 12809  
\tex\_iffalse:D . . . . . 309, 1418  
\tex\_iffontchar:D . . . . . 546  
\tex\_ifhbox:D . . . . . 310, 27089  
\tex\_ifhmode:D . . . . . 311, 1429  
\tex\_ifincsname:D . . . . . 690  
\tex\_ifinner:D . . . . . 312, 1431  
\tex\_ifjfont:D . . . . . 1124  
\tex\_ifmbox:D . . . . . 1125  
\tex\_ifmdir:D . . . . . 1126  
\tex\_ifmmode:D . . . . . 313, 1428  
\tex\_ifnum:D . . . . . 314, 1300, 1450  
\tex\_ifodd:D . . . . . 315, 1427, 8198, 9078  
\tex\_ifprimitive:D . . . . . 653, 783  
\tex\_iftbox:D . . . . . 1127  
\tex\_iftdir:D . . . . . 1129  
\tex\_iftfont:D . . . . . 1128  
\tex\_iftrue:D . . . . . 316, 1417  
\tex\_ifvbox:D . . . . . 317, 27090  
\tex\_ifvmode:D . . . . . 318, 1430  
\tex\_ifvoid:D . . . . . 319, 27091  
\tex\_ifx:D . . . . . 320, 1208,  
1225, 1310, 1316, 1322, 1328, 1426  
\tex\_ifybox:D . . . . . 1130  
\tex\_ifydir:D . . . . . 1131  
\tex\_ignoreddimen:D . . . . . 663  
\tex\_ignoreligaturesinfont:D . . . . . 918  
\tex\_ignorespaces:D . . . . . 321  
\tex\_immediate:D . . . . .  
. . . . . 322, 1888, 1890, 12978, 12986, 13027  
\tex\_immediateassigned:D . . . . . 832  
\tex\_immediateassignment:D . . . . . 833  
\tex\_indent:D . . . . . 323, 2237  
\tex\_inhibitglue:D . . . . . 1132  
\tex\_inhibitxspcode:D . . . . . 1133  
\tex\_initcatcodetable:D . . . . .  
. . . . . 834, 1257, 22588  
\tex\_input:D . . . . .  
. . . . . 324, 1194, 1347, 9578, 13533, 13922  
\tex\_inputlineno:D . . . . . 325, 1886, 11830  
\tex\_insert:D . . . . . 326  
\tex\_insertht:D . . . . . 664, 919  
\tex\_insertpenalties:D . . . . . 327  
\tex\_interactionmode:D . . . . .  
. . . . . 547, 27125, 27128, 27130  
\tex\_interlinepenalties:D . . . . . 548  
\tex\_interlinepenalty:D . . . . . 328  
\tex\_italiccorrection:D . . . . .  
. . . . . 192, 1200, 1359  
\tex\_jcharwidowpenalty:D . . . . . 1134  
\tex\_jfam:D . . . . . 1135  
\tex\_jfont:D . . . . . 1136  
\tex\_jis:D . . . . . 1137  
\tex\_jobname:D . . . . .  
. . . . . 329, 9658, 9742, 13376, 13377  
\tex\_kanjiskip:D . . . . . 1138, 9397  
\tex\_kansuji:D . . . . . 1139  
\tex\_kansujichar:D . . . . . 1140  
\tex\_kcatcode:D . . . . . 1141  
\tex\_kchar:D . . . . . 1174  
\tex\_kchardef:D . . . . . 1175  
\tex\_kern:D . . . . .  
. . . . . 330, 27377, 27667, 27676, 28238,  
28498, 28503, 28585, 28586, 28882,  
28883, 32063, 32065, 32114, 32116  
\tex\_kuten:D . . . . . 1142, 1176  
\tex\_language:D . . . . . 331, 1348  
\tex\_lastbox:D . . . . . 332, 27105, 27107  
\tex\_lastkern:D . . . . . 333  
\tex\_lastlinedepth:D . . . . . 665  
\tex\_lastlinefit:D . . . . . 549  
\tex\_lastnamedcs:D . . . . . 835  
\tex\_lastnodechar:D . . . . . 1143  
\tex\_lastnodesubtype:D . . . . . 1144  
\tex\_lastnodestype:D . . . . . 550  
\tex\_lastpenalty:D . . . . . 334  
\tex\_lastskip:D . . . . . 335  
\tex\_lastxpos:D . . . . . 666, 926  
\tex\_lastypos:D . . . . . 667, 927  
\tex\_latelua:D . . . . . 836, 1258, 28963  
\tex\_lateluafunction:D . . . . . 837  
\tex\_lccode:D . . . . . 336, 3642,  
3643, 3644, 5009, 5010, 5032, 5033,  
10490, 10492, 23405, 23415, 23484,  
23486, 23489, 23519, 26303, 26355  
\tex\_leaders:D . . . . . 337  
\tex\_left:D . . . . . 338, 1366  
\tex\_leftghost:D . . . . . 838, 1284  
\tex\_lefthyphenmin:D . . . . . 339  
\tex\_leftmarginkern:D . . . . . 691  
\tex\_leftskip:D . . . . . 340  
\tex\_leqno:D . . . . . 341

|                                        |                              |
|----------------------------------------|------------------------------|
| \tex_let:D . . . . .                   | 184, 187, 342, 708,          |
| 1193, 1194, 1197, 1198, 1199, 1200,    |                              |
| 1201, 1203, 1204, 1226, 1232, 1234,    |                              |
| 1238, 1243, 1244, 1245, 1246, 1247,    |                              |
| 1248, 1249, 1250, 1252, 1253, 1254,    |                              |
| 1255, 1256, 1257, 1258, 1259, 1260,    |                              |
| 1261, 1262, 1263, 1264, 1265, 1266,    |                              |
| 1267, 1268, 1269, 1270, 1271, 1272,    |                              |
| 1273, 1275, 1276, 1278, 1279, 1280,    |                              |
| 1282, 1283, 1284, 1285, 1286, 1288,    |                              |
| 1289, 1290, 1291, 1292, 1293, 1294,    |                              |
| 1295, 1296, 1297, 1298, 1304, 1305,    |                              |
| 1312, 1318, 1324, 1330, 1334, 1335,    |                              |
| 1336, 1339, 1342, 1345, 1346, 1347,    |                              |
| 1348, 1349, 1350, 1351, 1352, 1353,    |                              |
| 1354, 1355, 1358, 1359, 1360, 1361,    |                              |
| 1362, 1363, 1366, 1367, 1368, 1417,    |                              |
| 1418, 1419, 1420, 1421, 1422, 1423,    |                              |
| 1424, 1425, 1426, 1427, 1428, 1429,    |                              |
| 1430, 1431, 1432, 1433, 1434, 1435,    |                              |
| 1436, 1437, 1438, 1439, 1441, 1442,    |                              |
| 1443, 1444, 1445, 1446, 1447, 1448,    |                              |
| 1450, 1451, 1452, 1468, 1479, 1480,    |                              |
| 1493, 1494, 1944, 10882, 10884,        |                              |
| 10894, 23406, 23416, 32613, 32616      |                              |
| \tex_letcharcode:D . . . . .           | 839                          |
| \tex_letterspacefont:D . . . . .       | 692                          |
| \tex_limits:D . . . . .                | 343                          |
| \tex_linedir:D . . . . .               | 840                          |
| \tex_linedirection:D . . . . .         | 841                          |
| \tex_linepenalty:D . . . . .           | 344                          |
| \tex_lineskip:D . . . . .              | 345                          |
| \tex_lineskiplimit:D . . . . .         | 346                          |
| \tex_localbrokenpenalty:D .            | 842, 1285                    |
| \tex_localinterlinepenalty:D . . . . . | 843, 1286                    |
| \tex_localleftbox:D . . . . .          | 848, 1288                    |
| \tex_localrightbox:D . . . . .         | 849, 1289                    |
| \tex_long:D . . . . .                  | 347, 701, 702,               |
| 703, 1453, 1455, 1458, 1481, 1482,     |                              |
| 1483, 1484, 1485, 1487, 1489, 1490,    |                              |
| 1491, 1492, 1496, 1498, 1504, 1506     |                              |
| \tex_looseness:D . . . . .             | 348                          |
| \tex_lower:D . . . . .                 | 349, 27088                   |
| \tex_lowercase:D . . . . .             | 350, 1200, 3645, 5011, 5034, |
| 10521, 10628, 11899, 23406, 23416,     |                              |
| 23485, 26304, 26356, 32512, 32874      |                              |
| \tex_lpcode:D . . . . .                | 693                          |
| \tex_luabytecode:D . . . . .           | 844                          |
| \tex_luabytecodecall:D . . . . .       | 845                          |
| \tex_luacopyinputnodes:D . . . . .     | 846                          |
| \tex_luadef:D . . . . .                | 847                          |
| \tex_luaescapestring:D . . . . .       | 850, 1259, 28961             |
| \tex_luafunction:D . . . . .           | 851, 1260                    |
| \tex_luafunctioncall:D . . . . .       | 852                          |
| \tex_luatexbanner:D . . . . .          | 853                          |
| \tex_luatexrevision:D . . . . .        | 854, 32326                   |
| \tex_luatexversion:D . . . . .         | 855, 1302, 1333, 1470,       |
| 8323, 9002, 9395, 10871, 12929, 32324  |                              |
| \tex_mag:D . . . . .                   | 351                          |
| \tex_mapfile:D . . . . .               | 668, 1304                    |
| \tex_mapline:D . . . . .               | 669, 1305                    |
| \tex_mark:D . . . . .                  | 352                          |
| \tex_marks:D . . . . .                 | 551                          |
| \tex_mathaccent:D . . . . .            | 353                          |
| \tex_mathbin:D . . . . .               | 354                          |
| \tex_mathchar:D . . . . .              | 355                          |
| \tex_mathchardef:D . . . . .           | 314, 356,                    |
| 1476, 8331, 8332, 29570, 29572, 29574  |                              |
| \tex_mathchoice:D . . . . .            | 357                          |
| \tex_mathclose:D . . . . .             | 358                          |
| \tex_mathcode:D . . . . .              | 359, 10484, 10486            |
| \tex_mathdelimitersmode:D . . . . .    | 856                          |
| \tex_mathdir:D . . . . .               | 857, 1290                    |
| \tex_mathdirection:D . . . . .         | 858                          |
| \tex_mathdisplayskipmode:D . . . . .   | 859                          |
| \tex_matheqnogapstep:D . . . . .       | 860                          |
| \tex_mathinner:D . . . . .             | 360                          |
| \tex_mathnolimitsmode:D . . . . .      | 861                          |
| \tex_mathop:D . . . . .                | 361, 1349                    |
| \tex_mathopen:D . . . . .              | 362                          |
| \tex_mathoption:D . . . . .            | 862                          |
| \tex_mathord:D . . . . .               | 363                          |
| \tex_mathpenaltiesmode:D . . . . .     | 863                          |
| \tex_mathpunct:D . . . . .             | 364                          |
| \tex_mathrel:D . . . . .               | 365                          |
| \tex_mathrulesfam:D . . . . .          | 864                          |
| \tex_mathscriptboxmode:D . . . . .     | 866                          |
| \tex_mathscriptcharmode:D . . . . .    | 867                          |
| \tex_mathscriptsmode:D . . . . .       | 865                          |
| \tex_mathstyle:D . . . . .             | 868, 1261                    |
| \tex_mathsurround:D . . . . .          | 366                          |
| \tex_mathsurroundmode:D . . . . .      | 869                          |
| \tex_mathsurroundskip:D . . . . .      | 870                          |
| \tex_maxdeadcycles:D . . . . .         | 367                          |
| \tex_maxdepth:D . . . . .              | 368                          |
| \tex_mdfivesum:D .                     | 714, 782, 1394, 13710        |
| \tex_meaning:D . . . . .               | 369, 1207, 1224,             |
| 1308, 1314, 1320, 1326, 1441, 1442     |                              |
| \tex_medmuskip:D . . . . .             | 370                          |
| \tex_message:D . . . . .               | 371                          |
| \tex_middle:D . . . . .                | 552, 1367                    |
| \tex_mkern:D . . . . .                 | 372                          |
| \tex_month:D . . . . .                 | 373, 1320, 1324, 1350        |

|                                 |                                                           |                                  |                |
|---------------------------------|-----------------------------------------------------------|----------------------------------|----------------|
| \tex_moveleft:D .....           | 374, 27082                                                | \tex_pagedirection:D .....       | 879            |
| \tex_moveright:D .....          | 375, 27084                                                | \tex_pagediscards:D .....        | 556            |
| \tex_mskip:D .....              | 376                                                       | \tex_pagefillstretch:D .....     | 403            |
| \tex_muexpr:D ..                | 553, 14676, 14678,<br>14686, 14688, 14692, 14694, 14698   | \tex_pagefillstretch:D .....     | 404            |
| \tex_multiply:D .....           | 377                                                       | \tex_pagefilstretch:D .....      | 405            |
| \tex_muskip:D .....             | 378                                                       | \tex_pagefiststretch:D .....     | 1147           |
| \tex_muskipdef:D .....          | 379                                                       | \tex_pagegoal:D .....            | 406            |
| \tex_mutoglu:D .....            | 305, 554                                                  | \tex_pageheight:D .....          | 672, 930, 1293 |
| \tex_newlinechar:D              | 380, 1870, 3786,<br>3812, 3816, 4695, 11905, 12124, 13026 | \tex_pageleftoffset:D .....      | 880, 1265      |
| \tex_noalign:D .....            | 381                                                       | \tex_pagerightoffset:D .....     | 881, 1294      |
| \tex_noautospacing:D .....      | 1145                                                      | \tex_pageshrink:D .....          | 407            |
| \tex_noautoxspacing:D .....     | 1146                                                      | \tex_pagestretch:D .....         | 408            |
| \tex_noboundary:D .....         | 382                                                       | \tex_pagetopoffset:D .....       | 882, 1266      |
| \tex_noexpand:D .....           | 383, 1437                                                 | \tex_pagetotal:D .....           | 409            |
| \tex_nohrule:D .....            | 871                                                       | \tex_pagewidth:D .....           | 673, 931, 1295 |
| \tex_noindent:D .....           | 384                                                       | \tex_par:D .....                 | 410            |
| \tex_nokerns:D .....            | 872, 1262                                                 | \tex_pardir:D .....              | 883, 1296      |
| \tex_noligatures:D .....        | 670                                                       | \tex_pardirection:D .....        | 884            |
| \tex_noligs:D .....             | 873, 1263                                                 | \tex_parfillskip:D .....         | 411            |
| \tex_nolimits:D .....           | 385                                                       | \tex_parindent:D .....           | 412            |
| \tex_nonscript:D .....          | 386                                                       | \tex_parshape:D .....            | 413            |
| \tex_nonstopmode:D .....        | 387                                                       | \tex_parshapedimen:D .....       | 557            |
| \tex_normaldeviate:D .....      | 671, 928                                                  | \tex_parshapeindent:D .....      | 558            |
| \tex_nospaces:D .....           | 874                                                       | \tex_parshapelen:D .....         | 559            |
| \tex_novrule:D .....            | 875                                                       | \tex_parskip:D .....             | 414            |
| \tex_nulldelimiterspace:D ..... | 388                                                       | \tex_patterns:D .....            | 415            |
| \tex_nullfont:D .....           | 389, 11176                                                | \tex_pausing:D .....             | 416            |
| \tex_number:D .....             | 390, 8195, 27718                                          | \tex_pdfannot:D .....            | 582            |
| \tex_numexpr:D                  | 555, 8196, 16197, 23795                                   | \tex_pdfcatalog:D .....          | 583            |
| \tex_odelcode:D .....           | 1180                                                      | \tex_pdfcolorstack:D .....       | 585            |
| \tex_odelimiter:D .....         | 1181                                                      | \tex_pdfcolorstackinit:D .....   | 586            |
| \tex_omathaccent:D .....        | 1182                                                      | \tex_pdfcompresslevel:D .....    | 584            |
| \tex_omathchar:D .....          | 1183                                                      | \tex_pdfcreationdate:D .....     | 587            |
| \tex_omathchardef:D .....       | 1184, 1473, 1474, 8324, 8326, 8327                        | \tex_pdfdecimaldigits:D .....    | 588            |
| \tex_omathcode:D .....          | 1185                                                      | \tex_pdfdest:D .....             | 589            |
| \tex_omit:D .....               | 391                                                       | \tex_pdfdestmargin:D .....       | 590            |
| \tex_openin:D .....             | 392, 12780                                                | \tex_pdfendlink:D .....          | 591            |
| \tex_openout:D .....            | 393, 12978                                                | \tex_pdfendthread:D .....        | 592            |
| \tex_or:D .....                 | 394, 1419                                                 | \tex_pdfextension:D .....        | 885            |
| \tex_oradical:D .....           | 1186                                                      | \tex_pdffeedback:D .....         | 886            |
| \tex_outer:D .....              | 395, 1351, 32675                                          | \tex_pdffontattr:D .....         | 593            |
| \tex_output:D .....             | 396                                                       | \tex_pdffontname:D .....         | 594            |
| \tex_outputbox:D .....          | 876, 1264                                                 | \tex_pdffontobjnum:D .....       | 595            |
| \tex_outputpenalty:D .....      | 397                                                       | \tex_pdffgamma:D .....           | 596            |
| \tex_over:D .....               | 398, 1352                                                 | \tex_pdffgentounicode:D .....    | 599            |
| \tex_overfullrule:D .....       | 399                                                       | \tex_pdffglyptounicode:D .....   | 600            |
| \tex_overline:D .....           | 400                                                       | \tex_pdhorigin:D .....           | 601            |
| \tex_overwithdelims:D .....     | 401                                                       | \tex_pdfimageapplygamma:D .....  | 597            |
| \tex_pagebottomoffset:D ..      | 877, 1291                                                 | \tex_pdfimagegamma:D .....       | 598            |
| \tex_pagedepth:D .....          | 402                                                       | \tex_pdfimagehicolor:D .....     | 602            |
| \tex_pagedir:D .....            | 878, 1292, 1363                                           | \tex_pdfimageresolution:D .....  | 603            |
|                                 |                                                           | \tex_pdfincludchars:D .....      | 604            |
|                                 |                                                           | \tex_pdfinclusioncopyfonts:D ..  | 605            |
|                                 |                                                           | \tex_pdfinclusionerrorlevel:D .. | 607            |

