

The L^AT_EX3 Sources

The L^AT_EX3 Project*

February 10, 2017

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	3
3	Formal language conventions which apply generally	5
4	<code>TeX</code> concepts not supported by <code>LaTeX3</code>	5
II	The <code>l3bootstrap</code> package: Bootstrap code	6
1	Using the <code>LaTeX3</code> modules	6
1.1	Internal functions and variables	7
III	The <code>l3names</code> package: Namespace for primitives	8
1	Setting up the <code>LaTeX3</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	Using or removing tokens and arguments	17
4.1	Selecting tokens from delimited arguments	18
5	Predicates and conditionals	19
5.1	Tests on control sequences	20
5.2	Primitive conditionals	20
6	Internal kernel functions	21
V	The <code>l3expan</code> package: Argument expansion	24

1	Defining new variants	24
2	Methods for defining variants	25
3	Introducing the variants	25
4	Manipulating the first argument	26
5	Manipulating two arguments	28
6	Manipulating three arguments	28
7	Unbraced expansion	29
8	Preventing expansion	30
9	Controlled expansion	31
10	Internal functions and variables	32
VI	The <code>l3prg</code> package: Control structures	34
1	Defining a set of conditional functions	34
2	The boolean data type	36
3	Boolean expressions	37
4	Logical loops	38
5	Producing multiple copies	39
6	Detecting $\mathrm{T}_{\mathrm{E}}\mathrm{X}$'s mode	39
7	Primitive conditionals	40
8	Internal programming functions	40
VII	The <code>l3quark</code> package: Quarks	42
1	Introduction to quarks and scan marks	42
1.1	Quarks	42
2	Defining quarks	42
3	Quark tests	43
4	Recursion	43
5	An example of recursion with quarks	44
6	Internal quark functions	45

7	Scan marks	45
VIII	The l3token package: Token manipulation	47
1	Creating character tokens	47
2	Manipulating and interrogating character tokens	48
3	Generic tokens	51
4	Converting tokens	52
5	Token conditionals	52
6	Peeking ahead at the next token	55
7	Decomposing a macro definition	58
8	Description of all possible tokens	59
9	Internal functions	61
IX	The l3int package: Integers	62
1	Integer expressions	62
2	Creating and initialising integers	63
3	Setting and incrementing integers	64
4	Using integers	64
5	Integer expression conditionals	65
6	Integer expression loops	66
7	Integer step functions	68
8	Formatting integers	68
9	Converting from other formats to integers	70
10	Viewing integers	71
11	Constant integers	72
12	Scratch integers	72
13	Primitive conditionals	73
14	Internal functions	73

X	The <code>l3skip</code> package: Dimensions and skips	75
1	Creating and initialising <code>dim</code> variables	75
2	Setting <code>dim</code> variables	76
3	Utilities for dimension calculations	76
4	Dimension expression conditionals	77
5	Dimension expression loops	79
6	Using <code>dim</code> expressions and variables	80
7	Viewing <code>dim</code> variables	82
8	Constant dimensions	82
9	Scratch dimensions	82
10	Creating and initialising <code>skip</code> variables	82
11	Setting <code>skip</code> variables	83
12	<code>Skip</code> expression conditionals	84
13	Using <code>skip</code> expressions and variables	84
14	Viewing <code>skip</code> variables	84
15	Constant skips	85
16	Scratch skips	85
17	Inserting skips into the output	85
18	Creating and initialising <code>muskip</code> variables	85
19	Setting <code>muskip</code> variables	86
20	Using <code>muskip</code> expressions and variables	87
21	Viewing <code>muskip</code> variables	87
22	Constant muskips	87
23	Scratch muskips	87
24	Primitive conditional	88
25	Internal functions	88
XI	The <code>l3tl</code> package: Token lists	89

1	Creating and initialising token list variables	89
2	Adding data to token list variables	90
3	Modifying token list variables	91
4	Reassigning token list category codes	91
5	Token list conditionals	92
6	Mapping to token lists	94
7	Using token lists	96
8	Working with the content of token lists	96
9	The first token from a token list	98
10	Using a single item	100
11	Viewing token lists	100
12	Constant token lists	101
13	Scratch token lists	101
14	Internal functions	101
XII	The l3str package: Strings	102
1	Building strings	102
2	Adding data to string variables	103
2.1	String conditionals	103
3	Working with the content of strings	105
4	String manipulation	108
5	Viewing strings	109
6	Constant token lists	110
7	Scratch strings	110
7.1	Internal string functions	110
XIII	The l3seq package: Sequences and stacks	112
1	Creating and initialising sequences	112
2	Appending data to sequences	113

3	Recovering items from sequences	113
4	Recovering values from sequences with branching	114
5	Modifying sequences	115
6	Sequence conditionals	116
7	Mapping to sequences	116
8	Using the content of sequences directly	118
9	Sequences as stacks	119
10	Sequences as sets	120
11	Constant and scratch sequences	121
12	Viewing sequences	122
13	Internal sequence functions	122
XIV	The <code>l3clist</code> package: Comma separated lists	123
1	Creating and initialising comma lists	123
2	Adding data to comma lists	124
3	Modifying comma lists	125
4	Comma list conditionals	126
5	Mapping to comma lists	126
6	Using the content of comma lists directly	128
7	Comma lists as stacks	129
8	Using a single item	130
9	Viewing comma lists	130
10	Constant and scratch comma lists	131
XV	The <code>l3prop</code> package: Property lists	132
1	Creating and initialising property lists	132
2	Adding entries to property lists	133
3	Recovering values from property lists	133

4	Modifying property lists	134
5	Property list conditionals	134
6	Recovering values from property lists with branching	135
7	Mapping to property lists	135
8	Viewing property lists	136
9	Scratch property lists	136
10	Constants	137
11	Internal property list functions	137
XVI	The l3box package: Boxes	138
1	Creating and initialising boxes	138
2	Using boxes	139
3	Measuring and setting box dimensions	139
4	Box conditionals	140
5	The last box inserted	140
6	Constant boxes	141
7	Scratch boxes	141
8	Viewing box contents	141
9	Horizontal mode boxes	141
10	Vertical mode boxes	143
11	Primitive box conditionals	144
XVII	The l3coffins package: Coffin code layer	146
1	Creating and initialising coffins	146
2	Setting coffin content and poles	146
3	Joining and using coffins	147
4	Measuring coffins	148
5	Coffin diagnostics	148
5.1	Constants and variables	149

XVIII	The <code>l3color</code> package: Color support	150
1	Color in boxes	150
XIX	The <code>l3msg</code> package: Messages	151
1	Creating new messages	151
2	Contextual information for messages	152
3	Issuing messages	153
4	Redirecting messages	155
5	Low-level message functions	156
6	Kernel-specific functions	157
7	Expandable errors	159
8	Internal <code>l3msg</code> functions	159
XX	The <code>l3keys</code> package: Key–value interfaces	161
1	Creating keys	162
2	Sub-dividing keys	166
3	Choice and multiple choice keys	166
4	Setting keys	169
5	Handling of unknown keys	169
6	Selective key setting	170
7	Utility functions for keys	171
8	Low-level interface for parsing key–val lists	171
XXI	The <code>l3file</code> package: File and I/O operations	173
1	File operation functions	173
1.1	Input–output stream management	174
1.2	Reading from files	175

2	Writing to files	176
2.1	Wrapping lines in output	178
2.2	Constant input-output streams	179
2.3	Primitive conditionals	179
2.4	Internal file functions and variables	179
2.5	Internal input-output functions	179
XXII	The l3fp package: floating points	181
1	Creating and initialising floating point variables	182
2	Setting floating point variables	182
3	Using floating point numbers	183
4	Floating point conditionals	184
5	Floating point expression loops	186
6	Some useful constants, and scratch variables	187
7	Floating point exceptions	188
8	Viewing floating points	189
9	Floating point expressions	189
9.1	Input of floating point numbers	189
9.2	Precedence of operators	190
9.3	Operations	191
10	Disclaimer and roadmap	197
XXIII	The l3sort package: Sorting functions	200
1	Controlling sorting	200
XXIV	The l3candidates package: Experimental additions to l3kernel	201
1	Important notice	201
2	Additions to l3basics	201
3	Additions to l3box	202
3.1	Affine transformations	202
3.2	Viewing part of a box	203
4	Additions to l3clist	204
5	Additions to l3coffins	204

6	Additions to l3file	205
7	Additions to l3fp	207
8	Additions to l3int	207
9	Additions to l3keys	207
10	Additions to l3msg	207
11	Additions to l3prg	208
12	Additions to l3prop	209
13	Additions to l3seq	210
14	Additions to l3skip	211
15	Additions to l3tl	212
16	Additions to l3tokens	216
XXV	The l3sys package: System/runtime functions	217
1	The name of the job	217
2	Date and time	217
	2.1 Engine	217
	2.2 Output format	218
XXVI	The l3luatex package: LuaTeX-specific functions	219
1	Breaking out to Lua	219
	1.1 T _E X code interfaces	219
	1.2 Lua interfaces	220
XXVII	The l3drivers package: Drivers	221
1	Box clipping	221
2	Box rotation and scaling	221
3	Color support	222

4	Drawing	222
4.1	Path construction	223
4.2	Stroking and filling	223
4.3	Stroke options	224
4.4	Color	225
4.5	Inserting \TeX material	226
4.6	Coordinate system transformations	226
XXVIII	Implementation	226
1	l3bootstrap implementation	226
1.1	Format-specific code	226
1.2	The <code>\pdfstrcmp</code> primitive in \XeTeX	227
1.3	Loading support Lua code	227
1.4	Engine requirements	228
1.5	Extending allocators	230
1.6	Character data	230
1.7	The \LaTeX3 code environment	232
2	l3names implementation	233
3	l3basics implementation	254
3.1	Renaming some \TeX primitives (again)	254
3.2	Defining some constants	257
3.3	Defining functions	257
3.4	Selecting tokens	258
3.5	Gobbling tokens from input	259
3.6	Conditional processing and definitions	259
3.7	Dissecting a control sequence	265
3.8	Exist or free	267
3.9	Defining and checking (new) functions	268
3.10	More new definitions	271
3.11	Copying definitions	273
3.12	Undefining functions	273
3.13	Generating parameter text from argument count	273
3.14	Defining functions from a given number of arguments	274
3.15	Using the signature to define functions	275
3.16	Checking control sequence equality	277
3.17	Diagnostic functions	278
3.18	Doing nothing functions	278
3.19	Breaking out of mapping functions	278

4	l3expan implementation	279
4.1	General expansion	279
4.2	Hand-tuned definitions	283
4.3	Definitions with the automated technique	285
4.4	Last-unbraced versions	286
4.5	Preventing expansion	287
4.6	Controlled expansion	288
4.7	Defining function variants	289
5	l3prg implementation	295
5.1	Primitive conditionals	295
5.2	Defining a set of conditional functions	295
5.3	The boolean data type	295
5.4	Boolean expressions	298
5.5	Logical loops	303
5.6	Producing multiple copies	304
5.7	Detecting T _E X’s mode	305
5.8	Internal programming functions	306
6	l3quark implementation	307
6.1	Quarks	307
6.2	Scan marks	310
7	l3token implementation	311
7.1	Manipulating and interrogating character tokens	311
7.2	Creating character tokens	313
7.3	Generic tokens	317
7.4	Token conditionals	318
7.5	Peeking ahead at the next token	326
7.6	Decomposing a macro definition	331
8	l3int implementation	331
8.1	Integer expressions	332
8.2	Creating and initialising integers	334
8.3	Setting and incrementing integers	336
8.4	Using integers	336
8.5	Integer expression conditionals	337
8.6	Integer expression loops	340
8.7	Integer step functions	342
8.8	Formatting integers	343
8.9	Converting from other formats to integers	349
8.10	Viewing integer	351
8.11	Constant integers	352
8.12	Scratch integers	353

9	l3skip implementation	353
9.1	Length primitives renamed	353
9.2	Creating and initialising <code>dim</code> variables	354
9.3	Setting <code>dim</code> variables	355
9.4	Utilities for dimension calculations	355
9.5	Dimension expression conditionals	356
9.6	Dimension expression loops	358
9.7	Using <code>dim</code> expressions and variables	359
9.8	Viewing <code>dim</code> variables	360
9.9	Constant dimensions	361
9.10	Scratch dimensions	361
9.11	Creating and initialising <code>skip</code> variables	361
9.12	Setting <code>skip</code> variables	362
9.13	Skip expression conditionals	363
9.14	Using <code>skip</code> expressions and variables	363
9.15	Inserting skips into the output	364
9.16	Viewing <code>skip</code> variables	364
9.17	Constant skips	364
9.18	Scratch skips	364
9.19	Creating and initialising <code>muskip</code> variables	365
9.20	Setting <code>muskip</code> variables	366
9.21	Using <code>muskip</code> expressions and variables	366
9.22	Viewing <code>muskip</code> variables	367
9.23	Constant muskips	367
9.24	Scratch muskips	367
10	l3tl implementation	367
10.1	Functions	367
10.2	Constant token lists	369
10.3	Adding to token list variables	369
10.4	Reassigning token list category codes	372
10.5	Modifying token list variables	375
10.6	Token list conditionals	379
10.7	Mapping to token lists	383
10.8	Using token lists	384
10.9	Working with the contents of token lists	385
10.10	Token by token changes	387
10.11	The first token from a token list	389
10.12	Using a single item	394
10.13	Viewing token lists	394
10.14	Scratch token lists	395
10.15	Deprecated functions	395

11	l3str implementation	395
11.1	Creating and setting string variables	395
11.2	String comparisons	396
11.3	Accessing specific characters in a string	399
11.4	Counting characters	403
11.5	The first character in a string	405
11.6	String manipulation	406
11.7	Viewing strings	408
11.8	Unicode data for case changing	408
12	l3seq implementation	411
12.1	Allocation and initialisation	412
12.2	Appending data to either end	415
12.3	Modifying sequences	416
12.4	Sequence conditionals	418
12.5	Recovering data from sequences	419
12.6	Mapping to sequences	423
12.7	Using sequences	425
12.8	Sequence stacks	425
12.9	Viewing sequences	426
12.10	Scratch sequences	427
13	l3clist implementation	427
13.1	Allocation and initialisation	427
13.2	Removing spaces around items	429
13.3	Adding data to comma lists	430
13.4	Comma lists as stacks	431
13.5	Modifying comma lists	433
13.6	Comma list conditionals	435
13.7	Mapping to comma lists	436
13.8	Using comma lists	439
13.9	Using a single item	440
13.10	Viewing comma lists	442
13.11	Scratch comma lists	442
14	l3prop implementation	442
14.1	Allocation and initialisation	443
14.2	Accessing data in property lists	444
14.3	Property list conditionals	448
14.4	Recovering values from property lists with branching	450
14.5	Mapping to property lists	450
14.6	Viewing property lists	451

15	l3box implementation	451
15.1	Creating and initialising boxes	451
15.2	Measuring and setting box dimensions	453
15.3	Using boxes	453
15.4	Box conditionals	453
15.5	The last box inserted	454
15.6	Constant boxes	454
15.7	Scratch boxes	454
15.8	Viewing box contents	455
15.9	Horizontal mode boxes	456
15.10	Vertical mode boxes	457
16	l3coffins Implementation	458
16.1	Coffins: data structures and general variables	458
16.2	Basic coffin functions	460
16.3	Measuring coffins	464
16.4	Coffins: handle and pole management	464
16.5	Coffins: calculation of pole intersections	467
16.6	Aligning and typesetting of coffins	470
16.7	Coffin diagnostics	474
16.8	Messages	479
17	l3color Implementation	480
18	l3msg implementation	481
18.1	Creating messages	481
18.2	Messages: support functions and text	483
18.3	Showing messages: low level mechanism	484
18.4	Displaying messages	486
18.5	Kernel-specific functions	493
18.6	Expandable errors	498
18.7	Showing variables	499
19	l3keys Implementation	502
19.1	Low-level interface	502
19.2	Constants and variables	506
19.3	The key defining mechanism	508
19.4	Turning properties into actions	510
19.5	Creating key properties	515
19.6	Setting keys	519
19.7	Utilities	524
19.8	Messages	526

20	l3file implementation	527
20.1	File operations	527
20.2	Input operations	532
20.2.1	Variables and constants	532
20.2.2	Stream management	533
20.2.3	Reading input	536
20.3	Output operations	537
20.3.1	Variables and constants	537
20.4	Stream management	538
20.4.1	Deferred writing	539
20.4.2	Immediate writing	539
20.4.3	Special characters for writing	540
20.4.4	Hard-wrapping lines to a character count	541
20.5	Messages	546
20.6	Deprecated functions	547
21	l3fp implementation	547
22	l3fp-aux implementation	547
22.1	Internal representation	547
22.2	Internal storage of floating points numbers	548
22.3	Using arguments and semicolons	549
22.4	Constants, and structure of floating points	549
22.5	Overflow, underflow, and exact zero	551
22.6	Expanding after a floating point number	552
22.7	Packing digits	553
22.8	Decimate (dividing by a power of 10)	555
22.9	Functions for use within primitive conditional branches	557
22.10	Integer floating points	558
22.11	Small integer floating points	559
22.12	Length of a floating point array	560
22.13	x-like expansion expandably	560
22.14	Messages	561
23	l3fp-traps Implementation	561
23.1	Flags	561
23.2	Traps	562
23.3	Errors	565
23.4	Messages	566
24	l3fp-round implementation	566
24.1	Rounding tools	566
24.2	The round function	570

25	l3fp-parse implementation	573
25.1	Work plan	573
25.1.1	Storing results	575
25.1.2	Precedence and infix operators	576
25.1.3	Prefix operators, parentheses, and functions	579
25.1.4	Numbers and reading tokens one by one	579
25.2	Main auxiliary functions	581
25.3	Helpers	582
25.4	Parsing one number	583
25.4.1	Numbers: trimming leading zeros	588
25.4.2	Number: small significand	590
25.4.3	Number: large significand	592
25.4.4	Number: beyond 16 digits, rounding	593
25.4.5	Number: finding the exponent	596
25.5	Constants, functions and prefix operators	599
25.5.1	Prefix operators	599
25.5.2	Constants	601
25.5.3	Functions	602
25.6	Main functions	604
25.7	Infix operators	605
25.7.1	Closing parentheses and commas	607
25.7.2	Usual infix operators	608
25.7.3	Juxtaposition	609
25.7.4	Multi-character cases	610
25.7.5	Ternary operator	611
25.7.6	Comparisons	611
25.8	Candidate: defining new l3fp functions	614
25.9	Messages	615
26	l3fp-logic Implementation	616
26.1	Syntax of internal functions	616
26.2	Existence test	616
26.3	Comparison	616
26.4	Floating point expression loops	618
26.5	Extrema	621
26.6	Boolean operations	623
26.7	Ternary operator	623

27	l3fp-basics Implementation	624
27.1	Common to several operations	625
27.2	Addition and subtraction	625
27.2.1	Sign, exponent, and special numbers	626
27.2.2	Absolute addition	628
27.2.3	Absolute subtraction	630
27.3	Multiplication	635
27.3.1	Signs, and special numbers	635
27.3.2	Absolute multiplication	636
27.4	Division	638
27.4.1	Signs, and special numbers	638
27.4.2	Work plan	639
27.4.3	Implementing the significand division	642
27.5	Square root	647
27.6	Setting the sign	654
28	l3fp-extended implementation	654
28.1	Description of fixed point numbers	654
28.2	Helpers for numbers with extended precision	655
28.3	Multiplying a fixed point number by a short one	656
28.4	Dividing a fixed point number by a small integer	656
28.5	Adding and subtracting fixed points	657
28.6	Multiplying fixed points	658
28.7	Combining product and sum of fixed points	659
28.8	Extended-precision floating point numbers	662
28.9	Dividing extended-precision numbers	664
28.10	Inverse square root of extended precision numbers	667
28.11	Converting from fixed point to floating point	669
29	l3fp-expo implementation	671
29.1	Logarithm	671
29.1.1	Work plan	671
29.1.2	Some constants	672
29.1.3	Sign, exponent, and special numbers	672
29.1.4	Absolute ln	672
29.2	Exponential	680
29.2.1	Sign, exponent, and special numbers	680
29.3	Power	684

30	l3fp-trig Implementation	690
30.1	Direct trigonometric functions	690
30.1.1	Filtering special cases	691
30.1.2	Distinguishing small and large arguments	694
30.1.3	Small arguments	694
30.1.4	Argument reduction in degrees	695
30.1.5	Argument reduction in radians	696
30.1.6	Computing the power series	702
30.2	Inverse trigonometric functions	705
30.2.1	Arctangent and arccotangent	706
30.2.2	Arcsine and arccosine	710
30.2.3	Arccosecant and arcsecant	712
31	l3fp-convert implementation	714
31.1	Trimming trailing zeros	714
31.2	Scientific notation	714
31.3	Decimal representation	716
31.4	Token list representation	717
31.5	Formatting	718
31.6	Convert to dimension or integer	718
31.7	Convert from a dimension	719
31.8	Use and eval	720
31.9	Convert an array of floating points to a comma list	720
32	l3fp-random Implementation	721
32.1	Engine support	721
32.2	Random floating point	723
32.3	Random integer	723
33	l3fp-assign implementation	725
33.1	Assigning values	725
33.2	Updating values	726
33.3	Showing values	727
33.4	Some useful constants and scratch variables	727
34	l3sort implementation	727
34.1	Variables	727
34.2	Finding available \toks registers	728
34.3	Protected user commands	730
34.4	Merge sort	733
34.5	Expandable sorting	736
34.6	Messages	740
34.7	Deprecated functions	742

35	l3candidates Implementation	742
35.1	Additions to l3basics	742
35.2	Additions to l3box	743
35.3	Affine transformations	743
35.4	Viewing part of a box	750
35.5	Additions to l3clist	753
35.6	Additions to l3coffins	753
35.7	Rotating coffins	753
35.8	Resizing coffins	758
35.9	Coffin diagnostics	760
35.10	Additions to l3file	760
35.11	Additions to l3fp-assign	762
35.12	Additions to l3int	762
35.13	Additions to l3keys	764
35.14	Additions to l3msg	764
35.15	Additions to l3prg	765
35.16	Additions to l3prop	766
35.17	Additions to l3seq	767
35.18	Additions to l3skip	769
35.19	Additions to l3tl	770
35.19.1	Unicode case changing	772
35.20	Additions to l3tokens	795
36	l3sys implementation	796
36.1	The name of the job	796
36.2	Time and date	797
36.3	Detecting the engine	797
36.4	Detecting the output	798
37	l3luatex implementation	799
37.1	Breaking out to Lua	799
37.2	Messages	799
37.3	Lua functions for internal use	800
37.4	Format mode code: font loader	800
38	l3drivers Implementation	802
38.1	pdfmode driver	802
38.1.1	Basics	802
38.1.2	Color	803
38.2	dvipdfmx driver	804
38.2.1	Basics	804
38.2.2	Color	805
38.3	xdvipdfmx driver	805
38.3.1	Color	805
38.4	Common code for PDF production	806
38.4.1	Box operations	806
38.5	Drawing	807
38.6	dvips driver	812
38.6.1	Basics	812
38.7	Driver-specific auxiliaries	812

38.7.1	Box operations	813
38.7.2	Color	814
38.8	Drawing	814
38.9	dvisvgm driver	821
38.9.1	Basics	821
38.10	Driver-specific auxiliaries	821
38.10.1	Box operations	822
38.10.2	Color	824
38.11	Drawing	824

Index	832
--------------	------------

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:

- D** The **D** specifier means *do not use*. All of the T_EX 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!
- 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 in general *not* expandable, unless specifically noted.

- f** The **f** specifier stands for *full expansion*, and in contrast to **x** stops at the first non-expandable item (reading the argument from left to right) without trying to expand it. For example, when setting a token list variable (a macro used for storage), the sequence

```
\tl_set:Nn \l_my_a_tl { A }
\tl_set:Nn \l_my_b_tl { B }
\tl_set:Nf \l_my_a_tl { \l_my_a_tl \l_my_b_tl }
```

will leave `\l_my_a_tl` with the content `A\l_my_b_tl`, as `A` cannot be expanded and so terminates expansion before `\l_my_b_tl` 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 arbitrary string).

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:

bool Either true or false.

box Box register.

clist Comma separated list.

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

dim “Rigid” lengths.

fp floating-point values;

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.

int Integer-valued count register.

prop Property list.

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

skip “Rubber” lengths.

stream An input or output stream (for reading from or writing to, respectively).

tl Token list variables: placeholder for a token list.

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 in truth one and the same. 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 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
```

```
\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:

<code>\seq_new:N</code>	<code>\seq_new:N</code> $\langle sequence \rangle$
<code>\seq_new:c</code>	

When a number of variants are described, the arguments are usually illustrated only for the base function. Here, $\langle sequence \rangle$ 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 argument (in plain T_EX terms, inside an `\edef`), as well as within an **f**-type argument. These fully expandable functions are indicated in the documentation by a star:

<code>\cs_to_str:N</code> ★	<code>\cs_to_str:N</code> $\langle cs \rangle$
-----------------------------	--

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 $\langle cs \rangle$, shorthand for a $\langle control\ sequence \rangle$.

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:

<code>\seq_map_function:NN</code> ☆	<code>\seq_map_function:NN</code> $\langle seq \rangle$ $\langle function \rangle$
-------------------------------------	--

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:

<code>\xetex_if_engine:TF</code> ★	<code>\xetex_if_engine:TF</code> $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
------------------------------------	---

The underlining and italic of TF indicates that `\xetex_if_engine:T`, `\xetex_if_engine:F` and `\xetex_if_engine:TF` are all available. Usually, the illustration will use the TF variant, and so both $\langle true\ code \rangle$ and $\langle false\ code \rangle$ will be shown. The two variant forms T and F take only $\langle true\ code \rangle$ and $\langle false\ code \rangle$, 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:

<code>\l_tmpa_tl</code>	
-------------------------	--

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 “**T_EXhackers note**” section:

<code>\token_to_str:N</code> ★	<code>\token_to_str:N</code> $\langle token \rangle$
--------------------------------	--

The normal description text.

T_EXhackers 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 $\text{\LaTeX}3$ 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 $\langle true\ code \rangle$ or the $\langle false\ code \rangle$ 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 \TeX concepts not supported by $\text{\LaTeX}3$

The \TeX concept of an “`\outer`” macro is *not supported* at all by $\text{\LaTeX}3$. As such, the functions provided here may break when used on top of $\text{\LaTeX}2_\epsilon$ if `\outer` tokens are used in the arguments.

Part II

The l3bootstrap 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_ε and are loaded all as one with the usual `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions. These modules will also form the basis of the L^AT_EX3 format, but work in this area is incomplete and not included in this documentation at present.

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

`\ExplSyntaxOn`
`\ExplSyntaxOff`

Updated: 2011-08-13

`\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`

`\RequirePackage{expl3}`
`\ProvidesExplPackage` *<package>* *<date>* *<version>* *<description>*

These functions act broadly in the same way as the corresponding L^AT_EX 2_ε 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_ε provides in turning on `\makeatletter` within package and class code.) The *<date>* should be given in the format *<year>/<month>/<day>*.

`\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 alike are loaded with usual L^AT_EX 2_ε 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}
```

1.1 Internal functions and variables

`\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 `\ExplSyntaxOn/\ExplSyntaxOff`.

Part III

The l3names 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;
- switches to the category code régime for programming;
- provides support settings for building the code as a T_EX format.

This module is entirely dedicated to primitives, 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_YT_EX and LuaT_EX should be consulted for details of the primitives. These are named based on the engine which first introduced them:

`\tex_...` Introduced by T_EX itself;
`\etex_...` Introduced by the ϵ -T_EX extensions;
`\pdftex_...` Introduced by pdfT_EX;
`\xetex_...` Introduced by X_YT_EX;
`\luatex_...` Introduced by LuaT_EX;
`\utex_...` Introduced by X_YT_EX and LuaT_EX;
`\ptex_...` Introduced by pT_EX;
`\uptex_...` Introduced by upT_EX.

Part IV

The l3basics 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`**`\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 will be 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). The later will be 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 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” will be 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 will be 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 will result 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 will 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 will 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 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

<code>\cs_new:Npn</code>	<code>\cs_new:Npn <function> <parameters> {<code>}</code>
<code>\cs_new:cpn</code>	Creates <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the
<code>\cs_new:Npx</code>	<i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_new:cpx</code>	definition is global and an error will result if the <i><function></i> is already defined.

<code>\cs_new_nopar:Npn</code>	<code>\cs_new_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_nopar:cpn</code>	Creates <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the
<code>\cs_new_nopar:Npx</code>	<i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. When the
<code>\cs_new_nopar:cpx</code>	<i><function></i> is used the <i><parameters></i> absorbed cannot contain <code>\par</code> tokens. The definition
	is global and an error will result if the <i><function></i> is already defined.

<code>\cs_new_protected:Npn</code>	<code>\cs_new_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_protected:cpn</code>	Creates <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the
<code>\cs_new_protected:Npx</code>	<i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_new_protected:cpx</code>	<i><function></i> will not expand within an x-type argument. The definition is global and an
	error will result if the <i><function></i> is already defined.

<code>\cs_new_protected_nopar:Npn</code>	<code>\cs_new_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_protected_nopar:cpn</code>	
<code>\cs_new_protected_nopar:Npx</code>	
<code>\cs_new_protected_nopar:cpx</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 argument. The definition is global and an error will result if the *<function>* is already defined.

<code>\cs_set:Npn</code>	<code>\cs_set:Npn <function> <parameters> {<code>}</code>
<code>\cs_set:cpn</code>	Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the
<code>\cs_set:Npx</code>	<i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_set:cpx</code>	assignment of a meaning to the <i><function></i> is restricted to the current \TeX group level.

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

<code>\cs_set_protected:Npn</code>	<code>\cs_set_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_protected:cpn</code>	Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the
<code>\cs_set_protected:Npx</code>	<i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_set_protected:cpx</code>	assignment of a meaning to the <i><function></i> is restricted to the current \TeX group level.
	The <i><function></i> will not expand within an x-type argument.

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

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

<code>\cs_gset:Npn</code>	<code>\cs_gset:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset:cpn</code>	
<code>\cs_gset:Npx</code>	
<code>\cs_gset:cpx</code>	

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

<code>\cs_gset_nopar:Npn</code>	<code>\cs_gset_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_nopar:cpn</code>	
<code>\cs_gset_nopar:Npx</code>	
<code>\cs_gset_nopar:cpx</code>	

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

<code>\cs_gset_protected:Npn</code>	<code>\cs_gset_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_protected:cpn</code>	
<code>\cs_gset_protected:Npx</code>	
<code>\cs_gset_protected:cpx</code>	

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

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

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

3.3 Defining new functions using the signature

<code>\cs_new:Nn</code>	<code>\cs_new:Nn <function> {<code>}</code>
<code>\cs_new:(cn Nx cx)</code>	

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

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

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

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

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

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

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

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

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

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

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

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

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ 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 $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ 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 $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. The $\langle function \rangle$ will not expand within an x-type argument. The assignment of a meaning to the $\langle function \rangle$ 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 $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The $\langle function \rangle$ will not expand within an x-type argument. The assignment of a meaning to the $\langle function \rangle$ 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 $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ 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 $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ 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 $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The $\langle function \rangle$ will not expand within an x-type argument. The assignment of a meaning to the $\langle function \rangle$ 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 $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The $\langle function \rangle$ will not expand within an x-type argument. The assignment of a meaning to the $\langle function \rangle$ is global.

<code>\cs_generate_from_arg_count:NNnn</code>	<code>\cs_generate_from_arg_count:NNnn <function> <creator> <number></code>
<code>\cs_generate_from_arg_count:(cNnn Ncnn)</code>	<code><code></code>

Updated: 2012-01-14

Uses the $\langle creator \rangle$ function (which should have signature Npn , for example `\cs_new:Npn`) to define a $\langle function \rangle$ which takes $\langle number \rangle$ arguments and has $\langle code \rangle$ as replacement text. The $\langle number \rangle$ of arguments is an integer expression, evaluated as detailed for `\int_eval:n`.

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 $\langle control\ sequence_1 \rangle$ and sets it to have the same meaning as $\langle control\ sequence_2 \rangle$ or $\langle token \rangle$. 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 $\langle control\ sequence_1 \rangle$ to have the same meaning as $\langle control\ sequence_2 \rangle$ (or $\langle token \rangle$). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the $\langle control\ sequence_1 \rangle$ 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 $\langle control\ sequence_1 \rangle$ to have the same meaning as $\langle control\ sequence_2 \rangle$ (or $\langle token \rangle$). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the $\langle control\ sequence_1 \rangle$ 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 $\langle control\ sequence \rangle$ 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 $\langle control\ sequence \rangle$ control sequence. This will show the $\langle replacement\ text \rangle$ for a macro.

\TeX hackers note: This is \TeX ’s `\meaning` primitive. The `c` variant correctly reports undefined arguments.

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

Updated: 2015-08-03

`\cs_show:N` $\langle control\ sequence \rangle$

Displays the definition of the $\langle control\ sequence \rangle$ on the terminal.

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

3.7 Converting to and from control sequences

`\use:c` ★ `\use:c` $\{\langle control\ sequence\ name \rangle\}$

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

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` $\langle control\ sequence \rangle$

`\cs_if_exist_use:NTF` $\langle control\ sequence \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests whether the $\langle control\ sequence \rangle$ is currently defined (whether as a function or another control sequence type), and if it is inserts the $\langle control\ sequence \rangle$ into the input stream followed by the $\langle true\ code \rangle$. Otherwise the $\langle false\ code \rangle$ is used.

`\cs:w` ★ `\cs:w` $\langle control\ sequence\ name \rangle$ `\cs_end:`

`\cs_end:` ★

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

T_EXhackers note: These are the T_EX 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 sequence will *not* include the current escape token, cf. `\token_to_str:N`. Full expansion of this function requires exactly 2 expansion steps, and so an x-type expansion, or two o-type expansions will be required to convert the *<control sequence>* to a sequence of characters in the input stream. In most cases, an f-expansion will be correct as well, but this loses a space at the start of the result.

4 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 in absorbing them the outer set will be 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 {<group₁>}
\use:nn ★ \use:nn {<group₁>} {<group₂>}
\use:nnn ★ \use:nnn {<group₁>} {<group₂>} {<group₃>}
\use:nnnn ★ \use:nnnn {<group₁>} {<group₂>} {<group₃>} {<group₄>}

```

As illustrated, these functions will absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument will be removed leaving the remaining tokens in the input stream. The category code of these tokens will also be 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 } }
```

will result in the input stream containing

```
abc { def }
```

i.e. only the outer braces will be removed.

```

\use_i:nn ★ \use_i:nn {<arg₁>} {<arg₂>}
\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 will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

<hr/>		
<code>\use_i:nnn</code>	★	<code>\use_i:nnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩}</code>
<code>\use_ii:nnn</code>	★	These functions absorb three arguments from the input stream. The function <code>\use_i:nnn</code> discards the second and third arguments, and leaves the content of the first argument in the input stream. <code>\use_ii:nnn</code> and <code>\use_iii:nnn</code> work similarly, leaving the content of second or third arguments in the input stream, respectively. The category code of these tokens will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.
<code>\use_iii:nnn</code>	★	

<hr/>		
<code>\use_i:nnnn</code>	★	<code>\use_i:nnnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩} {⟨arg₄⟩}</code>
<code>\use_ii:nnnn</code>	★	These functions absorb four arguments from the input stream. The function <code>\use_i:nnnn</code> discards the second, third and fourth arguments, and leaves the content of the first argument in the input stream. <code>\use_ii:nnnn</code> , <code>\use_iii:nnnn</code> and <code>\use_iv:nnnn</code> work similarly, leaving the content of second, third or fourth arguments in the input stream, respectively. The category code of these tokens will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.
<code>\use_iii:nnnn</code>	★	
<code>\use_iv:nnnn</code>	★	

<hr/>		
<code>\use_i_ii:nnn</code>	★	<code>\use_i_ii:nnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩}</code>
<hr/>		
		This functions will absorb three arguments and leave the content of the first and second in the input stream. The category code of these tokens will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect. An example:

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

will result in the input stream containing

`abc { def }`

i.e. the outer braces will be removed and the third group will be removed.

<hr/>		
<code>\use_none:n</code>	★	<code>\use_none:n {⟨group₁⟩}</code>
<code>\use_none:nn</code>	★	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 <code>n</code> arguments may be an unbraced single token (<i>i.e.</i> an <code>N</code> argument).
<code>\use_none:nnn</code>	★	
<code>\use_none:nnnn</code>	★	
<code>\use_none:nnnnn</code>	★	
<code>\use_none:nnnnnn</code>	★	
<code>\use_none:nnnnnnn</code>	★	
<code>\use_none:nnnnnnnn</code>	★	
<hr/>		

<hr/>		
<code>\use:x</code>		<code>\use:x {⟨expandable tokens⟩}</code>
<hr/>		

<code>Updated: 2011-12-31</code>		Fully expands the <i>⟨expandable tokens⟩</i> and inserts the result into the input stream at the current location. Any hash characters (<code>#</code>) in the argument must be doubled.
----------------------------------	--	--

4.1 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></code> <code>\q_recursion_stop</code>

Absorb the *<balanced text>* form 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</code> <code>text> \q_stop</code> <code>\use_i_delimit_by_q_recursion_stop:nw {<inserted tokens></code> <code><balanced text> \q_recursion_stop</code>

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

5 Predicates and conditionals

L^AT_EX3 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 will turn 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 carry 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 will usually be 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 will 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 will be 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_tl || \cs_if_free_p:N \g_tmpz_tl
} {\true code} {\false code}

```

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

Primitive conditionals There is a third variety of conditional, which is the original concept used in plain $\text{T}_{\text{E}}\text{X}$ and $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X } 2_{\varepsilon}$. 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.

```

\c_true_bool
\c_false_bool

```

Constants that represent `true` and `false`, respectively. Used to implement predicates.

5.1 Tests on control sequences

```

\cs_if_eq_p:NN * \cs_if_eq_p:NN {\cs_1} {\cs_2}
\cs_if_eq:NNTF * \cs_if_eq:NNTF {\cs_1} {\cs_2} {\true code} {\false code}

```

Compares the definition of two $\langle\textit{control sequence}\rangle$ and is logically `true` the same, *i.e.* if they have exactly the same definition when examined with `\cs_show:N`.

```

\cs_if_exist_p:N * \cs_if_exist_p:N \langle\textit{control sequence}\rangle
\cs_if_exist_p:c * \cs_if_exist:NNTF \langle\textit{control sequence}\rangle {\true code} {\false code}
\cs_if_exist:NNTF *
\cs_if_exist:cTF *

```

Tests whether the $\langle\textit{control sequence}\rangle$ is currently defined (whether as a function or another control sequence type). Any valid definition of $\langle\textit{control sequence}\rangle$ will evaluate as `true`.

```

\cs_if_free_p:N * \cs_if_free_p:N \langle\textit{control sequence}\rangle
\cs_if_free_p:c * \cs_if_free:NNTF \langle\textit{control sequence}\rangle {\true code} {\false code}
\cs_if_free:NNTF *
\cs_if_free:cTF *

```

Tests whether the $\langle\textit{control sequence}\rangle$ is currently free to be defined. This test will be `false` if the $\langle\textit{control sequence}\rangle$ currently exists (as defined by `\cs_if_exist:N`).

5.2 Primitive conditionals

The $\varepsilon\text{-T}_{\text{E}}\text{X}$ engine itself provides many different conditionals. Some expand whatever comes after them and others don’t. Hence the names for these underlying functions will often contain 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 will prefix primitive conditionals with `\if_`.

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

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

<code>\if_meaning:w</code>	★	<code>\if_meaning:w <arg₁> <arg₂> <true code> \else: <false code> \fi:</code>
----------------------------	---	---

`\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`.

<code>\if:w</code>	★	<code>\if:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_charcode:w</code>	★	<code>\if_catcode:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_catcode:w</code>	★	These conditionals will expand any following tokens until two unexpandable tokens are left. If you wish to prevent this expansion, prefix the token in question with <code>\exp_not:N</code> . <code>\if_catcode:w</code> tests if the category codes of the two tokens are the same whereas <code>\if:w</code> tests if the character codes are identical. <code>\if_charcode:w</code> is an alternative name for <code>\if:w</code> .

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

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.

<code>\if_mode_horizontal:</code>	★	<code>\if_mode_horizontal: <true code> \else: <false code> \fi:</code>
<code>\if_mode_vertical:</code>	★	Execute <i><true code></i> if currently in horizontal mode, otherwise execute <i><false code></i> . Similar for the other functions.
<code>\if_mode_math:</code>	★	
<code>\if_mode_inner:</code>	★	

6 Internal kernel functions

<code>__chk_if_exist_cs:N</code>	<code>__chk_if_exist_cs:N <cs></code>	
<code>__chk_if_exist_cs:c</code>		This function checks that <i><cs></i> exists according to the criteria for <code>\cs_if_exist_p:N</code> , and if not raises a kernel-level error.

<code>__chk_if_free_cs:N</code>	<code>__chk_if_free_cs:N <cs></code>	
<code>__chk_if_free_cs:c</code>		This function checks that <i><cs></i> is free according to the criteria for <code>\cs_if_free_p:N</code> , and if not raises a kernel-level error.

<hr/> <code>__chk_if_exist_var:N</code> <hr/>	<code>__chk_if_exist_var:N <var></code> This function checks that <i><var></i> is defined according to the criteria for <code>\cs_if_free_p:N</code> , and if not raises a kernel-level error. This function is only created if the package option <code>check-declarations</code> is active.
<hr/> <code>__chk_log:x</code> <hr/>	<code>__chk_log:x {<message text>}</code> If the <code>log-functions</code> option is active, this function writes the <i><message text></i> to the log file using <code>\iow_log:x</code> . Otherwise, the <i><message text></i> is ignored using <code>\use_none:n</code> .
<hr/> <code>__chk_suspend_log:</code> <code>__chk_resume_log:</code> <hr/>	<code>__chk_suspend_log: ... __chk_log:x ... __chk_resume_log:</code> Any <code>__chk_log:x</code> command between <code>__chk_suspend_log:</code> and <code>__chk_resume_log:</code> is suppressed. These commands can be nested.
<hr/> <code>__cs_count_signature:N *</code> <code>__cs_count_signature:c *</code> <hr/>	<code>__cs_count_signature:N <function></code> Splits the <i><function></i> into the <i><name></i> (<i>i.e.</i> the part before the colon) and the <i><signature></i> (<i>i.e.</i> after the colon). The <i><number></i> of tokens in the <i><signature></i> is then left in the input stream. If there was no <i><signature></i> then the result is the marker value <code>-1</code> .
<hr/> <code>__cs_split_function:NN *</code> <hr/>	<code>__cs_split_function:NN <function> <processor></code> Splits the <i><function></i> into the <i><name></i> (<i>i.e.</i> the part before the colon) and the <i><signature></i> (<i>i.e.</i> after the colon). This information is then placed in the input stream after the <i><processor></i> function in three parts: the <i><name></i> , the <i><signature></i> and a logic token indicating if a colon was found (to differentiate variables from function names). The <i><name></i> will not include the escape character, and both the <i><name></i> and <i><signature></i> are made up of tokens with category code 12 (other). The <i><processor></i> should be a function with argument specification <code>:nnN</code> (plus any trailing arguments needed).
<hr/> <code>__cs_get_function_name:N *</code> <hr/>	<code>__cs_get_function_name:N <function></code> Splits the <i><function></i> into the <i><name></i> (<i>i.e.</i> the part before the colon) and the <i><signature></i> (<i>i.e.</i> after the colon). The <i><name></i> is then left in the input stream without the escape character present made up of tokens with category code 12 (other).
<hr/> <code>__cs_get_function_signature:N *</code> <hr/>	<code>__cs_get_function_signature:N <function></code> Splits the <i><function></i> into the <i><name></i> (<i>i.e.</i> the part before the colon) and the <i><signature></i> (<i>i.e.</i> after the colon). The <i><signature></i> is then left in the input stream made up of tokens with category code 12 (other).
<hr/> <code>__cs_tmp:w</code> <hr/>	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).
<hr/> <code>__kernel_register_show:N</code> <code>__kernel_register_show:c</code> <hr/>	<code>__kernel_register_show:N <register></code> Used to show the contents of a T _E X register at the terminal, formatted such that internal parts of the mechanism are not visible.

<code>_prg_case_end:nw</code> ★	<code>_prg_case_end:nw {<code>} <tokens> \q_mark {<true code>} \q_mark {<false code>} \q_stop</code>
------------------------------------	--

Used to terminate case statements (`\int_case:nnTF`, *etc.*) by removing trailing *<tokens>* and the end marker `\q_stop`, inserting the *<code>* for the successful case (if one is found) and either the `true code` or `false code` for the over all outcome, as appropriate.

Part V

The l3expan package

Argument expansion

This module provides generic methods for expanding T_EX 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 L^AT_EX3 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 from the `\exp_` module. 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` will expand the content of third argument once before any expansion of the first and second arguments. If `\seq_gpush:No` was not defined it could be coded in the following way:

```
\exp_args:NNo \seq_gpush:Nn
  \g_file_name_stack
  \l_tmpa_tl
```

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:No`

```
\cs_new:Npn \seq_gpush:No { \exp_args:NNo \seq_gpush:Nn }
```

Providing variants in this way in style files is uncritical 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

`\cs_generate_variant:Nn`

Updated: 2015-08-06

`\cs_generate_variant:Nn` \langle parent control sequence \rangle $\{$ \langle variant argument specifiers \rangle $\}$

This function is used to define argument-specifier variants of the \langle parent control sequence \rangle for L^AT_EX3 code-level macros. The \langle parent control sequence \rangle is first separated into the \langle base name \rangle and \langle original argument specifier \rangle . The comma-separated list of \langle variant argument specifiers \rangle is then used to define variants of the \langle original argument specifier \rangle where these are not already defined. For each \langle variant \rangle given, a function is created which will expand its arguments as detailed and pass them to the \langle parent control sequence \rangle . So for example

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

will create a new function `\foo:cn` which will expand its first argument into a control sequence name and pass the result to `\foo:Nn`. Similarly

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

would generate the functions `\foo:NV` and `\foo:cV` in the same way. The `\cs_generate_variant:Nn` function can only be applied if the \langle parent control sequence \rangle is already defined. Only `n` and `N` arguments can be changed to other types. If the \langle parent control sequence \rangle is protected or if the \langle variant \rangle involves `x` arguments, then the \langle variant control sequence \rangle will also be protected. The \langle variant \rangle is created globally, as is any `\exp_args:N \langle variant \rangle` function needed to carry out the expansion.

3 Introducing the variants

The available internal functions for argument expansion come in two flavours, some of them are faster than others. Therefore (when speed is important) it is usually best to follow the following guidelines when defining new functions that are supposed to come with variant forms:

- Arguments that might need expansion should come first in the list of arguments to make processing faster.
- Arguments that should consist of single tokens should come first.
- Arguments that need full expansion (*i.e.*, are denoted with `x`) should be avoided if possible as they can not be processed expandably, *i.e.*, functions of this type will not work correctly in arguments that are themselves subject to `x` expansion.
- In general, unless in the last position, multi-token arguments `n`, `f`, and `o` will need special processing when more than one argument is being expanded. This special processing is not fast. 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`, with possible trailing `N` or `n`, which are not expanded.

The `V` type returns the value of a register, which can be one of `tl`, `int`, `skip`, `dim`, `toks`, 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 `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 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 } }
```

will result in the call `\example:n { 3 , \int_eval:n { 3 + 4 } }` while using `\example:x` instead results in `\example:n { 3 , 7 }` at the cost of being protected. If you use this type of expansion in conditional processing then you should stick to using `TF` type functions only as it does not try to 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 *emph*first non-expandable token. This means for example that both

```
\tl_set:N0 \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.

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.

<hr/> <code>\exp_args:No</code> ★ <hr/>	<code>\exp_args:No</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$...
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded once, and the result is inserted in braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.
<hr/> <code>\exp_args:Nc</code> ★ <hr/> <code>\exp_args:cc</code> ★ <hr/>	<code>\exp_args:Nc</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded until only characters remain, and are then turned into a control sequence. (An internal error will occur if such a conversion is not possible). The result is inserted into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged. The <code>:cc</code> variant constructs the $\langle function \rangle$ name in the same manner as described for the $\langle tokens \rangle$.
<hr/> <code>\exp_args:Nv</code> ★ <hr/>	<code>\exp_args:Nv</code> $\langle function \rangle$ $\langle variable \rangle$
	This function absorbs two arguments (the names of the $\langle function \rangle$ and the $\langle variable \rangle$). The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.
<hr/> <code>\exp_args:Nv</code> ★ <hr/>	<code>\exp_args:Nv</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded until only characters remain, and are then turned into a control sequence. (An internal error will occur if such a conversion is not possible). This control sequence should be the name of a $\langle variable \rangle$. The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.
<hr/> <code>\exp_args:Nf</code> ★ <hr/>	<code>\exp_args:Nf</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are fully expanded until the first non-expandable token or space is found, and the result is inserted in braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.
<hr/> <code>\exp_args:Nx</code> <hr/>	<code>\exp_args:Nx</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$) and exhaustively expands the $\langle tokens \rangle$ second. The result is inserted in braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.

5 Manipulating two arguments

<hr/>	
<code>\exp_args:NNo</code> *	<code>\exp_args:NNo <token> {<tokens>}</code>
<code>\exp_args:Nnc</code> *	
<code>\exp_args:Nnv</code> *	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.
<code>\exp_args:NNf</code> *	
<code>\exp_args:Nco</code> *	
<code>\exp_args:Ncf</code> *	
<code>\exp_args:Ncc</code> *	
<code>\exp_args:NVV</code> *	
<hr/>	
<code>\exp_args:Nno</code> *	<code>\exp_args:Nno <token> {<tokens>}</code>
<code>\exp_args:NnV</code> *	
<code>\exp_args:Nnf</code> *	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 special (slower) processing.
<code>\exp_args:Noo</code> *	
<code>\exp_args:Nof</code> *	
<code>\exp_args:Noc</code> *	
<code>\exp_args:Nff</code> *	
<code>\exp_args:Nfo</code> *	
<code>\exp_args:Nnc</code> *	
<hr/>	
Updated: 2012-01-14	
<hr/>	
<code>\exp_args:NNx</code>	<code>\exp_args:NNx <token> {<tokens>}</code>
<code>\exp_args:Nnx</code>	
<code>\exp_args:Ncx</code>	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.
<code>\exp_args:Nox</code>	
<code>\exp_args:Nxo</code>	
<code>\exp_args:Nxx</code>	
<hr/>	

6 Manipulating three arguments

<hr/>	
<code>\exp_args:NNNo</code> *	<code>\exp_args:NNNo <token> {<tokens>}</code>
<code>\exp_args:NNNV</code> *	
<code>\exp_args:Nccc</code> *	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, <i>etc.</i>
<code>\exp_args:NcNc</code> *	
<code>\exp_args:NcNo</code> *	
<code>\exp_args:Ncco</code> *	
<hr/>	
<code>\exp_args:NNoo</code> *	<code>\exp_args:NNoo <token> {<tokens>}</code>
<code>\exp_args:NNno</code> *	
<code>\exp_args:Nnno</code> *	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, <i>etc.</i> These functions need special (slower) processing.
<code>\exp_args:Nnnc</code> *	
<code>\exp_args:Nooo</code> *	
<hr/>	

<code>\exp_args:NNNx</code>	<code>\exp_args:NNnx</code>	<code>\langle token_1 \rangle \langle token_2 \rangle \{\langle tokens_1 \rangle\} \{\langle tokens_2 \rangle\}</code>
<code>\exp_args:NNnx</code>		
<code>\exp_args:NNox</code>		
<code>\exp_args:Nnnx</code>		
<code>\exp_args:Nnox</code>		
<code>\exp_args:Noox</code>		
<code>\exp_args:Ncnx</code>		
<code>\exp_args:Nccx</code>		

New: 2015-08-12

7 Unbraced expansion

<code>\exp_last_unbraced:Nv</code>	★	<code>\exp_last_unbraced:Nno</code>	<code>\langle token \rangle \langle tokens_1 \rangle \langle tokens_2 \rangle</code>
<code>\exp_last_unbraced:(Nf No Nv)</code>	★		
<code>\exp_last_unbraced:Nco</code>	★		
<code>\exp_last_unbraced:(NcV NNV NNo)</code>	★		
<code>\exp_last_unbraced:Nno</code>	★		
<code>\exp_last_unbraced:(Noo Nfo)</code>	★		
<code>\exp_last_unbraced:NNNV</code>	★		
<code>\exp_last_unbraced:NNNo</code>	★		
<code>\exp_last_unbraced:NnNo</code>	★		

Updated: 2012-02-12

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`, and `:Nfo` variants need special (slower) processing.

T_EXhackers note: As an optimization, the last argument is unbraced by some of those functions before expansion. This can cause problems if the argument is empty: for instance, `\exp_last_unbraced:Nf \foo_bar:w { } \q_stop` leads to an infinite loop, as the quark is `f`-expanded.

<code>\exp_last_unbraced:Nx</code>	<code>\exp_last_unbraced:Nx</code>	<code>\langle function \rangle \{\langle tokens \rangle\}</code>
------------------------------------	------------------------------------	--

This functions fully expands the `\langle tokens \rangle` and leaves the result in the input stream after reinsertion of `\langle function \rangle`. This function is not expandable.

<code>\exp_last_two_unbraced:Noo</code>	★	<code>\exp_last_two_unbraced:Noo</code>	<code>\langle token \rangle \langle tokens_1 \rangle \{\langle tokens_2 \rangle\}</code>
---	---	---	--

This function absorbs three arguments and expand 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`** $\langle token_1 \rangle$ $\langle token_2 \rangle$

Carries out a single expansion of $\langle token_2 \rangle$ (which may consume arguments) prior to the expansion of $\langle token_1 \rangle$. If $\langle token_2 \rangle$ is a \TeX primitive, it will be executed rather than expanded, while if $\langle token_2 \rangle$ has not expansion (for example, if it is a character) then it will be left unchanged. It is important to notice that $\langle token_1 \rangle$ may be *any* single token, including group-opening and -closing tokens ($\{$ or $\}$ assuming normal \TeX category codes). Unless specifically required, expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_arg:N` function.

\TeX hackers 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 'expandable' since they themselves will not appear after the expansion has completed.

`\exp_not:N` ★**`\exp_not:N`** $\langle token \rangle$

Prevents expansion of the $\langle token \rangle$ in a context where it would otherwise be expanded, for example an \mathbf{x} -type argument.

\TeX hackers note: This is the \TeX `\noexpand` primitive.

`\exp_not:c` ★**`\exp_not:c`** $\{\langle tokens \rangle\}$

Expands the $\langle tokens \rangle$ until only unexpandable content remains, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited.

`\exp_not:n` ★**`\exp_not:n`** $\{\langle tokens \rangle\}$

Prevents expansion of the $\langle tokens \rangle$ in a context where they would otherwise be expanded, for example an \mathbf{x} -type argument.

\TeX hackers note: This is the ε - \TeX `\unexpanded` primitive. Hence its argument *must* be surrounded by braces.

`\exp_not:V` ★**`\exp_not:V`** $\langle variable \rangle$

Recovers the content of the $\langle variable \rangle$, then prevents expansion of this material in a context where it would otherwise be expanded, for example an \mathbf{x} -type argument.

`\exp_not:v` ★**`\exp_not:v`** $\{\langle tokens \rangle\}$

Expands the $\langle tokens \rangle$ until only unexpandable content 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 is prevented in a context where it would otherwise be expanded, for example an \mathbf{x} -type argument.

<hr/> <code>\exp_not:o</code> ★	<code>\exp_not:o {⟨tokens⟩}</code>
	Expands the <i>⟨tokens⟩</i> once, then prevents any further expansion in a context where they would otherwise be expanded, for example an <i>x</i> -type argument.
<hr/> <code>\exp_not:f</code> ★	<code>\exp_not:f {⟨tokens⟩}</code>
	Expands <i>⟨tokens⟩</i> fully until the first unexpandable token is found. Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion.
<hr/> <code>\exp_stop_f:</code> ★	<code>\foo_bar:f { ⟨tokens⟩ \exp_stop_f: ⟨more tokens⟩ }</code>
<hr/> Updated: 2011-06-03 <hr/>	This function terminates an <i>f</i> -type expansion. Thus if a function <code>\foo_bar:f</code> starts an <i>f</i> -type expansion and all of <i>⟨tokens⟩</i> are expandable <code>\exp_stop_f:</code> will terminate the expansion of tokens even if <i>⟨more tokens⟩</i> are also expandable. The function itself is an implicit space token. Inside an <i>x</i> -type expansion, it will retain its form, but when typeset it produces the underlying space (␣).

9 Controlled expansion

The `expl3` language makes all efforts to hide the complexity of \TeX 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 \TeX 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. You will find these commands 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 *⟨expandable-tokens⟩* as that will break badly if unexpandable tokens are encountered in that place!

<hr/> <code>\exp:w</code> ★	<code>\exp:w ⟨expandable-tokens⟩ \exp_end:</code>
<code>\exp_end:</code> ★	Expands <i>⟨expandable-tokens⟩</i> until reaching <code>\exp_end:</code> at which point expansion stops.
<hr/> New: 2015-08-23 <hr/>	The full expansion of <i>⟨expandable-tokens⟩</i> has to be empty. If any token in <i>⟨expandable-tokens⟩</i> or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result <code>\exp_end:</code> will be misinterpreted later on. ²
	In typical use cases the <code>\exp_end:</code> will be hidden somewhere in the replacement text of <i>⟨expandable-tokens⟩</i> rather than being on the same expansion level than <code>\exp:w</code> , 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.

<code>\exp:w</code>	★
<code>\exp_end_continue_f:w</code>	★

New: 2015-08-23

`\exp:w` $\langle expandable-tokens \rangle$ `\exp_end_continue_f:w` $\langle further-tokens \rangle$

Expands $\langle expandable-tokens \rangle$ until reaching `\exp_end_continue_f:w` at which point expansion continues as an f-type expansion expanding $\langle further-tokens \rangle$ 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 will get removed.

The full expansion of $\langle expandable-tokens \rangle$ has to be empty. If any token in $\langle expandable-tokens \rangle$ 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.³

In typical use cases $\langle expandable-tokens \rangle$ 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` $\langle expandable-tokens \rangle$ `\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` $\langle expandable-tokens \rangle$ `\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.

<code>\exp:w</code>	★
<code>\exp_end_continue_f:nw</code>	★

New: 2015-08-23

`\exp:w` $\langle expandable-tokens \rangle$ `\exp_end_continue_f:nw` $\langle further-tokens \rangle$

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

10 Internal functions and variables

`\l__exp_internal_tl`

The `\exp_` module has its private variables to temporarily store results of the argument expansion. This is done to avoid interference with other functions using temporary variables.

²Due to the implementation you might get the character in position 0 in the current font (typically “‘”) in the output without any error message!

³In this particular case you may get a character into the output as well as an error message.

```

\::n \cs_set:Npn \exp_args:Ncof { \::c \::o \::f \::: }
\::N
\::p Internal forms for the base expansion types. These names do not conform to the general
\::c LATEX3 approach as this makes them more readily visible in the log and so forth.
\::o
\::f
\::x
\::v
\::V
\:::

```

Part VI

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.

T_EXhackers 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 will 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_new_protected_conditional:Npnn \<name>:<arg spec> <parameters>
\prg_set_protected_conditional:Npnn {\<conditions>} {\<code>}
\prg_new_protected_conditional:Nnn \prg_new_protected_conditional:Nnn \<name>:<arg spec>
\prg_set_protected_conditional:Nnn {\<conditions>} {\<code>}
```

Updated: 2012-02-06

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 will check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the **set** version will 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:

- `\<name>_p:<arg spec>` — 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.
- `\<name>:<arg spec>T` — a function with one more argument than the original `<arg spec>` demands. The `<true branch>` code in this additional argument will be left on the input stream only if the test is `true`.
- `\<name>:<arg spec>F` — a function with one more argument than the original `<arg spec>` demands. The `<false branch>` code in this additional argument will be left on the input stream only if the test is `false`.
- `\<name>:<arg spec>TF` — a function with two more argument than the original `<arg spec>` demands. The `<true branch>` code in the first additional argument will be left on the input stream if the test is `true`, while the `<false branch>` code in the second argument will be left on the input stream if the test is `false`.

The `<code>` of the test may use `<parameters>` as specified by the second argument to `\prg_set_conditional:Npnn`: this should match the `<argument specification>` 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 `<code>`, the functions `\prg_return_true:` and `\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_tl #1
  \prg_return_true:
\else:
  \if_meaning:w \l_tmpa_tl #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 `<conditions>` 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.

<code>\prg_new_eq_conditional:Nnn</code>	<code>\prg_new_eq_conditional:Nnn \<name1>:<arg spec1> \<name2>:<arg spec2></code>
<code>\prg_set_eq_conditional:Nnn</code>	<code>{<conditions>}</code>

These functions copy a family of conditionals. The `new` version will check for existing definitions (cf. `\cs_new:Npn`) whereas the `set` version will not (cf. `\cs_set:Npn`). The conditionals copied are depended on the comma-separated list of `<conditions>`, which should be one or more of `p`, `T`, `F` and `TF`.

<code>\prg_return_true:</code>	★	<code>\prg_return_true:</code>
<code>\prg_return_false:</code>	★	<code>\prg_return_false:</code>

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:Npnn`, *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.

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 will generally mean being constructed from predicate functions, possibly nested).

T_EXhackers note: The `bool` data type is not implemented using the `\iffalse/\iftrue` primitives, in contrast to `\newif`, *etc.*, in plain T_EX, L^AT_EX 2_ε and so on. Programmers should not base use of `bool` switches on any particular expectation of the implementation.

<code>\bool_new:N</code>	<code>\bool_new:N <boolean></code>
<code>\bool_new:c</code>	

Creates a new `<boolean>` or raises an error if the name is already taken. The declaration is global. The `<boolean>` will initially be `false`.

<code>\bool_set_false:N</code>	<code>\bool_set_false:N <boolean></code>
<code>\bool_set_false:c</code>	
<code>\bool_gset_false:N</code>	Sets <code><boolean></code> logically <code>false</code> .
<code>\bool_gset_false:c</code>	

<code>\bool_set_true:N</code>	<code>\bool_set_true:N <boolean></code>
<code>\bool_set_true:c</code>	
<code>\bool_gset_true:N</code>	Sets <code><boolean></code> logically <code>true</code> .
<code>\bool_gset_true:c</code>	

<code>\bool_set_eq:NN</code>	<code>\bool_set_eq:NN</code> $\langle boolean_1 \rangle$ $\langle boolean_2 \rangle$
<code>\bool_set_eq:(cN Nc cc)</code>	Sets the content of $\langle boolean_1 \rangle$ equal to that of $\langle boolean_2 \rangle$.
<code>\bool_gset_eq:NN</code>	
<code>\bool_gset_eq:(cN Nc cc)</code>	
<hr/>	
<code>\bool_set:Nn</code>	<code>\bool_set:Nn</code> $\langle boolean \rangle$ $\{\langle boolexpr \rangle\}$
<code>\bool_set:cn</code>	Evaluates the $\langle boolean \text{ expression} \rangle$ as described for <code>\bool_if:nTF</code> , and sets the $\langle boolean \rangle$
<code>\bool_gset:Nn</code>	variable to the logical truth of this evaluation.
<code>\bool_gset:cn</code>	
<hr/>	
Updated: 2012-07-08	
<hr/>	
<code>\bool_if_p:N</code> *	<code>\bool_if_p:N</code> $\langle boolean \rangle$
<code>\bool_if_p:c</code> *	<code>\bool_if:N</code> TF $\langle boolean \rangle$ $\{\langle true \text{ code} \rangle\}$ $\{\langle false \text{ code} \rangle\}$
<code>\bool_if:N</code> TF *	Tests the current truth of $\langle boolean \rangle$, and continues expansion based on this result.
<code>\bool_if:c</code> TF *	
<hr/>	
<code>\bool_show:N</code>	<code>\bool_show:N</code> $\langle boolean \rangle$
<code>\bool_show:c</code>	Displays the logical truth of the $\langle boolean \rangle$ on the terminal.
<hr/>	
New: 2012-02-09	
Updated: 2015-08-01	
<hr/>	
<code>\bool_show:n</code>	<code>\bool_show:n</code> $\{\langle boolean \text{ expression} \rangle\}$
<hr/>	
New: 2012-02-09	
Updated: 2015-08-07	
<hr/>	
<code>\bool_if_exist_p:N</code> *	<code>\bool_if_exist_p:N</code> $\langle boolean \rangle$
<code>\bool_if_exist_p:c</code> *	<code>\bool_if_exist:N</code> TF $\langle boolean \rangle$ $\{\langle true \text{ code} \rangle\}$ $\{\langle false \text{ code} \rangle\}$
<code>\bool_if_exist:N</code> TF *	Tests whether the $\langle boolean \rangle$ is currently defined. This does not check that the $\langle boolean \rangle$
<code>\bool_if_exist:c</code> TF *	really is a boolean variable.
<hr/>	
New: 2012-03-03	
<hr/>	
<code>\l_tmpa_bool</code>	A scratch boolean for local assignment. It is never used by the kernel code, and so is
<code>\l_tmpb_bool</code>	safe for use with any L ^A T _E X3-defined function. However, it may be overwritten by other
<hr/>	
A scratch boolean for global assignment. It is never used by the kernel code, and so is	
<code>\g_tmpa_bool</code>	safe for use with any L ^A T _E X3-defined function. However, it may be overwritten by other
<code>\g_tmpb_bool</code>	non-kernel code and so should only be used for short-term storage.
<hr/>	

3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean $\langle true \rangle$ or $\langle false \rangle$ values, it seems only fitting that we also provide a parser for $\langle boolean \text{ expressions} \rangle$.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean $\langle true \rangle$ or $\langle false \rangle$. It supports the logical operations And, Or and Not as the well-known infix operators `&&`, `||` and `!` 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.

At present, the infix operators `&&` and `||` perform lazy evaluation as well, but this will change in a future release.

<code>\bool_if_p:n</code> ☆	<code>\bool_if_p:n {<boolean expression>}</code>
<code>\bool_if:nTF</code> ☆	<code>\bool_if:nTF {<boolean expression>} {<true code>} {<false code>}</code>

Updated: 2012-07-08 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.

<code>\bool_not_p:n</code> ☆	<code>\bool_not_p:n {<boolean expression>}</code>
------------------------------	---

Updated: 2012-07-08 Function version of `!(<boolean expression>)` within a boolean expression.

<code>\bool_xor_p:nn</code> ☆	<code>\bool_xor_p:nn {<boolexpr₁>} {<boolexpr₂>}</code>
-------------------------------	---

Updated: 2012-07-08 Implements an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operator.

4 Logical loops

Loops using either boolean expressions or stored boolean values.

<code>\bool_do_until:Nn</code> ☆	<code>\bool_do_until:Nn <boolean> {<code>}</code>
----------------------------------	---

`\bool_do_until:cn` ☆ Places the *<code>* in the input stream for T_EX to process, and then checks the logical value of the *<boolean>*. If it is `false` then the *<code>* will be inserted into the input stream again and the process will loop until the *<boolean>* is `true`.

<code>\bool_do_while:Nn</code> ☆	<code>\bool_do_while:Nn <boolean> {<code>}</code>
----------------------------------	---

`\bool_do_while:cn` ☆ Places the *<code>* in the input stream for T_EX to process, and then checks the logical value of the *<boolean>*. If it is `true` then the *<code>* will be inserted into the input stream again and the process will loop until the *<boolean>* is `false`.

<code>\bool_until_do:Nn</code> ☆	<code>\bool_until_do:Nn <boolean> {<code>}</code>
----------------------------------	---

`\bool_until_do:cn` ☆ 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 will then loop until the *<boolean>* is `true`.

<hr/>	
<code>\bool_while_do:Nn</code> ☆	<code>\bool_while_do:Nn <boolean> {<code>}</code>
<code>\bool_while_do:cn</code> ☆	
<hr/>	This function firsts checks the logical value of the <i><boolean></i> . If it is <code>true</code> the <i><code></i> is placed in the input stream and expanded. After the completion of the <i><code></i> the truth of the <i><boolean></i> is re-evaluated. The process will then loop until the <i><boolean></i> is <code>false</code> .
<hr/>	
<code>\bool_do_until:nn</code> ☆	<code>\bool_do_until:nn {<boolean expression>} {<code>}</code>
Updated: 2012-07-08	
<hr/>	Places the <i><code></i> in the input stream for T _E X to process, and then checks the logical value of the <i><boolean expression></i> as described for <code>\bool_if:nTF</code> . If it is <code>false</code> then the <i><code></i> will be inserted into the input stream again and the process will loop until the <i><boolean expression></i> evaluates to <code>true</code> .
<hr/>	
<code>\bool_do_while:nn</code> ☆	<code>\bool_do_while:nn {<boolean expression>} {<code>}</code>
Updated: 2012-07-08	
<hr/>	Places the <i><code></i> in the input stream for T _E X to process, and then checks the logical value of the <i><boolean expression></i> as described for <code>\bool_if:nTF</code> . If it is <code>true</code> then the <i><code></i> will be inserted into the input stream again and the process will loop until the <i><boolean expression></i> evaluates to <code>false</code> .
<hr/>	
<code>\bool_until_do:nn</code> ☆	<code>\bool_until_do:nn {<boolean expression>} {<code>}</code>
Updated: 2012-07-08	
<hr/>	This function firsts checks the logical value of the <i><boolean expression></i> (as described for <code>\bool_if:nTF</code>). If it is <code>false</code> the <i><code></i> is placed in the input stream and expanded. After the completion of the <i><code></i> the truth of the <i><boolean expression></i> is re-evaluated. The process will then loop until the <i><boolean expression></i> is <code>true</code> .
<hr/>	
<code>\bool_while_do:nn</code> ☆	<code>\bool_while_do:nn {<boolean expression>} {<code>}</code>
Updated: 2012-07-08	
<hr/>	This function firsts checks the logical value of the <i><boolean expression></i> (as described for <code>\bool_if:nTF</code>). If it is <code>true</code> the <i><code></i> is placed in the input stream and expanded. After the completion of the <i><code></i> the truth of the <i><boolean expression></i> is re-evaluated. The process will then loop until the <i><boolean expression></i> is <code>false</code> .

5 Producing multiple copies

<hr/>	
<code>\prg_replicate:nn</code> ☆	<code>\prg_replicate:nn {<integer expression>} {<tokens>}</code>
Updated: 2011-07-04	
<hr/>	Evaluates the <i><integer expression></i> (which should be zero or positive) and creates the resulting number of copies of the <i><tokens></i> . The function is both expandable and safe for nesting. It yields its result after two expansion steps.

6 Detecting T_EX's mode

<hr/>	
<code>\mode_if_horizontal_p:</code> ☆	<code>\mode_if_horizontal_p:</code>
<code>\mode_if_horizontal:TF</code> ☆	<code>\mode_if_horizontal:TF {<true code>} {<false code>}</code>
<hr/>	Detects if T _E X is currently in horizontal mode.

<code>\mode_if_inner_p:</code>	★	<code>\mode_if_inner_p:</code>
<code>\mode_if_inner:</code>	<u>TF</u> ★	<code>\mode_if_inner:TF</code> <code>{\true code}</code> <code>{\false code}</code>

Detects if T_EX is currently in inner mode.

<code>\mode_if_math_p:</code>	★	<code>\mode_if_math:TF</code> <code>{\true code}</code> <code>{\false code}</code>
<code>\mode_if_math:</code>	<u>TF</u> ★	

Detects if T_EX is currently in maths mode.

Updated: 2011-09-05

<code>\mode_if_vertical_p:</code>	★	<code>\mode_if_vertical_p:</code>
<code>\mode_if_vertical:</code>	<u>TF</u> ★	<code>\mode_if_vertical:TF</code> <code>{\true code}</code> <code>{\false code}</code>

Detects if T_EX is currently in vertical mode.

7 Primitive conditionals

<code>\if_predicate:w</code>	★	<code>\if_predicate:w</code> <code><predicate></code> <code><true code></code> <code>\else:</code> <code><false code></code> <code>\fi:</code>
------------------------------	---	--

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`.)

<code>\if_bool:N</code>	★	<code>\if_bool:N</code> <code><boolean></code> <code><true code></code> <code>\else:</code> <code><false code></code> <code>\fi:</code>
-------------------------	---	---

This function takes a boolean variable and branches according to the result.

8 Internal programming functions

<code>\group_align_safe_begin:</code>	★	<code>\group_align_safe_begin:</code>
<code>\group_align_safe_end:</code>	★	<code>...</code>
		<code>\group_align_safe_end:</code>

Updated: 2011-08-11

These functions are used to enclose material in a T_EX 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 T_EX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\peek_after:Nw` will 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.

<code>__prg_break_point:Nn</code>	★	<code>__prg_break_point:Nn</code> <code>\<type>_map_break:</code> <code><tokens></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. After the loop ends, the `<tokens>` are 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_ii:nn`.

<u>_prg_map_break:Nn</u> ★	_prg_map_break:Nn \<type>_map_break: {\<user code>} ... _prg_break_point: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 is simply used as a recognizable marker for the <type>.
<u>\g__prg_map_int</u>	This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions _prg_map_1:w, _prg_map_2:w, etc., labelled by \g__prg_map_int hold functions to be mapped over various list datatypes in inline and variable mappings.
<u>_prg_break_point:</u> ★	This copy of \prg_do_nothing: is used to mark the end of a fast short-term recursions: the function _prg_break:n uses this to break out of the loop.
<u>_prg_break:</u> ★ <u>_prg_break:n</u> ★	_prg_break:n {\<tokens>} ... _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 <tokens> in the input stream.

Part VII

The l3quark package

Quarks

1 Introduction to quarks and scan marks

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_`. Scan marks are for internal use by the kernel: they are not intended for more general use.

1.1 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, with the most command use case as 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_new:N <quark>
```

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

<u><code>\q_stop</code></u>	Used as a marker for delimited arguments, such as <code>\cs_set:Npn \tmp:w #1#2 \q_stop {#1}</code>
<u><code>\q_mark</code></u>	Used as a marker for delimited arguments when <code>\q_stop</code> is already in use.
<u><code>\q_nil</code></u>	Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself may need to be tested (in contrast to <code>\q_stop</code> , which is only ever used as a delimiter).
<u><code>\q_no_value</code></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 <code>\prop_get:NnN</code> 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 later should therefore only be used when the argument can definitely take more than a single token.

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

<code>\q_recursion_tail</code>	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.
--------------------------------	---

<code>\q_recursion_stop</code>	This quark is added <i>after</i> the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.
--------------------------------	---

<code>\quark_if_recursion_tail_stop:N</code>	<code>\quark_if_recursion_tail_stop:N <token></code>
--	--

Tests if $\langle token \rangle$ contains only the marker `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

<code>\quark_if_recursion_tail_stop:n</code>	<code>\quark_if_recursion_tail_stop:n {<token list>}</code>
<code>\quark_if_recursion_tail_stop:o</code>	

Updated: 2011-09-06

Tests if the $\langle token list \rangle$ contains only `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

<code>\quark_if_recursion_tail_stop_do:Nn</code>	<code>\quark_if_recursion_tail_stop_do:Nn <token> {<insertion>}</code>
--	--

Tests if $\langle token \rangle$ contains only the marker `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The $\langle insertion \rangle$ code is then added to the input stream after the recursion has ended.

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

Updated: 2011-09-06

Tests if the $\langle token list \rangle$ contains only `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The $\langle insertion \rangle$ code is then added to the input stream after the recursion has ended.

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_db1:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map_db1: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 will do 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 will overwrite the definition of `__my_map_dbl_fn:nn`.

6 Internal quark functions

```
\__quark_if_recursion_tail_break:NN \__quark_if_recursion_tail_break:nN {<token list>}
\__quark_if_recursion_tail_break:nN \<type>_map_break:
```

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:.`

7 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence will never expand in an expansion context and will be (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).

The scan marks system is only for internal use by the kernel team in a small number of very specific places. These functions should not be used more generally.

<code>_scan_new:N</code>	<code>_scan_new:N</code> $\langle scan\ mark \rangle$
----------------------------	---

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

<code>\s_stop</code>	Used at the end of a set of instructions, as a marker that can be jumped to using <code>_use_none_delimit_by_s_stop:w</code> .
-----------------------	--

<code>_use_none_delimit_by_s_stop:w</code>	<code>_use_none_delimit_by_s_stop:w</code> $\langle tokens \rangle$ <code>\s_stop</code>
--	---

Removes the $\langle tokens \rangle$ and `\s_stop` from the input stream. This leads to a low-level TeX error if `\s_stop` is absent.

Part VIII

The l3token 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 will have two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most functions we will 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 will take 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 8.

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.

<code>\char_generate:nn</code> ★	<code>\char_generate:nn {<charcode>} {<catcode>}</code>
----------------------------------	---

New: 2015-09-09

Generates a character token of the given $\langle charcode \rangle$ and $\langle catcode \rangle$ (both of which may be integer expressions). The $\langle catcode \rangle$ 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)

and other values will raise an error.

The $\langle charcode \rangle$ may be any one valid for the engine in use. Note however that for X_YTeX releases prior to 0.99992 only the 8-bit range (0 to 255) is accepted due to engine limitations.

2 Manipulating and interrogating character tokens

<code>\char_set_catcode_escape:N</code>	<code>\char_set_catcode_letter:N <character></code>
<code>\char_set_catcode_group_begin:N</code>	
<code>\char_set_catcode_group_end:N</code>	
<code>\char_set_catcode_math_toggle:N</code>	
<code>\char_set_catcode_alignment:N</code>	
<code>\char_set_catcode_end_line:N</code>	
<code>\char_set_catcode_parameter:N</code>	
<code>\char_set_catcode_math_superscript:N</code>	
<code>\char_set_catcode_math_subscript:N</code>	
<code>\char_set_catcode_ignore:N</code>	
<code>\char_set_catcode_space:N</code>	
<code>\char_set_catcode_letter:N</code>	
<code>\char_set_catcode_other:N</code>	
<code>\char_set_catcode_active:N</code>	
<code>\char_set_catcode_comment:N</code>	
<code>\char_set_catcode_invalid:N</code>	

Updated: 2015-11-11

Sets the category code of the $\langle character \rangle$ to that indicated in the function name. Depending on the current category code of the $\langle token \rangle$ the escape token may also be needed:

`\char_set_catcode_other:N \%`

The assignment is local.

<code>\char_set_catcode_escape:n</code>	<code>\char_set_catcode_letter:n {⟨integer expression⟩}</code>
<code>\char_set_catcode_group_begin:n</code>	
<code>\char_set_catcode_group_end:n</code>	
<code>\char_set_catcode_math_toggle:n</code>	
<code>\char_set_catcode_alignment:n</code>	
<code>\char_set_catcode_end_line:n</code>	
<code>\char_set_catcode_parameter:n</code>	
<code>\char_set_catcode_math_superscript:n</code>	
<code>\char_set_catcode_math_subscript:n</code>	
<code>\char_set_catcode_ignore:n</code>	
<code>\char_set_catcode_space:n</code>	
<code>\char_set_catcode_letter:n</code>	
<code>\char_set_catcode_other:n</code>	
<code>\char_set_catcode_active:n</code>	
<code>\char_set_catcode_comment:n</code>	
<code>\char_set_catcode_invalid:n</code>	

Updated: 2015-11-11

Sets the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer expression \rangle$. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local.

<code>\char_set_catcode:nn</code>	<code>\char_set_catcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
-----------------------------------	---

Updated: 2015-11-11

These functions set the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer expression \rangle$. The first $\langle integer expression \rangle$ 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.

<code>\char_value_catcode:n</code> ★	<code>\char_value_catcode:n {⟨integer expression⟩}</code>
--------------------------------------	---

Expands to the current category code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.

<code>\char_show_value_catcode:n</code>	<code>\char_show_value_catcode:n {⟨integer expression⟩}</code>
---	--

Displays the current category code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.

<code>\char_set_lccode:nn</code>	<code>\char_set_lccode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
----------------------------------	--

Updated: 2015-08-06

Sets up the behaviour of the $\langle character \rangle$ when found inside `\tl_to_lowercase:n`, such that $\langle character_1 \rangle$ will be converted into $\langle character_2 \rangle$. The two $\langle characters \rangle$ may be specified using an $\langle integer expression \rangle$ for the character code concerned. This may include the \TeX ‘ $\langle character \rangle$ ’ 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.

<hr/> <hr/>	<hr/>
<code>\char_value_lccode:n</code> ★	<code>\char_value_lccode:n {\langle integer expression \rangle}</code>
	Expands to the current lower case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <hr/>	<hr/>
<code>\char_show_value_lccode:n</code>	<code>\char_show_value_lccode:n {\langle integer expression \rangle}</code>
	Displays the current lower case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <hr/>	<hr/>
<code>\char_set_uccode:nn</code>	<code>\char_set_uccode:nn {\langle intexpr_1 \rangle} {\langle intexpr_2 \rangle}</code>
Updated: 2015-08-06	Sets up the behaviour of the $\langle character \rangle$ when found inside <code>\tl_to_uppercase:n</code> , such that $\langle character_1 \rangle$ will be converted into $\langle character_2 \rangle$. The two $\langle characters \rangle$ may be specified using an $\langle integer expression \rangle$ for the character code concerned. This may include the TeX ‘ $\langle character \rangle$ ’ method for converting a single character into its character code:
	<pre> \char_set_uccode:nn { ‘\a } { ‘\A } % Standard behaviour \char_set_uccode:nn { ‘\A } { ‘\A - 32 } \char_set_uccode:nn { 60 } { 50 } </pre>
	The setting applies within the current TeX group.
<hr/> <hr/>	<hr/>
<code>\char_value_uccode:n</code> ★	<code>\char_value_uccode:n {\langle integer expression \rangle}</code>
	Expands to the current upper case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <hr/>	<hr/>
<code>\char_show_value_uccode:n</code>	<code>\char_show_value_uccode:n {\langle integer expression \rangle}</code>
	Displays the current upper case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <hr/>	<hr/>
<code>\char_set_mathcode:nn</code>	<code>\char_set_mathcode:nn {\langle intexpr_1 \rangle} {\langle intexpr_2 \rangle}</code>
Updated: 2015-08-06	This function sets up the math code of $\langle character \rangle$. The $\langle character \rangle$ is specified as an $\langle integer expression \rangle$ which will be used as the character code of the relevant character. The setting applies within the current TeX group.
<hr/> <hr/>	<hr/>
<code>\char_value_mathcode:n</code> ★	<code>\char_value_mathcode:n {\langle integer expression \rangle}</code>
	Expands to the current math code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <hr/>	<hr/>
<code>\char_show_value_mathcode:n</code>	<code>\char_show_value_mathcode:n {\langle integer expression \rangle}</code>
	Displays the current math code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <hr/>	<hr/>
<code>\char_set_sfcode:nn</code>	<code>\char_set_sfcode:nn {\langle intexpr_1 \rangle} {\langle intexpr_2 \rangle}</code>
Updated: 2015-08-06	This function sets up the space factor for the $\langle character \rangle$. The $\langle character \rangle$ is specified as an $\langle integer expression \rangle$ which will be used as the character code of the relevant character. The setting applies within the current TeX group.

<hr/> <code>\char_value_sfcode:n</code> ★ <hr/>	<code>\char_value_sfcode:n {⟨integer expression⟩}</code> Expands to the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <code>\char_show_value_sfcode:n</code> <hr/>	<code>\char_show_value_sfcode:n {⟨integer expression⟩}</code> Displays the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <code>\l_char_active_seq</code> New: 2012-01-23 Updated: 2015-11-11 <hr/>	Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category $\langle active \rangle$ (catcode 13). Each entry in the sequence consists of a single escaped token, for example <code>\~</code> . Active tokens should be added to the sequence when they are defined for general document use.
<hr/> <code>\l_char_special_seq</code> New: 2012-01-23 Updated: 2015-11-11 <hr/>	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 $\langle letter \rangle$ (catcode 11) or $\langle other \rangle$ (catcode 12). Each entry in the sequence consists of a single escaped token, for example <code>\</code> for the backslash or <code>\{</code> for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.

3 Generic tokens

<hr/> <code>\token_new:Nn</code> <hr/>	<code>\token_new:Nn ⟨token₁⟩ {⟨token₂⟩}</code> Defines $\langle token_1 \rangle$ to globally be a snapshot of $\langle token_2 \rangle$. This will be an implicit representation of $\langle token_2 \rangle$.
<hr/> <code>\c_group_begin_token</code> <code>\c_group_end_token</code> <code>\c_math_toggle_token</code> <code>\c_alignment_token</code> <code>\c_parameter_token</code> <code>\c_math_superscript_token</code> <code>\c_math_subscript_token</code> <code>\c_space_token</code> <hr/>	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.
<hr/> <code>\c_catcode_letter_token</code> <code>\c_catcode_other_token</code> <hr/>	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.
<hr/> <code>\c_catcode_active_tl</code> <hr/>	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

<code>\token_to_meaning:N</code>	★	<code>\token_to_meaning:N</code>	$\langle token \rangle$
<code>\token_to_meaning:c</code>	★		

Inserts the current meaning of the $\langle token \rangle$ into the input stream as a series of characters of category code 12 (other). This will be the primitive \TeX description of the $\langle token \rangle$, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` will be described as **macros**.

\TeX hackers note: This is the \TeX primitive `\meaning`.

<code>\token_to_str:N</code>	★	<code>\token_to_str:N</code>	$\langle token \rangle$
<code>\token_to_str:c</code>	★		

Converts the given $\langle token \rangle$ into a series of characters with category code 12 (other). The current escape character will be the first character in the sequence, although this will also have category code 12 (the escape character is part of the $\langle token \rangle$). This function requires only a single expansion.

\TeX hackers note: `\token_to_str:N` is the \TeX primitive `\string` renamed.

5 Token conditionals

<code>\token_if_group_begin_p:N</code>	★	<code>\token_if_group_begin_p:N</code>	$\langle token \rangle$
<code>\token_if_group_begin:NTF</code>	★	<code>\token_if_group_begin:NTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests if $\langle token \rangle$ 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.

<code>\token_if_group_end_p:N</code>	★	<code>\token_if_group_end_p:N</code>	$\langle token \rangle$
<code>\token_if_group_end:NTF</code>	★	<code>\token_if_group_end:NTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests if $\langle token \rangle$ 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.

<code>\token_if_math_toggle_p:N</code>	★	<code>\token_if_math_toggle_p:N</code>	$\langle token \rangle$
<code>\token_if_math_toggle:NTF</code>	★	<code>\token_if_math_toggle:NTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a math shift token (`$` when normal \TeX category codes are in force).

<code>\token_if_alignment_p:N</code>	★	<code>\token_if_alignment_p:N</code>	$\langle token \rangle$
<code>\token_if_alignment:NTF</code>	★	<code>\token_if_alignment:NTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an alignment token (`&` when normal \TeX category codes are in force).

<code>\token_if_parameter_p:N</code>	★	<code>\token_if_parameter_p:N</code>	$\langle token \rangle$
<code>\token_if_parameter:NTF</code>	★	<code>\token_if_alignment:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a macro parameter token (# when normal T_EX category codes are in force).

<code>\token_if_math_superscript_p:N</code>	★	<code>\token_if_math_superscript_p:N</code>	$\langle token \rangle$
<code>\token_if_math_superscript:NTF</code>	★	<code>\token_if_math_superscript:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a superscript token (^ when normal T_EX category codes are in force).

<code>\token_if_math_subscript_p:N</code>	★	<code>\token_if_math_subscript_p:N</code>	$\langle token \rangle$
<code>\token_if_math_subscript:NTF</code>	★	<code>\token_if_math_subscript:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a subscript token (_ when normal T_EX category codes are in force).

<code>\token_if_space_p:N</code>	★	<code>\token_if_space_p:N</code>	$\langle token \rangle$
<code>\token_if_space:NTF</code>	★	<code>\token_if_space:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ 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</code>	$\langle token \rangle$
<code>\token_if_letter:NTF</code>	★	<code>\token_if_letter:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a letter token.

<code>\token_if_other_p:N</code>	★	<code>\token_if_other_p:N</code>	$\langle token \rangle$
<code>\token_if_other:NTF</code>	★	<code>\token_if_other:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an “other” token.

<code>\token_if_active_p:N</code>	★	<code>\token_if_active_p:N</code>	$\langle token \rangle$
<code>\token_if_active:NTF</code>	★	<code>\token_if_active:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an active character.

<code>\token_if_eq_catcode_p:NN</code>	★	<code>\token_if_eq_catcode_p:NN</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$
<code>\token_if_eq_catcode:NNTF</code>	★	<code>\token_if_eq_catcode:NNTF</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the two $\langle tokens \rangle$ have the same category code.

<code>\token_if_eq_charcode_p:NN</code>	★	<code>\token_if_eq_charcode_p:NN</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$
<code>\token_if_eq_charcode:NNTF</code>	★	<code>\token_if_eq_charcode:NNTF</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the two $\langle tokens \rangle$ have the same character code.

<code>\token_if_eq_meaning_p:NN</code>	★	<code>\token_if_eq_meaning_p:NN</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$
<code>\token_if_eq_meaning:NNTF</code>	★	<code>\token_if_eq_meaning:NNTF</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the two $\langle tokens \rangle$ 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>

Updated: 2011-05-23 Tests if the $\langle token \rangle$ is a \TeX macro.

<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 $\langle token \rangle$ 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 $\langle token \rangle$ is expandable. This test returns $\langle false \rangle$ 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>

Updated: 2012-01-20 Tests if the $\langle token \rangle$ is a long macro.

<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>

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is a protected macro: a macro which is both protected and long will return logical false .

<code>\token_if_protected_long_macro_p:N</code>	★	<code>\token_if_protected_long_macro_p:N <token></code>
<code>\token_if_protected_long_macro:NTF</code>	★	<code>\token_if_protected_long_macro:NTF <token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is a protected long macro.

<code>\token_if_chardef_p:N</code>	★	<code>\token_if_chardef_p:N <token></code>
<code>\token_if_chardef:NTF</code>	★	<code>\token_if_chardef:NTF <token> {\true code} {\false code}</code>

Updated: 2012-01-20 Tests if the $\langle token \rangle$ is defined to be a chardef.

\TeX hackers note: Booleans, boxes and small integer constants are implemented as chardefs.

<code>\token_if_mathchardef_p:N</code>	★	<code>\token_if_mathchardef_p:N <token></code>
<code>\token_if_mathchardef:NTF</code>	★	<code>\token_if_mathchardef:NTF <token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a mathchardef.

<code>\token_if_dim_register_p:N</code>	★	<code>\token_if_dim_register_p:N <token></code>
<code>\token_if_dim_register:NTF</code>	★	<code>\token_if_dim_register:NTF <token> {\true code} {\false code}</code>

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a dimension register.

<code>\token_if_int_register_p:N</code>	★	<code>\token_if_int_register_p:N</code>	$\langle token \rangle$
<code>\token_if_int_register:NTF</code>	★	<code>\token_if_int_register:NTF</code>	$\langle token \rangle \{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a integer register.

T_EXhackers note: Constant integers may be implemented as integer registers, chardefs, or mathchardefs depending on their value.

<code>\token_if_muskip_register_p:N</code>	★	<code>\token_if_muskip_register_p:N</code>	$\langle token \rangle$
<code>\token_if_muskip_register:NTF</code>	★	<code>\token_if_muskip_register:NTF</code>	$\langle token \rangle \{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$

New: 2012-02-15

Tests if the $\langle token \rangle$ is defined to be a muskip register.

<code>\token_if_skip_register_p:N</code>	★	<code>\token_if_skip_register_p:N</code>	$\langle token \rangle$
<code>\token_if_skip_register:NTF</code>	★	<code>\token_if_skip_register:NTF</code>	$\langle token \rangle \{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a skip register.

<code>\token_if_toks_register_p:N</code>	★	<code>\token_if_toks_register_p:N</code>	$\langle token \rangle$
<code>\token_if_toks_register:NTF</code>	★	<code>\token_if_toks_register:NTF</code>	$\langle token \rangle \{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a toks register (not used byL^AT_EX3).

<code>\token_if_primitive_p:N</code>	★	<code>\token_if_primitive_p:N</code>	$\langle token \rangle$
<code>\token_if_primitive:NTF</code>	★	<code>\token_if_primitive:NTF</code>	$\langle token \rangle \{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$

Updated: 2011-05-23

Tests if the $\langle token \rangle$ is an engine primitive.

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.

<code>\peek_after:Nw</code>	<code>\peek_after:Nw</code>	$\langle function \rangle$	$\langle token \rangle$
-----------------------------	-----------------------------	----------------------------	-------------------------

Locally sets the test variable `\l_peek_token` equal to $\langle token \rangle$ (as an implicit token, *not* as a token list), and then expands the $\langle function \rangle$. The $\langle token \rangle$ will remain in the input stream as the next item after the $\langle function \rangle$. The $\langle token \rangle$ here may be \sqcup , $\{$ 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`

`\peek_gafter:Nw` $\langle function \rangle$ $\langle token \rangle$

Globally sets the test variable `\g_peek_token` equal to $\langle token \rangle$ (as an implicit token, *not* as a token list), and then expands the $\langle function \rangle$. The $\langle token \rangle$ will remain in the input stream as the next item after the $\langle function \rangle$. The $\langle token \rangle$ 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`

`\peek_catcode:NTF` $\langle test token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-12-20

Tests if the next $\langle token \rangle$ in the input stream has the same category code as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the $\langle token \rangle$ will be left in the input stream after the $\langle true code \rangle$ or $\langle false code \rangle$ (as appropriate to the result of the test).

`\peek_catcode_ignore_spaces:NTF`

`\peek_catcode_ignore_spaces:NTF` $\langle test token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-12-20

Tests if the next non-space $\langle token \rangle$ in the input stream has the same category code as the $\langle test token \rangle$ (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 $\langle token \rangle$ will be left in the input stream after the $\langle true code \rangle$ or $\langle false code \rangle$ (as appropriate to the result of the test).

`\peek_catcode_remove:NTF`

`\peek_catcode_remove:NTF` $\langle test token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-12-20

Tests if the next $\langle token \rangle$ in the input stream has the same category code as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the $\langle token \rangle$ will be removed from the input stream if the test is true. The function will then place either the $\langle true code \rangle$ or $\langle false code \rangle$ in the input stream (as appropriate to the result of the test).

`\peek_catcode_remove_ignore_spaces:NTF`

`\peek_catcode_remove_ignore_spaces:NTF` $\langle test token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-12-20

Tests if the next non-space $\langle token \rangle$ in the input stream has the same category code as the $\langle test token \rangle$ (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 $\langle token \rangle$ will be removed from the input stream if the test is true. The function will then place either the $\langle true code \rangle$ or $\langle false code \rangle$ in the input stream (as appropriate to the result of the test).

`\peek_charcode:NTF`

`\peek_charcode:NTF` $\langle test token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-12-20

Tests if the next $\langle token \rangle$ in the input stream has the same character code as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the $\langle token \rangle$ will be left in the input stream after the $\langle true code \rangle$ or $\langle false code \rangle$ (as appropriate to the result of the test).

<u>\peek_charcode_ignore_spaces:NTF</u>	\peek_charcode_ignore_spaces:NTF <i><test token></i> <i>{<true code>}</i> <i>{<false code>}</i>
Updated: 2012-12-20	

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>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<u>\peek_charcode_remove:NTF</u>	\peek_charcode_remove:NTF <i><test token></i> <i>{<true code>}</i> <i>{<false code>}</i>
Updated: 2012-12-20	

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>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<u>\peek_charcode_remove_ignore_spaces:NTF</u>	\peek_charcode_remove_ignore_spaces:NTF <i><test token></i> <i>{<true code>}</i> <i>{<false code>}</i>
Updated: 2012-12-20	

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>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<u>\peek_meaning:NTF</u>	\peek_meaning:NTF <i><test token></i> <i>{<true code>}</i> <i>{<false code>}</i>
Updated: 2011-07-02	

Tests if the next *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the *<token>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<u>\peek_meaning_ignore_spaces:NTF</u>	\peek_meaning_ignore_spaces:NTF <i><test token></i> <i>{<true code>}</i> <i>{<false code>}</i>
Updated: 2012-12-05	

Tests if the next non-space *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<u>\peek_meaning_remove:NTF</u>	\peek_meaning_remove:NTF <i><test token></i> <i>{<true code>}</i> <i>{<false code>}</i>
Updated: 2011-07-02	

Tests if the next *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the *<token>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* 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></code>
	<code>{<true code>} {<false code>}</code>

Updated: 2012-12-05

Tests if the next non-space $\langle token \rangle$ in the input stream has the same meaning as the $\langle test token \rangle$ (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the $\langle token \rangle$ will be removed from the input stream if the test is true. The function will then place either the $\langle true code \rangle$ or $\langle false code \rangle$ in the input stream (as appropriate to the result of the test).

7 Decomposing a macro definition

These functions decompose T_EX macros into their constituent parts: if the $\langle token \rangle$ passed is not a macro then no decomposition can occur. In the later case, all three functions leave `\scan_stop:` in the input stream.

<code>\token_get_arg_spec:N</code> ★	<code>\token_get_arg_spec:N <token></code>
--------------------------------------	--

If the $\langle token \rangle$ is a macro, this function will leave the primitive T_EX argument specification in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

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

will leave `#1#2` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` will be left in the input stream.

T_EXhackers note: If the arg spec. contains the string `->`, then the `spec` function will produce incorrect results.

<code>\token_get_replacement_spec:N</code> ★	<code>\token_get_replacement_spec:N <token></code>
--	--

If the $\langle token \rangle$ is a macro, this function will leave the replacement text in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

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

will leave `x#1 y#2` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` will be left in the input stream.

T_EXhackers note: If the arg spec. contains the string `->`, then the `spec` function will produce incorrect results.

`\token_get_prefix_spec:N` ★

`\token_get_prefix_spec:N` $\langle token \rangle$

If the $\langle token \rangle$ is a macro, this function will leave the \TeX prefixes applicable in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

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

will leave `\long` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` will be left in the input stream

8 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 \TeX '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 \LuaTeX and \XeTeX 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).⁴

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` $\langle token \rangle$ (when the $\langle token \rangle$ is expandable) results in an internal token, displayed (temporarily) as `\notexpanded: $\langle token \rangle$` , whose shape coincides with the $\langle token \rangle$ and whose meaning differs from `\relax`.
- An `\outer endtemplate:` (expanding to another internal token, `end of alignment template`) can be encountered when peeking ahead at the next token.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.

⁴In \LuaTeX , there is also the case of “bytes”, which behave as character tokens of category code 12 (other) and character code between 1114112 and 1114366. They are used to output individual bytes to files, rather than UTF-8.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We will call these tokens *explicit* character tokens. Category codes that a character token can have are listed below by giving a sample output of the T_EX 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 will 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 (`\tl`, `\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 be `\protected` or not, `\long` or not (the opposite of what L^AT_EX3 calls `\nopro`), and `\outer` or not (unused in L^AT_EX3). Their `\meaning` takes the form

`<properties> macro: <parameters>-><replacement>`

where `<properties>` is among `\protected\long\outer`, `<parameters>` describes parameters that the macro expects, such as `#1#2#3`, and `<replacement>` describes how the parameters are manipulated, such as `#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 \TeX 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`.

9 Internal functions

`_char_generate:nn` ★

New: 2016-03-25

`_char_generate:nn` $\{\langle charcode \rangle\}$ $\{\langle catcode \rangle\}$

This function is identical in operation to the public `\char_generate:nn` but omits various sanity tests. In particular, this means it is used in certain places where engine variations need to be accounted for by the kernel. The $\langle catcode \rangle$ must give an explicit integer after a single expansion.

Part IX

The l3int 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

<code>\int_eval:n</code>	★	<code>\int_eval:n {⟨integer expression⟩}</code>
--------------------------	---	---

Evaluates the *⟨integer expression⟩*, expanding any integer and token list variables within the *⟨expression⟩* to their content (without requiring `\int_use:N/\tl_use:N`) and applying the standard mathematical rules. 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 ) }
```

both evaluate to -6 . The *⟨integer expression⟩* may contain the operators `+`, `-`, `*` and `/`, along with parenthesis `(` and `)`. Any functions within the expressions should expand to an *⟨integer denotation⟩*: a sequence of a sign and digits matching the regex `\-?[0-9]+`. After expansion `\int_eval:n` yields an *⟨integer denotation⟩* which is left in the input stream.

T_EXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an *⟨internal integer⟩*, and therefore requires suitable termination if used in a T_EX-style integer assignment.

<code>\int_abs:n</code>	★	<code>\int_abs:n {⟨integer expression⟩}</code>
-------------------------	---	--

Updated: 2012-09-26

Evaluates the *⟨integer expression⟩* as described for `\int_eval:n` and leaves the absolute value of the result in the input stream as an *⟨integer denotation⟩* after two expansions.

<code>\int_div_round:nn</code>	★	<code>\int_div_round:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
--------------------------------	---	--

Updated: 2012-09-26

Evaluates the two *⟨integer expressions⟩* 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 *⟨integer expression⟩*. The result is left in the input stream as an *⟨integer denotation⟩* after two expansions.

<hr/> <code>\int_div_truncate:nn</code> ★ <hr/>	<code>\int_div_truncate:nn {\langle integer_1 \rangle} {\langle integer_2 \rangle}</code>
Updated: 2012-02-09	Evaluates the two $\langle integer expressions \rangle$ as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using <code>/</code> rounds to the closest integer instead. The result is left in the input stream as an $\langle integer denotation \rangle$ after two expansions.

<hr/> <code>\int_max:nn</code> ★	<code>\int_max:nn {\langle integer_1 \rangle} {\langle integer_2 \rangle}</code>
<hr/> <code>\int_min:nn</code> ★	<code>\int_min:nn {\langle integer_1 \rangle} {\langle integer_2 \rangle}</code>
Updated: 2012-09-26	Evaluates the $\langle integer expressions \rangle$ as described for <code>\int_eval:n</code> and leaves either the larger or smaller value in the input stream as an $\langle integer denotation \rangle$ after two expansions.

<hr/> <code>\int_mod:nn</code> ★	<code>\int_mod:nn {\langle integer_1 \rangle} {\langle integer_2 \rangle}</code>
Updated: 2012-09-26	Evaluates the two $\langle integer expressions \rangle$ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting <code>\int_div_truncate:nn {\langle integer_1 \rangle} {\langle integer_2 \rangle}</code> times $\langle integer_2 \rangle$ from $\langle integer_1 \rangle$. Thus, the result has the same sign as $\langle integer_1 \rangle$ and its absolute value is strictly less than that of $\langle integer_2 \rangle$. The result is left in the input stream as an $\langle integer denotation \rangle$ after two expansions.

2 Creating and initialising integers

<hr/> <code>\int_new:N</code>	<code>\int_new:N \langle integer \rangle</code>
<hr/> <code>\int_new:c</code>	Creates a new $\langle integer \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle integer \rangle$ will initially be equal to 0.

<hr/> <code>\int_const:Nn</code>	<code>\int_const:Nn \langle integer \rangle {\langle integer expression \rangle}</code>
<hr/> <code>\int_const:cn</code>	Creates a new constant $\langle integer \rangle$ or raises an error if the name is already taken. The value of the $\langle integer \rangle$ will be set globally to the $\langle integer expression \rangle$.
Updated: 2011-10-22	

<hr/> <code>\int_zero:N</code>	<code>\int_zero:N \langle integer \rangle</code>
<hr/> <code>\int_zero:c</code>	Sets $\langle integer \rangle$ to 0.
<hr/> <code>\int_gzero:N</code>	
<hr/> <code>\int_gzero:c</code>	

<hr/> <code>\int_zero_new:N</code>	<code>\int_zero_new:N \langle integer \rangle</code>
<hr/> <code>\int_zero_new:c</code>	Ensures that the $\langle integer \rangle$ exists globally by applying <code>\int_new:N</code> if necessary, then applies <code>\int_(g)zero:N</code> to leave the $\langle integer \rangle$ set to zero.
<hr/> <code>\int_gzero_new:N</code>	
<hr/> <code>\int_gzero_new:c</code>	
New: 2011-12-13	

<hr/> <code>\int_set_eq:NN</code>	<code>\int_set_eq:NN \langle integer_1 \rangle \langle integer_2 \rangle</code>
<hr/> <code>\int_set_eq:(cN Nc cc)</code>	Sets the content of $\langle integer_1 \rangle$ equal to that of $\langle integer_2 \rangle$.
<hr/> <code>\int_gset_eq:NN</code>	
<hr/> <code>\int_gset_eq:(cN Nc cc)</code>	

<code>\int_if_exist_p:N</code>	★	<code>\int_if_exist_p:N <integer></code>
<code>\int_if_exist_p:c</code>	★	<code>\int_if_exist:NTF <integer> {<true code>} {<false code>}</code>
<code>\int_if_exist:NTF</code>	★	
<code>\int_if_exist:cTF</code>	★	Tests whether the <code><int></code> is currently defined. This does not check that the <code><int></code> really is an integer variable.

New: 2012-03-03

3 Setting and incrementing integers

<code>\int_add:Nn</code>	<code>\int_add:Nn <integer> {<integer expression>}</code>
<code>\int_add:cn</code>	
<code>\int_gadd:Nn</code>	Adds the result of the <code><integer expression></code> to the current content of the <code><integer></code> .
<code>\int_gadd:cn</code>	

Updated: 2011-10-22

<code>\int_decr:N</code>	<code>\int_decr:N <integer></code>
<code>\int_decr:c</code>	
<code>\int_gdecr:N</code>	Decreases the value stored in <code><integer></code> by 1.
<code>\int_gdecr:c</code>	

<code>\int_incr:N</code>	<code>\int_incr:N <integer></code>
<code>\int_incr:c</code>	
<code>\int_gincr:N</code>	Increases the value stored in <code><integer></code> by 1.
<code>\int_gincr:c</code>	

<code>\int_set:Nn</code>	<code>\int_set:Nn <integer> {<integer expression>}</code>
<code>\int_set:cn</code>	
<code>\int_gset:Nn</code>	Sets <code><integer></code> to the value of <code><integer expression></code> , which must evaluate to an integer (as described for <code>\int_eval:n</code>).
<code>\int_gset:cn</code>	

Updated: 2011-10-22

<code>\int_sub:Nn</code>	<code>\int_sub:Nn <integer> {<integer expression>}</code>
<code>\int_sub:cn</code>	
<code>\int_gsub:Nn</code>	Subtracts the result of the <code><integer expression></code> from the current content of the <code><integer></code> .
<code>\int_gsub:cn</code>	

Updated: 2011-10-22

4 Using integers

<code>\int_use:N</code>	★	<code>\int_use:N <integer></code>
<code>\int_use:c</code>	★	
Updated: 2011-10-22		Recovers the content of an <code><integer></code> and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where an <code><integer></code> is required (such as in the first and third arguments of <code>\int_compare:nNnTF</code>).

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 {\langle intexpr_1 \rangle} \langle relation \rangle {\langle intexpr_2 \rangle}
\int_compare:nNnTF ★ \int_compare:nNnTF
                        {\langle intexpr_1 \rangle} \langle relation \rangle {\langle intexpr_2 \rangle}
                        {\langle true code \rangle} {\langle false code \rangle}
```

This function first evaluates each of the $\langle integer\ expressions \rangle$ as described for `\int_eval:n`. The two results are then compared using the $\langle relation \rangle$:

Equal	=
Greater than	>
Less than	<

```
\int_compare_p:n ★ \int_compare_p:n
\int_compare:nTF ★ {
                    \langle intexpr_1 \rangle \langle relation_1 \rangle
                    ...
                    \langle intexpr_N \rangle \langle relation_N \rangle
                    \langle intexpr_{N+1} \rangle
                }
\int_compare:nTF
{
    \langle intexpr_1 \rangle \langle relation_1 \rangle
    ...
    \langle intexpr_N \rangle \langle relation_N \rangle
    \langle intexpr_{N+1} \rangle
}
{\langle true code \rangle} {\langle false code \rangle}
```

Updated: 2013-01-13

This function evaluates the $\langle integer\ expressions \rangle$ as described for `\int_eval:n` and compares consecutive result using the corresponding $\langle relation \rangle$, namely it compares $\langle intexpr_1 \rangle$ and $\langle intexpr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle intexpr_2 \rangle$ and $\langle intexpr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle intexpr_N \rangle$ and $\langle intexpr_{N+1} \rangle$ using the $\langle relation_N \rangle$. The test yields **true** if all comparisons are **true**. Each $\langle integer\ expression \rangle$ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is **false**, then no other $\langle integer\ expression \rangle$ is evaluated and no other comparison is performed. The $\langle relations \rangle$ can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

<code>\int_case:n</code>	★	<code>\int_case:nnTF {⟨test integer expression⟩}</code>
<code>\int_case:nnTF</code>	★	<code>{</code>
<hr/> New: 2013-07-24 <hr/>		<code>{⟨intexpr case₁⟩} {⟨code case₁⟩}</code>
		<code>{⟨intexpr case₂⟩} {⟨code case₂⟩}</code>
		<code>...</code>
		<code>{⟨intexpr case_n⟩} {⟨code case_n⟩}</code>
		<code>}</code>
		<code>{⟨true code⟩}</code>
		<code>{⟨false code⟩}</code>

This function evaluates the $\langle test\ integer\ expression \rangle$ and compares this in turn to each of the $\langle integer\ expression\ cases \rangle$. If the two are equal then the associated $\langle code \rangle$ is left in the input stream. If any of the cases are matched, the $\langle true\ code \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle false\ code \rangle$ 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! }
```

will leave “Medium” in the input stream.

<code>\int_if_even_p:n</code>	★	<code>\int_if_odd_p:n {⟨integer expression⟩}</code>
<code>\int_if_even:nTF</code>	★	<code>\int_if_odd:nTF {⟨integer expression⟩}</code>
<code>\int_if_odd_p:n</code>	★	<code>{⟨true code⟩} {⟨false code⟩}</code>
<code>\int_if_odd:nTF</code>	★	

This function first evaluates the $\langle integer\ expression \rangle$ as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate.

6 Integer expression loops

<code>\int_do_until:nNnn</code>	☆	<code>\int_do_until:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
---------------------------------	---	---

Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for `\int_compare:nNnTF`. If the test is **false** then the $\langle code \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation \rangle$ is **true**.

<code>\int_do_while:nNnn</code>	☆	<code>\int_do_while:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
---------------------------------	---	---

Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for `\int_compare:nNnTF`. If the test is **true** then the $\langle code \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation \rangle$ is **false**.

<hr/> <code>\int_until_do:nNnn</code> ☆ <hr/>	<code>\int_until_do:nNnn {<intexpr₁>} <relation> {<intexpr₂>} {<code>}</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 false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\int_while_do:nNnn</code> ☆ <hr/>	<code>\int_while_do:nNnn {<intexpr₁>} <relation> {<intexpr₂>} {<code>}</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 true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .
<hr/> <code>\int_do_until:nn</code> ☆ <hr/>	<code>\int_do_until:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	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 false then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is true .
<hr/> <code>\int_do_while:nn</code> ☆ <hr/>	<code>\int_do_while:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	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 true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<hr/> <code>\int_until_do:nn</code> ☆ <hr/>	<code>\int_until_do:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	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 false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\int_while_do:nn</code> ☆ <hr/>	<code>\int_while_do:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	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 true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .

7 Integer step functions

`\int_step_function:nnnN` ★

New: 2012-06-04

Updated: 2014-05-30

`\int_step_function:nnnN` $\langle initial\ value \rangle$ $\langle step \rangle$ $\langle final\ value \rangle$ $\langle function \rangle$

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]
```

`\int_step_inline:nnnn`

New: 2012-06-04

Updated: 2014-05-30

`\int_step_inline:nnnn` $\langle initial\ value \rangle$ $\langle step \rangle$ $\langle final\ value \rangle$ $\langle code \rangle$

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`).

`\int_step_variable:nnnNn`

New: 2012-06-04

Updated: 2014-05-30

`\int_step_variable:nnnNn`
 $\langle initial\ value \rangle$ $\langle step \rangle$ $\langle final\ value \rangle$ $\langle tl\ var \rangle$ $\langle code \rangle$

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 $\langle tl\ var \rangle$.

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` $\langle integer\ expression \rangle$

Places the value of the $\langle integer\ expression \rangle$ in the input stream as digits, with category code 12 (other).

`\int_to_alph:n` ★ `\int_to_alph:n {⟨integer expression⟩}`

`\int_to_Alph:n` ★

Updated: 2011-09-17

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_alph:n { 1 }`

places **a** in the input stream,

`\int_to_alph:n { 26 }`

is represented as **z** and

`\int_to_alph: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_alph: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 (which will often be 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_alph:n` function is defined as

```
\cs_new:Npn \int_to_alph:n #1
{
  \int_to_symbols:nnn {#1} { 26 }
  {
    { 1 } { a }
    { 2 } { b }
    ...
    { 26 } { z }
  }
}
```

`\int_to_bin:n` ★

New: 2014-02-11

`\int_to_bin:n {⟨integer expression⟩}`

Calculates the value of the *⟨integer expression⟩* 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> ★	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).
New: 2014-02-11	

<code>\int_to_oct:n</code> ★	<code>\int_to_oct:n {⟨integer expression⟩}</code>
New: 2014-02-11	
	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> ★	Calculates the value of the <i>⟨integer expression⟩</i> and converts it into the appropriate representation in the <i>⟨base⟩</i> ; the later 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).
Updated: 2014-02-11	

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> ☆	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>). The Roman numerals are letters with category code 11 (letter).
Updated: 2011-10-22	

9 Converting from other formats to integers

<code>\int_from_alph:n</code> ★	<code>\int_from_alph:n {⟨letters⟩}</code>
Updated: 2014-08-25	
	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_alph:n</code> and <code>\int_to_Alph:n</code> .

<code>\int_from_bin:n</code> ★	<code>\int_from_bin:n {⟨binary number⟩}</code>
New: 2014-02-11	
Updated: 2014-08-25	
	Converts the <i>⟨binary number⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨binary number⟩</i> 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 <code>\int_to_bin:n</code> .

<hr/> <code>\int_from_hex:n</code> ★ <hr/>	<code>\int_from_hex:n {⟨hexadecimal number⟩}</code>
New: 2014-02-11 Updated: 2014-08-25 <hr/>	Converts the <i>⟨hexadecimal number⟩</i> into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the <i>⟨hexadecimal number⟩</i> by upper or lower case letters. The <i>⟨hexadecimal number⟩</i> 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 <code>\int_to_hex:n</code> and <code>\int_to_Hex:n</code> .
<hr/> <code>\int_from_oct:n</code> ★ <hr/>	<code>\int_from_oct:n {⟨octal number⟩}</code>
New: 2014-02-11 Updated: 2014-08-25 <hr/>	Converts the <i>⟨octal number⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨octal number⟩</i> 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 <code>\int_to_oct:n</code> .
<hr/> <code>\int_from_roman:n</code> ★ <hr/>	<code>\int_from_roman:n {⟨roman numeral⟩}</code>
Updated: 2014-08-25 <hr/>	Converts the <i>⟨roman numeral⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨roman numeral⟩</i> is first converted to a string, with no expansion. The <i>⟨roman numeral⟩</i> may be in upper or lower case; if the numeral contains characters besides <code>mdclxvi</code> or <code>MDCLXVI</code> then the resulting value will be -1. This is the inverse function of <code>\int_to_roman:n</code> and <code>\int_to_Roman:n</code> .
<hr/> <code>\int_from_base:nn</code> ★ <hr/>	<code>\int_from_base:nn {⟨number⟩} {⟨base⟩}</code>
Updated: 2014-08-25 <hr/>	Converts the <i>⟨number⟩</i> expressed in <i>⟨base⟩</i> into the appropriate value in base 10. The <i>⟨number⟩</i> is first converted to a string, with no expansion. The <i>⟨number⟩</i> should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum <i>⟨base⟩</i> value is 36. This is the inverse function of <code>\int_to_base:nn</code> and <code>\int_to_Base:nn</code> .

10 Viewing integers

<hr/> <code>\int_show:N</code> <code>\int_show:c</code> <hr/>	<code>\int_show:N ⟨integer⟩</code>
	Displays the value of the <i>⟨integer⟩</i> on the terminal.
<hr/> <code>\int_show:n</code> <hr/>	<code>\int_show:n {⟨integer expression⟩}</code>
New: 2011-11-22 Updated: 2015-08-07 <hr/>	Displays the result of evaluating the <i>⟨integer expression⟩</i> on the terminal.

11 Constant integers

`\c_minus_one`
`\c_zero`
`\c_one`
`\c_two`
`\c_three`
`\c_four`
`\c_five`
`\c_six`
`\c_seven`
`\c_eight`
`\c_nine`
`\c_ten`
`\c_eleven`
`\c_twelve`
`\c_thirteen`
`\c_fourteen`
`\c_fifteen`
`\c_sixteen`
`\c_thirty_two`
`\c_one_hundred`
`\c_two_hundred_fifty_five`
`\c_two_hundred_fifty_six`
`\c_one_thousand`
`\c_ten_thousand`

Integer values used with primitive tests and assignments: self-terminating nature makes these more convenient and faster than literal numbers.

`\c_max_int`

The maximum value that can be stored as an integer.

`\c_max_register_int`

Maximum number of registers.

`\c_max_char_int`

Maximum character code completely supported by the engine.

12 Scratch integers

`\l_tmpa_int`
`\l_tmpb_int`

Scratch integer for local assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_int`
`\g_tmpb_int`

Scratch integer for global assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

13 Primitive conditionals

<code>\if_int_compare:w</code> ★	<code>\if_int_compare:w <integer> <relation> <integer></code> <code> <true code></code> <code>\else:</code> <code> <false code></code> <code>\fi:</code>
----------------------------------	--

Compare two integers using `<relation>`, which must be one of =, < or > with category code 12. The `\else:` branch is optional.

T_EXhackers note: These are both names for the T_EX primitive `\ifnum`.

<code>\if_case:w</code> ★	<code>\if_case:w <integer> <case₀></code> <code>\or: ★</code> <code> <case₁></code> <code> \or: ...</code> <code> \else: <default></code> <code>\fi:</code>
---------------------------	---

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`).

T_EXhackers note: These are the T_EX primitives `\ifcase` and `\or`.

<code>\if_int_odd:w</code> ★	<code>\if_int_odd:w <tokens> <optional space></code> <code> <true code></code> <code>\else:</code> <code> <true code></code> <code>\fi:</code>
------------------------------	--

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.

T_EXhackers note: This is the T_EX primitive `\ifodd`.

14 Internal functions

<code>__int_to_roman:w</code> ★	<code>__int_to_roman:w <integer> <space> or <non-expandable token></code>
----------------------------------	--

Converts `<integer>` to its lower case Roman representation. Expansion ends when a space or non-expandable token is found. Note that this function produces a string of letters with category code 12 and that protected functions *are* expanded by this process. Negative `<integer>` values result in no output, although the function does not terminate expansion until a suitable endpoint is found in the same way as for positive numbers.

T_EXhackers note: This is the T_EX primitive `\romannumeral` renamed.

<code>__int_value:w</code>	★	<code>__int_value:w</code> $\langle integer \rangle$
		<code>__int_value:w</code> $\langle tokens \rangle$ $\langle optional\ space \rangle$

Expands $\langle tokens \rangle$ until an $\langle integer \rangle$ is formed. One space may be gobbled in the process.

TeXhackers note: This is the TeX primitive `\number`.

<code>__int_eval:w</code>	★	<code>__int_eval:w</code> $\langle intexpr \rangle$ <code>__int_eval_end:</code>
<code>__int_eval_end:</code>	★	

Evaluates $\langle integer\ expression \rangle$ as described for `\int_eval:n`. The evaluation stops when an unexpandable token which is not a valid part of an integer is read or when `__int_eval_end:` is reached. The latter is gobbled by the scanner mechanism: `__int_eval_end:` itself is unexpandable but used correctly the entire construct is expandable.

TeXhackers note: This is the ε -TeX primitive `\numexpr`.

<code>__prg_compare_error:</code>	<code>__prg_compare_error:</code>
<code>__prg_compare_error:Nw</code>	<code>__prg_compare_error:Nw</code> $\langle token \rangle$

These are used within `\int_compare:nTF`, `\dim_compare:nTF` and so on to recover correctly if the n-type argument does not contain a properly-formed relation.

Part X

The l3skip package

Dimensions and skips

L^AT_EX3 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 μ). 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`

`\dim_new:N` $\langle dimension \rangle$

Creates a new $\langle dimension \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle dimension \rangle$ will initially be equal to 0 pt.

`\dim_const:Nn`
`\dim_const:cn`

`\dim_const:Nn` $\langle dimension \rangle$ $\{ \langle dimension expression \rangle \}$

Creates a new constant $\langle dimension \rangle$ or raises an error if the name is already taken. The value of the $\langle dimension \rangle$ will be set globally to the $\langle dimension expression \rangle$.

New: 2012-03-05

`\dim_zero:N`
`\dim_zero:c`
`\dim_gzero:N`
`\dim_gzero:c`

`\dim_zero:N` $\langle dimension \rangle$

Sets $\langle dimension \rangle$ to 0 pt.

`\dim_zero_new:N`
`\dim_zero_new:c`
`\dim_gzero_new:N`
`\dim_gzero_new:c`

`\dim_zero_new:N` $\langle dimension \rangle$

Ensures that the $\langle dimension \rangle$ exists globally by applying `\dim_new:N` if necessary, then applies `\dim_(g)zero:N` to leave the $\langle dimension \rangle$ set to zero.

New: 2012-01-07

`\dim_if_exist_p:N` ★
`\dim_if_exist_p:c` ★
`\dim_if_exist:NTF` ★
`\dim_if_exist:cTF` ★

`\dim_if_exist_p:N` $\langle dimension \rangle$

`\dim_if_exist:NTF` $\langle dimension \rangle$ $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$

Tests whether the $\langle dimension \rangle$ is currently defined. This does not check that the $\langle dimension \rangle$ really is a dimension variable.

New: 2012-03-03

2 Setting dim variables

<code>\dim_add:Nn</code>	<code>\dim_add:Nn <dimension> {<dimension expression>}</code>
<code>\dim_add:cn</code>	
<code>\dim_gadd:Nn</code>	Adds the result of the $\langle dimension\ expression \rangle$ to the current content of the $\langle dimension \rangle$.
<code>\dim_gadd:cn</code>	

Updated: 2011-10-22

<code>\dim_set:Nn</code>	<code>\dim_set:Nn <dimension> {<dimension expression>}</code>
<code>\dim_set:cn</code>	
<code>\dim_gset:Nn</code>	Sets $\langle dimension \rangle$ to the value of $\langle dimension\ expression \rangle$, which must evaluate to a length with units.
<code>\dim_gset:cn</code>	

Updated: 2011-10-22

<code>\dim_set_eq:NN</code>	<code>\dim_set_eq:NN <dimension₁> <dimension₂></code>
<code>\dim_set_eq:(cN Nc cc)</code>	
<code>\dim_gset_eq:NN</code>	Sets the content of $\langle dimension_1 \rangle$ equal to that of $\langle dimension_2 \rangle$.
<code>\dim_gset_eq:(cN Nc cc)</code>	

<code>\dim_sub:Nn</code>	<code>\dim_sub:Nn <dimension> {<dimension expression>}</code>
<code>\dim_sub:cn</code>	
<code>\dim_gsub:Nn</code>	Subtracts the result of the $\langle dimension\ expression \rangle$ from the current content of the $\langle dimension \rangle$.
<code>\dim_gsub:cn</code>	

Updated: 2011-10-22

3 Utilities for dimension calculations

<code>\dim_abs:n</code>	★ <code>\dim_abs:n {<dimexpr>}</code>
Updated: 2012-09-26	Converts the $\langle dimexpr \rangle$ to its absolute value, leaving the result in the input stream as a $\langle dimension\ denotation \rangle$.

<code>\dim_max:nn</code>	★ <code>\dim_max:nn {<dimexpr₁>} {<dimexpr₂>}</code>
<code>\dim_min:nn</code>	★ <code>\dim_min:nn {<dimexpr₁>} {<dimexpr₂>}</code>
New: 2012-09-09	
Updated: 2012-09-26	Evaluates the two $\langle dimension\ expressions \rangle$ and leaves either the maximum or minimum value in the input stream as appropriate, as a $\langle dimension\ denotation \rangle$.

`\dim_ratio:nn` ☆

Updated: 2011-10-22

`\dim_ratio:nn {⟨dimexpr1⟩} {⟨dimexpr2⟩}`

Parses the two *⟨dimension expressions⟩* and converts the ratio of the two to a form suitable for use inside a *⟨dimension expression⟩*. 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 ration 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
```

will display 327680/655360 on the terminal.

4 Dimension expression conditionals

`\dim_compare_p:nNn` ☆

`\dim_compare:nNnTF` ☆

`\dim_compare_p:nNn {⟨dimexpr1⟩} ⟨relation⟩ {⟨dimexpr2⟩}`

`\dim_compare:nNnTF`

`{⟨dimexpr1⟩} ⟨relation⟩ {⟨dimexpr2⟩}`

`{⟨true code⟩} {⟨false code⟩}`

This function first evaluates each of the *⟨dimension expressions⟩* as described for `\dim_eval:n`. The two results are then compared using the *⟨relation⟩*:

Equal	=
Greater than	>
Less than	<

<code>\dim_compare_p:n</code> ★ <code>\dim_compare:nTF</code> ★	<code>\dim_compare_p:n</code> { $\langle dimexpr_1 \rangle$ $\langle relation_1 \rangle$... $\langle dimexpr_N \rangle$ $\langle relation_N \rangle$ $\langle dimexpr_{N+1} \rangle$ } <code>\dim_compare:nTF</code> { $\langle dimexpr_1 \rangle$ $\langle relation_1 \rangle$... $\langle dimexpr_N \rangle$ $\langle relation_N \rangle$ $\langle dimexpr_{N+1} \rangle$ } { $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }
--	--

Updated: 2013-01-13

This function evaluates the $\langle dimension\ expressions \rangle$ as described for `\dim_eval:n` and compares consecutive result using the corresponding $\langle relation \rangle$, namely it compares $\langle dimexpr_1 \rangle$ and $\langle dimexpr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle dimexpr_2 \rangle$ and $\langle dimexpr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle dimexpr_N \rangle$ and $\langle dimexpr_{N+1} \rangle$ using the $\langle relation_N \rangle$. The test yields `true` if all comparisons are `true`. Each $\langle dimension\ expression \rangle$ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other $\langle dimension\ expression \rangle$ is evaluated and no other comparison is performed. The $\langle relations \rangle$ can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

<code>\dim_case:nn</code> ★	<code>\dim_case:nnTF {⟨test dimension expression⟩}</code>
<code>\dim_case:nnTF</code> ★	<code>{</code> <code>{⟨dimexpr case₁⟩} {⟨code case₁⟩}</code> <code>{⟨dimexpr case₂⟩} {⟨code case₂⟩}</code> <code>...</code> <code>{⟨dimexpr case_n⟩} {⟨code case_n⟩}</code> <code>}</code> <code>{⟨true code⟩}</code> <code>{⟨false code⟩}</code>

New: 2013-07-24

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. 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! }
```

will leave “Medium” in the input stream.

5 Dimension expression loops

<code>\dim_do_until:nNnn</code> ☆	<code>\dim_do_until:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX 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⟩* will be inserted into the input stream again and a loop will occur until the *⟨relation⟩* is **true**.

<code>\dim_do_while:nNnn</code> ☆	<code>\dim_do_while:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX 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⟩* will be inserted into the input stream again and a loop will occur until the *⟨relation⟩* is **false**.

<code>\dim_until_do:nNnn</code> ☆	<code>\dim_until_do:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}</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 T_EX the test will be repeated, and a loop will occur until the test is **true**.

<hr/> <code>\dim_while_do:nNnn</code> ☆ <hr/>	<code>\dim_while_do:nNnn {<dimexpr₁>} <relation> {<dimexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><dimension expressions></i> as described for <code>\dim_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .
<hr/> <code>\dim_do_until:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_do_until:nn {<dimension relation>} {<code>}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is false then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is true .
<hr/> <code>\dim_do_while:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_do_while:nn {<dimension relation>} {<code>}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<hr/> <code>\dim_until_do:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_until_do:nn {<dimension relation>} {<code>}</code>
	Evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\dim_while_do:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_while_do:nn {<dimension relation>} {<code>}</code>
	Evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .

6 Using dim expressions and variables

<hr/> <code>\dim_eval:n</code> ☆ <hr/> <div>Updated: 2011-10-22</div>	<code>\dim_eval:n {<dimension expression>}</code>
	Evaluates the <i><dimension expression></i> , expanding any dimensions and token list variables within the <i><expression></i> to their content (without requiring <code>\dim_use:N/\tl_use:N</code>) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a <i><dimension denotation></i> after two expansions. This will be expressed in points (pt), and will require suitable termination if used in a T _E X-style assignment as it is <i>not</i> an <i><internal dimension></i> .
<hr/> <code>\dim_use:N</code> ☆ <code>\dim_use:c</code> ☆ <hr/>	<code>\dim_use:N <dimension></code>
	Recovers the content of a <i><dimension></i> and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a <i><dimension></i> is required (such as in the argument of <code>\dim_eval:n</code>).

T_EXhackers note: `\dim_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

<code>\dim_to_decimal:n</code> ★	<code>\dim_to_decimal:n {⟨dimexpr⟩}</code>
----------------------------------	--

New: 2014-07-15

Evaluates the $\langle dimension expression \rangle$, 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.

<code>\dim_to_decimal_in_bp:n</code> ★	<code>\dim_to_decimal_in_bp:n {⟨dimexpr⟩}</code>
--	--

New: 2014-07-15

Evaluates the $\langle dimension expression \rangle$, 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.

<code>\dim_to_decimal_in_sp:n</code> ★	<code>\dim_to_decimal_in_sp:n {⟨dimexpr⟩}</code>
--	--

New: 2015-05-18

Evaluates the $\langle dimension expression \rangle$, and leaves the result, expressed in scaled points (`sp`) in the input stream, with *no units*. The result will necessarily be an integer.

<code>\dim_to_decimal_in_unit:nn</code> ★	<code>\dim_to_decimal_in_unit:nn {⟨dimexpr₁⟩} {⟨dimexpr₂⟩}</code>
---	---

New: 2014-07-15

Evaluates the $\langle dimension expressions \rangle$, and leaves the value of $\langle dimexpr_1 \rangle$, expressed in a unit given by $\langle dimexpr_2 \rangle$, 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`.

<hr/> <code>\dim_to_fp:n</code> ★ <hr/>	<code>\dim_to_fp:n {⟨<i>dimexpr</i>⟩}</code>
<hr/> New: 2012-05-08 <hr/>	Expands to an internal floating point number equal to the value of the $\langle dimexpr \rangle$ in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, <code>\dim_to_fp:n</code> can be used to speed up parts of a computation where a low precision is acceptable.

7 Viewing dim variables

<hr/> <code>\dim_show:N</code> <code>\dim_show:c</code> <hr/>	<code>\dim_show:N ⟨<i>dimension</i>⟩</code> Displays the value of the $\langle dimension \rangle$ on the terminal.
<hr/> <code>\dim_show:n</code> <hr/>	<code>\dim_show:n {⟨<i>dimension expression</i>⟩}</code>
<hr/> New: 2011-11-22 Updated: 2015-08-07 <hr/>	Displays the result of evaluating the $\langle dimension expression \rangle$ on the terminal.

8 Constant dimensions

<hr/> <code>\c_max_dim</code> <hr/>	The maximum value that can be stored as a dimension. This can also be used as a component of a skip.
<hr/> <code>\c_zero_dim</code> <hr/>	A zero length as a dimension. This can also be used as a component of a skip.

9 Scratch dimensions

<hr/> <code>\l_tmpa_dim</code> <code>\l_tmpb_dim</code> <hr/>	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.
<hr/> <code>\g_tmpa_dim</code> <code>\g_tmpb_dim</code> <hr/>	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.

10 Creating and initialising skip variables

<hr/> <code>\skip_new:N</code> <code>\skip_new:c</code> <hr/>	<code>\skip_new:N ⟨<i>skip</i>⟩</code> Creates a new $\langle skip \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle skip \rangle$ will initially be equal to 0pt.
--	---

<code>\skip_const:Nn</code>	<code>\skip_const:Nn <skip> {<skip expression>}</code>
<code>\skip_const:cn</code>	Creates a new constant <i><skip></i> or raises an error if the name is already taken. The value of the <i><skip></i> will be set globally to the <i><skip expression></i> .
New: 2012-03-05	

<code>\skip_zero:N</code>	<code>\skip_zero:N <skip></code>
<code>\skip_zero:c</code>	Sets <i><skip></i> to 0 pt.
<code>\skip_gzero:N</code>	
<code>\skip_gzero:c</code>	

<code>\skip_zero_new:N</code>	<code>\skip_zero_new:N <skip></code>
<code>\skip_zero_new:c</code>	Ensures that the <i><skip></i> exists globally by applying <code>\skip_new:N</code> if necessary, then applies <code>\skip_(g)zero:N</code> to leave the <i><skip></i> set to zero.
<code>\skip_gzero_new:N</code>	
<code>\skip_gzero_new:c</code>	
<hr/>	
New: 2012-01-07	

<code>\skip_if_exist_p:N</code> *	<code>\skip_if_exist_p:N</code> $\langle skip \rangle$
<code>\skip_if_exist_p:c</code> *	<code>\skip_if_exist:NTF</code> $\langle skip \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\skip_if_exist:NTF</code> *	Tests whether the $\langle skip \rangle$ is currently defined. This does not check that the $\langle skip \rangle$ really is a skip variable.
<code>\skip_if_exist:cTF</code> *	

New: 2012-03-03

11 Setting skip variables

<code>\skip_add:Nn</code>	<code>\skip_add:Nn <skip> {<skip expression>}</code>
<code>\skip_add:cn</code>	Adds the result of the <i><skip expression></i> to the current content of the <i><skip></i> .
<code>\skip_gadd:Nn</code>	
<code>\skip_gadd:cn</code>	
Updated: 2011-10-22	

<code>\skip_set:Nn</code>	<code>\skip_set:Nn <skip> {<skip expression>}</code>
<code>\skip_set:cn</code>	Sets <i><skip></i> to the value of <i><skip expression></i> , which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).
<code>\skip_gset:Nn</code>	
<code>\skip_gset:cn</code>	

Updated: 2011-10-22

<code>\skip_set_eq:NN</code>	<code>\skip_set_eq:NN <skip₁₂</code>
<code>\skip_set_eq:(cN Nc cc)</code>	Sets the content of <i><skip_{1 equal to that of <i><skip_{2.}</i>}</i>
<code>\skip_gset_eq:NN</code>	
<code>\skip_gset_eq:(cN Nc cc)</code>	

<code>\skip_sub:Nn</code>	<code>\skip_sub:Nn <skip> {<skip expression>}</code>
<code>\skip_sub:cn</code>	Subtracts the result of the <i><skip expression></i> from the current content of the <i><skip></i> .
<code>\skip_gsub:Nn</code>	
<code>\skip_gsub:cn</code>	

Updated: 2011-10-22

12 Skip expression conditionals

<code>\skip_if_eq_p:n</code> ★	<code>\skip_if_eq_p:nn {\langle skipexpr_1 \rangle} {\langle skipexpr_2 \rangle}</code>
<code>\skip_if_eq:nnTF</code> ★	<code>\dim_compare:nTF</code> <code> {\langle skipexpr_1 \rangle} {\langle skipexpr_2 \rangle}</code> <code> {\langle true code \rangle} {\langle false code \rangle}</code>

This function first evaluates each of the $\langle skip\ expression \rangle$ 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.

<code>\skip_if_finite_p:n</code> ★	<code>\skip_if_finite_p:n {\langle skipexpr \rangle}</code>
<code>\skip_if_finite:nTF</code> ★	<code>\skip_if_finite:nTF {\langle skipexpr \rangle} {\langle true code \rangle} {\langle false code \rangle}</code>

New: 2012-03-05

Evaluates the $\langle skip\ expression \rangle$ as described for `\skip_eval:n`, and then tests if all of its components are finite.

13 Using skip expressions and variables

<code>\skip_eval:n</code> ★	<code>\skip_eval:n {\langle skip expression \rangle}</code>
-----------------------------	---

Updated: 2011-10-22

Evaluates the $\langle skip\ expression \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ 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 $\langle glue\ denotation \rangle$ after two expansions. This will be expressed in points (pt), and will require suitable termination if used in a T_EX-style assignment as it is *not* an $\langle internal\ glue \rangle$.

<code>\skip_use:N</code> ★	<code>\skip_use:N \langle skip \rangle</code>
<code>\skip_use:c</code> ★	

Recovers the content of a $\langle skip \rangle$ and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of `\skip_eval:n`).

T_EXhackers note: `\skip_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

14 Viewing skip variables

<code>\skip_show:N</code>	<code>\skip_show:N \langle skip \rangle</code>
<code>\skip_show:c</code>	

Displays the value of the $\langle skip \rangle$ on the terminal.

<code>\skip_show:n</code>	<code>\skip_show:n {\langle skip expression \rangle}</code>
---------------------------	---

New: 2011-11-22

Updated: 2015-08-07

Displays the result of evaluating the $\langle skip\ expression \rangle$ on the terminal.

15 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	

16 Scratch skips

<code>\l_tmpa_skip</code> <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 ^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_tmpa_skip</code> <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 ^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.
--	---

17 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 <i><skip></i> into the current list.
Updated: 2011-10-22	

T_EXhackers note: `\skip_horizontal:N` is the T_EX primitive `\hskip` renamed.

<code>\skip_vertical:N</code>	<code>\skip_vertical:N <skip></code>
<code>\skip_vertical:c</code>	<code>\skip_vertical:n {\<skipexpr>}</code>
<code>\skip_vertical:n</code>	Inserts a vertical <i><skip></i> into the current list.
Updated: 2011-10-22	

T_EXhackers note: `\skip_vertical:N` is the T_EX primitive `\vskip` renamed.

18 Creating and initialising muskip variables

<code>\muskip_new:N</code> <code>\muskip_new:c</code>	<code>\muskip_new:N <muskip></code>
	Creates a new <i><muskip></i> or raises an error if the name is already taken. The declaration is global. The <i><muskip></i> will initially be equal to 0 mu.

<hr/>	
<code>\muskip_const:Nn</code>	<code>\muskip_const:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_const:cn</code>	Creates a new constant $\langle muskip \rangle$ or raises an error if the name is already taken. The value of the $\langle muskip \rangle$ will be set globally to the $\langle muskip expression \rangle$.
New: 2012-03-05	
<hr/>	
<code>\muskip_zero:N</code>	<code>\skip_zero:N <muskip></code>
<code>\muskip_zero:c</code>	Sets $\langle muskip \rangle$ to 0 mu.
<code>\muskip_gzero:N</code>	
<code>\muskip_gzero:c</code>	
<hr/>	
<code>\muskip_zero_new:N</code>	<code>\muskip_zero_new:N <muskip></code>
<code>\muskip_zero_new:c</code>	Ensures that the $\langle muskip \rangle$ exists globally by applying <code>\muskip_new:N</code> if necessary, then applies <code>\muskip_(g)zero:N</code> to leave the $\langle muskip \rangle$ set to zero.
<code>\muskip_gzero_new:N</code>	
<code>\muskip_gzero_new:c</code>	
New: 2012-01-07	
<hr/>	
<code>\muskip_if_exist_p:N</code> ★	<code>\muskip_if_exist_p:N <muskip></code>
<code>\muskip_if_exist_p:c</code> ★	<code>\muskip_if_exist:NTF <muskip> {<true code>} {<false code>}</code>
<code>\muskip_if_exist:NTF</code> ★	Tests whether the $\langle muskip \rangle$ is currently defined. This does not check that the $\langle muskip \rangle$ really is a muskip variable.
<code>\muskip_if_exist:cTF</code> ★	
New: 2012-03-03	
<hr/>	

19 Setting muskip variables

<hr/>	
<code>\muskip_add:Nn</code>	<code>\muskip_add:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_add:cn</code>	Adds the result of the $\langle muskip expression \rangle$ to the current content of the $\langle muskip \rangle$.
<code>\muskip_gadd:Nn</code>	
<code>\muskip_gadd:cn</code>	
Updated: 2011-10-22	
<hr/>	
<code>\muskip_set:Nn</code>	<code>\muskip_set:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_set:cn</code>	Sets $\langle muskip \rangle$ to the value of $\langle muskip expression \rangle$, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu.
<code>\muskip_gset:Nn</code>	
<code>\muskip_gset:cn</code>	
Updated: 2011-10-22	
<hr/>	
<code>\muskip_set_eq:NN</code>	<code>\muskip_set_eq:NN <muskip₁₂</code>
<code>\muskip_set_eq:(cN Nc cc)</code>	Sets the content of $\langle muskip_1 \rangle$ equal to that of $\langle muskip_2 \rangle$.
<code>\muskip_gset_eq:NN</code>	
<code>\muskip_gset_eq:(cN Nc cc)</code>	
<hr/>	
<code>\muskip_sub:Nn</code>	<code>\muskip_sub:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_sub:cn</code>	Subtracts the result of the $\langle muskip expression \rangle$ from the current content of the $\langle skip \rangle$.
<code>\muskip_gsub:Nn</code>	
<code>\muskip_gsub:cn</code>	
Updated: 2011-10-22	
<hr/>	

20 Using muskip expressions and variables

<hr/> <code>\muskip_eval:n</code> ★	<code>\muskip_eval:n {⟨muskip expression⟩}</code>
<hr/> Updated: 2011-10-22	
	Evaluates the <i>⟨muskip expression⟩</i> , expanding any skips and token list variables within the <i>⟨expression⟩</i> to their content (without requiring <code>\muskip_use:N/\tl_use:N</code>) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a <i>⟨muglue denotation⟩</i> after two expansions. This will be expressed in mu, and will require suitable termination if used in a T _E X-style assignment as it is <i>not</i> an <i>⟨internal muglue⟩</i> .
<hr/> <code>\muskip_use:N</code> ★	<code>\muskip_use:N ⟨muskip⟩</code>
<hr/> <code>\muskip_use:c</code> ★	
	Recovers the content of a <i>⟨skip⟩</i> and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a <i>⟨dimension⟩</i> is required (such as in the argument of <code>\muskip_eval:n</code>).

T_EXhackers note: `\muskip_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

21 Viewing muskip variables

<hr/> <code>\muskip_show:N</code>	<code>\muskip_show:N ⟨muskip⟩</code>
<hr/> <code>\muskip_show:c</code>	
	Displays the value of the <i>⟨muskip⟩</i> on the terminal.

<hr/> <code>\muskip_show:n</code>	<code>\muskip_show:n {⟨muskip expression⟩}</code>
<hr/> New: 2011-11-22	
<hr/> Updated: 2015-08-07	
	Displays the result of evaluating the <i>⟨muskip expression⟩</i> on the terminal.

22 Constant muskips

<hr/> <code>\c_max_muskip</code>	The maximum value that can be stored as a muskip, with no stretch nor shrink component.
<hr/> <code>\c_zero_muskip</code>	A zero length as a muskip, with no stretch nor shrink component.

23 Scratch muskips

<hr/> <code>\l_tmpa_muskip</code>	Scratch muskip 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.
<hr/> <code>\l_tmpb_muskip</code>	

`\g_tmpa_muskip`
`\g_tmpb_muskip`

Scratch muskip 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.

24 Primitive conditional

`\if_dim:w` `\if_dim:w` $\langle dimen_1 \rangle$ $\langle relation \rangle$ $\langle dimen_2 \rangle$
 $\langle true\ code \rangle$
`\else:`
 $\langle false \rangle$
`\fi:`

Compare two dimensions. The $\langle relation \rangle$ is one of $<$, $=$ or $>$ with category code 12.

T_EXhackers note: This is the T_EX primitive `\ifdim`.

25 Internal functions

`__dim_eval:w` \star `__dim_eval:w` $\langle dimexpr \rangle$ `__dim_eval_end:`
`__dim_eval_end:` \star

Evaluates $\langle dimension\ expression \rangle$ as described for `\dim_eval:n`. The evaluation stops when an unexpandable token which is not a valid part of a dimension is read or when `__dim_eval_end:` is reached. The latter is gobbled by the scanner mechanism: `__dim_eval_end:` itself is unexpandable but used correctly the entire construct is expandable.

T_EXhackers note: This is the ε -T_EX primitive `\dimexpr`.

Part XI

The l3tl package

Token lists

T_EX works with tokens, and L^AT_EX3 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, function 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_EX 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 $\langle tl\ var \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle tl\ var \rangle$ will initially be empty.

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

Creates a new constant $\langle tl\ var \rangle$ or raises an error if the name is already taken. The value of the $\langle tl\ var \rangle$ will be set globally to the $\langle token\ list \rangle$.

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

Clears all entries from the $\langle tl\ var \rangle$.

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

2 Adding data to token list variables

<hr/>	
<code>\tl_set:Nn</code>	<code>\tl_set:Nn <tl var> {\tokens}</code>
<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>	
<hr/>	
Sets $\langle tl\ var \rangle$ to contain $\langle tokens \rangle$, removing any previous content from the variable.	
<hr/>	
<code>\tl_put_left:Nn</code>	<code>\tl_put_left:Nn <tl var> {\tokens}</code>
<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>	
<hr/>	
Appends $\langle tokens \rangle$ to the left side of the current content of $\langle tl\ var \rangle$.	
<hr/>	
<code>\tl_put_right:Nn</code>	<code>\tl_put_right:Nn <tl var> {\tokens}</code>
<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>	
<hr/>	
Appends $\langle tokens \rangle$ to the right side of the current content of $\langle tl\ var \rangle$.	

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 {abbccd} \tl_remove_all:Nn \l_tmpa_tl {bc}
```

will result in `\l_tmpa_tl` containing `abcd`.

4 Reassigning token list category codes

These functions allow the rescanning of tokens: re-apply T_EX'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).

<code>\tl_set_rescan:Nnn</code>	<code>\tl_set_rescan:Nnn <tl var> {<setup>} {<tokens>}</code>
<code>\tl_set_rescan:(Nno Nnx cnn cno cnx)</code>	
<code>\tl_gset_rescan:Nnn</code>	
<code>\tl_gset_rescan:(Nno Nnx cnn cno cnx)</code>	

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 $\langle setup \rangle$ will be those in force at the point of use of `\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 are relevant. See also `\tl_rescan:nn`.

TeXhackers note: The $\langle tokens \rangle$ are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\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. Only the case of a single line is supported in LuaTeX because of a bug in this engine.

<code>\tl_rescan:nn</code>	<code>\tl_rescan:nn {<setup>} {<tokens>}</code>
----------------------------	---

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 $\langle setup \rangle$ will be those in force at the point of use of `\tl_rescan:nn`.) The $\langle setup \rangle$ is run within a group and may contain any valid input, although only changes in category codes are relevant. See also `\tl_set_rescan:Nnn`, which is more robust than using `\tl_set:Nn` in the $\langle tokens \rangle$ argument of `\tl_rescan:nn`.

TeXhackers note: The $\langle tokens \rangle$ are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\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. Only the case of a single line is supported in LuaTeX because of a bug in this engine.

5 Token list conditionals

<code>\tl_if_blank_p:n</code>	★	<code>\tl_if_blank_p:n {<token list>}</code>
<code>\tl_if_blank_p:(V o)</code>	★	<code>\tl_if_blank:nTF {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_blank:nTF</code>	★	
<code>\tl_if_blank:(V o)TF</code>	★	

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.

<code>\tl_if_empty_p:N</code>	★	<code>\tl_if_empty_p:N <tl var></code>
<code>\tl_if_empty_p:c</code>	★	<code>\tl_if_empty:NNTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_empty:nTF</code>	★	Tests if the <i><token list variable></i> is entirely empty (<i>i.e.</i> contains no tokens at all).
<code>\tl_if_empty:cTF</code>	★	

<code>\tl_if_empty_p:n</code>	★	<code>\tl_if_empty_p:n {<token list>}</code>
<code>\tl_if_empty_p:(V o)</code>	★	<code>\tl_if_empty:nNTF {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_empty:nTF</code>	★	Tests if the <i><token list></i> is entirely empty (<i>i.e.</i> contains no tokens at all).
<code>\tl_if_empty:(V o)TF</code>	★	

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

<code>\tl_if_eq_p:NN</code>	★	<code>\tl_if_eq_p:NN <tl var₁> <tl var₂></code>
<code>\tl_if_eq_p:(Nc cN cc)</code>	★	<code>\tl_if_eq:NNTF <tl var₁> <tl var₂> {<true code>} {<false code>}</code>
<code>\tl_if_eq:NNTF</code>	★	Compares the content of two <i><token list variables></i> and is logically true if the two contain the same list of tokens (<i>i.e.</i> identical in both the list of characters they contain and the category codes of those characters). Thus for example
<code>\tl_if_eq:(Nc cN cc)TF</code>	★	

```

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

```

yields **false**.

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

Tests if *<token list₁>* and *<token list₂>* contain the same list of tokens, both in respect of character codes and category codes.

<code>\tl_if_in:NnTF</code>	★	<code>\tl_if_in:NnTF <tl var> {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_in:cnTF</code>	★	Tests if the <i><token list></i> is found in the content of the <i><tl var></i> . The <i><token list></i> cannot contain the tokens <code>{</code> , <code>}</code> or <code>#</code> (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_if_in:nnTF</code>	★	<code>\tl_if_in:nnTF {<token list₁>} {<token list₂>} {<true code>} {<false code>}</code>
<code>\tl_if_in:(Vn on no)TF</code>	★	Tests if <i><token list₂></i> is found inside <i><token list₁></i> . The <i><token list₂></i> cannot contain the tokens <code>{</code> , <code>}</code> or <code>#</code> (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_if_single_p:N</code>	★	<code>\tl_if_single_p:N <tl var></code>
<code>\tl_if_single_p:c</code>	★	<code>\tl_if_single:NNTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_single:NNTF</code>	★	Tests if the content of the <i><tl var></i> consists of a single item, <i>i.e.</i> 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 <code>\tl_count:N</code> .
<code>\tl_if_single:cTF</code>	★	

Updated: 2011-08-13

<code>\tl_if_single_p:n</code> ★	<code>\tl_if_single_p:n {⟨token list⟩}</code>
<code>\tl_if_single:nTF</code> ★	<code>\tl_if_single:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>

Updated: 2011-08-13

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`.

<code>\tl_case:Nn</code> ★	<code>\tl_case:NnTF ⟨test token list variable⟩</code>
<code>\tl_case:cn</code> ★	<code>{</code>
<code>\tl_case:NnTF</code> ★	<code> ⟨token list variable case₁⟩ {⟨code case₁⟩}</code>
<code>\tl_case:cnTF</code> ★	<code> ⟨token list variable case₂⟩ {⟨code case₂⟩}</code>
	<code> ...</code>
	<code> ⟨token list variable case_n⟩ {⟨code case_n⟩}</code>
	<code>}</code>
	<code>{⟨true code⟩}</code>
	<code>{⟨false code⟩}</code>

New: 2013-07-24

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. 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

<code>\tl_map_function:NN</code> ☆	<code>\tl_map_function:NN ⟨tl var⟩ ⟨function⟩</code>
<code>\tl_map_function:cN</code> ☆	

Updated: 2012-06-29

Applies *⟨function⟩* to every *⟨item⟩* in the *⟨tl var⟩*. The *⟨function⟩* will receive 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`.

<code>\tl_map_function:nN</code> ☆	<code>\tl_map_function:nN ⟨token list⟩ ⟨function⟩</code>
------------------------------------	--

Updated: 2012-06-29

Applies *⟨function⟩* to every *⟨item⟩* in the *⟨token list⟩*, The *⟨function⟩* will receive 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`.

<code>\tl_map_inline:Nn</code>	<code>\tl_map_inline:Nn ⟨tl var⟩ {⟨inline function⟩}</code>
<code>\tl_map_inline:cn</code>	

Updated: 2012-06-29

Applies the *⟨inline function⟩* to every *⟨item⟩* stored within the *⟨tl var⟩*. The *⟨inline function⟩* should consist of code which will receive the *⟨item⟩* as #1. One in line mapping can be nested inside another. See also `\tl_map_function:NN`.

<code>\tl_map_inline:nn</code>	<code>\tl_map_inline:nn ⟨token list⟩ {⟨inline function⟩}</code>
--------------------------------	---

Updated: 2012-06-29

Applies the *⟨inline function⟩* to every *⟨item⟩* stored within the *⟨token list⟩*. The *⟨inline function⟩* should consist of code which will receive the *⟨item⟩* as #1. One in line mapping can be nested inside another. See also `\tl_map_function:nN`.

<code>\tl_map_variable:NNn</code>	<code>\tl_map_variable:NNn <tl var> <variable> {<function>}</code>
<code>\tl_map_variable:cNn</code>	Applies the <i><function></i> to every <i><item></i> stored within the <i><tl var></i> . The <i><function></i> should consist of code which will receive the <i><item></i> stored in the <i><variable></i> . One variable mapping can be nested inside another. See also <code>\tl_map_inline:Nn</code> .
Updated: 2012-06-29	

<code>\tl_map_variable:nNn</code>	<code>\tl_map_variable:nNn <token list> <variable> {<function>}</code>
Updated: 2012-06-29	Applies the <i><function></i> to every <i><item></i> stored within the <i><token list></i> . The <i><function></i> should consist of code which will receive the <i><item></i> stored in the <i><variable></i> . One variable mapping can be nested inside another. See also <code>\tl_map_inline:nn</code> .

<code>\tl_map_break: ☆</code>	<code>\tl_map_break:</code>
Updated: 2012-06-29	Used to terminate a <code>\tl_map...</code> function before all entries in the <i><token list variable></i> have been processed. This will normally take 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 will lead to low level \TeX errors.

\TeX hackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before the *<tokens>* are inserted into the input stream. This will depend on the design of the mapping function.

<code>\tl_map_break:n ☆</code>	<code>\tl_map_break:n {<tokens>}</code>
Updated: 2012-06-29	Used to terminate a <code>\tl_map...</code> function before all entries in the <i><token list variable></i> have been processed, inserting the <i><tokens></i> after the mapping has ended. This will normally take place within a conditional statement, for example

```

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

```

Use outside of a `\tl_map...` scenario will lead to low level \TeX errors.

\TeX hackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before the *<tokens>* are inserted into the input stream. This will depend on the design of the mapping function.

7 Using token lists

`\tl_to_str:n` ★ `\tl_to_str:n {⟨token list⟩}`

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).

TeXhackers note: 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 {\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` ★ Recovers the content of a $\langle tl\ var \rangle$ and places it directly in the input stream. An error will be 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)` ★ 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 will ignore any unprotected spaces within $\langle tokens \rangle$. See also `\tl_count:N`. This function requires three expansions, giving an $\langle integer\ denotation \rangle$.

New: 2012-05-13

`\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 will ignore any unprotected spaces within the `<tl var>`. See also `\tl_count:n`. This function requires three expansions, giving an *<integer denotation>*.

`\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 will preserve 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 will 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 will preserve 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 will remove 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 will not expand further when appearing in an *x*-type argument expansion.

`\tl_trim_spaces:n` ★
 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 will not expand further when appearing in an *x*-type argument expansion.

`\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.

<code>\tl_sort:Nn</code>	<code>\tl_sort:Nn <tl var> {<comparison code>}</code>
<code>\tl_sort:cn</code>	Sorts the items in the <i><tl var></i> according to the <i><comparison code></i> , and assigns the result to <i><tl var></i> . The details of sorting comparison are described in Section 1.
<code>\tl_gsort:Nn</code>	
<code>\tl_gsort:cn</code>	
New: 2017-02-06	

<code>\tl_sort:nN</code> ★	<code>\tl_sort:nN {<token list>} <conditional></code>
New: 2017-02-06	Sorts the items in the <i><token list></i> , using the <i><conditional></i> to compare items, and leaves the result in the input stream. The <i><conditional></i> should have signature <code>:nnTF</code> , 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 will not expand further when appearing in an **x**-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.

<code>\tl_head:N</code> ★	<code>\tl_head:n {<token list>}</code>
<code>\tl_head:n</code> ★	Leaves in the input stream the first <i><item></i> in the <i><token list></i> , discarding the rest of the <i><token list></i> . All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example
<code>\tl_head:(V v f)</code> ★	
Updated: 2012-09-09	

`\tl_head:n { abc }`

and

`\tl_head:n { ~ abc }`

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

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

yields `␣ab`. A blank *<token list>* (see `\tl_if_blank:nTF`) will result 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 will not expand further when appearing in an **x**-type argument expansion.

<code>\tl_head:w</code>	★	<code>\tl_head:w <token list> { } \q_stop</code>
-------------------------	---	--

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) will result in a low-level T_EX 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.

<code>\tl_tail:N</code>	★	<code>\tl_tail:n {<token list>}</code>
<code>\tl_tail:n</code>	★	
<code>\tl_tail:(V v f)</code>	★	

Updated: 2012-09-01

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 }`

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

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

<code>\tl_if_head_eq_catcode_p:nN</code>	★	<code>\tl_if_head_eq_catcode_p:nN {<token list>} <test token></code>
<code>\tl_if_head_eq_catcode:nNTF</code>	★	<code>\tl_if_head_eq_catcode:nNTF {<token list>} <test token></code>
		<code>{<true code>} {<false code>}</code>

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 will always be **false**.

<code>\tl_if_head_eq_charcode_p:nN</code>	★	<code>\tl_if_head_eq_charcode_p:nN {<token list>} <test token></code>
<code>\tl_if_head_eq_charcode_p:fN</code>	★	<code>\tl_if_head_eq_charcode:nNTF {<token list>} <test token></code>
<code>\tl_if_head_eq_charcode:nNTF</code>	★	<code>{<true code>} {<false code>}</code>
<code>\tl_if_head_eq_charcode:fNTF</code>	★	

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 will always be **false**.

<code>\tl_if_head_eq_meaning_p:nN</code>	★	<code>\tl_if_head_eq_meaning_p:nN {<token list>} <test token></code>
<code>\tl_if_head_eq_meaning:nNTF</code>	★	<code>\tl_if_head_eq_meaning:nNTF {<token list>} <test token></code>
		<code>{<true code>} {<false code>}</code>

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 will always be **false**.

<code>\tl_if_head_is_group_p:n</code>	★	<code>\tl_if_head_is_group_p:n {⟨token list⟩}</code>
<code>\tl_if_head_is_group:nTF</code>	★	<code>\tl_if_head_is_group:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>

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.

<code>\tl_if_head_is_N_type_p:n</code>	★	<code>\tl_if_head_is_N_type_p:n {⟨token list⟩}</code>
<code>\tl_if_head_is_N_type:nTF</code>	★	<code>\tl_if_head_is_N_type:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>

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.

<code>\tl_if_head_is_space_p:n</code>	★	<code>\tl_if_head_is_space_p:n {⟨token list⟩}</code>
<code>\tl_if_head_is_space:nTF</code>	★	<code>\tl_if_head_is_space:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>

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

<code>\tl_item:nn</code>	★	<code>\tl_item:nn {⟨token list⟩} {⟨integer expression⟩}</code>
<code>\tl_item:Nn</code>	★	
<code>\tl_item:cn</code>	★	

New: 2014-07-17

Indexing items in the *⟨token list⟩* from 1 on the left, this function will evaluate the *⟨integer expression⟩* and leave the appropriate item from the *⟨token list⟩* in the input stream. If the *⟨integer expression⟩* 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 *⟨item⟩* will not expand further when appearing in an x-type argument expansion.

11 Viewing token lists

<code>\tl_show:N</code>	<code>\tl_show:N ⟨tl var⟩</code>
<code>\tl_show:c</code>	

Updated: 2015-08-01

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.

<hr/> <code>\tl_show:n</code> <hr/>	<code>\tl_show:n</code> $\langle token\ list \rangle$
Updated: 2015-08-07	Displays the $\langle token\ list \rangle$ on the terminal.

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

12 Constant token lists

<hr/> <code>\c_empty_tl</code> <hr/>	Constant that is always empty.
<hr/> <code>\c_space_tl</code> <hr/>	An explicit space character contained in a token list (compare this with <code>\c_space_token</code>). For use where an explicit space is required.

13 Scratch token lists

<hr/> <code>\l_tmpa_tl</code> <hr/> <code>\l_tmpb_tl</code> <hr/>	Scratch token 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.
<hr/> <code>\g_tmpa_tl</code> <hr/> <code>\g_tmpb_tl</code> <hr/>	Scratch token 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.

14 Internal functions

<hr/> <code>_tl_trim_spaces:nn</code> <hr/>	<code>_tl_trim_spaces:nn { \q_mark $\langle token\ list \rangle$ } {$\langle continuation \rangle$}</code> This function removes all leading and trailing explicit space characters from the $\langle token\ list \rangle$, and expands to the $\langle continuation \rangle$, followed by a brace group containing <code>\use_none:n \q_mark $\langle trimmed\ token\ list \rangle$</code> . For instance, <code>\tl_trim_spaces:n</code> is implemented by taking the $\langle continuation \rangle$ to be <code>\exp_not:o</code> , and the o-type expansion removes the <code>\q_mark</code> . This function is also used in <code>l3clist</code> and <code>l3candidates</code> .
--	--

Part XII

The l3str 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 expl3 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 will simply be 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 `\tl_to_str:n` for internal processing, and will not treat a token list or the corresponding string representation differently.

Note that as string variables are a special case of token list variables the coverage of `\str_...:N` functions is somewhat smaller than `\tl_...:N`.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) will generate strings from the appropriate input: these are documented in `l3basics`, `l3tl` and `l3token`, 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` $\langle str\ var \rangle$
Creates a new $\langle str\ var \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle str\ var \rangle$ will initially be empty.

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

New: 2015-09-18

`\str_const:Nn` $\langle str\ var \rangle$ $\{ \langle token\ list \rangle \}$
Creates a new constant $\langle str\ var \rangle$ or raises an error if the name is already taken. The value of the $\langle str\ var \rangle$ will be set globally to the $\langle token\ list \rangle$, converted to a string.

<code>\str_clear:N</code>	<code>\str_clear:N <str var></code>
<code>\str_clear:c</code>	
<code>\str_gclear:N</code>	Clears the content of the <code><str var></code> .
<code>\str_gclear:c</code>	
<hr/>	
New: 2015-09-18	

<code>\str_clear_new:N</code>	<code>\str_clear_new:N <str var></code>
<code>\str_clear_new:c</code>	
	Ensures that the <code><str var></code> exists globally by applying <code>\str_new:N</code> if necessary, then applies <code>\str_(g)clear:N</code> to leave the <code><str var></code> empty.
<hr/>	
New: 2015-09-18	

<code>\str_set_eq:NN</code>	<code>\str_set_eq:NN <str var₁> <str var₂></code>
<code>\str_set_eq:(cN Nc cc)</code>	
<code>\str_gset_eq:NN</code>	Sets the content of <code><str var₁></code> equal to that of <code><str var₂></code> .
<code>\str_gset_eq:(cN Nc cc)</code>	
<hr/>	
New: 2015-09-18	

2 Adding data to string variables

<code>\str_set:Nn</code>	<code>\str_set:Nn <str var> {<token list>}</code>
<code>\str_set:(Nx cn cx)</code>	
<code>\str_gset:Nn</code>	Converts the <code><token list></code> to a <code><string></code> , and stores the result in <code><str var></code> .
<code>\str_gset:(Nx cn cx)</code>	
<hr/>	
New: 2015-09-18	

<code>\str_put_left:Nn</code>	<code>\str_put_left:Nn <str var> {<token list>}</code>
<code>\str_put_left:(Nx cn cx)</code>	
<code>\str_gput_left:Nn</code>	Converts the <code><token list></code> to a <code><string></code> , and prepends the result to <code><str var></code> . The current contents of the <code><str var></code> are not automatically converted to a string.
<code>\str_gput_left:(Nx cn cx)</code>	
<hr/>	
New: 2015-09-18	

<code>\str_put_right:Nn</code>	<code>\str_put_right:Nn <str var> {<token list>}</code>
<code>\str_put_right:(Nx cn cx)</code>	
<code>\str_gput_right:Nn</code>	Converts the <code><token list></code> to a <code><string></code> , and appends the result to <code><str var></code> . The current contents of the <code><str var></code> are not automatically converted to a string.
<code>\str_gput_right:(Nx cn cx)</code>	
<hr/>	
New: 2015-09-18	

2.1 String conditionals

<code>\str_if_exist_p:N</code> ★	<code>\str_if_exist_p:N <str var></code>
<code>\str_if_exist_p:c</code> ★	<code>\str_if_exist:NTF <str var> {<true code>} {<false code>}</code>
<code>\str_if_exist:NTF</code> ★	
<code>\str_if_exist:cTF</code> ★	Tests whether the <code><str var></code> is currently defined. This does not check that the <code><str var></code> really is a string.
<hr/>	
New: 2015-09-18	

<code>\str_if_empty_p:N</code>	★	<code>\str_if_empty_p:N</code> $\langle \text{str var} \rangle$
<code>\str_if_empty_p:c</code>	★	<code>\str_if_empty:N</code> TF $\langle \text{str var} \rangle$ $\{\langle \text{true code} \rangle\}$ $\{\langle \text{false code} \rangle\}$
<code>\str_if_empty:N</code> TF	★	Tests if the $\langle \text{string variable} \rangle$ is entirely empty (<i>i.e.</i> contains no characters at all).
<code>\str_if_empty:c</code> TF	★	

New: 2015-09-18

<code>\str_if_eq_p:NN</code>	★	<code>\str_if_eq_p:NN</code> $\langle \text{str var}_1 \rangle$ $\langle \text{str var}_2 \rangle$
<code>\str_if_eq_p:(Nc cN cc)</code>	★	<code>\str_if_eq:NN</code> TF $\langle \text{str var}_1 \rangle$ $\langle \text{str var}_2 \rangle$ $\{\langle \text{true code} \rangle\}$ $\{\langle \text{false code} \rangle\}$
<code>\str_if_eq:NN</code> TF	★	Compares the content of two $\langle \text{str variables} \rangle$ and is logically true if the two contain the same characters.
<code>\str_if_eq:(Nc cN cc)</code> TF	★	

New: 2015-09-18

<code>\str_if_eq_p:nn</code>	★	<code>\str_if_eq_p:nn</code> $\{\langle \text{tl}_1 \rangle\}$ $\{\langle \text{tl}_2 \rangle\}$
<code>\str_if_eq_p:(Vn on no nV VV)</code>	★	<code>\str_if_eq:nn</code> TF $\{\langle \text{tl}_1 \rangle\}$ $\{\langle \text{tl}_2 \rangle\}$ $\{\langle \text{true code} \rangle\}$ $\{\langle \text{false code} \rangle\}$
<code>\str_if_eq:nn</code> TF	★	Compares the two $\langle \text{token lists} \rangle$ on a character by character basis, and is true if the two lists contain the same characters in the same order. Thus for example
<code>\str_if_eq:(Vn on no nV VV)</code> TF	★	

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

is logically true.