\tex\_pdfinfo:D ..... 608  
 \tex\_pdflastannot:D ..... 609  
 \tex\_pdflastlink:D ..... 610  
 \tex\_pdflastobj:D ..... 611  
 \tex\_pdflastxform:D ..... 612, 921  
 \tex\_pdflastximage:D ..... 613, 923  
 \tex\_pdflastximagecolordepth:D .. 615  
 \tex\_pdflastximagepages:D .. 616, 925  
 \tex\_pdflinkmargin:D ..... 617  
 \tex\_pdliteral:D ..... 618  
 \tex\_pdmajorversion:D ..... 619  
 \tex\_pdmorversion:D ..... 620  
 \tex\_pdfnames:D ..... 621  
 \tex\_pdfobj:D ..... 622  
 \tex\_pdfobjcompresslevel:D .... 623  
 \tex\_pdfoutline:D ..... 624  
 \tex\_pdfoutput:D ..... 539,  
     625, 929, 1342, 9439, 9445, 9453, 9757  
 \tex\_pdfpageattr:D ..... 626  
 \tex\_pdfpagebox:D ..... 628  
 \tex\_pdfpageref:D ..... 629  
 \tex\_pdfpageresources:D ..... 630  
 \tex\_pdfpagesattr:D ..... 627, 631  
 \tex\_pdxrefobj:D ..... 632  
 \tex\_pdxrefxform:D ..... 633, 935  
 \tex\_pdxrefximage:D ..... 634, 936  
 \tex\_pdxrestore:D ..... 635  
 \tex\_pdxretval:D ..... 636  
 \tex\_pdfsav:D ..... 637  
 \tex\_pdxsetmatrix:D ..... 638  
 \tex\_pdxstartlink:D ..... 639  
 \tex\_pdxstartthread:D ..... 640  
 \tex\_pdxsuppressptexinfo:D .... 641  
 \tex\_pdxtexbanner:D ..... 686, 1334  
 \tex\_pdxtexrevision:D 687, 1335, 32307  
 \tex\_pdxtexversion:D .....  
     ... 302, 688, 1301, 1336, 9396, 32305  
 \tex\_pdxthread:D ..... 642  
 \tex\_pdxthreadmargin:D ..... 643  
 \tex\_pdxtrailer:D ..... 644  
 \tex\_pdxuniquepresname:D ..... 645  
 \tex\_pdxvariable:D ..... 887  
 \tex\_pdxvorigin:D ..... 646  
 \tex\_pdxform:D ..... 647, 938  
 \tex\_pdxformname:D ..... 648  
 \tex\_pdximage:D ..... 649, 939  
 \tex\_pdximagebbox:D ..... 650  
 \tex\_penalty:D ..... 417  
 \tex\_pkmode:D ..... 674  
 \tex\_pkresolution:D ..... 675  
 \tex\_postbreakpenalty:D ..... 1148  
 \tex\_postdisplaypenalty:D ..... 418  
 \tex\_postexhyphenchar:D ... 888, 1267  
 \tex\_posthyphenchar:D .... 889, 1268  
 \tex\_prebinoppenalty:D ..... 890  
 \tex\_preibreakpenalty:D ..... 1149  
 \tex\_predisplaydirection:D .... 560  
 \tex\_predisplaygapfactor:D .... 891  
 \tex\_predisplaypenalty:D ..... 419  
 \tex\_predisplaysize:D ..... 420  
 \tex\_preehyphenchar:D .... 892, 1269  
 \tex\_prehyphenchar:D ..... 893, 1270  
 \tex\_prerelpenalty:D ..... 894  
 \tex\_pretolerance:D ..... 421  
 \tex\_prevdepth:D ..... 422  
 \tex\_prevgraf:D ..... 423  
 \tex\_primitive:D .....  
     ... 352, 676, 784, 2630, 9667, 9677  
 \tex\_protected:D .....  
     ... 561, 1481, 1483, 1485,  
         1486, 1487, 1488, 1489, 1490, 1491,  
         1492, 1500, 1502, 1504, 1506, 32675  
 \tex\_protrudechars:D ..... 677, 932  
 \tex\_ptexminorversion:D .....  
     ... 1150, 32316, 32335  
 \tex\_ptexrevision:D 1151, 32317, 32336  
 \tex\_ptexversion:D .....  
     ... 1152, 32311, 32314, 32330, 32333  
 \tex\_pxdimen:D ..... 678, 933  
 \tex\_quitvmode:D ..... 694  
 \tex\_radical:D ..... 424  
 \tex\_raise:D ..... 425, 27086  
 \tex\_randomseed:D .... 679, 934, 9694  
 \tex\_read:D ..... 426, 12025, 12829  
 \tex\_readline:D ..... 562, 12846  
 \tex\_readpapersizespecial:D .. 1153  
 \tex\_relax:D .....  
     ... 305, 427, 731, 1446, 8197, 14207  
 \tex\_relpriority:D ..... 428  
 \tex\_resettimer:D ..... 680, 785  
 \tex\_right:D ..... 429, 1368  
 \tex\_rightghost:D ..... 895, 1297  
 \tex\_righthyphenmin:D ..... 430  
 \tex\_rightmarginkern:D ..... 695  
 \tex\_rightsskip:D ..... 431  
 \tex\_roman numeral:D .....  
     ... 325, 325, 350, 432,  
         1439, 1451, 1773, 10533, 16199, 22661  
 \tex\_rpcode:D ..... 696  
 \tex\_savecatcodetable:D .....  
     ... 896, 1271, 22622, 22678  
 \tex\_savepos:D ..... 681, 937  
 \tex\_savinghyphcodes:D ..... 563  
 \tex\_savingvdiscards:D ..... 564  
 \tex\_scantextokens:D ..... 897, 1272  
 \tex\_scantokens:D .... 565, 3798, 3859  
 \tex\_scriptbaselineshiftfactor:D ..... 1155

```

\tex_scriptfont:D 433
\tex_scriptscripbaseelineshiftfactor:D
 1157
\tex_scriptscripfont:D 434
\tex_scriptscripstyle:D 435
\tex_scriptspace:D 436
\tex_scriptstyle:D 437
\tex_scrollmode:D 438
\tex_setbox:D ... 439, 27029, 27031,
 27035, 27037, 27052, 27061, 27070,
 27105, 27107, 27155, 27160, 27167,
 27172, 27179, 27185, 27199, 27205,
 27247, 27252, 27259, 27264, 27271,
 27276, 27283, 27289, 27304, 27310,
 27321, 27325, 28664, 32908, 32911
\tex_setfontid:D 898
\tex_setlanguage:D 440
\tex_setrandomseed:D ... 682, 940, 9710
\tex_sfcode:D 441, 10502, 10504
\tex_shapemode:D 899
\tex_shellescape:D ... 683, 786, 9729
\tex_shipout:D 442, 1203, 1227
\tex_show:D 443
\tex_showbox:D 444, 27142
\tex_showboxbreadth:D ... 445, 27138
\tex_showboxdepth:D ... 446, 27139
\tex_showgroups:D 566
\tex_showifs:D 567
\tex_showlists:D 447
\tex_showmode:D 1158
\tex_showthe:D 448
\tex_showtokens:D 412, 568, 1361, 4699, 12128
\tex_sjis:D 1159
\tex_skewchar:D 449
\tex_skip:D 450, 23522,
 23551, 23570, 23628, 23644, 23650
\tex_skipdef:D 451
\tex_space:D 191
\tex_spacefactor:D 452
\tex_spaceskip:D 453
\tex_span:D 454
\tex_special:D 455
\tex_splitbotmark:D 456
\tex_splitbotmarks:D 569
\tex_splitdiscards:D 570
\tex_splitfirstmark:D 457
\tex_splitfirstmarks:D 571
\tex_splitmaxdepth:D 458
\tex_splittopskip:D 459
\tex_strcmp:D 712, 1372, 4837, 13834, 16570
\tex_string:D 460, 1206,
 1210, 1309, 1315, 1321, 1327, 1444
\tex_suppressfontnotfounderror:D
 718, 1280
\tex_suppressifcsnameerror:D ...
 900, 1273
\tex_suppresslongerror:D .. 901, 1275
\tex_suppressmathparerror:D 902, 1276
\tex_suppressoutererror:D .. 903, 1278
\tex_suppressprimitiveerror:D .. 905
\tex_synctex:D 697
\tex_tabskip:D 461
\tex_tagcode:D 698
\tex_tate:D 1160
\tex_tbaselineshift:D 1161
\tex_textbaselineshiftfactor:D 1163
\tex_textdir:D 906, 1298
\tex_textdirection:D 907
\tex_textfont:D 462
\tex_textstyle:D 463
\tex_TeXeTstate:D 572
\tex_tfont:D 1164
\tex_the:D ... 163, 305, 343, 464,
 765, 770, 771, 1886, 2168, 2296,
 2300, 2629, 2671, 2762, 2781, 2796,
 2801, 7760, 8382, 8384, 9694, 10416,
 10486, 10492, 10498, 10504, 14437,
 14438, 14439, 14509, 14511, 14622,
 14624, 14699, 17188, 17679, 23065,
 23097, 23145, 23146, 23177, 23178,
 23184, 23185, 23643, 23753, 23798,
 23817, 23823, 23826, 23830, 27125
\tex_thickmuskip:D 465
\tex_thinmuskip:D 466
\tex_time:D 467, 1308, 1312
\tex_toks:D 468,
 2772, 2801, 7760, 7771, 7772, 7773,
 23038, 23065, 23097, 23134, 23145,
 23146, 23177, 23178, 23184, 23185,
 23189, 23199, 23209, 23467, 23485,
 23643, 23798, 23801, 23808, 23816,
 23817, 23822, 23823, 23826, 23830
\tex_toksapp:D 908
\tex_toksdef:D 469, 23316
\tex_tokspre:D 909
\tex_tolerance:D 470
\tex_topmark:D 471
\tex_topmarks:D 573
\tex_topskip:D 472
\tex_tpck:D 910
\tex_tracingassigns:D 574
\tex_tracingcommands:D 473
\tex_tracingfonts:D 684, 941, 1232, 1234, 1238
\tex_tracinggroups:D 575
\tex_tracingifs:D 576

```

|                                   |                         |                                   |      |
|-----------------------------------|-------------------------|-----------------------------------|------|
| \tex_tracinglostchars:D .....     | 474                     | \tex_Umathfractionnumvgap:D ...   | 989  |
| \tex_tracingmacros:D .....        | 475                     | \tex_Umathfractionrule:D .....    | 990  |
| \tex_tracingnesting:D .....       | 577, 3783, 9572, 13526  | \tex_Umathinnerbinspacing:D ...   | 991  |
| \tex_tracingonline:D .....        | 476, 27140              | \tex_Umathinnerclosespacing:D ... | 993  |
| \tex_tracingoutput:D .....        | 477                     | \tex_Umathinnerinnerspacing:D ... | 995  |
| \tex_tracingpages:D .....         | 478                     | \tex_Umathinneropenspacing:D ...  | 996  |
| \tex_tracingparagraphs:D .....    | 479                     | \tex_Umathinneropsspacing:D ...   | 997  |
| \tex_tracingrestores:D .....      | 480                     | \tex_Umathinnerordspacing:D ...   | 998  |
| \tex_tracingscantokens:D .....    | 578                     | \tex_Umathinnerpunctspacing:D ... | 1000 |
| \tex_tracingstats:D .....         | 481                     | \tex_Umathinnerrelspacing:D ...   | 1001 |
| \tex_uccode:D .....               | 482, 10496, 10498       | \tex_Umathlimitabovebgap:D ...    | 1002 |
| \tex_Uchar:D .....                | 943, 1279, 29254        | \tex_Umathlimitabovekern:D ...    | 1003 |
| \tex_Ucharcat:D                   | 944, 1384, 10581, 29259 | \tex_Umathlimitabovevgap:D ...    | 1004 |
| \tex_uchyph:D .....               | 483                     | \tex_Umathlimitbelowbgap:D ...    | 1005 |
| \tex_ucs:D .....                  | 1177                    | \tex_Umathlimitbelowkern:D ...    | 1006 |
| \tex_Udelcode:D .....             | 945                     | \tex_Umathlimitbelowvgap:D ...    | 1007 |
| \tex_Udelcodenum:D .....          | 946                     | \tex_Umathnolimitsubfactor:D ...  | 1008 |
| \tex_Udelimiter:D .....           | 947                     | \tex_Umathnolimitsupfactor:D ...  | 1009 |
| \tex_Udelimiterover:D .....       | 948                     | \tex_Umathopbinspacing:D ...      | 1010 |
| \tex_Udelimiterunder:D .....      | 949                     | \tex_Umathopclosespacing:D ...    | 1011 |
| \tex_Uhextensible:D .....         | 950                     | \tex_Umathopenbinspacing:D ...    | 1012 |
| \tex_Umathaccent:D .....          | 951                     | \tex_Umathopenclosespacing:D ...  | 1013 |
| \tex_Umathaxis:D .....            | 952                     | \tex_Umathopeninnerspacing:D ...  | 1014 |
| \tex_Umathbinbinspacing:D .....   | 953                     | \tex_Umathopenopenspacing:D ...   | 1015 |
| \tex_Umathbinclosespacing:D ...   | 954                     | \tex_Umathopenopsspacing:D ...    | 1016 |
| \tex_Umathbininnerspacing:D ...   | 955                     | \tex_Umathopenordspacing:D ...    | 1017 |
| \tex_Umathbinopenspacing:D ...    | 956                     | \tex_Umathopenpunctspacing:D ...  | 1018 |
| \tex_Umathbinopsspacing:D ...     | 957                     | \tex_Umathopenrelspacing:D ...    | 1019 |
| \tex_Umathbinordspacing:D ...     | 958                     | \tex_Umathoperatorsize:D ...      | 1020 |
| \tex_Umathbinpunctspacing:D ...   | 959                     | \tex_Umathopinnerspacing:D ...    | 1021 |
| \tex_Umathbinrelspacing:D ...     | 960                     | \tex_Umathopopenspacing:D ...     | 1022 |
| \tex_Umathchar:D .....            | 961                     | \tex_Umathopopspacing:D ...       | 1023 |
| \tex_Umathcharclass:D .....       | 962                     | \tex_Umathopordspacing:D ...      | 1024 |
| \tex_Umathchardef:D .....         | 963                     | \tex_Umathoppunctspacing:D ...    | 1025 |
| \tex_Umathcharfam:D .....         | 964                     | \tex_Umathoprelspacing:D ...      | 1026 |
| \tex_Umathcharnum:D .....         | 965                     | \tex_Umathordbinspacing:D ...     | 1027 |
| \tex_Umathcharnumdef:D .....      | 966                     | \tex_Umathordclosespacing:D ...   | 1028 |
| \tex_Umathcharslot:D .....        | 967                     | \tex_Umathordinnerspacing:D ...   | 1029 |
| \tex_Umathclosebinspacing:D ...   | 968                     | \tex_Umathordopenspacing:D ...    | 1030 |
| \tex_Umathcloseclosespacing:D ... | 970                     | \tex_Umathorddopspace:D ...       | 1031 |
| \tex_Umathcloseinnerspacing:D ... | 972                     | \tex_Umathordordspacing:D ...     | 1032 |
| \tex_Umathcloseopenspacing:D ...  | 973                     | \tex_Umathordpunctspacing:D ...   | 1033 |
| \tex_Umathcloseopspacing:D ...    | 974                     | \tex_Umathordrelspacing:D ...     | 1034 |
| \tex_Umathcloseordspacing:D ...   | 975                     | \tex_Umathoverbarkern:D ...       | 1035 |
| \tex_Umathclosepunctspacing:D ... | 977                     | \tex_Umathoverbarrule:D ...       | 1036 |
| \tex_Umathcloserelspacing:D ...   | 978                     | \tex_Umathoverbarvgap:D ...       | 1037 |
| \tex_Umathcode:D .....            | 979                     | \tex_Umathoverdelimterbgap:D ...  | 1039 |
| \tex_Umathcodenum:D .....         | 980                     | \tex_Umathoverdelimtervgap:D ...  | 1041 |
| \tex_Umathconnectoverlapmin:D     | 982                     | \tex_Umathpunctbinspacing:D ...   | 1042 |
| \tex_Umathfractiondelsize:D ...   | 983                     | \tex_Umathpunctclosespacing:D ... | 1044 |
| \tex_Umathfractiondenomdown:D ... | 985                     | \tex_Umathpunctinnerspacing:D ... | 1046 |
| \tex_Umathfractiondenomvgap:D ... | 987                     | \tex_Umathpunctopenspacing:D ...  | 1047 |
| \tex_Umathfractionnumup:D ...     | 988                     | \tex_Umathpuncttopspacing:D ...   | 1048 |
|                                   |                         | \tex_Umathpunctordspacing:D ...   | 1049 |

\tex\_Umathpunctpunctspacing:D . 1051  
 \tex\_Umathpunctrelspacing:D .. 1052  
 \tex\_Umathquad:D ..... 1053  
 \tex\_Umathradicaldegreeafter:D 1055  
 \tex\_Umathradicaldegreebefore:D 1057  
 \tex\_Umathradicaldegreeraise:D 1059  
 \tex\_Umathradicalkern:D ..... 1060  
 \tex\_Umathradicalrule:D ..... 1061  
 \tex\_Umathradicalvgap:D ..... 1062  
 \tex\_Umathrelbinspacing:D .... 1063  
 \tex\_Umathrelclosespacing:D .. 1064  
 \tex\_Umathrelinnerspacing:D .. 1065  
 \tex\_Umathrelopenspacing:D ... 1066  
 \tex\_Umathrellospacing:D ..... 1067  
 \tex\_Umathrelordspacing:D .... 1068  
 \tex\_Umathrelpunctspacing:D .. 1069  
 \tex\_Umathrelrelspacing:D .... 1070  
 \tex\_Umathskewedfractionhgap:D 1072  
 \tex\_Umathskewedfractionvgap:D 1074  
 \tex\_Umathspaceafterscript:D .. 1075  
 \tex\_Umathstackdenomdown:D ... 1076  
 \tex\_Umathstacknumup:D ..... 1077  
 \tex\_Umathstackvgap:D ..... 1078  
 \tex\_Umathsubshiftdown:D .... 1079  
 \tex\_Umathsubshiftdrop:D ..... 1080  
 \tex\_Umathsubsupshiftdown:D .. 1081  
 \tex\_Umathsubsupvgap:D ..... 1082  
 \tex\_Umathsubtopmax:D ..... 1083  
 \tex\_Umathsupbottommin:D ..... 1084  
 \tex\_Umathsupshiftdrop:D ..... 1085  
 \tex\_Umathsupshiftup:D ..... 1086  
 \tex\_Umathsupsubbottommax:D .. 1087  
 \tex\_Umathunderbarkern:D ..... 1088  
 \tex\_Umathunderbarrule:D ..... 1089  
 \tex\_Umathunderbarvgap:D .... 1090  
 \tex\_Umathunderdelimitterbgap:D 1092  
 \tex\_Umathunderdelimitervgap:D 1094  
 \tex\_undefined:D ..... 585, 586,  
     1232, 1304, 1305, 1312, 1318, 1324,  
     1330, 1334, 1335, 1336, 1961, 1969,  
     9027, 15131, 15146, 15199, 15220,  
     22988, 23406, 23416, 23494, 23594  
 \tex\_underline:D ..... 484, 1201  
 \tex\_unexpanded:D .....  
     ..... 579, 1354, 1438, 2545, 29257  
 \tex\_unhbox:D ..... 485, 27228  
 \tex\_unhcopy:D ..... 486, 27227  
 \tex\_uniformdeviate:D .....  
     ... 490, 685, 922, 923, 942, 7739,  
     7770, 9463, 22135, 22136, 22317, 22320  
 \tex\_unkern:D ..... 487  
 \tex\_unless:D ..... 580, 1422  
 \tex\_Unosubscript:D ..... 1095  
 \tex\_Unosuperscript:D ..... 1096  
 \tex\_unpenalty:D ..... 488  
 \tex\_unskip:D ..... 489  
 \tex\_unvbox:D ..... 490, 27317  
 \tex\_unvcopy:D ..... 491, 27316  
 \tex\_Uoverdelimiter:D ..... 1097  
 \tex\_uppercase:D ..... 492, 32876  
 \tex\_uptexrevision:D .... 1178, 32340  
 \tex\_uptexversion:D .... 1179, 32339  
 \tex\_Uradical:D ..... 1098  
 \tex\_Uroot:D ..... 1099  
 \tex\_Uskewed:D ..... 1100  
 \tex\_Uskewedwithdelims:D ..... 1101  
 \tex\_Ustack:D ..... 1102  
 \tex\_Ustartdisplaymath:D ..... 1103  
 \tex\_Ustartmath:D ..... 1104  
 \tex\_Ustopdisplaymath:D ..... 1105  
 \tex\_Ustopmath:D ..... 1106  
 \tex\_Usubscript:D ..... 1107  
 \tex\_Usuperscript:D ..... 1108  
 \tex\_Uunderdelimiter:D ..... 1109  
 \tex\_Uvextensible:D ..... 1110  
 \tex\_vadjust:D ..... 493  
 \tex\_valign:D ..... 494  
 \tex\_vbadness:D ..... 495  
 \tex\_vbox:D ..... 496, 27232,  
     27237, 27242, 27247, 27252, 27271,  
     27276, 27283, 27289, 27304, 27310  
 \tex\_vccenter:D ..... 497, 1353  
 \tex\_vfi:D ..... 1169  
 \tex\_vfil:D ..... 498  
 \tex\_vfill:D ..... 499  
 \tex\_vfilneg:D ..... 500  
 \tex\_vfuzz:D ..... 501  
 \tex\_voffset:D ..... 502, 1360  
 \tex\_vpack:D ..... 911  
 \tex\_vrule:D ..... 503, 28747  
 \tex\_vsize:D ..... 504  
 \tex\_vskip:D ..... 505, 14628  
 \tex\_vsplit:D ..... 506, 27321, 27326  
 \tex\_vss:D ..... 507  
 \tex\_vtop:D ... 508, 27234, 27259, 27264  
 \tex\_wd:D ..... 509, 27046  
 \tex\_widowpenalties:D ..... 581  
 \tex\_widowpenalty:D ..... 510  
 \tex\_write:D .....  
     ... 511, 1888, 1890, 13007, 13010, 13027  
 \tex\_xdef:D 512, 1494, 1498, 1502, 1506  
 \tex\_XeTeXcharclass:D ..... 719  
 \tex\_XeTeXcharglyph:D ..... 720  
 \tex\_XeTeXcountfeatures:D ..... 721  
 \tex\_XeTeXcountglyphs:D ..... 722  
 \tex\_XeTeXcountselectors:D .... 723  
 \tex\_XeTeXcountvariations:D ... 724  
 \tex\_XeTeXdashbreakstate:D .... 726

|                                           |                                     |                                                  |
|-------------------------------------------|-------------------------------------|--------------------------------------------------|
| \tex_XeTeXdefaultencoding:D . . . . .     | 725                                 | text commands:                                   |
| \tex_XeTeXfeaturecode:D . . . . .         | 727                                 | \l_text_accents_t1 . . . . .                     |
| \tex_XeTeXfeaturename:D . . . . .         | 728                                 | . . . . . 260, 263, 29541, 29750, 31760          |
| \tex_XeTeXfindfeaturebyname:D . . . . .   | 730                                 | \l_text_case_exclude_arg_t1 . . . . .            |
| \tex_XeTeXfindselectorbyname:D . . . . .  | 732                                 | . . . . . 262, 263, 29559, 30067                 |
| \tex_XeTeXfindvariationbyname:D . . . . . | 734                                 | \text_declare_expand_equivalent:Nn . . . . .     |
| \tex_XeTeXfirstfontchar:D . . . . .       | 735                                 | . . . . . 260, 29871                             |
| \tex_XeTeXfonttype:D . . . . .            | 736                                 | \text_declare_purify_equivalent:Nn . . . . .     |
| \tex_XeTeXgenerateactualtext:D . . . . .  | 738                                 | . . . . . 263, 263,                              |
| \tex_XeTeXglyph:D . . . . .               | 739                                 | 31608, 31621, 31622, 31623, 31624,               |
| \tex_XeTeXglyphbounds:D . . . . .         | 740                                 | 31641, 31666, 31667, 31668, 31670,               |
| \tex_XeTeXglyphindex:D . . . . .          | 741                                 | 31673, 31674, 31680, 31682, 31683,               |
| \tex_XeTeXglyphname:D . . . . .           | 742                                 | 31684, 31692, 31704, 31746, 31761                |
| \tex_XeTeXinputencoding:D . . . . .       | 743                                 | \text_expand . . . . . 262                       |
| \tex_XeTeXinputnormalization:D . . . . .  | 745                                 | \text_expand:n . . . . .                         |
| \tex_XeTeXinterchartokenstate:D . . . . . | 747                                 | . . . . . 260, 263, 29575, 29904, 31438          |
| \tex_XeTeXinterchartoks:D . . . . .       | 748                                 | \l_text_expand_exclude_t1 . . . . .              |
| \tex_XeTeXisdefaultselector:D . . . . .   | 750                                 | . . . . . 260, 263, 29565, 29751                 |
| \tex_XeTeXisexclusivefeature:D . . . . .  | 752                                 | \l_text_letterlike_t1 . . . . .                  |
| \tex_XeTeXlastfontchar:D . . . . .        | 753                                 | . . . . . 260, 263, 29541, 29779                 |
| \tex_XeTeXlinebreaklocale:D . . . . .     | 755                                 | \text_lowercase:n . . . . .                      |
| \tex_XeTeXlinebreakpenalty:D . . . . .    | 756                                 | . . . . . 69, 132, 262, 29882, 32977, 32979      |
| \tex_XeTeXlinebreakskip:D . . . . .       | 754                                 | \text_lowercase:nn . . . . .                     |
| \tex_XeTeXOTcountfeatures:D . . . . .     | 757                                 | . . . . . 262, 29882, 32980, 32982               |
| \tex_XeTeXOTcountlanguages:D . . . . .    | 758                                 | \l_text_math_arg_t1 . . . . .                    |
| \tex_XeTeXOTcountsheets:D . . . . .       | 759                                 | . . . . . 260,                                   |
| \tex_XeTeXOTfeaturetag:D . . . . .        | 760                                 | 263, 263, 29561, 29749, 30066, 31554             |
| \tex_XeTeXOTlanguage tag:D . . . . .      | 761                                 | \l_text_math_delims_t1 . . . . .                 |
| \tex_XeTeXOTscripttag:D . . . . .         | 762                                 | 260, 263,                                        |
| \tex_XeTeXpdffile:D . . . . .             | 763                                 | 263, 263, 29563, 29687, 29999, 31483             |
| \tex_XeTeXpdfpagecount:D . . . . .        | 764                                 | \text_purify:n . . . . .                         |
| \tex_XeTeXpicfile:D . . . . .             | 765                                 | . . . . . 263, 31432                             |
| \tex_XeTeXrevision:D . . . . .            | 766, 9673, 32348                    | \text_titlecase:n . . . . .                      |
| \tex_XeTeXselectorname:D . . . . .        | 767                                 | . . . . . 69, 130, 262, 29882, 32989, 32991      |
| \tex_XeTeXtracingfonts:D . . . . .        | 768                                 | \text_titlecase:nn . . . . .                     |
| \tex_XeTeXupwardsmode:D . . . . .         | 769                                 | . . . . . 262, 29882, 32992, 32994               |
| \tex_XeTeXuseglyphmetrics:D . . . . .     | 770                                 | \l_text_titlecase_check_letter_-                 |
| \tex_XeTeXvariation:D . . . . .           | 771                                 | bool . . . . . 262, 263, 29880, 30204            |
| \tex_XeTeXvariationdefault:D . . . . .    | 772                                 | \text_titlecase_first:n . . . . .                |
| \tex_XeTeXvariationmax:D . . . . .        | 773                                 | . . . . . 262, 29882                             |
| \tex_XeTeXvariationmin:D . . . . .        | 774                                 | \text_titlecase_first:nn . . . . .               |
| \tex_XeTeXvariationname:D . . . . .       | 775                                 | . . . . . 262, 29882, 32983, 32985               |
| \tex_XeTeXversion:D . . . . .             | . . . . .                           | \text_uppercase:n . . . . .                      |
|                                           | 776, 8325, 9003, 9403, 10872, 32347 | . . . . . 69, 130, 132, 262, 29882, 32983, 32985 |
| \tex_xkanjiskip:D . . . . .               | 1165                                | \text_uppercase:nn . . . . .                     |
| \tex_xleaders:D . . . . .                 | 513                                 | . . . . . 262, 29882, 32986, 32988               |
| \tex_xspaceskip:D . . . . .               | 514                                 | text internal commands:                          |
| \tex_xspcde:D . . . . .                   | 1166                                | \_\_text_change_case:nnn . . . . .               |
| \tex_ybaselineshift:D . . . . .           | 1167                                | . . . . . 29883, 29885, 29887, 29889,            |
| \tex_year:D . . . . .                     | 515, 1326, 1330                     | 29891, 29893, 29895, 29897, 29898                |
| \tex_yoko:D . . . . .                     | 1168                                | \_\_text_change_case_aux:nnn . . . . .           |
| \text . . . . .                           | 31628                               | . . . . . 29898                                  |

```

__text_change_case_break:w 29898
__text_change_case_char:nnnN
 29898,
 30307, 30318, 30327, 30341, 30589,
 30689, 30722, 30804, 30818, 30841
__text_change_case_char_-
 aux:nnnN 29898
__text_change_case_char_-
 lower:nnN 29898
__text_change_case_char_next_-
 end:nn 29898
__text_change_case_char_next_-
 lower:nn 29898
__text_change_case_char_next_-
 title:nn 29898
__text_change_case_char_next_-
 titleonly:nn 29898
__text_change_case_char_next_-
 upper:nn 29898
__text_change_case_char_-
 title:nnN 29898
__text_change_case_char_-
 title:nnnN 29898
__text_change_case_char_-
 titleonly:nnN 29898
__text_change_case_char_-
 titleonly:nnNN 29898
__text_change_case_char_-
 titleonly:nnNNN 29898
__text_change_case_char_-
 UTFviii:nnNN 29898
__text_change_case_char_-
 UTFviii:nnnNN 29898
__text_change_case_char_-
 UTFviii:nnnNNN 29898
__text_change_case_char_-
 UTFviii:nnnNNNN 30261
__text_change_case_char_-
 UTFviii:nnnNNNNN 29898
__text_change_case_cs_check:nnN
 29898
__text_change_case_end:w 29898
__text_change_case_exclude:nnN
 29898
__text_change_case_exclude:nnNN
 29898
__text_change_case_exclude:nnNn
 29898
__text_change_case_exclude:nnnN
 29898
__text_change_case_group_-
 lower:nnn 29898
__text_change_case_group_-
 title:nnn 29898
__text_change_case_group_-
 titleonly:nnn 29898
__text_change_case_group_-
 upper:nnn 29898
__text_change_case_if_greek:nTF
 30330
__text_change_case_if_greek_-
 accent:nTF 30330
__text_change_case_if_greek_-
 accent_p:n 30330
__text_change_case_if_greek_-
 diacritic:nTF 30330
__text_change_case_if_greek_-
 diacritic_p:n 30330
__text_change_case_if_greek_p:n
 30330
__text_change_case_if_takes_-
 dialytika:nTF 30330
__text_change_case_letterlike:nnnnN
 29898
__text_change_case_letterlike_-
 lower:nnN 29898
__text_change_case_letterlike_-
 title:nnN 29898
__text_change_case_letterlike_-
 titleonly:nnN 29898
__text_change_case_letterlike_-
 upper:nnN 29898
__text_change_case_loop:nnw . . .
 29898, 30351, 30360, 30372,
 30376, 30406, 30412, 30419, 30427,
 30439, 30545, 30557, 30569, 30576,
 30580, 30630, 30647, 30666, 30760,
 30772, 30784, 30789, 30799, 30816
__text_change_case_lower_-
 az:nnnN 30843
__text_change_case_lower_lt:nnN
 30591
__text_change_case_lower_-
 lt:nnnN 30595
__text_change_case_lower_lt:nnw
 30591
__text_change_case_lower_lt_-
 auxi:nnnN 30597, 30608
__text_change_case_lower_lt_-
 auxii:nnnN 30612, 30633
__text_change_case_lower_-
 sigma:nnN 29898
__text_change_case_lower_-
 sigma:NnnN 29898
__text_change_case_lower_-
 sigma:nnnN 29898, 30636, 30762