<code>\str_if_eq_x_p:nn</code>	★	<code>\str_if_eq_x_p:nn</code> $\{\langle \text{tl}_1 \rangle\}$ $\{\langle \text{tl}_2 \rangle\}$
<code>\str_if_eq_x:nn</code> TF	★	<code>\str_if_eq_x:nn</code> TF $\{\langle \text{tl}_1 \rangle\}$ $\{\langle \text{tl}_2 \rangle\}$ $\{\langle \text{true code} \rangle\}$ $\{\langle \text{false code} \rangle\}$

New: 2012-06-05

Compares the full expansion of two $\langle \text{token lists} \rangle$ on a character by character basis, and is true if the two lists contain the same characters in the same order. Thus for example

`\str_if_eq_x_p:nn { abc } { \tl_to_str:n { abc } }`

is logically true.

<code>\str_case:nn</code>	★	<code>\str_case:nn</code> TF $\{\langle \text{test string} \rangle\}$
<code>\str_case:(on nV nv)</code>	★	{
<code>\str_case:nn</code> TF	★	$\{\langle \text{string case}_1 \rangle\}$ $\{\langle \text{code case}_1 \rangle\}$
<code>\str_case:(on nV nv)</code> TF	★	$\{\langle \text{string case}_2 \rangle\}$ $\{\langle \text{code case}_2 \rangle\}$
		...
		$\{\langle \text{string case}_n \rangle\}$ $\{\langle \text{code case}_n \rangle\}$
		}
		$\{\langle \text{true code} \rangle\}$
		$\{\langle \text{false code} \rangle\}$

New: 2013-07-24

Updated: 2015-02-28

This function compares the $\langle \text{test string} \rangle$ in turn with each of the $\langle \text{string cases} \rangle$. If the two are equal (as described for `\str_if_eq:nnTF` then the associated $\langle \text{code} \rangle$ is left in the input stream. If any of the cases are matched, the $\langle \text{true code} \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle \text{false code} \rangle$ is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

<hr/> <code>\str_case_x:nnTF</code> ★ <hr/>	<code>\str_case_x:nnTF</code> $\{ \langle test\ string \rangle \}$
New: 2013-07-24	$\{$ $\{ \langle string\ case_1 \rangle \} \{ \langle code\ case_1 \rangle \}$ $\{ \langle string\ case_2 \rangle \} \{ \langle code\ case_2 \rangle \}$ \dots $\{ \langle string\ case_n \rangle \} \{ \langle code\ case_n \rangle \}$ $\}$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$

This function compares the full expansion of the $\langle test\ string \rangle$ in turn with the full expansion of the $\langle string\ cases \rangle$. If the two full expansions are equal (as described for `\str_if_eq:nnTF`) then the associated $\langle code \rangle$ is left in the input stream. If any of the cases are matched, the $\langle true\ code \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle false\ code \rangle$ is inserted. The function `\str_case_x:nn`, which does nothing if there is no match, is also available. The $\langle test\ string \rangle$ is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

3 Working with the content of strings

<hr/> <code>\str_use:N</code> ★ <hr/>	<code>\str_use:N</code> $\langle str\ var \rangle$
<code>\str_use:c</code> ★	
New: 2015-09-18	

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

<hr/> <code>\str_count:N</code> ★ <hr/>	<code>\str_count:n</code> $\{ \langle token\ list \rangle \}$
<code>\str_count:c</code> ★	
<code>\str_count:n</code> ★	
<code>\str_count_ignore_spaces:n</code> ★	
New: 2015-09-18	

Leaves in the input stream the number of characters in the string representation of $\langle token\ list \rangle$, 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.

<hr/> <code>\str_count_spaces:N</code> ★ <hr/>	<code>\str_count_spaces:n</code> $\{ \langle token\ list \rangle \}$
<code>\str_count_spaces:c</code> ★	
<code>\str_count_spaces:n</code> ★	
New: 2015-09-18	

Leaves in the input stream the number of space characters in the string representation of $\langle token\ list \rangle$, as an integer denotation. Of course, this function has no `_ignore_spaces` variant.

<code>\str_head:N</code>	★	<code>\str_head:n {⟨token list⟩}</code>
<code>\str_head:c</code>	★	
<code>\str_head:n</code>	★	
<code>\str_head_ignore_spaces:n</code>	★	

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 `\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.

<code>\str_tail:N</code>	★	<code>\str_tail:n {⟨token list⟩}</code>
<code>\str_tail:c</code>	★	
<code>\str_tail:n</code>	★	
<code>\str_tail_ignore_spaces:n</code>	★	

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` will trim only that space, while `\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.

<code>\str_item:Nn</code>	★	<code>\str_item:nn {⟨token list⟩} {⟨integer expression⟩}</code>
<code>\str_item:nn</code>	★	
<code>\str_item_ignore_spaces:nn</code>	★	

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 `\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 {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}
\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. Positive $\langle indices \rangle$ are counted from the start of the string, 1 being the first character, and negative $\langle indices \rangle$ are counted from the end of the string, -1 being the last character. If either of $\langle start\ index \rangle$ or $\langle end\ index \rangle$ is 0, the result is empty. 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 } }

```

will print `bcde`, `cdef`, `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.

4 String manipulation

<code>\str_lower_case:n</code>	☆
<code>\str_lower_case:f</code>	☆
<code>\str_upper_case:n</code>	☆
<code>\str_upper_case:f</code>	☆

New: 2015-03-01

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

`\str_upper_case: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:cpn
  {
    user
    \str_upper_case:f { \tl_head:n {#1} }
    \str_lower_case: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_fold_case:n` for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the `\tl_lower_case:n(n)`, `\tl_upper_case:n(n)` and `\tl_mixed_case:n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

T_EXhackers note: As with all `expl3` functions, the input supported by `\str_fold_case:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfT_EX *only* the Latin alphabet characters A–Z will be case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both X_ƎT_EX and LuaT_EX, subject only to the fact that X_ƎT_EX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

`\str_fold_case:n` ★ `\str_fold_case:n {(tokens)}`

`\str_fold_case:V` ★

New: 2014-06-19

Updated: 2016-03-07

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_fold_case: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_fold_case:n` follows the “full” scheme defined by the Unicode Consortium (*e.g.* `SS` folds 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_fold_case: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 will be 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`.

5 Viewing strings

`\str_show:N` `\str_show:N {str var}`

`\str_show:c`

`\str_show:n`

New: 2015-09-18

Displays the content of the $\langle str var \rangle$ on the terminal.

6 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

```

New: 2015-09-19

Constant strings, containing a single character token, with category code 12.

7 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.

7.1 Internal string functions

```

\__str_if_eq_x:nn ★ \__str_if_eq_x:nn {\t1} {\t2}

```

Compares the full expansion of two *<token lists>* on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Leaves 0 in the input stream if the condition is true, and +1 or -1 otherwise.

```

\__str_if_eq_x_return:nn \__str_if_eq_x_return:nn {\t1} {\t2}

```

Compares the full expansion of two *<token lists>* on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Either `\prg_return_true:` or `\prg_return_false:` is then left in the input stream. This is a version of `\str_if_eq_x:nnTF` coded for speed.

```

\__str_to_other:n ★ \__str_to_other:n {\token list}

```

Converts the *<token list>* to a *<other string>*, 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.

<hr/> <hr/>	<code>_str_count:n</code> ★	<code>_str_count:n {⟨other string⟩}</code>	This function expects an argument that is entirely made of characters with category “other”, as produced by <code>_str_to_other:n</code> . It leaves in the input stream the number of character tokens in the <i>⟨other string⟩</i> , faster than the analogous <code>\str_count:n</code> function.
<hr/> <hr/>	<code>_str_range:nnn</code> ★	<code>_str_range:nnn {⟨other string⟩} {⟨start index⟩} {⟨end index⟩}</code>	Identical to <code>\str_range:nnn</code> except that the first argument is expected to be entirely made of characters with category “other”, as produced by <code>_str_to_other:n</code> , and the result is also an <i>⟨other string⟩</i> .

Part XIII

The l3seq package

Sequences and stacks

L^AT_EX3 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 L^AT_EX3. This is achieved using a number of dedicated stack functions.

1 Creating and initialising sequences

<code>\seq_new:N</code>	<code>\seq_new:N <sequence></code>
<code>\seq_new:c</code>	

Creates a new *⟨sequence⟩* or raises an error if the name is already taken. The declaration is global. The *⟨sequence⟩* will initially contain no items.

<code>\seq_clear:N</code>	<code>\seq_clear:N <sequence></code>
<code>\seq_clear:c</code>	
<code>\seq_gclear:N</code>	
<code>\seq_gclear:c</code>	

Clears all items from the *⟨sequence⟩*.

<code>\seq_clear_new:N</code>	<code>\seq_clear_new:N <sequence></code>
<code>\seq_clear_new:c</code>	
<code>\seq_gclear_new:N</code>	
<code>\seq_gclear_new:c</code>	

Ensures that the *⟨sequence⟩* exists globally by applying `\seq_new:N` if necessary, then applies `\seq_(g)clear:N` to leave the *⟨sequence⟩* empty.

<code>\seq_set_eq:NN</code>	<code>\seq_set_eq:NN <sequence₁> <sequence₂></code>
<code>\seq_set_eq:(cN Nc cc)</code>	
<code>\seq_gset_eq:NN</code>	
<code>\seq_gset_eq:(cN Nc cc)</code>	

Sets the content of *⟨sequence₁⟩* equal to that of *⟨sequence₂⟩*.

<code>\seq_set_from_clist:NN</code>	<code>\seq_set_from_clist:NN <sequence> <comma-list></code>
<code>\seq_set_from_clist:(cN Nc cc)</code>	
<code>\seq_set_from_clist:Nn</code>	
<code>\seq_set_from_clist:cn</code>	
<code>\seq_gset_from_clist:NN</code>	
<code>\seq_gset_from_clist:(cN Nc cc)</code>	
<code>\seq_gset_from_clist:Nn</code>	
<code>\seq_gset_from_clist:cn</code>	

New: 2014-07-17

Converts the data in the *⟨comma list⟩* into a *⟨sequence⟩*: the original *⟨comma list⟩* is unchanged.

```
\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 $\langle token list \rangle$ into $\langle items \rangle$ separated by $\langle delimiter \rangle$, and assigns the result to the $\langle sequence \rangle$. Spaces on both sides of each $\langle item \rangle$ are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of `l3clist` functions. Empty $\langle items \rangle$ are preserved by `\seq_set_split:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn <sequence> {<>}`. The $\langle delimiter \rangle$ may not contain `{`, `}` or `#` (assuming \TeX 's normal category code régime). If the $\langle delimiter \rangle$ is empty, the $\langle token list \rangle$ is split into $\langle items \rangle$ as a $\langle token list \rangle$.

```
\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc
```

```
\seq_concat:NNN <sequence1> <sequence2> <sequence3>
```

Concatenates the content of $\langle sequence_2 \rangle$ and $\langle sequence_3 \rangle$ together and saves the result in $\langle sequence_1 \rangle$. The items in $\langle sequence_2 \rangle$ will be 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 $\langle sequence \rangle$ is currently defined. This does not check that the $\langle sequence \rangle$ really is a sequence variable.

2 Appending data to sequences

```
\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 $\langle item \rangle$ to the left of the $\langle sequence \rangle$.

```
\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 $\langle item \rangle$ to the right of the $\langle sequence \rangle$.

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 $\langle token list variable \rangle$ 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 $\langle sequence \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token list variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ will contain the special marker `\q_no_value`.

<hr/> <code>\seq_get_right:NN</code> <code>\seq_get_right:cN</code> <hr/> Updated: 2012-05-19	<code>\seq_get_right:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Stores the right-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token list variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ will contain the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_pop_left:NN</code> <code>\seq_pop_left:cN</code> <hr/> Updated: 2012-05-14	<code>\seq_pop_left:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ will contain the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_gpop_left:NN</code> <code>\seq_gpop_left:cN</code> <hr/> Updated: 2012-05-14	<code>\seq_gpop_left:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ will contain the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_pop_right:NN</code> <code>\seq_pop_right:cN</code> <hr/> Updated: 2012-05-19	<code>\seq_pop_right:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ will contain the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_gpop_right:NN</code> <code>\seq_gpop_right:cN</code> <hr/> Updated: 2012-05-19	<code>\seq_gpop_right:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ will contain the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_item:Nn</code> ★ <code>\seq_item:cn</code> ★ <hr/> New: 2014-07-17	<code>\seq_item:Nn</code> $\langle sequence \rangle$ $\{ \langle integer expression \rangle \}$ Indexing items in the $\langle sequence \rangle$ from 1 at the top (left), this function will evaluate the $\langle integer expression \rangle$ and leave the appropriate item from the sequence in the input stream. If the $\langle integer expression \rangle$ is negative, indexing occurs from the bottom (right) of the sequence. When the $\langle integer expression \rangle$ is larger than the number of items in the $\langle sequence \rangle$ (as calculated by <code>\seq_count:N</code>) then the function will expand to nothing.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ will 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:cNTF`

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 a `<sequence>` in the `<token list variable>` without removing it from a `<sequence>`. The `<token list variable>` is assigned locally.

`\seq_get_right:NNTF`
`\seq_get_right:cNTF`

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 a `<sequence>` in the `<token list variable>` without removing it from a `<sequence>`. The `<token list variable>` is assigned locally.

`\seq_pop_left:NNTF`
`\seq_pop_left:cNTF`

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 a `<sequence>` in the `<token list variable>`, i.e. removes the item from a `<sequence>`. Both the `<sequence>` and the `<token list variable>` are assigned locally.

`\seq_gpop_left:NNTF`
`\seq_gpop_left:cNTF`

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 a `<sequence>` in the `<token list variable>`, i.e. removes the item from a `<sequence>`. 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 a `<sequence>` in the `<token list variable>`, i.e. removes the item from a `<sequence>`. 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 a `<sequence>` in the `<token list variable>`, i.e. removes the item from a `<sequence>`. 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 $\langle sequence \rangle$, leaving the left most copy of each item in the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

TeXhackers note: This function iterates through every item in the $\langle sequence \rangle$ and does a comparison with the $\langle items \rangle$ 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 $\langle item \rangle$ from the $\langle sequence \rangle$. The $\langle item \rangle$ 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
```

```
\seq_reverse:N <sequence>
```

Reverses the order of the items stored in the $\langle sequence \rangle$.

New: 2014-07-18

```
\seq_sort:Nn
\seq_sort:cn
\seq_gsort:Nn
\seq_gsort:cn
```

```
\seq_sort:Nn <sequence> {<comparison code>}
```

Sorts the items in the $\langle sequence \rangle$ according to the $\langle comparison code \rangle$, and assigns the result to $\langle sequence \rangle$. The details of sorting comparison are described in Section 1.

New: 2017-02-06

6 Sequence conditionals

```
\seq_if_empty_p:N ★
\seq_if_empty_p:c ★
\seq_if_empty:NTF ★
\seq_if_empty:cTF ★
```

```
\seq_if_empty_p:N <sequence>
```

```
\seq_if_empty:NNTF <sequence> {<true code>} {<false code>}
```

Tests if the $\langle sequence \rangle$ 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> {<item>} {<true code>} {<false code>}
```

Tests if the $\langle item \rangle$ is present in the $\langle sequence \rangle$.

7 Mapping to sequences

```
\seq_map_function:NN ★
\seq_map_function:cN ★
```

```
\seq_map_function:NN <sequence> <function>
```

Applies $\langle function \rangle$ to every $\langle item \rangle$ stored in the $\langle sequence \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. The $\langle items \rangle$ are returned from left to right. The function `\seq_map_inline:Nn` is faster than `\seq_map_function:NN` for sequences with more than about 10 items. One mapping may be nested inside another.

Updated: 2012-06-29

<hr/> <code>\seq_map_inline:Nn</code> <hr/>	<code>\seq_map_inline:Nn <sequence> {<inline function>}</code>
<code>\seq_map_inline:cn</code> <hr/>	Applies <i><inline function></i> to every <i><item></i> stored within the <i><sequence></i> . The <i><inline function></i> should consist of code which will receive the <i><item></i> as #1. One in line mapping can be nested inside another. The <i><items></i> are returned from left to right.
Updated: 2012-06-29	

<hr/> <code>\seq_map_variable:NNn</code> <hr/>	<code>\seq_map_variable:NNn <sequence> <tl var.> {<function using tl var.>}</code>
<code>\seq_map_variable:(Ncn cNn ccn)</code> <hr/>	Stores each entry in the <i><sequence></i> in turn in the <i><tl var.></i> and applies the <i><function using tl var.></i> The <i><function></i> will usually consist of code making use of the <i><tl var.></i> , but this is not enforced. One variable mapping can be nested inside another. The <i><items></i> are returned from left to right.
Updated: 2012-06-29	

<hr/> <code>\seq_map_break: ☆</code> <hr/>	<code>\seq_map_break:</code>
Updated: 2012-06-29	Used to terminate a <code>\seq_map...</code> function before all entries in the <i><sequence></i> have been processed. This will normally take 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 will lead to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before further items are taken from the input stream. This will depend on the design of the mapping function.

`\seq_map_break:n` ☆

Updated: 2012-06-29

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

Used to terminate a `\seq_map...` function before all entries in the *<sequence>* have been processed, inserting the *<tokens>* after the mapping has ended. This will normally take place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \seq_map_break:n { <tokens> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\seq_map...` scenario will lead to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before the *<tokens>* are inserted into the input stream. This will depend on the design of the mapping function.

`\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>* will include 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>* {*<separator between two>*}

{*<separator between more than two>*} {*<separator between final two>*}

Places the contents of the *<seq var>* in the input stream, with the appropriate *<separator>* between the items. Namely, if the sequence 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 sequence has exactly two items, then they are placed in the input stream separated by the *<separator 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 will be 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~ }
```

will insert “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.

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

`\seq_use:Nn` ★
`\seq_use:cn` ★

New: 2013-05-26

`\seq_use:Nn` $\langle seq\ var \rangle$ $\{\langle separator \rangle\}$

Places the contents of the $\langle seq\ var \rangle$ in the input stream, with the $\langle separator \rangle$ between the items. If the sequence has a single item, it is placed in the input stream with no $\langle separator \rangle$, and an empty sequence produces no output. An error will be 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~ }
```

will insert “a and b and c and de and f” in the input stream.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle items \rangle$ will 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` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$

Reads the top item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ will contain the special marker `\q_no_value`.

`\seq_pop:NN`
`\seq_pop:cn`

Updated: 2012-05-14

`\seq_pop:NN` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$

Pops the top item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ will contain the special marker `\q_no_value`.

`\seq_gpop:NN`
`\seq_gpop:cn`

Updated: 2012-05-14

`\seq_gpop:NN` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$

Pops the top item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token\ list\ variable \rangle$ will contain the special marker `\q_no_value`.

`\seq_get:NNTF`
`\seq_get:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_get:NNTF` $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

If the $\langle sequence \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle token\ list\ variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle sequence \rangle$ is non-empty, stores the top item from a $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token\ list\ variable \rangle$ 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 $\langle seq\ var_3 \rangle$ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence $\backslash l_ \langle pkg \rangle_internal_seq$, then $\langle seq\ var_3 \rangle$ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ through

```

\seq_concat:NNN <seq var3> <seq var1> <seq var2>
\seq_remove_duplicates:N <seq var3>

```

or by adding items to (a copy of) $\langle seq\ var_1 \rangle$ 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 $\langle seq\ var_2 \rangle$ is short compared to $\langle seq\ var_1 \rangle$.

The difference of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by removing items of the $\langle seq\ var_2 \rangle$ from (a copy of) the $\langle seq\ var_1 \rangle$ 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 $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by computing the difference between $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ and storing the result as $\backslash l_ \langle pkg \rangle_internal_seq$, then the difference between $\langle seq\ var_2 \rangle$ and $\langle seq\ var_1 \rangle$, and finally concatenating the two differences to get the symmetric differences.

```

\seq_set_eq:NN \l\_ \langle pkg \rangle\_internal\_seq <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn \l\_ \langle pkg \rangle\_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 pkg \rangle\_internal\_seq

```

11 Constant and scratch sequences

$\backslash c_empty_seq$ 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` $\langle sequence \rangle$
Displays the entries in the $\langle sequence \rangle$ in the terminal.

13 Internal sequence functions

`\s__seq`

This scan mark (equal to `\scan_stop:`) marks the beginning of a sequence variable.

`__seq_item:n` ★

`__seq_item:n` $\{\langle item \rangle\}$

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error will be raised. The definition should always be set globally.

`__seq_push_item_def:n`
`__seq_push_item_def:x`

`__seq_push_item_def:n` $\{\langle code \rangle\}$

Saves the definition of `__seq_item:n` and redefines it to accept one parameter and expand to $\langle code \rangle$. 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`.

Part XIV

The l3clist package

Comma separated lists

Comma lists contain ordered data where items can be added to the left or right end of the list. The resulting ordered list can then be mapped over using `\clist_map_function:NN`. Several items can be added at once, and spaces are removed from both sides of each item on input. Hence,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~ a ~ , ~ {b} ~ }
\clist_put_right:Nn \l_my_clist { ~ { c ~ } , d }
```

results in `\l_my_clist` containing `a,{b},{c~},d`. Comma lists cannot contain empty items, thus

```
\clist_clear_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

will leave `true` in the input stream. To include an item which contains a comma, or starts or ends with a space, surround it with braces. The sequence data type should be preferred to comma lists if items are to contain `{`, `}`, or `#` (assuming the usual \TeX category codes apply).

1 Creating and initialising comma lists

```
\clist_new:N
\clist_new:c
```

```
\clist_new:N <comma list>
```

Creates a new *<comma list>* or raises an error if the name is already taken. The declaration is global. The *<comma list>* will initially contain no items.

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

```
\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>* will be set globally to the *<comma list>*.

New: 2014-07-05

```
\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.

<code>\clist_set_eq:NN</code>	<code>\clist_set_eq:NN <comma list₁> <comma list₂></code>
<code>\clist_set_eq:(cN Nc cc)</code>	Sets the content of <code><comma list₁></code> equal to that of <code><comma list₂></code> .
<code>\clist_gset_eq:NN</code>	
<code>\clist_gset_eq:(cN Nc cc)</code>	

<code>\clist_set_from_seq:NN</code>	<code>\clist_set_from_seq:NN <comma list> <sequence></code>
<code>\clist_set_from_seq:(cN Nc cc)</code>	
<code>\clist_gset_from_seq:NN</code>	
<code>\clist_gset_from_seq:(cN Nc cc)</code>	

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.

<code>\clist_concat:NNN</code>	<code>\clist_concat:NNN <comma list₁> <comma list₂> <comma list₃></code>
<code>\clist_concat:ccc</code>	Concatenates the content of <code><comma list₂></code> and <code><comma list₃></code> together and saves the result in <code><comma list₁></code> . The items in <code><comma list₂></code> will be placed at the left side of the new comma list.
<code>\clist_gconcat:NNN</code>	
<code>\clist_gconcat:ccc</code>	

<code>\clist_if_exist_p:N *</code>	<code>\clist_if_exist_p:N <comma list></code>
<code>\clist_if_exist_p:c *</code>	<code>\clist_if_exist:NTF <comma list> {\true code} {\false code}</code>
<code>\clist_if_exist:NTF *</code>	Tests whether the <code><comma list></code> is currently defined. This does not check that the <code><comma list></code> really is a comma list.
<code>\clist_if_exist:cTF *</code>	

New: 2012-03-03

2 Adding data to comma lists

<code>\clist_set:Nn</code>	<code>\clist_set:Nn <comma list> {\<item₁>,...,<item_n>}</code>
<code>\clist_set:(NV No Nx cn cV co cx)</code>	
<code>\clist_gset:Nn</code>	
<code>\clist_gset:(NV No Nx cn cV co cx)</code>	

New: 2011-09-06

Sets `<comma list>` to contain the `<items>`, removing any previous content from the variable. Spaces are removed from both sides of each item.

<code>\clist_put_left:Nn</code>	<code>\clist_put_left:Nn <comma list> {\<item₁>,...,<item_n>}</code>
<code>\clist_put_left:(NV No Nx cn cV co cx)</code>	
<code>\clist_gput_left:Nn</code>	
<code>\clist_gput_left:(NV No Nx cn cV co cx)</code>	

Updated: 2011-09-05

Appends the `<items>` to the left of the `<comma list>`. Spaces are removed from both sides of each item.

<code>\clist_put_right:Nn</code>	<code>\clist_put_right:Nn <comma list> {\<item_1>,...,<item_n>}</code>
<code>\clist_put_right:(NV No Nx cn cV co cx)</code>	
<code>\clist_gput_right:Nn</code>	
<code>\clist_gput_right:(NV No Nx cn cV co cx)</code>	

Updated: 2011-09-05

Appends the $\langle items \rangle$ to the right of the $\langle comma list \rangle$. Spaces are removed from both sides of each item.

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.

<code>\clist_remove_duplicates:N</code>	<code>\clist_remove_duplicates:N <comma list></code>
<code>\clist_remove_duplicates:c</code>	
<code>\clist_gremove_duplicates:N</code>	
<code>\clist_gremove_duplicates:c</code>	

Removes duplicate items from the $\langle comma list \rangle$, leaving the left most copy of each item in the $\langle comma list \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

T_EXhackers 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 will not work if any of the items in the $\langle comma list \rangle$ contains `{`, `}`, or `#` (assuming the usual T_EX category codes apply).

<code>\clist_remove_all:Nn</code>	<code>\clist_remove_all:Nn <comma list> {\<item>}</code>
<code>\clist_remove_all:cn</code>	
<code>\clist_gremove_all:Nn</code>	
<code>\clist_gremove_all:cn</code>	

Updated: 2011-09-06

Removes every occurrence of $\langle item \rangle$ from the $\langle comma list \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

T_EXhackers note: The $\langle item \rangle$ may not contain `{`, `}`, or `#` (assuming the usual T_EX category codes apply).

<code>\clist_reverse:N</code>	<code>\clist_reverse:N <comma list></code>
<code>\clist_reverse:c</code>	
<code>\clist_greverse:N</code>	
<code>\clist_greverse:c</code>	

New: 2014-07-18

Reverses the order of items stored in the $\langle comma list \rangle$.

<code>\clist_reverse:n</code>	<code>\clist_reverse:n {\<comma list>}</code>
-------------------------------	---

New: 2014-07-18

Leaves the items in the $\langle comma list \rangle$ in the input stream in reverse order. Braces and spaces are preserved by this process.

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

<code>\clist_sort:Nn</code>	<code>\clist_sort:Nn <clist var> {<comparison code>}</code>
<code>\clist_sort:cn</code>	
<code>\clist_gsort:Nn</code>	Sorts the items in the <i><clist var></i> according to the <i><comparison code></i> , and assigns the result to <i><clist var></i> . The details of sorting comparison are described in Section 1.
<code>\clist_gsort:cn</code>	

New: 2017-02-06

4 Comma list conditionals

<code>\clist_if_empty_p:N</code> *	<code>\clist_if_empty_p:N <comma list></code>
<code>\clist_if_empty_p:c</code> *	<code>\clist_if_empty:NtF <comma list> {<true code>} {<false code>}</code>
<code>\clist_if_empty:NtF</code> *	
<code>\clist_if_empty:cTf</code> *	Tests if the <i><comma list></i> is empty (containing no items).

<code>\clist_if_empty_p:n</code> *	<code>\clist_if_empty_p:n {<comma list>}</code>
<code>\clist_if_empty:nTf</code> *	<code>\clist_if_empty:nTf {<comma list>} {<true code>} {<false code>}</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.

<code>\clist_if_in:NnTf</code>	<code>\clist_if_in:NnTf <comma list> {<item>} {<true code>} {<false code>}</code>
<code>\clist_if_in:(NV No cn cV co)Tf</code>	
<code>\clist_if_in:nnTf</code>	
<code>\clist_if_in:(nV no)Tf</code>	

Updated: 2011-09-06

Tests if the *<item>* is present in the *<comma list>*. In the case of an n-type *<comma list>*, spaces are stripped from each item, but braces are not removed. Hence,

`\clist_if_in:nnTf { a , {b}~ , {b} , c } { b } {true} {false}`

yields false.

TeXhackers note: The *<item>* may not contain `{`, `}`, or `#` (assuming the usual TeX category codes apply), and should not contain `,` nor start or end with a space.

5 Mapping to comma lists

The functions described in this section apply a specified function to each item of a comma list.

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 your 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:cN` ☆
`\clist_map_function:nN` ☆

Updated: 2012-06-29

`\clist_map_function:NN` $\langle comma list \rangle$ $\langle function \rangle$

Applies $\langle function \rangle$ to every $\langle item \rangle$ stored in the $\langle comma list \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. The $\langle items \rangle$ are returned from left to right. The function `\clist_map_inline:Nn` is in general more efficient than `\clist_map_function:NN`. One mapping may be nested inside another.

`\clist_map_inline:Nn`
`\clist_map_inline:cn`
`\clist_map_inline:nn`

Updated: 2012-06-29

`\clist_map_inline:Nn` $\langle comma list \rangle$ $\{ \langle inline function \rangle \}$

Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle comma list \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. One in line mapping can be nested inside another. The $\langle items \rangle$ are returned from left to right.

`\clist_map_variable:NNn`
`\clist_map_variable:cNn`
`\clist_map_variable:nNn`

Updated: 2012-06-29

`\clist_map_variable:NNn` $\langle comma list \rangle$ $\langle tl var. \rangle$ $\{ \langle function using tl var. \rangle \}$

Stores each entry in the $\langle comma list \rangle$ in turn in the $\langle tl var. \rangle$ and applies the $\langle function using tl var. \rangle$. The $\langle function \rangle$ will usually consist of code making use of the $\langle tl var. \rangle$, but this is not enforced. One variable mapping can be nested inside another. The $\langle items \rangle$ are returned from left to right.

`\clist_map_break:` ☆

Updated: 2012-06-29

`\clist_map_break:`

Used to terminate a `\clist_map...` function before all entries in the $\langle comma list \rangle$ have been processed. This will normally take 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 will lead to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before further items are taken from the input stream. This will depend on the design of the mapping function.

`\clist_map_break:n` ☆

Updated: 2012-06-29

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

Used to terminate a `\clist_map...` function before all entries in the *<comma list>* have been processed, inserting the *<tokens>* after the mapping has ended. This will normally take place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break:n { <tokens> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\clist_map...` scenario will lead to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before the *<tokens>* are inserted into the input stream. This will depend 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>* will include those which are duplicates, *i.e.* every item in a *<comma list>* is unique.

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 will be 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~ }
```

will insert “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>* will not expand further when appearing in an x-type argument expansion.

<code>\clist_use:Nn</code> ★	<code>\clist_use:Nn <clist var> {<separator>}</code>
<code>\clist_use:cn</code> ★	Places the contents of the <i><clist var></i> in the input stream, with the <i><separator></i> 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 will be raised if the variable does not exist or if it is invalid.
New: 2013-05-26	

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

will insert “a and b and c and de and f” in the input stream.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<items>* will 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.

<code>\clist_get:NN</code>	<code>\clist_get:NN <comma list> <token list variable></code>
<code>\clist_get:cN</code>	Stores the left-most item from a <i><comma list></i> in the <i><token list variable></i> without removing it from the <i><comma list></i> . The <i><token list variable></i> is assigned locally. If the <i><comma list></i> is empty the <i><token list variable></i> will contain the marker value <code>\q_no_value</code> .
Updated: 2012-05-14	

<code>\clist_get:NNTF</code>	<code>\clist_get:NNTF <comma list> <token list variable> {<true code>} {<false code>}</code>
<code>\clist_get:cNTF</code>	If the <i><comma list></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><comma list></i> is non-empty, stores the top item from the <i><comma list></i> in the <i><token list variable></i> without removing it from the <i><comma list></i> . The <i><token list variable></i> is assigned locally.
New: 2012-05-14	

<code>\clist_pop:NN</code>	<code>\clist_pop:NN <comma list> <token list variable></code>
<code>\clist_pop:cN</code>	Pops the left-most item from a <i><comma list></i> into the <i><token list variable></i> , <i>i.e.</i> removes the item from the comma list and stores it in the <i><token list variable></i> . Both of the variables are assigned locally.
Updated: 2011-09-06	

<code>\clist_gpop:NN</code>	<code>\clist_gpop:NN <comma list> <token list variable></code>
<code>\clist_gpop:cN</code>	Pops the left-most item from a <i><comma list></i> into the <i><token list variable></i> , <i>i.e.</i> removes the item from the comma list and stores it in the <i><token list variable></i> . The <i><comma list></i> is modified globally, while the assignment of the <i><token list variable></i> is local.

<code>\clist_pop:NNTF</code>	<code>\clist_pop:NNTF <comma list> <token list variable> {<true code>} {<false code>}</code>
<code>\clist_pop:cNTF</code>	If the <i><comma list></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><comma list></i> is non-empty, pops the top item from the <i><comma list></i> in the <i><token list variable></i> , <i>i.e.</i> removes the item from the <i><comma list></i> . Both the <i><comma list></i> and the <i><token list variable></i> are assigned locally.
New: 2012-05-14	

<code>\clist_gpop:NNTF</code>	<code>\clist_gpop:NNTF <comma list> <token list variable> {<true code>} {<false code>}</code>
<code>\clist_gpop:cNTF</code>	If the <i><comma list></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><comma list></i> is non-empty, pops the top item from the <i><comma list></i> in the <i><token list variable></i> , <i>i.e.</i> removes the item from the <i><comma list></i> . The <i><comma list></i> is modified globally, while the <i><token list variable></i> is assigned locally.
New: 2012-05-14	

<code>\clist_push:Nn</code>	<code>\clist_push:Nn <comma list> {<items>}</code>
<code>\clist_push:(NV No Nx cn cV co cx)</code>	
<code>\clist_gpush:Nn</code>	
<code>\clist_gpush:(NV No Nx cn cV co cx)</code>	
Adds the {<items>} to the top of the <i><comma list></i> . Spaces are removed from both sides of each item.	

8 Using a single item

<code>\clist_item:Nn</code> ★	<code>\clist_item:Nn <comma list> {<integer expression>}</code>
<code>\clist_item:cn</code> ★	
<code>\clist_item:nn</code> ★	Indexing items in the <i><comma list></i> from 1 at the top (left), this function will evaluate the <i><integer expression></i> and leave the appropriate item from the comma list in the input stream. If the <i><integer expression></i> is negative, indexing occurs from the bottom (right) of the comma list. When the <i><integer expression></i> is larger than the number of items in the <i><comma list></i> (as calculated by <code>\clist_count:N</code>) then the function will expand to nothing.
New: 2014-07-17	

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

9 Viewing comma lists

<code>\clist_show:N</code>	<code>\clist_show:N <comma list></code>
<code>\clist_show:c</code>	Displays the entries in the <i><comma list></i> in the terminal.
Updated: 2015-08-03	
<code>\clist_show:n</code>	<code>\clist_show:n {<tokens>}</code>
Updated: 2013-08-03	Displays the entries in the comma list in the terminal.

10 Constant and scratch comma lists

<code>\c_empty_clist</code>	Constant that is always empty.
-----------------------------	--------------------------------

New: 2012-07-02

<code>\l_tmpa_clist</code> <code>\l_tmpb_clist</code>	Scratch comma lists for local assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX 3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	--

New: 2011-09-06

<code>\g_tmpa_clist</code> <code>\g_tmpb_clist</code>	Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX 3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	---

New: 2011-09-06

Part XV

The l3prop package

Property lists

L^AT_EX3 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 will overwrite 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

<code>\prop_new:N</code>
<code>\prop_new:c</code>

`\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 $\langle property\ list \rangle$ will initially contain no entries.

<code>\prop_clear:N</code>
<code>\prop_clear:c</code>
<code>\prop_gclear:N</code>
<code>\prop_gclear:c</code>

`\prop_clear:N` $\langle property\ list \rangle$

Clears all entries from the $\langle property\ list \rangle$.

<code>\prop_clear_new:N</code>
<code>\prop_clear_new:c</code>
<code>\prop_gclear_new:N</code>
<code>\prop_gclear_new:c</code>

`\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.

<code>\prop_set_eq:NN</code>
<code>\prop_set_eq:(cN Nc cc)</code>
<code>\prop_gset_eq:NN</code>
<code>\prop_gset_eq:(cN Nc cc)</code>

`\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 $\langle property\ list_2 \rangle$.

2 Adding entries to property lists

$\backslash\text{prop_put:Nnn}$ $\backslash\text{prop_put:}(\text{NnV Nno Nnx NVn NVV Non Noo cnn cnV cno cnx cVn cVV con coo})$ $\backslash\text{prop_gput:Nnn}$ $\backslash\text{prop_gput:}(\text{NnV Nno Nnx NVn NVV Non Noo cnn cnV cno cnx cVn cVV con coo})$	$\backslash\text{prop_put:Nnn } \langle\text{property list}\rangle$ $\{\langle\text{key}\rangle\} \{\langle\text{value}\rangle\}$
--	---

Updated: 2012-07-09

Adds an entry to the $\langle\text{property list}\rangle$ which may be accessed using the $\langle\text{key}\rangle$ and which has $\langle\text{value}\rangle$. Both the $\langle\text{key}\rangle$ and $\langle\text{value}\rangle$ may contain any $\langle\text{balanced text}\rangle$. The $\langle\text{key}\rangle$ is stored after processing with $\backslash\text{tl_to_str:n}$, meaning that category codes are ignored. If the $\langle\text{key}\rangle$ is already present in the $\langle\text{property list}\rangle$, the existing entry is overwritten by the new $\langle\text{value}\rangle$.

$\backslash\text{prop_put_if_new:Nnn}$ $\backslash\text{prop_put_if_new:cnn}$ $\backslash\text{prop_gput_if_new:Nnn}$ $\backslash\text{prop_gput_if_new:cnn}$	$\backslash\text{prop_put_if_new:Nnn } \langle\text{property list}\rangle \{\langle\text{key}\rangle\} \{\langle\text{value}\rangle\}$
--	---

If the $\langle\text{key}\rangle$ is present in the $\langle\text{property list}\rangle$ then no action is taken. If the $\langle\text{key}\rangle$ is not present in the $\langle\text{property list}\rangle$ then a new entry is added. Both the $\langle\text{key}\rangle$ and $\langle\text{value}\rangle$ may contain any $\langle\text{balanced text}\rangle$. The $\langle\text{key}\rangle$ is stored after processing with $\backslash\text{tl_to_str:n}$, meaning that category codes are ignored.

3 Recovering values from property lists

$\backslash\text{prop_get:NnN}$ $\backslash\text{prop_get:}(\text{NVN NoN cnN cVN coN})$	$\backslash\text{prop_get:NnN } \langle\text{property list}\rangle \{\langle\text{key}\rangle\} \langle\text{tl var}\rangle$
---	---

Updated: 2011-08-28

Recovers the $\langle\text{value}\rangle$ stored with $\langle\text{key}\rangle$ from the $\langle\text{property list}\rangle$, and places this in the $\langle\text{token list variable}\rangle$. If the $\langle\text{key}\rangle$ is not found in the $\langle\text{property list}\rangle$ then the $\langle\text{token list variable}\rangle$ will contain the special marker $\backslash\text{q_no_value}$. The $\langle\text{token list variable}\rangle$ is set within the current $\text{T}_{\text{E}}\text{X}$ group. See also $\backslash\text{prop_get:NnNTF}$.

$\backslash\text{prop_pop:NnN}$ $\backslash\text{prop_pop:}(\text{NoN cnN coN})$	$\backslash\text{prop_pop:NnN } \langle\text{property list}\rangle \{\langle\text{key}\rangle\} \langle\text{tl var}\rangle$
---	---

Updated: 2011-08-18

Recovers the $\langle\text{value}\rangle$ stored with $\langle\text{key}\rangle$ from the $\langle\text{property list}\rangle$, and places this in the $\langle\text{token list variable}\rangle$. If the $\langle\text{key}\rangle$ is not found in the $\langle\text{property list}\rangle$ then the $\langle\text{token list variable}\rangle$ will contain the special marker $\backslash\text{q_no_value}$. The $\langle\text{key}\rangle$ and $\langle\text{value}\rangle$ are then deleted from the property list. Both assignments are local. See also $\backslash\text{prop_pop:NnNTF}$.

$\backslash\text{prop_gpop:NnN}$ $\backslash\text{prop_gpop:}(\text{NoN cnN coN})$	$\backslash\text{prop_gpop:NnN } \langle\text{property list}\rangle \{\langle\text{key}\rangle\} \langle\text{tl var}\rangle$
---	--

Updated: 2011-08-18

Recovers the $\langle\text{value}\rangle$ stored with $\langle\text{key}\rangle$ from the $\langle\text{property list}\rangle$, and places this in the $\langle\text{token list variable}\rangle$. If the $\langle\text{key}\rangle$ is not found in the $\langle\text{property list}\rangle$ then the $\langle\text{token list variable}\rangle$ will contain the special marker $\backslash\text{q_no_value}$. The $\langle\text{key}\rangle$ and $\langle\text{value}\rangle$ are then deleted from the property list. The $\langle\text{property list}\rangle$ is modified globally, while the assignment of the $\langle\text{token list variable}\rangle$ is local. See also $\backslash\text{prop_gpop:NnNTF}$.

<code>\prop_item:Nn</code> ★	<code>\prop_item:Nn</code> $\langle property list \rangle$ $\{\langle key \rangle\}$
---	--

<code>\prop_item:cn</code> ★

New: 2014-07-17

Expands to the $\langle value \rangle$ corresponding to the $\langle key \rangle$ in the $\langle property list \rangle$. If the $\langle key \rangle$ is missing, this has an empty expansion.

T_EXhackers 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 $\langle value \rangle$ will not expand further when appearing in an *x*-type argument expansion.

4 Modifying property lists

<code>\prop_remove:Nn</code>

<code>\prop_remove:(NV cn cV)</code>

<code>\prop_gremove:Nn</code>

<code>\prop_gremove:(NV cn cV)</code>

New: 2012-05-12

<code>\prop_remove:Nn</code> $\langle property list \rangle$ $\{\langle key \rangle\}$
--

Removes the entry listed under $\langle key \rangle$ from the $\langle property list \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ no change occurs, *i.e* there is no need to test for the existence of a key before deleting it.

5 Property list conditionals

<code>\prop_if_exist_p:N</code> ★
--

<code>\prop_if_exist_p:c</code> ★
--

<code>\prop_if_exist:N\underline{TF}</code> ★

<code>\prop_if_exist:c\underline{TF}</code> ★

New: 2012-03-03

<code>\prop_if_exist_p:N</code> $\langle property list \rangle$

<code>\prop_if_exist:N\underline{TF}</code> $\langle property list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests whether the $\langle property list \rangle$ is currently defined. This does not check that the $\langle property list \rangle$ really is a property list variable.

<code>\prop_if_empty_p:N</code> ★
--

<code>\prop_if_empty_p:c</code> ★
--

<code>\prop_if_empty:N\underline{TF}</code> ★

<code>\prop_if_empty:c\underline{TF}</code> ★

<code>\prop_if_empty_p:N</code> $\langle property list \rangle$

<code>\prop_if_empty:N\underline{TF}</code> $\langle property list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle property list \rangle$ is empty (containing no entries).

<code>\prop_if_in_p:Nn</code>	★	<code>\prop_if_in:Nn\underline{TF}</code> $\langle property list \rangle$ $\{\langle key \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
-------------------------------	----------------------------------	---

<code>\prop_if_in_p:(NV No cn cV co)</code>	★
---	----------------------------------

<code>\prop_if_in:Nn\underline{TF}</code>	★
--	----------------------------------

<code>\prop_if_in:(NV No cn cV co)\underline{TF}</code>	★
--	----------------------------------

Updated: 2011-09-15

Tests if the $\langle key \rangle$ is present in the $\langle property list \rangle$, making the comparison using the method described by `\str_if_eq:nn \underline{TF}` .

T_EXhackers note: This function iterates through every key–value pair in the $\langle property list \rangle$ and is therefore slower than using the non-expandable `\prop_get:Nn \underline{TF}` .

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.

<u>\prop_get:NnNTF</u>	<u>\prop_get:NnNTF</u> $\langle\textit{property list}\rangle$ $\{\langle\textit{key}\rangle\}$ $\langle\textit{token list variable}\rangle$
<u>\prop_get:(NVN NoN cnN cVN coN)TF</u>	$\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$
Updated: 2012-05-19	

If the $\langle\textit{key}\rangle$ is not present in the $\langle\textit{property list}\rangle$, leaves the $\langle\textit{false code}\rangle$ in the input stream. The value of the $\langle\textit{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\textit{key}\rangle$ is present in the $\langle\textit{property list}\rangle$, stores the corresponding $\langle\textit{value}\rangle$ in the $\langle\textit{token list variable}\rangle$ without removing it from the $\langle\textit{property list}\rangle$, then leaves the $\langle\textit{true code}\rangle$ in the input stream. The $\langle\textit{token list variable}\rangle$ is assigned locally.

<u>\prop_pop:NnNTF</u>	<u>\prop_pop:NnNTF</u> $\langle\textit{property list}\rangle$ $\{\langle\textit{key}\rangle\}$ $\langle\textit{token list variable}\rangle$ $\{\langle\textit{true code}\rangle\}$
<u>\prop_pop:cnNTF</u>	$\{\langle\textit{false code}\rangle\}$
New: 2011-08-18	If the $\langle\textit{key}\rangle$ is not present in the $\langle\textit{property list}\rangle$, leaves the $\langle\textit{false code}\rangle$ in the input stream. The value of the $\langle\textit{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\textit{key}\rangle$ is present in the $\langle\textit{property list}\rangle$, pops the corresponding $\langle\textit{value}\rangle$ in the $\langle\textit{token list variable}\rangle$, <i>i.e.</i> removes the item from the $\langle\textit{property list}\rangle$. Both the $\langle\textit{property list}\rangle$ and the $\langle\textit{token list variable}\rangle$ are assigned locally.
Updated: 2012-05-19	

<u>\prop_gpop:NnNTF</u>	<u>\prop_gpop:NnNTF</u> $\langle\textit{property list}\rangle$ $\{\langle\textit{key}\rangle\}$ $\langle\textit{token list variable}\rangle$ $\{\langle\textit{true code}\rangle\}$
<u>\prop_gpop:cnNTF</u>	$\{\langle\textit{false code}\rangle\}$
New: 2011-08-18	If the $\langle\textit{key}\rangle$ is not present in the $\langle\textit{property list}\rangle$, leaves the $\langle\textit{false code}\rangle$ in the input stream. The value of the $\langle\textit{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\textit{key}\rangle$ is present in the $\langle\textit{property list}\rangle$, pops the corresponding $\langle\textit{value}\rangle$ in the $\langle\textit{token list variable}\rangle$, <i>i.e.</i> removes the item from the $\langle\textit{property list}\rangle$. The $\langle\textit{property list}\rangle$ is modified globally, while the $\langle\textit{token list variable}\rangle$ is assigned locally.
Updated: 2012-05-19	

7 Mapping to property lists

<u>\prop_map_function:NN</u> ☆	<u>\prop_map_function:NN</u> $\langle\textit{property list}\rangle$ $\langle\textit{function}\rangle$
<u>\prop_map_function:cN</u> ☆	Applies $\langle\textit{function}\rangle$ to every $\langle\textit{entry}\rangle$ stored in the $\langle\textit{property list}\rangle$. The $\langle\textit{function}\rangle$ will receive two argument for each iteration: the $\langle\textit{key}\rangle$ and associated $\langle\textit{value}\rangle$. The order in which $\langle\textit{entries}\rangle$ are returned is not defined and should not be relied upon.
Updated: 2013-01-28	

<u>\prop_map_inline:Nn</u>	<u>\prop_map_inline:Nn</u> $\langle\textit{property list}\rangle$ $\{\langle\textit{inline function}\rangle\}$
<u>\prop_map_inline:cn</u>	Applies $\langle\textit{inline function}\rangle$ to every $\langle\textit{entry}\rangle$ stored within the $\langle\textit{property list}\rangle$. The $\langle\textit{inline function}\rangle$ should consist of code which will receive the $\langle\textit{key}\rangle$ as #1 and the $\langle\textit{value}\rangle$ as #2. The order in which $\langle\textit{entries}\rangle$ are returned is not defined and should not be relied upon.
Updated: 2013-01-08	

\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 will normally take 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 will lead to low level T_EX errors.

\prop_map_break:n ☆

Updated: 2012-06-29

\prop_map_break:n {*⟨tokens⟩*}

Used to terminate a `\prop_map...` function before all entries in the *⟨property list⟩* have been processed, inserting the *⟨tokens⟩* after the mapping has ended. This will normally take place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break:n { <tokens> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map...` scenario will lead to low level T_EX errors.

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.

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.

11 Internal property list functions

`\s__prop`

The internal token used at the beginning of property lists. This is also used after each $\langle key \rangle$ (see `__prop_pair:wn`).

`__prop_pair:wn`

`__prop_pair:wn $\langle key \rangle$ \s__prop $\{ \langle item \rangle \}$`

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 will be raised. The definition should always be set globally.

`\l__prop_internal_tl`

Token list used to store new key–value pairs to be inserted by functions of the `\prop_put:Nnn` family.

`__prop_split:NnTF`

Updated: 2013-01-08

`__prop_split:NnTF $\langle property list \rangle$ $\{ \langle key \rangle \}$ $\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$`

Splits the $\langle property list \rangle$ at the $\langle key \rangle$, giving three token lists: the $\langle extract \rangle$ of $\langle property list \rangle$ before the $\langle key \rangle$, the $\langle value \rangle$ associated with the $\langle key \rangle$ and the $\langle extract \rangle$ of the $\langle property list \rangle$ after the $\langle value \rangle$. Both $\langle extracts \rangle$ retain the internal structure of a property list, and the concatenation of the two $\langle extracts \rangle$ is a property list. If the $\langle key \rangle$ is present in the $\langle property list \rangle$ then the $\langle true code \rangle$ is left in the input stream, with **#1**, **#2**, and **#3** replaced by the first $\langle extract \rangle$, the $\langle value \rangle$, and the second $\langle extract \rangle$. If the $\langle key \rangle$ is not present in the $\langle property list \rangle$ then the $\langle false code \rangle$ is left in the input stream, with no trailing material. Both $\langle true code \rangle$ and $\langle false code \rangle$ 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 $\langle true code \rangle$ for the three extracts from the property list. The $\langle key \rangle$ comparison takes place as described for `\str_if_eq:nn`.

Part XVI

The l3box 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

<code>\box_new:N</code>	<code>\box_new:N</code> $\langle box \rangle$
<code>\box_new:c</code>	Creates a new $\langle box \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle box \rangle$ will initially be void.

<code>\box_clear:N</code>	<code>\box_clear:N</code> $\langle box \rangle$
<code>\box_clear:c</code>	Clears the content of the $\langle box \rangle$ by setting the box equal to <code>\c_void_box</code> .
<code>\box_gclear:N</code>	
<code>\box_gclear:c</code>	

<code>\box_clear_new:N</code>	<code>\box_clear_new:N</code> $\langle box \rangle$
<code>\box_clear_new:c</code>	Ensures that the $\langle box \rangle$ exists globally by applying <code>\box_new:N</code> if necessary, then applies <code>\box_(g)clear:N</code> to leave the $\langle box \rangle$ empty.
<code>\box_gclear_new:N</code>	
<code>\box_gclear_new:c</code>	

<code>\box_set_eq:NN</code>	<code>\box_set_eq:NN</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$
<code>\box_set_eq:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$.
<code>\box_gset_eq:NN</code>	
<code>\box_gset_eq:(cN Nc cc)</code>	

<code>\box_set_eq_clear:NN</code>	<code>\box_set_eq_clear:NN</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$
<code>\box_set_eq_clear:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ within the current TeX group equal to that of $\langle box_2 \rangle$, then clears $\langle box_2 \rangle$ globally.

<code>\box_gset_eq_clear:NN</code>	<code>\box_gset_eq_clear:NN</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$
<code>\box_gset_eq_clear:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$, then clears $\langle box_2 \rangle$. These assignments are global.

<code>\box_if_exist_p:N</code> ★	<code>\box_if_exist_p:N</code> $\langle box \rangle$
<code>\box_if_exist_p:c</code> ★	<code>\box_if_exist:NTF</code> $\langle box \rangle$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$
<code>\box_if_exist:NTF</code> ★	Tests whether the $\langle box \rangle$ is currently defined. This does not check that the $\langle box \rangle$ really is a box.
<code>\box_if_exist:cTF</code> ★	

New: 2012-03-03

2 Using boxes

`\box_use:N`
`\box_use:c`

`\box_use:N` $\langle box \rangle$

Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting.

TeXhackers note: This is the TeX primitive `\copy`.

`\box_use_clear:N`
`\box_use_clear:c`

`\box_use_clear:N` $\langle box \rangle$

Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting, then globally clears the content of the $\langle box \rangle$.

TeXhackers note: This is the TeX primitive `\box`.

`\box_move_right:nn`
`\box_move_left:nn`

`\box_move_right:nn` $\{\langle dimexpr \rangle\} \{\langle box function \rangle\}$

This function operates in vertical mode, and inserts the material specified by the $\langle box function \rangle$ such that its reference point is displaced horizontally by the given $\langle dimexpr \rangle$ from the reference point for typesetting, to the right or left as appropriate. The $\langle box function \rangle$ 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` $\{\langle dimexpr \rangle\} \{\langle box function \rangle\}$

This function operates in horizontal mode, and inserts the material specified by the $\langle box function \rangle$ such that its reference point is displaced vertical by the given $\langle dimexpr \rangle$ from the reference point for typesetting, up or down as appropriate. The $\langle box function \rangle$ 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` $\langle box \rangle$

Calculates the depth (below the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

TeXhackers note: This is the TeX primitive `\dp`.

`\box_ht:N`
`\box_ht:c`

`\box_ht:N` $\langle box \rangle$

Calculates the height (above the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

TeXhackers note: This is the TeX primitive `\ht`.

<code>\box_wd:N</code>	<code>\box_wd:N <box></code>
<code>\box_wd:c</code>	Calculates the width of the <code><box></code> in a form suitable for use in a <code><dimension expression></code> .

TeXhackers note: This is the TeX primitive `\wd`.

<code>\box_set_dp:Nn</code>	<code>\box_set_dp:Nn <box> {<dimension expression>}</code>
<code>\box_set_dp:cn</code>	Set the depth (below the baseline) of the <code><box></code> to the value of the <code>{<dimension expression>}</code> . This is a global assignment.

<code>\box_set_ht:Nn</code>	<code>\box_set_ht:Nn <box> {<dimension expression>}</code>
<code>\box_set_ht:cn</code>	Set the height (above the baseline) of the <code><box></code> to the value of the <code>{<dimension expression>}</code> . This is a global assignment.

<code>\box_set_wd:Nn</code>	<code>\box_set_wd:Nn <box> {<dimension expression>}</code>
<code>\box_set_wd:cn</code>	Set the width of the <code><box></code> to the value of the <code>{<dimension expression>}</code> . This is a global assignment.

4 Box conditionals

<code>\box_if_empty_p:N</code> *	<code>\box_if_empty_p:N <box></code>
<code>\box_if_empty_p:c</code> *	<code>\box_if_empty:NTF <box> {<true code>} {<false code>}</code>
<code>\box_if_empty:NTF</code> *	Tests if <code><box></code> is a empty (equal to <code>\c_empty_box</code>).
<code>\box_if_empty:cTF</code> *	

<code>\box_if_horizontal_p:N</code> *	<code>\box_if_horizontal_p:N <box></code>
<code>\box_if_horizontal_p:c</code> *	<code>\box_if_horizontal:NTF <box> {<true code>} {<false code>}</code>
<code>\box_if_horizontal:NTF</code> *	Tests if <code><box></code> is a horizontal box.
<code>\box_if_horizontal:cTF</code> *	

<code>\box_if_vertical_p:N</code> *	<code>\box_if_vertical_p:N <box></code>
<code>\box_if_vertical_p:c</code> *	<code>\box_if_vertical:NTF <box> {<true code>} {<false code>}</code>
<code>\box_if_vertical:NTF</code> *	Tests if <code><box></code> is a vertical box.
<code>\box_if_vertical:cTF</code> *	

5 The last box inserted

<code>\box_set_to_last:N</code>	<code>\box_set_to_last:N <box></code>
<code>\box_set_to_last:c</code>	Sets the <code><box></code> 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 <code><box></code> will always be void as it is not possible to recover the last added item.
<code>\box_gset_to_last:N</code>	
<code>\box_gset_to_last:c</code>	

6 Constant boxes

`\c_empty_box`

Updated: 2012-11-04

This is a permanently empty box, which is neither set as horizontal nor vertical.

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` $\langle box \rangle$

Shows full details of the content of the $\langle box \rangle$ in the terminal.

`\box_show:Nnn`

`\box_show:cnn`

New: 2012-05-11

`\box_show:Nnn` $\langle box \rangle$ $\langle intexpr_1 \rangle$ $\langle intexpr_2 \rangle$

Display the contents of $\langle box \rangle$ in the terminal, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.

`\box_log:N`

`\box_log:c`

New: 2012-05-11

`\box_log:N` $\langle box \rangle$

Writes full details of the content of the $\langle box \rangle$ to the log.

`\box_log:Nnn`

`\box_log:cnn`

New: 2012-05-11

`\box_log:Nnn` $\langle box \rangle$ $\langle intexpr_1 \rangle$ $\langle intexpr_2 \rangle$

Writes the contents of $\langle box \rangle$ to the log, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.

9 Horizontal mode boxes

`\hbox:n`

`\hbox:n` $\{ \langle contents \rangle \}$

Typesets the $\langle contents \rangle$ into a horizontal box of natural width and then includes this box in the current list for typesetting.

<hr/> <code>\hbox_to_wd:nn</code> <hr/>	<code>\hbox_to_wd:nn {<dimexpr>} {<contents>}</code> Typesets the $\langle contents \rangle$ into a horizontal box of width $\langle dimexpr \rangle$ and then includes this box in the current list for typesetting.
<hr/> <code>\hbox_to_zero:n</code> <hr/>	<code>\hbox_to_zero:n {<contents>}</code> Typesets the $\langle contents \rangle$ into a horizontal box of zero width and then includes this box in the current list for typesetting.
<hr/> <code>\hbox_set:Nn</code> <code>\hbox_set:cn</code> <code>\hbox_gset:Nn</code> <code>\hbox_gset:cn</code> <hr/>	<code>\hbox_set:Nn <box> {<contents>}</code> Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$.
<hr/> <code>\hbox_set_to_wd:Nnn</code> <code>\hbox_set_to_wd:cnn</code> <code>\hbox_gset_to_wd:Nnn</code> <code>\hbox_gset_to_wd:cnn</code> <hr/>	<code>\hbox_set_to_wd:Nnn <box> {<dimexpr>} {<contents>}</code> Typesets the $\langle contents \rangle$ to the width given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$.
<hr/> <code>\hbox_overlap_right:n</code> <hr/>	<code>\hbox_overlap_right:n {<contents>}</code> Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material will protrude to the right of the insertion point.
<hr/> <code>\hbox_overlap_left:n</code> <hr/>	<code>\hbox_overlap_left:n {<contents>}</code> Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material will protrude to the left of the insertion point.
<hr/> <code>\hbox_set:Nw</code> <code>\hbox_set:cw</code> <code>\hbox_set_end:</code> <code>\hbox_gset:Nw</code> <code>\hbox_gset:cw</code> <code>\hbox_gset_end:</code> <hr/>	<code>\hbox_set:Nw <box> <contents> \hbox_set_end:</code> Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$. In contrast to <code>\hbox_set:Nn</code> this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.
<hr/> <code>\hbox_unpack:N</code> <code>\hbox_unpack:c</code> <hr/>	<code>\hbox_unpack:N <box></code> Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.
<hr/> <code>\hbox_unpack_clear:N</code> <code>\hbox_unpack_clear:c</code> <hr/>	<code>\hbox_unpack_clear:N <box></code> Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The $\langle box \rangle$ is then cleared globally.

T_EXhackers note: This is the T_EX primitive `\unhcopy`.

T_EXhackers note: This is the T_EX primitive `\unhbox`.

10 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 will have 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 will typically be non-zero.

`\vbox:n`

`\vbox:n {<contents>}`

Updated: 2011-12-18

Typesets the `<contents>` into a vertical box of natural height and includes this box in the current list for typesetting.

T_EXhackers note: This is the T_EX primitive `\vbox`.

`\vbox_top:n`

`\vbox_top:n {<contents>}`

Updated: 2011-12-18

Typesets the `<contents>` into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box will be equal to that of the *first* item added to the box.

T_EXhackers note: This is the T_EX primitive `\vtop`.

`\vbox_to_ht:nn`

`\vbox_to_ht:nn {<dimexpr>} {<contents>}`

Updated: 2011-12-18

Typesets the `<contents>` into a vertical box of height `<dimexpr>` and then includes this box in the current list for typesetting.

`\vbox_to_zero:n`

`\vbox_to_zero:n {<contents>}`

Updated: 2011-12-18

Typesets the `<contents>` 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`

`\vbox_set:Nn <box> {<contents>}`

Typesets the `<contents>` at natural height and then stores the result inside the `<box>`.

Updated: 2011-12-18

`\vbox_set_top:Nn`
`\vbox_set_top:cn`
`\vbox_gset_top:Nn`
`\vbox_gset_top:cn`

`\vbox_set_top:Nn <box> {<contents>}`

Typesets the `<contents>` at natural height and then stores the result inside the `<box>`. The baseline of the box will be equal to that of the *first* item added to the box.

Updated: 2011-12-18

`\vbox_set_to_ht:Nnn`
`\vbox_set_to_ht:cnn`
`\vbox_gset_to_ht:Nnn`
`\vbox_gset_to_ht:cnn`

`\vbox_set_to_ht:Nnn <box> {<dimexpr>} {<contents>}`

Typesets the `<contents>` to the height given by the `<dimexpr>` and then stores the result inside the `<box>`.

Updated: 2011-12-18

```
\vbox_set:Nw
\vbox_set:cw
\vbox_set_end:
\vbox_gset:Nw
\vbox_gset:cw
\vbox_gset_end:
```

Updated: 2011-12-18

```
\vbox_set:Nw <box> <contents> \vbox_set_end:
```

Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. In contrast to $\backslash\text{vbox_set:Nn}$ this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.

```
\vbox_set_split_to_ht:NNn
```

Updated: 2011-10-22

```
\vbox_set_split_to_ht:NNn <box1> <box2> {<dimexpr>}
```

Sets $\langle box_1 \rangle$ to contain material to the height given by the $\langle dimexpr \rangle$ by removing content from the top of $\langle box_2 \rangle$ (which must be a vertical box).

T_EXhackers note: This is the T_EX primitive $\backslash\text{vsplit}$.

```
\vbox_unpack:N
\vbox_unpack:c
```

```
\vbox_unpack:N <box>
```

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

T_EXhackers note: This is the T_EX primitive $\backslash\text{unvcopy}$.

```
\vbox_unpack_clear:N
\vbox_unpack_clear:c
```

```
\vbox_unpack:N <box>
```

Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The $\langle box \rangle$ is then cleared globally.

T_EXhackers note: This is the T_EX primitive $\backslash\text{unvbox}$.

11 Primitive box conditionals

```
\if_hbox:N ★
```

```
\if_hbox:N <box>
  <true code>
\else:
  <false code>
\fi:
```

Tests if $\langle box \rangle$ is a horizontal box.

T_EXhackers note: This is the T_EX primitive $\backslash\text{ifhbox}$.

```
\if_vbox:N ★
```

```
\if_vbox:N <box>
  <true code>
\else:
  <false code>
\fi:
```

Tests if $\langle box \rangle$ is a vertical box.

T_EXhackers note: This is the T_EX primitive $\backslash\text{ifvbox}$.

`\if_box_empty:N` ★ `\if_box_empty:N` $\langle box \rangle$
 $\langle true\ code \rangle$
`\else:`
 $\langle false\ code \rangle$
`\fi:`
Tests if $\langle box \rangle$ is an empty (void) box.

TeXhackers note: This is the TeX primitive `\ifvoid`.

Part XVII

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

<code>\coffin_new:N</code>
<code>\coffin_new:c</code>
New: 2011-08-17

`\coffin_new:N` $\langle coffin \rangle$

Creates a new $\langle coffin \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle coffin \rangle$ will initially be empty.

<code>\coffin_clear:N</code>
<code>\coffin_clear:c</code>
New: 2011-08-17

`\coffin_clear:N` $\langle coffin \rangle$

Clears the content of the $\langle coffin \rangle$ within the current T_EX group level.

<code>\coffin_set_eq:NN</code>
<code>\coffin_set_eq:(Nc cN cc)</code>
New: 2011-08-17

`\coffin_set_eq:NN` $\langle coffin_1 \rangle$ $\langle coffin_2 \rangle$

Sets both the content and poles of $\langle coffin_1 \rangle$ equal to those of $\langle coffin_2 \rangle$ within the current T_EX group level.

<code>\coffin_if_exist_p:N</code> ★
<code>\coffin_if_exist_p:c</code> ★
<code>\coffin_if_exist:NTF</code> ★
<code>\coffin_if_exist:cTF</code> ★
New: 2012-06-20

`\coffin_if_exist_p:N` $\langle box \rangle$

`\coffin_if_exist:NTF` $\langle box \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests whether the $\langle coffin \rangle$ is currently defined.

2 Setting coffin content and poles

All coffin functions create and manipulate coffins locally within the current T_EX group level.

<code>\hcoffin_set:Nn</code>
<code>\hcoffin_set:cn</code>
New: 2011-08-17
Updated: 2011-09-03

`\hcoffin_set:Nn` $\langle coffin \rangle$ $\{\langle material \rangle\}$

Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.

<code>\hcoffin_set:Nw</code>
<code>\hcoffin_set:cw</code>
<code>\hcoffin_set_end:</code>
New: 2011-09-10

`\hcoffin_set:Nw` $\langle coffin \rangle$ $\langle material \rangle$ `\hcoffin_set_end:`

Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ 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.

<hr/> <code>\vcoffin_set:Nnn</code> <hr/> <code>\vcoffin_set:cnn</code> <hr/> <div>New: 2011-08-17 Updated: 2012-05-22</div> <hr/>	<code>\vcoffin_set:Nnn <coffin> {{width}} {{material}}</code> <p>Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.</p>
<hr/> <code>\vcoffin_set:Nnw</code> <hr/> <code>\vcoffin_set:cnw</code> <hr/> <code>\vcoffin_set_end:</code> <hr/> <div>New: 2011-09-10 Updated: 2012-05-22</div> <hr/>	<code>\vcoffin_set:Nnw <coffin> {{width}} <material> \vcoffin_set_end:</code> <p>Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ 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.</p>
<hr/> <code>\coffin_set_horizontal_pole:Nnn</code> <hr/> <code>\coffin_set_horizontal_pole:cnn</code> <hr/> <div>New: 2012-07-20</div> <hr/>	<code>\coffin_set_horizontal_pole:Nnn <coffin> {{pole}} {{offset}}</code> <p>Sets the $\langle pole \rangle$ to run horizontally through the $\langle coffin \rangle$. The $\langle pole \rangle$ will be located at the $\langle offset \rangle$ from the bottom edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.</p>
<hr/> <code>\coffin_set_vertical_pole:Nnn</code> <hr/> <code>\coffin_set_vertical_pole:cnn</code> <hr/> <div>New: 2012-07-20</div> <hr/>	<code>\coffin_set_vertical_pole:Nnn <coffin> {{pole}} {{offset}}</code> <p>Sets the $\langle pole \rangle$ to run vertically through the $\langle coffin \rangle$. The $\langle pole \rangle$ will be located at the $\langle offset \rangle$ from the left-hand edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.</p>

3 Joining and using coffins

<code>\coffin_attach:NnnNnnnn</code>	<code>\coffin_attach:NnnNnnnn</code>
<code>\coffin_attach:(cnnNnnnn Nnnncnnn cnnncnnn)</code>	$\langle coffin_1 \rangle \{ \langle coffin_1 - pole_1 \rangle \} \{ \langle coffin_1 - pole_2 \rangle \}$ $\langle coffin_2 \rangle \{ \langle coffin_2 - pole_1 \rangle \} \{ \langle coffin_2 - pole_2 \rangle \}$ $\{ \langle x - offset \rangle \} \{ \langle y - offset \rangle \}$

This function attaches $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ is not altered, *i.e.* $\langle coffin_2 \rangle$ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1 - pole_1 \rangle$ and $\langle coffin_1 - pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2 - pole_1 \rangle$ and $\langle coffin_2 - pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x - offset \rangle$ and $\langle y - offset \rangle$. The two offsets should be given as dimension expressions.

```
\coffin_join:NnnNnnnn
\coffin_join:(cnnNnnnn|Nnncnnnn|cnncnnnn)
```

```
\coffin_join:NnnNnnnn
  <coffin_1> {<coffin_1-pole_1>} {<coffin_1-pole_2>}
  <coffin_2> {<coffin_2-pole_1>} {<coffin_2-pole_2>}
  {<x-offset>} {<y-offset>}
```

This function joins $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ may expand. The new bounding box will cover the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

```
\coffin_typeset:Nnnnn
\coffin_typeset:cnnnn
```

Updated: 2012-07-20

```
\coffin_typeset:Nnnnn <coffin> {<pole_1>} {<pole_2>}
  {<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.

4 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 $\langle dimension expression \rangle$.

5 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 $\langle handles \rangle$ will be 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` $\langle coffin \rangle$ $\{\langle pole_1 \rangle\}$ $\{\langle pole_2 \rangle\}$ $\{\langle color \rangle\}$

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 $\langle handle \rangle$ will be 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` $\langle coffin \rangle$

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.

5.1 Constants and variables

`\c_empty_coffin`

A permanently empty coffin.

`\l_tmpa_coffin`
`\l_tmppb_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 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 XVIII

The **l3color** 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.

```
\color_ensure_current:
```

New: 2011-09-03

```
\color_ensure_current:
```

Ensures that material inside a box will use 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 XIX

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 will automatically be wrapped to the length available in the console. As a result, formatting is only needed where it will help 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 `LaTeX` while still being filterable at a more granular level. Thus for example

```
\msg_new:nnnn { mymodule } { submodule / message } ...
```

will allow only those messages from the `submodule` to be filtered out.

```
\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 will be 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 will be 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 will be 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.

<code>\msg_if_exist_p:nn</code> ★	<code>\msg_if_exist_p:nn {<module>} {<message>}</code>
<code>\msg_if_exist:nnTF</code> ★	<code>\msg_if_exist:nnTF {<module>} {<message>} {<true code>} {<false code>}</code>
New: 2012-03-03	Tests whether the <i><message></i> for the <i><module></i> is currently defined.

2 Contextual information for messages

<code>\msg_line_context:</code> ☆	<code>\msg_line_context:</code>
	Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is preceded by the text <code>on line</code> .

<code>\msg_line_number:</code> ★	<code>\msg_line_number:</code>
	Prints the current line number when a message is given.

<code>\msg_fatal_text:n</code> ★	<code>\msg_fatal_text:n {<module>}</code>
	Produces the standard text
	<code>Fatal <module> error</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.

<code>\msg_critical_text:n</code> ★	<code>\msg_critical_text:n {<module>}</code>
	Produces the standard text
	<code>Critical <module> error</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.

<code>\msg_error_text:n</code> ★	<code>\msg_error_text:n {<module>}</code>
	Produces the standard text
	<code><module> error</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.

<code>\msg_warning_text:n</code> ★	<code>\msg_warning_text:n {<module>}</code>
	Produces the standard text
	<code><module> warning</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.

`\msg_info_text:n` ★ `\msg_info_text:n {<module>}`

Produces the standard text:

`<module> info`

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_see_documentation_text:n` ★ `\msg_see_documentation_text:n {<module>}`

Produces the standard text

`See the <module> documentation for further information.`

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.

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 will be ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments will be 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:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}`
`\msg_fatal:nnxxxx`
`\msg_fatal:nnnnnn` Issues `<module>` error `<message>`, passing `<arg one>` to `<arg four>` to the text-creating
`\msg_fatal:nnxxx` functions. After issuing a fatal error the T_EX run will halt.
`\msg_fatal:nnnn`
`\msg_fatal:nnxx`
`\msg_fatal:nnn`
`\msg_fatal:nnx`
`\msg_fatal:nn`

Updated: 2012-08-11

`\msg_critical:nnnnnn` `\msg_critical:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}`
`\msg_critical:nnxxxx`

`\msg_critical:nnnnnn` Issues `<module>` error `<message>`, passing `<arg one>` to `<arg four>` to the text-creating
`\msg_critical:nnxxx` functions. After issuing a critical error, T_EX will stop reading the current input file. This
`\msg_critical:nnnn` may halt the T_EX run (if the current file is the main file) or may abort reading a sub-file.
`\msg_critical:nnxx`

`\msg_critical:nnn` **T_EXhackers note:** The T_EX `\endinput` primitive is used to exit the file. In particular,
`\msg_critical:nnx` the rest of the current line remains in the input stream.
`\msg_critical:nn`

Updated: 2012-08-11

<code>\msg_error:nnnnnn</code> <code>\msg_error:nnxxxx</code> <code>\msg_error:nnnnn</code> <code>\msg_error:nnxxx</code> <code>\msg_error:nnnn</code> <code>\msg_error:nnxx</code> <code>\msg_error:nnn</code> <code>\msg_error:nnx</code> <code>\msg_error:nn</code>	<code>\msg_error:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code> <p>Issues <i><module></i> error <i><message></i>, passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The error will interrupt processing and issue the text at the terminal. After user input, the run will continue.</p>
--	--

Updated: 2012-08-11

<code>\msg_warning:nnnnnn</code> <code>\msg_warning:nnxxxx</code> <code>\msg_warning:nnnnn</code> <code>\msg_warning:nnxxx</code> <code>\msg_warning:nnnn</code> <code>\msg_warning:nnxx</code> <code>\msg_warning:nnn</code> <code>\msg_warning:nnx</code> <code>\msg_warning:nn</code>	<code>\msg_warning:nnxxxx {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code> <p>Issues <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 will be added to the log file and the terminal, but the T_EX run will not be interrupted.</p>
--	--

Updated: 2012-08-11

<code>\msg_info:nnnnnn</code> <code>\msg_info:nnxxxx</code> <code>\msg_info:nnnnn</code> <code>\msg_info:nnxxx</code> <code>\msg_info:nnnn</code> <code>\msg_info:nnxx</code> <code>\msg_info:nnn</code> <code>\msg_info:nnx</code> <code>\msg_info:nn</code>	<code>\msg_info:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code> <p>Issues <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 will be added to the log file.</p>
---	--

Updated: 2012-08-11

<code>\msg_log:nnnnnn</code> <code>\msg_log:nnxxxx</code> <code>\msg_log:nnnnn</code> <code>\msg_log:nnxxx</code> <code>\msg_log:nnnn</code> <code>\msg_log:nnxx</code> <code>\msg_log:nnn</code> <code>\msg_log:nnx</code> <code>\msg_log:nn</code>	<code>\msg_log:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code> <p>Issues <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 will be added to the log file: the output is briefer than <code>\msg_info:nnnnnn</code>.</p>
--	---

Updated: 2012-08-11

<code>\msg_none:nnnnnn</code>	<code>\msg_none:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_none:nnxxxx</code>	
<code>\msg_none:nnnnn</code>	Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).
<code>\msg_none:nnxxx</code>	
<code>\msg_none:nnnn</code>	
<code>\msg_none:nnxx</code>	
<code>\msg_none:nnn</code>	
<code>\msg_none:nnx</code>	
<code>\msg_none:nn</code>	

Updated: 2012-08-11

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 will raise 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 will raise errors 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.

<code>\msg_redirect_class:nn</code>	<code>\msg_redirect_class:nn {<class one>} {<class two>}</code>
-------------------------------------	---

Updated: 2012-04-27

Changes the behaviour of messages of *<class one>* so that they are processed using the code for those of *<class two>*.

<code>\msg_log:n</code>	<code>\msg_log:n {<text>}</code>
-------------------------	--

New: 2012-06-28	Writes to the log file with the <i><text></i> laid out in the format
-----------------	--

```

.....
. <text>
.....

```

where the *<text>* will be wrapped to fit within the current line length. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

<code>\msg_term:n</code>	<code>\msg_term:n {<text>}</code>
--------------------------	---

New: 2012-06-28	Writes to the terminal and log file with the <i><text></i> laid out in the format
-----------------	---

```

*****
* <text>
*****

```

where the *<text>* will be wrapped to fit within the current line length. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

6 Kernel-specific functions

Messages from L^AT_EX3 itself are handled by the general message system, but have their own functions. This allows some text to be pre-defined, and also ensures that serious errors can be handled properly.

<code>_msg_kernel_new:nnnn</code>	<code>_msg_kernel_new:nnnn {<module>} {<message>} {<text>} {<more text>}</code>
------------------------------------	--

<code>_msg_kernel_new:nnn</code>	
-----------------------------------	--

Updated: 2011-08-16	
---------------------	--

Creates a kernel *<message>* for a given *<module>*. The message will be 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 will be raised if the *<message>* already exists.

<code>_msg_kernel_set:nnnn</code>	<code>_msg_kernel_set:nnnn {<module>} {<message>} {<text>} {<more text>}</code>
------------------------------------	--

<code>_msg_kernel_set:nnn</code>	
-----------------------------------	--

Sets up the text for a kernel *<message>* for a given *<module>*. The message will be 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.

```

\_msg_kernel_fatal:nnnnnn
\_msg_kernel_fatal:nnxxxx
\_msg_kernel_fatal:nnnnn
\_msg_kernel_fatal:nnxxx
\_msg_kernel_fatal:nnnn
\_msg_kernel_fatal:nnxx
\_msg_kernel_fatal:nnn
\_msg_kernel_fatal:nnx
\_msg_kernel_fatal:nn

```

Updated: 2012-08-11

```

\_msg_kernel_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 T_EX run will halt. Cannot be redirected.

```

\_msg_kernel_error:nnnnnn
\_msg_kernel_error:nnxxxx
\_msg_kernel_error:nnnnn
\_msg_kernel_error:nnxxx
\_msg_kernel_error:nnnn
\_msg_kernel_error:nnxx
\_msg_kernel_error:nnn
\_msg_kernel_error:nnx
\_msg_kernel_error:nn

```

Updated: 2012-08-11

```

\_msg_kernel_error: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. The error will stop processing and issue the text at the terminal. After user input, the run will continue. Cannot be redirected.

```

\_msg_kernel_warning:nnnnnn
\_msg_kernel_warning:nnxxxx
\_msg_kernel_warning:nnnnn
\_msg_kernel_warning:nnxxx
\_msg_kernel_warning:nnnn
\_msg_kernel_warning:nnxx
\_msg_kernel_warning:nnn
\_msg_kernel_warning:nnx
\_msg_kernel_warning:nn

```

Updated: 2012-08-11

```

\_msg_kernel_warning:nnnnnn {\module} {\message} {\arg one} {\arg
two}} {\arg three}} {\arg four}}

```

Issues kernel *<module>* warning *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The warning text will be added to the log file, but the T_EX run will not be interrupted.

```

\_msg_kernel_info:nnnnnn
\_msg_kernel_info:nnxxxx
\_msg_kernel_info:nnnnn
\_msg_kernel_info:nnxxx
\_msg_kernel_info:nnnn
\_msg_kernel_info:nnxx
\_msg_kernel_info:nnn
\_msg_kernel_info:nnx
\_msg_kernel_info:nn

```

Updated: 2012-08-11

```

\_msg_kernel_info:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg
three}} {\arg four}}

```

Issues kernel *<module>* information *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The information text will be added to the log file.

7 Expandable errors

In a few places, the L^AT_EX3 kernel needs to produce errors in an expansion only context. This 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. However, the interface is similar, with the important caveat that the message text and arguments are not expanded, and messages should be very short.

```

\__msg_kernel_expandable_error:nnnnnn ★ \__msg_kernel_expandable_error:nnnnnn {<module>} {<message>}
\__msg_kernel_expandable_error:nnnnnn ★ {<arg one>} {<arg two>} {<arg three>} {<arg four>}
\__msg_kernel_expandable_error:nnnn ★
\__msg_kernel_expandable_error:nnnn ★
\__msg_kernel_expandable_error:nnnn ★
\__msg_kernel_expandable_error:nnnn ★

```

New: 2011-11-23

Issues an error, passing *<arg one>* to *<arg four>* to the text-creating functions. The resulting string must be shorter than a line, otherwise it will be cropped.

```

\__msg_expandable_error:n ★ \__msg_expandable_error:n {<error message>}

```

New: 2011-08-11

Updated: 2011-08-13

Issues an “Undefined error” message from T_EX itself, and prints the *<error message>*. The *<error message>* must be short: it is cropped at the end of one line.

T_EXhackers note: This function expands to an empty token list after two steps. Tokens inserted in response to T_EX’s prompt are read with the current category code setting, and inserted just after the place where the error message was issued.

8 Internal l3msg functions

The following functions are used in several kernel modules.

```

\__msg_log_next: \__msg_log_next: <show-command>

```

New: 2015-08-05

Causes the next *<show-command>* to send its output to the log file instead of the terminal. This allows for instance `\cs_log:N` to be defined as `__msg_log_next:\cs_show:N`. The effect of this command lasts until the next use of `__msg_show_wrap:Nn` or `__msg_show_wrap:n` or `__msg_show_variable:NNNnn`, in other words until the next time the ε -T_EX primitive `\showtokens` would have been used for showing to the terminal or until the next **variable-not-defined** error.

```

\__msg_show_pre:nnnnnn \__msg_show_pre:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>}
\__msg_show_pre:(nnxxxx|nnnnnV) {<arg three>} {<arg four>}

```

New: 2015-08-05

Prints the *<message>* from *<module>* in the terminal (or log file if `__msg_log_next:` was issued) without formatting. Used in messages which print complex variable contents completely.