```

```

__text_change_case_lower_-
 sigma:nnnNN 29898
__text_change_case_lower_-
 sigma:nnNw 29898
__text_change_case_lower_-
 sigma:nnw 29898
__text_change_case_lower_-
 tr:NnnN 30747
__text_change_case_lower_-
 tr:nnnN 30747, 30844
__text_change_case_lower_-
 tr:nnnNN 30747
__text_change_case_lower_-
 tr:nnNw 30747
__text_change_case_math_-
 group:nnNn 29898
__text_change_case_math_-
 loop:nnNw 29898
__text_change_case_math_N_-
 type:nnNN 29898
__text_change_case_math_-
 search:nnNNN 29898
__text_change_case_math_-
 space:nnNw 29898
__text_change_case_N_type:nnN 29898
__text_change_case_N_type:nnnN .
 29898
__text_change_case_N_type_-
 aux:nnN 29898
__text_change_case_result:n . 29898
__text_change_case_setup:NN ...
 31369, 31376, 31378
__text_change_case_setup:Nn ...
 31400, 31420, 31422
__text_change_case_space:nnw . 29898
__text_change_case_store:n 29898,
 30302, 30324, 30350, 30359, 30371,
 30375, 30386, 30391, 30403, 30567,
 30578, 30624, 30638, 30663, 30691,
 30718, 30741, 30758, 30770, 30782,
 30787, 30798, 30811, 30829, 30836
__text_change_case_store:nw . 29898
__text_change_case_title_-
 el:nnnN 30584
__text_change_case_title_nl:nnN
 30712
__text_change_case_title_-
 nl:nnnN 30712
__text_change_case_title_nl:nnw
 30712
__text_change_case_upper_-
 az:nnnN 30843
__text_change_case_upper_-
 de-alt:nnnN 30294
__text_change_case_upper_-
 de-alt:nnnNN 30294
__text_change_case_upper_el:nnn
 30330
__text_change_case_upper_-
 el:NnnN 30330
__text_change_case_upper_-
 el:nnnN 30330
__text_change_case_upper_-
 el:nnNw 30330
__text_change_case_upper_el_-
 dialytika:N 30330
__text_change_case_upper_el_-
 dialytika:nnN 30330
__text_change_case_upper_el_-
 gobble:nnN 30330
__text_change_case_upper_el_-
 gobble:nnw 30330
__text_change_case_upper_el_-
 hiatus:nnN 30330
__text_change_case_upper_el_-
 hiatus:nnNw 30330
__text_change_case_upper_lt:nnN
 30669
__text_change_case_upper_-
 lt:nnnN 30673
__text_change_case_upper_lt:nnw
 30669
__text_change_case_upper_lt_-
 aux:nnnN 30675, 30686
__text_change_case_upper_-
 tr:nnnN 30821, 30846
__text_change_cases_lower_-
 lt:nnnN 30591
__text_change_cases_lower_lt_-
 auxi:nnnN 30591
__text_change_cases_lower_lt_-
 auxii:nnnN 30591
__text_change_cases_upper_-
 lt:nnnN 30669
__text_change_cases_upper_lt_-
 aux:nnnN 30669
__text_char_catcode:N .
 29494, 30148,
 30158, 30159, 30303, 30396, 30397,
 30568, 30579, 30626, 30627, 30628,
 30639, 30664, 30692, 30719, 30742,
 30759, 30771, 30783, 30788, 30832
\c__text_chardef_group_begin_-
 token 29569, 29651
\c__text_chardef_group_end_token
 29569, 29659
\c__text_chardef_space_token ...
 29569, 29638

```

```

\c__text_dotless_i_t1 . 30798, 30847
\c__text_dotted_I_t1 .. 30837, 30847
__text_end_env:n 31667
__text_expand:n 29575
__text_expand_cs:N 29575
__text_expand_cs_expand:N ... 29575
__text_expand_encoding:N 29575
__text_expand_encoding_escape:N
 29575
__text_expand_encoding_escape>NN
 29831, 29834
__text_expand_end:w 29575
__text_expand_exclude:N 29575
__text_expand_exclude>NN 29575
__text_expand_exclude:Nn 29575
__text_expand_exclude:nN 29575
__text_expand_explicit:N 29575
__text_expand_group:n 29575
__text_expand_implicit:N 29575
__text_expand_letterlike:N .. 29575
__text_expand_letterlike>NN .. 29575
__text_expand_loop:w 29575
__text_expand_math_group:Nn .. 29575
__text_expand_math_loop:Nw .. 29575
__text_expand_math_N_type:NN .. 29575
__text_expand_math_search:NNN .. 29575
__text_expand_math_space:Nw .. 29575
__text_expand_N_type:N 29575
__text_expand_N_type_auxi:N .. 29575
__text_expand_N_type_auxii:N .. 29575
__text_expand_N_type_auxiii:N .. 29575
__text_expand_noexpand:nn .. 29575
__text_expand_noexpand:w 29856, 29864
__text_expand_protect:N 29575
__text_expand_protect:nN .. 29575
__text_expand_protect:Nw .. 29575
__text_expand_replace:N 29575
__text_expand_replace:n 29575
__text_expand_result:n 29575
__text_expand_space:w 29575
__text_expand_store:n 29575
__text_expand_store:nw .. 29575
\c__text_final_sigma_t1
 30183, 30198, 30847
\c__text_grosses_Eszett_t1
 30324, 30847
\c__text_I_ogonek_t1 30847
\c__text_i_ogonek_t1 30847
__text_if_expandable:NTF
 29526, 29853, 31598
__text_if_recursion_tail_stop:N
 31431
__text_if_recursion_tail_stop_-
 do:Nn 29408, 29631, 29693, 29718, 29762,
 29784, 29992, 30009, 30034, 30078,
 31477, 31489, 31530, 31558, 31605
__text_loop:Nn 31689, 31697, 31701,
 31709, 31720, 31763, 31768, 31795
__text_loop:nn 30888, 30897, 30933, 31238
__text_loop>NNn .. 31814, 31820, 31822
\l__text_math_mode_t1 29568
\c__text_mathchardef_group_-
 begin_token 29569, 29652
\c__text_mathchardef_group_end_-
 token 29569, 29660
\c__text_mathchardef_space_token
 29569, 29642
__text_purify:n 31432
__text_purify_accent:NN 31747
__text_purify_end:w 31432
__text_purify_expand:N 31432
__text_purify_group:n 31432
__text_purify_loop:w 31432
__text_purify_math_cmd:N 31432
__text_purify_math_cmd:n 31563, 31567
__text_purify_math_cmd:NN ... 31432
__text_purify_math_cmd:Nn ... 31432
__text_purify_math_end:w 31432
__text_purify_math_group:NNn .. 31432
__text_purify_math_loop>NNw .. 31432
__text_purify_math_N_type:NNN 31432
__text_purify_math_result:n ...
 31501,
 31505, 31506, 31507, 31512, 31568
__text_purify_math_search:NNN .. 31432
__text_purify_math_space>NNw .. 31432
__text_purify_math_start>NNw .. 31432
__text_purify_math_stop:Nw ...
 31512, 31531
__text_purify_math_store:n .. 31432
__text_purify_math_store:nw .. 31432
__text_purify_N_type:N 31432
__text_purify_N_type_aux:N .. 31432
__text_purify_protect:N 31432
__text_purify_replace:N 31432
__text_purify_replace:n 31432
__text_purify_result:n ...
 31446, 31450, 31451, 31452
__text_purify_space:w 31432
__text_purify_store:n 31432
__text_purify_store:nw .. 31432
__text_quark_if_nil:NTF 29403, 29817
__text_quark_if_nil_p:n .. 29403

```

\c\_\_text\_sigma\_tl ..... 30197, 30847  
 \\_\_text\_tmp:n ..... 31706,  
     31711, 31767, 31774, 31781, 31819  
 \\_\_text\_tmp:nnn ... 30899, 30930,  
     30931, 31711, 31712, 31786, 31787  
 \\_\_text\_tmp:nnnnn 31198, 31235, 31236  
 \\_\_text\_tmp:w ..... 30852,  
     30855, 30871, 30874, 30875, 30876,  
     30877, 30878, 30879, 30880, 30893,  
     30915, 31140, 31143, 31161, 31167,  
     31168, 31169, 31170, 31171, 31172,  
     31173, 31174, 31175, 31176, 31177,  
     31180, 31192, 31195, 31196, 31197,  
     31217, 31337, 31340, 31359, 31365  
 \\_\_text\_tmp\_aux:n ..... 31783, 31786  
 \\_\_text\_token\_to\_explicit:N ....  
     ..... 29409, 31587  
 \\_\_text\_token\_to\_explicit:n .. 29409  
 \\_\_text\_token\_to\_explicit\_auxi:w  
     ..... 29409  
 \\_\_text\_token\_to\_explicit\_-  
     auxii:w ..... 29409  
 \\_\_text\_token\_to\_explicit\_-  
     auxiii:w ..... 29409  
 \\_\_text\_token\_to\_explicit\_char:N  
     ..... 29409  
 \\_\_text\_token\_to\_explicit\_cs:N 29409  
 \\_\_text\_token\_to\_explicit\_cs\_-  
     aux:N ..... 29409  
 \\_\_text\_use\_i\_delimit\_by\_q\_-  
     recursion\_stop:nw .....  
     ..... 29406, 29697, 29766,  
     29788, 30013, 30082, 31493, 31562  
 \textbaselineshiftfactor ..... 1162  
 \textbf ..... 31633  
 \textdir ..... 906  
 \textdirection ..... 907  
 \textfont ..... 462  
 \textit ..... 31635  
 \textmd ..... 31634  
 \textnormal ..... 31629  
 \textrm ..... 31630  
 \textsc ..... 31638  
 \textsf ..... 31631  
 \textsl ..... 31636  
 \textstyle ..... 463  
 \texttt ..... 18684, 31632  
 \textulc ..... 31639  
 \textup ..... 31637  
 \TeXeTstate ..... 572  
 \tfont ..... 1164  
 \TH ..... 29557, 31389, 31730  
 \th ..... 29557, 31389, 31743  
 \the ..... 45, 124, 125, 126,  
     127, 128, 129, 130, 131, 132, 133, 464  
 \thickmuskip ..... 465  
 \thinmuskip ..... 466  
 thousand commands:  
     \c\_one\_thousand ..... 32757  
     \c\_ten\_thousand ..... 32759  
 \time ..... 467, 1309, 9683, 9685  
 \tiny ..... 28726, 31664  
 tl commands:  
     \c\_empty\_tl ..... 58,  
     517, 551, 2653, 3589, 3605, 3607,  
     3640, 3947, 8726, 8732, 9810, 9829,  
     12040, 29289, 32419, 32456, 32475  
     \l\_my\_tl ..... 226, 232  
     \c\_novalue\_tl ..... 48, 58, 3641, 4056  
     \c\_space\_tl .....  
         58, 3650, 4508, 5303, 10265, 10274,  
         11834, 13458, 13460, 14104, 28509,  
         28553, 28620, 29302, 29373, 29624,  
         29645, 29731, 29983, 30049, 31470,  
         31546, 31680, 32171, 32173, 32691  
 \tl\_analysis\_map\_inline:Nn .....  
     ..... 225, 23669, 25222  
 \tl\_analysis\_map\_inline:nn .....  
     ..... 225, 23669, 25783  
 \tl\_analysis\_show:N 225, 23696, 32866  
 \tl\_analysis\_show:n 225, 23696, 32868  
 \tl\_build\_begin:N ..... 272, 273,  
     273, 273, 1017, 1196, 24385, 24884,  
     25248, 25341, 26090, 32375, 32390  
 \tl\_build\_clear:N ..... 272, 32390  
 \tl\_build\_end:N ..... 272, 273,  
     1017, 1196, 1197, 24415, 24423,  
     24894, 25303, 25360, 26124, 32463  
 \tl\_build\_gbegin:N .....  
     .... 272, 273, 273, 273, 32375, 32391  
 \tl\_build\_gclear:N ..... 272, 32390  
 \tl\_build\_gend:N ..... 273, 32463  
 \tl\_build\_get:N ..... 273  
 \tl\_build\_get:NN ..... 273, 32449  
 \tl\_build\_gput\_left:Nn ... 273, 32432  
 \tl\_build\_gput\_right:Nn .. 273, 32392  
 \tl\_build\_put\_left:Nn ... 273, 32432  
 \tl\_build\_put\_right:Nn .. 273, 1045,  
     1198, 24392, 24410, 24418, 24422,  
     24472, 24475, 24508, 24522, 24526,  
     24651, 24665, 24706, 24731, 24740,  
     24750, 24782, 24795, 24799, 24881,  
     24887, 24893, 24897, 24940, 25208,  
     25224, 25242, 25311, 25356, 25369,  
     26109, 26148, 26180, 26242, 26245,  
     26260, 26304, 26320, 26356, 32392  
 \tl\_case:Nn ..... 48, 4086