<code>_msg_show_variable:NNNnn</code>
New: 2015-08-04

`_msg_show_variable:NNNnn` $\langle variable \rangle$ $\langle if-exist \rangle$ $\langle if-empty \rangle$ $\{ \langle msg \rangle \}$ $\{ \langle formatted content \rangle \}$

If the $\langle variable \rangle$ does not exist according to $\langle if-exist \rangle$ (typically `\cs_if_exist:NTF`) then throw an error and do nothing more. Otherwise, if $\langle msg \rangle$ is not empty, display the message `LaTeX/kernel/show- $\langle msg \rangle$` with `\token_to_str:N` $\langle variable \rangle$ as a first argument, and a second argument that is `?` or empty depending on the result of $\langle if-empty \rangle$ (typically `\tl_if_empty:NTF`) on the $\langle variable \rangle$. Then display the $\langle formatted content \rangle$ by giving it as an argument to `_msg_show_wrap:n`.

<code>_msg_show_wrap:Nn</code>
New: 2015-08-03
Updated: 2015-08-07

`_msg_show_wrap:Nn` $\langle function \rangle$ $\{ \langle expression \rangle \}$

Shows or logs the $\langle expression \rangle$ (turned into a string), an equal sign, and the result of applying the $\langle function \rangle$ to the $\{ \langle expression \rangle \}$. For instance, if the $\langle function \rangle$ is `\int_eval:n` and the $\langle expression \rangle$ is `1+2` then this will log `> 1+2=3`. The case where the $\langle function \rangle$ is `\tl_to_str:n` is special: then the string representation of the $\langle expression \rangle$ is only logged once.

<code>_msg_show_wrap:n</code>
New: 2015-08-03

`_msg_show_wrap:n` $\{ \langle formatted text \rangle \}$

Shows or logs the $\langle formatted text \rangle$. After expansion, unless it is empty, the $\langle formatted text \rangle$ must contain `>`, and the part of $\langle formatted text \rangle$ before the first `>` is removed. Failure to do so causes low-level \TeX errors.

<code>_msg_show_item:n</code>
<code>_msg_show_item:nn</code>
<code>_msg_show_item_unbraced:nn</code>
Updated: 2012-09-09

`_msg_show_item:n` $\langle item \rangle$
`_msg_show_item:nn` $\langle item-key \rangle$ $\langle item-value \rangle$

Auxiliary functions used within the last argument of `_msg_show_variable:NNNnn` or `_msg_show_wrap:n` to format variable items correctly for display. The `_msg_show_item:n` version is used for simple lists, the `_msg_show_item:nn` and `_msg_show_item_unbraced:nn` versions for key-value like data structures.

<code>\c_msg_coding_error_text_tl</code>

The text

This is a coding error.

used by kernel functions when erroneous programming input is encountered.

Part XX

The l3keys 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` will be 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`; spaces are *ignored* in key names. As will be 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
}
```

will create a key called `\l_mymodule_tmp_tl`, and not one called `key`.

1 Creating keys

`\keys_define:nn`

Updated: 2015-11-07

`\keys_define:nn {<module>} {<keyval list>}`

Parses the *<keyval list>* and defines the keys listed there for *<module>*. The *<module>* name should be a text value, but there are no restrictions on the nature of the text. 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.*, will override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so will 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 will define the key within the current T_EX scope.

```

.bool_set:N
.bool_set:c
.bool_gset:N
.bool_gset:c

```

Updated: 2013-07-08

$\langle key \rangle$.bool_set:N = $\langle boolean \rangle$

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to $\langle value \rangle$ (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.

```

.bool_set_inverse:N
.bool_set_inverse:c
.bool_gset_inverse:N
.bool_gset_inverse:c

```

New: 2011-08-28

Updated: 2013-07-08

$\langle key \rangle$.bool_set_inverse:N = $\langle boolean \rangle$

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to the logical inverse of $\langle value \rangle$ (which must be either `true` or `false`). If the $\langle boolean \rangle$ does not exist, it will be created globally at the point that the key is set up.

```

.choice:

```

$\langle key \rangle$.choice:

Sets $\langle key \rangle$ to act as a choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section 3.

```

.choices:nn
.choices:(Vn|on|xn)

```

New: 2011-08-21

Updated: 2013-07-10

$\langle key \rangle$.choices:nn = $\{\langle choices \rangle\}$ $\{\langle code \rangle\}$

Sets $\langle key \rangle$ to act as a choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of $\langle choices \rangle$ (indexed from 1). Choices are discussed in detail in section 3.

```

.clist_set:N
.clist_set:c
.clist_gset:N
.clist_gset:c

```

New: 2011-09-11

$\langle key \rangle$.clist_set:N = $\langle comma list variable \rangle$

Defines $\langle key \rangle$ to set $\langle comma list variable \rangle$ to $\langle value \rangle$. Spaces around commas and empty items will be stripped. If the variable does not exist, it will be created globally at the point that the key is set up.

```

.code:n

```

Updated: 2013-07-10

$\langle key \rangle$.code:n = $\{\langle code \rangle\}$

Stores the $\langle code \rangle$ for execution when $\langle key \rangle$ is used. The $\langle code \rangle$ can include one parameter (#1), which will be the $\langle value \rangle$ given for the $\langle key \rangle$. The x-type variant will expand $\langle code \rangle$ at the point where the $\langle key \rangle$ is created.

`.default:n`
`.default:(V|o|x)`
Updated: 2013-07-09

$\langle key \rangle$ `.default:n = { $\langle default \rangle$ }`

Creates a $\langle default \rangle$ value for $\langle key \rangle$, which is used if no value is given. This will be used if only the key name is given, but not if a blank $\langle value \rangle$ 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 will not trigger an error.

`.dim_set:N`
`.dim_set:c`
`.dim_gset:N`
`.dim_gset:c`

$\langle key \rangle$ `.dim_set:N = $\langle dimension \rangle$`

Defines $\langle key \rangle$ to set $\langle dimension \rangle$ to $\langle value \rangle$ (which must a dimension expression). If the variable does not exist, it will be created globally at the point that the key is set up.

`.fp_set:N`
`.fp_set:c`
`.fp_gset:N`
`.fp_gset:c`

$\langle key \rangle$ `.fp_set:N = $\langle floating point \rangle$`

Defines $\langle key \rangle$ to set $\langle floating point \rangle$ to $\langle value \rangle$ (which must a floating point expression). If the variable does not exist, it will be created globally at the point that the key is set up.

`.groups:n`
New: 2013-07-14

$\langle key \rangle$ `.groups:n = { $\langle groups \rangle$ }`

Defines $\langle key \rangle$ as belonging to the $\langle groups \rangle$ declared. Groups provide a “secondary axis” for selectively setting keys, and are described in Section 6.

`.inherit:n`
New: 2016-11-22

$\langle key \rangle$ `.inherit:n = { $\langle parents \rangle$ }`

Specifies that the $\langle key \rangle$ path should inherit the keys listed as $\langle parents \rangle$. For example, with setting

```
\keys_define:n { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:n { } { bar .inherit:n = foo }
```

setting

```
\keys_set:n { bar } { test = a }
```

will be equivalent to

```
\keys_set:n { foo } { test = a }
```

<hr/> <code>.initial:n</code> <hr/>	<code><key> .initial:n = {<value>}</code>
<code>.initial:(V o x)</code>	Initialises the <code><key></code> with the <code><value></code> , equivalent to
<hr/> Updated: 2013-07-09 <hr/>	<code>\keys_set:nn {<module>} { <key> = <value> }</code>
<hr/> <code>.int_set:N</code> <hr/>	<code><key> .int_set:N = <integer></code>
<code>.int_set:c</code>	Defines <code><key></code> to set <code><integer></code> to <code><value></code> (which must be an integer expression). If the
<code>.int_gset:N</code>	variable does not exist, it will be created globally at the point that the key is set up.
<code>.int_gset:c</code> <hr/>	
<hr/> <code>.meta:n</code> <hr/>	<code><key> .meta:n = {<keyval list>}</code>
<hr/> Updated: 2013-07-10 <hr/>	Makes <code><key></code> a meta-key, which will set <code><keyval list></code> in one go. If <code><key></code> is given with a
	value at the time the key is used, then the value will be passed through to the subsidiary
	<code><keys></code> for processing (as #1).
<hr/> <code>.meta:nn</code> <hr/>	<code><key> .meta:nn = {<path>} {<keyval list>}</code>
<hr/> New: 2013-07-10 <hr/>	Makes <code><key></code> a meta-key, which will set <code><keyval list></code> in one go using the <code><path></code> in place of
	the current one. If <code><key></code> is given with a value at the time the key is used, then the value
	will be passed through to the subsidiary <code><keys></code> for processing (as #1).
<hr/> <code>.multichoice:</code> <hr/>	<code><key> .multichoice:</code>
<hr/> New: 2011-08-21 <hr/>	Sets <code><key></code> to act as a multiple choice key. Each valid choice for <code><key></code> must then be
	created, as discussed in section 3.
<hr/> <code>.multichoices:nn</code> <hr/>	<code><key> .multichoices:nn {<choices>} {<code>}</code>
<code>.multichoices:(Vn on xn)</code>	Sets <code><key></code> to act as a multiple choice key, and defines a series <code><choices></code> which are im-
<hr/> New: 2011-08-21 <hr/>	plemented using the <code><code></code> . Inside <code><code></code> , <code>\l_keys_choice_tl</code> will be the name of the
<hr/> Updated: 2013-07-10 <hr/>	choice made, and <code>\l_keys_choice_int</code> will be the position of the choice in the list of
	<code><choices></code> (indexed from 1). Choices are discussed in detail in section 3.
<hr/> <code>.skip_set:N</code> <hr/>	<code><key> .skip_set:N = <skip></code>
<code>.skip_set:c</code>	Defines <code><key></code> to set <code><skip></code> to <code><value></code> (which must be a skip expression). If the variable
<code>.skip_gset:N</code>	does not exist, it will be created globally at the point that the key is set up.
<code>.skip_gset:c</code> <hr/>	
<hr/> <code>.tl_set:N</code> <hr/>	<code><key> .tl_set:N = <token list variable></code>
<code>.tl_set:c</code>	Defines <code><key></code> to set <code><token list variable></code> to <code><value></code> . If the variable does not exist, it will
<code>.tl_gset:N</code>	be created globally at the point that the key is set up.
<code>.tl_gset:c</code> <hr/>	
<hr/> <code>.tl_set_x:N</code> <hr/>	<code><key> .tl_set_x:N = <token list variable></code>
<code>.tl_set_x:c</code>	Defines <code><key></code> to set <code><token list variable></code> to <code><value></code> , which will be subjected to an x-
<code>.tl_gset_x:N</code>	type expansion (<i>i.e.</i> using <code>\tl_set:Nx</code>). If the variable does not exist, it will be created
<code>.tl_gset_x:c</code> <hr/>	globally at the point that the key is set up.

<code>.undefine:</code>	<code><key> .undefine:</code>
-------------------------	-------------------------------------

New: 2015-07-14	Removes the definition of the <code><key></code> within the current scope.
-----------------	--

<code>.value_forbidden:n</code>	<code><key> .value_forbidden:n = true false</code>
---------------------------------	--

New: 2015-07-14	Specifies that <code><key></code> cannot receive a <code><value></code> when used. If a <code><value></code> is given then an error will be issued. Setting the property <code>false</code> will cancel the restriction.
-----------------	--

<code>.value_required:n</code>	<code><key> .value_required:n = true false</code>
--------------------------------	---

New: 2015-07-14	Specifies that <code><key></code> must receive a <code><value></code> when used. If a <code><value></code> is not given then an error will be issued. Setting the property <code>false</code> will cancel 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 { module / 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 `module/subgroup/key`.

As will be 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:nnxxx { 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: 2015-11-07

```
\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 will be illustrated later.

```
\l_keys_key_tl  
\l_keys_path_tl  
\l_keys_value_tl
```

Updated: 2015-07-14

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_tl`, and will have been processed by `\tl_to_str:n`.

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_tl`, and will have been processed by `\tl_to_str:n`.

5 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` will look 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_tl'~to~'#1'.  
}
```

```

\keys_set_known:nnN      \keys_set_known:nnN {<module>} {<keyval list>} {<tl>}
\keys_set_known:(nVN|nvN|noN)
\keys_set_known:nn
\keys_set_known:(nV|nv|no)

```

New: 2011-08-23
Updated: 2015-11-07

In some cases, the desired behavior is to simply ignore unknown keys, collecting up information on these for later processing. The `\keys_set_known:nnN` 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. The key-value pairs for each *unknown* key name will be stored in the `<tl>` in a comma-separated form (*i.e.* an edited version of the `<keyval list>`). The `\keys_set_known:nn` version skips this stage.

Use of `\keys_set_known:nnN` can be nested, with the correct residual `<keyval list>` returned at each stage.

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` will attempt 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_tl         ,
  key-three .groups:n = { second }         ,
  key-four  .fp_set:N = \l_my_a_fp         ,
}

```

will assign `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:nnnN</code>	<code>\keys_set_filter:nnnN {<module>} {<groups>} {<keyval list>} <tl></code>
<code>\keys_set_filter:(nnVN nnvN nnoN)</code>	
<code>\keys_set_filter:nnn</code>	
<code>\keys_set_filter:(nnV nnv nno)</code>	

New: 2013-07-14

Updated: 2015-11-07

Activates key filtering in an “opt-out” sense: keys assigned to any of the $\langle groups \rangle$ specified will be ignored. The $\langle groups \rangle$ are given as a comma-separated list. Unknown keys are not assigned to any group and will thus always be set. The key–value pairs for each key which is filtered out will be 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.

<code>\keys_set_groups:nnn</code>	<code>\keys_set_groups:nnn {<module>} {<groups>} {<keyval list>}</code>
<code>\keys_set_groups:(nnV nnv nno)</code>	

New: 2013-07-14

Updated: 2015-11-07

Activates key filtering in an “opt-in” sense: only keys assigned to one or more of the $\langle groups \rangle$ specified will be set. The $\langle groups \rangle$ are given as a comma-separated list. Unknown keys are not assigned to any group and will thus never be set.

7 Utility functions for keys

<code>\keys_if_exist_p:nn</code> ★	<code>\keys_if_exist_p:nn {<module>} {<key>}</code>
<code>\keys_if_exist:nnTF</code> ★	<code>\keys_if_exist:nnTF {<module>} {<key>} {<true code>} {<false code>}</code>

Updated: 2015-11-07

Tests if the $\langle key \rangle$ exists for $\langle module \rangle$, *i.e.* if any code has been defined for $\langle key \rangle$.

<code>\keys_if_choice_exist_p:nnn</code> ★	<code>\keys_if_choice_exist_p:nnn {<module>} {<key>} {<choice>}</code>
<code>\keys_if_choice_exist:nnnTF</code> ★	<code>\keys_if_choice_exist:nnnTF {<module>} {<key>} {<choice>} {<true code>} {<false code>}</code>

New: 2011-08-21

Updated: 2015-11-07

Tests if the $\langle choice \rangle$ is defined for the $\langle key \rangle$ within the $\langle module \rangle$, *i.e.* if any code has been defined for $\langle key \rangle / \langle choice \rangle$. The test is **false** if the $\langle key \rangle$ itself is not defined.

<code>\keys_show:nn</code>	<code>\keys_show:nn {<module>} {<key>}</code>
----------------------------	---

Updated: 2015-08-09

Shows the information associated to the $\langle key \rangle$ for a $\langle module \rangle$, including the function which is used to actually implement it.

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 $\langle key-value list \rangle$ into $\langle keys \rangle$ and associated $\langle values \rangle$. 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 will have exactly one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

`\keyval_parse:NNn`

Updated: 2011-09-08

`\keyval_parse:NNn` $\langle function_1 \rangle$ $\langle function_2 \rangle$ $\{\langle key-value list \rangle\}$

Parses the $\langle key-value list \rangle$ into a series of $\langle keys \rangle$ and associated $\langle values \rangle$, or keys alone (if no $\langle value \rangle$ was given). $\langle function_1 \rangle$ should take one argument, while $\langle function_2 \rangle$ should absorb two arguments. After `\keyval_parse:NNn` has parsed the $\langle key-value list \rangle$, $\langle function_1 \rangle$ will be used to process keys given with no value and $\langle function_2 \rangle$ will be used to process keys given with a value. The order of the $\langle keys \rangle$ in the $\langle key-value list \rangle$ will be preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

will be 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 $\langle key \rangle$ and $\langle value \rangle$, then one *outer* set of braces is removed from the $\langle key \rangle$ and $\langle value \rangle$ as part of the processing.

Part XXI

The l3file 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 T_EX will attempt to locate them both the operating system path and entries in the T_EX file database (most T_EX systems use such a database). Thus the “current path” for T_EX is somewhat broader than that for other programs.

For functions which expect a *<file name>* argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Any active characters (as declared in `\l_char_active_seq`) will *not* be expanded, allowing the direct use of these in file names. File names will be quoted using `"` tokens if they contain spaces: as a result, `"` tokens are *not* permitted in file names.

1 File operation functions

`\g_file_current_name_tl`

Contains the name of the current L^AT_EX file. This variable should not be modified: it is intended for information only. It will be equal to `\c_sys_jobname_str` at the start of a L^AT_EX run and will be modified each time a file is read using `\file_input:n`.

`\file_if_exist:nTF`

Updated: 2012-02-10

`\file_if_exist:nTF {<file name>} {<true code>} {<false code>}`

Searches for *<file name>* using the current T_EX search path and the additional paths controlled by `\file_path_include:n`.

`\file_add_path:nN`

Updated: 2012-02-10

`\file_add_path:nN {<file name>} <tl var>`

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. If the file is not found then the *<tl var>* will contain the marker `\q_no_value`.

`\file_input:n`

Updated: 2012-02-17

`\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 will be raised if the file is not found.

`\file_path_include:n`

Updated: 2012-07-04

`\file_path_include:n {<path>}`

Adds *<path>* to the list of those used to search when reading files. The assignment is local. The *<path>* is processed in the same way as a *<file name>*, *i.e.*, with x-type expansion except active characters.

<hr/> <code>\file_path_remove:n</code> <hr/>	<code>\file_path_remove:n {<path>}</code>
Updated: 2012-07-04	Removes $\langle path \rangle$ from the list of those used to search when reading files. The assignment is local. The $\langle path \rangle$ is processed in the same way as a $\langle file\ name \rangle$, <i>i.e.</i> , with <code>x</code> -type expansion except active characters.

<hr/> <code>\file_list:</code> <hr/>	<code>\file_list:</code>
	This function will list all files loaded using <code>\file_input:n</code> in the log file.

1.1 Input–output stream management

As T_EX is limited to 16 input streams and 16 output streams, direct use of the streams by the programmer is not supported in L^AT_EX3. 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.

<hr/> <code>\ior_new:N</code> <hr/>	<code>\ior_new:N <stream></code>
<code>\ior_new:c</code>	<code>\iow_new:N <stream></code>
<code>\iow_new:N</code>	
<code>\iow_new:c</code>	
New: 2011-09-26	Globally reserves the name of the $\langle stream \rangle$, either for reading or for writing as appropriate. The $\langle stream \rangle$ is not opened until the appropriate <code>\..._open:Nn</code> function is used. Attempting to use a $\langle stream \rangle$ which has not been opened is an error, and the $\langle stream \rangle$ will behave as the corresponding <code>\c_term_....</code>
Updated: 2011-12-27	

<hr/> <code>\ior_open:Nn</code> <hr/>	<code>\ior_open:Nn <stream> {<file name>}</code>
<code>\ior_open:cn</code>	
Updated: 2012-02-10	Opens $\langle file\ name \rangle$ for reading using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file\ name \rangle$ until a <code>\ior_close:N</code> instruction is given or the T _E X run ends.

<hr/> <code>\ior_open:NnTF</code> <hr/>	<code>\ior_open:NnTF <stream> {<file name>} {<true code>} {<false code>}</code>
<code>\ior_open:cnTF</code>	
New: 2013-01-12	Opens $\langle file\ name \rangle$ for reading using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file\ name \rangle$ until a <code>\ior_close:N</code> instruction is given or the T _E X run ends. The $\langle true\ code \rangle$ is then inserted into the input stream. If the file is not found, no error is raised and the $\langle false\ code \rangle$ is inserted into the input stream.

<hr/> <code>\iow_open:Nn</code> <hr/>	<code>\iow_open:Nn <stream> {<file name>}</code>
<code>\iow_open:cn</code>	
Updated: 2012-02-09	Opens $\langle file\ name \rangle$ for writing using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file\ name \rangle$ until a <code>\iow_close:N</code> instruction is given or the T _E X run ends. Opening a file for writing will clear any existing content in the file (<i>i.e.</i> writing is <i>not</i> additive).

<code>\ior_close:N</code>	<code>\ior_close:N <stream></code>
<code>\ior_close:c</code>	<code>\ior_close:N <stream></code>
<code>\iow_close:N</code>	
<code>\iow_close:c</code>	

Updated: 2012-07-31

<code>\ior_list_streams:</code>	<code>\ior_list_streams:</code>
<code>\iow_list_streams:</code>	<code>\iow_list_streams:</code>

Updated: 2015-08-01

Closes the $\langle stream \rangle$. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.

Displays a list of the file names associated with each open stream: intended for tracking down problems.

1.2 Reading from files

<code>\ior_get:NN</code>	<code>\ior_get:NN <stream> <token list variable></code>
--------------------------	---

New: 2012-06-24

Function that reads one or more lines (until an equal number of left and right braces are found) from the input $\langle stream \rangle$ and stores the result locally in the $\langle token list \rangle$ variable. If the $\langle stream \rangle$ is not open, input is requested from the terminal. The material read from the $\langle stream \rangle$ will be tokenized by T_EX according to the category codes in force when the function is used. Note that any blank lines will be converted to the token `\par`. Therefore, if skipping blank lines is requires a test such as

```
\ior_get:NN \l_my_stream \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
...
```

may be used. Also notice that if multiple lines are read to match braces then the resulting token list will contain `\par` tokens. As normal T_EX tokenization is in force, any lines which do not end in a comment character (usually `%`) will have the line ending converted to a space, so for example input

```
a b c
```

will result in a token list `a b c .`

T_EXhackers note: This protected macro expands to the T_EX primitive `\read` along with the `to` keyword.

\ior_str_get:NnNew: 2016-12-04

\ior_str_get:Nn $\langle stream \rangle$ $\langle token\ list\ variable \rangle$

Function that reads one line from the input $\langle stream \rangle$ and stores the result locally in the $\langle token\ list \rangle$ variable. If the $\langle stream \rangle$ is not open, input is requested from the terminal. The material is read from the $\langle stream \rangle$ 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 will always only read one line and any blank lines in the input will result in the $\langle token\ list\ variable \rangle$ being empty. Unlike **\ior_get:Nn**, line ends do not receive any special treatment. Thus input

a b c

will result in a token list a b c with the letters a, b, and c having category code 12.

TeXhackers note: This protected macro is a wrapper around the ε -TeX primitive **\readline**. However, the end-line character normally added by this primitive is not included in the result of **\ior_str_get:Nn**.

\ior_if_eof_p:N ★**\ior_if_eof:Ntf** ★Updated: 2012-02-10

\ior_if_eof_p:N $\langle stream \rangle$ **\ior_if_eof:Ntf** $\langle stream \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests if the end of a $\langle stream \rangle$ has been reached during a reading operation. The test will also return a **true** value if the $\langle stream \rangle$ is not open.

2 Writing to files

\iow_now:Nn**\iow_now:(Nx|cn|cx)**Updated: 2012-06-05

\iow_now:Nn $\langle stream \rangle$ $\{\langle tokens \rangle\}$

This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ immediately (*i.e.* the write operation is called on expansion of **\iow_now:Nn**).

\iow_log:n**\iow_log:x****\iow_log:n** $\{\langle tokens \rangle\}$

This function writes the given $\langle tokens \rangle$ to the log (transcript) file immediately: it is a dedicated version of **\iow_now:Nn**.

\iow_term:n**\iow_term:x****\iow_term:n** $\{\langle tokens \rangle\}$

This function writes the given $\langle tokens \rangle$ 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** $\langle stream \rangle$ $\{\langle tokens \rangle\}$

This functions 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.* **\iow_shipout_x:Nn**).

TeXhackers note: When using expl3 with a format other than L^AT_EX, new line characters inserted using **\iow_newline:** or using the line-wrapping code **\iow_wrap:nnnN** will not be recognized in the argument of **\iow_shipout:Nn**. This may lead to the insertion of additionnal unwanted line-breaks.

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

Updated: 2012-09-08

`\iow_shipout_x:Nn <stream> {<tokens>}`

This functions 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).

T_EXhackers note: This is a wrapper around the T_EX primitive `\write`. When using `expl3` with a format other than L^AT_EX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` will not be recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additionnal unwanted line-breaks.

`\iow_char:N` ★

`\iow_char:N \<char>`

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 `\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 `\iow_now:Nn`).

T_EXhackers note: When using `expl3` with a format other than L^AT_EX, the character inserted by `\iow_newline:` will not be recognized by T_EX, which may lead to the insertion of additionnal unwanted line-breaks. This issue only affects `\iow_shipout:Nn`, `\iow_shipout_x:Nn` and direct uses of primitive operations.

2.1 Wrapping lines in output

`\iow_wrap:nnnN`

New: 2012-06-28
Updated: 2015-08-05

`\iow_wrap:nnnN` $\langle text \rangle$ $\langle run-on text \rangle$ $\langle set up \rangle$ $\langle function \rangle$

This function will wrap the $\langle text \rangle$ to a fixed number of characters per line. At the start of each line which is wrapped, the $\langle run-on text \rangle$ will be inserted. The line character count targeted will be the value of `\l_iow_line_count_int` minus the number of characters in the $\langle run-on text \rangle$ 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 $\langle text \rangle$ and $\langle run-on text \rangle$ are exhaustively expanded by the function, with the following substitutions:

- `\` 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_indent:n` may be used to indent a part of the $\langle text \rangle$ (not the $\langle run-on text \rangle$).

Additional functions may be added to the wrapping by using the $\langle set up \rangle$, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the $\langle text \rangle$ 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 $\langle function \rangle$, which will typically be a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the $\langle function \rangle$) will consist of characters of category “other” (category code 12), with the exception of spaces which will have category “space” (category code 10). This means that the output will *not* expand further when written to a file.

T_EXhackers note: Internally, `\iow_wrap:nnnN` carries out an `x`-type expansion on the $\langle text \rangle$ 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 $\langle text \rangle$.

`\iow_indent:n`

New: 2011-09-21

`\iow_indent:n` $\langle text \rangle$

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents $\langle text \rangle$ by four spaces. This function will not cause a line break, and only affects lines which start within the scope of the $\langle text \rangle$. In case the indented $\langle text \rangle$ 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 T_EX system in use: the standard value is 78, which is typically correct for unmodified T_EXlive and MiK_TE_X systems.

`\c_catcode_other_space_tl`

New: 2011-09-05

Token list containing one character with category code 12, (“other”), and character code 32 (space).

2.2 Constant input–output streams

<code>\c_term_ior</code>	Constant input stream for reading from the terminal. Reading from this stream using <code>\ior_get:NN</code> or similar will result in a prompt from T _E X of the form
--------------------------	---

`<tl>=`

<code>\c_log_iow</code> <code>\c_term_iow</code>	Constant output streams for writing to the log and to the terminal (plus the log), respectively.
---	--

2.3 Primitive conditionals

<code>\if_eof:w</code> ★	<pre> \if_eof:w <stream> <true code> \else: <false code> \fi: </pre> <p>Tests if the <code><stream></code> returns “end of file”, which is true for non-existent files. The <code>\else:</code> branch is optional.</p>
--------------------------	---

T_EXhackers note: This is the T_EX primitive `\ifeof`.

2.4 Internal file functions and variables

<code>\g_file_internal_ior</code>	Used to test for the existence of files when opening.
-----------------------------------	---

<code>\l_file_internal_name_tl</code>	Used to return the full name of a file for internal use. This is set by <code>\file_if_exist:nTF</code> and <code>__file_if_exist:nT</code> , and the value may then be used to load a file directly provided no further operations intervene.
---------------------------------------	---

<code>__file_name_sanitize:nn</code>	<code>__file_name_sanitize:nn {<name>} {<tokens>}</code>
---------------------------------------	---

<small>New: 2012-02-09</small>	Exhaustively-expands the <code><name></code> with the exception of any category <code><active></code> (catcode 13) tokens, which are not expanded. The list of <code><active></code> tokens is taken from <code>\l_char_active_seq</code> . The <code><sanitized name></code> is then inserted (in braces) after the <code><tokens></code> , which should further process the file name. If any spaces are found in the name after expansion, an error is raised.
--------------------------------	---

2.5 Internal input–output functions

<code>__ior_open:Nn</code>	<code>__ior_open:Nn <stream> {<file name>}</code>
-----------------------------	--

<code>__ior_open:No</code>	This function has identical syntax to the public version. However, it does not take precautions against active characters in the <code><file name></code> , and it does not attempt to add a <code><path></code> to the <code><file name></code> : it is therefore intended to be used by higher-level functions which have already fully expanded the <code><file name></code> and which need to perform multiple open or close operations. See for example the implementation of <code>\file_add_path:nN</code> ,
-----------------------------	---

`_iow_with:Nnn`

`New: 2014-08-23`

`_iow_with:Nnn` $\langle integer \rangle$ $\{\langle value \rangle\}$ $\{\langle code \rangle\}$

If the $\langle integer \rangle$ is equal to the $\langle value \rangle$ then this function simply runs the $\langle code \rangle$. Otherwise it saves the current value of the $\langle integer \rangle$, sets it to the $\langle value \rangle$, runs the $\langle code \rangle$, and restores the $\langle integer \rangle$ 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.

Part XXII

The l3fp 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: $x < y$, $x \leq y$, $x > y$, $x \neq y$ etc.
 - Boolean logic: negation $!x$, conjunction $x \&\& y$, disjunction $x || y$, ternary operator $x ? y : z$.
 - Exponentials: $\exp x$, $\ln x$, x^y .
 - Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and $\text{sin}d\,x$, $\text{cos}d\,x$, $\text{tan}d\,x$, $\text{cot}d\,x$, $\text{sec}d\,x$, $\text{csc}d\,x$ 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 $\text{asind}\,x$, $\text{acosd}\,x$, $\text{atand}\,x$, $\text{acotd}\,x$, $\text{asecd}\,x$, $\text{acscd}\,x$ 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: $\max(x, y, \dots)$, $\min(x, y, \dots)$, $\text{abs}(x)$.
 - Rounding functions ($n = 0$ by default, $t = \text{NaN}$ 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: $\text{rand}()$, $\text{randint}(m, n)$ in pdfTeX and LuaTeX engines.
 - 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 points, expressing dimensions in points and ignoring the stretch and shrink components of skips.

Floating point numbers can be given either explicitly (in a form such as `1.234e-34`, or `-.0001`), or as a stored floating point variable, which is automatically replaced by its current value. 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} $.`

But in all fairness, this 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.

```
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnun } { m }
{ \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\calcnun { 2 pi * sin ( 2.3 ^ 5 ) }
```

1 Creating and initialising floating point variables

<code>\fp_new:N</code>	<code>\fp_new:N <fp var></code>
<code>\fp_new:c</code>	
Updated: 2012-05-08	Creates a new <i><fp var></i> or raises an error if the name is already taken. The declaration is global. The <i><fp var></i> will initially be +0.

<code>\fp_const:Nn</code>	<code>\fp_const:Nn <fp var> {<floating point expression>}</code>
<code>\fp_const:cn</code>	
Updated: 2012-05-08	Creates a new constant <i><fp var></i> or raises an error if the name is already taken. The <i><fp var></i> will be set globally equal to the result of evaluating the <i><floating point expression></i> .

<code>\fp_zero:N</code>	<code>\fp_zero:N <fp var></code>
<code>\fp_zero:c</code>	
<code>\fp_gzero:N</code>	Sets the <i><fp var></i> to +0.
<code>\fp_gzero:c</code>	
Updated: 2012-05-08	

<code>\fp_zero_new:N</code>	<code>\fp_zero_new:N <fp var></code>
<code>\fp_zero_new:c</code>	
<code>\fp_gzero_new:N</code>	Ensures that the <i><fp var></i> exists globally by applying <code>\fp_new:N</code> if necessary, then applies <code>\fp_(g)zero:N</code> to leave the <i><fp var></i> set to +0.
<code>\fp_gzero_new:c</code>	
Updated: 2012-05-08	

2 Setting floating point variables

<code>\fp_set:Nn</code>	<code>\fp_set:Nn <fp var> {<floating point expression>}</code>
<code>\fp_set:cn</code>	
<code>\fp_gset:Nn</code>	Sets <i><fp var></i> equal to the result of computing the <i><floating point expression></i> .
<code>\fp_gset:cn</code>	
Updated: 2012-05-08	

<code>\fp_set_eq:NN</code>	<code>\fp_set_eq:NN <fp var₁> <fp var₂></code>
<code>\fp_set_eq:(cN Nc cc)</code>	Sets the floating point variable $\langle fp\ var_1 \rangle$ equal to the current value of $\langle fp\ var_2 \rangle$.
<code>\fp_gset_eq:NN</code>	
<code>\fp_gset_eq:(cN Nc cc)</code>	
Updated: 2012-05-08	

<code>\fp_add:Nn</code>	<code>\fp_add:Nn <fp var> {<floating point expression>}</code>
<code>\fp_add:cn</code>	
<code>\fp_gadd:Nn</code>	Adds the result of computing the $\langle floating\ point\ expression \rangle$ to the $\langle fp\ var \rangle$.
<code>\fp_gadd:cn</code>	
Updated: 2012-05-08	

<code>\fp_sub:Nn</code>	<code>\fp_sub:Nn <fp var> {<floating point expression>}</code>
<code>\fp_sub:cn</code>	
<code>\fp_gsub:Nn</code>	Subtracts the result of computing the $\langle floating\ point\ expression \rangle$ from the $\langle fp\ var \rangle$.
<code>\fp_gsub:cn</code>	
Updated: 2012-05-08	

3 Using floating point numbers

<code>\fp_eval:n</code> ★	<code>\fp_eval:n {<floating point expression>}</code>
New: 2012-05-08	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. This function is identical to <code>\fp_to_decimal:n</code> .
Updated: 2012-07-08	

<code>\fp_to_decimal:N</code> ★	<code>\fp_to_decimal:N <fp var></code>
<code>\fp_to_decimal:c</code> ★	<code>\fp_to_decimal:n {<floating point expression>}</code>
<code>\fp_to_decimal:n</code> ★	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.
New: 2012-05-08	
Updated: 2012-07-08	

<code>\fp_to_dim:N</code> ★	<code>\fp_to_dim:N <fp var></code>
<code>\fp_to_dim:c</code> ★	<code>\fp_to_dim:n {<floating point expression>}</code>
<code>\fp_to_dim:n</code> ★	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 <code>\fp_to_decimal:n</code> , 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 T _E X dimensions, leading to overflow errors if used as a dimension. The values $\pm\infty$ and NaN trigger an “invalid operation” exception.
Updated: 2016-03-22	

<code>\fp_to_int:N</code>	★	<code>\fp_to_int:N <fp var></code>
<code>\fp_to_int:c</code>	★	<code>\fp_to_int:n {<floating point expression>}</code>
<code>\fp_to_int:n</code>	★	Evaluates the <i><floating point expression></i> , 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. The values $\pm\infty$ and NaN trigger an “invalid operation” exception.

Updated: 2012-07-08

<code>\fp_to_scientific:N</code>	★	<code>\fp_to_scientific:N <fp var></code>
<code>\fp_to_scientific:c</code>	★	<code>\fp_to_scientific:n {<floating point expression>}</code>
<code>\fp_to_scientific:n</code>	★	Evaluates the <i><floating point expression></i> and expresses the result in scientific notation:

New: 2012-05-08
Updated: 2016-03-22

<optional -><digit>.<15 digits>e<optional sign><exponent>

The leading *<digit>* 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).

<code>\fp_to_tl:N</code>	★	<code>\fp_to_tl:N <fp var></code>
<code>\fp_to_tl:c</code>	★	<code>\fp_to_tl:n {<floating point expression>}</code>
<code>\fp_to_tl:n</code>	★	Evaluates the <i><floating point expression></i> 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 (see <code>\fp_to_scientific:n</code>). 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 <code>\fp_to_decimal:n</code>). Negative numbers start with - . The special values ± 0 , $\pm\infty$ and NaN are rendered as 0 , -0 , inf , -inf , and nan respectively. Normal category codes apply and thus inf or nan , if produced, will be made up of letters.

Updated: 2016-03-22

<code>\fp_use:N</code>	★	<code>\fp_use:N <fp var></code>
<code>\fp_use:c</code>	★	Inserts the value of the <i><fp var></i> 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. This function is identical to <code>\fp_to_decimal:N</code> .

Updated: 2012-07-08

4 Floating point conditionals

<code>\fp_if_exist_p:N</code>	★	<code>\fp_if_exist_p:N <fp var></code>
<code>\fp_if_exist_p:c</code>	★	<code>\fp_if_exist:NTF <fp var> {<true code>} {<false code>}</code>
<code>\fp_if_exist:NTF</code>	★	Tests whether the <i><fp var></i> is currently defined. This does not check that the <i><fp var></i> really is a floating point variable.
<code>\fp_if_exist:cTF</code>	★	

Updated: 2012-05-08

<code>\fp_compare_p:nNn</code> ★ <code>\fp_compare:nNnTF</code> ★	<code>\fp_compare_p:nNn {<fpexpr₁>} <relation> {<fpexpr₂>}</code> <code>\fp_compare:nNnTF {<fpexpr₁>} <relation> {<fpexpr₂>} {<true code>} {<false code>}</code>
--	---

Updated: 2012-05-08

Compares the $\langle fpexpr_1 \rangle$ and the $\langle fpexpr_2 \rangle$, and returns **true** if the $\langle relation \rangle$ is obeyed. Two floating point numbers x and y may obey four mutually exclusive relations: $x \langle y, x=y, x \rangle y$, or x and y are not ordered. The latter case occurs exactly when either operand is NaN, and this relation is denoted by the symbol ?. 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

```

<code>\fp_compare_p:n</code> ★ <code>\fp_compare:nTF</code> ★	<code>\fp_compare_p:n</code> <code>{</code> <code> <fpexpr₁> <relation₁></code> <code> ...</code> <code> <fpexpr_N> <relation_N></code> <code> <fpexpr_{N+1}></code> <code>}</code> <code>\fp_compare:nTF</code> <code>{</code> <code> <fpexpr₁> <relation₁></code> <code> ...</code> <code> <fpexpr_N> <relation_N></code> <code> <fpexpr_{N+1}></code> <code>}</code> <code>{<true code>} {<false code>}</code>
--	---

Updated: 2012-12-14

Evaluates the $\langle floating point expressions \rangle$ as described for `\fp_eval:n` and compares consecutive result using the corresponding $\langle relation \rangle$, namely it compares $\langle intexpr_1 \rangle$ and $\langle intexpr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle intexpr_2 \rangle$ and $\langle intexpr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle intexpr_N \rangle$ and $\langle intexpr_{N+1} \rangle$ using the $\langle relation_N \rangle$. The test yields **true** if all comparisons are **true**. Each $\langle floating point expression \rangle$ is evaluated only once. Contrarily to `\int_compare:nTF`, all $\langle floating point expressions \rangle$ are computed, even if one comparison is **false**. Two floating point numbers x and y may obey four mutually exclusive relations: $x \langle y, x=y, x \rangle y$, or x and y are not ordered. The latter case occurs exactly when one of the operands is NaN, and this relation is denoted by the symbol ?. Each $\langle relation \rangle$ can be any (non-empty) combination of $<$, $=$, $>$, and $?$, plus an optional leading ! (which negates the $\langle relation \rangle$), with the restriction that the $\langle relation \rangle$ may not start with $?$, as this symbol has a different meaning (in combination with $:$) within floatin point expressions. The comparison $x \langle relation \rangle y$ is then **true** if the $\langle relation \rangle$ does not start with ! and the actual relation ($<$, $=$, $>$, or $?$) between x and y appears within the $\langle relation \rangle$, or on the contrary if the $\langle relation \rangle$ starts with ! and the relation between x and y does not appear within the $\langle relation \rangle$. Common choices of $\langle relation \rangle$ include \geq (greater or equal), \neq (not equal), $!?$ or \leq (comparable).

5 Floating point expression loops

<code>\fp_do_until:nNnn</code> ☆	<code>\fp_do_until:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
New: 2012-08-16	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> . If the test is false then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is true .
<code>\fp_do_while:nNnn</code> ☆	<code>\fp_do_while:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
New: 2012-08-16	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> . If the test is true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<code>\fp_until_do:nNnn</code> ☆	<code>\fp_until_do:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<code>\fp_while_do:nNnn</code> ☆	<code>\fp_while_do:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .
<code>\fp_do_until:nn</code> ☆	<code>\fp_do_until:nn { <fpexpr₁> <relation> <fpexpr₂> } {<code>}</code>
New: 2012-08-16	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is false then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is true .
<code>\fp_do_while:nn</code> ☆	<code>\fp_do_while:nn { <fpexpr₁> <relation> <fpexpr₂> } {<code>}</code>
New: 2012-08-16	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<code>\fp_until_do:nn</code> ☆	<code>\fp_until_do:nn { <fpexpr₁> <relation> <fpexpr₂> } {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .

<hr/> <code>\fp_while_do:nn</code> ☆ <hr/>	<code>\fp_while_do:nn { <fpexpr₁> <relation> <fpexpr₂> } {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true. After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .

<hr/> <code>\fp_step_function:nnnN</code> ☆ <code>\fp_step_function:nnnc</code> ☆ <hr/>	<code>\fp_step_function:nnnN {<initial value>} {<step>} {<final value>} <function></code>
New: 2016-11-21 Updated: 2016-12-06	This function first evaluates the <i><initial value></i> , <i><step></i> and <i><final value></i> , all of which should be floating point expressions. The <i><function></i> is then placed in front of each <i><value></i> from the <i><initial value></i> to the <i><final value></i> in turn (using <i><step></i> between each <i><value></i>). The <i><step></i> must be non-zero. If the <i><step></i> is positive, the loop stops when the <i><value></i> becomes larger than the <i><final value></i> . If the <i><step></i> is negative, the loop stops when the <i><value></i> becomes smaller than the <i><final value></i> . The <i><function></i> 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]
```

T_EXhackers 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.

<hr/> <code>\fp_step_inline:nnnn</code> <hr/>	<code>\fp_step_inline:nnnn {<initial value>} {<step>} {<final value>} {<code>}</code>
New: 2016-11-21 Updated: 2016-12-06	This function first evaluates the <i><initial value></i> , <i><step></i> and <i><final value></i> , all of which should be floating point expressions. Then for each <i><value></i> from the <i><initial value></i> to the <i><final value></i> in turn (using <i><step></i> between each <i><value></i>), the <i><code></i> is inserted into the input stream with #1 replaced by the current <i><value></i> . Thus the <i><code></i> should define a function of one argument (#1).

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`

New: 2012-05-08

Infinity, with either sign. These can be input directly in a floating point expression as **inf** and **-inf**.

<hr/> <code>\c_e_fp</code> <hr/>	The value of the base of the natural logarithm, $e = \exp(1)$.
<hr/> Updated: 2012-05-08 <hr/>	
<hr/> <code>\c_pi_fp</code> <hr/>	The value of π . This can be input directly in a floating point expression as <code>pi</code> .
<hr/> Updated: 2013-11-17 <hr/>	
<hr/> <code>\c_one_degree_fp</code> <hr/>	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 <code>deg</code> .
<hr/> New: 2012-05-08 Updated: 2013-11-17 <hr/>	
<hr/> <code>\l_tmpa_fp</code> <code>\l_tmpb_fp</code> <hr/>	Scratch floating points 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.
<hr/> <code>\g_tmpa_fp</code> <code>\g_tmpb_fp</code> <hr/>	Scratch floating points 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.

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 $0/0$, $\ln(0)$, or $10^{**1e9999}$. The IEEE standard defines 5 types of exceptions.

- *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 almost any operation involving a NaN. This normally results in a NaN, except for conversion functions whose target type does not have a notion of NaN (e.g., `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating e.g., $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.
- *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 is associated a “flag”, which can be either *on* or *off*. By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions only raise the corresponding flag. The state of the flag can be tested and modified. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and turn the flag on, or only turn the flag on, or do nothing at all.

<code>\fp_if_flag_on_p:n</code> ★	<code>\fp_if_flag_on_p:n {⟨exception⟩}</code>
<code>\fp_if_flag_on:nTF</code> ★	<code>\fp_if_flag_on:nTF {⟨exception⟩} {⟨true code⟩} {⟨false code⟩}</code>
New: 2012-08-08	Tests if the flag for the $\langle exception \rangle$ is on, which normally means the given $\langle exception \rangle$ has occurred. <i>This function is experimental, and may be altered or removed.</i>

<code>\fp_flag_off:n</code>	<code>\fp_flag_off:n {⟨exception⟩}</code>
New: 2012-08-08	Locally turns off the flag which indicates whether the $\langle exception \rangle$ has occurred. <i>This function is experimental, and may be altered or removed.</i>

<code>\fp_flag_on:n</code> ★	<code>\fp_flag_on:n {⟨exception⟩}</code>
New: 2012-08-08	Locally turns on the flag to indicate (or pretend) that the $\langle exception \rangle$ has occurred. Note that this function is expandable: it is used internally by <code>l3fp</code> to signal when exceptions do occur. <i>This function is experimental, and may be altered or removed.</i>

<code>\fp_trap:nn</code>	<code>\fp_trap:nn {⟨exception⟩} {⟨trap type⟩}</code>
New: 2012-07-19 Updated: 2012-08-08	All occurrences of the $\langle exception \rangle$ (<code>invalid_operation</code> , <code>division_by_zero</code> , <code>overflow</code> , or <code>underflow</code>) within the current group are treated as $\langle trap type \rangle$, which can be <ul style="list-style-type: none"> • none: the $\langle exception \rangle$ will be entirely ignored, and leave no trace; • flag: the $\langle exception \rangle$ will turn the corresponding flag on when it occurs; • error: additionally, the $\langle exception \rangle$ will halt the \TeX run and display some information about the current operation in the terminal.

This function is experimental, and may be altered or removed.

8 Viewing floating points

<code>\fp_show:N</code>	<code>\fp_show:N ⟨fp var⟩</code>
<code>\fp_show:c</code>	<code>\fp_show:n {⟨floating point expression⟩}</code>
<code>\fp_show:n</code>	Evaluates the $\langle floating point expression \rangle$ and displays the result in the terminal.
New: 2012-05-08 Updated: 2015-08-07	

9 Floating point expressions

9.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm 0.d_1d_2\dots d_{16} \cdot 10^n$, a normal floating point number, with $d_i \in [0, 9]$, $d_1 \neq 0$, and $|n| \leq 10000$;
- ± 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);

(*not yet*) subnormal numbers $\pm 0.d_1d_2 \dots d_{16} \cdot 10^{-10000}$ with $d_1 = 0$.

Normal floating point numbers are stored in base 10, with 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**, 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 0 characters, and an optional . character), 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.

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 will be ignored.
- Any unrecognizable string triggers an error, and produces a **NaN**.

Note that **e-1** is not a representation of 10^{-1} , because it could be mistaken with the difference of “e” and 1. This is consistent with several other programming languages. However, in order to avoid confusions, **e-1** is not considered to be this difference either. To input the base of natural logarithms, use **exp(1)** or **\c_e_fp**.

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 **+**, **-**, **!**.

- Binary `*`, `/`, and implicit multiplication by juxtaposition (`2pi`, `3(4+5)`, *etc.*).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc.*
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).

The precedence of operations can be overridden using parentheses. In particular, those precedences imply that

$$\begin{aligned}\sin 2\pi &= \sin(2\pi) = 0, \\ 2^{2\max(3,4)} &= 2^{2\max(3,4)} = 256.\end{aligned}$$

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`.

```
?: \fp_eval:n { <operand_1> ? <operand_2> : <operand_3> }
```

The ternary operator `?:` results in `<operand_2>` if `<operand_1>` is true, and `<operand_3>` if it is false (equal to ± 0). All three `<operands>` are evaluated in all cases. 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 { <operand_1> <operand_2> }
```

If `<operand_1>` is true (non-zero), use that value, otherwise the value of `<operand_2>`. Both `<operands>` are evaluated in all cases.

```
&& \fp_eval:n { <operand_1> && <operand_2> }
```

If `<operand_1>` is false (equal to ± 0), use that value, otherwise the value of `<operand_2>`. Both `<operands>` are evaluated in all cases.

```

<      \fp_eval:n
=      {
>      \langle operand_1 \rangle \langle relation_1 \rangle
?      ...
      \langle operand_N \rangle \langle relation_N \rangle
Updated: 2013-12-14 \langle operand_{N+1} \rangle
      }

```

Each $\langle relation \rangle$ consists of a non-empty string of $<$, $=$, $>$, and $?$, optionally preceded by $!$, and may not start with $?$. This evaluates to $+1$ if all comparisons $\langle operand_i \rangle \langle relation_j \rangle$ are true, and $+0$ otherwise. All $\langle operands \rangle$ are evaluated in all cases. See `\fp_compare:nTF` for details.

```

+ \fp_eval:n { \langle operand_1 \rangle + \langle operand_2 \rangle }
- \fp_eval:n { \langle operand_1 \rangle - \langle operand_2 \rangle }

```

Computes the sum or the difference of its two $\langle operands \rangle$. The “invalid operation” exception occurs for $\infty - \infty$. “Underflow” and “overflow” occur when appropriate.

```

* \fp_eval:n { \langle operand_1 \rangle * \langle operand_2 \rangle }
/ \fp_eval:n { \langle operand_1 \rangle / \langle operand_2 \rangle }

```

Computes the product or the ratio of its two $\langle operands \rangle$. 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.

```

+ \fp_eval:n { + \langle operand \rangle }
- \fp_eval:n { - \langle operand \rangle }
! \fp_eval:n { ! \langle operand \rangle }

```

The unary $+$ does nothing, the unary $-$ changes the sign of the $\langle operand \rangle$, and $! \langle operand \rangle$ evaluates to 1 if $\langle operand \rangle$ is false and 0 otherwise (this is the `not` boolean function). Those operations never raise exceptions.

```

** \fp_eval:n { \langle operand_1 \rangle ** \langle operand_2 \rangle }
^ \fp_eval:n { \langle operand_1 \rangle ^ \langle operand_2 \rangle }

```

Raises $\langle operand_1 \rangle$ to the power $\langle operand_2 \rangle$. This operation is right associative, hence `2 ** 2 ** 3` equals $2^{2^3} = 256$. The “invalid operation” exception occurs if $\langle operand_1 \rangle$ is negative or -0 , and $\langle operand_2 \rangle$ is not an integer, unless the result is zero (in that case, the sign is chosen arbitrarily to be $+0$). “Division by zero” occurs when raising ± 0 to a strictly negative power. “Underflow” and “overflow” occur when appropriate.

```

abs \fp_eval:n { abs( \langle fpexpr \rangle ) }

```

Computes the absolute value of the $\langle fpexpr \rangle$. This function does not raise any exception beyond those raised when computing its operand $\langle fpexpr \rangle$. See also `\fp_abs:n`.

```

exp \fp_eval:n { exp( \langle fpexpr \rangle ) }

```

Computes the exponential of the $\langle fpexpr \rangle$. “Underflow” and “overflow” occur when appropriate.

```
ln    \fp_eval:n { ln( <fpexpr> ) }
```

Computes the natural logarithm of the $\langle fpepr \rangle$. 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.

```

max  \fp_eval:n { max( <fpexpr1> , <fpexpr2> , ... ) }
min  \fp_eval:n { min( <fpexpr1> , <fpexpr2> , ... ) }

```

Evaluates each $\langle fpxpr \rangle$ and computes the largest (smallest) of those. If any of the $\langle fpxpr \rangle$ is a NaN, the result is NaN. Those operations do not raise exceptions.

round	<code>\fp_eval:n { round (<fpexpr>) }</code>
trunc	<code>\fp_eval:n { round (<fpexpr₁> , <fpexpr₂>) }</code>
ceil	<code>\fp_eval:n { round (<fpexpr₁> , <fpexpr₂> , <fpexpr₃>) }</code>
floor	Only round accepts a third argument. Evaluates <i><fpexpr></i> .

New: 2013-12-14
Updated: 2015-08-08

Only **round** accepts a third argument. Evaluates $\langle fpeexpr_1 \rangle = x$ and $\langle fpeexpr_2 \rangle = n$ and $\langle fpeexpr_3 \rangle = t$ 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 is neither $\pm \infty$ nor an integer, then an “invalid operation” exception is raised. When $\langle fpeexpr_2 \rangle$ is omitted, $n = 0$, i.e., $\langle fpeexpr_1 \rangle$ is rounded to an integer. The rounding direction depends on the function.

- **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**, or the deprecated **round-**, yields the largest multiple of 10^{-n} smaller or equal to x (“round towards negative infinity”);
- **ceil**, or the deprecated **round+**, yields the smallest multiple of 10^{-n} greater or equal to x (“round towards positive infinity”);
- **trunc**, or the deprecated **round0**, 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 $\langle fpepr_2 \rangle < -9984$).

<code>sin</code>	<code>\fp_eval:n { sin(<fpexpr>) }</code>
<code>cos</code>	<code>\fp_eval:n { cos(<fpexpr>) }</code>
<code>tan</code>	<code>\fp_eval:n { tan(<fpexpr>) }</code>
<code>cot</code>	<code>\fp_eval:n { cot(<fpexpr>) }</code>
<code>csc</code>	<code>\fp_eval:n { csc(<fpexpr>) }</code>
<code>sec</code>	<code>\fp_eval:n { sec(<fpexpr>) }</code>

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 analog $\text{sind}(8 \times 180)$ 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.

<code>sind</code>	<code>\fp_eval:n { sind(<fpexpr>) }</code>
<code>cosd</code>	<code>\fp_eval:n { cosd(<fpexpr>) }</code>
<code>tand</code>	<code>\fp_eval:n { tand(<fpexpr>) }</code>
<code>cotd</code>	<code>\fp_eval:n { cotd(<fpexpr>) }</code>
<code>cscd</code>	<code>\fp_eval:n { cscd(<fpexpr>) }</code>
<code>secd</code>	<code>\fp_eval:n { secd(<fpexpr>) }</code>

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 analog $\text{sind}(8 \times 180)$ 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.

<code>asin</code>	<code>\fp_eval:n { asin(<fpexpr>) }</code>
<code>acos</code>	<code>\fp_eval:n { acos(<fpexpr>) }</code>
<code>acsc</code>	<code>\fp_eval:n { acsc(<fpexpr>) }</code>
<code>asec</code>	<code>\fp_eval:n { asec(<fpexpr>) }</code>

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.

<code>asind</code>	<code>\fp_eval:n { asind(<fpexpr>) }</code>
<code>acosd</code>	<code>\fp_eval:n { acosd(<fpexpr>) }</code>
<code>acscd</code>	<code>\fp_eval:n { acscd(<fpexpr>) }</code>
<code>asecd</code>	<code>\fp_eval:n { asecd(<fpexpr>) }</code>

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.

atan	<code>\fp_eval:n { atan(<fpexpr>) }</code>
acot	<code>\fp_eval:n { atan(<fpexpr₁> , <fpexpr₂>) }</code>
<hr/>	
New: 2013-11-02	<code>\fp_eval:n { acot(<fpexpr>) }</code>
	<code>\fp_eval:n { acot(<fpexpr₁> , <fpexpr₂>) }</code>

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. Only the “underflow” exception can occur.

atand	<code>\fp_eval:n { atand(<fpexpr>) }</code>
acotd	<code>\fp_eval:n { atand(<fpexpr₁> , <fpexpr₂>) }</code>
<hr/>	
New: 2013-11-02	<code>\fp_eval:n { acotd(<fpexpr>) }</code>
	<code>\fp_eval:n { acotd(<fpexpr₁> , <fpexpr₂>) }</code>

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. Only the “underflow” exception can occur.

sqrt	<code>\fp_eval:n { sqrt(<fpexpr>) }</code>
-------------	--

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; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\text{NaN}} = \text{NaN}$.

<hr/> rand <hr/>	<code>\fp_eval:n { rand() }</code>
<hr/> New: 2016-12-05 <hr/>	Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded. Available in pdfTeX and LuaTeX engines only.
<p>TeXhackers note: This is based on pseudo-random numbers provided by the engine’s primitive <code>\pdfuniformdeviate</code> in pdfTeX and <code>\uniformdeviate</code> in LuaTeX. The underlying code in pdfTeX and LuaTeX is based on Metapost, which follows an additive scheme recommended in Section 3.6 of “The Art of Computer Programming, Volume 2”.</p> <p>While we are more careful than <code>\uniformdeviate</code> 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.</p> <p>The random seed can be queried using <code>\pdfrandomseed</code> and set using <code>\pdfsetrandomseed</code> (in LuaTeX <code>\randomseed</code> and <code>\setrandomseed</code>). 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 $[0, 2^{28} - 1]$.</p>	
<hr/> randint <hr/>	<code>\fp_eval:n { randint(<fpexpr>) }</code>
<hr/> New: 2016-12-05 <hr/>	<code>\fp_eval:n { randint(<fpexpr₁₂</code>
	Produces a pseudo-random integer between 1 and <code><fpexpr></code> or between <code><fpexpr_{1 and <code><fpexpr_{2 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.}</code>}</code>
<hr/> inf <hr/>	The special values $+\infty$, $-\infty$, and NaN are represented as <code>inf</code> , <code>-inf</code> and <code>nan</code> (see <code>\c_inf_fp</code> , <code>\c_minus_inf_fp</code> and <code>\c_nan_fp</code>).
<hr/> nan <hr/>	
<hr/> pi <hr/>	The value of π (see <code>\c_pi_fp</code>).
<hr/> deg <hr/>	The value of 1° in radians (see <code>\c_one_degree_fp</code>).

<hr/>	
<code>em</code>	Those units of measurement are equal to their values in <code>pt</code> , namely
<code>ex</code>	
<code>in</code>	$1\text{in} = 72.27\text{pt}$
<code>pt</code>	$1\text{pt} = 1\text{pt}$
<code>pc</code>	
<code>cm</code>	$1\text{pc} = 12\text{pt}$
<code>mm</code>	
<code>dd</code>	$1\text{cm} = \frac{1}{2.54}\text{in} = 28.45275590551181\text{pt}$
<code>cc</code>	
<code>nd</code>	$1\text{mm} = \frac{1}{25.4}\text{in} = 2.845275590551181\text{pt}$
<code>nc</code>	
<code>bp</code>	$1\text{dd} = 0.376065\text{mm} = 1.07000856496063\text{pt}$
<code>sp</code>	$1\text{cc} = 12\text{dd} = 12.84010277952756\text{pt}$
<hr/>	
	$1\text{nd} = 0.375\text{mm} = 1.066978346456693\text{pt}$
	$1\text{nc} = 12\text{nd} = 12.80374015748031\text{pt}$
	$1\text{bp} = \frac{1}{72}\text{in} = 1.00375\text{pt}$
	$1\text{sp} = 2^{-16}\text{pt} = 1.52587890625e - 5\text{pt}.$

The values of the (font-dependent) units `em` and `ex` are gathered from \TeX when the surrounding floating point expression is evaluated.

<hr/>	
<code>true</code>	Other names for 1 and +0.
<code>false</code>	
<hr/>	

<hr/>	
<code>\fp_abs:n</code> ★	<code>\fp_abs:n</code> $\{\langle\textit{floating point expression}\rangle\}$
New: 2012-05-14	
Updated: 2012-07-08	
<hr/>	
	Evaluates the $\langle\textit{floating point expression}\rangle$ as described for <code>\fp_eval:n</code> and leaves the absolute value of the result in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, <code>abs()</code> can be used.
<hr/>	
<code>\fp_max:nn</code> ★	<code>\fp_max:nn</code> $\{\langle\textit{fp expression 1}\rangle\} \{\langle\textit{fp expression 2}\rangle\}$
<code>\fp_min:nn</code> ★	
New: 2012-09-26	
<hr/>	
	Evaluates the $\langle\textit{floating point expressions}\rangle$ as described for <code>\fp_eval:n</code> and leaves the resulting larger (<code>max</code>) or smaller (<code>min</code>) value in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, <code>max()</code> and <code>min()</code> can be used.

10 Disclaimer and roadmap

The package may break down if the escape character is among `0123456789_+`; 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.

- Decide what exponent range to consider.
- Support signalling `nan`.

- Modulo and remainder, and rounding functions `quantize`, `quantize0`, `quantize+`, `quantize-`, `quantize=`, `round=`. Should the modulo also be provided as (catcode 12) `%`?
- `\fp_format:nn` $\{\langle fpepr \rangle\}$ $\{\langle format \rangle\}$, but what should $\langle format \rangle$ 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 `\fp_if_nan:nTF`, and an `isnan` function?
- Support keyword arguments?

`Pgfmth` also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs. (Exclamation points mark important bugs.)

- 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.
- The overflow trap receives the wrong argument in `l3fp-expo` (see `exp(1e5678)` in `m3fp-traps001`).

Possible optimizations/improvements.

- Document that `l3trial/l3fp-types` introduces tools for adding new types.

- In subsection 9.1, write a grammar.
- Fix the `TWO BARS` business with the index.
- 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)? Perhaps for including comments inside the computation itself??

Part XXIII

The l3sort 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: }
}
```

will result 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 will yield a trivial sort: the final order is identical to the original order. Conversely, using a *comparison code* consisting only of `\sort_return_swapped:` will reverse the list (in a fairly inefficient way).

T_EXhackers 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 `l3sort` 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.

Part XXIV

The l3candidates package

Experimental additions to l3kernel

1 Important notice

This module provides a space in which functions can be added to l3kernel (expl3) while still being experimental.

As such, the functions here may not remain in their current form, or indeed at all, in l3kernel in the future.

In contrast to the material in l3experimental, 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 l3basics

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

New: 2014-08-22
Updated: 2015-08-03

`\cs_log:N` $\langle control\ sequence \rangle$

Writes the definition of the $\langle control\ sequence \rangle$ in the log file. See also `\cs_show:N` which displays the result in the terminal.

`__kernel_register_log:N`
`__kernel_register_log:c`

Updated: 2015-08-03

`__kernel_register_log:N` $\langle register \rangle$

Used to write the contents of a TeX register to the log file in a form similar to `__kernel_register_show:N`.

3 Additions to l3box

3.1 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in T_EX 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.

<code>\box_resize:Nnn</code>	<code>\box_resize:Nnn <box> {<x-size>} {<y-size>}</code>
<code>\box_resize:cnn</code>	

Resize the $\langle box \rangle$ to $\langle x-size \rangle$ horizontally and $\langle y-size \rangle$ vertically (both of the sizes are dimension expressions). The $\langle y-size \rangle$ is the vertical size (height plus depth) of the box. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus negative y -sizes will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current T_EX group level.

<code>\box_resize_to_ht_plus_dp:Nn</code>	<code>\box_resize_to_ht_plus_dp:Nn <box> {<y-size>}</code>
<code>\box_resize_to_ht_plus_dp:cn</code>	

Resize the $\langle box \rangle$ to $\langle y-size \rangle$ vertically, scaling the horizontal size by the same amount ($\langle y-size \rangle$ is a dimension expression). The $\langle y-size \rangle$ is the vertical size (height plus depth) of the box. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative size will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus negative y -sizes will result in a box with depth dependent on the height of the original box and height dependent on the depth of the original. The resizing applies within the current T_EX group level.

<code>\box_resize_to_ht:Nn</code>	<code>\box_resize_to_ht:Nn <box> {<y-size>}</code>
<code>\box_resize_to_ht:cn</code>	

Resize the $\langle box \rangle$ to $\langle y-size \rangle$ vertically, scaling the horizontal size by the same amount ($\langle y-size \rangle$ is a dimension expression). The $\langle y-size \rangle$ is the height only, not including depth, of the box. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative size will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus negative y -sizes will result in a box with depth dependent on the height of the original box and height dependent on the depth of the original. The resizing applies within the current T_EX group level.

<code>\box_resize_to_wd:Nn</code>	<code>\box_resize_to_wd:Nn <box> {<x-size>}</code>
<code>\box_resize_to_wd:cn</code>	

Resize the $\langle box \rangle$ to $\langle x-size \rangle$ horizontally, scaling the vertical size by the same amount ($\langle x-size \rangle$ is a dimension expression). The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative size will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus negative y -sizes will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current T_EX group level.

<code>\box_resize_to_wd_and_ht:Nnn</code>	<code>\box_resize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}</code>
<code>\box_resize_to_wd_and_ht:cnn</code>	

New: 2014-07-03

Resize the $\langle box \rangle$ to a *height* of $\langle x-size \rangle$ horizontally and $\langle y-size \rangle$ vertically (both of the sizes are dimension expressions). The $\langle y-size \rangle$ is the *height* of the box, ignoring any depth. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged.

<code>\box_rotate:Nn</code>	<code>\box_rotate:Nn <box> {<angle>}</code>
<code>\box_rotate:cnn</code>	

Rotates the $\langle box \rangle$ by $\langle angle \rangle$ (in degrees) anti-clockwise about its reference point. The reference point of the updated box will be moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the rotation is applied. The rotation applies within the current T_EX group level.

<code>\box_scale:Nnn</code>	<code>\box_scale:Nnn <box> {<x-scale>} {<y-scale>}</code>
<code>\box_scale:cnn</code>	

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 $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the scaling is applied. Negative scalings will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. Thus negative y -scales will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current T_EX group level.

3.2 Viewing part of a box

<code>\box_clip:N</code>	<code>\box_clip:N <box></code>
<code>\box_clip:c</code>	

Clips the $\langle box \rangle$ in the output so that only material inside the bounding box is displayed in the output. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the clipping is applied. The clipping applies within the current T_EX group level.

These functions require the L^AT_EX3 native drivers: they will not work with the L^AT_EX 2_ε graphics drivers!

T_EXhackers 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.

<code>\box_trim:Nnnnn</code>	<code>\box_trim:Nnnnn <box> {<left>} {<bottom>} {<right>} {<top>}</code>
<code>\box_trim:cnnnn</code>	

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 $\langle dimension expressions \rangle$. Material output of the bounding box will still be displayed in the output unless `\box_clip:N` is subsequently applied. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the trim operation is applied. The adjustment applies within the current T_EX group level. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

<hr/> <code>\box_viewport:Nnnnn</code> <hr/>	<code>\box_viewport:Nnnnn <box> {\llx} {\lly} {\urx} {\ury}</code>
<code>\box_viewport:cnnnn</code> <hr/>	Adjusts the bounding box of the <code><box></code> such that it has lower-left co-ordinates (<code><llx></code> , <code><lly></code>) and upper-right co-ordinates (<code><urx></code> , <code><ury></code>). All four co-ordinate positions are <i><dimension expressions></i> . Material output of the bounding box will still be displayed in the output unless <code>\box_clip:N</code> is subsequently applied. The updated <code><box></code> will be an hbox, irrespective of the nature of the <code><box></code> before the viewport operation is applied. The adjustment applies within the current T _E X group level.

4 Additions to l3clist

<hr/> <code>\clist_log:N</code> <hr/>	<code>\clist_log:N <comma list></code>
<code>\clist_log:c</code> <hr/>	Writes the entries in the <code><comma list></code> in the log file. See also <code>\clist_show:N</code> which displays the result in the terminal.
<hr/> <code>\clist_log:n</code> <hr/>	<code>\clist_log:n {\tokens}</code>
<code>\clist_log:n</code> <hr/>	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.
<hr/> <code>\clist_rand_item:N</code> ★ <hr/>	<code>\clist_rand_item:N <clist var></code>
<code>\clist_rand_item:c</code> ★ <hr/>	<code>\clist_rand_item:n {\<comma list>}</code>
<code>\clist_rand_item:n</code> ★ <hr/>	Selects a pseudo-random item of the <code><comma list></code> . If the <code><comma list></code> has no item, the result is empty. This is only available in pdfT _E X and LuaT _E X.
<hr/> <div>New: 2016-12-06</div> <hr/>	

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

5 Additions to l3coffins

<hr/> <code>\coffin_resize:Nnn</code> <hr/>	<code>\coffin_resize:Nnn <coffin> {\width} {\total-height}</code>
<code>\coffin_resize:cnn</code> <hr/>	Resized the <code><coffin></code> to <code><width></code> and <code><total-height></code> , both of which should be given as dimension expressions.
<hr/> <code>\coffin_rotate:Nn</code> <hr/>	<code>\coffin_rotate:Nn <coffin> {\angle}</code>
<code>\coffin_rotate:cn</code> <hr/>	Rotates the <code><coffin></code> by the given <code><angle></code> (given in degrees counter-clockwise). This process will rotate both the coffin content and poles. Multiple rotations will not result in the bounding box of the coffin growing unnecessarily.
<hr/> <code>\coffin_scale:Nnn</code> <hr/>	<code>\coffin_scale:Nnn <coffin> {\x-scale} {\y-scale}</code>
<code>\coffin_scale:cnn</code> <hr/>	Scales the <code><coffin></code> by a factors <code><x-scale></code> and <code><y-scale></code> in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

<code>\coffin_log_structure:N</code>
<code>\coffin_log_structure:c</code>
New: 2014-08-22

`\coffin_log_structure:N` $\langle coffin \rangle$

This function writes the structural information about the $\langle coffin \rangle$ in the log file. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin. See also `\coffin_show_structure:N` which displays the result in the terminal.

6 Additions to l3file

<code>\file_if_exist_input:nTF</code>
New: 2014-07-02

`\file_if_exist_input:n` $\{\langle file\ name \rangle\}$
`\file_if_exist_input:nTF` $\{\langle file\ name \rangle\} \{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$

Searches for $\langle file\ name \rangle$ using the current T_EX search path and the additional paths controlled by `\file_path_include:n`. If found, inserts the $\langle true\ code \rangle$ then reads in the file as additional L^AT_EX source as described for `\file_input:n`. Note that `\file_if_exist_input:n` does not raise an error if the file is not found, in contrast to `\file_input:n`.

<code>\ior_map_inline:Nn</code>
New: 2012-02-11

`\ior_map_inline:Nn` $\langle stream \rangle \{\langle inline\ function \rangle\}$

Applies the $\langle inline\ function \rangle$ to $\langle lines \rangle$ obtained by reading one or more lines (until an equal number of left and right braces are found) from the $\langle stream \rangle$. The $\langle inline\ function \rangle$ should consist of code which will receive the $\langle line \rangle$ as #1. Note that T_EX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T_EX also ignores any trailing new-line marker from the file it reads.

<code>\ior_str_map_inline:Nn</code>
New: 2012-02-11

`\ior_str_map_inline:Nn` $\{\langle stream \rangle\} \{\langle inline\ function \rangle\}$

Applies the $\langle inline\ function \rangle$ to every $\langle line \rangle$ in the $\langle stream \rangle$. The material is read from the $\langle stream \rangle$ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The $\langle inline\ function \rangle$ should consist of code which will receive the $\langle line \rangle$ as #1. Note that T_EX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T_EX also ignores any trailing new-line marker from the file it reads.

\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 will normally take 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 will lead to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before further items are taken from the input stream. This will depend on the design of the mapping function.

\ior_map_break:n

New: 2012-06-29

\ior_map_break:n {*⟨tokens⟩*}

Used to terminate a `\ior_map...` function before all lines in the *⟨stream⟩* have been processed, inserting the *⟨tokens⟩* after the mapping has ended. This will normally take place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \ior_map_break:n { <tokens> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\ior_map...` scenario will lead to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before the *⟨tokens⟩* are inserted into the input stream. This will depend on the design of the mapping function.

\ior_log_streams:**\iow_log_streams:**

New: 2014-08-22

\ior_log_streams:**\iow_log_streams:**

Writes in the log file a list of the file names associated with each open stream: intended for tracking down problems.

7 Additions to l3fp

<code>\fp_log:N</code>	<code>\fp_log:N <fp var></code>
<code>\fp_log:c</code>	<code>\fp_log:n {<floating point expression>}</code>
<code>\fp_log:n</code>	Evaluates the <i><floating point expression></i> and writes the result in the log file.
New: 2014-08-22	
Updated: 2015-08-07	

8 Additions to l3int

<code>\int_log:N</code>	<code>\int_log:N <integer></code>
<code>\int_log:c</code>	Writes the value of the <i><integer></i> in the log file.
New: 2014-08-22	
Updated: 2015-08-03	

<code>\int_log:n</code>	<code>\int_log:n {<integer expression>}</code>
New: 2014-08-22	
Updated: 2015-08-07	

<code>\int_rand:nn</code> ★	<code>\int_rand:nn {<intexpr₁>} {<intexpr₂>}</code>
New: 2016-12-06	
Evaluates the two <i><integer expressions></i> and produces a pseudo-random number between the two (with bounds included). This is only available in pdfTeX and LuaTeX.	

9 Additions to l3keys

<code>\keys_log:nn</code>	<code>\keys_log:nn {<module>} {<key>}</code>
New: 2014-08-22	
Writes in the log file the function which is used to actually implement a <i><key></i> for a <i><module></i> .	

10 Additions to l3msg

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, the message text and arguments are not expanded, 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. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

```

\msg_expandable_error:nnnnnn ★ \msg_expandable_error:nnnnnn {<module>} {<message>} {<arg one>} {<arg
\msg_expandable_error:nnffff ★ two>} {<arg three>} {<arg four>}
\msg_expandable_error:nnnnnn ★
\msg_expandable_error:nnffff ★
\msg_expandable_error:nnnnn ★
\msg_expandable_error:nnff ★
\msg_expandable_error:nnn ★
\msg_expandable_error:nnf ★
\msg_expandable_error:nn ★

```

New: 2015-08-06

Issues an “Undefined error” message from T_EX 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.

11 Additions to l3prg

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 } } % is skipped
  }
}
{ ! \int_compare_p:n { 2 = 4 } }

```

the line marked with `is 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_lazy_all_p:n ★ \bool_lazy_all_p:n { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} }
\bool_lazy_all:nTF ★ \bool_lazy_all:nTF { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} } {<true code>}
\bool_lazy_all:nTF {<false code>}

```

New: 2015-11-15

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` will be evaluated. See also `\bool_lazy_and:nnTF` when there are only two <boolean expressions>.

<code>\bool_lazy_and_p:nn</code> ★	<code>\bool_lazy_and_p:nn {<boolexpr₁>} {<boolexpr₂>}</code>
<code>\bool_lazy_and:nnTF</code> ★	<code>\bool_lazy_and:nnTF {<boolexpr₁>} {<boolexpr₂>} {<true code>} {<false code>}</code>
New: 2015-11-15	
<p>Implements the “And” operation between two boolean expressions, hence is true if both are true. Contrarily to the infix operator <code>&&</code>, the <code><boolexpr₂></code> will only be evaluated if it is needed to determine the result of <code>\bool_lazy_and:nnTF</code>. See also <code>\bool_lazy_all:nnTF</code> when there are more than two <code><boolean expressions></code>.</p>	

<code>\bool_lazy_any_p:n</code> ★	<code>\bool_lazy_any_p:n { {<boolexpr₁>} {<boolexpr₂>} ... {<boolexpr_N>} }</code>
<code>\bool_lazy_any:nnTF</code> ★	<code>\bool_lazy_any:nnTF { {<boolexpr₁>} {<boolexpr₂>} ... {<boolexpr_N>} } {<true code>} {<false code>}</code>
New: 2015-11-15	
<p>Implements the “Or” operation on the <code><boolean expressions></code>, hence is true if any of them is true and false if all of them are false. Contrarily to the infix operator <code> </code>, only the <code><boolean expressions></code> which are needed to determine the result of <code>\bool_lazy_any:nnTF</code> will be evaluated. See also <code>\bool_lazy_or:nnTF</code> when there are only two <code><boolean expressions></code>.</p>	

<code>\bool_lazy_or_p:nn</code> ★	<code>\bool_lazy_or_p:nn {<boolexpr₁>} {<boolexpr₂>}</code>
<code>\bool_lazy_or:nnTF</code> ★	<code>\bool_lazy_or:nnTF {<boolexpr₁>} {<boolexpr₂>} {<true code>} {<false code>}</code>
New: 2015-11-15	
<p>Implements the “Or” operation between two boolean expressions, hence is true if either one is true. Contrarily to the infix operator <code> </code>, the <code><boolexpr₂></code> will only be evaluated if it is needed to determine the result of <code>\bool_lazy_or:nnTF</code>. See also <code>\bool_lazy_any:nnTF</code> when there are more than two <code><boolean expressions></code>.</p>	

<code>\bool_log:N</code>	<code>\bool_log:N <boolean></code>
<code>\bool_log:c</code>	Writes the logical truth of the <code><boolean></code> in the log file.
New: 2014-08-22	
Updated: 2015-08-03	

<code>\bool_log:n</code>	<code>\bool_log:n {<boolean expression>}</code>
New: 2014-08-22	
Updated: 2015-08-07	
Writes the logical truth of the <code><boolean expression></code> in the log file.	

12 Additions to l3prop

<code>\prop_count:N</code> ★	<code>\prop_count:N <property list></code>
<code>\prop_count:c</code> ★	Leaves the number of key–value pairs in the <code><property list></code> in the input stream as an <code><integer denotation></code> .

<code>\prop_map_tokens:Nn</code> ☆	<code>\prop_map_tokens:Nn <property list> {<code>}</code>
<code>\prop_map_tokens:cn</code> ☆	Analogue of <code>\prop_map_function:NN</code> which maps several tokens instead of a single function. The <code><code></code> receives each key–value pair in the <code><property list></code> as two trailing brace groups. For instance,

`\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }`

will expand 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.

<code>\prop_log:N</code>	<code>\prop_log:N</code> $\langle property list \rangle$
<code>\prop_log:c</code>	Writes the entries in the $\langle property list \rangle$ in the log file.
New: 2014-08-12	

<code>\prop_rand_key_value:N</code> ☆	<code>\prop_rand_key_value:N</code> $\langle prop var \rangle$
<code>\prop_rand_key_value:c</code> ☆	Selects a pseudo-random key–value pair in the $\langle property list \rangle$ and returns $\{\langle key \rangle\}\{\langle value \rangle\}$. If the $\langle property list \rangle$ is empty the result is empty. This is only available in pdfTeX and LuaTeX.
New: 2016-12-06	

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

13 Additions to l3seq

<code>\seq_mapthread_function:NNN</code> ☆	<code>\seq_mapthread_function:NNN</code> $\langle seq_1 \rangle$ $\langle seq_2 \rangle$ $\langle function \rangle$
<code>\seq_mapthread_function:(NcN cNN ccN)</code> ☆	

Applies $\langle function \rangle$ to every pair of items $\langle seq_1-item \rangle$ – $\langle seq_2-item \rangle$ from the two sequences, returning items from both sequences from left to right. The $\langle function \rangle$ will receive two n-type arguments for each iteration. The mapping will terminate when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

<code>\seq_set_filter:NNn</code>	<code>\seq_set_filter:NNn</code> $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{\langle inline boolexpr \rangle\}$
<code>\seq_gset_filter:NNn</code>	Evaluates the $\langle inline boolexpr \rangle$ for every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline boolexpr \rangle$ will receive the $\langle item \rangle$ as #1. The sequence of all $\langle items \rangle$ for which the $\langle inline boolexpr \rangle$ evaluated to <code>true</code> is assigned to $\langle sequence_1 \rangle$.

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level TeX errors.

<code>\seq_set_map:NNn</code>	<code>\seq_set_map:NNn</code> $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{\langle inline function \rangle\}$
<code>\seq_gset_map:NNn</code>	Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. The sequence resulting from x-expanding $\langle inline function \rangle$ applied to each $\langle item \rangle$ is assigned to $\langle sequence_1 \rangle$. As such, the code in $\langle inline function \rangle$ should be expandable.
New: 2011-12-22	

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level TeX errors.

<code>\seq_log:N</code>	<code>\seq_log:N</code> $\langle sequence \rangle$
<code>\seq_log:c</code>	Writes the entries in the $\langle sequence \rangle$ in the log file.
New: 2014-08-12	

<hr/> <code>\seq_rand_item:N</code> ★	<code>\seq_rand_item:N</code> $\langle seq\ var\rangle$
<code>\seq_rand_item:c</code> ★	
<hr/> New: 2016-12-06 <hr/>	
	Selects a pseudo-random item of the $\langle sequence\rangle$. If the $\langle sequence\rangle$ is empty the result is empty. This is only available in pdfTeX and LuaTeX.
	TeXhackers note: The result is returned within the <code>\unexpanded</code> primitive (<code>\exp_not:n</code>), which means that the $\langle item\rangle$ will not expand further when appearing in an x-type argument expansion.

14 Additions to l3skip

<hr/> <code>\skip_split_finite_else_action:nnNN</code>	<code>\skip_split_finite_else_action:nnNN</code> $\{\langle skipexpr\rangle\}$ $\{\langle action\rangle\}$ $\langle dimen_1\rangle$ $\langle dimen_2\rangle$
	Checks if the $\langle skipexpr\rangle$ contains finite glue. If it does then it assigns $\langle dimen_1\rangle$ the stretch component and $\langle dimen_2\rangle$ the shrink component. If it contains infinite glue set $\langle dimen_1\rangle$ and $\langle dimen_2\rangle$ to 0pt and place #2 into the input stream: this is usually an error or warning message of some sort.
<hr/> <code>\dim_log:N</code>	<code>\dim_log:N</code> $\langle dimension\rangle$
<code>\dim_log:c</code>	
<hr/> New: 2014-08-22 <hr/> Updated: 2015-08-03 <hr/>	Writes the value of the $\langle dimension\rangle$ in the log file.
<hr/> <code>\dim_log:n</code>	<code>\dim_log:n</code> $\{\langle dimension\ expression\rangle\}$
<hr/> New: 2014-08-22 <hr/> Updated: 2015-08-07 <hr/>	Writes the result of evaluating the $\langle dimension\ expression\rangle$ in the log file.
<hr/> <code>\skip_log:N</code>	<code>\skip_log:N</code> $\langle skip\rangle$
<code>\skip_log:c</code>	
<hr/> New: 2014-08-22 <hr/> Updated: 2015-08-03 <hr/>	Writes the value of the $\langle skip\rangle$ in the log file.
<hr/> <code>\skip_log:n</code>	<code>\skip_log:n</code> $\{\langle skip\ expression\rangle\}$
<hr/> New: 2014-08-22 <hr/> Updated: 2015-08-07 <hr/>	Writes the result of evaluating the $\langle skip\ expression\rangle$ in the log file.
<hr/> <code>\muskip_log:N</code>	<code>\muskip_log:N</code> $\langle muskip\rangle$
<code>\muskip_log:c</code>	
<hr/> New: 2014-08-22 <hr/> Updated: 2015-08-03 <hr/>	Writes the value of the $\langle muskip\rangle$ in the log file.
<hr/> <code>\muskip_log:n</code>	<code>\muskip_log:n</code> $\{\langle muskip\ expression\rangle\}$
<hr/> New: 2014-08-22 <hr/> Updated: 2015-08-07 <hr/>	Writes the result of evaluating the $\langle muskip\ expression\rangle$ in the log file.

15 Additions to l3tl

<code>\tl_if_single_token_p:n</code> ★ <code>\tl_if_single_token:nTF</code> ★	<code>\tl_if_single_token_p:n {⟨token list⟩}</code> <code>\tl_if_single_token:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>
--	---

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.

<code>\tl_reverse_tokens:n</code> ★	<code>\tl_reverse_tokens:n {⟨tokens⟩}</code>
-------------------------------------	--

This function, which works directly on T_EX tokens, reverses the order of the *⟨tokens⟩*: the first will be the last and the last will become first. Spaces are preserved. The reversal also operates within brace groups, but the braces themselves are not exchanged, as this would lead to an unbalanced token list. For instance, `\tl_reverse_tokens:n {a~{b()}}` leaves `{() (b)~a` in the input stream. This function requires two steps of expansion.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the token list will not expand further when appearing in an x-type argument expansion.

<code>\tl_count_tokens:n</code> ★	<code>\tl_count_tokens:n {⟨tokens⟩}</code>
-----------------------------------	--

Counts the number of T_EX 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. This function requires three expansions, giving an *⟨integer denotation⟩*.

<code>\tl_lower_case:n</code> ★ <code>\tl_upper_case:n</code> ★ <code>\tl_mixed_case:n</code> ★ <code>\tl_lower_case:nn</code> ★ <code>\tl_upper_case:nn</code> ★ <code>\tl_mixed_case:nn</code> ★	<code>\tl_upper_case:n {⟨tokens⟩}</code> <code>\tl_upper_case:nn {⟨language⟩} {⟨tokens⟩}</code> These functions are intended to be applied to input which may be regarded broadly as “text”. They traverse the <i>⟨tokens⟩</i> and change the case of characters as discussed below. The character code of the characters replaced may be arbitrary: the replacement characters will have standard document-level category codes (11 for letters, 12 for letter-like characters which can also be case-changed). Begin-group and end-group characters in the <i>⟨tokens⟩</i> are normalized and become <code>{</code> and <code>}</code> , respectively.
---	--

New: 2014-06-30
Updated: 2016-01-12

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_lower_case:n`, `\str_upper_case:n` and `\str_fold_case:n`.

The functions perform expansion on the input in most cases. In particular, input in the form of token lists or expandable functions will be expanded *unless* it falls within one of the special handling classes described below. This expansion approach means that in general the result of case changing will match the “natural” outcome expected from a “functional” approach to case modification. For example

```
\tl_set:Nn \l_tmpa_tl { hello }
\tl_upper_case:n { \l_tmpa_tl \c_space_tl world }
```

will produce

```
HELLO WORLD
```

The expansion approach taken means that in package mode any L^AT_EX 2_ε “robust” commands which may appear in the input should be converted to engine-protected versions using for example the `\robustify` command from the `etoolbox` package.

`\l_tl_case_change_math_tl`

Case changing will not take place within math mode material so for example

```
\tl_upper_case:n { Some~text~$y = mx + c$~with~{Braces} }
```

will become

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

Material inside math mode is left entirely unchanged: in particular, no expansion is undertaken.

Detection of math mode is controlled by the list of tokens in `\l_tl_case_change_math_tl`, which should be in open–close pairs. In package mode the standard settings is

```
$ $ \ ( \)
```

Note that while expansion occurs when searching the text it does not apply to math mode material (which should be unaffected by case changing). As such, whilst the opening token for math mode may be “hidden” inside a command/macro, the closing one cannot be as this is being searched for in math mode. Typically, in the types of “text” the case changing functions are intended to apply to this should not be an issue.

`\l_tl_case_change_exclude_tl`

Case changing can be prevented by using any command on the list `\l_tl_case_change_exclude_tl`. Each entry should be a function to be followed by one argument: the latter will be preserved as-is with no expansion. Thus for example following

```
\tl_put_right:Nn \l_tl_case_change_exclude_tl { \NoChangeCase }
```

the input

```
\tl_upper_case:n
{ Some~text~$y = mx + c$~with~\NoChangeCase {Protection} }
```

will result in

```
SOME TEXT $y = mx + c$ WITH \NoChangeCase {Protection}
```

Notice that the case changing mapping preserves the inclusion of the escape functions: it is left to other code to provide suitable definitions (typically equivalent to `\use:n`). In particular, the result of case changing is returned protected by `\exp_not:n`.

When used with L^AT_EX 2_ε the commands `\cite`, `\ensuremath`, `\label` and `\ref` are automatically included in the list for exclusion from case changing.

`\l_tl_case_change_accents_tl`

This list specifies accent commands which should be left unexpanded in the output. This allows for example

```
\tl_upper_case:n { \" { a } }
```

to yield

```
\" { A }
```

irrespective of the expandability of `\"`.

The standard contents of this variable is `\", \', \., \^, \', \~, \c, \H, \k, \r, \t, \u` and `\v`.

“Mixed” case conversion may be regarded informally as converting the first character of the *<tokens>* to upper case and the rest to lower case. However, the process is more complex than this as there are some situations where a single lower case character maps to a special form, for example *ij* in Dutch which becomes *IJ*. As such, `\tl_mixed_case:n(n)` implement a more sophisticated mapping which accounts for this and for modifying accents on the first letter. Spaces at the start of the *<tokens>* are ignored when finding the first “letter” for conversion.

```
\tl_mixed_case:n { hello~WORLD } % => "Hello world"
\tl_mixed_case:n { ~hello~WORLD } % => " Hello world"
\tl_mixed_case:n { {hello}~WORLD } % => "{Hello} world"
```

When finding the first “letter” for this process, any content in math mode or covered by `\l_tl_case_change_exclude_tl` is ignored.

(Note that the Unicode Consortium describe this as “title case”, but that in English title case applies on a word-by-word basis. The “mixed” case implemented here is a lower level concept needed for both “title” and “sentence” casing of text.)

`\l_tl_mixed_case_ignore_tl`

The list of characters to ignore when searching for the first “letter” in mixed-casing is determined by `\l_tl_mixed_change_ignore_tl`. This has the standard setting

```
( [ { ' -
```

where comparisons are made on a character basis.

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 `XƎTeX` or `LuaTeX` 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` font encoding. Thus for example `Ād` can be case-changed using `pdfTeX`. For `pTeX` only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Context-sensitive mappings are enabled: language-dependent cases are discussed below. Context detection will expand input but treats any unexpandable control sequences as “failures” to match a context.

Language-sensitive conversions are enabled using the *<language>* 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 lower casing I-dot and introduced when upper casing i-dotless.
- German (**de-alt**). An alternative mapping for German in which the lower case *Eszett* maps to a *großes Eszett*.
- Lithuanian (**lt**). The lower case letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lower casing of the relevant upper case letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when upper casing 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 mixed cased input produces IJ rather than Ij. The output retains two separate letters, thus this transformation *is* available using pdfTeX.

Creating additional context-sensitive mappings requires knowledge of the underlying mapping implementation used here. The team are happy to add these to the kernel where they are well-documented (*e.g.* in Unicode Consortium or relevant government publications).

```
\tl_set_from_file:Nnn
\tl_set_from_file:cnn
\tl_gset_from_file:Nnn
\tl_gset_from_file:cnn
```

New: 2014-06-25

```
\tl_set_from_file:Nnn <tl> {<setup>} {<filename>}
```

Defines $\langle tl \rangle$ to the contents of $\langle filename \rangle$. Category codes may need to be set appropriately via the $\langle setup \rangle$ argument.

```
\tl_set_from_file_x:Nnn
\tl_set_from_file_x:cnn
\tl_gset_from_file_x:Nnn
\tl_gset_from_file_x:cnn
```

New: 2014-06-25

```
\tl_set_from_file_x:Nnn <tl> {<setup>} {<filename>}
```

Defines $\langle tl \rangle$ to the contents of $\langle filename \rangle$, expanding the contents of the file as it is read. Category codes and other definitions may need to be set appropriately via the $\langle setup \rangle$ argument.

```
\tl_log:N
\tl_log:c
```

New: 2014-08-22

Updated: 2015-08-01

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

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

```
\tl_log:n
```

New: 2014-08-22

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

Writes the $\langle token list \rangle$ in the log file. See also $\backslash tl_show:n$ which displays the result in the terminal.

<code>\tl_rand_item:N</code> ★	<code>\tl_rand_item:N</code> $\langle tl\ var \rangle$
<code>\tl_rand_item:c</code> ★	<code>\tl_rand_item:n</code> $\{\langle token\ list \rangle\}$
<code>\tl_rand_item:n</code> ★	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 only available in pdfTeX and LuaTeX.

New: 2016-12-06

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

16 Additions to l3tokens

<code>\peek_N_type:TF</code> <u>TF</u>	<code>\peek_N_type:TF</code> $\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
--	---

Updated: 2012-12-20

Tests if the next $\langle token \rangle$ in the input stream can be safely grabbed as an N-type argument. The test will be $\langle false \rangle$ 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 will take the $\langle false \rangle$ branch, even though the next $\langle token \rangle$ is in fact a valid N-type argument. The $\langle token \rangle$ will be left in the input stream after the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ (as appropriate to the result of the test).

Part XXV

The l3sys package

System/runtime functions

1 The name of the job

`\c_sys_jobname_str`

New: 2015-09-19

Constant that gets the “job name” assigned when T_EX starts.

T_EXhackers note: This copies the contents of the primitive `\jobname`. It is a constant that is set by T_EX and should not be overwritten by the package.

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.

T_EXhackers 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.

2.1 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

`\c_sys_engine_str`

New: 2015-09-19

`\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 -pT_EX and ε -upT_EX as expl3 requires the ε -T_EX extensions. Each conditional is true for *exactly one* supported engine. In particular, `\sys_if_engine ptex_p:` is true for ε -pT_EX but false for ε -upT_EX.

The current engine given as a lower case string: will be one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

2.2 Output format

<code>\sys_if_output_dvi_p:</code>	★	<code>\sys_if_output_dvi:TF</code>	<code>{\true code}}</code>	<code>{\false code}}</code>
------------------------------------	---	------------------------------------	----------------------------	-----------------------------

<code>\sys_if_output_dvi:</code>	★	<code>TF</code>
----------------------------------	---	-----------------

<code>\sys_if_output_pdf_p:</code>	★	
------------------------------------	---	--

<code>\sys_if_output_pdf:</code>	★	<code>TF</code>
----------------------------------	---	-----------------

New: 2015-09-19

Conditionals which give the current output mode the \TeX run is operating in. This will always be 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.

<code>\c_sys_output_str</code>

New: 2015-09-19

The current output mode given as a lower case string: will be one of `dvi` or `pdf`.

Part XXVI

The l3`luatex` package

LuaTeX-specific functions

1 Breaking out to Lua

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 or XeTeX these will raise an error: use `\sys_if_engine luatex:T` to avoid this. Details of coding the LuaTeX engine are detailed in the LuaTeX manual.

1.1 TeX code interfaces

<code>\lua_now_x:n</code>	★	<code>\lua_now:n {⟨token list⟩}</code>
---------------------------	---	--

<code>\lua_now:n</code>	★
-------------------------	---

New: 2015-06-29

The *⟨token list⟩* is first tokenized by TeX, which will include 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 will execute the *⟨Lua input⟩* immediately, and in an expandable manner.

In the case of the `\lua_now_x:n` version the input is fully expanded by TeX in an x-type manner *but* the function remains fully expandable.

TeXhackers note: `\lua_now_x:n` is a macro wrapper around `\directlua:` when LuaTeX is in use two expansions will be required to yield the result of the Lua code.

<code>\lua_shipout_x:n</code>		<code>\lua_shipout:n {⟨token list⟩}</code>
-------------------------------	--	--

<code>\lua_shipout:n</code>	
-----------------------------	--

New: 2015-06-30

The *⟨token list⟩* is first tokenized by TeX, which will include 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_x:n` version the input is fully expanded by TeX in an x-type manner during the shipout operation.

TeXhackers note: At a TeX level, the *⟨Lua input⟩* is stored as a “whatsit”.

<hr/>	<code>\lua_escape_x:n</code> ★	<code>\lua_escape:n {⟨token list⟩}</code>
<hr/>	<code>\lua_escape:n</code> ★	Converts the <i>⟨token list⟩</i> 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 <code>\n</code> and <code>\r</code> , respectively.
<hr/>	New: 2015-06-29	In the case of the <code>\lua_escape_x:n</code> version the input is fully expanded by $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ in an <i>x</i> -type manner <i>but</i> the function remains fully expandable.
		$\mathrm{T}_{\mathrm{E}}\mathrm{X}$hackers note: <code>\lua_escape_x:n</code> is a macro wrapper around <code>\luaescapestring</code> : when $\mathrm{LuaT}_{\mathrm{E}}\mathrm{X}$ is in use two expansions will be required to yield the result of the Lua code.

1.2 Lua interfaces

As well as interfaces for $\mathrm{T}_{\mathrm{E}}\mathrm{X}$, there are a small number of Lua functions provided here. Currently these are intended for internal use only.

<hr/>	<code>l3kernel.strptime</code>	<code>\l3kernel.strptime(⟨str one⟩, ⟨str two⟩)</code>
		Compares the two strings and returns 0 to $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ if the two are identical.
<hr/>	<code>l3kernel.charcat</code>	<code>\l3kernel.charcat(⟨charcode⟩, ⟨catcode⟩)</code>
		Constructs a character of <i>⟨charcode⟩</i> and <i>⟨catcode⟩</i> and returns the result to $\mathrm{T}_{\mathrm{E}}\mathrm{X}$.

Part XXVII

The l3drivers package

Drivers

T_EX relies on drivers in order to carry out a number of tasks, such as using color, including graphics and setting up hyper-links. The nature of the code required depends on the exact driver in use. Currently, L^AT_EX3 is aware of the following drivers:

- **pdfmode**: The “driver” for direct PDF output by *both* pdfT_EX and LuaT_EX (no separate driver is used in this case: the engine deals with PDF creation itself).
- **dvips**: The dvips program, which works in conjugation with pdfT_EX or LuaT_EX in DVI mode.
- **dvipdfmx**: The dvipdfmx program, which works in conjugation with pdfT_EX or LuaT_EX in DVI mode.
- **dvisvgm**: The dvisvgm program, which works in conjugation with pdfT_EX or LuaT_EX in DVI mode to create SVG output.
- **xdvipdfmx**: The driver used by X_YT_EX.

The code here is all very low-level, and should not in general be used outside of the kernel. It is also important to note that many of the functions here are closely tied to the immediate level “up”, and they must be used in the correct contexts.

1 Box clipping

`_driver_box_use_clip:N`

New: 2011-11-11

`_driver_box_use_clip:N <box>`

Inserts the content of the *<box>* at the current insertion point such that any material outside of the bounding box will not be displayed by the driver. The material in the *<box>* is still placed in the output stream: the clipping takes place at a driver level.

This function should only be used within a surrounding horizontal box construct.

2 Box rotation and scaling

`_driver_box_use_rotate:Nn`

New: 2016-05-12

`_driver_box_use_rotate:Nn <box> {<angle>}`

Inserts the content of the *<box>* at the current insertion point rotated by the *<angle>* (expressed in degrees). The material is inserted with no apparent height or width, and is rotated such the the T_EX reference point of the box is the center of rotation and remains the reference point after rotation. It is the responsibly of the code using this function to adjust the apparent size of the box to be correct at the T_EX side.

This function should only be used within a surrounding horizontal box construct.

<code>__driver_box_use_scale:Nnn</code>	<code>__driver_box_use_scale:Nnn <box> {<x-scale>} {<y-scale>}</code>
--	--

New: 2016-05-12

Inserts the content of the $\langle box \rangle$ at the current insertion point scale by the $\langle x-scale \rangle$ and $\langle y-scale \rangle$. The material is inserted with no apparent height or width. It is the responsibly of the code using this function to adjust the apparent size of the box to be correct at the \TeX side.

This function should only be used within a surrounding horizontal box construct.

3 Color support

<code>__driver_color_ensure_current:</code>	<code>__driver_color_ensure_current:</code>
--	--

New: 2011-09-03

Updated: 2012-05-18

Ensures that the color used to typeset material is that which was set when the material was placed in a box. This function is therefore required inside any “color safe” box to ensure that the box may be inserted in a location where the foreground color has been altered, while preserving the color used in the box.

4 Drawing

The drawing functions provided here are *highly* experimental. They are inspired heavily by the system layer of `pgf` (most have the same interface as the same functions in the latter’s `\pgfsys@... namespace`). They are intended to form the basis for higher level drawing interfaces, which themselves are likely to be further abstracted for user access. Again, this model is heavily inspired by `pgf` and `Tikz`.

These low level drawing interfaces abstract from the driver raw requirements but still require an appreciation of the concepts of PostScript/PDF/SVG graphic creation.

<code>__driver_draw_begin:</code>	<code>__driver_draw_begin:</code>
<code>__driver_draw_end:</code>	<code>__driver_draw_end:</code>

Defines a drawing environment. This will be a scope for the purposes of the graphics state. Depending on the driver, other set up may or may not take place here. The natural size of the $\langle content \rangle$ should be zero from the \TeX perspective: allowance for the size of the content must be made at a higher level (or indeed this can be skipped if the content is to overlap other material).

<code>__driver_draw_scope_begin:</code>	<code>__driver_draw_scope_begin:</code>
<code>__driver_draw_scope_end:</code>	<code>__driver_draw_scope_end:</code>

Defines a scope for drawing settings and so on. Changes to the graphic state and concepts such as color or linewidth are localised to a scope. This function pair must never be used if an partial path is under construction: such paths must be entirely contained at one unbroken scope level. Note that scopes do not form \TeX groups and may not be aligned with them.

4.1 Path construction

<u><code>__driver_draw_moveto:nn</code></u>	<code>__driver_draw_move:nn {<x>} {<y>}</code>	Moves the current drawing reference point to $(\langle x \rangle, \langle y \rangle)$; any active transformation matrix will apply.
<u><code>__driver_draw_lineto:nn</code></u>	<code>__driver_draw_lineto:nn {<x>} {<y>}</code>	Adds a path from the current drawing reference point to $(\langle x \rangle, \langle y \rangle)$; any active transformation matrix will apply. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.
<u><code>__driver_draw_curveto:nnnnnn</code></u>	<code>__driver_draw_curveto:nnnnnn {<x₁>} {<y₁>} {<x₂>} {<y₂>} {<x₃>} {<y₃>}</code>	Adds a Bezier curve path from the current drawing reference point to $(\langle x_3 \rangle, \langle y_3 \rangle)$, using $(\langle x_1 \rangle, \langle y_1 \rangle)$ and $(\langle x_2 \rangle, \langle y_2 \rangle)$ as control points; any active transformation matrix will apply. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.
<u><code>__driver_draw_rectangle:nnnn</code></u>	<code>__driver_draw_rectangle:nnnn {<x>} {<y>} {<width>} {<height>}</code>	Adds rectangular path from $(\langle x_1 \rangle, \langle y_1 \rangle)$ of $\langle height \rangle$ and $\langle width \rangle$; any active transformation matrix will apply. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.
<u><code>__driver_draw_closepath:</code></u>	<code>__driver_draw_closepath:</code>	Closes an existing path, adding a line from the current point to the start of path. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

4.2 Stroking and filling

<u><code>__driver_draw_stroke:</code></u>	<code><path construction></code>	
<u><code>__driver_draw_closestroke:</code></u>	<code>__driver_draw_stroke:</code>	
		Draws a line along the current path, which will also be closed when <code>__driver_draw_closestroke:</code> is used. The nature of the line drawn is influenced by settings for
		<ul style="list-style-type: none"> • Line thickness • Stroke color (or the current color if no specific stroke color is set) • Line capping (how non-closed line ends should look) • Join style (how a bend in the path should be rendered) • Dash pattern

The path may also be used for clipping.

<code>__driver_draw_fill:</code>	<code><path construction></code>
<code>__driver_draw_fillstroke:</code>	<code>__driver_draw_fill:</code>

Fills the area surrounded by the current path: this will be closed prior to filling if it is not already. The `fillstroke` version will also stroke the path as described for `__driver_draw_stroke:`. The fill is influenced by the setting for fill color (or the current color if no specific stroke color is set). The path may also be used for clipping. For paths which are self-intersecting or comprising multiple parts, the determination of which areas are inside the path is made using the non-zero winding number rule unless the even-odd rule is active.

<code>__driver_draw_nonzero_rule:</code>	<code>__driver_draw_nonzero_rule:</code>
<code>__driver_draw_evenodd_rule:</code>	

Active either the non-zero winding number or the even-odd rule, respectively, for determining what is inside a fill or clip area. For technical reasons, these command are not influenced by scoping and apply on an ongoing basis.

<code>__driver_draw_clip:</code>	<code><path construction></code>
	<code>__driver_draw_clip:</code>

Indicates that the current path should be used for clipping, such that any subsequent material outside of the path (but within the current scope) will not be shown. This command should be given once a path is complete but before it is stroked or filled (if appropriate). This command is *not* affected by scoping: it applies to exactly one path as shown.

<code>__driver_draw_discardpath:</code>	<code><path construction></code>
	<code>__driver_draw_discardpath:</code>

Discards the current path without stroking or filling. This is primarily useful for paths constructed purely for clipping, as this alone does not end the paths existence.

4.3 Stroke options

<code>__driver_draw_linewidth:n</code>	<code>__driver_draw_linewidth:n {<dimexpr>}</code>
---	---

Sets the width to be used for stroking to `<dimexpr>`.

<code>__driver_draw_dash:nn</code>	<code>__driver_draw_dash:nn {<dash pattern>} {<phase>}</code>
-------------------------------------	--

Sets the pattern of dashing to be used when stroking a line. The `<dash pattern>` should be a comma-separated list of dimension expressions. This is then interpreted as a series of pairs of line-on and line-off lengths. For example `3pt, 4pt` means that 3pt on, 4pt off, 3pt on, and so on. A more complex pattern will also repeat: `3pt, 4pt, 1pt, 2pt` results in 3pt on, 4pt off, 1pt on, 2pt off, 3pt on, and so on. An odd number of entries means that the last is repeated, for example `3pt` is equal to `3pt, 3pt`. An empty pattern yields a solid line.

The `<phase>` specifies an offset at the start of the cycle. For example, with a pattern `3pt` a phase of `1pt` will mean that the output is 2pt on, 3pt off, 3pt on, 3pt on, *etc.*

<code>_driver_draw_cap_butt:</code>	<code>_driver_draw_cap_butt:</code>
<code>_driver_draw_cap_rectangle:</code>	
<code>_driver_draw_cap_round:</code>	

Sets the style of terminal stroke position to one of butt, rectangle or round.

<code>_driver_draw_join_bevel:</code>	<code>_driver_draw_cap_butt:</code>
<code>_driver_draw_join_miter:</code>	Sets the style of stroke joins to one of bevel, miter or round.
<code>_driver_draw_join_round:</code>	

<code>_driver_draw_miterlimit:n</code>	<code>_driver_draw_miterlimit:n {<dimexpr>}</code>
---	---

Sets the miter limit of lines joined as a miter, as described in the PDF and PostScript manuals.

4.4 Color

<code>_driver_draw_color_cmyk:nnnn</code>	<code>_driver_draw_color_cmyk:nnnn {<cyan>} {<magenta>} {<yellow>}</code>
<code>_driver_draw_color_cmyk_fill:nnnn</code>	<code>{<black>}</code>
<code>_driver_draw_color_cmyk_stroke:nnnn</code>	

Sets the color for drawing to the CMYK values specified, all of which are fp expressions which should evaluate to between 0 and 1. The **fill** and **stroke** versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

<code>_driver_draw_color_gray:n</code>	<code>_driver_draw_color_gray:n {<gray>}</code>
<code>_driver_draw_color_gray_fill:n</code>	
<code>_driver_draw_color_gray_stroke:n</code>	

Sets the color for drawing to the grayscale value specified, which is fp expressions which should evaluate to between 0 and 1. The **fill** and **stroke** versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

<code>_driver_draw_color_rgb:nnn</code>	<code>_driver_draw_color_gray:n {<red>} {<green>} {<blue>}</code>
<code>_driver_draw_color_rgb_fill:nnn</code>	
<code>_driver_draw_color_rgb_stroke:nnn</code>	

Sets the color for drawing to the RGB values specified, all of which are fp expressions which should evaluate to between 0 and 1. The **fill** and **stroke** versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

4.5 Inserting T_EX material

```
\_driver_draw_hbox:nnnnnn
```

```
\_driver_draw_hbox:Nnnnnnn <box>
{\<a>} {\<b>} {\<c>} {\<d>} {\<x>} {\<y>}
```

Inserts the $\langle box \rangle$ as an hbox with the box reference point placed at (x, y) . The transformation matrix $[abcd]$ will be applied to the box, allowing it to be in synchronisation with any scaling, rotation or skewing applying more generally. Note that T_EX material should not be inserted directly into a drawing as it will not be in the correct location. Also note that as for other drawing elements the box here will have no size from a T_EX perspective.

4.6 Coordinate system transformations

```
\_driver_draw_transformcm:nnnnnn
```

```
\_driver_draw_transformcm:nnnnnn {\<a>} {\<b>} {\<c>} {\<d>}
{\<x>} {\<y>}
```

Applies the transformation matrix $[abcd]$ and offset vector (x, y) to the current graphic state. This will affect any subsequent items in the same scope but not those already given.

Part XXVIII

Implementation

1 l3bootstrap implementation

```
1 <*initex | package>
2 <@@=expl>
```

1.1 Format-specific code

The very first thing to do is to bootstrap the iniT_EX system so that everything else will actually work. T_EX does not start with some pretty basic character codes set up.

```
3 <*initex>
4 \catcode '\{ = 1 %
5 \catcode '\} = 2 %
6 \catcode '\# = 6 %
7 \catcode '\^ = 7 %
8 </initex>
```

Tab characters should not show up in the code, but to be on the safe side.

```
9 <*initex>
10 \catcode '\^^I = 10 %
11 </initex>
```

For LuaT_EX, the extra primitives need to be enabled. This is not needed in package mode: common formats have the primitives enabled.

```
12 <*initex>
13 \begingroup\expandafter\expandafter\expandafter\endgroup
14 \expandafter\ifx\csname directlua\endcsname\relax
15 \else
```



```

16 \directlua{tex.enableprimitives("", tex.extraprimitives())}%
17 \fi
18 </initex>

```

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.

```

19 <*package>
20 \begingroup
21 \expandafter\ifx\csname directlua\endcsname\relax
22 \else
23 \directlua{%
24     local i
25     local t = { }
26     for _,i in pairs(tex.extraprimitives("luatex")) do
27         if string.match(i,"^U") then
28             if not string.match(i,"^Uchar$") then
29                 table.insert(t,i)
30             end
31         end
32     end
33     tex.enableprimitives("", t)
34 }%
35 \fi
36 \endgroup
37 </package>

```

1.2 The `\pdfstrcmp` primitive in X_YT_EX

Only pdfT_EX has a primitive called `\pdfstrcmp`. The X_YT_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”.

```

38 \begingroup\expandafter\expandafter\expandafter\endgroup
39 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
40 \let\pdfstrcmp\strcmp
41 \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.

```

42 \begingroup\expandafter\expandafter\expandafter\endgroup
43 \expandafter\ifx\csname directlua\endcsname\relax
44 \else
45 \ifnum\luatexversion<70 %
46 \else

```

In package mode a category code table is needed: either use a pre-loaded allocator or provide one using the L^AT_EX 2_ε-based generic code. In format mode the table used here can be hard-coded into the Lua.

```

47 \begin{package}
48 \begin{group}\expandafter\expandafter\expandafter\endgroup
49 \expandafter\ifx\csname newcatcodetable\endcsname\relax
50 \input{ltluatex}%
51 \fi
52 \newcatcodetable\ucharcat@table
53 \directlua{
54   l3kernel = l3kernel or { }
55   local charcat_table = \number\ucharcat@table\space
56   l3kernel.charcat_table = charcat_table
57 }%
58 \end{package}
59 \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 will reveal what mode is in operation.

```

60 \ifnum 0%
61 \directlua{
62   if status.ini_version then
63     tex.write("1")
64   end
65 }>0 %
66 \everyjob\expandafter{%
67   \the\expandafter\everyjob
68   \csname\detokenize{lua_now_x:n}\endcsname{require("expl3")}%
69 }%
70 \fi
71 \fi
72 \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.

```

73 \begin{group}
74 \def\next{\endgroup}%
75 \def\ShortText{Required primitives not found}%
76 \def\LongText%
77 {%
78   LaTeX3 requires the e-TeX primitives and additional functionality as
79   described in the README file.
80   \LineBreak
81   These are available in the engines\LineBreak
82   - pdfTeX v1.40\LineBreak
83   - XeTeX v0.9994\LineBreak
84   - LuaTeX v0.70\LineBreak
85   - e-(u)pTeX mid-2012\LineBreak
86   or later.\LineBreak
87   \LineBreak
88 }%
89 \ifnum0%
90 \expandafter\ifx\csname pdfstrcmp\endcsname\relax

```

```

91 \else
92 \expandafter\ifx\csname pdftexversion\endcsname\relax
93 1%
94 \else
95 \ifnum\pdftexversion<140 \else 1\fi
96 \fi
97 \fi
98 \expandafter\ifx\csname directlua\endcsname\relax
99 \else
100 \ifnum\luatexversion<40 \else 1\fi
101 \fi
102 =0 %
103 \newlinechar'\^^J %
104 <*initex>
105 \def\LineBreak{^^J}%
106 \edef\next
107 {%
108 \errhelp
109 {%
110 \LongText
111 For pdfTeX and XeTeX the '-etex' command-line switch is also
112 needed.\LineBreak
113 \LineBreak
114 Format building will abort!\LineBreak
115 }%
116 \errmessage{\ShortText}%
117 \endgroup
118 \noexpand\end
119 }%
120 </initex>
121 <*package>
122 \def\LineBreak{\noexpand\MessageBreak}%
123 \expandafter\ifx\csname PackageError\endcsname\relax
124 \def\LineBreak{^^J}%
125 \def\PackageError#1#2#3%
126 {%
127 \errhelp{#3}%
128 \errmessage{#1 Error: #2}%
129 }%
130 \fi
131 \edef\next
132 {%
133 \noexpand\PackageError{expl3}{\ShortText}
134 {\LongText Loading of expl3 will abort!}%
135 \endgroup
136 \noexpand\endinput
137 }%
138 </package>
139 \fi
140 \next

```

1.5 Extending allocators

In format mode, allocating registers is handled by `l3alloc`. However, in package mode it's much safer to rely on more general code. For example, the ability to extend \TeX 's allocation routine to allow for $\varepsilon\text{-}\text{\TeX}$ has been around since 1997 in the `etex` package.

Loading this support is delayed until here as we are now sure that the $\varepsilon\text{-}\text{\TeX}$ 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.

```
141 \begin{package}
142 \begin{group}
143   \def\@tempa{LaTeX2e}%
144   \def\next{}%
145   \ifx\fmtname\@tempa
146     \expandafter\ifx\csname extrafloats\endcsname\relax
147       \def\next
148         {%
149           \RequirePackage{etex}%
150           \csname reserveinserts\endcsname{32}%
151         }%
152     \fi
153   \fi
154 \expandafter\endgroup
155 \next
156 \end{package}
```

1.6 Character data

\TeX needs various pieces of data to be set about characters, in particular which ones to treat as letters and which `\lccode` values apply as these affect hyphenation. It makes most sense to set this and related information up in one place. Whilst for \LaTeX hyphenation patterns can be read anywhere, other engines have to build them into the format and so we *must* do this set up before reading the patterns. For the Unicode engines, there are shared loaders available to obtain the relevant information directly from the Unicode Consortium data files. These need standard (Ini) \TeX category codes and primitive availability and must therefore be loaded *very* early. This has a knock-on effect on the 8-bit set up: it makes sense to do the definitions for those here as well so it is all in one place.

For \XeTeX and \LuaTeX , which are natively Unicode engines, simply load the Unicode data.

```
157 \begin{initex}
158 \ifdefined\Umathcode
```

```

159 \input load-unicode-data %
160 \input load-unicode-math-classes %
161 \else

```

For the 8-bit engines a font encoding scheme must be chosen. At present, this is the EC (T1) scheme, with the assumption that languages for which this is not appropriate will be used with one of the Unicode engines.

```

162 \begingroup

```

Lower case chars: map to themselves when lower casing and down by "20 when upper casing. (The characters a–z are set up correctly by IniT_EX.)

```

163 \def\temp{%
164   \ifnum\count0>\count2 %
165   \else
166     \global\lccode\count0 = \count0 %
167     \global\uccode\count0 = \numexpr\count0 - "20\relax
168     \advance\count0 by 1 %
169     \expandafter\temp
170   \fi
171 }
172 \count0 = "A0 %
173 \count2 = "BC %
174 \temp
175 \count0 = "E0 %
176 \count2 = "FF %
177 \temp

```

Upper case chars: map up by "20 when lower casing, to themselves when upper casing and require an \sfcode of 999. (The characters A–Z are set up correctly by IniT_EX.)

```

178 \def\temp{%
179   \ifnum\count0>\count2 %
180   \else
181     \global\lccode\count0 = \numexpr\count0 + "20\relax
182     \global\uccode\count0 = \count0 %
183     \global\sfcode\count0 = 999 %
184     \advance\count0 by 1 %
185     \expandafter\temp
186   \fi
187 }
188 \count0 = "80 %
189 \count2 = "9C %
190 \temp
191 \count0 = "C0 %
192 \count2 = "DF %
193 \temp

```

A few special cases where things are not as one might expect using the above pattern: dotless-I, dotless-J, dotted-I and d-bar.

```

194 \global\lccode'\^Y = '\^Y %
195 \global\uccode'\^Y = '\I %
196 \global\lccode'\^Z = '\^Z %
197 \global\uccode'\^Y = '\J %
198 \global\lccode"9D = '\i %
199 \global\uccode"9D = "9D %
200 \global\lccode"9E = "9E %
201 \global\uccode"9E = "D0 %

```

Allow hyphenation at a zero-width glyph (used to break up ligatures or to place accents between characters).

```
202 \global\lccode23 = 23 %
203 \endgroup
204 \fi
```

In all cases it makes sense to set up - to map to itself: this allows hyphenation of the rest of a word following it (suggested by Lars Helström).

```
205 \global\lccode'\- = '\- %
206 \</initex>
```

1.7 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` will be a “do nothing” command until `\ExplSyntaxOn` is used. For format mode, there is no need to save category codes so that step is skipped.

```
207 \protected\def\ExplSyntaxOff{}%
208 \<{*package}
209 \protected\edef\ExplSyntaxOff
210 {%
211   \protected\def\ExplSyntaxOff{}%
212   \catcode 9 = \the\catcode 9\relax
213   \catcode 32 = \the\catcode 32\relax
214   \catcode 34 = \the\catcode 34\relax
215   \catcode 38 = \the\catcode 38\relax
216   \catcode 58 = \the\catcode 58\relax
217   \catcode 94 = \the\catcode 94\relax
218   \catcode 95 = \the\catcode 95\relax
219   \catcode 124 = \the\catcode 124\relax
220   \catcode 126 = \the\catcode 126\relax
221   \endlinechar = \the\endlinechar\relax
222   \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
223 }%
224 \</package>
```

(End definition for \ExplSyntaxOff. This function is documented on page 6.)

The code environment is now set up.

```
225 \catcode 9 = 9\relax
226 \catcode 32 = 9\relax
227 \catcode 34 = 12\relax
228 \catcode 38 = 4\relax
229 \catcode 58 = 11\relax
230 \catcode 94 = 7\relax
231 \catcode 95 = 11\relax
232 \catcode 124 = 12\relax
233 \catcode 126 = 10\relax
234 \endlinechar = 32\relax
```

\l__kernel_expl_bool The status for experimental code syntax: this is on at present.

```
235 \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` will alter the definition of `\ExplSyntaxOff` and so in package mode this function should not be used until after the end of the loading process!

```

236 \protected \def \ExplSyntaxOn
237 {
238   \bool_if:NF \l__kernel_expl_bool
239   {
240     \cs_set_protected:Npx \ExplSyntaxOff
241     {
242       \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
243       \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
244       \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
245       \char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
246       \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
247       \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
248       \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
249       \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
250       \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
251       \tex_endlinechar:D =
252       \tex_the:D \tex_endlinechar:D \scan_stop:
253       \bool_set_false:N \l__kernel_expl_bool
254       \cs_set_protected:Npn \ExplSyntaxOff { }
255     }
256   }
257   \char_set_catcode_ignore:n { 9 } % tab
258   \char_set_catcode_ignore:n { 32 } % space
259   \char_set_catcode_other:n { 34 } % double quote
260   \char_set_catcode_alignment:n { 38 } % ampersand
261   \char_set_catcode_letter:n { 58 } % colon
262   \char_set_catcode_math_superscript:n { 94 } % circumflex
263   \char_set_catcode_letter:n { 95 } % underscore
264   \char_set_catcode_other:n { 124 } % pipe
265   \char_set_catcode_space:n { 126 } % tilde
266   \tex_endlinechar:D = 32 \scan_stop:
267   \bool_set_true:N \l__kernel_expl_bool
268 }

```

(End definition for `\ExplSyntaxOn`. This function is documented on page 6.)

269 `</initex | package>`

2 l3names implementation

270 `<*initex | package>`

No prefix substitution here.

271 `<@@=`

The code here simply renames all of the primitives to new, internal, names. In format mode, it also deletes all of the existing names (although some do come back later).

\tex_undefined:D This function does not exist at all, but is the name used by the plain T_EX format for an undefined function. So it should be marked here as “taken”.

(End definition for `\tex_undefined:D`.)

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.

```
272 \let \tex_global:D \global
273 \let \tex_let:D \let
```

Everything is inside a (rather long) group, which keeps `__kernel_primitive:NN` trapped.

```
274 \begingroup
```

`__kernel_primitive:NN` A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```
275 \long \def \__kernel_primitive:NN #1#2
276 {
277   \tex_global:D \tex_let:D #2 #1
278   (*initex)
279   \tex_global:D \tex_let:D #1 \tex_undefined:D
280   (/initex)
281 }
```

(End definition for `__kernel_primitive:NN`.)

To allow extracting “just the names”, a bit of DocStrip fiddling.

```
282 (/initex | package)
283 (*initex | 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.

```
284 \__kernel_primitive:NN \tex_space:D
285 \__kernel_primitive:NN \tex_italiccorrection:D
286 \__kernel_primitive:NN \tex_hyphen:D
```

Now all the other primitives.

```
287 \__kernel_primitive:NN \tex_above:D
288 \__kernel_primitive:NN \tex_abovedisplayshortskip:D
289 \__kernel_primitive:NN \tex_abovedisplayskip:D
290 \__kernel_primitive:NN \tex_abovewithdelims:D
291 \__kernel_primitive:NN \tex_accent:D
292 \__kernel_primitive:NN \tex_adjdemerits:D
293 \__kernel_primitive:NN \tex_advance:D
294 \__kernel_primitive:NN \tex_afterassignment:D
295 \__kernel_primitive:NN \tex_aftergroup:D
296 \__kernel_primitive:NN \tex_atop:D
297 \__kernel_primitive:NN \tex_atopwithdelims:D
298 \__kernel_primitive:NN \tex_badness:D
299 \__kernel_primitive:NN \tex_baselineskip:D
300 \__kernel_primitive:NN \tex_batchmode:D
301 \__kernel_primitive:NN \tex_begingroup:D
302 \__kernel_primitive:NN \tex_belowdisplayshortskip:D
303 \__kernel_primitive:NN \tex_belowdisplayskip:D
304 \__kernel_primitive:NN \tex_binoppenalty:D
305 \__kernel_primitive:NN \tex_botmark:D
```


306	_kernel_primitive:NN \box	\tex_box:D
307	_kernel_primitive:NN \boxmaxdepth	\tex_boxmaxdepth:D
308	_kernel_primitive:NN \brokenpenalty	\tex_brokenpenalty:D
309	_kernel_primitive:NN \catcode	\tex_catcode:D
310	_kernel_primitive:NN \char	\tex_char:D
311	_kernel_primitive:NN \chardef	\tex_chardef:D
312	_kernel_primitive:NN \cleaders	\tex_cleaders:D
313	_kernel_primitive:NN \closein	\tex_closein:D
314	_kernel_primitive:NN \closeout	\tex_closeout:D
315	_kernel_primitive:NN \clubpenalty	\tex_clubpenalty:D
316	_kernel_primitive:NN \copy	\tex_copy:D
317	_kernel_primitive:NN \count	\tex_count:D
318	_kernel_primitive:NN \countdef	\tex_countdef:D
319	_kernel_primitive:NN \cr	\tex_cr:D
320	_kernel_primitive:NN \crrcr	\tex_crrcr:D
321	_kernel_primitive:NN \csname	\tex_csname:D
322	_kernel_primitive:NN \day	\tex_day:D
323	_kernel_primitive:NN \deadcycles	\tex_deadcycles:D
324	_kernel_primitive:NN \def	\tex_def:D
325	_kernel_primitive:NN \defaultthyphenchar	\tex_defaultthyphenchar:D
326	_kernel_primitive:NN \defaultskewchar	\tex_defaultskewchar:D
327	_kernel_primitive:NN \delcode	\tex_delcode:D
328	_kernel_primitive:NN \delimiter	\tex_delimiter:D
329	_kernel_primitive:NN \delimiterfactor	\tex_delimiterfactor:D
330	_kernel_primitive:NN \delimitershortfall	\tex_delimitershortfall:D
331	_kernel_primitive:NN \dimen	\tex_dimen:D
332	_kernel_primitive:NN \dimendef	\tex_dimendef:D
333	_kernel_primitive:NN \discretionary	\tex_discretionary:D
334	_kernel_primitive:NN \displayindent	\tex_displayindent:D
335	_kernel_primitive:NN \displaylimits	\tex_displaylimits:D
336	_kernel_primitive:NN \displaystyle	\tex_displaystyle:D
337	_kernel_primitive:NN \displaywidowpenalty	\tex_displaywidowpenalty:D
338	_kernel_primitive:NN \displaywidth	\tex_displaywidth:D
339	_kernel_primitive:NN \divide	\tex_divide:D
340	_kernel_primitive:NN \doublehyphendemerits	\tex_doublehyphendemerits:D
341	_kernel_primitive:NN \dp	\tex_dp:D
342	_kernel_primitive:NN \dump	\tex_dump:D
343	_kernel_primitive:NN \edef	\tex_edef:D
344	_kernel_primitive:NN \else	\tex_else:D
345	_kernel_primitive:NN \emergencystretch	\tex_emergencystretch:D
346	_kernel_primitive:NN \end	\tex_end:D
347	_kernel_primitive:NN \endcsname	\tex_endcsname:D
348	_kernel_primitive:NN \endgroup	\tex_endgroup:D
349	_kernel_primitive:NN \endinput	\tex_endinput:D
350	_kernel_primitive:NN \endlinechar	\tex_endlinechar:D
351	_kernel_primitive:NN \eqno	\tex_eqno:D
352	_kernel_primitive:NN \errhelp	\tex_errhelp:D
353	_kernel_primitive:NN \errmessage	\tex_errmessage:D
354	_kernel_primitive:NN \errorcontextlines	\tex_errorcontextlines:D
355	_kernel_primitive:NN \errorstopmode	\tex_errorstopmode:D
356	_kernel_primitive:NN \escapechar	\tex_escapechar:D
357	_kernel_primitive:NN \everycr	\tex_everycr:D
358	_kernel_primitive:NN \everydisplay	\tex_everydisplay:D
359	_kernel_primitive:NN \everyhbox	\tex_everyhbox:D

360	_kernel_primitive:NN	\everyjob	\tex_everyjob:D
361	_kernel_primitive:NN	\everymath	\tex_everymath:D
362	_kernel_primitive:NN	\everypar	\tex_everypar:D
363	_kernel_primitive:NN	\everyvbox	\tex_everyvbox:D
364	_kernel_primitive:NN	\exhyphenpenalty	\tex_exhyphenpenalty:D
365	_kernel_primitive:NN	\expandafter	\tex_expandafter:D
366	_kernel_primitive:NN	\fam	\tex_fam:D
367	_kernel_primitive:NN	\fi	\tex_fi:D
368	_kernel_primitive:NN	\finalhyphendemerits	\tex_finalhyphendemerits:D
369	_kernel_primitive:NN	\firstmark	\tex_firstmark:D
370	_kernel_primitive:NN	\floatingpenalty	\tex_floatingpenalty:D
371	_kernel_primitive:NN	\font	\tex_font:D
372	_kernel_primitive:NN	\fontdimen	\tex_fontdimen:D
373	_kernel_primitive:NN	\fontname	\tex_fontname:D
374	_kernel_primitive:NN	\futurelet	\tex_futurelet:D
375	_kernel_primitive:NN	\gdef	\tex_gdef:D
376	_kernel_primitive:NN	\global	\tex_global:D
377	_kernel_primitive:NN	\globaldefs	\tex_globaldefs:D
378	_kernel_primitive:NN	\halign	\tex_halign:D
379	_kernel_primitive:NN	\hangafter	\tex_hangafter:D
380	_kernel_primitive:NN	\hangindent	\tex_hangindent:D
381	_kernel_primitive:NN	\hbadness	\tex_hbadness:D
382	_kernel_primitive:NN	\hbox	\tex_hbox:D
383	_kernel_primitive:NN	\hfil	\tex_hfil:D
384	_kernel_primitive:NN	\hfill	\tex_hfill:D
385	_kernel_primitive:NN	\hfilneg	\tex_hfilneg:D
386	_kernel_primitive:NN	\hfuzz	\tex_hfuzz:D
387	_kernel_primitive:NN	\hoffset	\tex_hoffset:D
388	_kernel_primitive:NN	\holdinginserts	\tex_holdinginserts:D
389	_kernel_primitive:NN	\hrule	\tex_hrule:D
390	_kernel_primitive:NN	\hsize	\tex_hsize:D
391	_kernel_primitive:NN	\hskip	\tex_hskip:D
392	_kernel_primitive:NN	\hss	\tex_hss:D
393	_kernel_primitive:NN	\ht	\tex_ht:D
394	_kernel_primitive:NN	\hyphenation	\tex_hyphenation:D
395	_kernel_primitive:NN	\hyphenchar	\tex_hyphenchar:D
396	_kernel_primitive:NN	\hyphenpenalty	\tex_hyphenpenalty:D
397	_kernel_primitive:NN	\if	\tex_if:D
398	_kernel_primitive:NN	\ifcase	\tex_ifcase:D
399	_kernel_primitive:NN	\ifcat	\tex_ifcat:D
400	_kernel_primitive:NN	\ifdim	\tex_ifdim:D
401	_kernel_primitive:NN	\ifeof	\tex_ifeof:D
402	_kernel_primitive:NN	\iffalse	\tex_iffalse:D
403	_kernel_primitive:NN	\ifhbox	\tex_ifhbox:D
404	_kernel_primitive:NN	\ifhmode	\tex_ifhmode:D
405	_kernel_primitive:NN	\ifinner	\tex_ifinner:D
406	_kernel_primitive:NN	\ifmmode	\tex_ifmmode:D
407	_kernel_primitive:NN	\ifnum	\tex_ifnum:D
408	_kernel_primitive:NN	\ifodd	\tex_ifodd:D
409	_kernel_primitive:NN	\iftrue	\tex_iftrue:D
410	_kernel_primitive:NN	\ifvbox	\tex_ifvbox:D
411	_kernel_primitive:NN	\ifvmode	\tex_ifvmode:D
412	_kernel_primitive:NN	\ifvoid	\tex_ifvoid:D
413	_kernel_primitive:NN	\ifx	\tex_ifx:D

414	<code>_kernel_primitive:NN \ignorespaces</code>	<code>\tex_ignorespaces:D</code>
415	<code>_kernel_primitive:NN \immediate</code>	<code>\tex_immediate:D</code>
416	<code>_kernel_primitive:NN \indent</code>	<code>\tex_indent:D</code>
417	<code>_kernel_primitive:NN \input</code>	<code>\tex_input:D</code>
418	<code>_kernel_primitive:NN \inputlineno</code>	<code>\tex_inputlineno:D</code>
419	<code>_kernel_primitive:NN \insert</code>	<code>\tex_insert:D</code>
420	<code>_kernel_primitive:NN \insertpenalties</code>	<code>\tex_insertpenalties:D</code>
421	<code>_kernel_primitive:NN \interlinepenalty</code>	<code>\tex_interlinepenalty:D</code>
422	<code>_kernel_primitive:NN \jobname</code>	<code>\tex_jobname:D</code>
423	<code>_kernel_primitive:NN \kern</code>	<code>\tex_kern:D</code>
424	<code>_kernel_primitive:NN \language</code>	<code>\tex_language:D</code>
425	<code>_kernel_primitive:NN \lastbox</code>	<code>\tex_lastbox:D</code>
426	<code>_kernel_primitive:NN \lastkern</code>	<code>\tex_lastkern:D</code>
427	<code>_kernel_primitive:NN \lastpenalty</code>	<code>\tex_lastpenalty:D</code>
428	<code>_kernel_primitive:NN \lastskip</code>	<code>\tex_lastskip:D</code>
429	<code>_kernel_primitive:NN \lccode</code>	<code>\tex_lccode:D</code>
430	<code>_kernel_primitive:NN \leaders</code>	<code>\tex_leaders:D</code>
431	<code>_kernel_primitive:NN \left</code>	<code>\tex_left:D</code>
432	<code>_kernel_primitive:NN \lefthyphenmin</code>	<code>\tex_lefthyphenmin:D</code>
433	<code>_kernel_primitive:NN \leftskip</code>	<code>\tex_leftskip:D</code>
434	<code>_kernel_primitive:NN \leqno</code>	<code>\tex_leqno:D</code>
435	<code>_kernel_primitive:NN \let</code>	<code>\tex_let:D</code>
436	<code>_kernel_primitive:NN \limits</code>	<code>\tex_limits:D</code>
437	<code>_kernel_primitive:NN \linepenalty</code>	<code>\tex_linepenalty:D</code>
438	<code>_kernel_primitive:NN \lineskip</code>	<code>\tex_lineskip:D</code>
439	<code>_kernel_primitive:NN \lineskiplimit</code>	<code>\tex_lineskiplimit:D</code>
440	<code>_kernel_primitive:NN \long</code>	<code>\tex_long:D</code>
441	<code>_kernel_primitive:NN \looseness</code>	<code>\tex_looseness:D</code>
442	<code>_kernel_primitive:NN \lower</code>	<code>\tex_lower:D</code>
443	<code>_kernel_primitive:NN \lowercase</code>	<code>\tex_lowercase:D</code>
444	<code>_kernel_primitive:NN \mag</code>	<code>\tex_mag:D</code>
445	<code>_kernel_primitive:NN \mark</code>	<code>\tex_mark:D</code>
446	<code>_kernel_primitive:NN \mathaccent</code>	<code>\tex_mathaccent:D</code>
447	<code>_kernel_primitive:NN \mathbin</code>	<code>\tex_mathbin:D</code>
448	<code>_kernel_primitive:NN \mathchar</code>	<code>\tex_mathchar:D</code>
449	<code>_kernel_primitive:NN \mathchardef</code>	<code>\tex_mathchardef:D</code>
450	<code>_kernel_primitive:NN \mathchoice</code>	<code>\tex_mathchoice:D</code>
451	<code>_kernel_primitive:NN \mathclose</code>	<code>\tex_mathclose:D</code>
452	<code>_kernel_primitive:NN \mathcode</code>	<code>\tex_mathcode:D</code>
453	<code>_kernel_primitive:NN \mathinner</code>	<code>\tex_mathinner:D</code>
454	<code>_kernel_primitive:NN \mathop</code>	<code>\tex_mathop:D</code>
455	<code>_kernel_primitive:NN \mathopen</code>	<code>\tex_mathopen:D</code>
456	<code>_kernel_primitive:NN \mathord</code>	<code>\tex_mathord:D</code>
457	<code>_kernel_primitive:NN \mathpunct</code>	<code>\tex_mathpunct:D</code>
458	<code>_kernel_primitive:NN \mathrel</code>	<code>\tex_mathrel:D</code>
459	<code>_kernel_primitive:NN \mathsurround</code>	<code>\tex_mathsurround:D</code>
460	<code>_kernel_primitive:NN \maxdeadcycles</code>	<code>\tex_maxdeadcycles:D</code>
461	<code>_kernel_primitive:NN \maxdepth</code>	<code>\tex_maxdepth:D</code>
462	<code>_kernel_primitive:NN \meaning</code>	<code>\tex_meaning:D</code>
463	<code>_kernel_primitive:NN \medmuskip</code>	<code>\tex_medmuskip:D</code>
464	<code>_kernel_primitive:NN \message</code>	<code>\tex_message:D</code>
465	<code>_kernel_primitive:NN \mkern</code>	<code>\tex_mkern:D</code>
466	<code>_kernel_primitive:NN \month</code>	<code>\tex_month:D</code>
467	<code>_kernel_primitive:NN \moveleft</code>	<code>\tex_moveleft:D</code>

468	_kernel_primitive:NN	\moveright	\tex_moveright:D
469	_kernel_primitive:NN	\mskip	\tex_mskip:D
470	_kernel_primitive:NN	\multiply	\tex_multiply:D
471	_kernel_primitive:NN	\muskip	\tex_muskip:D
472	_kernel_primitive:NN	\muskipdef	\tex_muskipdef:D
473	_kernel_primitive:NN	\newlinechar	\tex_newlinechar:D
474	_kernel_primitive:NN	\noalign	\tex_noalign:D
475	_kernel_primitive:NN	\noboundary	\tex_noboundary:D
476	_kernel_primitive:NN	\noexpand	\tex_noexpand:D
477	_kernel_primitive:NN	\noindent	\tex_noindent:D
478	_kernel_primitive:NN	\nolimits	\tex_nolimits:D
479	_kernel_primitive:NN	\nonscript	\tex_nonscript:D
480	_kernel_primitive:NN	\nonstopmode	\tex_nonstopmode:D
481	_kernel_primitive:NN	\nulldelimiterspace	\tex_nulldelimiterspace:D
482	_kernel_primitive:NN	\nullfont	\tex_nullfont:D
483	_kernel_primitive:NN	\number	\tex_number:D
484	_kernel_primitive:NN	\omit	\tex_omit:D
485	_kernel_primitive:NN	\openin	\tex_openin:D
486	_kernel_primitive:NN	\openout	\tex_openout:D
487	_kernel_primitive:NN	\or	\tex_or:D
488	_kernel_primitive:NN	\outer	\tex_outer:D
489	_kernel_primitive:NN	\output	\tex_output:D
490	_kernel_primitive:NN	\outputpenalty	\tex_outputpenalty:D
491	_kernel_primitive:NN	\over	\tex_over:D
492	_kernel_primitive:NN	\overfullrule	\tex_overfullrule:D
493	_kernel_primitive:NN	\overline	\tex_overline:D
494	_kernel_primitive:NN	\overwithdelims	\tex_overwithdelims:D
495	_kernel_primitive:NN	\pagedepth	\tex_pagedepth:D
496	_kernel_primitive:NN	\pagefilllstretch	\tex_pagefilllstretch:D
497	_kernel_primitive:NN	\pagefillstretch	\tex_pagefillstretch:D
498	_kernel_primitive:NN	\pagefilstretch	\tex_pagefilstretch:D
499	_kernel_primitive:NN	\pagegoal	\tex_pagegoal:D
500	_kernel_primitive:NN	\pageshrink	\tex_pageshrink:D
501	_kernel_primitive:NN	\pagestretch	\tex_pagestretch:D
502	_kernel_primitive:NN	\pagetotal	\tex_pagetotal:D
503	_kernel_primitive:NN	\par	\tex_par:D
504	_kernel_primitive:NN	\parfillskip	\tex_parfillskip:D
505	_kernel_primitive:NN	\parindent	\tex_parindent:D
506	_kernel_primitive:NN	\parshape	\tex_parshape:D
507	_kernel_primitive:NN	\parskip	\tex_parskip:D
508	_kernel_primitive:NN	\patterns	\tex_patterns:D
509	_kernel_primitive:NN	\pausing	\tex_pausing:D
510	_kernel_primitive:NN	\penalty	\tex_penalty:D
511	_kernel_primitive:NN	\postdisplaypenalty	\tex_postdisplaypenalty:D
512	_kernel_primitive:NN	\predisdisplaypenalty	\tex_predisdisplaypenalty:D
513	_kernel_primitive:NN	\predisplaysize	\tex_predisplaysize:D
514	_kernel_primitive:NN	\pretolerance	\tex_pretolerance:D
515	_kernel_primitive:NN	\prevdepth	\tex_prevdepth:D
516	_kernel_primitive:NN	\prevgraf	\tex_prevgraf:D
517	_kernel_primitive:NN	\radical	\tex_radical:D
518	_kernel_primitive:NN	\raise	\tex_raise:D
519	_kernel_primitive:NN	\read	\tex_read:D
520	_kernel_primitive:NN	\relax	\tex_relax:D
521	_kernel_primitive:NN	\relpenalty	\tex_relpenalty:D

522	_kernel_primitive:NN	\right	\tex_right:D
523	_kernel_primitive:NN	\righthypenmin	\tex_righthypenmin:D
524	_kernel_primitive:NN	\rightskip	\tex_rightskip:D
525	_kernel_primitive:NN	\romannumeral	\tex_romannumeral:D
526	_kernel_primitive:NN	\scriptfont	\tex_scriptfont:D
527	_kernel_primitive:NN	\scriptscriptfont	\tex_scriptscriptfont:D
528	_kernel_primitive:NN	\scriptscriptstyle	\tex_scriptscriptstyle:D
529	_kernel_primitive:NN	\scriptspace	\tex_scriptspace:D
530	_kernel_primitive:NN	\scriptstyle	\tex_scriptstyle:D
531	_kernel_primitive:NN	\scrollmode	\tex_scrollmode:D
532	_kernel_primitive:NN	\setbox	\tex_setbox:D
533	_kernel_primitive:NN	\setlanguage	\tex_setlanguage:D
534	_kernel_primitive:NN	\sfcode	\tex_sfcode:D
535	_kernel_primitive:NN	\shipout	\tex_shipout:D
536	_kernel_primitive:NN	\show	\tex_show:D
537	_kernel_primitive:NN	\showbox	\tex_showbox:D
538	_kernel_primitive:NN	\showboxbreadth	\tex_showboxbreadth:D
539	_kernel_primitive:NN	\showboxdepth	\tex_showboxdepth:D
540	_kernel_primitive:NN	\showlists	\tex_showlists:D
541	_kernel_primitive:NN	\showthe	\tex_showthe:D
542	_kernel_primitive:NN	\skewchar	\tex_skewchar:D
543	_kernel_primitive:NN	\skip	\tex_skip:D
544	_kernel_primitive:NN	\skipdef	\tex_skipdef:D
545	_kernel_primitive:NN	\spacefactor	\tex_spacefactor:D
546	_kernel_primitive:NN	\spaceskip	\tex_spaceskip:D
547	_kernel_primitive:NN	\span	\tex_span:D
548	_kernel_primitive:NN	\special	\tex_special:D
549	_kernel_primitive:NN	\splitbotmark	\tex_splitbotmark:D
550	_kernel_primitive:NN	\splitfirstmark	\tex_splitfirstmark:D
551	_kernel_primitive:NN	\splitmaxdepth	\tex_splitmaxdepth:D
552	_kernel_primitive:NN	\splittopskip	\tex_splittopskip:D
553	_kernel_primitive:NN	\string	\tex_string:D
554	_kernel_primitive:NN	\tabskip	\tex_tabskip:D
555	_kernel_primitive:NN	\textfont	\tex_textfont:D
556	_kernel_primitive:NN	\textstyle	\tex_textstyle:D
557	_kernel_primitive:NN	\the	\tex_the:D
558	_kernel_primitive:NN	\thickmuskip	\tex_thickmuskip:D
559	_kernel_primitive:NN	\thinmuskip	\tex_thinmuskip:D
560	_kernel_primitive:NN	\time	\tex_time:D
561	_kernel_primitive:NN	\toks	\tex_toks:D
562	_kernel_primitive:NN	\toksdef	\tex_toksdef:D
563	_kernel_primitive:NN	\tolerance	\tex_tolerance:D
564	_kernel_primitive:NN	\topmark	\tex_topmark:D
565	_kernel_primitive:NN	\topskip	\tex_topskip:D
566	_kernel_primitive:NN	\tracingcommands	\tex_tracingcommands:D
567	_kernel_primitive:NN	\tracinglostchars	\tex_tracinglostchars:D
568	_kernel_primitive:NN	\tracingmacros	\tex_tracingmacros:D
569	_kernel_primitive:NN	\tracingonline	\tex_tracingonline:D
570	_kernel_primitive:NN	\tracingoutput	\tex_tracingoutput:D
571	_kernel_primitive:NN	\tracingpages	\tex_tracingpages:D
572	_kernel_primitive:NN	\tracingparagraphs	\tex_tracingparagraphs:D
573	_kernel_primitive:NN	\tracingrestores	\tex_tracingrestores:D
574	_kernel_primitive:NN	\tracingstats	\tex_tracingstats:D
575	_kernel_primitive:NN	\uccode	\tex_uccode:D

576	__kernel_primitive:NN	\uchyph	\tex_uchyph:D
577	__kernel_primitive:NN	\underline	\tex_underline:D
578	__kernel_primitive:NN	\unhbox	\tex_unhbox:D
579	__kernel_primitive:NN	\unhcopy	\tex_unhcopy:D
580	__kernel_primitive:NN	\unkern	\tex_unkern:D
581	__kernel_primitive:NN	\unpenalty	\tex_unpenalty:D
582	__kernel_primitive:NN	\unskip	\tex_unskip:D
583	__kernel_primitive:NN	\unvbox	\tex_unvbox:D
584	__kernel_primitive:NN	\unvcopy	\tex_unvcopy:D
585	__kernel_primitive:NN	\uppercase	\tex_uppercase:D
586	__kernel_primitive:NN	\vadjust	\tex_vadjust:D
587	__kernel_primitive:NN	\valign	\tex_valign:D
588	__kernel_primitive:NN	\vbadness	\tex_vbadness:D
589	__kernel_primitive:NN	\vbox	\tex_vbox:D
590	__kernel_primitive:NN	\vcenter	\tex_vcenter:D
591	__kernel_primitive:NN	\vfil	\tex_vfil:D
592	__kernel_primitive:NN	\vfill	\tex_vfill:D
593	__kernel_primitive:NN	\vfilneg	\tex_vfilneg:D
594	__kernel_primitive:NN	\vfuzz	\tex_vfuzz:D
595	__kernel_primitive:NN	\voffset	\tex_voffset:D
596	__kernel_primitive:NN	\vrule	\tex_vrule:D
597	__kernel_primitive:NN	\vsize	\tex_vsize:D
598	__kernel_primitive:NN	\vskip	\tex_vskip:D
599	__kernel_primitive:NN	\vsplit	\tex_vsplit:D
600	__kernel_primitive:NN	\vss	\tex_vss:D
601	__kernel_primitive:NN	\vtop	\tex_vtop:D
602	__kernel_primitive:NN	\wd	\tex_wd:D
603	__kernel_primitive:NN	\widowpenalty	\tex_widowpenalty:D
604	__kernel_primitive:NN	\write	\tex_write:D
605	__kernel_primitive:NN	\xdef	\tex_xdef:D
606	__kernel_primitive:NN	\xleaders	\tex_xleaders:D
607	__kernel_primitive:NN	\xspaceskip	\tex_xspaceskip:D
608	__kernel_primitive:NN	\year	\tex_year:D

Since L^AT_EX3 requires at least the ε -T_EX extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

609	__kernel_primitive:NN	\beginL	\etex_beginL:D
610	__kernel_primitive:NN	\beginR	\etex_beginR:D
611	__kernel_primitive:NN	\botmarks	\etex_botmarks:D
612	__kernel_primitive:NN	\clubpenalties	\etex_clubpenalties:D
613	__kernel_primitive:NN	\currentgrouplevel	\etex_currentgrouplevel:D
614	__kernel_primitive:NN	\currentgrouptype	\etex_currentgrouptype:D
615	__kernel_primitive:NN	\currentifbranch	\etex_currentifbranch:D
616	__kernel_primitive:NN	\currentiflevel	\etex_currentiflevel:D
617	__kernel_primitive:NN	\currentifttype	\etex_currentifttype:D
618	__kernel_primitive:NN	\detokenize	\etex_detokenize:D
619	__kernel_primitive:NN	\dimexpr	\etex_dimexpr:D
620	__kernel_primitive:NN	\displaywidowpenalties	\etex_displaywidowpenalties:D
621	__kernel_primitive:NN	\endL	\etex_endL:D
622	__kernel_primitive:NN	\endR	\etex_endR:D
623	__kernel_primitive:NN	\eTeXrevision	\etex_eTeXrevision:D
624	__kernel_primitive:NN	\eTeXversion	\etex_eTeXversion:D
625	__kernel_primitive:NN	\everyeof	\etex_everyeof:D
626	__kernel_primitive:NN	\firstmarks	\etex_firstmarks:D

627	<code>__kernel_primitive:NN \fontchardp</code>	<code>\etex_fontchardp:D</code>
628	<code>__kernel_primitive:NN \fontcharht</code>	<code>\etex_fontcharht:D</code>
629	<code>__kernel_primitive:NN \fontcharic</code>	<code>\etex_fontcharic:D</code>
630	<code>__kernel_primitive:NN \fontcharwd</code>	<code>\etex_fontcharwd:D</code>
631	<code>__kernel_primitive:NN \glueexpr</code>	<code>\etex_glueexpr:D</code>
632	<code>__kernel_primitive:NN \glueshrink</code>	<code>\etex_glueshrink:D</code>
633	<code>__kernel_primitive:NN \glueshrinkorder</code>	<code>\etex_glueshrinkorder:D</code>
634	<code>__kernel_primitive:NN \gluestretch</code>	<code>\etex_gluestretch:D</code>
635	<code>__kernel_primitive:NN \gluestretchorder</code>	<code>\etex_gluestretchorder:D</code>
636	<code>__kernel_primitive:NN \gluetomu</code>	<code>\etex_gluetomu:D</code>
637	<code>__kernel_primitive:NN \ifcsname</code>	<code>\etex_ifcsname:D</code>
638	<code>__kernel_primitive:NN \ifdefined</code>	<code>\etex_ifdefined:D</code>
639	<code>__kernel_primitive:NN \iffontchar</code>	<code>\etex_iffontchar:D</code>
640	<code>__kernel_primitive:NN \interactionmode</code>	<code>\etex_interactionmode:D</code>
641	<code>__kernel_primitive:NN \interlinepenalties</code>	<code>\etex_interlinepenalties:D</code>
642	<code>__kernel_primitive:NN \lastlinefit</code>	<code>\etex_lastlinefit:D</code>
643	<code>__kernel_primitive:NN \lastnodetype</code>	<code>\etex_lastnodetype:D</code>
644	<code>__kernel_primitive:NN \marks</code>	<code>\etex_marks:D</code>
645	<code>__kernel_primitive:NN \middle</code>	<code>\etex_middle:D</code>
646	<code>__kernel_primitive:NN \muexpr</code>	<code>\etex_muexpr:D</code>
647	<code>__kernel_primitive:NN \mutoglua</code>	<code>\etex_mutoglua:D</code>
648	<code>__kernel_primitive:NN \numexpr</code>	<code>\etex_numexpr:D</code>
649	<code>__kernel_primitive:NN \pagediscards</code>	<code>\etex_pagediscards:D</code>
650	<code>__kernel_primitive:NN \parshapedimen</code>	<code>\etex_parshapedimen:D</code>
651	<code>__kernel_primitive:NN \parshapeindent</code>	<code>\etex_parshapeindent:D</code>
652	<code>__kernel_primitive:NN \parshapelength</code>	<code>\etex_parshapelength:D</code>
653	<code>__kernel_primitive:NN \predisplaydirection</code>	<code>\etex_predisplaydirection:D</code>
654	<code>__kernel_primitive:NN \protected</code>	<code>\etex_protected:D</code>
655	<code>__kernel_primitive:NN \readline</code>	<code>\etex_readline:D</code>
656	<code>__kernel_primitive:NN \savingshyphcodes</code>	<code>\etex_savingshyphcodes:D</code>
657	<code>__kernel_primitive:NN \savingsvdiscards</code>	<code>\etex_savingsvdiscards:D</code>
658	<code>__kernel_primitive:NN \scantokens</code>	<code>\etex_scantokens:D</code>
659	<code>__kernel_primitive:NN \showgroups</code>	<code>\etex_showgroups:D</code>
660	<code>__kernel_primitive:NN \showifs</code>	<code>\etex_showifs:D</code>
661	<code>__kernel_primitive:NN \showtokens</code>	<code>\etex_showtokens:D</code>
662	<code>__kernel_primitive:NN \splitbotmarks</code>	<code>\etex_splitbotmarks:D</code>
663	<code>__kernel_primitive:NN \splitdiscards</code>	<code>\etex_splitdiscards:D</code>
664	<code>__kernel_primitive:NN \splitfirstmarks</code>	<code>\etex_splitfirstmarks:D</code>
665	<code>__kernel_primitive:NN \TeXXeTstate</code>	<code>\etex_TeXeTstate:D</code>
666	<code>__kernel_primitive:NN \topmarks</code>	<code>\etex_topmarks:D</code>
667	<code>__kernel_primitive:NN \tracingassigns</code>	<code>\etex_tracingassigns:D</code>
668	<code>__kernel_primitive:NN \tracinggroups</code>	<code>\etex_tracinggroups:D</code>
669	<code>__kernel_primitive:NN \tracingifs</code>	<code>\etex_tracingifs:D</code>
670	<code>__kernel_primitive:NN \tracingnesting</code>	<code>\etex_tracingnesting:D</code>
671	<code>__kernel_primitive:NN \tracingscantokens</code>	<code>\etex_tracingscantokens:D</code>
672	<code>__kernel_primitive:NN \unexpanded</code>	<code>\etex_unexpanded:D</code>
673	<code>__kernel_primitive:NN \unless</code>	<code>\etex_unless:D</code>
674	<code>__kernel_primitive:NN \widowpenalties</code>	<code>\etex_widowpenalties:D</code>

The newer primitives are more complex: there are an awful lot of them, and we don't use them all at the moment. So the following is selective, based on those also available in LuaTeX or used in expl3. In the case of the pdfTeX primitives, we retain `pdf` at the start of the names *only* for directly PDF-related primitives, as there are a lot of pdfTeX primitives that start `\pdf...` but are not related to PDF output. These ones related to

PDF output or only work in PDF mode.

675	_kernel_primitive:NN	\pdfannot	\pdfTEX_pdfannot:D
676	_kernel_primitive:NN	\pdfcatalog	\pdfTEX_pdfcatalog:D
677	_kernel_primitive:NN	\pdfcompresslevel	\pdfTEX_pdfcompresslevel:D
678	_kernel_primitive:NN	\pdfcolorstack	\pdfTEX_pdfcolorstack:D
679	_kernel_primitive:NN	\pdfcolorstackinit	\pdfTEX_pdfcolorstackinit:D
680	_kernel_primitive:NN	\pdfcreationdate	\pdfTEX_pdfcreationdate:D
681	_kernel_primitive:NN	\pdfdecimaldigits	\pdfTEX_pdfdecimaldigits:D
682	_kernel_primitive:NN	\pdfdest	\pdfTEX_pdfdest:D
683	_kernel_primitive:NN	\pdfdestmargin	\pdfTEX_pdfdestmargin:D
684	_kernel_primitive:NN	\pdfendlink	\pdfTEX_pdfendlink:D
685	_kernel_primitive:NN	\pdfendthread	\pdfTEX_pdfendthread:D
686	_kernel_primitive:NN	\pdffontattr	\pdfTEX_pdffontattr:D
687	_kernel_primitive:NN	\pdffontname	\pdfTEX_pdffontname:D
688	_kernel_primitive:NN	\pdffontobjnum	\pdfTEX_pdffontobjnum:D
689	_kernel_primitive:NN	\pdfgamma	\pdfTEX_pdfgamma:D
690	_kernel_primitive:NN	\pdfimageapplygamma	\pdfTEX_pdfimageapplygamma:D
691	_kernel_primitive:NN	\pdfimagegamma	\pdfTEX_pdfimagegamma:D
692	_kernel_primitive:NN	\pdfgentounicode	\pdfTEX_pdfgentounicode:D
693	_kernel_primitive:NN	\pdfglyptounicode	\pdfTEX_pdfglyptounicode:D
694	_kernel_primitive:NN	\pdfhorigin	\pdfTEX_pdfhorigin:D
695	_kernel_primitive:NN	\pdfimagehicolor	\pdfTEX_pdfimagehicolor:D
696	_kernel_primitive:NN	\pdfimageresolution	\pdfTEX_pdfimageresolution:D
697	_kernel_primitive:NN	\pdfincludechars	\pdfTEX_pdfincludechars:D
698	_kernel_primitive:NN	\pdfinclusioncopyfonts	\pdfTEX_pdfinclusioncopyfonts:D
699	_kernel_primitive:NN	\pdfinclusionerrorlevel	\pdfTEX_pdfinclusionerrorlevel:D
700	_kernel_primitive:NN	\pdfinfo	\pdfTEX_pdfinfo:D
701	_kernel_primitive:NN	\pdflastannot	\pdfTEX_pdflastannot:D
702	_kernel_primitive:NN	\pdflastlink	\pdfTEX_pdflastlink:D
703	_kernel_primitive:NN	\pdflastobj	\pdfTEX_pdflastobj:D
704	_kernel_primitive:NN	\pdflastxform	\pdfTEX_pdflastxform:D
705	_kernel_primitive:NN	\pdflastximage	\pdfTEX_pdflastximage:D
706	_kernel_primitive:NN	\pdflastximagecolordepth	\pdfTEX_pdflastximagecolordepth:D
707	_kernel_primitive:NN	\pdflastximagepages	\pdfTEX_pdflastximagepages:D
708	_kernel_primitive:NN	\pdflinkmargin	\pdfTEX_pdflinkmargin:D
709	_kernel_primitive:NN	\pdfliteral	\pdfTEX_pdfliteral:D
710	_kernel_primitive:NN	\pdfminorversion	\pdfTEX_pdfminorversion:D
711	_kernel_primitive:NN	\pdfnames	\pdfTEX_pdfnames:D
712	_kernel_primitive:NN	\pdfobj	\pdfTEX_pdfobj:D
713	_kernel_primitive:NN	\pdfobjcompresslevel	\pdfTEX_pdfobjcompresslevel:D
714	_kernel_primitive:NN	\pdfoutline	\pdfTEX_pdfoutline:D
715	_kernel_primitive:NN	\pdfoutput	\pdfTEX_pdfoutput:D
716	_kernel_primitive:NN	\pdfpageattr	\pdfTEX_pdfpageattr:D
717	_kernel_primitive:NN	\pdfpagebox	\pdfTEX_pdfpagebox:D
718	_kernel_primitive:NN	\pdfpageref	\pdfTEX_pdfpageref:D
719	_kernel_primitive:NN	\pdfpageresources	\pdfTEX_pdfpageresources:D
720	_kernel_primitive:NN	\pdfpagesattr	\pdfTEX_pdfpagesattr:D
721	_kernel_primitive:NN	\pdfrefobj	\pdfTEX_pdfrefobj:D
722	_kernel_primitive:NN	\pdfrefxform	\pdfTEX_pdfrefxform:D
723	_kernel_primitive:NN	\pdfrefximage	\pdfTEX_pdfrefximage:D
724	_kernel_primitive:NN	\pdfrestore	\pdfTEX_pdfrestore:D
725	_kernel_primitive:NN	\pdfretval	\pdfTEX_pdfretval:D
726	_kernel_primitive:NN	\pdfsave	\pdfTEX_pdfsave:D
727	_kernel_primitive:NN	\pdfsetmatrix	\pdfTEX_pdfsetmatrix:D

728	_kernel_primitive:NN	\pdfstartlink	\pdfTEX_pdfstartlink:D
729	_kernel_primitive:NN	\pdfstartthread	\pdfTEX_pdfstartthread:D
730	_kernel_primitive:NN	\pdfsuppressptexinfo	\pdfTEX_pdfsuppressptexinfo:D
731	_kernel_primitive:NN	\pdfthread	\pdfTEX_pdfthread:D
732	_kernel_primitive:NN	\pdfthreadmargin	\pdfTEX_pdfthreadmargin:D
733	_kernel_primitive:NN	\pdftrailer	\pdfTEX_pdftrailer:D
734	_kernel_primitive:NN	\pdfuniqueresname	\pdfTEX_pdfuniqueresname:D
735	_kernel_primitive:NN	\pdfvorigin	\pdfTEX_pdfvorigin:D
736	_kernel_primitive:NN	\pdfxform	\pdfTEX_pdfxform:D
737	_kernel_primitive:NN	\pdfxformattr	\pdfTEX_pdfxformattr:D
738	_kernel_primitive:NN	\pdfxformname	\pdfTEX_pdfxformname:D
739	_kernel_primitive:NN	\pdfxformresources	\pdfTEX_pdfxformresources:D
740	_kernel_primitive:NN	\pdfximage	\pdfTEX_pdfximage:D
741	_kernel_primitive:NN	\pdfximagebbox	\pdfTEX_pdfximagebbox:D

While these are not.

742	_kernel_primitive:NN	\ifpdfabsdim	\pdfTEX_ifabsdim:D
743	_kernel_primitive:NN	\ifpdfabsnum	\pdfTEX_ifabsnum:D
744	_kernel_primitive:NN	\ifpdfprimitive	\pdfTEX_ifprimitive:D
745	_kernel_primitive:NN	\pdfadjustspacing	\pdfTEX_adjustspacing:D
746	_kernel_primitive:NN	\pdfcopyfont	\pdfTEX_copyfont:D
747	_kernel_primitive:NN	\pdfdraftmode	\pdfTEX_draftmode:D
748	_kernel_primitive:NN	\pdfeachlinedepth	\pdfTEX_eachlinedepth:D
749	_kernel_primitive:NN	\pdfeachlineheight	\pdfTEX_eachlineheight:D
750	_kernel_primitive:NN	\pdffirstlineheight	\pdfTEX_firstlineheight:D
751	_kernel_primitive:NN	\pdffontexpand	\pdfTEX_fontexpand:D
752	_kernel_primitive:NN	\pdffontsize	\pdfTEX_fontsize:D
753	_kernel_primitive:NN	\pdfignoreddimen	\pdfTEX_ignoreddimen:D
754	_kernel_primitive:NN	\pdfinsertht	\pdfTEX_insertht:D
755	_kernel_primitive:NN	\pdflastlinedepth	\pdfTEX_lastlinedepth:D
756	_kernel_primitive:NN	\pdflastxpos	\pdfTEX_lastxpos:D
757	_kernel_primitive:NN	\pdflastypos	\pdfTEX_lastypos:D
758	_kernel_primitive:NN	\pdfmapfile	\pdfTEX_mapfile:D
759	_kernel_primitive:NN	\pdfmapline	\pdfTEX_mapline:D
760	_kernel_primitive:NN	\pdfnoligatures	\pdfTEX_noligatures:D
761	_kernel_primitive:NN	\pdfnormaldeviate	\pdfTEX_normaldeviate:D
762	_kernel_primitive:NN	\pdfpageheight	\pdfTEX_pageheight:D
763	_kernel_primitive:NN	\pdfpagewidth	\pdfTEX_pagewidth:D
764	_kernel_primitive:NN	\pdfpkmode	\pdfTEX_pkmode:D
765	_kernel_primitive:NN	\pdfpkresolution	\pdfTEX_pkresolution:D
766	_kernel_primitive:NN	\pdfprimitive	\pdfTEX_primitive:D
767	_kernel_primitive:NN	\pdfprotrudechars	\pdfTEX_protrudechars:D
768	_kernel_primitive:NN	\pdfpxdimen	\pdfTEX_pxdimen:D
769	_kernel_primitive:NN	\pdfrandomseed	\pdfTEX_randomseed:D
770	_kernel_primitive:NN	\pdfsavepos	\pdfTEX_savepos:D
771	_kernel_primitive:NN	\pdfstrcmp	\pdfTEX_strcmp:D
772	_kernel_primitive:NN	\pdfsetrandomseed	\pdfTEX_setrandomseed:D
773	_kernel_primitive:NN	\pdfshellescape	\pdfTEX_shellescape:D
774	_kernel_primitive:NN	\pdftracingfonts	\pdfTEX_tracingfonts:D
775	_kernel_primitive:NN	\pdfuniformdeviate	\pdfTEX_uniformdeviate:D

The version primitives are not related to PDF mode but are related to pdfTEX so retain the full prefix.

776	_kernel_primitive:NN	\pdfTEXbanner	\pdfTEX_pdfTEXbanner:D
777	_kernel_primitive:NN	\pdfTEXrevision	\pdfTEX_pdfTEXrevision:D

```
778 \__kernel_primitive:NN \pdfTeXversion \pdfTeXpdfTeXversion:D
```

These ones appear in pdfTeX but don't have pdf in the name at all. (\syncTeX is odd as it's really not from pdfTeX but from SyncTeX!)

```
779 \__kernel_primitive:NN \efcode \pdfTeXefcode:D
780 \__kernel_primitive:NN \ifincsname \pdfTeXifincsname:D
781 \__kernel_primitive:NN \leftmarginkern \pdfTeXleftmarginkern:D
782 \__kernel_primitive:NN \letterspacefont \pdfTeXletterspacefont:D
783 \__kernel_primitive:NN \lpcode \pdfTeXlpcode:D
784 \__kernel_primitive:NN \quitvmode \pdfTeXquitvmode:D
785 \__kernel_primitive:NN \rightmarginkern \pdfTeXrightmarginkern:D
786 \__kernel_primitive:NN \rpcode \pdfTeXrpcode:D
787 \__kernel_primitive:NN \syncTeX \pdfTeXsyncTeX:D
788 \__kernel_primitive:NN \tagcode \pdfTeXtagcode: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.