\tl\_case:nn ..... 418  
 \tl\_case:nn(TF) ..... 511, 1192  
 \tl\_case:Nnn ..... 32869, 32871  
 \tl\_case:NnTF .....  
   .. 48, 4086, 4091, 4096, 32870, 32872  
 \tl\_clear:N .....  
   .. 43, 44, 3604, 3611, 3774, 5587,  
   9871, 9872, 13164, 13165, 13168,  
   13177, 13287, 13290, 13350, 13804,  
   15669, 24070, 26092, 32466, 32527  
 \tl\_clear\_new:N .....  
   44, 3610, 9875, 9876, 14044, 14045,  
   14046, 14047, 14048, 29873, 31610  
 \tl\_concat:NNN ..... 44, 3620, 4749  
 \tl\_const:Nn ..... 43,  
   527, 3592, 3640, 3648, 3650, 3766,  
   5424, 5429, 7532, 7587, 9549, 9869,  
   10630, 10877, 10900, 11403, 11461,  
   11758, 11759, 11798, 11803, 11805,  
   11807, 11809, 11811, 11816, 11817,  
   11824, 13061, 13067, 13505, 16225,  
   16226, 16227, 16228, 16229, 16237,  
   16326, 18425, 19728, 20175, 20176,  
   20177, 20178, 20179, 20180, 20181,  
   20182, 20183, 23783, 23842, 26389,  
   29277, 29292, 29315, 29327, 29356,  
   29386, 29387, 29388, 30857, 30901,  
   30917, 31145, 31182, 31200, 31219,  
   31342, 31372, 31374, 31392, 31393,  
   31408, 31415, 31766, 31817, 32512  
 \tl\_count:N ..... 27, 48, 51, 52, 4194  
 \tl\_count:n ..... 27, 48, 51, 52, 333,  
   426, 504, 735, 1618, 1622, 2010,  
   2060, 4194, 4561, 4576, 4588, 25140  
 \tl\_count\_tokens:n ..... 52, 4207  
 \tl\_gclear:N .. 43, 949, 3604, 3613,  
   9652, 9747, 9873, 9874, 23059, 32471  
 \tl\_gclear\_new:N .. 44, 3610, 9877, 9878  
 \tl\_gconcat:NNN ..... 44, 3620, 4750  
 \tl\_gput\_left:Nn .. 44, 3669, 6328, 6511  
 \tl\_gput\_right:Nn .....  
   ..... 44, 1570, 1571, 3567,  
   3717, 7662, 9655, 9750, 22404, 22849  
 \tl\_gremove\_all:Nn ..... 45, 3939  
 \tl\_gremove\_once:Nn ..... 45, 3933  
 \tl\_greplace\_all:Nnn .. 45, 3865, 3942  
 \tl\_greplace\_once:Nnn .. 45, 3865, 3936  
 \tl\_greverse:N ..... 52, 4544  
 .tl\_gset:N ..... 190, 15399  
 \tl\_gset:Nn ..... 44,  
   76, 273, 379, 387, 1199, 3651, 3778,  
   7631, 7839, 7908, 9488, 9494, 9503,  
   9504, 9514, 9515, 9529, 9791, 9793,  
   9795, 9797, 9799, 11520, 11549, 11595  
 \tl\_gset\_eq:NN ..... 44,  
   3607, 3616, 4746, 5507, 5523, 7555,  
   7556, 7557, 7558, 9101, 9883, 9884,  
   9885, 9886, 11432, 11433, 11434,  
   11435, 18430, 23049, 26384, 32952  
 \tl\_gset\_from\_file:Nnn ..... 32943  
 \tl\_gset\_from\_file\_x:Nnn ..... 32943  
 \tl\_gset\_rescan:Nnn ..... 46, 3767  
 .tl\_gset\_x:N ..... 190, 15399  
 \tl\_gsort:Nn ..... 53, 4293, 23047  
 \tl\_gtrim\_spaces:N ..... 53, 4237  
 \tl\_head:N ..... 54, 4293  
 \tl\_head:n 54, 54, 351, 401, 402, 409,  
   410, 2611, 4293, 4585, 29312, 29320  
 \tl\_head:w .....  
   54, 402, 403, 403, 4293, 29337, 29368  
 \tl\_if\_blank:nTF .. 46, 54, 54, 54,  
   3978, 4335, 4575, 4729, 4756, 6662,  
   6665, 9473, 10273, 10352, 10750,  
   11951, 12868, 13370, 13554, 13556,  
   13574, 13607, 13704, 13765, 13772,  
   13845, 13854, 15576, 15782, 23221,  
   25804, 29268, 29281, 29331, 29360,  
   30610, 30635, 30688, 30863, 32687  
 \tl\_if\_blank\_p:n .... 46, 3978, 13722  
 \tl\_if\_empty:N 4833, 4835, 10118, 10120  
 \tl\_if\_empty:NTF .....  
   .. 47, 3945, 11850, 11860, 13181,  
   13271, 13306, 13387, 13641, 13796,  
   13830, 15604, 15609, 15758, 26147  
 \tl\_if\_empty:nTF .....  
   ..... 47, 391, 393, 394, 551,  
   560, 1658, 1749, 2590, 2709, 3312,  
   3319, 3336, 3486, 3879, 3955, 3965,  
   4037, 4077, 4425, 4791, 5555, 6626,  
   7597, 9164, 9823, 9842, 9851, 9854,  
   10131, 10164, 11164, 11384, 11490,  
   12121, 12213, 12228, 12349, 12353,  
   12427, 12489, 12585, 12586, 12595,  
   12602, 12608, 12615, 13175, 13973,  
   13979, 13981, 13983, 14177, 15030,  
   15127, 15950, 16927, 17784, 22083,  
   22151, 23787, 23788, 26964, 29354  
 \tl\_if\_empty\_p:N .... 47, 3945, 14107  
 \tl\_if\_empty\_p:n .... 47, 3955, 3965  
 \tl\_if\_eq:NN ..... 417  
 \tl\_if\_eq:nn(TF) ..... 122, 122  
 \tl\_if\_eq:NNTF ..... 38, 47, 47,  
   47, 48, 63, 488, 3989, 4110, 7700,  
   11646, 12194, 12248, 28805, 28808  
 \tl\_if\_eq:NnTF ..... 47, 4001  
 \tl\_if\_eq:nnTF .....  
   .... 47, 63, 79, 79, 488, 4014, 10152  
 \tl\_if\_eq\_p>NN ..... 47, 3989

```

\tl_if_exist:N 4829, 4831
\tl_if_exist:NTF
 44, 3611, 3613, 3638, 4187, 10762,
 10773, 10852, 10860, 14126, 23698
\tl_if_exist_p:N 44, 3638, 14106
\tl_if_head_eq_catcode:nN
 403, 404, 404
\tl_if_head_eq_catcode:nNTF 55, 4341
\tl_if_head_eq_catcode_p:nN 55, 4341
\tl_if_head_eq_charcode:nN
 402, 404, 404
\tl_if_head_eq_charcode:nNTF ...
 55, 4341, 29270
\tl_if_head_eq_charcode_p:nN 55, 4341
\tl_if_head_eq_meaning:nN 403
\tl_if_head_eq_meaning:nNTF 55, 4341
\tl_if_head_eq_meaning_p:nN ...
 55, 4341, 25139
\tl_if_head_is_group:NTF
 55, 2587, 2706, 4361, 4399,
 4432, 4484, 9849, 13422, 29606,
 29710, 29932, 30026, 31463, 31522
\tl_if_head_is_group_p:n ... 55, 4432
\tl_if_head_is_N_type:n 403
\tl_if_head_is_N_type:nTF
 . 55, 2584, 2648, 2694, 2700, 2745,
 2760, 2805, 4074, 4344, 4358, 4375,
 4412, 4646, 13419, 29603, 29707,
 29929, 30023, 30144, 30180, 30347,
 30404, 30425, 30543, 30564, 30645,
 30698, 30726, 30767, 31460, 31519
\tl_if_head_is_N_type_p:n .. 55, 4412
\tl_if_head_is_space:nTF
 55, 4447, 4628, 4637, 5297
\tl_if_head_is_space_p:n ... 55, 4447
\tl_if_in:Nn
 560
\tl_if_in:nn
 393, 394
\tl_if_in:NnTF
 47, 3561, 3901, 4027, 4027, 4028
\tl_if_in:nnTF
 47, 393,
 417, 3815, 3885, 3887, 4027, 4028,
 4029, 4032, 4863, 4871, 9563, 11382,
 12107, 12109, 23966, 28600, 32361
\tl_if_novalue:nTF
 48, 4043
\tl_if_novalue_p:n ... 48, 4043
\tl_if_single:n
 394
\tl_if_single:NTF 48, 4057, 4058, 4059
\tl_if_single:nTF
 48, 530, 4058, 4059, 4060, 4061
\tl_if_single_p:N
 48, 4057
\tl_if_single_p:n ... 48, 4057, 4061
\tl_if_single_token:nTF
 48, 4072, 31403
\tl_if_single_token_p:n ... 48, 4072
\tl_item:Nn
 56, 4550
\tl_item:nn
 56, 409, 4550, 4576
\tl_log:N
 58, 4678, 5336
\tl_log:n
 58,
 337, 337, 805, 2160, 2176, 4680,
 4705, 5335, 9038, 9138, 18455, 32168
\tl_lower_case:n
 32977
\tl_lower_case:nn
 32977
\tl_map_break:
 50,
 237, 972, 4123, 4129, 4141, 4151,
 4158, 4166, 4172, 4178, 23692, 23693
\tl_map_break:n
 50, 50, 4178, 13563, 13660, 23054
\tl_map_function>NN
 49,
 49, 49, 49, 270, 270, 4119, 4202, 25210
\tl_map_function:nN
 49, 49, 49, 2054,
 4119, 4197, 5636, 5638, 7600, 24492
\tl_map_inline:Nn
 49, 49, 49,
 4133, 5420, 5422, 13659, 23054, 31760
\tl_map_inline:nn
 49,
 49, 4133, 9405, 11317, 11319,
 11321, 11331, 13064, 17837, 20916,
 23016, 31614, 31625, 31642, 31671
\tl_map_tokens:Nn ... 49, 4147, 13562
\tl_map_tokens:nn
 49, 4147
\tl_map_variable>NNn
 49, 4162
\tl_map_variable:nNn ... 49, 397, 4162
\tl_mixed_case:n
 32977
\tl_mixed_case:nn
 32977
\tl_new:N ... 43, 44, 134, 381, 3586,
 3611, 3613, 3999, 4000, 4710, 4711,
 4712, 4713, 5342, 5343, 7529, 7530,
 9548, 9656, 9751, 9766, 9811, 9866,
 9867, 10569, 11213, 11402, 11752,
 12139, 12140, 12717, 12720, 12742,
 12934, 12959, 13038, 13040, 13053,
 13055, 13056, 13058, 13362, 13392,
 13393, 14916, 14919, 14924, 14926,
 14932, 14933, 14935, 14936, 22400,
 22575, 22576, 22943, 23369, 23833,
 23834, 23840, 23841, 23851, 25770,
 26017, 26019, 27687, 27712, 27713,
 28718, 28955, 29541, 29544, 29559,
 29561, 29563, 29565, 29568, 32515
\tl_put_left:Nn
 44, 3669
\tl_put_right:Nn
 44, 1197, 3717, 7660, 10600,
 10602, 10605, 10607, 10608, 10610,
 10612, 10614, 10615, 10617, 10619,
 10621, 10623, 13312, 13315, 13320,
 13677, 24077, 26183, 32554, 32559
\tl_rand_item:N
 56, 4573
\tl_rand_item:n
 56, 4573

```

```

\tl_range:Nnn 57, 4580
\tl_range:nnn 57, 68, 272, 4580
\tl_range_braced:Nnn 272, 32481
\tl_range_braced:nnn 57, 272, 32481
\tl_range_unbraced:Nnn 272, 32481
\tl_range_unbraced:nnn 57, 272, 32481
\tl_remove_all:Nn 45, 45, 3939
\tl_remove_once:Nn 45, 3933
\tl_replace_all:Nnn 45, 485, 558, 3865, 3940, 7609
\tl_replace_once:Nnn 45, 3865, 3934, 10637
\tl_rescan:nn .. 46, 46, 222, 384, 3767
\tl_reverse:N 52, 52, 4544
\tl_reverse:n 52, 52, 4525, 4545, 4547
\tl_reverse_items:n .. 52, 52, 52, 4221
.tl_set:N 190, 15399
\tl_set:Nn 44, 45, 46, 76, 190,

 273, 273, 358, 379, 381, 387, 617,

 698, 1199, 3651, 3776, 4004, 4017,

 4018, 4173, 4694, 5576, 5744, 7599,

 7603, 7629, 7696, 7705, 7790, 7793,

 7810, 7818, 7837, 7846, 7905, 8020,

 8623, 9474, 9553, 9588, 9968, 9974,

 9983, 9990, 10242, 10598, 11250,

 11514, 11530, 11531, 11539, 11540,

 11542, 11548, 11551, 11584, 11585,

 11594, 11606, 11629, 11679, 11889,

 12123, 12151, 12234, 12827, 12840,

 12873, 13129, 13166, 13509, 13544,

 13634, 13662, 13777, 13779, 13781,

 13783, 13823, 14057, 14060, 14061,

 14062, 14063, 14070, 14071, 14072,

 14074, 14078, 14479, 14927, 15108,

 15428, 15437, 15467, 15479, 15487,

 15509, 15521, 15529, 15540, 15549,

 15560, 15675, 18758, 22699, 22709,

 24426, 25194, 25306, 25363, 25853,

 25862, 25940, 25972, 26282, 26561,

 26684, 27955, 28601, 28602, 28956,

 29542, 29545, 29560, 29562, 29564,

 29566, 29874, 31611, 32526, 32964
\tl_set_eq:NN 44, 528, 3605,

 3616, 4745, 5505, 5514, 7551, 7552,

 7553, 7554, 9100, 9879, 9880, 9881,

 9882, 11428, 11429, 11430, 11431,

 12197, 12205, 13680, 15006, 15584,

 15664, 15685, 18429, 23047, 26379
\tl_set_from_file:Nnn 32943
\tl_set_from_file_x:Nnn 32943
\tl_set_rescan:Nnn 46, 46, 222, 385, 655, 3767
.tl_set_x:N 190, 15399
\tl_show:N 58, 58, 973, 4678, 5333, 23704
\tl_show:n 58, 58, 268, 337, 337, 412, 412,

 805, 1191, 2156, 2173, 4678, 4690,

 5332, 9037, 9136, 10418, 10488,

 10494, 10500, 10506, 18453, 32166
\tl_show_analysis:N 32865
\tl_show_analysis:n 32867
\tl_sort:Nn 53, 4293, 23047
\tl_sort:nN .. 53, 954, 956, 4293, 23217
\tl_tail:N .. 54, 442, 4293, 5739, 25114
\tl_tail:n 54, 4293
\tl_to_lowercase:n 32873
\tl_to_str:N 51, 60, 164, 415, 644, 1048,

 4183, 4799, 4855, 4863, 5901, 10853,

 10861, 13124, 13135, 14021, 14035
\tl_to_str:n 46,

 46, 51, 51, 60, 69, 70, 117, 145,

 145, 164, 186, 231, 232, 311, 320,

 391, 394, 415, 420, 427, 584, 602,

 603, 1048, 1048, 1443, 1466, 1557,

 1562, 1643, 1725, 2190, 2871, 2885,

 2888, 2895, 2899, 3183, 3215, 3233,

 3545, 3556, 3788, 3882, 3957, 4182,

 4691, 4706, 4761, 4800, 4863, 4871,

 5006, 5028, 5052, 5059, 5113, 5120,

 5194, 5213, 5224, 5249, 5257, 5265,

 5271, 5283, 5294, 5419, 5532, 5537,

 5602, 6638, 8898, 8915, 8959, 9044,

 9558, 10867, 10875, 10982, 10986,

 11016, 11017, 11051, 11066, 11068,

 11070, 11158, 11377, 11382, 11501,

 11559, 11560, 11608, 11631, 11653,

 11654, 11986, 11987, 12286, 12287,

 12653, 12654, 12655, 12656, 12689,

 12690, 12691, 12692, 13046, 13062,

 13571, 13601, 13688, 14319, 14520,

 14619, 15845, 16363, 16367, 16384,

 16578, 16579, 17190, 17191, 17196,

 17200, 21839, 21893, 21967, 22474,

 22773, 24492, 26246, 26396, 28762,

 28835, 29492, 30275, 30278, 32171,

 32173, 32177, 32179, 32184, 32186,

 32356, 32640, 32668, 32679, 32682
\tl_to_uppercase:n 32875
\tl_trim_spaces:N 53, 4237
\tl_trim_spaces:n 52, 400,

 719, 4237, 7621, 13490, 13492, 32687
\tl_trim_spaces_apply:nN 53, 694, 4237, 9825, 10365, 11478
\tl_upper_case:n 32977
\tl_upper_case:nn 32977