```
789 </initex | names | package>
790 {*initex | package>
791 \tex_long:D \tex_def:D \use_ii:nn #1#2 {#2}
792 \tex_long:D \tex_def:D \use_none:n #1 { }
793 \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
794 {
795 \etex_ifdefined:D #1
796 \tex_expandafter:D \use_ii:nn
797 \tex_fi:D
798 \use_none:n { \tex_global:D \tex_let:D #2 #1 }
799 {*initex>
800 \tex_global:D \tex_let:D #1 \tex_undefined:D
801 </initex>
802 }
803 </initex | package>
804 {*initex | names | package>
```

XeTeX-specific primitives. Note that XeTeX's \strcmp is handled earlier and is “rolled up” into \pdfstrcmp. With the exception of the version primitives these don't carry XeTeX through into the “base” name. A few cross-compatibility names which lack the pdf of the original are handled later.

```
805 \__kernel_primitive:NN \suppressfontnotfounderror \xetex_suppressfontnotfounderror:D
806 \__kernel_primitive:NN \XeTeXcharclass \xetex_charclass:D
807 \__kernel_primitive:NN \XeTeXcharglyph \xetex_charglyph:D
808 \__kernel_primitive:NN \XeTeXcountfeatures \xetex_countfeatures:D
809 \__kernel_primitive:NN \XeTeXcountglyphs \xetex_countglyphs:D
810 \__kernel_primitive:NN \XeTeXcountselectors \xetex_countselectors:D
811 \__kernel_primitive:NN \XeTeXcountvariations \xetex_countvariations:D
812 \__kernel_primitive:NN \XeTeXdefaultencoding \xetex_defaultencoding:D
813 \__kernel_primitive:NN \XeTeXdashbreakstate \xetex_dashbreakstate:D
814 \__kernel_primitive:NN \XeTeXfeaturecode \xetex_featurecode:D
815 \__kernel_primitive:NN \XeTeXfeaturename \xetex_featurename:D
816 \__kernel_primitive:NN \XeTeXfindfeaturebyname \xetex_findfeaturebyname:D
817 \__kernel_primitive:NN \XeTeXfindselectorbyname \xetex_findselectorbyname:D
818 \__kernel_primitive:NN \XeTeXfindvariationbyname \xetex_findvariationbyname:D
819 \__kernel_primitive:NN \XeTeXfirstfontchar \xetex_firstfontchar:D
```

820	<code>__kernel_primitive:NN \XeTeXfonttype</code>	<code>\xetex_fonttype:D</code>
821	<code>__kernel_primitive:NN \XeTeXgenerateactualtext</code>	<code>\xetex_generateactualtext:D</code>
822	<code>__kernel_primitive:NN \XeTeXglyph</code>	<code>\xetex_glyph:D</code>
823	<code>__kernel_primitive:NN \XeTeXglyphbounds</code>	<code>\xetex_glyphbounds:D</code>
824	<code>__kernel_primitive:NN \XeTeXglyphindex</code>	<code>\xetex_glyphindex:D</code>
825	<code>__kernel_primitive:NN \XeTeXglyphname</code>	<code>\xetex_glyphname:D</code>
826	<code>__kernel_primitive:NN \XeTeXinputencoding</code>	<code>\xetex_inputencoding:D</code>
827	<code>__kernel_primitive:NN \XeTeXinputnormalization</code>	<code>\xetex_inputnormalization:D</code>
828	<code>__kernel_primitive:NN \XeTeXinterchartokenstate</code>	<code>\xetex_interchartokenstate:D</code>
829	<code>__kernel_primitive:NN \XeTeXinterchartoks</code>	<code>\xetex_interchartoks:D</code>
830	<code>__kernel_primitive:NN \XeTeXisdefaultselector</code>	<code>\xetex_isdefaultselector:D</code>
831	<code>__kernel_primitive:NN \XeTeXisexclusivefeature</code>	<code>\xetex_isexclusivefeature:D</code>
832	<code>__kernel_primitive:NN \XeTeXlastfontchar</code>	<code>\xetex_lastfontchar:D</code>
833	<code>__kernel_primitive:NN \XeTeXlinebreakskip</code>	<code>\xetex_linebreakskip:D</code>
834	<code>__kernel_primitive:NN \XeTeXlinebreaklocale</code>	<code>\xetex_linebreaklocale:D</code>
835	<code>__kernel_primitive:NN \XeTeXlinebreakpenalty</code>	<code>\xetex_linebreakpenalty:D</code>
836	<code>__kernel_primitive:NN \XeTeXOTcountfeatures</code>	<code>\xetex_OTcountfeatures:D</code>
837	<code>__kernel_primitive:NN \XeTeXOTcountlanguages</code>	<code>\xetex_OTcountlanguages:D</code>
838	<code>__kernel_primitive:NN \XeTeXOTcountscripts</code>	<code>\xetex_OTcountscripts:D</code>
839	<code>__kernel_primitive:NN \XeTeXOTfeaturetag</code>	<code>\xetex_OTfeaturetag:D</code>
840	<code>__kernel_primitive:NN \XeTeXOTlanguagetag</code>	<code>\xetex_OTlanguagetag:D</code>
841	<code>__kernel_primitive:NN \XeTeXOTscripttag</code>	<code>\xetex_OTscripttag:D</code>
842	<code>__kernel_primitive:NN \XeTeXpdffile</code>	<code>\xetex_pdffile:D</code>
843	<code>__kernel_primitive:NN \XeTeXpdfpagecount</code>	<code>\xetex_pdfpagecount:D</code>
844	<code>__kernel_primitive:NN \XeTeXpicfile</code>	<code>\xetex_picfile:D</code>
845	<code>__kernel_primitive:NN \XeTeXselectorname</code>	<code>\xetex_selectorname:D</code>
846	<code>__kernel_primitive:NN \XeTeXtracingfonts</code>	<code>\xetex_tracingfonts:D</code>
847	<code>__kernel_primitive:NN \XeTeXupwardsmode</code>	<code>\xetex_upwardsmode:D</code>
848	<code>__kernel_primitive:NN \XeTeXuseglyphmetrics</code>	<code>\xetex_useglyphmetrics:D</code>
849	<code>__kernel_primitive:NN \XeTeXvariation</code>	<code>\xetex_variation:D</code>
850	<code>__kernel_primitive:NN \XeTeXvariationdefault</code>	<code>\xetex_variationdefault:D</code>
851	<code>__kernel_primitive:NN \XeTeXvariationmax</code>	<code>\xetex_variationmax:D</code>
852	<code>__kernel_primitive:NN \XeTeXvariationmin</code>	<code>\xetex_variationmin:D</code>
853	<code>__kernel_primitive:NN \XeTeXvariationname</code>	<code>\xetex_variationname:D</code>

The version primitives retain XeTeX.

854	<code>__kernel_primitive:NN \XeTeXrevision</code>	<code>\xetex_XeTeXrevision:D</code>
855	<code>__kernel_primitive:NN \XeTeXversion</code>	<code>\xetex_XeTeXversion:D</code>

Primitives from pdfTeX that XeTeX renames: also helps with LuaTeX.

856	<code>__kernel_primitive:NN \ifprimitive</code>	<code>\pdfTEX_ifprimitive:D</code>
857	<code>__kernel_primitive:NN \primitive</code>	<code>\pdfTEX_primitive:D</code>
858	<code>__kernel_primitive:NN \shellescape</code>	<code>\pdfTEX_shellescape:D</code>

Primitives from LuaTeX, some of which have been ported back to XeTeX. Notice that `\expanded` was intended for pdfTeX 1.50 but as that was not released we call this a LuaTeX primitive.

859	<code>__kernel_primitive:NN \alignmark</code>	<code>\luaTeX_alignmark:D</code>
860	<code>__kernel_primitive:NN \aligntab</code>	<code>\luaTeX_aligntab:D</code>
861	<code>__kernel_primitive:NN \attribute</code>	<code>\luaTeX_attribute:D</code>
862	<code>__kernel_primitive:NN \attributedef</code>	<code>\luaTeX_attributedef:D</code>
863	<code>__kernel_primitive:NN \begincsname</code>	<code>\luaTeX_begincsname:D</code>
864	<code>__kernel_primitive:NN \catcodetable</code>	<code>\luaTeX_catcodetable:D</code>
865	<code>__kernel_primitive:NN \clearmarks</code>	<code>\luaTeX_clearmarks:D</code>
866	<code>__kernel_primitive:NN \crampeddisplaystyle</code>	<code>\luaTeX_crampeddisplaystyle:D</code>

867	_kernel_primitive:NN	\crampedscriptscriptstyle	\luatex_crampedscriptscriptstyle:D
868	_kernel_primitive:NN	\crampedscriptstyle	\luatex_crampedscriptstyle:D
869	_kernel_primitive:NN	\crampedtextstyle	\luatex_crampedtextstyle:D
870	_kernel_primitive:NN	\directlua	\luatex_directlua:D
871	_kernel_primitive:NN	\dviextension	\luatex_dviextension:D
872	_kernel_primitive:NN	\dvifedback	\luatex_dvifedback:D
873	_kernel_primitive:NN	\dvivariable	\luatex_dvivariable:D
874	_kernel_primitive:NN	\etoksapp	\luatex_etoksapp:D
875	_kernel_primitive:NN	\etokspre	\luatex_etokspre:D
876	_kernel_primitive:NN	\expanded	\luatex_expanded:D
877	_kernel_primitive:NN	\firstvalidlanguage	\luatex_firstvalidlanguage:D
878	_kernel_primitive:NN	\fontid	\luatex_fontid:D
879	_kernel_primitive:NN	\formatname	\luatex_formatname:D
880	_kernel_primitive:NN	\hjcode	\luatex_hjcode:D
881	_kernel_primitive:NN	\hpack	\luatex_hpack:D
882	_kernel_primitive:NN	\hyphenationbounds	\luatex_hyphenationbounds:D
883	_kernel_primitive:NN	\hyphenationmin	\luatex_hyphenationmin:D
884	_kernel_primitive:NN	\gleaders	\luatex_gleaders:D
885	_kernel_primitive:NN	\initcatcodetable	\luatex_initcatcodetable:D
886	_kernel_primitive:NN	\lastnamedcs	\luatex_lastnamedcs:D
887	_kernel_primitive:NN	\latelua	\luatex_latelua:D
888	_kernel_primitive:NN	\letcharcode	\luatex_letcharcode:D
889	_kernel_primitive:NN	\luaescapestring	\luatex_luaescapestring:D
890	_kernel_primitive:NN	\luafunction	\luatex_luafunction:D
891	_kernel_primitive:NN	\luatexbanner	\luatex_luatexbanner:D
892	_kernel_primitive:NN	\luatexdatestamp	\luatex_luatexdatestamp:D
893	_kernel_primitive:NN	\luatexrevision	\luatex_luatexrevision:D
894	_kernel_primitive:NN	\luatexversion	\luatex_luatexversion:D
895	_kernel_primitive:NN	\mathdisplayskipmode	\luatex_mathdisplayskipmode:D
896	_kernel_primitive:NN	\matheqnogapstep	\luatex_matheqnogapstep:D
897	_kernel_primitive:NN	\mathoption	\luatex_mathoption:D
898	_kernel_primitive:NN	\mathnolimitsmode	\luatex_mathnolimitsmode:D
899	_kernel_primitive:NN	\mathrulesfam	\luatex_mathrulesfam:D
900	_kernel_primitive:NN	\mathscriptsmode	\luatex_mathscriptsmode:D
901	_kernel_primitive:NN	\mathstyle	\luatex_mathstyle:D
902	_kernel_primitive:NN	\mathsurroundskip	\luatex_mathsurroundskip:D
903	_kernel_primitive:NN	\nohrule	\luatex_nohrule:D
904	_kernel_primitive:NN	\nokerns	\luatex_nokerns:D
905	_kernel_primitive:NN	\noligs	\luatex_noligs:D
906	_kernel_primitive:NN	\nospaces	\luatex_nospaces:D
907	_kernel_primitive:NN	\novrule	\luatex_novrule:D
908	_kernel_primitive:NN	\outputbox	\luatex_outputbox:D
909	_kernel_primitive:NN	\pageleftoffset	\luatex_pageleftoffset:D
910	_kernel_primitive:NN	\pagetopoffset	\luatex_pagetopoffset:D
911	_kernel_primitive:NN	\pdfextension	\luatex_pdfextension:D
912	_kernel_primitive:NN	\pdffeedback	\luatex_pdffeedback:D
913	_kernel_primitive:NN	\pdfvariable	\luatex_pdfvariable:D
914	_kernel_primitive:NN	\postexhyphenchar	\luatex_postexhyphenchar:D
915	_kernel_primitive:NN	\posthyphenchar	\luatex_posthyphenchar:D
916	_kernel_primitive:NN	\preexhyphenchar	\luatex_preexhyphenchar:D
917	_kernel_primitive:NN	\prehyphenchar	\luatex_prehyphenchar:D
918	_kernel_primitive:NN	\savecatcodetable	\luatex_savecatcodetable:D
919	_kernel_primitive:NN	\scantextokens	\luatex_scantextokens:D
920	_kernel_primitive:NN	\setfontid	\luatex_setfontid:D

921	<code>__kernel_primitive:NN \shapemode</code>	<code>\luatex_shapemode:D</code>
922	<code>__kernel_primitive:NN \suppressifcsnameerror</code>	<code>\luatex_suppressifcsnameerror:D</code>
923	<code>__kernel_primitive:NN \suppresslongerror</code>	<code>\luatex_suppresslongerror:D</code>
924	<code>__kernel_primitive:NN \suppressmathparerror</code>	<code>\luatex_suppressmathparerror:D</code>
925	<code>__kernel_primitive:NN \suppressoutererror</code>	<code>\luatex_suppressoutererror:D</code>
926	<code>__kernel_primitive:NN \toksapp</code>	<code>\luatex_toksapp:D</code>
927	<code>__kernel_primitive:NN \tokspre</code>	<code>\luatex_tokspre:D</code>
928	<code>__kernel_primitive:NN \tpack</code>	<code>\luatex_tpack:D</code>
929	<code>__kernel_primitive:NN \vpack</code>	<code>\luatex_vpack:D</code>

Slightly more awkward are the directional primitives in LuaTeX. These come from Omega/Aleph, but we do not support those engines and so it seems most sensible to treat them as LuaTeX primitives for prefix purposes. One here is “new” but fits into the general set.

930	<code>__kernel_primitive:NN \bodydir</code>	<code>\luatex_bodydir:D</code>
931	<code>__kernel_primitive:NN \boxdir</code>	<code>\luatex_boxdir:D</code>
932	<code>__kernel_primitive:NN \leftghost</code>	<code>\luatex_leftghost:D</code>
933	<code>__kernel_primitive:NN \localbrokenpenalty</code>	<code>\luatex_localbrokenpenalty:D</code>
934	<code>__kernel_primitive:NN \localinterlinepenalty</code>	<code>\luatex_localinterlinepenalty:D</code>
935	<code>__kernel_primitive:NN \lcalleftbox</code>	<code>\luatex_lcalleftbox:D</code>
936	<code>__kernel_primitive:NN \lcalrightbox</code>	<code>\luatex_lcalrightbox:D</code>
937	<code>__kernel_primitive:NN \mathdir</code>	<code>\luatex_mathdir:D</code>
938	<code>__kernel_primitive:NN \linedir</code>	<code>\luatex_linedir:D</code>
939	<code>__kernel_primitive:NN \pagebottomoffset</code>	<code>\luatex_pagebottomoffset:D</code>
940	<code>__kernel_primitive:NN \pagedir</code>	<code>\luatex_pagedir:D</code>
941	<code>__kernel_primitive:NN \pagerightoffset</code>	<code>\luatex_pagerightoffset:D</code>
942	<code>__kernel_primitive:NN \pardir</code>	<code>\luatex_pardir:D</code>
943	<code>__kernel_primitive:NN \rightghost</code>	<code>\luatex_rightghost:D</code>
944	<code>__kernel_primitive:NN \textdir</code>	<code>\luatex_textdir:D</code>

Primitives from pdfTeX that LuaTeX renames.

945	<code>__kernel_primitive:NN \adjustspacing</code>	<code>\pdfTeX_adjustspacing:D</code>
946	<code>__kernel_primitive:NN \copyfont</code>	<code>\pdfTeX_copyfont:D</code>
947	<code>__kernel_primitive:NN \draftmode</code>	<code>\pdfTeX_draftmode:D</code>
948	<code>__kernel_primitive:NN \expandglyphsinfont</code>	<code>\pdfTeX_fontexpand:D</code>
949	<code>__kernel_primitive:NN \ifabsdim</code>	<code>\pdfTeX_ifabsdim:D</code>
950	<code>__kernel_primitive:NN \ifabsnum</code>	<code>\pdfTeX_ifabsnum:D</code>
951	<code>__kernel_primitive:NN \ignoreligaturesinfont</code>	<code>\pdfTeX_ignoreligaturesinfont:D</code>
952	<code>__kernel_primitive:NN \insertht</code>	<code>\pdfTeX_insertht:D</code>
953	<code>__kernel_primitive:NN \lastsavedboxresourceindex</code>	<code>\pdfTeX_pdflastxform:D</code>
954	<code>__kernel_primitive:NN \lastsavedimageresourceindex</code>	<code>\pdfTeX_pdflastximage:D</code>
955	<code>__kernel_primitive:NN \lastsavedimageresourcepages</code>	<code>\pdfTeX_pdflastximagepages:D</code>
956	<code>__kernel_primitive:NN \lastxpos</code>	<code>\pdfTeX_lastxpos:D</code>
957	<code>__kernel_primitive:NN \lastypos</code>	<code>\pdfTeX_lastypos:D</code>
958	<code>__kernel_primitive:NN \normaldeviate</code>	<code>\pdfTeX_normaldeviate:D</code>
959	<code>__kernel_primitive:NN \outputmode</code>	<code>\pdfTeX_pdfoutput:D</code>
960	<code>__kernel_primitive:NN \pageheight</code>	<code>\pdfTeX_pageheight:D</code>
961	<code>__kernel_primitive:NN \pagewidth</code>	<code>\pdfTeX_pagewidth:D</code>
962	<code>__kernel_primitive:NN \protrudechars</code>	<code>\pdfTeX_protrudechars:D</code>
963	<code>__kernel_primitive:NN \pxdimen</code>	<code>\pdfTeX_pxdimen:D</code>
964	<code>__kernel_primitive:NN \randomseed</code>	<code>\pdfTeX_randomseed:D</code>
965	<code>__kernel_primitive:NN \useboxresource</code>	<code>\pdfTeX_pdfrefxform:D</code>
966	<code>__kernel_primitive:NN \useimageresource</code>	<code>\pdfTeX_pdfrefximage:D</code>
967	<code>__kernel_primitive:NN \savepos</code>	<code>\pdfTeX_savepos:D</code>
968	<code>__kernel_primitive:NN \saveboxresource</code>	<code>\pdfTeX_pdfxform:D</code>

969	<code>__kernel_primitive:NN \saveimageresource</code>	<code>\pdfTeX_pdfximage:D</code>
970	<code>__kernel_primitive:NN \setrandomseed</code>	<code>\pdfTeX_setrandomseed:D</code>
971	<code>__kernel_primitive:NN \tracingfonts</code>	<code>\pdfTeX_tracingfonts:D</code>
972	<code>__kernel_primitive:NN \uniformdeviate</code>	<code>\pdfTeX_uniformdeviate:D</code>

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...`. To keep things somewhat clear we therefore prefix all of these as `\utex...` (introduced by a Unicode TeX engine) and drop `\U(math)` from the names. Where there is a related TeX90 primitive or where it really seems required we keep the `math` part of the name.

973	<code>__kernel_primitive:NN \Uchar</code>	<code>\utex_char:D</code>
974	<code>__kernel_primitive:NN \Ucharcat</code>	<code>\utex_charcat:D</code>
975	<code>__kernel_primitive:NN \Udelcode</code>	<code>\utex_delcode:D</code>
976	<code>__kernel_primitive:NN \Udelcodenum</code>	<code>\utex_delcodenum:D</code>
977	<code>__kernel_primitive:NN \Udelimiter</code>	<code>\utex_delimiter:D</code>
978	<code>__kernel_primitive:NN \Udelimiterover</code>	<code>\utex_delimiterover:D</code>
979	<code>__kernel_primitive:NN \Udelimiterunder</code>	<code>\utex_delimiterunder:D</code>
980	<code>__kernel_primitive:NN \Uhextensible</code>	<code>\utex_hextensible:D</code>
981	<code>__kernel_primitive:NN \Umathaccent</code>	<code>\utex_mathaccent:D</code>
982	<code>__kernel_primitive:NN \Umathaxis</code>	<code>\utex_mathaxis:D</code>
983	<code>__kernel_primitive:NN \Umathbinbinspacing</code>	<code>\utex_binbinspacing:D</code>
984	<code>__kernel_primitive:NN \Umathbinclosespacing</code>	<code>\utex_binclosespacing:D</code>
985	<code>__kernel_primitive:NN \Umathbininnerspacing</code>	<code>\utex_bininnerspacing:D</code>
986	<code>__kernel_primitive:NN \Umathbinopenspacing</code>	<code>\utex_binopenspacing:D</code>
987	<code>__kernel_primitive:NN \Umathbinopspacing</code>	<code>\utex_binopspacing:D</code>
988	<code>__kernel_primitive:NN \Umathbinordspacing</code>	<code>\utex_binordspacing:D</code>
989	<code>__kernel_primitive:NN \Umathbinpunctspacing</code>	<code>\utex_binpunctspacing:D</code>
990	<code>__kernel_primitive:NN \Umathbinrelspacing</code>	<code>\utex_binrelspacing:D</code>
991	<code>__kernel_primitive:NN \Umathchar</code>	<code>\utex_mathchar:D</code>
992	<code>__kernel_primitive:NN \Umathcharclass</code>	<code>\utex_mathcharclass:D</code>
993	<code>__kernel_primitive:NN \Umathchardef</code>	<code>\utex_mathchardef:D</code>
994	<code>__kernel_primitive:NN \Umathcharfam</code>	<code>\utex_mathcharfam:D</code>
995	<code>__kernel_primitive:NN \Umathcharnum</code>	<code>\utex_mathcharnum:D</code>
996	<code>__kernel_primitive:NN \Umathcharnumdef</code>	<code>\utex_mathcharnumdef:D</code>
997	<code>__kernel_primitive:NN \Umathcharslot</code>	<code>\utex_mathcharslot:D</code>
998	<code>__kernel_primitive:NN \Umathclosebinspacing</code>	<code>\utex_closebinspacing:D</code>
999	<code>__kernel_primitive:NN \Umathcloseclosespacing</code>	<code>\utex_closeclosespacing:D</code>
1000	<code>__kernel_primitive:NN \Umathcloseinnerspacing</code>	<code>\utex_closeinnerspacing:D</code>
1001	<code>__kernel_primitive:NN \Umathcloseopenspacing</code>	<code>\utex_closeopenspacing:D</code>
1002	<code>__kernel_primitive:NN \Umathcloseopspacing</code>	<code>\utex_closeopspacing:D</code>
1003	<code>__kernel_primitive:NN \Umathcloseordspacing</code>	<code>\utex_closeordspacing:D</code>
1004	<code>__kernel_primitive:NN \Umathclosepunctspacing</code>	<code>\utex_closepunctspacing:D</code>
1005	<code>__kernel_primitive:NN \Umathcloserelspacing</code>	<code>\utex_closerelspacing:D</code>
1006	<code>__kernel_primitive:NN \Umathcode</code>	<code>\utex_mathcode:D</code>
1007	<code>__kernel_primitive:NN \Umathcodenum</code>	<code>\utex_mathcodenum:D</code>
1008	<code>__kernel_primitive:NN \Umathconnectoroverlapmin</code>	<code>\utex_connectoroverlapmin:D</code>
1009	<code>__kernel_primitive:NN \Umathfractiondelsize</code>	<code>\utex_fractiondelsize:D</code>
1010	<code>__kernel_primitive:NN \Umathfractiondenomdown</code>	<code>\utex_fractiondenomdown:D</code>
1011	<code>__kernel_primitive:NN \Umathfractiondenomvgap</code>	<code>\utex_fractiondenomvgap:D</code>
1012	<code>__kernel_primitive:NN \Umathfractionnumup</code>	<code>\utex_fractionnumup:D</code>
1013	<code>__kernel_primitive:NN \Umathfractionnumvgap</code>	<code>\utex_fractionnumvgap:D</code>
1014	<code>__kernel_primitive:NN \Umathfractionrule</code>	<code>\utex_fractionrule:D</code>
1015	<code>__kernel_primitive:NN \Umathinnerbinspacing</code>	<code>\utex_innerbinspacing:D</code>

1016	_kernel_primitive:NN	\Umathinnerclosespacing	\utex_innerclosespacing:D
1017	_kernel_primitive:NN	\Umathinnerinnerspacing	\utex_innerinnerspacing:D
1018	_kernel_primitive:NN	\Umathinneropenspacing	\utex_inneropenspacing:D
1019	_kernel_primitive:NN	\Umathinnerordspacing	\utex_innerordspacing:D
1020	_kernel_primitive:NN	\Umathinnerordspacing	\utex_innerordspacing:D
1021	_kernel_primitive:NN	\Umathinnerpunctspacing	\utex_innerpunctspacing:D
1022	_kernel_primitive:NN	\Umathinnerrelspacing	\utex_innerrelspacing:D
1023	_kernel_primitive:NN	\Umathlimitabovebgap	\utex_limitabovebgap:D
1024	_kernel_primitive:NN	\Umathlimitabovekern	\utex_limitabovekern:D
1025	_kernel_primitive:NN	\Umathlimitabovevgap	\utex_limitabovevgap:D
1026	_kernel_primitive:NN	\Umathlimitbelowbgap	\utex_limitbelowbgap:D
1027	_kernel_primitive:NN	\Umathlimitbelowkern	\utex_limitbelowkern:D
1028	_kernel_primitive:NN	\Umathlimitbelowvgap	\utex_limitbelowvgap:D
1029	_kernel_primitive:NN	\Umathnolimitsubfactor	\utex_nolimitsubfactor:D
1030	_kernel_primitive:NN	\Umathnolimitsupfactor	\utex_nolimitsupfactor:D
1031	_kernel_primitive:NN	\Umathopbinspacing	\utex_opbinspacing:D
1032	_kernel_primitive:NN	\Umathopclosespacing	\utex_opclosespacing:D
1033	_kernel_primitive:NN	\Umathopenbinspacing	\utex_openbinspacing:D
1034	_kernel_primitive:NN	\Umathopenclosespacing	\utex_openclosespacing:D
1035	_kernel_primitive:NN	\Umathopeninnerspacing	\utex_openinnerspacing:D
1036	_kernel_primitive:NN	\Umathopenopenspacing	\utex_openopenspacing:D
1037	_kernel_primitive:NN	\Umathopenopspacing	\utex_openopspacing:D
1038	_kernel_primitive:NN	\Umathopenordspacing	\utex_openordspacing:D
1039	_kernel_primitive:NN	\Umathopenpunctspacing	\utex_openpunctspacing:D
1040	_kernel_primitive:NN	\Umathopenrelspacing	\utex_openrelspacing:D
1041	_kernel_primitive:NN	\Umathoperatorsiz	\utex_operatorsize:D
1042	_kernel_primitive:NN	\Umathopinnerspacing	\utex_opinnerspacing:D
1043	_kernel_primitive:NN	\Umathopopenspacing	\utex_opopenspacing:D
1044	_kernel_primitive:NN	\Umathopopspacing	\utex_opopspacing:D
1045	_kernel_primitive:NN	\Umathopordspacing	\utex_opordspacing:D
1046	_kernel_primitive:NN	\Umathoppunctspacing	\utex_oppunctspacing:D
1047	_kernel_primitive:NN	\Umathoprelspacing	\utex_oprelspacing:D
1048	_kernel_primitive:NN	\Umathordbinspacing	\utex_ordbinspacing:D
1049	_kernel_primitive:NN	\Umathordclosespacing	\utex_ordclosespacing:D
1050	_kernel_primitive:NN	\Umathordinnerspacing	\utex_ordinnerspacing:D
1051	_kernel_primitive:NN	\Umathordopenspacing	\utex_ordopenspacing:D
1052	_kernel_primitive:NN	\Umathordopspacing	\utex_ordopspacing:D
1053	_kernel_primitive:NN	\Umathordordspacing	\utex_ordordspacing:D
1054	_kernel_primitive:NN	\Umathordpunctspacing	\utex_ordpunctspacing:D
1055	_kernel_primitive:NN	\Umathordrelspacing	\utex_ordrelspacing:D
1056	_kernel_primitive:NN	\Umathoverbarkern	\utex_overbarkern:D
1057	_kernel_primitive:NN	\Umathoverbarrule	\utex_overbarrule:D
1058	_kernel_primitive:NN	\Umathoverbarvgap	\utex_overbarvgap:D
1059	_kernel_primitive:NN	\Umathoverdelimiterbgap	\utex_overdelimiterbgap:D
1060	_kernel_primitive:NN	\Umathoverdelimitervgap	\utex_overdelimitervgap:D
1061	_kernel_primitive:NN	\Umathpunctbinspacing	\utex_punctbinspacing:D
1062	_kernel_primitive:NN	\Umathpunctclosespacing	\utex_punctclosespacing:D
1063	_kernel_primitive:NN	\Umathpunctinnerspacing	\utex_punctinnerspacing:D
1064	_kernel_primitive:NN	\Umathpunctopenspacing	\utex_punctopenspacing:D
1065	_kernel_primitive:NN	\Umathpunctopspacing	\utex_punctopspacing:D
1066	_kernel_primitive:NN	\Umathpunctordspacing	\utex_punctordspacing:D
1067	_kernel_primitive:NN	\Umathpunctpunctspacing	\utex_punctpunctspacing:D
1068	_kernel_primitive:NN	\Umathpunctrelspacing	\utex_punctrelspacing:D
1069	_kernel_primitive:NN	\Umathquad	\utex_quad:D

1070	_kernel_primitive:NN	\Umathradicaldegreeafter	\utex_radicaldegreeafter:D
1071	_kernel_primitive:NN	\Umathradicaldegreebefore	\utex_radicaldegreebefore:D
1072	_kernel_primitive:NN	\Umathradicaldegreeraise	\utex_radicaldegreeraise:D
1073	_kernel_primitive:NN	\Umathradicalkern	\utex_radicalkern:D
1074	_kernel_primitive:NN	\Umathradicalrule	\utex_radicalrule:D
1075	_kernel_primitive:NN	\Umathradicalvgap	\utex_radicalvgap:D
1076	_kernel_primitive:NN	\Umathrelbinspacing	\utex_relbinspacing:D
1077	_kernel_primitive:NN	\Umathrelclosespacing	\utex_relclosespacing:D
1078	_kernel_primitive:NN	\Umathrelinnerspacing	\utex_relinnerspacing:D
1079	_kernel_primitive:NN	\Umathrelopenspacing	\utex_relopenspacing:D
1080	_kernel_primitive:NN	\Umathrelopspacing	\utex_relopspacing:D
1081	_kernel_primitive:NN	\Umathrelordspacing	\utex_relordspacing:D
1082	_kernel_primitive:NN	\Umathrelpunctspacing	\utex_relpunctspacing:D
1083	_kernel_primitive:NN	\Umathrelrelspacing	\utex_relrelspacing:D
1084	_kernel_primitive:NN	\Umathskewedfractionhgap	\utex_skewedfractionhgap:D
1085	_kernel_primitive:NN	\Umathskewedfractionvgap	\utex_skewedfractionvgap:D
1086	_kernel_primitive:NN	\Umathspaceafterscript	\utex_spaceafterscript:D
1087	_kernel_primitive:NN	\Umathstackdenomdown	\utex_stackdenomdown:D
1088	_kernel_primitive:NN	\Umathstacknumup	\utex_stacknumup:D
1089	_kernel_primitive:NN	\Umathstackvgap	\utex_stackvgap:D
1090	_kernel_primitive:NN	\Umathsubshiftdown	\utex_subshiftdown:D
1091	_kernel_primitive:NN	\Umathsubshiftdrop	\utex_subshiftdrop:D
1092	_kernel_primitive:NN	\Umathsubsupshiftdown	\utex_subsupshiftdown:D
1093	_kernel_primitive:NN	\Umathsubsupvgap	\utex_subsupvgap:D
1094	_kernel_primitive:NN	\Umathsubtopmax	\utex_subtopmax:D
1095	_kernel_primitive:NN	\Umathsupbottommin	\utex_supbottommin:D
1096	_kernel_primitive:NN	\Umathsupshiftdrop	\utex_supshiftdrop:D
1097	_kernel_primitive:NN	\Umathsupshiftup	\utex_supshiftup:D
1098	_kernel_primitive:NN	\Umathsupsubbottommax	\utex_supsubbottommax:D
1099	_kernel_primitive:NN	\Umathunderbarkern	\utex_underbarkern:D
1100	_kernel_primitive:NN	\Umathunderbarrule	\utex_underbarrule:D
1101	_kernel_primitive:NN	\Umathunderbarvgap	\utex_underbarvgap:D
1102	_kernel_primitive:NN	\Umathunderdelimiterbgap	\utex_underdelimiterbgap:D
1103	_kernel_primitive:NN	\Umathunderdelimitervgap	\utex_underdelimitervgap:D
1104	_kernel_primitive:NN	\Uoverdelimiter	\utex_overdelimiter:D
1105	_kernel_primitive:NN	\Uradical	\utex_radical:D
1106	_kernel_primitive:NN	\Uroot	\utex_root:D
1107	_kernel_primitive:NN	\Uskewed	\utex_skewed:D
1108	_kernel_primitive:NN	\Uskewedwithdelims	\utex_skewedwithdelims:D
1109	_kernel_primitive:NN	\Ustack	\utex_stack:D
1110	_kernel_primitive:NN	\Ustartdisplaymath	\utex_startdisplaymath:D
1111	_kernel_primitive:NN	\Ustartmath	\utex_startmath:D
1112	_kernel_primitive:NN	\Ustopdisplaymath	\utex_stopdisplaymath:D
1113	_kernel_primitive:NN	\Ustopmath	\utex_stopmath:D
1114	_kernel_primitive:NN	\Usubscript	\utex_subscript:D
1115	_kernel_primitive:NN	\Usuperscript	\utex_superscript:D
1116	_kernel_primitive:NN	\Uunderdelimiter	\utex_underdelimiter:D
1117	_kernel_primitive:NN	\Uvextensible	\utex_vextensible:D

Primitives from p_TE_X.

1118	_kernel_primitive:NN	\autospaceing	\ptex_autospaceing:D
1119	_kernel_primitive:NN	\autoxspaceing	\ptex_autoxspaceing:D
1120	_kernel_primitive:NN	\dtou	\ptex_dtou:D
1121	_kernel_primitive:NN	\euc	\ptex_euc:D
1122	_kernel_primitive:NN	\ifdbbox	\ptex_ifdbbox:D

1123	<code>__kernel_primitive:NN \ifddir</code>	<code>\ptex_ifddir:D</code>
1124	<code>__kernel_primitive:NN \ifmdir</code>	<code>\ptex_ifmdir:D</code>
1125	<code>__kernel_primitive:NN \iftbox</code>	<code>\ptex_iftbox:D</code>
1126	<code>__kernel_primitive:NN \iftdir</code>	<code>\ptex_iftdir:D</code>
1127	<code>__kernel_primitive:NN \ifybox</code>	<code>\ptex_ifybox:D</code>
1128	<code>__kernel_primitive:NN \ifydir</code>	<code>\ptex_ifydir:D</code>
1129	<code>__kernel_primitive:NN \inhibitglue</code>	<code>\ptex_inhibitglue:D</code>
1130	<code>__kernel_primitive:NN \inhibitxspcode</code>	<code>\ptex_inhibitxspcode:D</code>
1131	<code>__kernel_primitive:NN \jcharwidowpenalty</code>	<code>\ptex_jcharwidowpenalty:D</code>
1132	<code>__kernel_primitive:NN \jfam</code>	<code>\ptex_jfam:D</code>
1133	<code>__kernel_primitive:NN \jfont</code>	<code>\ptex_jfont:D</code>
1134	<code>__kernel_primitive:NN \jis</code>	<code>\ptex_jis:D</code>
1135	<code>__kernel_primitive:NN \kanjiskip</code>	<code>\ptex_kanjiskip:D</code>
1136	<code>__kernel_primitive:NN \kansuji</code>	<code>\ptex_kansuji:D</code>
1137	<code>__kernel_primitive:NN \kansujichar</code>	<code>\ptex_kansujichar:D</code>
1138	<code>__kernel_primitive:NN \kcatcode</code>	<code>\ptex_kcatcode:D</code>
1139	<code>__kernel_primitive:NN \kuten</code>	<code>\ptex_kuten:D</code>
1140	<code>__kernel_primitive:NN \noautospadding</code>	<code>\ptex_noautospadding:D</code>
1141	<code>__kernel_primitive:NN \noautoxspacing</code>	<code>\ptex_noautoxspacing:D</code>
1142	<code>__kernel_primitive:NN \postbreakpenalty</code>	<code>\ptex_postbreakpenalty:D</code>
1143	<code>__kernel_primitive:NN \prebreakpenalty</code>	<code>\ptex_prebreakpenalty:D</code>
1144	<code>__kernel_primitive:NN \showmode</code>	<code>\ptex_showmode:D</code>
1145	<code>__kernel_primitive:NN \sjis</code>	<code>\ptex_sjis:D</code>
1146	<code>__kernel_primitive:NN \tate</code>	<code>\ptex_tate:D</code>
1147	<code>__kernel_primitive:NN \tbaselineshift</code>	<code>\ptex_tbaselineshift:D</code>
1148	<code>__kernel_primitive:NN \tfont</code>	<code>\ptex_tfont:D</code>
1149	<code>__kernel_primitive:NN \xkanjiskip</code>	<code>\ptex_xkanjiskip:D</code>
1150	<code>__kernel_primitive:NN \xspcode</code>	<code>\ptex_xspcode:D</code>
1151	<code>__kernel_primitive:NN \ybaselineshift</code>	<code>\ptex_ybaselineshift:D</code>
1152	<code>__kernel_primitive:NN \yoko</code>	<code>\ptex_yoko:D</code>

Primitives from upT_EX.

1153	<code>__kernel_primitive:NN \disablecjktoken</code>	<code>\uptex_disablecjktoken:D</code>
1154	<code>__kernel_primitive:NN \enablecjktoken</code>	<code>\uptex_enablecjktoken:D</code>
1155	<code>__kernel_primitive:NN \forcecjktoken</code>	<code>\uptex_forcecjktoken:D</code>
1156	<code>__kernel_primitive:NN \kchar</code>	<code>\uptex_kchar:D</code>
1157	<code>__kernel_primitive:NN \kchardef</code>	<code>\uptex_kchardef:D</code>
1158	<code>__kernel_primitive:NN \kuten</code>	<code>\uptex_kuten:D</code>
1159	<code>__kernel_primitive:NN \ucs</code>	<code>\uptex_ucs:D</code>

End of the “just the names” part of the source.

```
1160 </initex | names | package>
1161 <*initex | package>
```

The job is done: close the group (using the primitive renamed!).

```
1162 \tex_endgroup:D
```

L^AT_EX 2_ε will have moved a few primitives, so these are sorted out. A convenient test for L^AT_EX 2_ε is the \@@end saved primitive.

1163	<code><*package></code>	
1164	<code>\etex_ifdefined:D \@@end</code>	
1165	<code>\tex_let:D \tex_end:D</code>	<code>\@@end</code>
1166	<code>\tex_let:D \tex_everydisplay:D</code>	<code>\frozen@everydisplay</code>
1167	<code>\tex_let:D \tex_everymath:D</code>	<code>\frozen@everymath</code>
1168	<code>\tex_let:D \tex_hyphen:D</code>	<code>\@@hyph</code>
1169	<code>\tex_let:D \tex_input:D</code>	<code>\@@input</code>

```

1170 \tex_let:D \tex_italiccorrection:D \@@italiccorr
1171 \tex_let:D \tex_underline:D \@@underline

```

Some tidying up is needed for `\(pdf)tracingfonts`. Newer LuaTeX has this simply as `\tracingfonts`, but that will have been overwritten by the L^AT_EX 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 L^AT_EX 2_ε users will have expl3 loaded by fontspec. (We follow the usual kernel convention that @@ is used for saved primitives.)

```

1172 \tex_let:D \pdfTeX_tracingfonts:D \tex_undefined:D
1173 \etex_ifdefined:D \pdftracingfonts
1174 \tex_let:D \pdfTeX_tracingfonts:D \pdftracingfonts
1175 \tex_else:D
1176 \etex_ifdefined:D \luatex_directlua:D
1177 \luatex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
1178 \tex_let:D \pdfTeX_tracingfonts:D \luatextracingfonts
1179 \tex_fi:D
1180 \tex_fi:D
1181 \tex_fi:D

```

That is also true for the LuaTeX primitives under L^AT_EX 2_ε (depending on the format-building date). There are a few primitives that get the right names anyway so are missing here!

```

1182 \etex_ifdefined:D \luatexsuppressfontnotfounderror
1183 \tex_let:D \luatex_alignmark:D \luatexalignmark
1184 \tex_let:D \luatex_aligntab:D \luatexaligntab
1185 \tex_let:D \luatex_attribute:D \luatexattribute
1186 \tex_let:D \luatex_attributedef:D \luatexattributedef
1187 \tex_let:D \luatex_catcodetable:D \luatexcatcodetable
1188 \tex_let:D \luatex_clearmarks:D \luatexclearmarks
1189 \tex_let:D \luatex_crampeddisplaystyle:D \luatexcrampeddisplaystyle
1190 \tex_let:D \luatex_crampedscriptscriptstyle:D \luatexcrampedscriptscriptstyle
1191 \tex_let:D \luatex_crampedscriptstyle:D \luatexcrampedscriptstyle
1192 \tex_let:D \luatex_crampedtextstyle:D \luatexcrampedtextstyle
1193 \tex_let:D \luatex_fontid:D \luatexfontid
1194 \tex_let:D \luatex_formatname:D \luatexformatname
1195 \tex_let:D \luatex_gleaders:D \luatexgleaders
1196 \tex_let:D \luatex_initcatcodetable:D \luatexinitcatcodetable
1197 \tex_let:D \luatex_latelua:D \luatexlatelua
1198 \tex_let:D \luatex_luaescapestring:D \luatexluaescapestring
1199 \tex_let:D \luatex_luafunction:D \luatexluafunction
1200 \tex_let:D \luatex_mathstyle:D \luatexmathstyle
1201 \tex_let:D \luatex_nokerns:D \luatexnokerns
1202 \tex_let:D \luatex_noligs:D \luatexnoligs
1203 \tex_let:D \luatex_outputbox:D \luatexoutputbox
1204 \tex_let:D \luatex_pageleftoffset:D \luatexpageleftoffset
1205 \tex_let:D \luatex_pagetopoffset:D \luatexpagetopoffset
1206 \tex_let:D \luatex_postexhyphenchar:D \luatexpostexhyphenchar
1207 \tex_let:D \luatex_posthyphenchar:D \luatexposthyphenchar
1208 \tex_let:D \luatex_preexhyphenchar:D \luatexpreexhyphenchar
1209 \tex_let:D \luatex_prehyphenchar:D \luatexprehyphenchar
1210 \tex_let:D \luatex_savecatcodetable:D \luatexsavecatcodetable
1211 \tex_let:D \luatex_scantextokens:D \luatexscantextokens
1212 \tex_let:D \luatex_suppressifcsnameerror:D \luatexsuppressifcsnameerror

```

```

1213 \tex_let:D \luatex_suppresslongerror:D \luatexsuppresslongerror
1214 \tex_let:D \luatex_suppressmathparerror:D \luatexsuppressmathparerror
1215 \tex_let:D \luatex_suppressoutererror:D \luatexsuppressoutererror
1216 \tex_let:D \utex_char:D \luatexUchar
1217 \tex_let:D \xetex_suppressfontnotfounderror:D \luatexsuppressfontnotfounderror

```

Which also covers those slightly odd ones.

```

1218 \tex_let:D \luatex_bodydir:D \luatexbodydir
1219 \tex_let:D \luatex_boxdir:D \luatexboxdir
1220 \tex_let:D \luatex_leftghost:D \luatexleftghost
1221 \tex_let:D \luatex_localbrokenpenalty:D \luatexlocalbrokenpenalty
1222 \tex_let:D \luatex_localinterlinepenalty:D \luatexlocalinterlinepenalty
1223 \tex_let:D \luatex_localleftbox:D \luatexlocalleftbox
1224 \tex_let:D \luatex_localrightbox:D \luatexlocalrightbox
1225 \tex_let:D \luatex_mathdir:D \luatexmathdir
1226 \tex_let:D \luatex_pagebottomoffset:D \luatexpagebottomoffset
1227 \tex_let:D \luatex_pagedir:D \luatexpagedir
1228 \tex_let:D \pdfTeX_pageheight:D \luatexpageheight
1229 \tex_let:D \luatex_pagerightoffset:D \luatexpagerightoffset
1230 \tex_let:D \pdfTeX_pagewidth:D \luatexpagewidth
1231 \tex_let:D \luatex_pardir:D \luatexpardir
1232 \tex_let:D \luatex_rightghost:D \luatexrightghost
1233 \tex_let:D \luatex_textdir:D \luatextextdir
1234 \tex_fi:D

```

Only pdfTeX and LuaTeX define \pdfmapfile and \pdfmapline: Tidy up the fact that some format-building processes leave a couple of questionable decisions about that!

```

1235 \tex_ifnum:D 0
1236 \etex_ifdefined:D \pdfTeX_pdfTeXversion:D 1 \tex_fi:D
1237 \etex_ifdefined:D \luatex_luatexversion:D 1 \tex_fi:D
1238 = 0 %
1239 \tex_let:D \pdfTeX_mapfile:D \tex_undefined:D
1240 \tex_let:D \pdfTeX_mapline:D \tex_undefined:D
1241 \tex_fi:D
1242 \</package>

```

Older XeTeX versions use \XeTeX as the prefix for the Unicode math primitives it knows. That is tidied up here (we support XeTeX versions from 0.9994 but this change was in 0.9999).

```

1243 \<{*initex | package>
1244 \etex_ifdefined:D \XeTeXdelcode
1245 \tex_let:D \utex_delcode:D \XeTeXdelcode
1246 \tex_let:D \utex_delcodenum:D \XeTeXdelcodenum
1247 \tex_let:D \utex_delimiter:D \XeTeXdelimiter
1248 \tex_let:D \utex_mathaccent:D \XeTeXmathaccent
1249 \tex_let:D \utex_mathchar:D \XeTeXmathchar
1250 \tex_let:D \utex_mathchardef:D \XeTeXmathchardef
1251 \tex_let:D \utex_mathcharnum:D \XeTeXmathcharnum
1252 \tex_let:D \utex_mathcharnumdef:D \XeTeXmathcharnumdef
1253 \tex_let:D \utex_mathcode:D \XeTeXmathcode
1254 \tex_let:D \utex_mathcodenum:D \XeTeXmathcodenum
1255 \tex_fi:D

```

Up to v0.80, LuaTeX defines the pdfTeX version data: rather confusing. Removing them means that \pdfTeX_pdfTeXversion:D is a marker for pdfTeX alone: useful in engine-dependent code later.

```

1256 \etex_ifdefined:D \luatex luatexversion:D
1257 \tex_let:D \pdfTeX_pdftexbanner:D \tex_undefined:D
1258 \tex_let:D \pdfTeX_pdftexrevision:D \tex_undefined:D
1259 \tex_let:D \pdfTeX_pdftexversion:D \tex_undefined:D
1260 \tex_fi:D
1261 </initex | package>

```

For ConT_EXt, 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 ConT_EXt.

```

1262 <*package>
1263 \etex_ifdefined:D \normalend
1264 \tex_let:D \tex_end:D \normalend
1265 \tex_let:D \tex_everyjob:D \normaleveryjob
1266 \tex_let:D \tex_input:D \normalinput
1267 \tex_let:D \tex_language:D \normallanguage
1268 \tex_let:D \tex_mathop:D \normalmathop
1269 \tex_let:D \tex_month:D \normalmonth
1270 \tex_let:D \tex_outer:D \normalouter
1271 \tex_let:D \tex_over:D \normalover
1272 \tex_let:D \tex_vcenter:D \normalvcenter
1273 \tex_let:D \etex_unexpanded:D \normalunexpanded
1274 \tex_let:D \luatex_expanded:D \normalexpanded
1275 \tex_fi:D
1276 \etex_ifdefined:D \normalitaliccorrection
1277 \tex_let:D \tex_hoffset:D \normalhoffset
1278 \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
1279 \tex_let:D \tex_voffset:D \normalvoffset
1280 \tex_let:D \etex_showtokens:D \normalshowtokens
1281 \tex_let:D \luatex_bodydir:D \spac_directions_normal_body_dir
1282 \tex_let:D \luatex_pagedir:D \spac_directions_normal_page_dir
1283 \tex_fi:D
1284 \etex_ifdefined:D \normalleft
1285 \tex_let:D \tex_left:D \normalleft
1286 \tex_let:D \tex_middle:D \normalmiddle
1287 \tex_let:D \tex_right:D \normalright
1288 \tex_fi:D
1289 </package>
1290 </initex | package>

```

3 l3basics implementation

```

1291 <*initex | package>

```

3.1 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 do a few now, just to get started.⁵

⁵This 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.

<code>\if_true:</code>	Then some conditionals.	
<code>\if_false:</code>		<code>\tex_let:D \if_true: \tex_iftrue:D</code>
<code>\or:</code>		<code>\tex_let:D \if_false: \tex_iffalse:D</code>
<code>\else:</code>		<code>\tex_let:D \or: \tex_or:D</code>
<code>\fi:</code>		<code>\tex_let:D \else: \tex_else:D</code>
<code>\reverse_if:N</code>		<code>\tex_let:D \fi: \tex_fi:D</code>
<code>\if:w</code>		<code>\tex_let:D \reverse_if:N \tetex_unless:D</code>
<code>\if_charcode:w</code>		<code>\tex_let:D \if:w \tex_if:D</code>
<code>\if_catcode:w</code>		<code>\tex_let:D \if_charcode:w \tex_if:D</code>
<code>\if_meaning:w</code>		<code>\tex_let:D \if_catcode:w \tex_ifcat:D</code>
		<code>\tex_let:D \if_meaning:w \tex_ifx:D</code>

(End definition for `\if_true:` and others. These functions are documented on page 21.)

<code>\if_mode_math:</code>	\TeX lets us detect some if its modes.	
<code>\if_mode_horizontal:</code>		<code>\tex_let:D \if_mode_math: \tex_ifmmode:D</code>
<code>\if_mode_vertical:</code>		<code>\tex_let:D \if_mode_horizontal: \tex_ifhmode:D</code>
<code>\if_mode_inner:</code>		<code>\tex_let:D \if_mode_vertical: \tex_ifvmode:D</code>
		<code>\tex_let:D \if_mode_inner: \tex_ifinner:D</code>

(End definition for `\if_mode_math:` and others. These functions are documented on page 21.)

<code>\if_cs_exist:N</code>	Building csnames and testing if control sequences exist.	
<code>\if_cs_exist:w</code>		<code>\tex_let:D \if_cs_exist:N \tetex_ifdefined:D</code>
<code>\cs:w</code>		<code>\tex_let:D \if_cs_exist:w \tetex_ifcsname:D</code>
<code>\cs_end:</code>		<code>\tex_let:D \cs:w \tex_csname:D</code>
		<code>\tex_let:D \cs_end: \tex_endcsname:D</code>

(End definition for `\if_cs_exist:N` and others. These functions are documented on page 21.)

<code>\exp_after:wN</code>	The five <code>\exp_</code> functions are used in the <code>l3expan</code> module where they are described.	
<code>\exp_not:N</code>		<code>\tex_let:D \exp_after:wN \tex_expandafter:D</code>
<code>\exp_not:n</code>		<code>\tex_let:D \exp_not:N \tex_noexpand:D</code>
		<code>\tex_let:D \exp_not:n \tetex_unexpanded:D</code>
		<code>\tex_let:D \exp:w \tex_romannumeral:D</code>
		<code>\tex_chardef:D \exp_end: = 0 ~</code>

(End definition for `\exp_after:wN`, `\exp_not:N`, and `\exp_not:n`. These functions are documented on page 30.)

<code>\token_to_meaning:N</code>	Examining a control sequence or token.	
<code>\cs_meaning:N</code>		<code>\tex_let:D \token_to_meaning:N \tex_meaning:D</code>
		<code>\tex_let:D \cs_meaning:N \tex_meaning:D</code>

(End definition for `\token_to_meaning:N` and `\cs_meaning:N`. These functions are documented on page 52.)

<code>\tl_to_str:n</code>	Making strings.	
<code>\token_to_str:N</code>		<code>\tex_let:D \tl_to_str:n \tetex_detokenize:D</code>
		<code>\tex_let:D \token_to_str:N \tex_string:D</code>

(End definition for `\tl_to_str:n` and `\token_to_str:N`. These functions are documented on page 96.)

`\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:`

```

1319 \tex_let:D \scan_stop:          \tex_relax:D
1320 \tex_let:D \group_begin:        \tex_begingroup:D
1321 \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.)

`\if_int_compare:w` For integers.

`__int_to_roman:w`

```

1322 \tex_let:D \if_int_compare:w    \tex_ifnum:D
1323 \tex_let:D \__int_to_roman:w    \tex_romannumeral:D

```

(End definition for \if_int_compare:w and __int_to_roman:w. These functions are documented on page 73.)

`\group_insert_after:N` Adding material after the end of a group.

```

1324 \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`

```

1325 \tex_long:D \tex_def:D \exp_args:Nc #1#2
1326   { \exp_after:wN #1 \cs:w #2 \cs_end: }
1327 \tex_long:D \tex_def:D \exp_args:cc #1#2
1328   { \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 27.)

`\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.

```

1329 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
1330 \tex_long:D \tex_def:D \cs_meaning:c #1
1331   {
1332     \if_cs_exist:w #1 \cs_end:
1333       \exp_after:wN \use_i:nn
1334     \else:
1335       \exp_after:wN \use_ii:nn
1336     \fi:
1337     { \exp_args:Nc \cs_meaning:N {#1} }
1338     { \tl_to_str:n {undefined} }
1339   }
1340 \tex_let:D \token_to_meaning:c = \cs_meaning:c

```

(End definition for \token_to_meaning:c, \token_to_str:c, and \cs_meaning:c. These functions are documented on page 52.)

3.2 Defining some constants

`\c_zero` We need the constant `\c_zero` 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!

```
1341 \tex_chardef:D \c_zero      = 0 ~
```

(End definition for `\c_zero`. This variable is documented on page 72.)

`\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`.

```
1342 \etex_ifdefined:D \luatex luatexversion:D
1343 \tex_chardef:D \c_max_register_int = 65 535 ~
1344 \tex_else:D
1345 \tex_mathchardef:D \c_max_register_int = 32 767 ~
1346 \tex_fi:D
```

(End definition for `\c_max_register_int`. This variable is documented on page 72.)

3.3 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

`\cs_set_nopar:Npn` All assignment functions in L^AT_EX3 should be naturally protected; after all, the T_EX primitives for assignments are and it can be a cause of problems if others aren't.

```
\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
1347 \tex_let:D \cs_set_nopar:Npn          \tex_def:D
1348 \tex_let:D \cs_set_nopar:Npx          \tex_edef:D
1349 \etex_protected:D \tex_long:D \tex_def:D \cs_set:Npn
1350   { \tex_long:D \tex_def:D }
1351 \etex_protected:D \tex_long:D \tex_def:D \cs_set:Npx
1352   { \tex_long:D \tex_edef:D }
1353 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npn
1354   { \etex_protected:D \tex_def:D }
1355 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npx
1356   { \etex_protected:D \tex_edef:D }
1357 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npn
1358   { \etex_protected:D \tex_long:D \tex_def:D }
1359 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npx
1360   { \etex_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` Global versions of the above functions.

```
\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
1361 \tex_let:D \cs_gset_nopar:Npn          \tex_gdef:D
1362 \tex_let:D \cs_gset_nopar:Npx          \tex_xdef:D
1363 \cs_set_protected:Npn \cs_gset:Npn
1364   { \tex_long:D \tex_gdef:D }
1365 \cs_set_protected:Npn \cs_gset:Npx
1366   { \tex_long:D \tex_xdef:D }
1367 \cs_set_protected:Npn \cs_gset_protected_nopar:Npn
1368   { \etex_protected:D \tex_gdef:D }
1369 \cs_set_protected:Npn \cs_gset_protected_nopar:Npx
1370   { \etex_protected:D \tex_xdef:D }
```

```

1371 \cs_set_protected:Npn \cs_gset_protected:Npn
1372   { \etex_protected:D \tex_long:D \tex_gdef:D }
1373 \cs_set_protected:Npn \cs_gset_protected:Npx
1374   { \etex_protected:D \tex_long:D \tex_xdef:D }

```

(End definition for `\cs_gset_nopar:Npn` and others. These functions are documented on page 12.)

3.4 Selecting tokens

`\l__exp_internal_tl` Scratch token list variable for `l3expan`, used by `\use:x`, used in defining conditionals. We don't use `tl` methods because `l3basics` is loaded earlier.

```

1375 \cs_set_nopar:Npn \l__exp_internal_tl { }

```

(End definition for `\l__exp_internal_tl`.)

`\use:c` This macro grabs its argument and returns a csname from it.

```

1376 \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_tl` which will be set up in `l3expan`.

```

1377 \cs_set_protected:Npn \use:x #1
1378   {
1379     \cs_set_nopar:Npx \l__exp_internal_tl {#1}
1380     \l__exp_internal_tl
1381   }

```

(End definition for `\use:x`. This function is documented on page 18.)

`\use:n` These macros grab their arguments and returns them back to the input (with outer braces removed).

```

\use:nnn 1382 \cs_set:Npn \use:n #1 {#1}
\use:nnnn 1383 \cs_set:Npn \use:nn #1#2 {#1#2}
1384 \cs_set:Npn \use:nnn #1#2#3 {#1#2#3}
1385 \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 17.)

`\use_i:nn` The equivalent to L^AT_EX 2_ε's `\@firstoftwo` and `\@secondoftwo`.

```

\use_ii:nn 1386 \cs_set:Npn \use_i:nn #1#2 {#1}
1387 \cs_set:Npn \use_ii:nn #1#2 {#2}

```

(End definition for `\use_i:nn` and `\use_ii:nn`. These functions are documented on page 17.)

`\use_i:nnn` We also need something for picking up arguments from a longer list.

```

\use_ii:nnn 1388 \cs_set:Npn \use_i:nnn #1#2#3 {#1}
\use_iii:nnn 1389 \cs_set:Npn \use_ii:nnn #1#2#3 {#2}
\use_i_ii:nnn 1390 \cs_set:Npn \use_iii:nnn #1#2#3 {#3}
\use_i:nnnn 1391 \cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
\use_ii:nnnn 1392 \cs_set:Npn \use_i:nnnn #1#2#3#4 {#1}
\use_iii:nnnn 1393 \cs_set:Npn \use_ii:nnnn #1#2#3#4 {#2}
\use_iv:nnnn 1394 \cs_set:Npn \use_iii:nnnn #1#2#3#4 {#3}
1395 \cs_set:Npn \use_iv:nnnn #1#2#3#4 {#4}

```


(End definition for `\use_i:nnn` and others. These functions are documented on page 18.)

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.
`\use_none_delimit_by_q_stop:w`
`\use_none_delimit_by_q_recursion_stop:w`

```

1396 \cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
1397 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
1398 \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 19.)

`\use_i_delimit_by_q_nil: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.
`\use_i_delimit_by_q_stop:nw`
`\use_i_delimit_by_q_recursion_stop:nw`

```

1399 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
1400 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
1401 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw #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 19.)

3.5 Gobbling tokens from input

`\use_none:n` 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:nnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once will take care of gobbling all the tokens in one go.

```

1402 \cs_set:Npn \use_none:n #1 { }
1403 \cs_set:Npn \use_none:nn #1#2 { }
1404 \cs_set:Npn \use_none:nnn #1#2#3 { }
1405 \cs_set:Npn \use_none:nnnn #1#2#3#4 { }
1406 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5 { }
1407 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6 { }
1408 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7 { }
1409 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8 { }
1410 \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 18.)

3.6 Conditional processing and definitions

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 T_EX 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 T_EX 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 T_EX programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:` The idea here is that `\exp:w` will expand fully any `\else:` and the `\fi:` that are waiting to be discarded, before reaching the `\exp_end:` which will leave the expansion null. 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.

```

1411 \cs_set:Npn \prg_return_true:
1412   { \exp_after:wN \use_i:nn \exp:w }
1413 \cs_set:Npn \prg_return_false:
1414   { \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 will work.

(End definition for \prg_return_true: and \prg_return_false:. These functions are documented on page 36.)

`\prg_set_conditional:Npnn` 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 `{\langle name \rangle}` `{\langle signature \rangle}` `\langle boolean \rangle` `{\langle set or new \rangle}` `{\langle maybe protected \rangle}` `{\langle parameters \rangle}` `{TF, ...}` `{\langle code \rangle}` to the auxiliary function responsible for defining all conditionals.

```

1415 \cs_set_protected:Npn \prg_set_conditional:Npnn
1416   { \prg_generate_conditional_parm:nnNpnn { set } { } }
1417 \cs_set_protected:Npn \prg_new_conditional:Npnn
1418   { \prg_generate_conditional_parm:nnNpnn { new } { } }
1419 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn
1420   { \prg_generate_conditional_parm:nnNpnn { set } { _protected } }
1421 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn
1422   { \prg_generate_conditional_parm:nnNpnn { new } { _protected } }
1423 \cs_set_protected:Npn \prg_generate_conditional_parm:nnNpnn #1#2#3#4#
1424   {
1425     \cs_split_function:NN #3 \prg_generate_conditional:nnNnnnnn
1426     {#1} {#2} {#4}
1427   }

```

(End definition for \prg_set_conditional:Npnn and others. These functions are documented on page 34.)

`\prg_set_conditional:Nnn` 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, ...}` `{\langle code \rangle}` to the auxiliary function responsible for defining all conditionals. If

the *signature* has more than 9 letters, the definition is aborted since T_EX macros have at most 9 arguments. The erroneous case where the function name contains no colon is captured later.

```

1428 \cs_set_protected:Npn \prg_set_conditional:Nnn
1429   { \prg_generate_conditional_count:nnNnn { set } { } }
1430 \cs_set_protected:Npn \prg_new_conditional:Nnn
1431   { \prg_generate_conditional_count:nnNnn { new } { } }
1432 \cs_set_protected:Npn \prg_set_protected_conditional:Nnn
1433   { \prg_generate_conditional_count:nnNnn { set } { _protected } }
1434 \cs_set_protected:Npn \prg_new_protected_conditional:Nnn
1435   { \prg_generate_conditional_count:nnNnn { new } { _protected } }
1436 \cs_set_protected:Npn \prg_generate_conditional_count:nnNnn #1#2#3
1437   {
1438     \cs_split_function:NN #3 \prg_generate_conditional_count:nnNnnnn
1439     {#1} {#2}
1440   }
1441 \cs_set_protected:Npn \prg_generate_conditional_count:nnNnnnn #1#2#3#4#5
1442   {
1443     \cs_parm_from_arg_count:nnF
1444     { \prg_generate_conditional:nnNnnnn {#1} {#2} #3 {#4} {#5} }
1445     { \tl_count:n {#2} }
1446     {
1447       \msg_kernel_error:nxx { kernel } { bad-number-of-arguments }
1448       { \token_to_str:c { #1 : #2 } }
1449       { \tl_count:n {#2} }
1450       \use_none:nn
1451     }
1452   }

```

(End definition for `\prg_set_conditional:Nnn` and others. These functions are documented on page 34.)

`\prg_generate_conditional:nnNnnnn`
`\prg_generate_conditional:nnnnnnw`

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.

```

1453 \cs_set_protected:Npn \prg_generate_conditional:nnNnnnn #1#2#3#4#5#6#7#8
1454   {
1455     \if_meaning:w \c_false_bool #3
1456     \msg_kernel_error:nxx { kernel } { missing-colon }
1457     { \token_to_str:c {#1} }
1458     \exp_after:wN \use_none:nn
1459     \fi:
1460     \use:x
1461     {
1462       \exp_not:N \prg_generate_conditional:nnnnnnw
1463       \exp_not:n { {#4} {#5} {#1} {#2} {#6} {#8} }
1464       \tl_to_str:n {#7}
1465       \exp_not:n { , \q_recursion_tail , \q_recursion_stop }

```

```

1466     }
1467 }

```

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:nnnnnnn` cleans up. Otherwise, the error message is removed by the variant form.

```

1468 \cs_set_protected:Npn \__prg_generate_conditional:nnnnnnw #1#2#3#4#5#6#7 ,
1469 {
1470     \if_meaning:w \q_recursion_tail #7
1471     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1472     \fi:
1473     \use:c { __prg_generate_ #7 _form:wnnnnnn }
1474     \tl_if_empty:nF {#7}
1475     {
1476         \__msg_kernel_error:nnxx
1477         { kernel } { conditional-form-unknown }
1478         {#7} { \token_to_str:c { #3 : #4 } }
1479     }
1480     \use_none:nnnnnnn
1481     \q_stop
1482     {#1} {#2} {#3} {#4} {#5} {#6}
1483     \__prg_generate_conditional:nnnnnnw {#1} {#2} {#3} {#4} {#5} {#6}
1484 }

```

(End definition for `__prg_generate_conditional:nnNnnnnn` and `__prg_generate_conditional:nnnnnnw`.)

```

\__prg_generate_p_form:wnnnnnn
\__prg_generate_TF_form:wnnnnnn
\__prg_generate_T_form:wnnnnnn
\__prg_generate_F_form:wnnnnnn

```

How to generate the various forms. Those functions take the following arguments: 1: **set** or **new**, 2: empty or `_protected`, 3: function name 4: signature, 5: parameter text (or empty), 6: replacement. 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 variant 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.

```

1485 \cs_set_protected:Npn \__prg_generate_p_form:wnnnnnn
1486     #1 \q_stop #2#3#4#5#6#7
1487 {
1488     \if_meaning:w \scan_stop: #3 \scan_stop:
1489     \exp_after:wN \use_i:nn
1490     \else:
1491     \exp_after:wN \use_ii:nn
1492     \fi:
1493     {
1494         \exp_args:cc { cs_ #2 #3 :Npn } { #4 _p: #5 } #6
1495         { #7 \exp_end: \c_true_bool \c_false_bool }
1496     }
1497     {
1498         \__msg_kernel_error:nnx { kernel } { protected-predicate }
1499         { \token_to_str:c { #4 _p: #5 } }
1500     }
1501 }
1502 \cs_set_protected:Npn \__prg_generate_T_form:wnnnnnn
1503     #1 \q_stop #2#3#4#5#6#7

```

```

1504 {
1505   \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 T } #6
1506   { #7 \exp_end: \use:n \use_none:n }
1507 }
1508 \cs_set_protected:Npn \__prg_generate_F_form:wnnnnnn
1509   #1 \q_stop #2#3#4#5#6#7
1510 {
1511   \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 F } #6
1512   { #7 \exp_end: { } }
1513 }
1514 \cs_set_protected:Npn \__prg_generate_TF_form:wnnnnnn
1515   #1 \q_stop #2#3#4#5#6#7
1516 {
1517   \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 TF } #6
1518   { #7 \exp_end: }
1519 }

```

(End definition for __prg_generate_p_form:wnnnnnn and others.)

\prg_set_eq_conditional:NNn The setting-equal functions. Split both functions and feed $\{\langle name_1 \rangle\}$ $\{\langle signature_1 \rangle\}$ $\langle boolean_1 \rangle$ $\{\langle name_2 \rangle\}$ $\{\langle signature_2 \rangle\}$ $\langle boolean_2 \rangle$ $\langle copying\ function \rangle$ $\langle conditions \rangle$, $\backslash q_recursion_tail$, $\backslash q_recursion_stop$ to a first auxiliary.

```

1520 \cs_set_protected:Npn \prg_set_eq_conditional:NNn
1521 { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
1522 \cs_set_protected:Npn \prg_new_eq_conditional:NNn
1523 { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
1524 \cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
1525 {
1526   \use:x
1527   {
1528     \exp_not:N \__prg_set_eq_conditional:nnNnnNW
1529     \__cs_split_function:NN #2 \prg_do_nothing:
1530     \__cs_split_function:NN #3 \prg_do_nothing:
1531     \exp_not:N #1
1532     \tl_to_str:n {#4}
1533     \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
1534   }
1535 }

```

(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 35.)

$\backslash _prg_set_eq_conditional:nnNnnNW$ 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.

```

1536 \cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNW #1#2#3#4#5#6
1537 {
1538   \if_meaning:w \c_false_bool #3
1539   \_msg_kernel_error:nnx { kernel } { missing-colon }
1540   { \token_to_str:c {#1} }

```

```

1541     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1542 \fi:
1543 \if_meaning:w \c_false_bool #6
1544     \_msg_kernel_error:nxx { kernel } { missing-colon }
1545     { \token_to_str:c {#4} }
1546     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1547 \fi:
1548 \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
1549 }
1550 \cs_set_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
1551 {
1552     \if_meaning:w \q_recursion_tail #6
1553     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1554 \fi:
1555 \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
1556     \tl_if_empty:nF {#6}
1557     {
1558         \_msg_kernel_error:nxxx
1559         { kernel } { conditional-form-unknown }
1560         {#6} { \token_to_str:c { #1 : #2 } }
1561     }
1562     \use_none:nnnnnn
1563     \q_stop
1564     #5 {#1} {#2} {#3} {#4}
1565     \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
1566 }
1567 \cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \q_stop #2#3#4#5#6
1568 {
1569     \__chk_if_exist_cs:c { #5 _p : #6 }
1570     #2 { #3 _p : #4 } { #5 _p : #6 }
1571 }
1572 \cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \q_stop #2#3#4#5#6
1573 {
1574     \__chk_if_exist_cs:c { #5 : #6 TF }
1575     #2 { #3 : #4 TF } { #5 : #6 TF }
1576 }
1577 \cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \q_stop #2#3#4#5#6
1578 {
1579     \__chk_if_exist_cs:c { #5 : #6 T }
1580     #2 { #3 : #4 T } { #5 : #6 T }
1581 }
1582 \cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \q_stop #2#3#4#5#6
1583 {
1584     \__chk_if_exist_cs:c { #5 : #6 F }
1585     #2 { #3 : #4 F } { #5 : #6 F }
1586 }

```

(End definition for __prg_set_eq_conditional:nnnnNw 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
1587 \tex_chardef:D \c_true_bool = 1 ~
1588 \tex_chardef:D \c_false_bool = 0 ~

```

(End definition for `\c_true_bool` and `\c_false_bool`. These variables are documented on page 20.)

3.7 Dissecting a control sequence

```

\cs_to_str:N This converts a control sequence into the character string of its name, removing the
__cs_to_str:N leading escape character. This turns out to be a non-trivial matter as there are different
__cs_to_str:w 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_romannumeral: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_romannumeral:D` with `\c_zero`. 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` to the character 0. The initial `\tex_romannumeral: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_romannumeral:D`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```

1589 \cs_set:Npn \cs_to_str:N
1590 {

```

We implement the expansion scheme using `\tex_romannumeral:D` terminating it with `\c_zero` 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` so we make this dependency explicit.

```

1591 \tex_romannumeral:D
1592 \if:w \token_to_str:N \__cs_to_str:w \fi:
1593 \exp_after:wN \__cs_to_str:N \token_to_str:N
1594 }
1595 \cs_set:Npn \__cs_to_str:N #1 { \c_zero }
1596 \cs_set:Npn \__cs_to_str:w #1 \__cs_to_str:N
1597 { - \__int_value:w \fi: \exp_after:wN \c_zero }

```

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`. These functions are documented on page 17.)

`_cs_split_function:NN` 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 $\langle true \rangle$ or $\langle false \rangle$ is returned with $\langle true \rangle$ for when there is a colon in the function and $\langle false \rangle$ if there is not. Lastly, the second argument of `_cs_split_function:NN` is supposed to be a function taking three variables, one for name, one for signature, and one for the boolean. For example, `_cs_split_function:NN \foo_bar:cnx \use_i:nnn` as input becomes `\use_i:nnn {foo_bar} {cnx} \c_true_bool`.

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 `\q_mark`, then `\c_true_bool` as `#3`, and `#4` cleans up until `\q_stop`. Otherwise, the `#1` contains the function name and `\q_mark \c_true_bool`, `#2` is empty, `#3` is `\c_false_bool`, and `#4` cleans up. In both cases, `#5` is the $\langle processor \rangle$. The second auxiliary trims the trailing `\q_mark` from the function name if present (that is, if the original function had no colon).

```

1598 \cs_set:Npx \_cs_split_function:NN #1
1599 {
1600   \exp_not:N \exp_after:wN \exp_not:N \exp_after:wN
1601   \exp_not:N \exp_after:wN \exp_not:N \_cs_split_function_auxi:w
1602   \exp_not:N \cs_to_str:N #1 \exp_not:N \q_mark \c_true_bool
1603   \token_to_str:N : \exp_not:N \q_mark \c_false_bool
1604   \exp_not:N \q_stop
1605 }
1606 \use:x
1607 {
1608   \cs_set:Npn \exp_not:N \_cs_split_function_auxi:w
1609     ##1 \token_to_str:N : ##2 \exp_not:N \q_mark ##3##4 \exp_not:N \q_stop ##5
1610 }
1611 { \_cs_split_function_auxii:w #5 #1 \q_mark \q_stop {#2} #3 }
1612 \cs_set:Npn \_cs_split_function_auxii:w #1#2 \q_mark #3 \q_stop
1613 { #1 {#2} }

```

(End definition for `_cs_split_function:NN`, `_cs_split_function_auxi:w`, and `_cs_split_function_auxii:w`.)

`_cs_get_function_name:N` Simple wrappers.

```

\cs_get_function_signature:N
1614 \cs_set:Npn \_cs_get_function_name:N #1
1615 { \_cs_split_function:NN #1 \use_i:nnn }
1616 \cs_set:Npn \_cs_get_function_signature:N #1
1617 { \_cs_split_function:NN #1 \use_ii:nnn }

```

(End definition for `_cs_get_function_name:N` and `_cs_get_function_signature:N`.)

3.8 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` Two versions for checking existence. For the `N` form we firstly check for `\scan_stop:` and
`\cs_if_exist_p:c` then if it is in the hash table. There is no problem when inputting something like `\else:`
`\cs_if_exist:NTF` or `\fi:` as `TEX` will only ever skip input in case the token tested against is `\scan_stop:`.
`\cs_if_exist:cTF`

```

1618 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1619 {
1620   \if_meaning:w #1 \scan_stop:
1621   \prg_return_false:
1622   \else:
1623     \if_cs_exist:N #1
1624     \prg_return_true:
1625     \else:
1626       \prg_return_false:
1627     \fi:
1628   \fi:
1629 }
```

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.

```

1630 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1631 {
1632   \if_cs_exist:w #1 \cs_end:
1633   \exp_after:wN \use_i:nn
1634   \else:
1635     \exp_after:wN \use_ii:nn
1636   \fi:
1637   {
1638     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1639     \prg_return_false:
1640     \else:
1641       \prg_return_true:
1642     \fi:
1643   }
1644   \prg_return_false:
1645 }
```

(End definition for `\cs_if_exist:NTF`. This function is documented on page 20.)

`\cs_if_free_p:N` The logical reversal of the above.

```

1646 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
1647 {
1648   \if_meaning:w #1 \scan_stop:
1649   \prg_return_true:
1650   \else:
1651     \if_cs_exist:N #1
1652     \prg_return_false:

```

```

1653     \else:
1654       \prg_return_true:
1655     \fi:
1656   \fi:
1657 }
1658 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1659 {
1660   \if_cs_exist:w #1 \cs_end:
1661     \exp_after:wN \use_i:nn
1662   \else:
1663     \exp_after:wN \use_ii:nn
1664   \fi:
1665   {
1666     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1667     \prg_return_true:
1668   \else:
1669     \prg_return_false:
1670   \fi:
1671 }
1672 { \prg_return_true: }
1673 }

```

(End definition for `\cs_if_free:NTF`. This function is documented on page 20.)

`\cs_if_exist_use:N` 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.

```

1674 \cs_set:Npn \cs_if_exist_use:NTF #1#2
1675   { \cs_if_exist:NTF #1 { #1 #2 } }
1676 \cs_set:Npn \cs_if_exist_use:NF #1
1677   { \cs_if_exist:NTF #1 { #1 } }
1678 \cs_set:Npn \cs_if_exist_use:NT #1 #2
1679   { \cs_if_exist:NTF #1 { #1 #2 } { } }
1680 \cs_set:Npn \cs_if_exist_use:N #1
1681   { \cs_if_exist:NTF #1 { #1 } { } }
1682 \cs_set:Npn \cs_if_exist_use:cTF #1#2
1683   { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
1684 \cs_set:Npn \cs_if_exist_use:cF #1
1685   { \cs_if_exist:cTF {#1} { \use:c {#1} } }
1686 \cs_set:Npn \cs_if_exist_use:cT #1#2
1687   { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
1688 \cs_set:Npn \cs_if_exist_use:c #1
1689   { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }

```

(End definition for `\cs_if_exist_use:NTF`. This function is documented on page 16.)

3.9 Defining and checking (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 definitions here are only temporary, they will be redefined later on.

`\iow_log:x` We define a routine to write only to the log file. And a similar one for writing to both
`\iow_term:x` the log file and the terminal. These will be redefined later by `l3io`.

```
1690 \cs_set_protected:Npn \iow_log:x
1691   { \tex_immediate:D \tex_write:D -1 }
1692 \cs_set_protected:Npn \iow_term:x
1693   { \tex_immediate:D \tex_write:D 16 }
```

(End definition for `\iow_log:x` and `\iow_term:x`. These functions are documented on page 176.)

`__chk_log:x` This function is used to write some information to the log file in case the `log-function`
`__chk_suspend_log:` option is set. Otherwise its argument is ignored. Using this function rather than di-
`__chk_resume_log:` rectly using `\iow_log:x` allows for `__chk_suspend_log:` which disables such messages
until the matching `__chk_resume_log:`. These two commands are used to improve the
logging for complicated datatypes. They should come in pairs, which can be nested.
The function `\exp_not:o` is defined in `l3expan` later on but `__chk_suspend_log:` and
`__chk_resume_log:` are not used before that point.

```
1694 \*initex>
1695 \cs_set_protected:Npn \__chk_log:x { \use_none:n }
1696 \cs_set_protected:Npn \__chk_suspend_log: { }
1697 \cs_set_protected:Npn \__chk_resume_log: { }
1698 \*initex>
1699 \*package>
1700 \tex_ifodd:D \l@expl@log@functions@bool
1701   \cs_set_protected:Npn \__chk_log:x { \iow_log:x }
1702   \cs_set_protected:Npn \__chk_suspend_log:
1703     {
1704       \cs_set_protected:Npx \__chk_resume_log:
1705         {
1706           \cs_set_protected:Npn \__chk_resume_log:
1707             { \exp_not:o { \__chk_resume_log: } }
1708           \cs_set_protected:Npn \__chk_log:x
1709             { \exp_not:o { \__chk_log:x } }
1710         }
1711       \cs_set_protected:Npn \__chk_log:x { \use_none:n }
1712     }
1713   \cs_set_protected:Npn \__chk_resume_log: { }
1714 \else:
1715   \cs_set_protected:Npn \__chk_log:x { \use_none:n }
1716   \cs_set_protected:Npn \__chk_suspend_log: { }
1717   \cs_set_protected:Npn \__chk_resume_log: { }
1718 \fi:
1719 \*package>
```

(End definition for `__chk_log:x`, `__chk_suspend_log:`, and `__chk_resume_log:`.)

`_msg_kernel_error:nxxx` If an internal error occurs before L^AT_EX3 has loaded `l3msg` then the code should issue a
`_msg_kernel_error:nxx` usable if terse error message and halt. This can only happen if a coding error is made by
`_msg_kernel_error:nn` 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.

```
1720 \cs_set_protected:Npn \_msg_kernel_error:nxxx #1#2#3#4
```

```

1721 {
1722   \tex_newlinechar:D = '\^^J \tex_relax:D
1723   \tex_errmessage:D
1724   {
1725     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
1726     Argh,~internal~LaTeX3~error! ^^J ^^J
1727     Module ~ #1 , ~ message~name~"#2": ^^J
1728     Arguments~'#3'~and~'#4' ^^J ^^J
1729     This~is~one~for~The~LaTeX3~Project:~bailing~out
1730   }
1731   \tex_end:D
1732 }
1733 \cs_set_protected:Npn \_msg_kernel_error:nxx #1#2#3
1734 { \_msg_kernel_error:nxxx {#1} {#2} {#3} { } }
1735 \cs_set_protected:Npn \_msg_kernel_error:nn #1#2
1736 { \_msg_kernel_error:nxxx {#1} {#2} { } { } }

(End definition for \_msg_kernel_error:nxxx, \_msg_kernel_error:nxx, and \_msg_kernel_error:nn.)

```

\msg_line_context: Another one from l3msg which will be altered later.

```

1737 \cs_set:Npn \msg_line_context:
1738 { on~line~ \tex_the:D \tex_inputlineno:D }

(End definition for \msg_line_context:. This function is documented on page 152.)

```

__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 $\langle csname \rangle$ 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!

__chk_if_free_cs:c

```

1739 \cs_set_protected:Npn \__chk_if_free_cs:N #1
1740 {
1741   \cs_if_free:NF #1
1742   {
1743     \_msg_kernel_error:nxxx { kernel } { command-already-defined }
1744     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1745   }
1746 }
1747 \*package
1748 \tex_ifodd:D \l@expl@log@functions@bool
1749 \cs_set_protected:Npn \__chk_if_free_cs:N #1
1750 {
1751   \cs_if_free:NF #1
1752   {
1753     \_msg_kernel_error:nxxx { kernel } { command-already-defined }
1754     { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1755   }
1756   \__chk_log:x { Defining~\token_to_str:N #1~ \msg_line_context: }
1757 }
1758 \fi:
1759 \*package
1760 \cs_set_protected:Npn \__chk_if_free_cs:c
1761 { \exp_args:Nc \__chk_if_free_cs:N }

```

(End definition for __chk_if_free_cs:N.)

`__chk_if_exist_var:N` Create the checking function for variable definitions when the option is set.

```

1762 <*package>
1763 \tex_ifodd:D \l@expl@check@declarations@bool
1764 \cs_set_protected:Npn \__chk_if_exist_var:N #1
1765 {
1766   \cs_if_exist:NF #1
1767   {
1768     \__msg_kernel_error:nxx { check } { non-declared-variable }
1769     { \token_to_str:N #1 }
1770   }
1771 }
1772 \fi:
1773 </package>

```

(End definition for `__chk_if_exist_var:N`.)