```

```

\tl_use:N 51,
 176, 180, 183, 4185, 9651, 9746, 15198
\g_tmpa_tl 59, 4710
\l_tmpa_tl 5, 45, 59,
 1206, 1208, 1225, 1308, 1310, 1314,
 1316, 1320, 1322, 1326, 1328, 4712
\g_tmpb_tl 59, 4710
\l_tmpb_tl 59, 1207,
 1208, 1223, 1225, 1309, 1310, 1315,
 1316, 1321, 1322, 1327, 1328, 4712
tl internal commands:
__tl_act:NNNn
 406, 406, 407, 408, 4211, 4463, 4530
__tl_act_count_group:n .. 4213, 4220
__tl_act_count_group:nn 4207
__tl_act_count_normal:N . 4212, 4218
__tl_act_count_normal:nN 4207
__tl_act_count_space: ... 4214, 4219
__tl_act_count_space:n 4207
__tl_act_end:w 4463
__tl_act_end:wn 399, 4498, 4502
__tl_act_group:nwNNN 4463
__tl_act_if_head_is_space:nTF ..
 407, 4463
__tl_act_if_head_is_space:w .. 4463
__tl_act_if_head_is_space_-
 true:w 4463
__tl_act_loop:w 406, 407, 4463
__tl_act_normal:NwNNN 4463
__tl_act_output:n 408, 4463
__tl_act_result:n .. .
 407, 4498, 4519, 4521, 4522, 4523, 4524
__tl_act_reverse 408
__tl_act_reverse_output:n .. .
 4463, 4539, 4541, 4543
__tl_act_space:wwNNN 407, 4463
__tl_analysis:n .. .
 964, 973, 23392, 23671, 23700, 23708
__tl_analysis_a:n .. 23396, 23421
__tl_analysis_a_bgroup:w .. .
 23452, 23474
__tl_analysis_a_cs:ww 23531
__tl_analysis_a_egroup:w .. .
 23454, 23474
__tl_analysis_a_group:nw .. 23474
__tl_analysis_a_group_aux:w .. 23474
__tl_analysis_a_group_auxii:w 23474
__tl_analysis_a_group_test:w .. 23474
__tl_analysis_a_loop:w .. 23428,
 23431, 23472, 23514, 23528, 23546
__tl_analysis_a_safe:N .. .
 23453, 23495, 23531
__tl_analysis_a_space:w 23451, 23457
__tl_analysis_a_space:w .. .
 966, 23457
__tl_analysis_a_store: .. .
 966, 23468, 23510, 23516
__tl_analysis_a_type:w .. 23432, 23433
__tl_analysis_b:n .. 23397, 23559
__tl_analysis_b_char:Nww .. .
 23586, 23592
__tl_analysis_b_cs:Nww .. 23588, 23616
__tl_analysis_b_cs_test:ww .. 23616
__tl_analysis_b_loop:w .. .
 972, 23559, 23662, 23667
__tl_analysis_b_normal:wwN .. .
 23572, 23637
__tl_analysis_b_normals:ww .. .
 970, 971, 23569, 23572, 23613, 23623
__tl_analysis_b_special:w .. .
 23575, 23634
__tl_analysis_b_special_char:wN .. .
 23634
__tl_analysis_b_special_space:w .. .
 23634
\l__tl_analysis_char_token .. .
 961, 966,
 967, 23363, 23461, 23466, 23504, 23509
__tl_analysis_cs_space_count:NN .. .
 23376, 23545, 23619
__tl_analysis_cs_space_count:w .. .
 23376
__tl_analysis_cs_space_count_-
 end:w 23376
__tl_analysis_disable:n .. .
 23401, 23423, 23489, 23542
__tl_analysis_extract_charcode: .. .
 23370, 23484
__tl_analysis_extract_charcode_-
 aux:w .. .
 23370
\l__tl_analysis_index_int .. .
 968, 969,
 23366, 23426, 23429, 23467, 23485,
 23522, 23525, 23551, 23553, 23640
__tl_analysis_map_inline_aux:Nn .. .
 23669
__tl_analysis_map_inline_-
 aux:nnn .. .
 23669
\l__tl_analysis_nesting_int .. .
 965, 23367, 23427, 23518, 23527
\l__tl_analysis_normal_int .. .
 23365, 23425, 23470, 23512,
 23523, 23526, 23543, 23552, 23557
\g__tl_analysis_result_tl .. .
 972, 23369, 23561, 23691, 23714
__tl_analysis_show: .. .
 23702, 23710, 23712

```

```

__tl_analysis_show_active:n . . .
..... 23727, 23756
__tl_analysis_show_cs:n 23723, 23756
\c__tl_analysis_show_etc_str . .
..... 975, 23776, 23778, 23783
__tl_analysis_show_long:nn . 23756
__tl_analysis_show_long-
aux:nnnn 23756, 23762
__tl_analysis_show_loop:wNw . 23712
__tl_analysis_show_normal:n . .
..... 23730, 23736
__tl_analysis_show_value:N . .
..... 23741, 23765
\l__tl_analysis_token . .
..... 961, 962, 965, 966,
967, 23363, 23373, 23432, 23436,
23439, 23442, 23490, 23494, 23509
\l__tl_analysis_type_int . .
..... 965, 968, 23368,
23435, 23450, 23518, 23520, 23524
__tl_build_begin:NN . 32375, 32420
__tl_build_begin>NNN . 1197, 32375
__tl_build_end_loop>NN 32463
__tl_build_get>NNN . .
..... 32449, 32465, 32470
__tl_build_get:w 32449
__tl_build_get_end:w 32449
__tl_build_last>NNn . .
..... 1196, 1197, 32387, 32392, 32453
__tl_build_put:nn 1197, 32392, 32444
__tl_build_put:nw 1197, 32392
__tl_build_put_left>NNn 32432
__tl_case:NnTF . .
..... 4089, 4094, 4099, 4104, 4106
__tl_case:nnTF 4086
__tl_case:Nw 4086
__tl_case_end:nw 4086
__tl_count:n 398, 4194
__tl_head_aux:n 4298, 4300
__tl_head_auxi:nw 4293
__tl_head_auxii:n 4293
__tl_head_exp_not:w 404, 4341
__tl_if_blank_p>NNw 3978
__tl_if_empty_if:n 372, 391,
392, 3965, 3981, 4075, 4079, 4320, 4459
__tl_if_head_eq_empty_arg:w . .
..... 403, 404, 4341
__tl_if_head_eq_meaning-
normal:nN 4376, 4380
__tl_if_head_eq_meaning-
special:nN 4377, 4389
__tl_if_head_is_group_fi-
false:w 4432
__tl_if_head_is_N_type_auxi:w . .
..... 405, 4412
__tl_if_head_is_N_type_auxii:n . .
..... 4427, 4430
__tl_if_head_is_N_type_auxii:mn . .
..... 4412
__tl_if_head_is_N_type_auxiii:n . .
..... 4412
__tl_if_head_is_space:w 4447
__tl_if_novalue:w 4043
__tl_if_recursion_tail_break:nN
... 409, 3764, 4129, 4158, 4172, 4566
__tl_if_recursion_tail_stop:nTF
... 3764
__tl_if_recursion_tail_stop_p:n
... 3764
__tl_if_single:nnw 394, 4061
\l__tl_internal_a_t1 . .
..... 412, 3769, 3771, 3774, 3999, 4017,
4021, 4694, 4700, 32951, 32952,
32960, 32962, 32964, 32971, 32973
\l__tl_internal_b_t1 . .
..... 3999, 4004, 4007, 4018, 4021
__tl_item:nn 4550
__tl_item_aux:nn 4550
__tl_map_function:Nn 396, 4119, 4138
__tl_map_tokens:nn 4147
__tl_map_variable:Nnn 4162
__tl_quark_if_nil:n 3765
__tl_quark_if_nil:nTF 3890
__tl_range:Nnnn 4580, 32483, 32488
__tl_range:nnNn 4580
__tl_range:nnnNn 4580
__tl_range:w 409, 4580
__tl_range_braced:w 409, 1200, 32481
__tl_range_collect:nn 1200, 4580
__tl_range_collect_braced:w . .
..... 409, 1200, 32481
__tl_range_collect_group:nN . 4580
__tl_range_collect_group:nn . .
..... 4648, 4657
__tl_range_collect_N:nN 4580
__tl_range_collect_space:nw . 4580
__tl_range_collect_unbraced:w 32481
__tl_range_items:nnNn 409
__tl_range_normalize:nn . .
..... 4595, 4599, 4659
__tl_range_skip:w 409, 4580
__tl_range_skip_spaces:n 4580
__tl_range_unbraced:w 32481
__tl_replace:NnNNNnn
..... 387, 388, 3866, 3868, 3870, 3872, 3877
__tl_replace_auxi:NnnNNNnn 388, 3877