`__chk_if_exist_cs:N` This function issues an error message when the control sequence in its argument does
`__chk_if_exist_cs:c` not exist.

```

1774 \cs_set_protected:Npn \__chk_if_exist_cs:N #1
1775 {
1776   \cs_if_exist:NF #1
1777   {
1778     \__msg_kernel_error:nxx { kernel } { command-not-defined }
1779     { \token_to_str:N #1 }
1780   }
1781 }
1782 \cs_set_protected:Npn \__chk_if_exist_cs:c
1783 { \exp_args:Nc \__chk_if_exist_cs:N }

```

(End definition for `__chk_if_exist_cs:N`.)

3.10 More new definitions

`\cs_new_nopar:Npn` Function which check that the control sequence is free before defining it.

```

\cs_new_nopar:Npx 1784 \cs_set:Npn \__cs_tmp:w #1#2
\cs_new:Npn        1785 {
\cs_new:Npx        1786   \cs_set_protected:Npn #1 ##1
\cs_new_protected_nopar:Npn 1787   {
\cs_new_protected_nopar:Npx 1788     \__chk_if_free_cs:N ##1
\cs_new_protected:Npn      1789     #2 ##1
\cs_new_protected:Npx      1790   }
\__cs_tmp:w          1791 }
\__cs_tmp:w \cs_new_nopar:Npn      1792 \cs_gset_nopar:Npn
\__cs_tmp:w \cs_new_nopar:Npx      1793 \cs_gset_nopar:Npx
\__cs_tmp:w \cs_new:Npn            1794 \cs_gset:Npn
\__cs_tmp:w \cs_new:Npx            1795 \cs_gset:Npx
\__cs_tmp:w \cs_new_protected_nopar:Npn 1796 \cs_gset_protected_nopar:Npn
\__cs_tmp:w \cs_new_protected_nopar:Npx 1797 \cs_gset_protected_nopar:Npx
\__cs_tmp:w \cs_new_protected:Npn    1798 \cs_gset_protected:Npn
\__cs_tmp:w \cs_new_protected:Npx    1799 \cs_gset_protected:Npx

```

(End definition for `\cs_new_nopar:Npn` and others. These functions are documented on page 11.)

`\cs_set_nopar:cpn` 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` $\langle string \rangle \langle rep-text \rangle$ will turn $\langle string \rangle$ into a `csname` and then assign $\langle rep-text \rangle$ to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

1800 \cs_set:Npn \__cs_tmp:w #1#2
1801   { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
1802 \__cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1803 \__cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
1804 \__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1805 \__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1806 \__cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1807 \__cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End definition for `\cs_set_nopar:cpn` and others. These functions are documented on page 11.)

`\cs_set:cpn` Variants of the `\cs_set:Npn` versions which make a `csname` out of the first arguments.
`\cs_set:cpx` We may also do this globally.

```

1808 \__cs_tmp:w \cs_set:cpn \cs_set:Npn
1809 \__cs_tmp:w \cs_set:cpx \cs_set:Npx
1810 \__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
1811 \__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1812 \__cs_tmp:w \cs_new:cpn \cs_new:Npn
1813 \__cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End definition for `\cs_set:cpn` and others. These functions are documented on page 11.)

`\cs_set_protected_nopar:cpn` Variants of the `\cs_set_protected_nopar:Npn` versions which make a `csname` out of the first arguments. We may also do this globally.

```

1814 \__cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
1815 \__cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
1816 \__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
1817 \__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1818 \__cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1819 \__cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npx

```

(End definition for `\cs_set_protected_nopar:cpn` and others. These functions are documented on page 12.)

`\cs_set_protected:cpn` Variants of the `\cs_set_protected:Npn` versions which make a `csname` out of the first arguments. We may also do this globally.

```

1820 \__cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
1821 \__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
1822 \__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
1823 \__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
1824 \__cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
1825 \__cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

```

(End definition for `\cs_set_protected:cpn` and others. These functions are documented on page 11.)

3.11 Copying definitions

`\cs_set_eq:NN` These macros allow us to copy the definition of a control sequence to another control sequence.

`\cs_set_eq:cN` The = sign allows us to define funny char tokens like = itself or `_` with this function.

`\cs_set_eq:Nc` For the definition of `\c_space_char{~}` to work we need the ~ after the =.

`\cs_set_eq:cc` `\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While

`\cs_gset_eq:NN` `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it

`\cs_gset_eq:cN` long in order to throw an “already defined” error rather than “runaway argument”.

`\cs_gset_eq:Nc`

`\cs_gset_eq:cc`

```

1826 \cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
1827 \cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
1828 \cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
1829 \cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
1830 \cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }
1831 \cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }
1832 \cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1833 \cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }
1834 \cs_new_protected:Npn \cs_new_eq:NN #1
1835 {
1836   \__chk_if_free_cs:N #1
1837   \tex_global:D \cs_set_eq:NN #1
1838 }
1839 \cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1840 \cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
1841 \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.)

3.12 undefining functions

`\cs_undefine:NN` The following function is used to free the main memory from the definition of some

`\cs_undefine:c` 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.

```

1842 \cs_new_protected:Npn \cs_undefine:N #1
1843 { \cs_gset_eq:NN #1 \tex_undefined:D }
1844 \cs_new_protected:Npn \cs_undefine:c #1
1845 {
1846   \if_cs_exist:w #1 \cs_end:
1847   \exp_after:wN \use:n
1848   \else:
1849   \exp_after:wN \use_none:n
1850   \fi:
1851   { \cs_gset_eq:cN {#1} \tex_undefined:D }
1852 }

```

(End definition for `\cs_undefine:N`. This function is documented on page 15.)

3.13 Generating parameter text from argument count

`_cs_parm_from_arg_count:nnF` \LaTeX 3 provides shorthands to define control sequences and conditionals with a simple

`_cs_parm_from_arg_count_test:nnF` 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.

```

1853 \cs_set_protected:Npn \__cs_parm_from_arg_count:nnF #1#2
1854 {
1855   \exp_args:Nx \__cs_parm_from_arg_count_test:nnF
1856   {
1857     \exp_after:wN \exp_not:n
1858     \if_case:w \__int_eval:w #2 \__int_eval_end:
1859       { }
1860       \or: { ##1 }
1861       \or: { ##1##2 }
1862       \or: { ##1##2##3 }
1863       \or: { ##1##2##3##4 }
1864       \or: { ##1##2##3##4##5 }
1865       \or: { ##1##2##3##4##5##6 }
1866       \or: { ##1##2##3##4##5##6##7 }
1867       \or: { ##1##2##3##4##5##6##7##8 }
1868       \or: { ##1##2##3##4##5##6##7##8##9 }
1869       \else: { \c_false_bool }
1870     \fi:
1871   }
1872   {#1}
1873 }
1874 \cs_set_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
1875 {
1876   \if_meaning:w \c_false_bool #1
1877   \exp_after:wN \use_ii:nn
1878   \else:
1879   \exp_after:wN \use_i:nn
1880   \fi:
1881   { #2 {#1} }
1882 }

```

(End definition for `__cs_parm_from_arg_count:nnF` and `__cs_parm_from_arg_count_test:nnF`.)

3.14 Defining functions from a given number of arguments

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.

```

1883 \cs_new:Npn \__cs_count_signature:N #1
1884 { \int_eval:n { \__cs_split_function:NN #1 \__cs_count_signature:nnN } }
1885 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
1886 {
1887   \if_meaning:w \c_true_bool #3
1888   \tl_count:n {#2}
1889   \else:
1890   -1

```



```

1891     \fi:
1892   }
1893   \cs_new:Npn \__cs_count_signature:c
1894     { \exp_args:Nc \__cs_count_signature:N }

```

(End definition for `__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:Ncnn

```

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

```

1895 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
1896 {
1897   \__cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
1898   {
1899     \_msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
1900     { \token_to_str:N #1 } { \int_eval:n {#3} }
1901     \use_none:n
1902   }
1903   {#4}
1904 }

```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```

1905 \cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
1906   { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
1907 \cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
1908   { \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.)

3.15 Using the signature to define functions

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}`.

```

\cs_set:Nn
\cs_set:Nx

```

We want to define `\cs_set:Nn` as

```

\cs_set_protected:Npn \cs_set:Nn #1#2
{
  \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn
  { \__cs_count_signature:N #1 } {#2}
}

```

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.

```

1909 \cs_set:Npn \__cs_tmp:w #1#2#3
1910 {
1911   \cs_new_protected:cpx { cs_ #1 : #2 }

```

```

\cs_new:Nn
\cs_new:Nx

```

275

```

\cs_new_nopar:Nn
\cs_new_nopar:Nx

```

```

\cs_new_protected:Nn
\cs_new_protected:Nx

```

```

\cs_new_protected_nopar:Nn

```

```

1912     {
1913         \exp_not:N \__cs_generate_from_signature:NNn
1914         \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
1915     }
1916 }
1917 \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
1918 {
1919     \__cs_split_function:NN #2 \__cs_generate_from_signature:nnNNNn
1920     #1 #2
1921 }
1922 \cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
1923 {
1924     \bool_if:NTF #3
1925     {
1926         \str_if_eq_x:nnF { }
1927         { \tl_map_function:nN {#2} \__cs_generate_from_signature:n }
1928         {
1929             \__msg_kernel_error:nnx { kernel } { non-base-function }
1930             { \token_to_str:N #5 }
1931         }
1932         \cs_generate_from_arg_count:NNnn
1933         #5 #4 { \tl_count:n {#2} } {#6}
1934     }
1935     {
1936         \__msg_kernel_error:nnx { kernel } { missing-colon }
1937         { \token_to_str:N #5 }
1938     }
1939 }
1940 \cs_new:Npn \__cs_generate_from_signature:n #1
1941 {
1942     \if:w n #1 \else: \if:w N #1 \else:
1943     \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi: \fi:
1944 }

```

Then we define the 24 variants beginning with N.

```

1945 \__cs_tmp:w { set } { Nn } { Npn }
1946 \__cs_tmp:w { set } { Nx } { Npx }
1947 \__cs_tmp:w { set_nopar } { Nn } { Npn }
1948 \__cs_tmp:w { set_nopar } { Nx } { Npx }
1949 \__cs_tmp:w { set_protected } { Nn } { Npn }
1950 \__cs_tmp:w { set_protected } { Nx } { Npx }
1951 \__cs_tmp:w { set_protected_nopar } { Nn } { Npn }
1952 \__cs_tmp:w { set_protected_nopar } { Nx } { Npx }
1953 \__cs_tmp:w { gset } { Nn } { Npn }
1954 \__cs_tmp:w { gset } { Nx } { Npx }
1955 \__cs_tmp:w { gset_nopar } { Nn } { Npn }
1956 \__cs_tmp:w { gset_nopar } { Nx } { Npx }
1957 \__cs_tmp:w { gset_protected } { Nn } { Npn }
1958 \__cs_tmp:w { gset_protected } { Nx } { Npx }
1959 \__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
1960 \__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
1961 \__cs_tmp:w { new } { Nn } { Npn }
1962 \__cs_tmp:w { new } { Nx } { Npx }
1963 \__cs_tmp:w { new_nopar } { Nn } { Npn }
1964 \__cs_tmp:w { new_nopar } { Nx } { Npx }

```

```

1965 \__cs_tmp:w { new_protected } { Nn } { Npn }
1966 \__cs_tmp:w { new_protected } { Nx } { Npx }
1967 \__cs_tmp:w { new_protected_nopar } { Nn } { Npn }
1968 \__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
1969 \cs_set:Npn \__cs_tmp:w #1#2
1970 {
1971   \cs_new_protected:cpx { cs_ #1 : c #2 }
1972   {
1973     \exp_not:N \exp_args:Nc
1974     \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
1975   }
1976 }
1977 \__cs_tmp:w { set } { n }
1978 \__cs_tmp:w { set } { x }
1979 \__cs_tmp:w { set_nopar } { n }
1980 \__cs_tmp:w { set_nopar } { x }
1981 \__cs_tmp:w { set_protected } { n }
1982 \__cs_tmp:w { set_protected } { x }
1983 \__cs_tmp:w { set_protected_nopar } { n }
1984 \__cs_tmp:w { set_protected_nopar } { x }
1985 \__cs_tmp:w { gset } { n }
1986 \__cs_tmp:w { gset } { x }
1987 \__cs_tmp:w { gset_nopar } { n }
1988 \__cs_tmp:w { gset_nopar } { x }
1989 \__cs_tmp:w { gset_protected } { n }
1990 \__cs_tmp:w { gset_protected } { x }
1991 \__cs_tmp:w { gset_protected_nopar } { n }
1992 \__cs_tmp:w { gset_protected_nopar } { x }
1993 \__cs_tmp:w { new } { n }
1994 \__cs_tmp:w { new } { x }
1995 \__cs_tmp:w { new_nopar } { n }
1996 \__cs_tmp:w { new_nopar } { x }
1997 \__cs_tmp:w { new_protected } { n }
1998 \__cs_tmp:w { new_protected } { x }
1999 \__cs_tmp:w { new_protected_nopar } { n }
2000 \__cs_tmp:w { new_protected_nopar } { x }

```

(End definition for \cs_set:cn and others. These functions are documented on page 13.)

3.16 Checking control sequence equality

```

\cs_if_eq_p:NN Check if two control sequences are identical.
\cs_if_eq_p:cN
\cs_if_eq_p:Nc
\cs_if_eq_p:cc
\cs_if_eq:NNTF
\cs_if_eq:cNTF
\cs_if_eq:NcTF
\cs_if_eq:ccTF
2001 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
2002 {
2003   \if_meaning:w #1#2
2004   \prg_return_true: \else: \prg_return_false: \fi:
2005 }
2006 \cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
2007 \cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
2008 \cs_new:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }

```

```

2009 \cs_new:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
2010 \cs_new:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
2011 \cs_new:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
2012 \cs_new:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNT }
2013 \cs_new:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
2014 \cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
2015 \cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
2016 \cs_new:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
2017 \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 20.)

3.17 Diagnostic functions

`__kernel_register_show:N` Simply using the `\show` primitive does not allow for line-wrapping, so instead use `__-msg_show_variable:NNNnn` (defined in `l3msg`). This checks that the variable exists (using `\cs_if_exist:NTF`), then displays the third argument, namely $\>\sim\langle variable \rangle=\langle value \rangle$.

```

2018 \cs_new_protected:Npn \__kernel_register_show:N #1
2019 {
2020   \__msg_show_variable:NNNnn #1 \cs_if_exist:NTF ? { }
2021   { > ~ \token_to_str:N #1 = \tex_the:D #1 }
2022 }
2023 \cs_new_protected:Npn \__kernel_register_show:c
2024 { \exp_args:Nc \__kernel_register_show:N }

```

(End definition for `__kernel_register_show:N`.)

`\cs_show:N` 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 `\iow_wrap:nnnN` for line-wrapping. The `\cs_show:c` command converts its argument to a control sequence within a group to avoid showing `\relax` for undefined control sequences.

```

2025 \cs_new_protected:Npn \cs_show:N #1
2026 { \__msg_show_wrap:n { > ~ \token_to_str:N #1 = \cs_meaning:N #1 } }
2027 \cs_new_protected:Npn \cs_show:c
2028 { \group_begin: \exp_args:NNc \group_end: \cs_show:N }

```

(End definition for `\cs_show:N`. This function is documented on page 16.)

3.18 Doing nothing functions

`\prg_do_nothing:` This does not fit anywhere else!

```

2029 \cs_new_nopar:Npn \prg_do_nothing: { }

```

(End definition for `\prg_do_nothing:`. This function is documented on page 9.)

3.19 Breaking out of mapping functions

`__prg_break_point: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.

```

2030 \cs_new_eq:NN \__prg_break_point:Nn \use_ii:nn
2031 \cs_new:Npn \__prg_map_break:Nn #1#2#3 \__prg_break_point:Nn #4#5
2032 {
2033     #5
2034     \if_meaning:w #1 #4
2035     \exp_after:wN \use_iii:nnn
2036     \fi:
2037     \__prg_map_break:Nn #1 {#2}
2038 }

```

(End definition for `__prg_break_point:Nn` and `__prg_map_break:Nn`.)

`__prg_break_point:` Very simple analogues of `__prg_break_point:Nn` and `__prg_map_break:Nn`, for use
`__prg_break:` in fast short-term recursions which are not mappings, do not need to support nesting,
`__prg_break:n` and in which nothing has to be done at the end of the loop.

```

2039 \cs_new_eq:NN \__prg_break_point: \prg_do_nothing:
2040 \cs_new:Npn \__prg_break: #1 \__prg_break_point: { }
2041 \cs_new:Npn \__prg_break:n #1#2 \__prg_break_point: {#1}

```

(End definition for `__prg_break_point:`, `__prg_break:`, and `__prg_break:n`.)

```

2042 </initex | package>

```

4 l3expan implementation

```

2043 <*initex | package>

```

```

2044 <@@=exp>

```

`\exp_after:wN` These are defined in `l3basics`.

`\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 30.)

4.1 General expansion

In this section a general mechanism for defining functions to handle argument handling 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 4.3. In section 4.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l__exp_internal_tl` This scratch token list variable is defined in `l3basics`, as it is needed “early”. This is just a reminder that is the case!

(End definition for `\l__exp_internal_tl`.)

This code uses internal functions with names that start with `\::` to perform the expansions. All macros are `long` as this turned out to be desirable 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 will require 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.

```
2045 \cs_new:Npn \_exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
2046 \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.

```
2047 \cs_new:Npn \::: #1 {#1}
```

(End definition for \:::.)

\::n This function is used to skip an argument that doesn't need to be expanded.

```
2048 \cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(End definition for \::n.)

\::N This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
2049 \cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(End definition for \::N.)

\::p This function is used to skip an argument that is delimited by a left brace and doesn't need to be expanded. It should not be wrapped in braces in the result.

```
2050 \cs_new:Npn \::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(End definition for \::p.)

\::c This function is used to skip an argument that is turned into a control sequence without expansion.

```
2051 \cs_new:Npn \::c #1 \::: #2#3
2052 { \exp_after:wN \_exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(End definition for \::c.)

\::o This function is used to expand an argument once.

```
2053 \cs_new:Npn \::o #1 \::: #2#3
2054 { \exp_after:wN \_exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(End definition for \::o.)

\::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 or a space. We introduce **\exp_stop_f:** to mark such an end of expansion marker. In the example shown earlier the scanning was stopped once \TeX had fully expanded **\cs_set_eq:Nc \aaa { b \l_tmpa_tl b }** into **\cs_set_eq:NN \aaa = \blurb** which then turned out to contain the non-expandable token **\cs_set_eq:NN**. Since the expansion of **\exp:w \exp_end_continue_f:w** is $\langle null \rangle$, 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.

```

2055 \cs_new:Npn \::f #1 \::: #2#3
2056 {
2057   \exp_after:wN \__exp_arg_next:nnn
2058   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2059   {#1} {#2}
2060 }
2061 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }

```

(End definition for **\::f** and **\exp_stop_f:**.)

\::x This function is used to expand an argument fully.

```

2062 \cs_new_protected:Npn \::x #1 \::: #2#3
2063 {
2064   \cs_set_nopar:Npx \l__exp_internal_tl { {#3} }
2065   \exp_after:wN \__exp_arg_next:nnn \l__exp_internal_tl {#1} {#2}
2066 }

```

(End definition for **\::x**.)

\::v These functions return the value of a register, i.e., one of **tl**, **clist**, **int**, **skip**, **dim** and **\::V muskip**. 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 will terminate using **\exp_end:**. The argument is returned in braces.

```

2067 \cs_new:Npn \::V #1 \::: #2#3
2068 {
2069   \exp_after:wN \__exp_arg_next:nnn
2070   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2071   {#1} {#2}
2072 }
2073 \cs_new:Npn \::v # 1\::: #2#3
2074 {
2075   \exp_after:wN \__exp_arg_next:nnn
2076   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2077   {#1} {#2}
2078 }

```

(End definition for **\::v** and **\::V**.)

__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 do here is try to find

out whether the token will expand to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the register in question when it has been prefixed with `\exp_not:N` and the register itself. If it is a macro, the prefixed `\exp_not:N` will temporarily turn it into the primitive `\scan_stop:`.

```

2079 \cs_new:Npn \__exp_eval_register:N #1
2080 {
2081   \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.

```

2082   \if_meaning:w \scan_stop: #1
2083   \__exp_eval_error_msg:w
2084   \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`.

```

2085   \else:
2086     \exp_after:wN \use_i_ii:nnn
2087   \fi:
2088   \exp_after:wN \exp_end: \tex_the:D #1
2089 }
2090 \cs_new:Npn \__exp_eval_register:c #1
2091 { \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_tl}

```

```

2092 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
2093 {
2094   \fi:
2095   \fi:
2096   \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
2097   \exp_end:
2098 }

```

(End definition for `__exp_eval_register:N` and `__exp_eval_error_msg:w`.)

4.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable and therefore allow to prefix them with `\tex_global:D` for example.

`\exp_args:No` Those lovely runs of expansion!

```

2099 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
2100 \cs_new:Npn \exp_args:NNo #1#2#3
2101   { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
2102 \cs_new:Npn \exp_args:NNNo #1#2#3#4
2103   { \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 27.)

`\exp_args:Nc` In l3basics.

`\exp_args:cc` *(End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 27.)*

`\exp_args:NNc` Here are the functions that turn their argument into csnames but are expandable.

```

2104 \cs_new:Npn \exp_args:NNc #1#2#3
2105   { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
2106 \cs_new:Npn \exp_args:Ncc #1#2#3
2107   { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
2108 \cs_new:Npn \exp_args:Nccc #1#2#3#4
2109   {
2110     \exp_after:wN #1
2111     \cs:w #2 \exp_after:wN \cs_end:
2112     \cs:w #3 \exp_after:wN \cs_end:
2113     \cs:w #4 \cs_end:
2114   }
```

(End definition for `\exp_args:NNc`, `\exp_args:Ncc`, and `\exp_args:Nccc`. These functions are documented on page 28.)

`\exp_args:Nf`

`\exp_args:Nv`

`\exp_args:Nv`

```

2115 \cs_new:Npn \exp_args:Nf #1#2
2116   { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
2117 \cs_new:Npn \exp_args:Nv #1#2
2118   {
2119     \exp_after:wN #1 \exp_after:wN
2120     { \exp:w \__exp_eval_register:c {#2} }
2121   }
2122 \cs_new:Npn \exp_args:NV #1#2
2123   {
2124     \exp_after:wN #1 \exp_after:wN
2125     { \exp:w \__exp_eval_register:N #2 }
2126   }
```

(End definition for `\exp_args:Nf`, `\exp_args:Nv`, and `\exp_args:Nv`. These functions are documented on page 27.)

`\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.

```

2127 \cs_new:Npn \exp_args:NNf #1#2#3
2128 {
2129   \exp_after:wN #1
2130   \exp_after:wN #2
2131   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2132 }
2133 \cs_new:Npn \exp_args:NNv #1#2#3
2134 {
2135   \exp_after:wN #1
2136   \exp_after:wN #2
2137   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2138 }
2139 \cs_new:Npn \exp_args:NNV #1#2#3
2140 {
2141   \exp_after:wN #1
2142   \exp_after:wN #2
2143   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2144 }
2145 \cs_new:Npn \exp_args:Nco #1#2#3
2146 {
2147   \exp_after:wN #1
2148   \cs:w #2 \exp_after:wN \cs_end:
2149   \exp_after:wN {#3}
2150 }
2151 \cs_new:Npn \exp_args:Ncf #1#2#3
2152 {
2153   \exp_after:wN #1
2154   \cs:w #2 \exp_after:wN \cs_end:
2155   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2156 }
2157 \cs_new:Npn \exp_args:NVV #1#2#3
2158 {
2159   \exp_after:wN #1
2160   \exp_after:wN { \exp:w \exp_after:wN
2161     \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
2162   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2163 }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page 28.)

`\exp_args:Ncco` A few more that we can hand-tune.

```

2164 \cs_new:Npn \exp_args:NNNV #1#2#3#4
2165 {
2166   \exp_after:wN #1
2167   \exp_after:wN #2
2168   \exp_after:wN #3
2169   \exp_after:wN { \exp:w \__exp_eval_register:N #4 }
2170 }
2171 \cs_new:Npn \exp_args:NcNc #1#2#3#4
2172 {
2173   \exp_after:wN #1

```

```

2174     \cs:w #2 \exp_after:wN \cs_end:
2175     \exp_after:wN #3
2176     \cs:w #4 \cs_end:
2177   }
2178 \cs_new:Npn \exp_args:NcNo #1#2#3#4
2179 {
2180     \exp_after:wN #1
2181     \cs:w #2 \exp_after:wN \cs_end:
2182     \exp_after:wN #3
2183     \exp_after:wN {#4}
2184 }
2185 \cs_new:Npn \exp_args:Ncco #1#2#3#4
2186 {
2187     \exp_after:wN #1
2188     \cs:w #2 \exp_after:wN \cs_end:
2189     \cs:w #3 \exp_after:wN \cs_end:
2190     \exp_after:wN {#4}
2191 }

```

(End definition for `\exp_args:Ncco` and others. These functions are documented on page 28.)

4.3 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 are all not long: they don't actually take any arguments themselves.

`\exp_args:Nx`

```

2192 \cs_new_protected:Npn \exp_args:Nx { \::x \::: }

```

(End definition for `\exp_args:Nx`. This function is documented on page 27.)

`\exp_args:Nnc`

Here are the actual function definitions, using the helper functions above.

```

\exp_args:Nfo 2193 \cs_new:Npn \exp_args:Nnc { \::n \::c \::: }
\exp_args:Nff 2194 \cs_new:Npn \exp_args:Nfo { \::f \::o \::: }
\exp_args:Nnf 2195 \cs_new:Npn \exp_args:Nff { \::f \::f \::: }
\exp_args:Nno 2196 \cs_new:Npn \exp_args:Nnf { \::n \::f \::: }
\exp_args:NnV 2197 \cs_new:Npn \exp_args:Nno { \::n \::o \::: }
\exp_args:Noo 2198 \cs_new:Npn \exp_args:NnV { \::n \::V \::: }
\exp_args:Nof 2199 \cs_new:Npn \exp_args:Noo { \::o \::o \::: }
\exp_args:Noc 2200 \cs_new:Npn \exp_args:Nof { \::o \::f \::: }
\exp_args:NNx 2201 \cs_new:Npn \exp_args:Noc { \::o \::c \::: }
\exp_args:Ncx 2202 \cs_new_protected:Npn \exp_args:NNx { \::N \::x \::: }
\exp_args:Nnx 2203 \cs_new_protected:Npn \exp_args:Ncx { \::c \::x \::: }
\exp_args:Nox 2204 \cs_new_protected:Npn \exp_args:Nnx { \::n \::x \::: }
\exp_args:Nxo 2205 \cs_new_protected:Npn \exp_args:Nox { \::o \::x \::: }
\exp_args:Nxx 2206 \cs_new_protected:Npn \exp_args:Nxo { \::x \::o \::: }
\exp_args:Nxx 2207 \cs_new_protected:Npn \exp_args:Nxx { \::x \::x \::: }

```

(End definition for `\exp_args:Nnc` and others. These functions are documented on page 28.)

`\exp_args:NNno`

```

\exp_args:NNoo 2208 \cs_new:Npn \exp_args:NNno { \::N \::n \::o \::: }
\exp_args:NNnc 2209 \cs_new:Npn \exp_args:NNoo { \::N \::o \::o \::: }
\exp_args:NNno
\exp_args:Nooo
\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Noox

```

```

2210 \cs_new:Npn \exp_args:Nnnc { \::n \::n \::c \::: }
2211 \cs_new:Npn \exp_args:Nnno { \::n \::n \::o \::: }
2212 \cs_new:Npn \exp_args:Nooo { \::o \::o \::o \::: }
2213 \cs_new_protected:Npn \exp_args:NNNx { \::N \::N \::x \::: }
2214 \cs_new_protected:Npn \exp_args:NNnx { \::N \::n \::x \::: }
2215 \cs_new_protected:Npn \exp_args:NNox { \::N \::o \::x \::: }
2216 \cs_new_protected:Npn \exp_args:Nnnx { \::n \::n \::x \::: }
2217 \cs_new_protected:Npn \exp_args:Nnox { \::n \::o \::x \::: }
2218 \cs_new_protected:Npn \exp_args:Nccx { \::c \::c \::x \::: }
2219 \cs_new_protected:Npn \exp_args:Ncnx { \::c \::n \::x \::: }
2220 \cs_new_protected:Npn \exp_args:Noox { \::o \::o \::x \::: }

```

(End definition for `\exp_args:Nnno` and others. These functions are documented on page 28.)

4.4 Last-unbraced versions

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

\__exp_arg_last_unbraced:nn
\::f_unbraced
\::o_unbraced
\::V_unbraced
\::v_unbraced
\::x_unbraced
2221 \cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
2222 \cs_new:Npn \::f_unbraced \::: #1#2
2223 {
2224   \exp_after:wN \__exp_arg_last_unbraced:nn
2225   \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
2226 }
2227 \cs_new:Npn \::o_unbraced \::: #1#2
2228 { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
2229 \cs_new:Npn \::V_unbraced \::: #1#2
2230 {
2231   \exp_after:wN \__exp_arg_last_unbraced:nn
2232   \exp_after:wN { \exp:w \__exp_eval_register:N #2 } {#1}
2233 }
2234 \cs_new:Npn \::v_unbraced \::: #1#2
2235 {
2236   \exp_after:wN \__exp_arg_last_unbraced:nn
2237   \exp_after:wN { \exp:w \__exp_eval_register:c {#2} } {#1}
2238 }
2239 \cs_new_protected:Npn \::x_unbraced \::: #1#2
2240 {
2241   \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
2242   \l__exp_internal_tl
2243 }

```

(End definition for `__exp_arg_last_unbraced:nn` and others.)

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

\exp_last_unbraced:NV
\exp_last_unbraced:Nv
\exp_last_unbraced:Nf
\exp_last_unbraced:No
\exp_last_unbraced:Nco
\exp_last_unbraced:NcV
\exp_last_unbraced:NNV
\exp_last_unbraced:NNNo
\exp_last_unbraced:NNNV
\exp_last_unbraced:NNNo
\exp_last_unbraced:Nno
\exp_last_unbraced:Noo
\exp_last_unbraced:Nfo
\exp_last_unbraced:NnNo
\exp_last_unbraced:Nx
2244 \cs_new:Npn \exp_last_unbraced:NV #1#2
2245 { \exp_after:wN #1 \exp:w \__exp_eval_register:N #2 }
2246 \cs_new:Npn \exp_last_unbraced:Nv #1#2
2247 { \exp_after:wN #1 \exp:w \__exp_eval_register:c {#2} }
2248 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
2249 \cs_new:Npn \exp_last_unbraced:Nf #1#2
2250 { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
2251 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3

```

```

2252 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
2253 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
2254 {
2255   \exp_after:wN #1
2256   \cs:w #2 \exp_after:wN \cs_end:
2257   \exp:w \__exp_eval_register:N #3
2258 }
2259 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
2260 {
2261   \exp_after:wN #1
2262   \exp_after:wN #2
2263   \exp:w \__exp_eval_register:N #3
2264 }
2265 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3
2266 { \exp_after:wN #1 \exp_after:wN #2 #3 }
2267 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
2268 {
2269   \exp_after:wN #1
2270   \exp_after:wN #2
2271   \exp_after:wN #3
2272   \exp:w \__exp_eval_register:N #4
2273 }
2274 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
2275 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
2276 \cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \:: }
2277 \cs_new:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \:: }
2278 \cs_new:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \:: }
2279 \cs_new:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \:: }
2280 \cs_new_protected:Npn \exp_last_unbraced:Nx { \::x_unbraced \:: }

```

(End definition for `\exp_last_unbraced:NV` and others. These functions are documented on page 29.)

`\exp_last_two_unbraced:Noo` If #2 is a single token then this can be implemented as

`__exp_last_two_unbraced:noN`

```

\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.

```

2281 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
2282 { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
2283 \cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
2284 { \exp_after:wN #3 #2 #1 }

```

(End definition for `\exp_last_two_unbraced:Noo` and `__exp_last_two_unbraced:noN`. These functions are documented on page 29.)

4.5 Preventing expansion

`\exp_not:o`
`\exp_not:c`
`\exp_not:f`
`\exp_not:V`
`\exp_not:v`

```

2285 \cs_new:Npn \exp_not:o #1 { \etex_unexpanded:D \exp_after:wN {#1} }
2286 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
2287 \cs_new:Npn \exp_not:f #1
2288 { \etex_unexpanded:D \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
2289 \cs_new:Npn \exp_not:V #1

```

```

2290 {
2291   \etex_unexpanded:D \exp_after:wN
2292     { \exp:w \__exp_eval_register:N #1 }
2293 }
2294 \cs_new:Npn \exp_not:v #1
2295 {
2296   \etex_unexpanded:D \exp_after:wN
2297     { \exp:w \__exp_eval_register:c {#1} }
2298 }

```

(End definition for `\exp_not:o` and others. These functions are documented on page 31.)

4.6 Controlled expansion

`\exp:w` To trigger a sequence of “arbitrary” many expansions we need a method to invoke \TeX ’s
`\exp_end:` expansion mechanism in such a way that a) we are able to stop it in a controlled manner
`\exp_end_continue_f:w` and b) that the result of what triggered the expansion in the first place is null, i.e., that
`\exp_end_continue_f:nw` 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_romannumeral: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.

```

2299 %\cs_new_eq:NN \exp:w \tex_romannumeral:D

```

So to stop the expansion sequence in a controlled way all we need to provide is `\c_zero` as part of expanded tokens. As this is an integer constant it will immediately stop `\tex_romannumeral:D`’s search for a number.

```

2300 %\cs_new_eq:NN \exp_end: \c_zero

```

(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 $\langle number \rangle$ will continue searching for an optional space (and it will continue expansion doing that) — see \TeX book page 269 for details.

```

2301 \tex_catcode:D ‘^^@=13
2302 \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.⁶

```

2303 \cs_new:Npn ^^@{\expansionERROR}
2304 \cs_new:Npn \exp_end_continue_f:nw #1 { ‘^^@ #1 }
2305 \tex_catcode:D ‘^^@=15

```

(End definition for `\exp:w` and others. These functions are documented on page 31.)

⁶Need to get a real error message.

4.7 Defining function variants

2306 `<@@=cs>`

`\cs_generate_variant:Nn` #1 : Base form of a function; e.g., `\tl_set:Nn`
 #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.

```
2307 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
2308 {
2309   \__chk_if_exist_cs:N #1
2310   \__cs_generate_variant:N #1
2311   \exp_after:wN \__cs_split_function:NN
2312   \exp_after:wN #1
2313   \exp_after:wN \__cs_generate_variant:nnNN
2314   \exp_after:wN #1
2315   \tl_to_str:n {#2} , \scan_stop: , \q_recursion_stop
2316 }
```

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 25.)

```
\__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 all primitive TeX conditionals are expandable.

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 `\q_mark` is taken as an argument of the `wwNw` auxiliary, and #3 is `\cs_new_protected:Npx`, otherwise it is `\cs_new:Npx`.

```
2317 \cs_new_protected:Npx \__cs_generate_variant:N #1
2318 {
2319   \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
2320   \exp_not:N \exp_not:N #1 #1
2321   \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx
2322   \exp_not:N \else:
2323   \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
2324   \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
2325   \exp_not:N \q_mark
2326   \exp_not:N \q_mark \cs_new_protected:Npx
2327   \tl_to_str:n { pr }
2328   \exp_not:N \q_mark \cs_new:Npx
2329   \exp_not:N \q_stop
2330   \exp_not:N \fi:
```

```

2331 }
2332 \use:x
2333 {
2334   \cs_new_protected:Npn \exp_not:N \__cs_generate_variant:ww
2335     ##1 \tl_to_str:n { ma } ##2 \exp_not:N \q_mark
2336 }
2337 { \__cs_generate_variant:wwNw #1 }
2338 \use:x
2339 {
2340   \cs_new_protected:Npn \exp_not:N \__cs_generate_variant:wwNw
2341     ##1 \tl_to_str:n { pr } ##2 \exp_not:N \q_mark
2342     ##3 ##4 \exp_not:N \q_stop
2343 }
2344 { \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.

```

2345 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
2346 {
2347   \if_meaning:w \c_false_bool #3
2348     \__msg_kernel_error:nnx { kernel } { missing-colon }
2349     { \token_to_str:c {#1} }
2350     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2351   \fi:
2352   \__cs_generate_variant:Nnnw #4 {#1}{#2}
2353 }

```

(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}`, it would be better to 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` should be defined using `\exp_args:Nnx`, not `\exp_args:Nox`, to avoid double o expansion.
- Lastly, `\cs_generate_variant:Nn \foo:on {xn}` should trigger an error, because we do not have a means to replace o-expansion by x-expansion.

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 two cases: `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` in the form `<processed variant signature> \q_mark <errors> \q_stop <base function> <new function>`. If all went well, `<errors>` is empty; otherwise, it is a kernel error message, followed by some clean-up code (`\use_none:nnnn`).

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`.

```

2354 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
2355 {
2356   \if_meaning:w \scan_stop: #4
2357   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2358   \fi:
2359   \use:x
2360   {
2361     \exp_not:N \__cs_generate_variant:wwNN
2362     \__cs_generate_variant_loop:nNwN { }
2363     #4
2364     \__cs_generate_variant_loop_end:nwwwNNnn
2365     \q_mark
2366     #3 ~
2367     { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
2368     { }
2369     \q_stop
2370     \exp_not:N #1 {#2} {#4}
2371   }
2372   \__cs_generate_variant:Nnnw #1 {#2} {#3}
2373 }
```

(End definition for `__cs_generate_variant:Nnnw`.)

<code>__cs_generate_variant_loop:nNwN</code>	#1 :	Last few (consecutive) letters common between the base and variant (in fact, <code>__cs_generate_variant_loop_same:w</code>
<code>__cs_generate_variant_loop_same:w</code>		<code>cs_generate_variant_same:N</code> <i><letter></i> for each letter).
<code>__cs_generate_variant_loop_end:nwwwNNnn</code>	#2 :	Next variant letter.
<code>__cs_generate_variant_loop_long:wNNnn</code>	#3 :	Remainder of variant form.
<code>__cs_generate_variant_loop_invalid:NNwNNnn</code>	#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 with a base letter of `N` or `n`. 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 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 neither `n` nor `N`. Again, 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.

```

2374 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \q_mark #4
2375 {
2376   \if:w #2 #4
2377     \exp_after:wN \__cs_generate_variant_loop_same:w
2378   \else:
2379     \if:w N #4 \else:
2380       \if:w n #4 \else:
2381         \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
2382       \fi:
2383     \fi:
2384   \fi:
2385   #1
2386   \prg_do_nothing:
2387   #2
2388   \__cs_generate_variant_loop:nNwN { } #3 \q_mark
2389 }
2390 \cs_new:Npn \__cs_generate_variant_loop_same:w
2391   #1 \prg_do_nothing: #2#3#4
2392 {
2393   #3 { #1 \__cs_generate_variant_same:N #2 }
2394 }
2395 \cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
2396   #1#2 \q_mark #3 ~ #4 \q_stop #5#6#7#8
2397 {
2398   \scan_stop: \scan_stop: \fi:
2399   \exp_not:N \q_mark
2400   \exp_not:N \q_stop

```

```

2401     \exp_not:N #6
2402     \exp_not:c { #7 : #8 #1 #3 }
2403   }
2404 \cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \q_stop #2#3#4#5
2405 {
2406   \exp_not:n
2407   {
2408     \q_mark
2409     \_msg_kernel_error:nxxx { kernel } { variant-too-long }
2410     {#5} { \token_to_str:N #3 }
2411     \use_none:nnnn
2412     \q_stop
2413     #3
2414     #3
2415   }
2416 }
2417 \cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
2418 #1#2 \fi: \fi: \fi: #3 \q_stop #4#5#6#7
2419 {
2420   \fi: \fi: \fi:
2421   \exp_not:n
2422   {
2423     \q_mark
2424     \_msg_kernel_error:nnxxx { kernel } { invalid-variant }
2425     {#7} { \token_to_str:N #5 } {#1} {#2}
2426     \use_none:nnnn
2427     \q_stop
2428     #5
2429     #5
2430   }
2431 }

```

(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.

```

2432 \cs_new:Npn \__cs_generate_variant_same:N #1
2433 {
2434   \if:w N #1
2435     N
2436   \else:
2437     \if:w p #1
2438       p
2439     \else:
2440       n
2441     \fi:
2442   \fi:
2443 }

```

(End definition for __cs_generate_variant_same:N.)

__cs_generate_variant:wwNN If the variant form has already been defined, log its existence. 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.

```

2444 \cs_new_protected:Npn \__cs_generate_variant:wwNN
2445   #1 \q_mark #2 \q_stop #3#4
2446   {
2447     #2
2448     \cs_if_free:NTF #4
2449     {
2450       \group_begin:
2451         \__cs_generate_internal_variant:n {#1}
2452         \__cs_tmp:w #4 { \exp_not:c { \exp_args:N #1 } \exp_not:N #3 }
2453       \group_end:
2454     }
2455     {
2456       \__chk_log:x
2457       {
2458         Variant~\token_to_str:N #4~%
2459         already~defined;~ not~ changing~ it~ \msg_line_context:
2460       }
2461     }
2462   }

```

(End definition for `__cs_generate_variant:wwNN`.)

`__cs_generate_internal_variant:n`
`__cs_generate_internal_variant:wwnw`
`__cs_generate_internal_variant_loop:n`

Test if `\exp_args:N #1` is already defined and if not define it via the `\::` commands using the chars in #1. If #1 contains an x (this is the place where having converted the original comma-list argument to a string is very important), the result should be protected, and the next variant to be defined using that internal variant should be protected.

```

2463 \cs_new_protected:Npx \__cs_generate_internal_variant:n #1
2464   {
2465     \exp_not:N \__cs_generate_internal_variant:wwnNwnn
2466     #1 \exp_not:N \q_mark
2467     { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx }
2468     \cs_new_protected:cpx
2469     \token_to_str:N x \exp_not:N \q_mark
2470     { }
2471     \cs_new:cpx
2472     \exp_not:N \q_stop
2473     { \exp_args:N #1 }
2474     {
2475       \exp_not:N \__cs_generate_internal_variant_loop:n #1
2476       { : \exp_not:N \use_i:nn }
2477     }
2478   }
2479 \use:x
2480   {
2481     \cs_new_protected:Npn \exp_not:N \__cs_generate_internal_variant:wwnNwnn
2482       ##1 \token_to_str:N x ##2 \exp_not:N \q_mark
2483       ##3 ##4 ##5 \exp_not:N \q_stop ##6 ##7
2484   }
2485   {
2486     #3
2487     \cs_if_free:cT {#6} { #4 {#6} {#7} }
2488   }

```

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.

```

2489 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
2490 {
2491     \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
2492     \__cs_generate_internal_variant_loop:n
2493 }

```

(End definition for `__cs_generate_internal_variant:n`, `__cs_generate_internal_variant:www`, and `__cs_generate_internal_variant_loop:n`.)

```

2494 </initex | package>

```

5 l3prg implementation

The following test files are used for this code: `m3prg001.lvt`, `m3prg002.lvt`, `m3prg003.lvt`.

```

2495 <*initex | package>

```

5.1 Primitive conditionals

```

\if_bool:N
\if_predicate:w

```

Those two primitive TeX conditionals are synonyms.

```

2496 \cs_new_eq:NN \if_bool:N \tex_ifodd:D
2497 \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 40.)

5.2 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 34.)

5.3 The boolean data type

```

2498 <@@=bool>

```

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```

\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Npnn
\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
\prg_set_eq_conditional:NNN
\prg_new_eq_conditional:NNN
\prg_return_true:
\prg_return_false:

```

```

2499 \cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
2500 \cs_generate_variant:Nn \bool_new:N { c }

```

(End definition for `\bool_new:N`. This function is documented on page 36.)

```

\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.

```

2501 \cs_new_protected:Npn \bool_set_true:N #1
2502 { \cs_set_eq:NN #1 \c_true_bool }
2503 \cs_new_protected:Npn \bool_set_false:N #1
2504 { \cs_set_eq:NN #1 \c_false_bool }
2505 \cs_new_protected:Npn \bool_gset_true:N #1
2506 { \cs_gset_eq:NN #1 \c_true_bool }

```

```

2507 \cs_new_protected:Npn \bool_gset_false:N #1
2508 { \cs_gset_eq:NN #1 \c_false_bool }
2509 \cs_generate_variant:Nn \bool_set_true:N { c }
2510 \cs_generate_variant:Nn \bool_set_false:N { c }
2511 \cs_generate_variant:Nn \bool_gset_true:N { c }
2512 \cs_generate_variant:Nn \bool_gset_false:N { c }

```

(End definition for `\bool_set_true:N` and others. These functions are documented on page 36.)

```

\bool_set_eq:NN The usual copy code.
\bool_set_eq:cN 2513 \cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN
\bool_set_eq:Nc 2514 \cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc
\bool_set_eq:cc 2515 \cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN
\bool_gset_eq:NN 2516 \cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc
\bool_gset_eq:cN 2517 \cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
\bool_gset_eq:Nc 2518 \cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
\bool_gset_eq:cN 2519 \cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
\bool_gset_eq:cc 2520 \cs_new_eq:NN \bool_gset_eq:cc \cs_gset_eq:cc

```

(End definition for `\bool_set_eq:NN` and `\bool_gset_eq:NN`. These functions are documented on page 37.)

`\bool_set:Nn` This function evaluates a boolean expression and assigns the first argument the meaning
`\bool_set:cn` `\c_true_bool` or `\c_false_bool`.

```

\bool_gset:Nn 2521 \cs_new_protected:Npn \bool_set:Nn #1#2
\bool_gset:cn 2522 { \tex_chardef:D #1 = \bool_if_p:n {#2} }
2523 \cs_new_protected:Npn \bool_gset:Nn #1#2
2524 { \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }
2525 \cs_generate_variant:Nn \bool_set:Nn { c }
2526 \cs_generate_variant:Nn \bool_gset:Nn { c }

```

(End definition for `\bool_set:Nn` and `\bool_gset:Nn`. These functions are documented on page 37.)

Booleans are not based on token lists but do need checking: this code complements similar material in `l3tl`.

```

2527 \*package)
2528 \if_bool:N \l@expl@check@declarations@bool
2529 \cs_set_protected:Npn \bool_set_true:N #1
2530 {
2531   \__chk_if_exist_var:N #1
2532   \cs_set_eq:NN #1 \c_true_bool
2533 }
2534 \cs_set_protected:Npn \bool_set_false:N #1
2535 {
2536   \__chk_if_exist_var:N #1
2537   \cs_set_eq:NN #1 \c_false_bool
2538 }
2539 \cs_set_protected:Npn \bool_gset_true:N #1
2540 {
2541   \__chk_if_exist_var:N #1
2542   \cs_gset_eq:NN #1 \c_true_bool
2543 }
2544 \cs_set_protected:Npn \bool_gset_false:N #1
2545 {
2546   \__chk_if_exist_var:N #1

```

```

2547     \cs_gset_eq:NN #1 \c_false_bool
2548   }
2549   \cs_set_protected:Npn \bool_set_eq:NN #1
2550   {
2551     \__chk_if_exist_var:N #1
2552     \cs_set_eq:NN #1
2553   }
2554   \cs_set_protected:Npn \bool_gset_eq:NN #1
2555   {
2556     \__chk_if_exist_var:N #1
2557     \cs_gset_eq:NN #1
2558   }
2559   \cs_set_protected:Npn \bool_set:Nn #1#2
2560   {
2561     \__chk_if_exist_var:N #1
2562     \tex_chardef:D #1 = \bool_if_p:n {#2}
2563   }
2564   \cs_set_protected:Npn \bool_gset:Nn #1#2
2565   {
2566     \__chk_if_exist_var:N #1
2567     \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
2568   }
2569   \fi:
2570 \endpackage

```

\bool_if_p:N Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```

\bool_if_p:c
\bool_if:NTF
\bool_if:cTF
2571 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
2572 {
2573   \if_meaning:w \c_true_bool #1
2574   \prg_return_true:
2575   \else:
2576   \prg_return_false:
2577   \fi:
2578 }
2579 \cs_generate_variant:Nn \bool_if_p:N { c }
2580 \cs_generate_variant:Nn \bool_if:NT { c }
2581 \cs_generate_variant:Nn \bool_if:NF { c }
2582 \cs_generate_variant:Nn \bool_if:NTF { c }

```

(End definition for `\bool_if:NTF`. This function is documented on page 37.)

\bool_show:N Show the truth value of the boolean, as true or false.

```

\bool_show:c
\bool_show:n
2583 \cs_new_protected:Npn \bool_show:N #1
2584 {
2585   \__msg_show_variable:NNNnn #1 \bool_if_exist:NTF ? { }
2586   { > ~ \token_to_str:N #1 = \__bool_to_str:n {#1} }
2587 }
2588 \cs_new_protected:Npn \bool_show:n
2589 { \__msg_show_wrap:Nn \__bool_to_str:n }
2590 \cs_new:Npn \__bool_to_str:n #1
2591 { \bool_if:nTF {#1} { true } { false } }
2592 \cs_generate_variant:Nn \bool_show:N { c }

```

(End definition for `\bool_show:N`, `\bool_show:n`, and `_bool_to_str:n`. These functions are documented on page 37.)

`\l_tmpa_bool` A few booleans just if you need them.

```
\l_tmpb_bool 2593 \bool_new:N \l_tmpa_bool
\g_tmpa_bool 2594 \bool_new:N \l_tmpb_bool
\g_tmpb_bool 2595 \bool_new:N \g_tmpa_bool
2596 \bool_new:N \g_tmpb_bool
```

(End definition for `\l_tmpa_bool` and others. These variables are documented on page 37.)

`\bool_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```
\bool_if_exist_p:c 2597 \prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N
\bool_if_exist:NTF 2598 { TF , T , F , p }
\bool_if_exist:cTF 2599 \prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c
2600 { TF , T , F , p }
```

(End definition for `\bool_if_exist:NTF`. This function is documented on page 37.)

5.4 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”. We shall use the terms Not, And, Or, Open and Close for these operations.

`\bool_if:nTF`

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 `GetNotNext` function, which eventually reverses the logic compared to `GetNext`.
- 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 $\langle true \rangle$ or $\langle false \rangle$.

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:

$\langle true \rangle$ **And** Current truth value is true, logical And seen, continue with `GetNext` to examine truth value of next boolean (sub-)expression.

$\langle false \rangle$ **And** Current truth value is false, logical And seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return $\langle false \rangle$.

$\langle true \rangle$ **Or** Current truth value is true, logical Or seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return $\langle true \rangle$.

$\langle false \rangle$ **Or** Current truth value is false, logical Or seen, continue with `GetNext` to examine truth value of next boolean (sub-)expression.

$\langle true \rangle$ **Close** Current truth value is true, Close seen, return $\langle true \rangle$.

$\langle false \rangle$ **Close** Current truth value is false, Close seen, return $\langle false \rangle$.

We introduce an additional Stop operation with the same semantics as the Close operation.

$\langle true \rangle$ **Stop** Current truth value is true, return $\langle true \rangle$.

$\langle false \rangle$ **Stop** Current truth value is false, return $\langle false \rangle$.

The reasons for this follow below.

```

2601 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
2602 {
2603   \if_predicate:w \bool_if_p:n {#1}
2604   \prg_return_true:
2605   \else:
2606   \prg_return_false:
2607   \fi:
2608 }

```

(End definition for `\bool_if:nTF`. This function is documented on page 38.)

```

\bool_if_p:n
\_bool_if_left_parentheses:wwwn
\_bool_if_right_parentheses:wwwn
\_bool_if_or:wwwn

```

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 \TeX . This will be closed at the end of the expression parsing (see S below).

Minimal (“short-circuit”) evaluation of boolean expressions requires skipping to the end of the current parenthesized group when $\langle true \rangle ||$ is seen, but to the next `||` or closing parenthesis when $\langle false \rangle \&\&$ is seen. To avoid having separate functions for the two cases, we transform the boolean expression by doubling each parenthesis and adding parenthesis around each `||`. This ensures that `&&` will bind tighter than `||`.

The replacement is done in three passes, for left and right parentheses and for `||`. At each pass, the part of the expression that has been transformed is stored before `\q_nil`, the rest lies until the first `\q_mark`, followed by an empty brace group. A trailing marker ensures that the auxiliaries’ delimited arguments will not run-away. As long as the delimiter matches inside the expression, material is moved before `\q_nil` and we continue. Afterwards, the trailing marker is taken as a delimiter, `#4` is the next auxiliary, immediately followed by a new `\q_nil` delimiter, which indicates that nothing has been treated at this pass. The last step calls `__bool_if_parse:NNNww` which cleans up and triggers the evaluation of the expression itself.

```

2609 \cs_new:Npn \bool_if_p:n #1
2610 {
2611   \group_align_safe_begin:
2612   \__bool_if_left_parentheses:wwwn \q_nil
2613   #1 \q_mark { }
2614   ( \q_mark { \__bool_if_right_parentheses:wwwn \q_nil }
2615   ) \q_mark { \__bool_if_or:wwwn \q_nil }
2616   || \q_mark \__bool_if_parse:NNNww
2617   \q_stop
2618 }
2619 \cs_new:Npn \__bool_if_left_parentheses:wwwn #1 \q_nil #2 ( #3 \q_mark #4
2620 { #4 \__bool_if_left_parentheses:wwwn #1 #2 (( \q_nil #3 \q_mark {#4} }
2621 \cs_new:Npn \__bool_if_right_parentheses:wwwn #1 \q_nil #2 ) #3 \q_mark #4
2622 { #4 \__bool_if_right_parentheses:wwwn #1 #2 )) \q_nil #3 \q_mark {#4} }
2623 \cs_new:Npn \__bool_if_or:wwwn #1 \q_nil #2 || #3 \q_mark #4
2624 { #4 \__bool_if_or:wwwn #1 #2 )|| ( \q_nil #3 \q_mark {#4} }

```

(End definition for `\bool_if_p:n` and others. These functions are documented on page 38.)

`__bool_if_parse:NNNww` After removing extra tokens from the transformation phase, start evaluating. At the end, we will need to finish the special `align_safe` group before finally returning a `\c_true_bool` or `\c_false_bool` as there might otherwise be something left in front in the input stream. For this we call the Stop operation, denoted simply by a `S` following the last Close operation.

```
2625 \cs_new:Npn \__bool_if_parse:NNNww #1#2#3#4 \q_mark #5 \q_stop
2626 {
2627   \__bool_get_next:NN \use_i:nn (( #4 )) S
2628 }
```

(End definition for `__bool_if_parse:NNNww`.)

`__bool_get_next:NN` The GetNext operation. This is a switch: if what follows is neither `!` nor `(`, we assume it is a predicate. The first argument is `\use_ii:nn` if the logic must eventually be reversed (after a `!`), otherwise it is `\use_i:nn`. 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.

```
2629 \cs_new:Npn \__bool_get_next:NN #1#2
2630 {
2631   \use:c
2632   {
2633     __bool_
2634     \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
2635     :Nw
2636   }
2637   #1 #2
2638 }
```

(End definition for `__bool_get_next:NN`.)

`__bool_!:Nw` The Not operation reverses the logic: discard the `!` token and call the GetNext operation with its first argument reversed.

```
2639 \cs_new:cpn { __bool_!:Nw } #1#2
2640 { \exp_after:wN \__bool_get_next:NN #1 \use_ii:nn \use_i:nn }
```

(End definition for `__bool_!:Nw`.)

`__bool_(:Nw` The Open operation starts a sub-expression after discarding the token. This is done by calling GetNext, with a post-processing step which looks for And, Or or Close afterwards.

```
2641 \cs_new:cpn { __bool_(:Nw } #1#2
2642 {
2643   \exp_after:wN \__bool_choose:NNN \exp_after:wN #1
2644   \__int_value:w \__bool_get_next:NN \use_i:nn
2645 }
```

(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.

```
2646 \cs_new:cpn { __bool_p:Nw } #1
2647 { \exp_after:wN \__bool_choose:NNN \exp_after:wN #1 \__int_value:w }
```

(End definition for _bool_p:Nw.)

_bool_choose:NNN Branching the eight-way switch. The arguments are 1: \use_i:nn or \use_ii:nn, 2: 0 or 1 encoding the current truth value, 3: the next operation, And, Or, Close or Stop. If #1 is \use_ii:nn, the logic of #2 must be reversed.

```

2648 \cs_new:Npn \_bool\_choose:NNN #1#2#3
2649 {
2650   \use:c
2651   {
2652     __bool\_#3\_
2653     #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: }
2654     :w
2655   }
2656 }
```

(End definition for _bool_choose:NNN.)

bool)_0:w Closing a group is just about returning the result. The Stop operation is similar except it closes the special alignment group before returning the boolean.

```

\_bool\_)\_1:w
\_bool\_S\_0:w 2657 \cs_new:cpn { \_bool\_)\_0:w } { \c_false_bool }
\_bool\_S\_1:w 2658 \cs_new:cpn { \_bool\_)\_1:w } { \c_true_bool }
2659 \cs_new:cpn { \_bool\_S\_0:w } { \group_align_safe_end: \c_false_bool }
2660 \cs_new:cpn { \_bool\_S\_1:w } { \group_align_safe_end: \c_true_bool }
```

(End definition for _bool_)_0:w and others.)

bool&_1:w Two cases where we simply continue scanning. We must remove the second & or |.

```

\_bool\_|\_0:w 2661 \cs_new:cpn { \_bool\_&\_1:w } & { \_bool\_get\_next:NN \use\_i:nn }
2662 \cs_new:cpn { \_bool\_|\_0:w } | { \_bool\_get\_next:NN \use\_i:nn }
```

(End definition for _bool_&_1:w and _bool_|_0:w.)

bool&_0:w When the truth value has already been decided, we have to throw away the remainder of the current group as we are doing minimal evaluation. This is slightly tricky as there are no braces so we have to play match the () manually.

```

\_bool\_|\_1:w
\_bool\_eval\_skip\_to\_end\_auxi:Nw 2663 \cs_new:cpn { \_bool\_&\_0:w } &
\_bool\_eval\_skip\_to\_end\_auxii:Nw 2664 { \_bool\_eval\_skip\_to\_end\_auxi:Nw \c_false_bool }
\_bool\_eval\_skip\_to\_end\_auxiii:Nw 2665 \cs_new:cpn { \_bool\_|\_1:w } |
2666 { \_bool\_eval\_skip\_to\_end\_auxi:Nw \c_true_bool }
```

There is always at least one) waiting, namely the outer one. However, we are facing the problem that there may be more than one that need to be finished off and we have to detect the correct number of them. Here is a complicated example showing how this is done. After evaluating the following, we realize we must skip everything after the first And. Note the extra Close at the end.

```
\c_false_bool && ((abc) && xyz) && ((xyz) && (def)))
```

First read up to the first Close. This gives us the list we first read up until the first right parenthesis so we are looking at the token list

```
((abc
```

This contains two Open markers so we must remove two groups. Since no evaluation of the contents is to be carried out, it doesn't matter how we remove the groups as long as we wind up with the correct result. We therefore first remove a () pair and what preceded the Open – but leave the contents as it may contain Open tokens itself – leaving

```
(abc && xyz) && ((xyz) && (def)))
```

Another round of this gives us

```
(abc && xyz
```

which still contains an Open so we remove another () pair, giving us

```
abc && xyz && ((xyz) && (def)))
```

Again we read up to a Close and again find Open tokens:

```
abc && xyz && ((xyz
```

Further reduction gives us

```
(xyz && (def)))
```

and then

```
(xyz && (def
```

with reduction to

```
xyz && (def))
```

and ultimately we arrive at no Open tokens being skipped and we can finally close the group nicely.

```
2667 %% (
2668 \cs_new:Npn \__bool_eval_skip_to_end_auxi:Nw #1#2 )
2669 {
2670   \__bool_eval_skip_to_end_auxii:Nw #1#2 ( % )
2671   \q_no_value \q_stop
2672   {#2}
2673 }
```

If no right parenthesis, then #3 is no_value and we are done, return the boolean #1. If there is, we need to grab a () pair and then recurse

```
2674 \cs_new:Npn \__bool_eval_skip_to_end_auxii:Nw #1#2 ( #3#4 \q_stop #5 % )
2675 {
2676   \quark_if_no_value:NTF #3
2677   {#1}
2678   { \__bool_eval_skip_to_end_auxiii:Nw #1 #5 }
2679 }
```

Keep the boolean, throw away anything up to the (as it is irrelevant, remove a () pair but remember to reinsert #3 as it may contain (tokens!

```
2680 \cs_new:Npn \__bool_eval_skip_to_end_auxiii:Nw #1#2 ( #3 )
2681 { % (
2682   \__bool_eval_skip_to_end_auxi:Nw #1#3 )
2683 }
```

(End definition for __bool_&_0:w and others.)

`\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.

```
2684 \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 38.)

`\bool_xor_p:nn` Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```
2685 \cs_new:Npn \bool_xor_p:nn #1#2
2686 {
2687   \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
2688     \c_false_bool
2689     \c_true_bool
2690 }
```

(End definition for `\bool_xor_p:nn`. This function is documented on page 38.)

5.5 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.

`\bool_while_do:cn`

`\bool_until_do:Nn`

`\bool_until_do:cn`

```
2691 \cs_new:Npn \bool_while_do:Nn #1#2
2692 { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
2693 \cs_new:Npn \bool_until_do:Nn #1#2
2694 { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
2695 \cs_generate_variant:Nn \bool_while_do:Nn { c }
2696 \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 39.)

`\bool_do_while:Nn` A do-while loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

`\bool_do_while:cn`

`\bool_do_until:Nn`

`\bool_do_until:cn`

```
2697 \cs_new:Npn \bool_do_while:Nn #1#2
2698 { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
2699 \cs_new:Npn \bool_do_until:Nn #1#2
2700 { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
2701 \cs_generate_variant:Nn \bool_do_while:Nn { c }
2702 \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 38.)

`\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`

```
2703 \cs_new:Npn \bool_while_do:nn #1#2
2704 {
2705   \bool_if:nT {#1}
2706   {
2707     #2
2708     \bool_while_do:nn {#1} {#2}
2709   }
2710 }
```

```

2711 \cs_new:Npn \bool_do_while:nn #1#2
2712 {
2713   #2
2714   \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
2715 }
2716 \cs_new:Npn \bool_until_do:nn #1#2
2717 {
2718   \bool_if:nF {#1}
2719   {
2720     #2
2721     \bool_until_do:nn {#1} {#2}
2722   }
2723 }
2724 \cs_new:Npn \bool_do_until:nn #1#2
2725 {
2726   #2
2727   \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
2728 }

```

(End definition for `\bool_while_do:nn` and others. These functions are documented on page 39.)

5.6 Producing multiple copies

```

2729 <@@=prg>

```

`\prg_replicate:nn`

This function uses a cascading csname technique by David Kastrup (who else :-)

`__prg_replicate:N`

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 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.

`__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`

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it will severely affect 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} {<code>} }`. 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.

`__prg_replicate_first-:n`
`__prg_replicate_first_0:n`
`__prg_replicate_first_1:n`
`__prg_replicate_first_2:n`
`__prg_replicate_first_3:n`
`__prg_replicate_first_4:n`
`__prg_replicate_first_5:n`
`__prg_replicate_first_6:n`
`__prg_replicate_first_7:n`
`__prg_replicate_first_8:n`
`__prg_replicate_first_9:n`

```

2730 \cs_new:Npn \prg_replicate:nn #1
2731 {
2732   \exp:w
2733   \exp_after:wN \__prg_replicate_first:N
2734   \__int_value:w \__int_eval:w #1 \__int_eval_end:
2735   \cs_end:
2736 }
2737 \cs_new:Npn \__prg_replicate:N #1

```

Then comes all the functions that do the hard work of inserting all the copies. The first function takes `:n` as a parameter.

Users shouldn't ask for something to be replicated once or even not at all but...

End definition for \prg_replicate:nn and others. These functions are documented on page 39.)

```

2777 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
2778 { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_vertical:TF`. This function is documented on page 40.)

`\mode_if_horizontal_p:` For testing horizontal mode.

```

\mode_if_horizontal:TF
2779 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
2780 { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_horizontal:TF`. This function is documented on page 39.)

`\mode_if_inner_p:` For testing inner mode.

```

\mode_if_inner:TF
2781 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
2782 { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_inner:TF`. This function is documented on page 40.)

`\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.

```

\mode_if_math:TF
2783 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
2784 { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_math:TF`. This function is documented on page 40.)

5.8 Internal programming functions

`\group_align_safe_begin:` \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 will get some sort of weird error message because the underlying `\futurelet` will store the token at the end of the alignment template. This could be a `&_4` 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.

```

2785 \cs_new:Npn \group_align_safe_begin:
2786 { \if_int_compare:w \if_false: { \fi: ‘} = \c_zero \fi: }
2787 \cs_new:Npn \group_align_safe_end:
2788 { \if_int_compare:w ‘{ = \c_zero } \fi: }

```

(End definition for `\group_align_safe_begin:` and `\group_align_safe_end:.`)

```

2789 <@@=prg>

```

`\g__prg_map_int` A nesting counter for mapping.

```

2790 \int_new:N \g__prg_map_int

```

(End definition for `\g__prg_map_int.`)

`__prg_break_point:Nn` These are defined in `l3basics`, as they are needed “early”. This is just a reminder that is the case!
`__prg_map_break:Nn`

(End definition for `_prg_break_point:Nn` and `_prg_map_break:Nn`.)

`_prg_break_point:` Also done in `l3basics` as in format mode these are needed within `l3alloc`.
`_prg_break:`
`_prg_break:n` (End definition for `_prg_break_point:`, `_prg_break:`, and `_prg_break:n`.)
2791 `</initex | package>`

6 l3quark implementation

The following test files are used for this code: `m3quark001.lvt`.

2792 `<*initex | package>`

6.1 Quarks

2793 `<@@=quark>`

`\quark_new:N` Allocate a new quark.

2794 `\cs_new_protected:Npn \quark_new:N #1 { \tl_const:Nn #1 {#1} }`

(End definition for `\quark_new:N`. This function is documented on page 42.)

`\q_nil` Some “public” quarks. `\q_stop` is an “end of argument” marker, `\q_nil` is a empty value
`\q_mark` and `\q_no_value` marks an empty argument.

`\q_no_value` 2795 `\quark_new:N \q_nil`
`\q_stop` 2796 `\quark_new:N \q_mark`
2797 `\quark_new:N \q_no_value`
2798 `\quark_new:N \q_stop`

(End definition for `\q_nil` and others. These variables are documented on page 43.)

`\q_recursion_tail` Quarks for ending recursions. Only ever used there! `\q_recursion_tail` is appended to
`\q_recursion_stop` 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.

2799 `\quark_new:N \q_recursion_tail`
2800 `\quark_new:N \q_recursion_stop`

(End definition for `\q_recursion_tail` and `\q_recursion_stop`. These variables are documented on page 44.)

`\quark_if_recursion_tail_stop:N` When doing recursions, it is easy to spend a lot of time testing if the end marker has
`\quark_if_recursion_tail_stop_do:Nn` 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 wrapper 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.

2801 `\cs_new:Npn \quark_if_recursion_tail_stop:N #1`
2802 `{`
2803 `\if_meaning:w \q_recursion_tail #1`
2804 `\exp_after:wN \use_none_delimit_by_q_recursion_stop:w`
2805 `\fi:`
2806 `}`
2807 `\cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1`

```

2808 {
2809   \if_meaning:w \q_recursion_tail #1
2810   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2811   \else:
2812     \exp_after:wN \use_none:n
2813   \fi:
2814 }

```

(End definition for \quark_if_recursion_tail_stop:N and \quark_if_recursion_tail_stop_do:Nn. These functions are documented on page 44.)

\quark_if_recursion_tail_stop:n 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.

```

\quark_if_recursion_tail_stop:o
\quark_if_recursion_tail_stop_do:nn
\quark_if_recursion_tail_stop_do:on
\__quark_if_recursion_tail:w
2815 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
2816 {
2817   \tl_if_empty:oTF
2818   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
2819   { \use_none_delimit_by_q_recursion_stop:w }
2820   { }
2821 }
2822 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
2823 {
2824   \tl_if_empty:oTF
2825   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
2826   { \use_i_delimit_by_q_recursion_stop:nw }
2827   { \use_none:n }
2828 }
2829 \cs_new:Npn \__quark_if_recursion_tail:w
2830 #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
2831 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
2832 \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 44.)

__quark_if_recursion_tail_break:NN Analogs of the \quark_if_recursion_tail_stop... functions. Break the mapping using #2.

```

\__quark_if_recursion_tail_break:nN
2833 \cs_new:Npn \__quark_if_recursion_tail_break:NN #1#2
2834 {
2835   \if_meaning:w \q_recursion_tail #1
2836   \exp_after:wN #2
2837   \fi:
2838 }
2839 \cs_new:Npn \__quark_if_recursion_tail_break:nN #1#2
2840 {
2841   \tl_if_empty:oTF
2842   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
2843   { #2 }
2844   { }
2845 }

```

(End definition for __quark_if_recursion_tail_break:NN and __quark_if_recursion_tail_break:nN.)

```

\quark_if_nil_p:N Here we test if we found a special quark as the first argument. We better start with
\quark_if_nil:N\TF \q_no_value as the first argument since the whole thing may otherwise loop if #1 is
\quark_if_no_value_p:N wrongly given a string like aabc instead of a single token.7
\quark_if_no_value_p:c
\quark_if_no_value:N\TF
\quark_if_no_value:c\TF
2846 \prg_new_conditional:Nnn \quark_if_nil:N { p, T , F , TF }
2847 {
2848   \if_meaning:w \q_nil #1
2849   \prg_return_true:
2850   \else:
2851     \prg_return_false:
2852   \fi:
2853 }
2854 \prg_new_conditional:Nnn \quark_if_no_value:N { p, T , F , TF }
2855 {
2856   \if_meaning:w \q_no_value #1
2857   \prg_return_true:
2858   \else:
2859     \prg_return_false:
2860   \fi:
2861 }
2862 \cs_generate_variant:Nn \quark_if_no_value_p:N { c }
2863 \cs_generate_variant:Nn \quark_if_no_value:NT { c }
2864 \cs_generate_variant:Nn \quark_if_no_value:NF { c }
2865 \cs_generate_variant:Nn \quark_if_no_value:NTF { c }

```

(End definition for `\quark_if_nil:N\TF` and `\quark_if_no_value:NTF`. These functions are documented on page 43.)

```

\quark_if_nil_p:n Let us explain \quark_if_nil:n(TF). Expanding \__quark_if_nil:w once is safe
\quark_if_nil_p:V thanks to the trailing \q_nil ??!. The result of expanding once is empty if and only
\quark_if_nil_p:o if both delimited arguments #1 and #2 are empty and #3 is delimited by the last to-
\quark_if_nil:n\TF kens ?!. Thanks to the leading {}, the argument #1 is empty if and only if the argument
\quark_if_nil:V\TF of \quark_if_nil:n starts with \q_nil. The argument #2 is empty if and only if this
\quark_if_nil:o\TF \q_nil is followed immediately by ? or by {}, coming either from the trailing tokens in
\quark_if_no_value_p:n the definition of \quark_if_nil:n, or from its argument. In the first case, \__quark-
\quark_if_no_value:n\TF if_nil:w is followed by {} \q_nil {}? ! \q_nil ?! , hence #3 is delimited by the final ?!,
\__quark_if_nil:w and the test returns true as wanted. In the second case, the result is not empty since
\__quark_if_no_value:w the first ?! in the definition of \quark_if_nil:n stop #3.
2866 \prg_new_conditional:Nnn \quark_if_nil:n { p, T , F , TF }
2867 {
2868   \__tl_if_empty_return:o
2869   { \__quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
2870 }
2871 \cs_new:Npn \__quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
2872 \prg_new_conditional:Nnn \quark_if_no_value:n { p, T , F , TF }
2873 {
2874   \__tl_if_empty_return:o
2875   { \__quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
2876 }
2877 \cs_new:Npn \__quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
2878 \cs_generate_variant:Nn \quark_if_nil_p:n { V , o }
2879 \cs_generate_variant:Nn \quark_if_nil:n\TF { V , o }
2880 \cs_generate_variant:Nn \quark_if_nil:nT { V , o }

```

⁷It may still loop in special circumstances however!

```
2881 \cs_generate_variant:Nn \quark_if_nil:nF { V , o }
```

(End definition for \quark_if_nil:nTF and others. These functions are documented on page 43.)

\q__tl_act_mark These private quarks are needed by l3tl, but that is loaded before the quark module, hence their definition is deferred.

```
2882 \quark_new:N \q__tl_act_mark
2883 \quark_new:N \q__tl_act_stop
```

(End definition for \q__tl_act_mark and \q__tl_act_stop.)

6.2 Scan marks

```
2884 <@@=scan>
```

\g__scan_marks_tl The list of all scan marks currently declared.

```
2885 \tl_new:N \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.

```
2886 \cs_new_protected:Npn \__scan_new:N #1
2887 {
2888   \tl_if_in:NnTF \g__scan_marks_tl { #1 }
2889   {
2890     \__msg_kernel_error:nxx { kernel } { scanmark-already-defined }
2891     { \token_to_str:N #1 }
2892   }
2893   {
2894     \tl_gput_right:Nn \g__scan_marks_tl {#1}
2895     \cs_new_eq:NN #1 \scan_stop:
2896   }
2897 }
```

(End definition for __scan_new:N.)

\s__stop We only declare one scan mark here, more can be defined by specific modules.

```
2898 \__scan_new:N \s__stop
```

(End definition for \s__stop.)

__use_none_delimit_by_s__stop:w Similar to \use_none_delimit_by_q_stop:w.

```
2899 \cs_new:Npn \__use_none_delimit_by_s__stop:w #1 \s__stop { }
```

(End definition for __use_none_delimit_by_s__stop:w.)

\s__seq This private scan mark is needed by l3seq, but that is loaded before the quark module, hence its definition is deferred.

```
2900 \__scan_new:N \s__seq
```

(End definition for \s__seq.)

```
2901 </initex | package>
```

7 l3token implementation

2902 $\langle *initex | package \rangle$

2903 $\langle @@=char \rangle$

7.1 Manipulating and interrogating character tokens

Simple wrappers around the primitives.

```

\char_set_catcode:nn
\char_value_catcode:n
\char_show_value_catcode:n
2904 \cs_new_protected:Npn \char_set_catcode:nn #1#2
2905 {
2906   \tex_catcode:D \__int_eval:w #1 \__int_eval_end:
2907   = \__int_eval:w #2 \__int_eval_end:
2908 }
2909 \cs_new:Npn \char_value_catcode:n #1
2910 { \tex_the:D \tex_catcode:D \__int_eval:w #1 \__int_eval_end: }
2911 \cs_new_protected:Npn \char_show_value_catcode:n #1
2912 { \__msg_show_wrap: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 49.)

```

\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
2913 \cs_new_protected:Npn \char_set_catcode_escape:N #1
2914 { \char_set_catcode:nn { '#1 } \c_zero }
2915 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
2916 { \char_set_catcode:nn { '#1 } \c_one }
2917 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
2918 { \char_set_catcode:nn { '#1 } \c_two }
2919 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
2920 { \char_set_catcode:nn { '#1 } \c_three }
2921 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
2922 { \char_set_catcode:nn { '#1 } \c_four }
2923 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
2924 { \char_set_catcode:nn { '#1 } \c_five }
2925 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
2926 { \char_set_catcode:nn { '#1 } \c_six }
2927 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
2928 { \char_set_catcode:nn { '#1 } \c_seven }
2929 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
2930 { \char_set_catcode:nn { '#1 } \c_eight }
2931 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
2932 { \char_set_catcode:nn { '#1 } \c_nine }
2933 \cs_new_protected:Npn \char_set_catcode_space:N #1
2934 { \char_set_catcode:nn { '#1 } \c_ten }
2935 \cs_new_protected:Npn \char_set_catcode_letter:N #1
2936 { \char_set_catcode:nn { '#1 } \c_eleven }
2937 \cs_new_protected:Npn \char_set_catcode_other:N #1
2938 { \char_set_catcode:nn { '#1 } \c_twelve }
2939 \cs_new_protected:Npn \char_set_catcode_active:N #1
2940 { \char_set_catcode:nn { '#1 } \c_thirteen }
2941 \cs_new_protected:Npn \char_set_catcode_comment:N #1
2942 { \char_set_catcode:nn { '#1 } \c_fourteen }
2943 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
2944 { \char_set_catcode:nn { '#1 } \c_fifteen }
```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 48.)

```

\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
2945 \cs_new_protected:Npn \char_set_catcode_escape:n #1
2946   { \char_set_catcode:nn {#1} \c_zero }
2947 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
2948   { \char_set_catcode:nn {#1} \c_one }
2949 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
2950   { \char_set_catcode:nn {#1} \c_two }
2951 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
2952   { \char_set_catcode:nn {#1} \c_three }
2953 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
2954   { \char_set_catcode:nn {#1} \c_four }
2955 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
2956   { \char_set_catcode:nn {#1} \c_five }
2957 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
2958   { \char_set_catcode:nn {#1} \c_six }
2959 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
2960   { \char_set_catcode:nn {#1} \c_seven }
2961 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
2962   { \char_set_catcode:nn {#1} \c_eight }
2963 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
2964   { \char_set_catcode:nn {#1} \c_nine }
2965 \cs_new_protected:Npn \char_set_catcode_space:n #1
2966   { \char_set_catcode:nn {#1} \c_ten }
2967 \cs_new_protected:Npn \char_set_catcode_letter:n #1
2968   { \char_set_catcode:nn {#1} \c_eleven }
2969 \cs_new_protected:Npn \char_set_catcode_other:n #1
2970   { \char_set_catcode:nn {#1} \c_twelve }
2971 \cs_new_protected:Npn \char_set_catcode_active:n #1
2972   { \char_set_catcode:nn {#1} \c_thirteen }
2973 \cs_new_protected:Npn \char_set_catcode_comment:n #1
2974   { \char_set_catcode:nn {#1} \c_fourteen }
2975 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
2976   { \char_set_catcode:nn {#1} \c_fifteen }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 49.)

```

\char_set_mathcode:nn
\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
Pretty repetitive, but necessary!
2977 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
2978   {
2979     \tex_mathcode:D \__int_eval:w #1 \__int_eval_end:
2980     = \__int_eval:w #2 \__int_eval_end:
2981   }
2982 \cs_new:Npn \char_value_mathcode:n #1
2983   { \tex_the:D \tex_mathcode:D \__int_eval:w #1 \__int_eval_end: }
2984 \cs_new_protected:Npn \char_show_value_mathcode:n #1
2985   { \__msg_show_wrap:n { > ~ \char_value_mathcode:n {#1} } }
2986 \cs_new_protected:Npn \char_set_lccode:nn #1#2
2987   {
2988     \tex_lccode:D \__int_eval:w #1 \__int_eval_end:
2989     = \__int_eval:w #2 \__int_eval_end:

```

```

2990 }
2991 \cs_new:Npn \char_value_lccode:n #1
2992 { \tex_the:D \tex_lccode:D \__int_eval:w #1\__int_eval_end: }
2993 \cs_new_protected:Npn \char_show_value_lccode:n #1
2994 { \__msg_show_wrap:n { > ~ \char_value_lccode:n {#1} } }
2995 \cs_new_protected:Npn \char_set_uccode:nn #1#2
2996 {
2997   \tex_uccode:D \__int_eval:w #1 \__int_eval_end:
2998   = \__int_eval:w #2 \__int_eval_end:
2999 }
3000 \cs_new:Npn \char_value_uccode:n #1
3001 { \tex_the:D \tex_uccode:D \__int_eval:w #1\__int_eval_end: }
3002 \cs_new_protected:Npn \char_show_value_uccode:n #1
3003 { \__msg_show_wrap:n { > ~ \char_value_uccode:n {#1} } }
3004 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
3005 {
3006   \tex_sfcode:D \__int_eval:w #1 \__int_eval_end:
3007   = \__int_eval:w #2 \__int_eval_end:
3008 }
3009 \cs_new:Npn \char_value_sfcode:n #1
3010 { \tex_the:D \tex_sfcode:D \__int_eval:w #1\__int_eval_end: }
3011 \cs_new_protected:Npn \char_show_value_sfcode:n #1
3012 { \__msg_show_wrap:n { > ~ \char_value_sfcode:n {#1} } }

```

(End definition for `\char_set_mathcode:nn` and others. These functions are documented on page 50.)

`\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`).

```

3013 \seq_new:N \l_char_special_seq
3014 \seq_set_split:Nnn \l_char_special_seq { }
3015 { \ \ " \# \$ \% \& \ \ ^ \_ \{ \} \~ }
3016 \seq_new:N \l_char_active_seq
3017 \seq_set_split:Nnn \l_char_active_seq { }
3018 { \ " \$ \& \^ \_ \~ }

```

(End definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 51.)

7.2 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_set_active_eq:nc`
`\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.

```

3019 \group_begin:
3020   \char_set_catcode_active:N \^^@
3021   \cs_set_protected:Npn \__char_tmp:nN #1#2
3022   {
3023     \cs_new_protected:cpn { #1 :nN } ##1
3024     {
3025       \group_begin:
3026         \char_set_lccode:nn { \^^@ } { ##1 }
3027         \tex_lowercase:D { \group_end: #2 ^^@ }
3028     }

```

```

3029     \cs_new_protected:cpx { #1 :NN } ##1
3030     { \exp_not:c { #1 : nN } { '##1 } }
3031   }
3032   \__char_tmp:nN { char_set_active_eq } \cs_set_eq:NN
3033   \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
3034 \group_end:
3035 \cs_generate_variant:Nn \char_set_active_eq:NN { Nc }
3036 \cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
3037 \cs_generate_variant:Nn \char_set_active_eq:nN { nc }
3038 \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 47.)

```

\char_generate:nn
\__char_generate:nn
\__char_generate_aux:nn
\__char_generate_aux:nnw
  \l__char_tmp_tl
  \c__char_max_int
\__char_generate_invalid_catcode:

```

The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (Xe_{La}TeX, Lua_{TeX}). There are though various issues which are covered below. At the interface layer, turn the two arguments into integers up-front so this is only done once.

```

3039 \cs_new:Npn \char_generate:nn #1#2
3040 {
3041   \exp:w \exp_after:wN \__char_generate_aux:w
3042   \__int_value:w \__int_eval:w #1 \exp_after:wN ;
3043   \__int_value:w \__int_eval:w #2 ;
3044 }
3045 \cs_new:Npn \__char_generate:nn #1#2
3046 {
3047   \exp:w \exp_after:wN
3048   \__char_generate_aux:nnw \exp_after:wN
3049   { \__int_value:w \__int_eval:w #1 \exp_after:wN }
3050   {#2} \exp_end:
3051 }

```

Before doing any actual conversion, first some special case filtering. The `\Ucharcat` primitive cannot make active chars, so that is turned off here: if the primitive gets altered then the code is already in place for 8-bit engines and will kick in for Lua_{TeX} too. Spaces are also banned here as Lua_{TeX} emulation only makes normal (charcode 32 spaces. However, `^^@` 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.