```

\\_tl\_replace\_auxii:nNNnn . . . . .  
       . . . . . 388, 389, 3877  
\\_tl\_replace\_next:w . . . . .  
       . . . . . 387, 389, 3870, 3872, 3877  
\\_tl\_replace\_next\_aux:w . . . . . 3877  
\\_tl\_replace\_wrap:w . . . . .  
       . . . . . 387, 389, 3866, 3868, 3877  
\\_tl\_rescan>NNw 384, 3767, 3851, 3856  
\\_tl\_rescan\_aux: . . . . . 3767  
\c\_tl\_rescan\_marker\_t1 . . . . .  
       . . . . . 386, 3766, 3793, 3801, 3831, 3863  
\\_tl\_reverse\_group\_preserve:n . . . . .  
       . . . . . 4532, 4540  
\\_tl\_reverse\_group\_preserve:nn 4525  
\\_tl\_reverse\_items:nwNwn . . . . . 4221  
\\_tl\_reverse\_items:wn . . . . . 4221  
\\_tl\_reverse\_normal:N . . . . . 4531, 4538  
\\_tl\_reverse\_normal:nN . . . . . 4525  
\\_tl\_reverse\_space: . . . . . 4533, 4542  
\\_tl\_reverse\_space:n . . . . . 4525  
\\_tl\_set\_rescan:nNN . . . . .  
       . . . . . 384, 385, 3788, 3810  
\\_tl\_set\_rescan>NNnn . . . . . 384, 3767  
\\_tl\_set\_rescan\_multi:nNN . . . . .  
       . . . . . 384, 385, 386, 3767, 3818, 3840  
\\_tl\_set\_rescan\_single:nnNN . . . . .  
       . . . . . 385, 3810  
\\_tl\_set\_rescan\_single>NNww . . . . . 386  
\\_tl\_set\_rescan\_single\_aux:nnnNN . . . . .  
       . . . . . 3810  
\\_tl\_set\_rescan\_single\_aux:w . . . . .  
       . . . . . 386, 3810  
\\_tl\_show:n . . . . . 4690  
\\_tl\_show>NN . . . . . 4678  
\\_tl\_show:w . . . . . 4690  
\\_tl\_tl\_head:w . . . . . 4293, 4383  
\\_tl\_tmp:w . . . . .  
       . . . . . 393, 400, 4036, 4037, 4043,  
       4056, 4253, 4292, 4463, 4478, 4714  
\\_tl\_trim\_mark: . . . . .  
       . . . . . 399, 400, 4240, 4245, 4253  
\\_tl\_trim\_spaces:nn . . . . .  
       . . . . . 694, 4239, 4245, 4253  
\\_tl\_trim\_spaces\_auxi:w . . . . . 400, 4253  
\\_tl\_trim\_spaces\_auxii:w . . . . . 400, 4253  
\\_tl\_trim\_spaces\_auxiii:w . . . . . 400, 4253  
\\_tl\_trim\_spaces\_auxiv:w . . . . . 400, 4253  
\\_tl\_use\_none\_delimit\_by\_q\_act\_-  
       stop:w . . . . . 4463  
\\_tl\_use\_none\_delimit\_by\_s\_act\_-  
       stop:w . . . . . 4497, 4502  
token commands:  
\c\_alignment\_token . . . . . 134,  
       580, 10819, 10880, 10919, 23602, 29499  
\c\_parameter\_token . . . . .  
       . . . . . 134, 580, 1043, 10880, 10923, 10926  
\g\_peek\_token . . . . . 138, 138, 11210, 11224  
\l\_peek\_token . . . . . 138, 138, 591,  
       593, 1201, 11210, 11222, 11239,  
       11278, 11290, 11310, 11360, 11361,  
       11362, 11365, 32542, 32543, 32544  
\c\_space\_token 34, 55, 58, 134, 141,  
       274, 403, 581, 2729, 4363, 4401,  
       10828, 10880, 10943, 11239, 11362,  
       13236, 23436, 23466, 23608, 24142,  
       24177, 29471, 29508, 29635, 32544  
\token\_get\_arg\_spec:N . . . . . 32995  
\token\_get\_prefix\_spec:N . . . . . 32995  
\token\_get\_replacement\_spec:N . . . . . 32995  
\token\_if\_active:NTF 136, 10956, 30238  
\token\_if\_active\_p:N . . . . . 136, 10956,  
       13440, 29842, 30194, 30213, 31576  
\token\_if\_alignment:NTF . . . . .  
       . . . . . 135, 135, 10917  
\token\_if\_alignment\_p:N . . . . . 135, 10917  
\token\_if\_chardef:NTF . . . . .  
       . . . . . 137, 11029, 23745  
\token\_if\_chardef\_p:N . . . . .  
       . . . . . 137, 11029, 29439  
\token\_if\_cs:NTF . . . . . 136, 10993,  
       29738, 30057, 30357, 30411, 31583  
\token\_if\_cs\_p:N . . . . .  
       . . . . . 136, 10993, 29841, 30433, 30551,  
       30653, 30706, 30734, 30779, 31575  
\token\_if\_dim\_register:NTF . . . . .  
       . . . . . 137, 11029, 23747  
\token\_if\_dim\_register\_p:N 137, 11029  
\token\_if\_eq\_catcode:NNTF . . . . . 136,  
       139, 139, 139, 274, 2729, 10966  
\token\_if\_eq\_catcode\_p:NN 136, 10966  
\token\_if\_eq\_charcode:NNTF . . . . .  
       . . . . . 136, 139, 139, 140, 140,  
       274, 10971, 12548, 13236, 20438,  
       24177, 24182, 24805, 25005, 25018,  
       25020, 25058, 25180, 26150, 26229  
\token\_if\_eq\_charcode\_p:NN 136, 10971  
\token\_if\_eq\_meaning:NNTF . . . . .  
       . . . . . 136, 140, 140, 140,  
       140, 274, 2732, 2743, 10961, 13288,  
       16697, 17712, 17771, 18706, 18708,  
       18713, 18777, 18963, 21036, 24499,  
       24811, 24844, 24993, 25016, 25048,  
       25175, 25178, 26200, 26227, 26268,  
       26285, 29670, 29676, 29695, 29721,  
       29855, 30011, 30037, 31491, 31532  
\token\_if\_eq\_meaning\_p:NN . . . . .  
       . . . . . 136, 10961, 29534, 29635, 29637,  
       29641, 29651, 29652, 29659, 29660

```

\token_if_expandable:NTF
 136, 10998, 23743, 29528
\token_if_expandable_p:N
 136, 10998, 13432
\token_if_font_selection:NTF . .
 137, 11029
\token_if_font_selection_p:N . .
 137, 11029
\token_if_group_begin:NTF 135, 10902
\token_if_group_begin_p:N 135, 10902
\token_if_group_end:NTF . 135, 10907
\token_if_group_end_p:N . 135, 10907
\token_if_int_register:NTF . .
 137, 11029, 23748
\token_if_int_register_p:N 137, 11029
\token_if_letter:N 583
\token_if_letter:NTF
 136, 10946, 30157, 30209, 30575
\token_if_letter_p:N . .
 136, 10946, 30191, 30212
\token_if_long_macro:NTF . 136, 11029
\token_if_long_macro_p:N . 136, 11029
\token_if_macro:NTF . .
 136, 2195, 2204, 2213, 10976, 11153
\token_if_macro_p:N . . 136, 10976
\token_if_math_subscript:NTF . .
 135, 10936
\token_if_math_subscript_p:N . .
 135, 10936
\token_if_math_superscript:NTF . .
 135, 10930
\token_if_math_superscript_p:N . .
 135, 10930
\token_if_math_toggle:NTF 135, 10912
\token_if_math_toggle_p:N 135, 10912
\token_if_mathchardef:NTF . .
 137, 11029, 23746
\token_if_mathchardef_p:N . .
 137, 11029, 29440
\token_if_muskip_register:NTF . .
 137, 11029
\token_if_muskip_register_p:N . .
 137, 11029
\token_if_other:NTF . . 136, 10951
\token_if_other_p:N . . 136, 10951
\token_if_parameter:NTF . 135, 10922
\token_if_parameter_p:N . 135, 10922
\token_if_primitive:NTF . 138, 11078
\token_if_primitive_p:N . 138, 11078
\token_if_protected_long_-
 macro:NTF . . 137, 2626, 11029
\token_if_protected_long_macro_-
 p:N . . 137, 11029, 13439, 29532
\token_if_protected_macro:NTF . .
 136, 2625, 11029
\token_if_protected_macro_p:N . .
 136, 11029, 13438, 29532
\token_if_skip_register:NTF . .
 138, 11029, 23749
\token_if_skip_register_p:N . .
 138, 11029
\token_if_space:NTF . . 135, 10941
\token_if_space_p:N . . 135, 10941
\token_if_toks_register:NTF . .
 138, 357, 2828, 11029, 23750
\token_if_toks_register_p:N . .
 138, 11029
\token_new:Nn 32877
\token_to_meaning:N
 15, 134, 142, 582, 585, 1441,
 1457, 1896, 2198, 2207, 2216, 2834,
 2885, 10880, 10982, 11050, 11157,
 11365, 23373, 23739, 23764, 29476
\token_to_str:N 5,
 17, 60, 134, 142, 164, 325, 405,
 405, 509, 584, 766, 768, 1046, 1443,
 1457, 1457, 1621, 1630, 1662, 1685,
 1733, 1738, 1753, 1774, 1775, 1795,
 1896, 2022, 2057, 2064, 2152, 2172,
 2185, 2811, 2905, 2990, 3005, 3020,
 3027, 3053, 3062, 3113, 3179, 3200,
 3391, 3415, 3419, 3434, 3564, 3766,
 4416, 4436, 4687, 4976, 5393, 5401,
 5404, 7749, 8183, 8421, 9143, 9198,
 9549, 9664, 10390, 10736, 10740,
 10762, 10765, 10773, 10776, 10852,
 10853, 10860, 10861, 10880, 11064,
 11065, 11070, 11072, 11073, 11074,
 11075, 11076, 11744, 13108, 13109,
 13110, 13111, 13112, 13119, 13446,
 13505, 13626, 13808, 16005, 16047,
 16121, 16362, 16377, 16584, 16585,
 17074, 17075, 17104, 17271, 17322,
 17354, 17374, 17389, 17401, 17402,
 17415, 17416, 17441, 17450, 17452,
 17477, 17480, 17505, 17507, 17521,
 17537, 17555, 17625, 17635, 17636,
 17651, 17652, 17985, 18029, 18221,
 18460, 22448, 22768, 22797, 22803,
 23314, 23330, 23381, 23462, 23505,
 23535, 23584, 23595, 23597, 23599,
 23609, 23645, 23656, 23702, 23738,
 23763, 23784, 24088, 24095, 24196,
 24200, 24923, 26173, 26405, 26984,
 26986, 27147, 27737, 27954, 28905,
 29450, 29451, 29838, 29846, 29873,
 29874, 30102, 30105, 30114, 31372,
```

|                                                                                                                                                                                         |  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| 31374, 31392, 31393, 31406, 31409,<br>31413, 31416, 31572, 31580, 31610,<br>31611, 31750, 31753, 31757, 31766,<br>31818, 32639, 32667, 32679, 32682                                     |  |
| token internal commands:                                                                                                                                                                |  |
| \c__token_A_int ..... 11147, 11184                                                                                                                                                      |  |
| \__token_delimit_by_\font:w .. 11010                                                                                                                                                    |  |
| \__token_delimit_by_char":w .. 11010                                                                                                                                                    |  |
| \__token_delimit_by_count:w .. 11010                                                                                                                                                    |  |
| \__token_delimit_by_dimen:w .. 11010                                                                                                                                                    |  |
| \__token_delimit_by_macro:w .. 11010                                                                                                                                                    |  |
| \__token_delimit_by_muskip:w .. 11010                                                                                                                                                   |  |
| \__token_delimit_by_skip:w .. 11010                                                                                                                                                     |  |
| \__token_delimit_by_toks:w .. 11010                                                                                                                                                     |  |
| \__token_if_macro_p:w ..... 10976                                                                                                                                                       |  |
| \__token_if_primitive:NNw .. 11078                                                                                                                                                      |  |
| \__token_if_primitive:Nw .. 11078                                                                                                                                                       |  |
| \__token_if_primitive_loop:N .. 11078                                                                                                                                                   |  |
| \__token_if_primitive_lua:N .. 11078                                                                                                                                                    |  |
| \__token_if_primitive_nullfont:N<br>..... 11078                                                                                                                                         |  |
| \__token_if_primitive_space:w .. 11078                                                                                                                                                  |  |
| \__token_if_primitive_undefined:N<br>..... 11078                                                                                                                                        |  |
| \__token_tmp:w ..... 584, 11011,<br>11020, 11021, 11022, 11023, 11024,<br>11025, 11026, 11027, 11030, 11064,<br>11065, 11066, 11067, 11069, 11071,<br>11072, 11073, 11074, 11075, 11076 |  |
| \toks ..... 468, 11076                                                                                                                                                                  |  |
| \toksapp ..... 908                                                                                                                                                                      |  |
| \toksdef ..... 469, 23310                                                                                                                                                               |  |
| \tokspre ..... 909                                                                                                                                                                      |  |
| \tolerance ..... 470                                                                                                                                                                    |  |
| \topmark ..... 471                                                                                                                                                                      |  |
| \topmarks ..... 573                                                                                                                                                                     |  |
| \topskip ..... 472                                                                                                                                                                      |  |
| \tpack ..... 910                                                                                                                                                                        |  |
| \tracingassigns ..... 574                                                                                                                                                               |  |
| \tracingcommands ..... 473                                                                                                                                                              |  |
| \tracingfonts ..... 941                                                                                                                                                                 |  |
| \tracinggroups ..... 575                                                                                                                                                                |  |
| \tracingifs ..... 576                                                                                                                                                                   |  |
| \tracinglostchars ..... 474                                                                                                                                                             |  |
| \tracingmacros ..... 475                                                                                                                                                                |  |
| \tracingnesting ..... 577                                                                                                                                                               |  |
| \tracingonline ..... 476                                                                                                                                                                |  |
| \tracingoutput ..... 477                                                                                                                                                                |  |
| \tracingpages ..... 478                                                                                                                                                                 |  |
| \tracingparagraphs ..... 479                                                                                                                                                            |  |
| \tracingrestores ..... 480                                                                                                                                                              |  |
| \tracingscantokens ..... 578                                                                                                                                                            |  |
| \tracingstats ..... 481                                                                                                                                                                 |  |
| true ..... 218                                                                                                                                                                          |  |
| trunc ..... 214                                                                                                                                                                         |  |
| \ttfamily ..... 31647                                                                                                                                                                   |  |
| two commands:                                                                                                                                                                           |  |
| \c_thirty_two ..... 32749                                                                                                                                                               |  |
| \c_two_hundred_fifty_five .... 32753                                                                                                                                                    |  |
| \c_two_hundred_fifty_six .... 32755                                                                                                                                                     |  |
| U                                                                                                                                                                                       |  |
| \u ..... xxii, 1015,<br>29543, 31801, 31882, 31883, 31898,<br>31899, 31908, 31909, 31922, 31923,<br>31924, 31950, 31951, 31976, 31977                                                   |  |
| \uccode ..... 482                                                                                                                                                                       |  |
| \Uchar ..... 943                                                                                                                                                                        |  |
| \Ucharcat ..... 944                                                                                                                                                                     |  |
| \uchyph ..... 483                                                                                                                                                                       |  |
| \ucs ..... 1177                                                                                                                                                                         |  |
| \Udelcode ..... 945                                                                                                                                                                     |  |
| \Udelcodenum ..... 946                                                                                                                                                                  |  |
| \Udelimiter ..... 947                                                                                                                                                                   |  |
| \Udelimiterover ..... 948                                                                                                                                                               |  |
| \Udelimiterunder ..... 949                                                                                                                                                              |  |
| \Uhextensible ..... 950                                                                                                                                                                 |  |
| \Umathaccent ..... 951                                                                                                                                                                  |  |
| \Umathaxis ..... 952                                                                                                                                                                    |  |
| \Umathbinbinspacing ..... 953                                                                                                                                                           |  |
| \Umathbinclosespacing ..... 954                                                                                                                                                         |  |
| \Umathbininnerspacing ..... 955                                                                                                                                                         |  |
| \Umathbinopenspacing ..... 956                                                                                                                                                          |  |
| \Umathbinopspacing ..... 957                                                                                                                                                            |  |
| \Umathbinordspacing ..... 958                                                                                                                                                           |  |
| \Umathbinpunctspacing ..... 959                                                                                                                                                         |  |
| \Umathbinrelspacing ..... 960                                                                                                                                                           |  |
| \Umathchar ..... 961                                                                                                                                                                    |  |
| \Umathcharclass ..... 962                                                                                                                                                               |  |
| \Umathchardef ..... 963                                                                                                                                                                 |  |
| \Umathcharfam ..... 964                                                                                                                                                                 |  |
| \Umathcharnum ..... 965                                                                                                                                                                 |  |
| \Umathcharnumdef ..... 966                                                                                                                                                              |  |
| \Umathcharslot ..... 967                                                                                                                                                                |  |
| \Umathclosebinspacing ..... 968                                                                                                                                                         |  |
| \Umathcloseclosespacing ..... 969                                                                                                                                                       |  |
| \Umathcloseinnerspacing ..... 971                                                                                                                                                       |  |
| \Umathcloseopenspacing ..... 973                                                                                                                                                        |  |
| \Umathcloseopspacing ..... 974                                                                                                                                                          |  |
| \Umathcloseordspacing ..... 975                                                                                                                                                         |  |
| \Umathclosepunctspacing ..... 976                                                                                                                                                       |  |
| \Umathcloserelspacing ..... 978                                                                                                                                                         |  |
| \Umathcode ..... 979                                                                                                                                                                    |  |
| \Umathcodenum ..... 980                                                                                                                                                                 |  |
| \Umathconnectoroverlapmin ..... 981                                                                                                                                                     |  |
| \Umathfractiondelsize ..... 983                                                                                                                                                         |  |
| \Umathfractiondenomdown ..... 984                                                                                                                                                       |  |
| \Umathfractiondenomvgap ..... 986                                                                                                                                                       |  |

|                                   |      |                                     |                 |
|-----------------------------------|------|-------------------------------------|-----------------|
| \Umathfractionnumup . . . . .     | 988  | \Umathpunctordspacing . . . . .     | 1049            |
| \Umathfractionnumvgap . . . . .   | 989  | \Umathpunctpunctspacing . . . . .   | 1050            |
| \Umathfractionrule . . . . .      | 990  | \Umathpunctrelspacing . . . . .     | 1052            |
| \Umathinnerbinspacing . . . . .   | 991  | \Umathquad . . . . .                | 1053            |
| \Umathinnerclosespacing . . . . . | 992  | \Umathradicaldegreeafter . . . . .  | 1054            |
| \Umathinnerinnerspacing . . . . . | 994  | \Umathradicaldegreebefore . . . . . | 1056            |
| \Umathinneropenspacing . . . . .  | 996  | \Umathradicaldegreeraise . . . . .  | 1058            |
| \Umathinneropsspacing . . . . .   | 997  | \Umathradicalkern . . . . .         | 1060            |
| \Umathinnerordspacing . . . . .   | 998  | \Umathradicalrule . . . . .         | 1061            |
| \Umathinnerpunctspacing . . . . . | 999  | \Umathradicalvgap . . . . .         | 1062            |
| \Umathinnerrelspacing . . . . .   | 1001 | \Umathrelbinspacing . . . . .       | 1063            |
| \Umathlimitabovebgap . . . . .    | 1002 | \Umathrelclosespacing . . . . .     | 1064            |
| \Umathlimitabovekern . . . . .    | 1003 | \Umathrelinnerspacing . . . . .     | 1065            |
| \Umathlimitabovevgap . . . . .    | 1004 | \Umathrelopsspacing . . . . .       | 1066            |
| \Umathlimitbelowbgap . . . . .    | 1005 | \Umathrelopsspacing . . . . .       | 1067            |
| \Umathlimitbelowkern . . . . .    | 1006 | \Umathrelordspacing . . . . .       | 1068            |
| \Umathlimitbelowvgap . . . . .    | 1007 | \Umathrelpunctspacing . . . . .     | 1069            |
| \Umathnolimitssubfactor . . . . . | 1008 | \Umathrelrelspacing . . . . .       | 1070            |
| \Umathnolimitsupfactor . . . . .  | 1009 | \Umathskewedfractionhgap . . . . .  | 1071            |
| \Umathopbinspacing . . . . .      | 1010 | \Umathskewedfractionvgap . . . . .  | 1073            |
| \Umathopclosespacing . . . . .    | 1011 | \Umathspaceafterscript . . . . .    | 1075            |
| \Umathopenbinspacing . . . . .    | 1012 | \Umathstackdenomdown . . . . .      | 1076            |
| \Umathopenclosespacing . . . . .  | 1013 | \Umathstacknumup . . . . .          | 1077            |
| \Umathopeninnerspacing . . . . .  | 1014 | \Umathstackvgap . . . . .           | 1078            |
| \Umathopenopenspacing . . . . .   | 1015 | \Umathsubshiftdown . . . . .        | 1079            |
| \Umathopenopspacing . . . . .     | 1016 | \Umathsubshiftdrop . . . . .        | 1080            |
| \Umathopenordspacing . . . . .    | 1017 | \Umathsubsupshiftdown . . . . .     | 1081            |
| \Umathopenpunctspacing . . . . .  | 1018 | \Umathsubsupvgap . . . . .          | 1082            |
| \Umathopenrelspacing . . . . .    | 1019 | \Umathsubtopmax . . . . .           | 1083            |
| \Umathoperatorsize . . . . .      | 1020 | \Umathsupbottommin . . . . .        | 1084            |
| \Umathopinnerspacing . . . . .    | 1021 | \Umathsupshiftdrop . . . . .        | 1085            |
| \Umathopopenspacing . . . . .     | 1022 | \Umathsupshiftup . . . . .          | 1086            |
| \Umathopopspacing . . . . .       | 1023 | \Umathsupsubbottommax . . . . .     | 1087            |
| \Umathopordspacing . . . . .      | 1024 | \Umathunderbarkern . . . . .        | 1088            |
| \Umathoppunctspacing . . . . .    | 1025 | \Umathunderbarrule . . . . .        | 1089            |
| \Umathoprelspacing . . . . .      | 1026 | \Umathunderbarvgap . . . . .        | 1090            |
| \Umathordbinspacing . . . . .     | 1027 | \Umathunderdelimiterbgap . . . . .  | 1091            |
| \Umathordclosespacing . . . . .   | 1028 | \Umathunderdelimitervgap . . . . .  | 1093            |
| \Umathordinnerspacing . . . . .   | 1029 | undefined commands:                 |                 |
| \Umathordopenspacing . . . . .    | 1030 | .undefined: . . . . .               | 190, 15415      |
| \Umathordopspacing . . . . .      | 1031 | \underline . . . . .                | 484             |
| \Umathordordspacing . . . . .     | 1032 | \unexpanded . . . . .               | 579, 2713, 2737 |
| \Umathordpunctspacing . . . . .   | 1033 | \unhbox . . . . .                   | 485             |
| \Umathordrelspacing . . . . .     | 1034 | \unhcropy . . . . .                 | 486             |
| \Umathoverbarkern . . . . .       | 1035 | \uniformdeviate . . . . .           | 942             |
| \Umathoverbarrule . . . . .       | 1036 | \unkern . . . . .                   | 487             |
| \Umathoverbarvgap . . . . .       | 1037 | \unless . . . . .                   | 580             |
| \Umathoverdelimiterbgap . . . . . | 1038 | \Unosubscript . . . . .             | 1095            |
| \Umathoverdelimitervgap . . . . . | 1040 | \Unosuperscript . . . . .           | 1096            |
| \Umathpunctbinspacing . . . . .   | 1042 | \unpenalty . . . . .                | 488             |
| \Umathpunctclosespacing . . . . . | 1043 | \unskip . . . . .                   | 489             |
| \Umathpunctinnerspacing . . . . . | 1045 | \unvbox . . . . .                   | 490             |
| \Umathpunctopenspacing . . . . .  | 1047 | \unvcopy . . . . .                  | 491             |
| \Umathpuncttopspacing . . . . .   | 1048 | \Uoverdelimiter . . . . .           | 1097            |



|                                      |                                                                                                                                                                                                                                                                                        |                             |      |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------|------|
| \useboxresource . . . . .            | 935                                                                                                                                                                                                                                                                                    | \Usubscript . . . . .       | 1107 |
| \usefont . . . . .                   | 31624                                                                                                                                                                                                                                                                                  | \U superscript . . . . .    | 1108 |
| \useimageresource . . . . .          | 936                                                                                                                                                                                                                                                                                    | \U underdelimiter . . . . . | 1109 |
| \Uskewed . . . . .                   | 1100                                                                                                                                                                                                                                                                                   | \Uvextensible . . . . .     | 1110 |
| \Uskewedwithdelims . . . . .         | 1101                                                                                                                                                                                                                                                                                   |                             |      |
| \Ustack . . . . .                    | 1102                                                                                                                                                                                                                                                                                   |                             |      |
| \Ustartdisplaymath . . . . .         | 1103                                                                                                                                                                                                                                                                                   |                             |      |
| \Ustartmath . . . . .                | 1104                                                                                                                                                                                                                                                                                   |                             |      |
| \Ustopdisplaymath . . . . .          | 1105                                                                                                                                                                                                                                                                                   |                             |      |
| \Ustopmath . . . . .                 | 1106                                                                                                                                                                                                                                                                                   |                             |      |
| \Usubscript . . . . .                | 1107                                                                                                                                                                                                                                                                                   |                             |      |
|                                      |                                                                                                                                                                                                                                                                                        |                             |      |
|                                      |                                                                                                                                                                                                                                                                                        | V                           |      |
| \v . . . . .                         | 29543, 31806,<br>31892, 31893, 31894, 31895, 31904,<br>31905, 31938, 31939, 31946, 31947,<br>31958, 31959, 31966, 31967, 31970,<br>31971, 31993, 31994, 31995, 31996,<br>31997, 31998, 31999, 32000, 32001,<br>32002, 32003, 32004, 32005, 32006,<br>32007, 32010, 32011, 32020, 32021 |                             |      |
| \vadjust . . . . .                   | 493                                                                                                                                                                                                                                                                                    |                             |      |
| \valign . . . . .                    | 494                                                                                                                                                                                                                                                                                    |                             |      |
| value commands:                      |                                                                                                                                                                                                                                                                                        |                             |      |
| .value_forbidden:n . . . . .         | 190, 15417                                                                                                                                                                                                                                                                             |                             |      |
| .value_required:n . . . . .          | 190, 15417                                                                                                                                                                                                                                                                             |                             |      |
| \vbadness . . . . .                  | 495                                                                                                                                                                                                                                                                                    |                             |      |
| \vbox . . . . .                      | 496                                                                                                                                                                                                                                                                                    |                             |      |
| vbox commands:                       |                                                                                                                                                                                                                                                                                        |                             |      |
| \vbox:n . . . . .                    | 240, 240, 244, 27231                                                                                                                                                                                                                                                                   |                             |      |
| \vbox_gset:Nn . . . . .              | 244, 27245, 27806                                                                                                                                                                                                                                                                      |                             |      |
| \vbox_gset:Nw . . . . .              | 245, 27281, 27878                                                                                                                                                                                                                                                                      |                             |      |
| \vbox_gset_end: . . . . .            | 245, 27281, 27880                                                                                                                                                                                                                                                                      |                             |      |
| \vbox_gset_split_to_ht:NNn . . . . . | 245, 27320                                                                                                                                                                                                                                                                             |                             |      |
| \vbox_gset_to_ht:Nnn . . . . .       | 245, 27269                                                                                                                                                                                                                                                                             |                             |      |
| \vbox_gset_to_ht:Nnw . . . . .       | 245, 27302                                                                                                                                                                                                                                                                             |                             |      |
| \vbox_gset_top:Nn . . . . .          | 244, 27257                                                                                                                                                                                                                                                                             |                             |      |
| \vbox_set:Nn . . . . .               | 244, 245, 27245, 27800                                                                                                                                                                                                                                                                 |                             |      |
| \vbox_set:Nw . . . . .               | 245, 27281, 27871                                                                                                                                                                                                                                                                      |                             |      |
| \vbox_set_end: . . . . .             | 245, 245, 27281, 27873                                                                                                                                                                                                                                                                 |                             |      |
| \vbox_set_split_to_ht:NNn . . . . .  | 245, 27320                                                                                                                                                                                                                                                                             |                             |      |
| \vbox_set_to_ht:Nnn . . . . .        | 245, 245, 27269                                                                                                                                                                                                                                                                        |                             |      |
| \vbox_set_to_ht:Nnw . . . . .        | 245, 27302                                                                                                                                                                                                                                                                             |                             |      |
| \vbox_set_top:Nn . . . . .           | 244, 27257, 27820, 27894                                                                                                                                                                                                                                                               |                             |      |
| \vbox_to_ht:nm . . . . .             | 244, 27235                                                                                                                                                                                                                                                                             |                             |      |
| \vbox_to_zero:n . . . . .            | 244, 27235                                                                                                                                                                                                                                                                             |                             |      |
| \vbox_top:n . . . . .                | 244, 27231                                                                                                                                                                                                                                                                             |                             |      |
| \vbox_unpack:N . . . . .             | 245, 27316, 27820, 27894                                                                                                                                                                                                                                                               |                             |      |
| \vbox_unpack_clear:N . . . . .       | 32918                                                                                                                                                                                                                                                                                  |                             |      |
| \vbox_unpack_drop:N . . . . .        | 246, 27316, 32918, 32920                                                                                                                                                                                                                                                               |                             |      |
| \vccenter . . . . .                  | 497                                                                                                                                                                                                                                                                                    |                             |      |
| vcoffin commands:                    |                                                                                                                                                                                                                                                                                        |                             |      |
| \vcoffin_gset:Nnn . . . . .          | 251, 27797                                                                                                                                                                                                                                                                             |                             |      |
| \vcoffin_gset:Nnw . . . . .          | 251, 27869                                                                                                                                                                                                                                                                             |                             |      |
| \vcoffin_gset_end: . . . . .         | 251, 27869                                                                                                                                                                                                                                                                             |                             |      |
| \vcoffin_set:Nnn . . . . .           | 251, 27797                                                                                                                                                                                                                                                                             |                             |      |
| \vcoffin_set:Nnw . . . . .           | 251, 27869                                                                                                                                                                                                                                                                             |                             |      |
| \vcoffin_set_end: . . . . .          | 251, 27869                                                                                                                                                                                                                                                                             |                             |      |
| \vfii . . . . .                      | 1169                                                                                                                                                                                                                                                                                   |                             |      |
| \vfii . . . . .                      | 498                                                                                                                                                                                                                                                                                    |                             |      |

|                                     |                     |                                     |                 |
|-------------------------------------|---------------------|-------------------------------------|-----------------|
| \vfill . . . . .                    | 499                 | \XeTeXglyphindex . . . . .          | 741             |
| \vfilneg . . . . .                  | 500                 | \XeTeXglyphname . . . . .           | 742             |
| \vfuzz . . . . .                    | 501                 | \XeTeXinputencoding . . . . .       | 743             |
| \voffset . . . . .                  | 502                 | \XeTeXinputnormalization . . . . .  | 744             |
| \vpack . . . . .                    | 911                 | \XeTeXinterchartokenstate . . . . . | 746             |
| \vrule . . . . .                    | 503                 | \XeTeXinterchartok . . . . .        | 748             |
| \vsize . . . . .                    | 504                 | \XeTeXisdefaultselector . . . . .   | 749             |
| \vskip . . . . .                    | 505                 | \XeTeXisexclusivefeature . . . . .  | 751             |
| \vsplit . . . . .                   | 506                 | \XeTeXlastfontchar . . . . .        | 753             |
| \vss . . . . .                      | 507                 | \XeTeXlinebreaklocale . . . . .     | 755             |
| \vtop . . . . .                     | 508                 | \XeTeXlinebreakpenalty . . . . .    | 756             |
| <b>W</b>                            |                     |                                     |                 |
| \wd . . . . .                       | 509                 | \XeTeXOTcountfeatures . . . . .     | 757             |
| \widowpenalties . . . . .           | 581                 | \XeTeXOTcountlanguages . . . . .    | 758             |
| \widowpenalty . . . . .             | 510                 | \XeTeXOTcountsscripts . . . . .     | 759             |
| \write . . . . .                    | 511                 | \XeTeXOTfeaturetag . . . . .        | 760             |
| <b>X</b>                            |                     |                                     |                 |
| \xdef . . . . .                     | 512                 | \XeTeXOTlanguagetag . . . . .       | 761             |
| xetex commands:                     |                     |                                     |                 |
| \xetex_if_engine:TF . . . . .       | 32881, 32883, 32885 | \XeTeXOTscripttag . . . . .         | 762             |
| \xetex_if_engine_p: . . . . .       | 32879               | \XeTeXpdffile . . . . .             | 763             |
| \XeTeXcharclass . . . . .           | 719                 | \XeTeXpdfpagecount . . . . .        | 764             |
| \XeTeXcharglyph . . . . .           | 720                 | \XeTeXpicfile . . . . .             | 765             |
| \XeTeXcountfeatures . . . . .       | 721                 | \XeTeXrevision . . . . .            | 766             |
| \XeTeXcountglyphs . . . . .         | 722                 | \XeTeXselectorname . . . . .        | 767             |
| \XeTeXcountselectors . . . . .      | 723                 | \XeTeXtracingfonts . . . . .        | 768             |
| \XeTeXcountvariations . . . . .     | 724                 | \XeTeXupwardsmode . . . . .         | 769             |
| \XeTeXdashbreakstate . . . . .      | 726                 | \XeTeXuseglyphmetrics . . . . .     | 770             |
| \XeTeXdefaultencoding . . . . .     | 725                 | \XeTeXvariation . . . . .           | 771             |
| \XeTeXfeaturecode . . . . .         | 727                 | \XeTeXvariationdefault . . . . .    | 772             |
| \XeTeXfeaturename . . . . .         | 728                 | \XeTeXvariationmax . . . . .        | 773             |
| \XeTeXfindfeaturebyname . . . . .   | 729                 | \XeTeXvariationmin . . . . .        | 774             |
| \XeTeXfindselectorbyname . . . . .  | 731                 | \XeTeXvariationname . . . . .       | 775             |
| \XeTeXfindvariationbyname . . . . . | 733                 | \XeTeXversion . . . . .             | 776             |
| \XeTeXfirstfontchar . . . . .       | 735                 | \xkanjiskip . . . . .               | 1165            |
| \XeTeXfonttype . . . . .            | 736                 | \xleaders . . . . .                 | 513             |
| \XeTeXgenerateactualtext . . . . .  | 737                 | \xspaceskip . . . . .               | 514             |
| \XeTeXglyph . . . . .               | 739                 | \xspcode . . . . .                  | 1166            |
| \XeTeXglyphbounds . . . . .         | 740                 | <b>Y</b>                            |                 |
|                                     |                     | \ybaselineshift . . . . .           | 1167            |
|                                     |                     | \year . . . . .                     | 515, 1327, 9688 |
|                                     |                     | \yoko . . . . .                     | 1168            |