```

3052 \cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
3053 {
3054   \if_int_compare:w #2 = \c_thirteen
3055   \__msg_kernel_expandable_error:nn { kernel } { char-active }
3056   \else:
3057   \if_int_compare:w #2 = \c_ten
3058   \if_int_compare:w #1 = \c_zero
3059   \__msg_kernel_expandable_error:nn { kernel } { char-null-space }
3060   \else:
3061   \__msg_kernel_expandable_error:nn { kernel } { char-space }
3062   \fi:
3063   \else:
3064   \if_int_odd:w 0
3065   \if_int_compare:w #2 < \c_one      1 \fi:
3066   \if_int_compare:w #2 = \c_five    1 \fi:
3067   \if_int_compare:w #2 = \c_nine    1 \fi:
3068   \if_int_compare:w #2 > \c_thirteen 1 \fi: \exp_stop_f:

```



```

3069     \__msg_kernel_expandable_error:nn { kernel }
3070     { char-invalid-catcode }
3071   \else:
3072     \if_int_odd:w 0
3073       \if_int_compare:w #1 < \c_zero      1 \fi:
3074       \if_int_compare:w #1 > \c__char_max_int 1 \fi: \exp_stop_f:
3075       \__msg_kernel_expandable_error:nn { kernel }
3076       { char-out-of-range }
3077     \else:
3078       \__char_generate_aux:nnw {#1} {#2}
3079     \fi:
3080   \fi:
3081 \fi:
3082 \fi:
3083 \exp_end:
3084 }
3085 \tl_new:N \l__char_tmp_tl

```

Engine-dependent definitions are now needed for the implementation. For LuaTeX and recent XeTeX releases 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 for speed. 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.

```

3086 \group_begin:
3087 \*package
3088   \char_set_catcode_active:N \^^L
3089   \cs_set:Npn \^^L { }
3090 \package
3091   \char_set_catcode_other:n { 0 }
3092   \if_int_odd:w 0
3093     \cs_if_exist:NT \luatex_directlua:D { 1 }
3094     \cs_if_exist:NT \utex_charcat:D { 1 } \exp_stop_f:
3095     \int_const:Nn \c__char_max_int { 1114111 }
3096     \cs_if_exist:NTF \luatex_directlua:D
3097     {
3098       \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
3099       {
3100         #3
3101         \exp_after:wN \exp_end:
3102         \luatex_directlua:D { l3kernel.charcat(#1, #2) }
3103       }
3104     }
3105     {
3106       \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
3107       {
3108         #3
3109         \exp_after:wN \exp_end:
3110         \utex_charcat:D #1 ~ #2 ~
3111       }
3112     }
3113   \else:

```

For engines where \Ucharcat isn't available (or emulated) then we have to work in macros, and cover only the 8-bit range. The first stage is to build up a `tl` containing `^^@` 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 `^^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.

```

3114 \int_const:Nn \c__char_max_int { 255 }
3115 \tl_set:Nn \l__char_tmp_tl { \exp_not:N \or: }
3116 \char_set_catcode_group_begin:n { 0 } % {
3117 \tl_put_right:Nn \l__char_tmp_tl { ^^@ \if_false: } }
3118 \char_set_catcode_group_end:n { 0 }
3119 \tl_put_right:Nn \l__char_tmp_tl { { \fi: \exp_not:N \or: ^^@ } % }
3120 \tl_set:Nx \l__char_tmp_tl { \l__char_tmp_tl }
3121 \char_set_catcode_math_toggle:n { 0 }
3122 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }

```

As \TeX will be very unhappy if it finds an alignment character inside a primitive `\halign` even when skipping false branches, some precautions are required. \TeX will be happy if the token is hidden inside `\unexpanded` (which needs to be the primitive). The expansion chain here is required so that the conditional gets cleaned up correctly (other code assumes there is exactly one token to skip during the clean-up).

```

3123 \char_set_catcode_alignment:n { 0 }
3124 \tl_put_right:Nn \l__char_tmp_tl
3125 {
3126 \or:
3127 \etex_unexpanded:D \exp_after:wN
3128 { \exp_after:wN ^^@ \exp_after:wN }
3129 }
3130 \tl_put_right:Nn \l__char_tmp_tl { \or: }
3131 \char_set_catcode_parameter:n { 0 }
3132 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
3133 \char_set_catcode_math_superscript:n { 0 }
3134 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
3135 \char_set_catcode_math_subscript:n { 0 }
3136 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
3137 \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. We also set up active tokens although they are (currently) filtered out by the interface layer (`\Ucharcat` cannot make active tokens).

```

3138 \char_set_catcode_space:n { 0 }
3139 \tl_put_right:Nn \l__char_tmp_tl { \use:n { \or: } ^^@ }
3140 \char_set_catcode_letter:n { 0 }
3141 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
3142 \char_set_catcode_other:n { 0 }
3143 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
3144 \char_set_catcode_active:n { 0 }
3145 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }

```

Convert the above temporary list into a series of constant token lists, one for each character code, using `\tex_lowercase:D` to convert `^^@` in each case. The x-type expansion ensures that `\tex_lowercase:D` receives the contents of the token list. In package mode, `^^L` is awkward hence this is done in three parts. Notice that at this stage `^^@` is active.

```

3146 \cs_set_protected:Npn \__char_tmp:n #1
3147 {
3148 \char_set_lccode:nn { 0 } {#1}

```

```

3149     \char_set_lccode:nn { 32 } {#1}
3150     \exp_args:Nx \tex_lowercase:D
3151     {
3152         \tl_const:Nn
3153         \exp_not:c { c__char_ \__int_to_roman:w #1 _tl }
3154         { \exp_not:o \l__char_tmp_tl }
3155     }
3156 }
3157 \*package>
3158 \int_step_function:nnnN { 0 } { 1 } { 11 } \__char_tmp:n
3159 \group_begin:
3160     \tl_replace_once:Nnn \l__char_tmp_tl { ^^@ } { \ERROR }
3161     \__char_tmp:n { 12 }
3162 \group_end:
3163 \int_step_function:nnnN { 13 } { 1 } { 255 } \__char_tmp:n
3164 \*package>
3165 \*initex>
3166 \int_step_function:nnnN { 0 } { 1 } { 255 } \__char_tmp:n
3167 \*initex>
3168 \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
3169 {
3170     #3
3171     \exp_after:wN \exp_after:wN
3172     \exp_after:wN \exp_end:
3173     \exp_after:wN \exp_after:wN
3174     \if_case:w #2
3175         \exp_last_unbraced:Nv \exp_stop_f:
3176         { c__char_ \__int_to_roman:w #1 _tl }
3177     \fi:
3178 }
3179 \fi:
3180 \group_end:

```

(End definition for `\char_generate:nn` and others. These functions are documented on page 48.)

7.3 Generic tokens

```

3181 \@@=token)

```

`\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` (End definition for `\token_to_meaning:N` and `\token_to_str:N`. These functions are documented on page 52.)
`\token_to_str:c`

`\token_new:Nn` Creates a new token.

```

3182 \cs_new_protected:Npn \token_new:Nn #1#2 { \cs_new_eq:NN #1 #2 }

```

(End definition for `\token_new:Nn`. This function is documented on page 51.)

`\c_group_begin_token` 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 `__chk_if_free_cs:N` check.

`\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`

```

3183 \group_begin:
3184   \__chk_if_free_cs:N \c_group_begin_token
3185   \tex_global:D \tex_let:D \c_group_begin_token {
3186     \__chk_if_free_cs:N \c_group_end_token
3187     \tex_global:D \tex_let:D \c_group_end_token }
3188   \char_set_catcode_math_toggle:N \*
3189   \cs_new_eq:NN \c_math_toggle_token *
3190   \char_set_catcode_alignment:N \*
3191   \cs_new_eq:NN \c_alignment_token *
3192   \cs_new_eq:NN \c_parameter_token #
3193   \cs_new_eq:NN \c_math_superscript_token ^
3194   \char_set_catcode_math_subscript:N \*
3195   \cs_new_eq:NN \c_math_subscript_token *
3196   \__chk_if_free_cs:N \c_space_token
3197   \use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~
3198   \cs_new_eq:NN \c_catcode_letter_token a
3199   \cs_new_eq:NN \c_catcode_other_token 1
3200 \group_end:

```

(End definition for `\c_group_begin_token` and others. These functions are documented on page 51.)

`\c_catcode_active_tl` Not an implicit token!

```

3201 \group_begin:
3202   \char_set_catcode_active:N \*
3203   \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
3204 \group_end:

```

(End definition for `\c_catcode_active_tl`. This variable is documented on page 51.)

7.4 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.
`\token_if_group_begin:NTF`

```

3205 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
3206 {
3207   \if_catcode:w \exp_not:N #1 \c_group_begin_token
3208     \prg_return_true: \else: \prg_return_false: \fi:
3209 }

```

(End definition for `\token_if_group_begin:N`. This function is documented on page 52.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.
`\token_if_group_end:NTF`

```

3210 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
3211 {
3212   \if_catcode:w \exp_not:N #1 \c_group_end_token
3213     \prg_return_true: \else: \prg_return_false: \fi:
3214 }

```

(End definition for `\token_if_group_end:N`. This function is documented on page 52.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.
`\token_if_math_toggle:NTF`

```

3215 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
3216 {

```

```

3217 \if_catcode:w \exp_not:N #1 \c_math_toggle_token
3218 \prg_return_true: \else: \prg_return_false: \fi:
3219 }

```

(End definition for `\token_if_math_toggle:NTF`. This function is documented on page 52.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.

`\token_if_alignment:NTF`

```

3220 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
3221 {
3222 \if_catcode:w \exp_not:N #1 \c_alignment_token
3223 \prg_return_true: \else: \prg_return_false: \fi:
3224 }

```

(End definition for `\token_if_alignment:NTF`. This function is documented on page 52.)

`\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 T_EX 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 will remain after the group.

```

3225 \group_begin:
3226 \cs_set_eq:NN \c_parameter_token \scan_stop:
3227 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
3228 {
3229 \if_catcode:w \exp_not:N #1 \c_parameter_token
3230 \prg_return_true: \else: \prg_return_false: \fi:
3231 }
3232 \group_end:

```

(End definition for `\token_if_parameter:NTF`. This function is documented on page 53.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.

`\token_if_math_superscript:NTF`

```

3233 \prg_new_conditional:Npnn \token_if_math_superscript:N #1
3234 { p , T , F , TF }
3235 {
3236 \if_catcode:w \exp_not:N #1 \c_math_superscript_token
3237 \prg_return_true: \else: \prg_return_false: \fi:
3238 }

```

(End definition for `\token_if_math_superscript:NTF`. This function is documented on page 53.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.

`\token_if_math_subscript:NTF`

```

3239 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
3240 {
3241 \if_catcode:w \exp_not:N #1 \c_math_subscript_token
3242 \prg_return_true: \else: \prg_return_false: \fi:
3243 }

```

(End definition for `\token_if_math_subscript:NTF`. This function is documented on page 53.)

\token_if_space_p:N Check if token is a space token. We use the constant `\c_space_token` for this.

\token_if_space:N \underline{TF}

```
3244 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
3245 {
3246     \if_catcode:w \exp_not:N #1 \c_space_token
3247     \prg_return_true: \else: \prg_return_false: \fi:
3248 }
```

(End definition for \token_if_space:N \underline{TF} . This function is documented on page 53.)

\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:N \underline{TF}

```
3249 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
3250 {
3251     \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
3252     \prg_return_true: \else: \prg_return_false: \fi:
3253 }
```

(End definition for \token_if_letter:N \underline{TF} . This function is documented on page 53.)

\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:N \underline{TF}

```
3254 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
3255 {
3256     \if_catcode:w \exp_not:N #1 \c_catcode_other_token
3257     \prg_return_true: \else: \prg_return_false: \fi:
3258 }
```

(End definition for \token_if_other:N \underline{TF} . This function is documented on page 53.)

\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.

\token_if_active:N \underline{TF}

```
3259 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
3260 {
3261     \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
3262     \prg_return_true: \else: \prg_return_false: \fi:
3263 }
```

(End definition for \token_if_active:N \underline{TF} . This function is documented on page 53.)

\token_if_eq_meaning_p:NN Check if the tokens #1 and #2 have same meaning.

\token_if_eq_meaning:NN \underline{TF}

```
3264 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
3265 {
3266     \if_meaning:w #1 #2
3267     \prg_return_true: \else: \prg_return_false: \fi:
3268 }
```

(End definition for \token_if_eq_meaning:NN \underline{TF} . This function is documented on page 53.)

\token_if_eq_catcode_p:NN Check if the tokens #1 and #2 have same category code.

\token_if_eq_catcode:NN \underline{TF}

```
3269 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
3270 {
3271     \if_catcode:w \exp_not:N #1 \exp_not:N #2
3272     \prg_return_true: \else: \prg_return_false: \fi:
3273 }
```

(End definition for `\token_if_eq_catcode:NNTF`. This function is documented on page 53.)

`\token_if_eq_charcode_p:N` Check if the tokens #1 and #2 have same character code.

`\token_if_eq_charcode:NNTF`

```

3274 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
3275 {
3276   \if_charcode:w \exp_not:N #1 \exp_not:N #2
3277   \prg_return_true: \else: \prg_return_false: \fi:
3278 }

```

(End definition for `\token_if_eq_charcode:NNTF`. This function is documented on page 53.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` will always output something like
`\token_if_macro:NTF` `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`.
`__token_if_macro_p:w` 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^AT_EX3) 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.

```

3279 \use:x
3280 {
3281   \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N ##1
3282   { p , T , F , TF }
3283   {
3284     \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
3285     \exp_not:N \token_to_meaning:N ##1 \tl_to_str:n { ma : }
3286     \exp_not:N \q_stop
3287   }
3288   \cs_new:Npn \exp_not:N \__token_if_macro_p:w
3289   ##1 \tl_to_str:n { ma } ##2 \c_colon_str ##3 \exp_not:N \q_stop
3290 }
3291 {
3292   \if_int_compare:w \__str_if_eq_x:nn { #2 } { cro } = \c_zero
3293   \prg_return_true:
3294   \else:
3295     \prg_return_false:
3296   \fi:
3297 }

```

(End definition for `\token_if_macro:NNTF` and `__token_if_macro_p:w`. These functions are documented on page 54.)

`\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.

```

3298 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
3299 {
3300   \if_catcode:w \exp_not:N #1 \scan_stop:

```

```

3301     \prg_return_true: \else: \prg_return_false: \fi:
3302 }

```

(End definition for `\token_if_cs:NTF`. This function is documented on page 54.)

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that T_EX will temporarily convert `\exp_not:N` $\langle token \rangle$ into `\scan_stop:` if $\langle token \rangle$ 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 T_EX's conditional apparatus).

```

3303 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
3304 {
3305     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
3306     \prg_return_false:
3307     \else:
3308         \if_cs_exist:N #1
3309         \prg_return_true:
3310         \else:
3311             \prg_return_false:
3312         \fi:
3313     \fi:
3314 }

```

(End definition for `\token_if_expandable:NTF`. This function is documented on page 54.)

`__token_delimit_by_char: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 `\q_stop`, and returns the first one and its delimiter. This result will eventually be compared to another string.

```

3315 \group_begin:
3316 \cs_set_protected:Npn __token_tmp:w #1
3317 {
3318     \use:x
3319     {
3320         \cs_new:Npn \exp_not:c { __token_delimit_by_ #1 :w }
3321         #####1 \tl_to_str:n {#1} #####2 \exp_not:N \q_stop
3322         { #####1 \tl_to_str:n {#1} }
3323     }
3324 }
3325 __token_tmp:w { char" }
3326 __token_tmp:w { count }
3327 __token_tmp:w { dimen }
3328 __token_tmp:w { macro }
3329 __token_tmp:w { muskip }
3330 __token_tmp:w { skip }
3331 __token_tmp:w { toks }
3332 \group_end:

```

(End definition for `__token_delimit_by_char:w` and others.)

`\token_if_chardef_p:N` 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

```

\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_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

```


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_x_return:nn` 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 five conditionals, `\cs_if_exist:cT` turns out to be `false`, 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).

```

3333 \group_begin:
3334 \cs_set_protected:Npn \_token_tmp:w #1#2#3
3335 {
3336   \use:x
3337   {
3338     \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } ####1
3339     { p , T , F , TF }
3340     {
3341       \cs_if_exist:cT { tex_ #2 :D }
3342       {
3343         \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 :D }
3344         \exp_not:N \prg_return_false:
3345         \exp_not:N \else:
3346         \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 def:D }
3347         \exp_not:N \prg_return_false:
3348         \exp_not:N \else:
3349       }
3350       \exp_not:N \_str_if_eq_x_return:nn
3351       {
3352         \exp_not:N \exp_after:wN
3353         \exp_not:c { \_token_delimit_by_ #2 :w }
3354         \exp_not:N \token_to_meaning:N ####1
3355         ? \tl_to_str:n {#2} \exp_not:N \q_stop
3356       }
3357       { \exp_not:n {#3} }
3358       \cs_if_exist:cT { tex_ #2 :D }
3359       {
3360         \exp_not:N \fi:
3361         \exp_not:N \fi:

```

```

3362     }
3363   }
3364 }
3365 }
3366 \__token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
3367 \__token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
3368 \__token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
3369 \__token_tmp:w { protected_macro } { macro }
3370   { \tl_to_str:n { \protected } macro }
3371 \__token_tmp:w { protected_long_macro } { macro }
3372   { \token_to_str:N \protected \tl_to_str:n { \long } macro }
3373 \__token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
3374 \__token_tmp:w { int_register } { count } { \token_to_str:N \count }
3375 \__token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
3376 \__token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
3377 \__token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
3378 \group_end:

```

(End definition for `\token_if_chardef:N` and others. These functions are documented on page 54.)

`\token_if_primitive_p:N`

We filter out macros first, because they cause endless trouble later otherwise.

`\token_if_primitive:N`**TF**

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 inexistent 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 `\q_stop`.

The meaning of each one of the five `\...mark` primitives has the form `<letters>:<user material>`. 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.

```

3379 \tex_chardef:D \c__token_A_int = 'A ~ %
3380 \use:x
3381 {
3382   \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N ##1
3383     { p , T , F , TF }
3384     {
3385       \exp_not:N \token_if_macro:NTF ##1
3386       \exp_not:N \prg_return_false:

```

```

3387     {
3388         \exp_not:N \exp_after:wN \exp_not:N \__token_if_primitive:NNw
3389         \exp_not:N \token_to_meaning:N ##1
3390         \tl_to_str:n { : : : } \exp_not:N \q_stop ##1
3391     }
3392 }
3393 \cs_new:Npn \exp_not:N \__token_if_primitive:NNw
3394   ##1##2 ##3 \c_colon_str ##4 \exp_not:N \q_stop
3395   {
3396     \exp_not:N \tl_if_empty:oTF
3397     { \exp_not:N \__token_if_primitive_space:w ##3 ~ }
3398     {
3399       \exp_not:N \__token_if_primitive_loop:N ##3
3400       \c_colon_str \exp_not:N \q_stop
3401     }
3402     { \exp_not:N \__token_if_primitive_nullfont:N }
3403   }
3404 }
3405 \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
3406 \cs_new:Npn \__token_if_primitive_nullfont:N #1
3407   {
3408     \if_meaning:w \tex_nullfont:D #1
3409     \prg_return_true:
3410   \else:
3411     \prg_return_false:
3412   \fi:
3413 }
3414 \cs_new:Npn \__token_if_primitive_loop:N #1
3415   {
3416     \if_int_compare:w '#1 < \c__token_A_int %
3417     \exp_after:wN \__token_if_primitive:Nw
3418     \exp_after:wN #1
3419   \else:
3420     \exp_after:wN \__token_if_primitive_loop:N
3421   \fi:
3422 }
3423 \cs_new:Npn \__token_if_primitive:Nw #1 #2 \q_stop
3424   {
3425     \if:w : #1
3426     \exp_after:wN \__token_if_primitive_undefined:N
3427   \else:
3428     \prg_return_false:
3429     \exp_after:wN \use_none:n
3430   \fi:
3431 }
3432 \cs_new:Npn \__token_if_primitive_undefined:N #1
3433   {
3434     \if_cs_exist:N #1
3435     \prg_return_true:
3436   \else:
3437     \prg_return_false:
3438   \fi:
3439 }

```

(End definition for `\token_if_primitive:NTF` and others. These functions are documented on page 55.)

7.5 Peeking ahead at the next token

3440 <@@=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 Storage tokens which are publicly documented: the token peeked.

\g_peek_token

3441 \cs_new_eq:NN \l_peek_token ?

3442 \cs_new_eq:NN \g_peek_token ?

(End definition for \l_peek_token and \g_peek_token. These variables are documented on page 56.)

\l__peek_search_token The token to search for as an implicit token: cf. \l__peek_search_tl.

3443 \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.

3444 \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.

__peek_true_aux:w

__peek_false:w

__peek_tmp:w

3445 \cs_new:Npn __peek_true:w { }

3446 \cs_new:Npn __peek_true_aux:w { }

3447 \cs_new:Npn __peek_false:w { }

3448 \cs_new:Npn __peek_tmp:w { }

(End definition for __peek_true:w and others.)

\peek_after:Nw Simple wrappers for \futurelet: no arguments absorbed here.

\peek_gafter:Nw

3449 \cs_new_protected:Npn \peek_after:Nw

3450 { \tex_futurelet:D \l_peek_token }

3451 \cs_new_protected:Npn \peek_gafter:Nw

3452 { \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 55.)

__peek_true_remove:w A function to remove the next token and then regain control.

3453 \cs_new_protected:Npn __peek_true_remove:w

3454 {

3455 \group_align_safe_end:

3456 \tex_afterassignment:D __peek_true_aux:w

3457 \cs_set_eq:NN __peek_tmp:w

3458 }

(End definition for __peek_true_remove:w.)

`__peek_token_generic:NNTF` The generic function stores 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.

```

3459 \cs_new_protected:Npn \__peek_token_generic:NNTF #1#2#3#4
3460 {
3461   \cs_set_eq:NN \l__peek_search_token #2
3462   \tl_set:Nn \l__peek_search_tl {#2}
3463   \cs_set:Npx \__peek_true:w
3464   {
3465     \exp_not:N \group_align_safe_end:
3466     \exp_not:n {#3}
3467   }
3468   \cs_set:Npx \__peek_false:w
3469   {
3470     \exp_not:N \group_align_safe_end:
3471     \exp_not:n {#4}
3472   }
3473   \group_align_safe_begin:
3474   \peek_after:Nw #1
3475 }
3476 \cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
3477 { \__peek_token_generic:NNTF #1 #2 {#3} { } }
3478 \cs_new_protected:Npn \__peek_token_generic:NNF #1#2#3
3479 { \__peek_token_generic:NNTF #1 #2 { } {#3} }

```

(End definition for `__peek_token_generic:NNTF`.)

`__peek_token_remove_generic:NNTF` For token removal there needs to be a call to the auxiliary function which does the work.

```

3480 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF #1#2#3#4
3481 {
3482   \cs_set_eq:NN \l__peek_search_token #2
3483   \tl_set:Nn \l__peek_search_tl {#2}
3484   \cs_set_eq:NN \__peek_true:w \__peek_true_remove:w
3485   \cs_set:Npx \__peek_true_aux:w { \exp_not:n {#3} }
3486   \cs_set:Npx \__peek_false:w
3487   {
3488     \exp_not:N \group_align_safe_end:
3489     \exp_not:n {#4}
3490   }
3491   \group_align_safe_begin:
3492   \peek_after:Nw #1
3493 }
3494 \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
3495 { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
3496 \cs_new_protected:Npn \__peek_token_remove_generic:NNF #1#2#3
3497 { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }

```

(End definition for `__peek_token_remove_generic:NNTF`.)

`__peek_execute_branches_meaning:` The meaning test is straight forward.

```

3498 \cs_new:Npn \__peek_execute_branches_meaning:
3499 {
3500   \if_meaning:w \l__peek_token \l__peek_search_token
3501   \exp_after:wN \__peek_true:w

```

```

3502     \else:
3503         \exp_after:wN \__peek_false:w
3504     \fi:
3505 }

```

(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 will only be 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 T_EX'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_tl. The \exp_not:N prevents outer macros (coming from non-L^AT_EX3 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.

```

3506 \cs_new:Npn \__peek_execute_branches_catcode:
3507 { \if_catcode:w \__peek_execute_branches_catcode_aux: }
3508 \cs_new:Npn \__peek_execute_branches_charcode:
3509 { \if_charcode:w \__peek_execute_branches_catcode_aux: }
3510 \cs_new:Npn \__peek_execute_branches_catcode_aux:
3511 {
3512     \if_catcode:w \exp_not:N \l__peek_token \scan_stop:
3513     \exp_after:wN \exp_after:wN
3514     \exp_after:wN \__peek_execute_branches_catcode_auxii:N
3515     \exp_after:wN \exp_not:N
3516     \else:
3517     \exp_after:wN \__peek_execute_branches_catcode_auxiii:
3518     \fi:
3519 }
3520 \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
3521 {
3522     \exp_not:N #1
3523     \exp_after:wN \exp_not:N \l__peek_search_tl
3524     \exp_after:wN \__peek_true:w
3525     \else:
3526     \exp_after:wN \__peek_false:w
3527     \fi:
3528     #1
3529 }
3530 \cs_new:Npn \__peek_execute_branches_catcode_auxiii:

```

```

3531 {
3532     \exp_not:N \l_peek_token
3533     \exp_after:wN \exp_not:N \l_peek_search_tl
3534     \exp_after:wN \_peek_true:w
3535 \else:
3536     \exp_after:wN \_peek_false:w
3537 \fi:
3538 }

```

(End definition for `_peek_execute_branches_catcode:` and others.)

`_peek_ignore_spaces_execute_branches:` This function removes one space token at a time, and calls `_peek_execute_branches:` when encountering the first non-space token. We directly use the primitive meaning test rather than `\token_if_eq_meaning:NNTF` because `\l_peek_token` may be an outer macro (coming from non- \LaTeX 3 packages). Spaces are removed using a side-effect of f-expansion: `\exp:w \exp_end_continue_f:w` removes one space.

```

3539 \cs_new_protected:Npn \_peek_ignore_spaces_execute_branches:
3540 {
3541     \if_meaning:w \l_peek_token \c_space_token
3542     \exp_after:wN \peek_after:Nw
3543     \exp_after:wN \_peek_ignore_spaces_execute_branches:
3544     \exp:w \exp_end_continue_f:w
3545 \else:
3546     \exp_after:wN \_peek_execute_branches:
3547 \fi:
3548 }

```

(End definition for `_peek_ignore_spaces_execute_branches:.`)

`_peek_def:nnnn` The public functions themselves cannot be defined using `\prg_new_conditional:Npnn` and so a couple of auxiliary functions are used. As a result, everything is done inside a group. As a result things are a bit complicated.

`_peek_def:nnnnn`

```

3549 \group_begin:
3550 \cs_set:Npn \_peek_def:nnnn #1#2#3#4
3551 {
3552     \_peek_def:nnnnn {#1} {#2} {#3} {#4} { TF }
3553     \_peek_def:nnnnn {#1} {#2} {#3} {#4} { T }
3554     \_peek_def:nnnnn {#1} {#2} {#3} {#4} { F }
3555 }
3556 \cs_set:Npn \_peek_def:nnnnn #1#2#3#4#5
3557 {
3558     \cs_new_protected:cpx { #1 #5 }
3559     {
3560         \tl_if_empty:nF {#2}
3561         { \exp_not:n { \cs_set_eq:NN \_peek_execute_branches: #2 } }
3562         \exp_not:c { #3 #5 }
3563         \exp_not:n {#4}
3564     }
3565 }

```

(End definition for `_peek_def:nnnn` and `_peek_def:nnnnn`.)

`\peek_catcode:NTF` With everything in place the definitions can take place. First for category codes.
`\peek_catcode_ignore_spaces:NTF`
`\peek_catcode_remove:NTF`
`\peek_catcode_remove_ignore_spaces:NTF`

```

3566 \_peek_def:nnnn { peek_catcode:N }

```

```

3567     { }
3568     { __peek_token_generic:NN }
3569     { \__peek_execute_branches_catcode: }
3570 \__peek_def:nnnn { peek_catcode_ignore_spaces:N }
3571     { \__peek_execute_branches_catcode: }
3572     { __peek_token_generic:NN }
3573     { \__peek_ignore_spaces_execute_branches: }
3574 \__peek_def:nnnn { peek_catcode_remove:N }
3575     { }
3576     { __peek_token_remove_generic:NN }
3577     { \__peek_execute_branches_catcode: }
3578 \__peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }
3579     { \__peek_execute_branches_catcode: }
3580     { __peek_token_remove_generic:NN }
3581     { \__peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_catcode:NTF and others. These functions are documented on page 56.)

\peek_charcode:NTF
\peek_charcode_ignore_spaces:NTF
\peek_charcode_remove:NTF
\peek_charcode_remove_ignore_spaces:NTF

Then for character codes.

```

3582 \__peek_def:nnnn { peek_charcode:N }
3583     { }
3584     { __peek_token_generic:NN }
3585     { \__peek_execute_branches_charcode: }
3586 \__peek_def:nnnn { peek_charcode_ignore_spaces:N }
3587     { \__peek_execute_branches_charcode: }
3588     { __peek_token_generic:NN }
3589     { \__peek_ignore_spaces_execute_branches: }
3590 \__peek_def:nnnn { peek_charcode_remove:N }
3591     { }
3592     { __peek_token_remove_generic:NN }
3593     { \__peek_execute_branches_charcode: }
3594 \__peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
3595     { \__peek_execute_branches_charcode: }
3596     { __peek_token_remove_generic:NN }
3597     { \__peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_charcode:NTF and others. These functions are documented on page 56.)

\peek_meaning:NTF
\peek_meaning_ignore_spaces:NTF
\peek_meaning_remove:NTF
\peek_meaning_remove_ignore_spaces:NTF

Finally for meaning, with the group closed to remove the temporary definition functions.

```

3598 \__peek_def:nnnn { peek_meaning:N }
3599     { }
3600     { __peek_token_generic:NN }
3601     { \__peek_execute_branches_meaning: }
3602 \__peek_def:nnnn { peek_meaning_ignore_spaces:N }
3603     { \__peek_execute_branches_meaning: }
3604     { __peek_token_generic:NN }
3605     { \__peek_ignore_spaces_execute_branches: }
3606 \__peek_def:nnnn { peek_meaning_remove:N }
3607     { }
3608     { __peek_token_remove_generic:NN }
3609     { \__peek_execute_branches_meaning: }
3610 \__peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
3611     { \__peek_execute_branches_meaning: }
3612     { __peek_token_remove_generic:NN }
3613     { \__peek_ignore_spaces_execute_branches: }

```



```
3614 \group_end:
```

(End definition for `\peek_meaning:N` and others. These functions are documented on page 57.)

7.6 Decomposing a macro definition

`\token_get_prefix_spec:N` 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.

```
3615 \exp_args:Nno \use:nn
3616 { \cs_new:Npn \__peek_get_prefix_arg_replacement:wN #1 }
3617 { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
3618 { #4 {#1} {#2} {#3} }
3619 \cs_new:Npn \token_get_prefix_spec:N #1
3620 {
3621   \token_if_macro:NTF #1
3622   {
3623     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3624     \token_to_meaning:N #1 \q_stop \use_i:nnn
3625   }
3626   { \scan_stop: }
3627 }
3628 \cs_new:Npn \token_get_arg_spec:N #1
3629 {
3630   \token_if_macro:NTF #1
3631   {
3632     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3633     \token_to_meaning:N #1 \q_stop \use_ii:nnn
3634   }
3635   { \scan_stop: }
3636 }
3637 \cs_new:Npn \token_get_replacement_spec:N #1
3638 {
3639   \token_if_macro:NTF #1
3640   {
3641     \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3642     \token_to_meaning:N #1 \q_stop \use_iii:nnn
3643   }
3644   { \scan_stop: }
3645 }
```

(End definition for `\token_get_prefix_spec:N` and others. These functions are documented on page 59.)

```
3646 </initex | package>
```

8 l3int implementation

```
3647 <*initex | package>
```

```
3648 <@@=int>
```

The following test files are used for this code: *m3int001,m3int002,m3int03*.

`\c_max_register_int` Done in l3basics.

(End definition for `\c_max_register_int`. This variable is documented on page 72.)

`__int_to_roman:w` Done in l3basics.

`\if_int_compare:w` (End definition for `__int_to_roman:w` and `\if_int_compare:w`.)

`\or:` Done in l3basics.

(End definition for `\or:`. This function is documented on page 73.)

`__int_value:w` Here are the remaining primitives for number comparisons and expressions.

```

\__int_eval:w      3649 \cs_new_eq:NN \__int_value:w      \tex_number:D
\__int_eval_end:  3650 \cs_new_eq:NN \__int_eval:w      \etex_numexpr:D
\if_int_odd:w     3651 \cs_new_eq:NN \__int_eval_end:    \tex_relax:D
\if_case:w       3652 \cs_new_eq:NN \if_int_odd:w      \tex_ifodd:D
                 3653 \cs_new_eq:NN \if_case:w        \tex_ifcase:D

```

(End definition for `__int_value:w` and others.)

8.1 Integer expressions

`\int_eval:n` Wrapper for `__int_eval:w`. Can be used in an integer expression or directly in the input stream. In format mode, there is already a definition in l3alloc for bootstrapping, which is therefore corrected to the “real” version here.

```

3654 \*initex>
3655 \cs_set:Npn \int_eval:n #1
3656 { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
3657 \*initex>
3658 \*package>
3659 \cs_new:Npn \int_eval:n #1
3660 { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
3661 \*package>

```

(End definition for `\int_eval:n`. This function is documented on page 62.)

`\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.

```

\__int_abs:N
\int_max:nn      3662 \cs_new:Npn \int_abs:n #1
\int_min:nn      3663 {
\__int_maxmin:wwN 3664 \__int_value:w \exp_after:wN \__int_abs:N
                 3665 \__int_value:w \__int_eval:w #1 \__int_eval_end:
                 3666 \exp_stop_f:
                 3667 }
                 3668 \cs_new:Npn \__int_abs:N #1
                 3669 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
                 3670 \cs_set:Npn \int_max:nn #1#2
                 3671 {
                 3672 \__int_value:w \exp_after:wN \__int_maxmin:wwN
                 3673 \__int_value:w \__int_eval:w #1 \exp_after:wN ;
                 3674 \__int_value:w \__int_eval:w #2 ;
                 3675 >
                 3676 \exp_stop_f:

```

```

3677 }
3678 \cs_set:Npn \int_min:nn #1#2
3679 {
3680   \__int_value:w \exp_after:wN \__int_maxmin:wwN
3681   \__int_value:w \__int_eval:w #1 \exp_after:wN ;
3682   \__int_value:w \__int_eval:w #2 ;
3683   <
3684   \exp_stop_f:
3685 }
3686 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
3687 {
3688   \if_int_compare:w #1 #3 #2 ~
3689   #1
3690   \else:
3691   #2
3692   \fi:
3693 }

```

(End definition for `\int_abs:n` and others. These functions are documented on page 62.)

`\int_div_truncate:nn` 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 expressions are 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.

```

3694 \cs_new:Npn \int_div_truncate:nn #1#2
3695 {
3696   \__int_value:w \__int_eval:w
3697   \exp_after:wN \__int_div_truncate:NwNw
3698   \__int_value:w \__int_eval:w #1 \exp_after:wN ;
3699   \__int_value:w \__int_eval:w #2 ;
3700   \__int_eval_end:
3701 }
3702 \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
3703 {
3704   \if_meaning:w 0 #1
3705   \c_zero
3706   \else:
3707   (
3708     #1#2
3709     \if_meaning:w - #1 + \else: - \fi:
3710     ( \if_meaning:w - #3 - \fi: #3#4 - \c_one ) / \c_two
3711   )
3712   \fi:
3713   / #3#4
3714 }

```

For the sake of completeness:

```

3715 \cs_new:Npn \int_div_round:nn #1#2
3716 { \__int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }

```

Finally there's the modulus operation.

```

3717 \cs_new:Npn \int_mod:nn #1#2
3718 {
3719   \__int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
3720   \__int_value:w \__int_eval:w #1 \exp_after:wN ;
3721   \__int_value:w \__int_eval:w #2 ;
3722   \__int_eval_end:
3723 }
3724 \cs_new:Npn \__int_mod:ww #1; #2;
3725 { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }

```

(End definition for `\int_div_truncate:nn` and others. These functions are documented on page 63.)

8.2 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, `\int_new:c` `\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.)

```

3726 \*package
3727 \cs_new_protected:Npn \int_new:N #1
3728 {
3729   \__chk_if_free_cs:N #1
3730   \cs:w newcount \cs_end: #1
3731 }
3732 \*package
3733 \cs_generate_variant:Nn \int_new:N { c }

```

(End definition for `\int_new:N`. This function is documented on page 63.)

`\int_const:Nn` 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.

```

\int_const:cn
\__int_constdef:Nw
\c_max_constdef_int
3734 \cs_new_protected:Npn \int_const:Nn #1#2
3735 {
3736   \int_compare:nNnTF {#2} < \c_zero
3737   {
3738     \int_new:N #1
3739     \int_gset:Nn #1 {#2}
3740   }
3741   {
3742     \int_compare:nNnTF {#2} > \c_max_constdef_int
3743     {
3744       \int_new:N #1
3745       \int_gset:Nn #1 {#2}
3746     }
3747     {
3748       \__chk_if_free_cs:N #1
3749       \tex_global:D \__int_constdef:Nw #1 =
3750       \__int_eval:w #2 \__int_eval_end:
3751     }
3752   }
3753 }

```

```

3754 \cs_generate_variant:Nn \int_const:Nn { c }
3755 \if_int_odd:w 0
3756   \cs_if_exist:NT \luatex luatexversion:D { 1 }
3757   \cs_if_exist:NT \uptex_disablecjktoken:D
3758     { \if_int_compare:w \ptex_jis:D "2121 = "3000 ~ 1 \fi: }
3759   \cs_if_exist:NT \xetex_XeTeXversion:D { 1 } ~
3760   \cs_if_exist:NTF \uptex_disablecjktoken:D
3761     { \cs_new_eq:NN \__int_constdef:Nw \uptex_kchardef:D }
3762     { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
3763   \__int_constdef:Nw \c__max_constdef_int 114111 ~
3764 \else:
3765   \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
3766   \tex_mathchardef:D \c__max_constdef_int 32767 ~
3767 \fi:

```

(End definition for `\int_const:Nn`, `__int_constdef:Nw`, and `\c__max_constdef_int`. These functions are documented on page 63.)

`\int_zero:N` Functions that reset an *integer* register to zero.

`\int_zero:c` 3768 \cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero }

`\int_gzero:N` 3769 \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero }

`\int_gzero:c` 3770 \cs_generate_variant:Nn \int_zero:N { c }

3771 \cs_generate_variant:Nn \int_gzero:N { c }

(End definition for `\int_zero:N` and `\int_gzero:N`. These functions are documented on page 63.)

`\int_zero_new:N` Create a register if needed, otherwise clear it.

`\int_zero_new:c` 3772 \cs_new_protected:Npn \int_zero_new:N #1

`\int_gzero_new:N` 3773 { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }

`\int_gzero_new:c` 3774 \cs_new_protected:Npn \int_gzero_new:N #1

3775 { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }

3776 \cs_generate_variant:Nn \int_zero_new:N { c }

3777 \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 63.)

`\int_set_eq:NN` Setting equal means using one integer inside the set function of another.

`\int_set_eq:cN` 3778 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }

`\int_set_eq:Nc` 3779 \cs_generate_variant:Nn \int_set_eq:NN { c }

`\int_set_eq:cc` 3780 \cs_generate_variant:Nn \int_set_eq:NN { Nc , cc }

`\int_gset_eq:NN` 3781 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }

`\int_gset_eq:cN` 3782 \cs_generate_variant:Nn \int_gset_eq:NN { c }

`\int_gset_eq:Nc` 3783 \cs_generate_variant:Nn \int_gset_eq:NN { Nc , cc }

`\int_gset_eq:cc`

(End definition for `\int_set_eq:NN` and `\int_gset_eq:NN`. These functions are documented on page 63.)

`\int_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

`\int_if_exist_p:c` 3784 \prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N

`\int_if_exist:NTF` 3785 { TF , T , F , p }

`\int_if_exist:cTF` 3786 \prg_new_eq_conditional:NNn \int_if_exist:c \cs_if_exist:c

3787 { TF , T , F , p }

(End definition for `\int_if_exist:NTF`. This function is documented on page 64.)

8.3 Setting and incrementing integers

```

\int_add:Nn Adding and subtracting to and from a counter ...
\int_add:cn 3788 \cs_new_protected:Npn \int_add:Nn #1#2
\int_gadd:Nn 3789 { \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
\int_gadd:cn 3790 \cs_new_protected:Npn \int_sub:Nn #1#2
\int_sub:Nn 3791 { \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
\int_sub:cn 3792 \cs_new_protected:Npn \int_gadd:Nn
\int_gsub:Nn 3793 { \tex_global:D \int_add:Nn }
\int_gsub:cn 3794 \cs_new_protected:Npn \int_gsub:Nn
3795 { \tex_global:D \int_sub:Nn }
3796 \cs_generate_variant:Nn \int_add:Nn { c }
3797 \cs_generate_variant:Nn \int_gadd:Nn { c }
3798 \cs_generate_variant:Nn \int_sub:Nn { c }
3799 \cs_generate_variant:Nn \int_gsub:Nn { c }

```

(End definition for `\int_add:Nn` and others. These functions are documented on page 64.)

```

\int_incr:N Incrementing and decrementing of integer registers is done with the following functions.
\int_incr:c 3800 \cs_new_protected:Npn \int_incr:N #1
\int_gincr:N 3801 { \tex_advance:D #1 \c_one }
\int_gincr:c 3802 \cs_new_protected:Npn \int_decr:N #1
\int_decr:N 3803 { \tex_advance:D #1 - \c_one }
\int_decr:c 3804 \cs_new_protected:Npn \int_gincr:N
\int_gdecr:N 3805 { \tex_global:D \int_incr:N }
\int_gdecr:c 3806 \cs_new_protected:Npn \int_gdecr:N
3807 { \tex_global:D \int_decr:N }
3808 \cs_generate_variant:Nn \int_incr:N { c }
3809 \cs_generate_variant:Nn \int_decr:N { c }
3810 \cs_generate_variant:Nn \int_gincr:N { c }
3811 \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End definition for `\int_incr:N` and others. These functions are documented on page 64.)

```

\int_set:Nn As integers are register-based TeX will issue an error if they are not defined. Thus there
\int_set:cn is no need for the checking code seen with token list variables.
\int_gset:Nn 3812 \cs_new_protected:Npn \int_set:Nn #1#2
\int_gset:cn 3813 { #1 ~ \__int_eval:w #2\__int_eval_end: }
3814 \cs_new_protected:Npn \int_gset:Nn { \tex_global:D \int_set:Nn }
3815 \cs_generate_variant:Nn \int_set:Nn { c }
3816 \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End definition for `\int_set:Nn` and `\int_gset:Nn`. These functions are documented on page 64.)

8.4 Using integers

```

\int_use:N Here is how counters are accessed:
\int_use:c 3817 \cs_new_eq:NN \int_use:N \tex_the:D

We hand-code this for some speed gain:
3818 %\cs_generate_variant:Nn \int_use:N { c }
3819 \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 64.)

8.5 Integer expression conditionals

`__prg_compare_error:`
`__prg_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 `__prg_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 `TEX` 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 `__prg_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```

3820 \cs_new_protected:Npn \__prg_compare_error:
3821 {
3822   \if_int_compare:w \c_zero \c_zero \fi:
3823   =
3824   \__prg_compare_error:
3825 }
3826 \cs_new:Npn \__prg_compare_error:Nw
3827 #1#2 \q_stop
3828 {
3829   { }
3830   \c_zero \fi:
3831   \__msg_kernel_expandable_error:nnn
3832     { kernel } { unknown-comparison } {#1}
3833   \prg_return_false:
3834 }
```

(End definition for `__prg_compare_error:` and `__prg_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`

Comparison tests using a simple syntax where only one set of braces is required, additional operators such as `!=` and `>=` are supported, and multiple comparisons can be performed at once, for instance `0 < 5 <= 1`. The idea is to loop through the argument, finding one operand at a time, and comparing it to the previous one. The looping auxiliary `__int_compare:Nw` reads one *operand* and one *comparison* symbol, and leaves roughly

```

<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 will terminate it. If the argument contains no relation symbol, `__prg_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 will be ended by the two odd-looking items `e` and `{=nd_}`, with a trailing `\q_stop` used to grab the entire argument when necessary.

```

3835 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
3836 {
3837   \exp_after:wN \__int_compare:w
3838   \__int_value:w \__int_eval:w #1 \__prg_compare_error:
3839 }
3840 \cs_new:Npn \__int_compare:w #1 \__prg_compare_error:
3841 {
3842   \exp_after:wN \if_false: \__int_value:w
3843   \__int_compare:Nw #1 e { = nd_ } \q_stop
3844 }

```

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 `__prg_compare_error:Nw` raises an error.

```

3845 \cs_new:Npn \__int_compare:Nw #1#2 \q_stop
3846 {
3847   \exp_after:wN \__int_compare:NNw
3848   \__int_to_roman:w - 0 #2 \q_mark
3849   #1#2 \q_stop
3850 }
3851 \cs_new:Npn \__int_compare:NNw #1#2#3 \q_mark
3852 {
3853   \etex_unexpanded:D
3854   \use:c
3855   {
3856     \__int_compare_ \token_to_str:N #1
3857     \if_meaning:w = #2 = \fi:
3858     :NNw
3859   }
3860   \__prg_compare_error:Nw #1
3861 }

```

When the last *operand* 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 *operand*, and #3 is one of `<`, `=`, or `>`. As announced earlier, we leave the *operand* 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 *operand* #2 and the comparison #3, and call `__int_compare:Nw` to look for additional operands, after evaluating the following expression.

```

3862 \cs_new:cpn { \__int_compare_end=:NNw } #1#2#3 e #4 \q_stop
3863 {
3864   {#3} \exp_stop_f:
3865   \prg_return_false: \else: \prg_return_true: \fi:

```



```

3866 }
3867 \cs_new:Npn \__int_compare:nnN #1#2#3
3868 {
3869     {#2} \exp_stop_f:
3870     \prg_return_false: \exp_after:wN \use_none_delimit_by_q_stop:w
3871     \fi:
3872     #1 #2 #3 \exp_after:wN \__int_compare:Nw \__int_value:w \__int_eval:w
3873 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `__prg_compare_error:Nw` *<token>* responsible for error detection.

```

3874 \cs_new:cpn { __int_compare=:NNw } #1#2#3 =
3875 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
3876 \cs_new:cpn { __int_compare:<:NNw } #1#2#3 <
3877 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
3878 \cs_new:cpn { __int_compare:>:NNw } #1#2#3 >
3879 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
3880 \cs_new:cpn { __int_compare==:NNw } #1#2#3 ==
3881 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
3882 \cs_new:cpn { __int_compare!=:NNw } #1#2#3 !=
3883 { \__int_compare:nnN { \if_int_compare:w } {#3} = }
3884 \cs_new:cpn { __int_compare<=:NNw } #1#2#3 <=
3885 { \__int_compare:nnN { \if_int_compare:w } {#3} > }
3886 \cs_new:cpn { __int_compare>=:NNw } #1#2#3 >=
3887 { \__int_compare:nnN { \if_int_compare:w } {#3} < }

```

(End definition for `\int_compare:nTF` and others. These functions are documented on page 65.)

`\int_compare_p:nNn` More efficient but less natural in typing.

```

\int_compare:nNnTF
3888 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
3889 {
3890     \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
3891     \prg_return_true:
3892     \else:
3893     \prg_return_false:
3894     \fi:
3895 }

```

(End definition for `\int_compare:nNnTF`. This function is documented on page 65.)

`\int_case:nn` For integer 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 l3basics.

```

\__int_case:nnTF
\__int_case:nw
\__int_case_end:nw
3896 \cs_new:Npn \int_case:nnTF #1
3897 {
3898     \exp:w
3899     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
3900 }
3901 \cs_new:Npn \int_case:nnT #1#2#3
3902 {
3903     \exp:w
3904     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
3905 }
3906 \cs_new:Npn \int_case:nnF #1#2
3907 {

```

```

3908     \exp:w
3909     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
3910   }
3911   \cs_new:Npn \int_case:nn #1#2
3912   {
3913     \exp:w
3914     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
3915   }
3916   \cs_new:Npn \__int_case:nnTF #1#2#3#4
3917   { \__int_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
3918   \cs_new:Npn \__int_case:nw #1#2#3
3919   {
3920     \int_compare:nNnTF {#1} = {#2}
3921     { \__int_case_end:nw {#3} }
3922     { \__int_case:nw {#1} }
3923   }
3924   \cs_new_eq:NN \__int_case_end:nw \__prg_case_end:nw

```

(End definition for `\int_case:nnTF` and others. These functions are documented on page 66.)

```

\int_if_odd_p:n A predicate function.
\int_if_odd:nTF 3925 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
\int_if_even_p:n 3926 {
\int_if_even:nTF 3927   \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
3928   \prg_return_true:
3929   \else:
3930   \prg_return_false:
3931   \fi:
3932 }
3933 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
3934 {
3935   \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
3936   \prg_return_false:
3937   \else:
3938   \prg_return_true:
3939   \fi:
3940 }

```

(End definition for `\int_if_odd:nTF` and `\int_if_even:nTF`. These functions are documented on page 66.)

8.6 Integer expression loops

`\int_while_do:nn` These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

```

\int_until_do:nn
\int_do_while:nn 3941 \cs_new:Npn \int_while_do:nn #1#2
\int_do_until:nn 3942 {
3943   \int_compare:nT {#1}
3944   {
3945     #2
3946     \int_while_do:nn {#1} {#2}
3947   }
3948 }
3949 \cs_new:Npn \int_until_do:nn #1#2

```

```

3950 {
3951   \int_compare:nF {#1}
3952   {
3953     #2
3954     \int_until_do:nn {#1} {#2}
3955   }
3956 }
3957 \cs_new:Npn \int_do_while:nn #1#2
3958 {
3959   #2
3960   \int_compare:nT {#1}
3961   { \int_do_while:nn {#1} {#2} }
3962 }
3963 \cs_new:Npn \int_do_until:nn #1#2
3964 {
3965   #2
3966   \int_compare:nF {#1}
3967   { \int_do_until:nn {#1} {#2} }
3968 }

```

(End definition for `\int_while_do:nn` and others. These functions are documented on page 67.)

`\int_while_do:nNnn` As above but not using the more natural syntax.

`\int_until_do:nNnn`

`\int_do_while:nNnn`

`\int_do_until:nNnn`

```

3969 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
3970 {
3971   \int_compare:nNnT {#1} #2 {#3}
3972   {
3973     #4
3974     \int_while_do:nNnn {#1} #2 {#3} {#4}
3975   }
3976 }
3977 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
3978 {
3979   \int_compare:nNnF {#1} #2 {#3}
3980   {
3981     #4
3982     \int_until_do:nNnn {#1} #2 {#3} {#4}
3983   }
3984 }
3985 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
3986 {
3987   #4
3988   \int_compare:nNnT {#1} #2 {#3}
3989   { \int_do_while:nNnn {#1} #2 {#3} {#4} }
3990 }
3991 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
3992 {
3993   #4
3994   \int_compare:nNnF {#1} #2 {#3}
3995   { \int_do_until:nNnn {#1} #2 {#3} {#4} }
3996 }

```

(End definition for `\int_while_do:nNnn` and others. These functions are documented on page 67.)

8.7 Integer step functions

`\int_step_function:nnnN`
`__int_step:wwwN`
`__int_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).

```

3997 \cs_new:Npn \int_step_function:nnnN #1#2#3
3998 {
3999   \exp_after:wN \__int_step:wwwN
4000   \__int_value:w \__int_eval:w #1 \exp_after:wN ;
4001   \__int_value:w \__int_eval:w #2 \exp_after:wN ;
4002   \__int_value:w \__int_eval:w #3 ;
4003 }
4004 \cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
4005 {
4006   \int_compare:nNnTF {#2} > \c_zero
4007   { \__int_step:NnnnN > }
4008   {
4009     \int_compare:nNnTF {#2} = \c_zero
4010     {
4011       \__msg_kernel_expandable_error:nnn { kernel } { zero-step } {#4}
4012       \use_none:nnnn
4013     }
4014     { \__int_step:NnnnN < }
4015   }
4016   {#1} {#2} {#3} #4
4017 }
4018 \cs_new:Npn \__int_step:NnnnN #1#2#3#4#5
4019 {
4020   \int_compare:nNnF {#2} #1 {#4}
4021   {
4022     #5 {#2}
4023     \exp_args:NNf \__int_step:NnnnN
4024     #1 { \int_eval:n { #2 + #3 } } {#3} {#4} #5
4025   }
4026 }
```

(End definition for `\int_step_function:nnnN`, `__int_step:wwwN`, and `__int_step:NnnnN`. These functions are documented on page 68.)

`\int_step_inline:nnnn`
`\int_step_variable:nnnNn`
`__int_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 `\int_step_function:nnnN`. We put a `__prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so no breaking function will recognize this break point as its own.

```

4027 \cs_new_protected:Npn \int_step_inline:nnnn
4028 {
4029   \int_gincr:N \g__prg_map_int
4030   \exp_args:NNc \__int_step:NNnnnn
4031   \cs_gset:Npn
4032   { __prg_map_ \int_use:N \g__prg_map_int :w }
4033 }
```

```

4034 \cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
4035 {
4036   \int_gincr:N \g__prg_map_int
4037   \exp_args:NNc \__int_step:NNnnnn
4038   \cs_gset:Npx
4039   { \__prg_map_ \int_use:N \g__prg_map_int :w }
4040   {#1}{#2}{#3}
4041   {
4042     \tl_set:Nn \exp_not:N #4 {##1}
4043     \exp_not:n {#5}
4044   }
4045 }
4046 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
4047 {
4048   #1 #2 ##1 {#6}
4049   \int_step_function:nnnN {#3} {#4} {#5} #2
4050   \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
4051 }

```

(End definition for `\int_step_inline:nnnn`, `\int_step_variable:nnnNn`, and `__int_step:NNnnnn`. These functions are documented on page 68.)

8.8 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```

4052 \cs_new_eq:NN \int_to_arabic:n \int_eval:n

```

(End definition for `\int_to_arabic:n`. This function is documented on page 68.)

`\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.

```

4053 \cs_new:Npn \int_to_symbols:nnn #1#2#3
4054 {
4055   \int_compare:nNnTF {#1} > {#2}
4056   {
4057     \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
4058     {
4059       \int_case:nn
4060       { 1 + \int_mod:nn { #1 - 1 } {#2} }
4061       {#3}
4062     }
4063     {#1} {#2} {#3}
4064   }
4065   { \int_case:nn {#1} {#3} }
4066 }
4067 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
4068 {
4069   \exp_args:Nf \int_to_symbols:nnn
4070   { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}

```

```

4071     #1
4072 }

```

(End definition for `\int_to_symbols:nnn` and `_int_to_symbols:nnnn`. These functions are documented on page 69.)

`\int_to_alph:n` These both use the above function with input functions that make sense for the alphabet in English.

`\int_to_Alph:n`

```

4073 \cs_new:Npn \int_to_alph:n #1
4074 {
4075   \int_to_symbols:nnn {#1} { 26 }
4076   {
4077     { 1 } { a }
4078     { 2 } { b }
4079     { 3 } { c }
4080     { 4 } { d }
4081     { 5 } { e }
4082     { 6 } { f }
4083     { 7 } { g }
4084     { 8 } { h }
4085     { 9 } { i }
4086     { 10 } { j }
4087     { 11 } { k }
4088     { 12 } { l }
4089     { 13 } { m }
4090     { 14 } { n }
4091     { 15 } { o }
4092     { 16 } { p }
4093     { 17 } { q }
4094     { 18 } { r }
4095     { 19 } { s }
4096     { 20 } { t }
4097     { 21 } { u }
4098     { 22 } { v }
4099     { 23 } { w }
4100     { 24 } { x }
4101     { 25 } { y }
4102     { 26 } { z }
4103   }
4104 }
4105 \cs_new:Npn \int_to_Alph:n #1
4106 {
4107   \int_to_symbols:nnn {#1} { 26 }
4108   {
4109     { 1 } { A }
4110     { 2 } { B }
4111     { 3 } { C }
4112     { 4 } { D }
4113     { 5 } { E }
4114     { 6 } { F }
4115     { 7 } { G }
4116     { 8 } { H }
4117     { 9 } { I }
4118     { 10 } { J }

```

```

4119      { 11 } { K }
4120      { 12 } { L }
4121      { 13 } { M }
4122      { 14 } { N }
4123      { 15 } { O }
4124      { 16 } { P }
4125      { 17 } { Q }
4126      { 18 } { R }
4127      { 19 } { S }
4128      { 20 } { T }
4129      { 21 } { U }
4130      { 22 } { V }
4131      { 23 } { W }
4132      { 24 } { X }
4133      { 25 } { Y }
4134      { 26 } { Z }
4135    }
4136  }

```

(End definition for `\int_to_alpha:n` and `\int_to_Alpha:n`. These functions are documented on page 69.)

```

\int_to_base:nn Converting from base ten (#1) to a second base (#2) starts with computing #1: if it is
\int_to_Base:nn a complicated calculation, we shouldn't perform it twice. Then check the sign, store it,
\__int_to_base:nn either - or \c_empty_tl, and feed the absolute value to the next auxiliary function.
\__int_to_Base:nn
\__int_to_base:nnN 4137 \cs_new:Npn \int_to_base:nn #1
\__int_to_Base:nnN 4138 { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
\__int_to_Base:nnN 4139 \cs_new:Npn \int_to_Base:nn #1
\__int_to_base:nnnN 4140 { \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } }
\__int_to_Base:nnnN 4141 \cs_new:Npn \__int_to_base:nn #1#2
\__int_to_letter:n 4142 {
\__int_to_Letter:n 4143   \int_compare:nNnTF {#1} < \c_zero
4144     { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
4145     { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
4146   }
4147 \cs_new:Npn \__int_to_Base:nn #1#2
4148 {
4149   \int_compare:nNnTF {#1} < \c_zero
4150     { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
4151     { \__int_to_Base:nnN {#1} {#2} \c_empty_tl }
4152 }

```

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.

```

4153 \cs_new:Npn \__int_to_base:nnN #1#2#3
4154 {
4155   \int_compare:nNnTF {#1} < {#2}
4156     { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
4157     {
4158       \exp_args:Nf \__int_to_base:nnnN
4159         { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }

```

```

4160         {#1}
4161         {#2}
4162         #3
4163     }
4164 }
4165 \cs_new:Npn \__int_to_base:nnnN #1#2#3#4
4166 {
4167     \exp_args:Nf \__int_to_base:nnN
4168     { \int_div_truncate:nn {#2} {#3} }
4169     {#3}
4170     #4
4171     #1
4172 }
4173 \cs_new:Npn \__int_to_Base:nnN #1#2#3
4174 {
4175     \int_compare:nNnTF {#1} < {#2}
4176     { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
4177     {
4178         \exp_args:Nf \__int_to_Base:nnnN
4179         { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
4180         {#1}
4181         {#2}
4182         #3
4183     }
4184 }
4185 \cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
4186 {
4187     \exp_args:Nf \__int_to_Base:nnN
4188     { \int_div_truncate:nn {#2} {#3} }
4189     {#3}
4190     #4
4191     #1
4192 }

```

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:`.

```

4193 \cs_new:Npn \__int_to_letter:n #1
4194 {
4195     \exp_after:wN \exp_after:wN
4196     \if_case:w \__int_eval:w #1 - \c_ten \__int_eval_end:
4197     a
4198     \or: b
4199     \or: c
4200     \or: d
4201     \or: e
4202     \or: f
4203     \or: g
4204     \or: h
4205     \or: i
4206     \or: j

```



```

4207     \or: k
4208     \or: l
4209     \or: m
4210     \or: n
4211     \or: o
4212     \or: p
4213     \or: q
4214     \or: r
4215     \or: s
4216     \or: t
4217     \or: u
4218     \or: v
4219     \or: w
4220     \or: x
4221     \or: y
4222     \or: z
4223     \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
4224     \fi:
4225 }
4226 \cs_new:Npn \__int_to_Letter:n #1
4227 {
4228     \exp_after:wN \exp_after:wN
4229     \if_case:w \__int_eval:w #1 - \c_ten \__int_eval_end:
4230         A
4231     \or: B
4232     \or: C
4233     \or: D
4234     \or: E
4235     \or: F
4236     \or: G
4237     \or: H
4238     \or: I
4239     \or: J
4240     \or: K
4241     \or: L
4242     \or: M
4243     \or: N
4244     \or: O
4245     \or: P
4246     \or: Q
4247     \or: R
4248     \or: S
4249     \or: T
4250     \or: U
4251     \or: V
4252     \or: W
4253     \or: X
4254     \or: Y
4255     \or: Z
4256     \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
4257     \fi:
4258 }

```

(End definition for `\int_to_base:nn` and others. These functions are documented on page 70.)

`\int_to_bin:n` Wrappers around the generic function.

```

\int_to_hex:n 4259 \cs_new:Npn \int_to_bin:n #1
\int_to_Hex:n 4260 { \int_to_base:nn {#1} { 2 } }
\int_to_oct:n 4261 \cs_new:Npn \int_to_hex:n #1
4262 { \int_to_base:nn {#1} { 16 } }
4263 \cs_new:Npn \int_to_Hex:n #1
4264 { \int_to_Base:nn {#1} { 16 } }
4265 \cs_new:Npn \int_to_oct:n #1
4266 { \int_to_base:nn {#1} { 8 } }

```

(End definition for `\int_to_bin:n` and others. These functions are documented on page 69.)

`\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 will be terminated by the conversion of the Q.

```

\__int_to_roman:N
\__int_to_roman:N
\__int_to_roman_i:w 4267 \cs_new:Npn \int_to_roman:n #1
\__int_to_roman_v:w 4268 {
\__int_to_roman_x:w 4269   \exp_after:wN \__int_to_roman:N
\__int_to_roman_l:w 4270   \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_roman_c:w 4271 }
\__int_to_roman_d:w 4272 \cs_new:Npn \__int_to_roman:N #1
\__int_to_roman_m:w 4273 {
\__int_to_roman_Q:w 4274   \use:c { __int_to_roman_ #1 :w }
\__int_to_Roman_i:w 4275   \__int_to_roman:N
\__int_to_Roman_v:w 4276 }
\__int_to_Roman_x:w 4277 \cs_new:Npn \int_to_Roman:n #1
\__int_to_Roman_l:w 4278 {
\__int_to_Roman_c:w 4279   \exp_after:wN \__int_to_Roman_aux:N
\__int_to_Roman_d:w 4280   \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_Roman_m:w 4281 }
\__int_to_Roman_Q:w 4282 \cs_new:Npn \__int_to_Roman_aux:N #1
4283 {
4284   \use:c { __int_to_Roman_ #1 :w }
4285   \__int_to_Roman_aux:N
4286 }
4287 \cs_new:Npn \__int_to_roman_i:w { i }
4288 \cs_new:Npn \__int_to_roman_v:w { v }
4289 \cs_new:Npn \__int_to_roman_x:w { x }
4290 \cs_new:Npn \__int_to_roman_l:w { l }
4291 \cs_new:Npn \__int_to_roman_c:w { c }
4292 \cs_new:Npn \__int_to_roman_d:w { d }
4293 \cs_new:Npn \__int_to_roman_m:w { m }
4294 \cs_new:Npn \__int_to_roman_Q:w #1 { }
4295 \cs_new:Npn \__int_to_Roman_i:w { I }
4296 \cs_new:Npn \__int_to_Roman_v:w { V }
4297 \cs_new:Npn \__int_to_Roman_x:w { X }
4298 \cs_new:Npn \__int_to_Roman_l:w { L }
4299 \cs_new:Npn \__int_to_Roman_c:w { C }
4300 \cs_new:Npn \__int_to_Roman_d:w { D }
4301 \cs_new:Npn \__int_to_Roman_m:w { M }
4302 \cs_new:Npn \__int_to_Roman_Q:w #1 { }

```

(End definition for `\int_to_roman:n` and others. These functions are documented on page 70.)

8.9 Converting from other formats to integers

`__int_pass_signs:wn` Called as `__int_pass_signs:wn <signs and digits> \q_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 `\q_stop`). More precisely, it deletes any `+` and passes any `-` to the input stream, hence should be called in an integer expression.

```

4303 \cs_new:Npn \__int_pass_signs:wn #1
4304 {
4305   \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
4306   \exp_after:wN \__int_pass_signs:wn
4307   \else:
4308     \exp_after:wN \__int_pass_signs_end:wn
4309     \exp_after:wN #1
4310   \fi:
4311 }
4312 \cs_new:Npn \__int_pass_signs_end:wn #1 \q_stop #2 { #2 #1 }

```

(End definition for `__int_pass_signs:wn` and `__int_pass_signs_end:wn`.)

`\int_from_alph:n` First take care of signs then loop through the input using the recursion quarks. The `__int_from_alph:nN` auxiliary collects in its first argument the value obtained so far, and the auxiliary `__int_from_alph:N` converts one letter to an expression which evaluates to the correct number.

```

4313 \cs_new:Npn \int_from_alph:n #1
4314 {
4315   \int_eval:n
4316   {
4317     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
4318     \q_stop { \__int_from_alph:nN { 0 } }
4319     \q_recursion_tail \q_recursion_stop
4320   }
4321 }
4322 \cs_new:Npn \__int_from_alph:nN #1#2
4323 {
4324   \quark_if_recursion_tail_stop_do:Nn #2 {#1}
4325   \exp_args:Nf \__int_from_alph:nN
4326   { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
4327 }
4328 \cs_new:Npn \__int_from_alph:N #1
4329 { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }

```

(End definition for `\int_from_alph:n`, `__int_from_alph:nN`, and `__int_from_alph:N`. These functions are documented on page 70.)

`\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.

```

4330 \cs_new:Npn \int_from_base:nn #1#2
4331 {
4332   \int_eval:n
4333   {
4334     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}

```

```

4335         \q_stop { \__int_from_base:nnN { 0 } {#2} }
4336         \q_recursion_tail \q_recursion_stop
4337     }
4338 }
4339 \cs_new:Npn \__int_from_base:nnN #1#2#3
4340 {
4341     \quark_if_recursion_tail_stop_do:Nn #3 {#1}
4342     \exp_args:Nf \__int_from_base:nnN
4343     { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
4344     {#2}
4345 }
4346 \cs_new:Npn \__int_from_base:N #1
4347 {
4348     \int_compare:nNnTF { '#1 } < { 58 }
4349     {#1}
4350     { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
4351 }

```

(End definition for `\int_from_base:nn`, `__int_from_base:nnN`, and `__int_from_base:N`. These functions are documented on page 71.)

`\int_from_bin:n` Wrappers around the generic function.

```

\int_from_hex:n 4352 \cs_new:Npn \int_from_bin:n #1
\int_from_oct:n 4353 { \int_from_base:nn {#1} \c_two }
4354 \cs_new:Npn \int_from_hex:n #1
4355 { \int_from_base:nn {#1} \c_sixteen }
4356 \cs_new:Npn \int_from_oct:n #1
4357 { \int_from_base:nn {#1} \c_eight }

```

(End definition for `\int_from_bin:n`, `\int_from_hex:n`, and `\int_from_oct:n`. These functions are documented on page 70.)

`\c__int_from_roman_i_int` Constants used to convert from Roman numerals to integers.

```

\c__int_from_roman_v_int 4358 \int_const:cn { c__int_from_roman_i_int } { 1 }
\c__int_from_roman_x_int 4359 \int_const:cn { c__int_from_roman_v_int } { 5 }
\c__int_from_roman_l_int 4360 \int_const:cn { c__int_from_roman_x_int } { 10 }
\c__int_from_roman_c_int 4361 \int_const:cn { c__int_from_roman_l_int } { 50 }
\c__int_from_roman_d_int 4362 \int_const:cn { c__int_from_roman_c_int } { 100 }
\c__int_from_roman_m_int 4363 \int_const:cn { c__int_from_roman_d_int } { 500 }
\c__int_from_roman_I_int 4364 \int_const:cn { c__int_from_roman_m_int } { 1000 }
\c__int_from_roman_V_int 4365 \int_const:cn { c__int_from_roman_I_int } { 1 }
\c__int_from_roman_X_int 4366 \int_const:cn { c__int_from_roman_V_int } { 5 }
\c__int_from_roman_L_int 4367 \int_const:cn { c__int_from_roman_X_int } { 10 }
\c__int_from_roman_L_int 4368 \int_const:cn { c__int_from_roman_L_int } { 50 }
\c__int_from_roman_C_int 4369 \int_const:cn { c__int_from_roman_C_int } { 100 }
\c__int_from_roman_D_int 4370 \int_const:cn { c__int_from_roman_D_int } { 500 }
\c__int_from_roman_M_int 4371 \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`.

```

4372 \cs_new:Npn \int_from_roman:n #1

```

```

4373 {
4374   \int_eval:n
4375   {
4376     (
4377       \c_zero
4378       \exp_after:wN \_int_from_roman:NN \tl_to_str:n {#1}
4379       \q_recursion_tail \q_recursion_tail \q_recursion_stop
4380     )
4381   }
4382 }
4383 \cs_new:Npn \_int_from_roman:NN #1#2
4384 {
4385   \quark_if_recursion_tail_stop:N #1
4386   \int_if_exist:cF { c__int_from_roman_ #1 _int }
4387   { \_int_from_roman_error:w }
4388   \quark_if_recursion_tail_stop_do:Nn #2
4389   { + \use:c { c__int_from_roman_ #1 _int } }
4390   \int_if_exist:cF { c__int_from_roman_ #2 _int }
4391   { \_int_from_roman_error:w }
4392   \int_compare:nNnTF
4393   { \use:c { c__int_from_roman_ #1 _int } }
4394   <
4395   { \use:c { c__int_from_roman_ #2 _int } }
4396   {
4397     + \use:c { c__int_from_roman_ #2 _int }
4398     - \use:c { c__int_from_roman_ #1 _int }
4399     \_int_from_roman:NN
4400   }
4401   {
4402     + \use:c { c__int_from_roman_ #1 _int }
4403     \_int_from_roman:NN #2
4404   }
4405 }
4406 \cs_new:Npn \_int_from_roman_error:w #1 \q_recursion_stop #2
4407 { #2 * \c_zero - \c_one }

```

(End definition for `\int_from_roman:n`, `_int_from_roman:NN`, and `_int_from_roman_error:w`. These functions are documented on page 71.)

8.10 Viewing integer

`\int_show:N` This is very similar to other registers done using `_kernel_register_show:N`, but `\int_show:c` differs because the variable `#1` may be `\currentgrouplevel` or `\currentgrouptype`, in which case the value must be expanded in the current scope rather than when processing `\iow_wrap:nnnN`.

```

4408 \cs_new_protected:Npn \int_show:N #1
4409 {
4410   \use:x
4411   {
4412     \exp_not:n
4413     { \_msg_show_variable:NNNnn #1 \cs_if_exist:NTF ? { } }
4414     { > ~ \token_to_str:N #1 = \tex_the:D #1 }
4415   }
4416 }

```

```
4417 \cs_generate_variant:Nn \int_show:N { c }
```

(End definition for `\int_show:N`. This function is documented on page 71.)

`\int_show:n` We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output.

```
4418 \cs_new_protected:Npn \int_show:n
4419 { \__msg_show_wrap:Nn \int_eval:n }
```

(End definition for `\int_show:n`. This function is documented on page 71.)

8.11 Constant integers

`\c_zero` Again, in l3basics

(End definition for `\c_zero`. This variable is documented on page 72.)

`\c_one` Low-number values not previously defined.

```

\c_two    4420 \int_const:Nn \c_one      { 1 }
\c_three  4421 \int_const:Nn \c_two    { 2 }
\c_four   4422 \int_const:Nn \c_three  { 3 }
\c_five   4423 \int_const:Nn \c_four   { 4 }
\c_six    4424 \int_const:Nn \c_five   { 5 }
\c_seven  4425 \int_const:Nn \c_six    { 6 }
\c_eight  4426 \int_const:Nn \c_seven  { 7 }
\c_nine   4427 \int_const:Nn \c_eight  { 8 }
\c_ten    4428 \int_const:Nn \c_nine   { 9 }
\c_eleven 4429 \int_const:Nn \c_ten    { 10 }
\c_twelve 4430 \int_const:Nn \c_eleven  { 11 }
\c_thirteen 4431 \int_const:Nn \c_twelve { 12 }
\c_fourteen 4432 \int_const:Nn \c_thirteen { 13 }
\c_fifteen 4433 \int_const:Nn \c_fourteen { 14 }
\c_sixteen 4434 \int_const:Nn \c_fifteen { 15 }
\c_sixteen 4435 \int_const:Nn \c_sixteen { 16 }
```

(End definition for `\c_one` and others. These variables are documented on page 72.)

`\c_minus_one` The actual allocation mechanism is in l3alloc; it requires `\c_one` to be defined. In package mode, reuse `\m@ne`.

```
4436 \*package
4437 \cs_new_eq:NN \c_minus_one \m@ne
4438 \*package
4439 \*initex
4440 \int_const:Nn \c_minus_one { -1 }
4441 \*initex
```

(End definition for `\c_minus_one`. This variable is documented on page 72.)

`\c_thirty_two` One middling value.

```
4442 \int_const:Nn \c_thirty_two { 32 }
```

(End definition for `\c_thirty_two`. This variable is documented on page 72.)

`\c_two_hundred_fifty_five` Two classic mid-range integer constants.

```

\c_two_hundred_fifty_six 4443 \int_const:Nn \c_two_hundred_fifty_five { 255 }
                          4444 \int_const:Nn \c_two_hundred_fifty_six   { 256 }
```

(End definition for `\c_two_hundred_fifty_five` and `\c_two_hundred_fifty_six`. These variables are documented on page 72.)

`\c_one_hundred` Simple runs of powers of ten.
`\c_one_thousand` 4445 `\int_const:Nn \c_one_hundred { 100 }`
`\c_ten_thousand` 4446 `\int_const:Nn \c_one_thousand { 1000 }`
4447 `\int_const:Nn \c_ten_thousand { 10000 }`

(End definition for `\c_one_hundred`, `\c_one_thousand`, and `\c_ten_thousand`. These variables are documented on page 72.)

`\c_max_int` The largest number allowed is $2^{31} - 1$
4448 `\int_const:Nn \c_max_int { 2 147 483 647 }`

(End definition for `\c_max_int`. This variable is documented on page 72.)

`\c_max_char_int` The largest character code is 1114111 (hexadecimal 10FFFF) in X_ƎT_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 $[0, 255]$.

```
4449 \int_const:Nn \c_max_char_int
4450 {
4451   \if_int_odd:w 0
4452     \cs_if_exist:NT \luatex luatexversion:D { 1 }
4453     \cs_if_exist:NT \xetex XeTeXversion:D { 1 } ~
4454     "10FFFF
4455   \else:
4456     "FF
4457   \fi:
4458 }
```

(End definition for `\c_max_char_int`. This variable is documented on page 72.)

8.12 Scratch integers

`\l_tmpa_int` We provide two local and two global scratch counters, maybe we need more or less.

`\l_tmpb_int` 4459 `\int_new:N \l_tmpa_int`
`\g_tmpa_int` 4460 `\int_new:N \l_tmpb_int`
`\g_tmpb_int` 4461 `\int_new:N \g_tmpa_int`
4462 `\int_new:N \g_tmpb_int`

(End definition for `\l_tmpa_int` and others. These variables are documented on page 72.)

4463 `</initex | package>`

9 l3skip implementation

4464 `<*initex | package>`

4465 `<@@=dim>`

9.1 Length primitives renamed

`\if_dim:w` Primitives renamed.
`__dim_eval:w` 4466 `\cs_new_eq:NN \if_dim:w \tex_ifdim:D`
`__dim_eval_end:` 4467 `\cs_new_eq:NN __dim_eval:w \etex_dimexpr:D`
4468 `\cs_new_eq:NN __dim_eval_end: \tex_relax:D`

(End definition for `\if_dim:w`, `__dim_eval:w`, and `__dim_eval_end:`. These functions are documented on page 88.)

9.2 Creating and initialising dim variables

\dim_new:N Allocating $\langle dim \rangle$ registers ...

```
\dim_new:c      4469 (*package)
                4470 \cs_new_protected:Npn \dim_new:N #1
                4471 {
                4472     \__chk_if_free_cs:N #1
                4473     \cs:w newdimen \cs_end: #1
                4474 }
                4475 \</package>
                4476 \cs_generate_variant:Nn \dim_new:N { c }
```

(End definition for `\dim_new:N`. This function is documented on page 75.)

\dim_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\dim_const:cn  4477 \cs_new_protected:Npn \dim_const:Nn #1
                4478 {
                4479     \dim_new:N #1
                4480     \dim_gset:Nn #1
                4481 }
                4482 \cs_generate_variant:Nn \dim_const:Nn { c }
```

(End definition for `\dim_const:Nn`. This function is documented on page 75.)

\dim_zero:N Reset the register to zero.

```
\dim_zero:c      4483 \cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_dim }
\dim_gzero:N      4484 \cs_new_protected:Npn \dim_gzero:N { \tex_global:D \dim_zero:N }
\dim_gzero:c      4485 \cs_generate_variant:Nn \dim_zero:N { c }
                4486 \cs_generate_variant:Nn \dim_gzero:N { c }
```

(End definition for `\dim_zero:N` and `\dim_gzero:N`. These functions are documented on page 75.)

\dim_zero_new:N Create a register if needed, otherwise clear it.

```
\dim_zero_new:c  4487 \cs_new_protected:Npn \dim_zero_new:N #1
\dim_gzero_new:N  4488 { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
\dim_gzero_new:c  4489 \cs_new_protected:Npn \dim_gzero_new:N #1
                4490 { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
                4491 \cs_generate_variant:Nn \dim_zero_new:N { c }
                4492 \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 75.)

\dim_if_exist_p:N Copies of the `cs` functions defined in `l3basics`.

```
\dim_if_exist_p:c  4493 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N
\dim_if_exist:NTF  4494 { TF , T , F , p }
\dim_if_exist:cTF  4495 \prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c
                4496 { TF , T , F , p }
```

(End definition for `\dim_if_exist:NTF`. This function is documented on page 75.)

9.3 Setting dim variables

```

\dim_set:Nn Setting dimensions is easy enough.
\dim_set:cn 4497 \cs_new_protected:Npn \dim_set:Nn #1#2
\dim_gset:Nn 4498 { #1 ~ \__dim_eval:w #2 \__dim_eval_end: }
\dim_gset:cn 4499 \cs_new_protected:Npn \dim_gset:Nn { \tex_global:D \dim_set:Nn }
4500 \cs_generate_variant:Nn \dim_set:Nn { c }
4501 \cs_generate_variant:Nn \dim_gset:Nn { c }

```

(End definition for `\dim_set:Nn` and `\dim_gset:Nn`. These functions are documented on page 76.)

```

\dim_set_eq:NN All straightforward.
\dim_set_eq:cn 4502 \cs_new_protected:Npn \dim_set_eq:NN #1#2 { #1 = #2 }
\dim_set_eq:Nc 4503 \cs_generate_variant:Nn \dim_set_eq:NN { c }
\dim_set_eq:cc 4504 \cs_generate_variant:Nn \dim_set_eq:NN { Nc , cc }
\dim_gset_eq:NN 4505 \cs_new_protected:Npn \dim_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\dim_gset_eq:cn 4506 \cs_generate_variant:Nn \dim_gset_eq:NN { c }
\dim_gset_eq:Nc 4507 \cs_generate_variant:Nn \dim_gset_eq:NN { Nc , cc }
\dim_gset_eq:cc
(End definition for \dim_set_eq:NN and \dim_gset_eq:NN. These functions are documented on page
76.)

```

```

\dim_add:Nn Using by here deals with the (incorrect) case \dimen123.
\dim_add:cn 4508 \cs_new_protected:Npn \dim_add:Nn #1#2
\dim_gadd:Nn 4509 { \tex_advance:D #1 by \__dim_eval:w #2 \__dim_eval_end: }
\dim_gadd:cn 4510 \cs_new_protected:Npn \dim_gadd:Nn { \tex_global:D \dim_add:Nn }
\dim_sub:Nn 4511 \cs_generate_variant:Nn \dim_add:Nn { c }
\dim_sub:cn 4512 \cs_generate_variant:Nn \dim_gadd:Nn { c }
\dim_gsub:Nn 4513 \cs_new_protected:Npn \dim_sub:Nn #1#2
\dim_gsub:cn 4514 { \tex_advance:D #1 by - \__dim_eval:w #2 \__dim_eval_end: }
4515 \cs_new_protected:Npn \dim_gsub:Nn { \tex_global:D \dim_sub:Nn }
4516 \cs_generate_variant:Nn \dim_sub:Nn { c }
4517 \cs_generate_variant:Nn \dim_gsub:Nn { c }

```

(End definition for `\dim_add:Nn` and others. These functions are documented on page 76.)

9.4 Utilities for dimension calculations

```

\dim_abs:n Functions for min, max, and absolute value with only one evaluation. The absolute value
\__dim_abs:N is evaluated by removing a leading - if present.
\dim_max:nn 4518 \cs_new:Npn \dim_abs:n #1
\dim_min:nn 4519 {
\__dim_maxmin:wwN 4520 \exp_after:wN \__dim_abs:N
4521 \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
4522 }
4523 \cs_new:Npn \__dim_abs:N #1
4524 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
4525 \cs_new:Npn \dim_max:nn #1#2
4526 {
4527 \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
4528 \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
4529 \dim_use:N \__dim_eval:w #2 ;
4530 >
4531 \__dim_eval_end:

```

```

4532 }
4533 \cs_new:Npn \dim_min:nn #1#2
4534 {
4535   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
4536   \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
4537   \dim_use:N \__dim_eval:w #2 ;
4538   <
4539   \__dim_eval_end:
4540 }
4541 \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
4542 {
4543   \if_dim:w #1 #3 #2 ~
4544     #1
4545   \else:
4546     #2
4547   \fi:
4548 }

```

(End definition for `\dim_abs:n` and others. These functions are documented on page 76.)

`\dim_ratio:nn` With dimension expressions, something like `10 pt * (5 pt / 10 pt)` will 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.

```

4549 \cs_new:Npn \dim_ratio:nn #1#2
4550 { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
4551 \cs_new:Npn \__dim_ratio:n #1
4552 { \__int_value:w \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_ratio:nn` and `__dim_ratio:n`. These functions are documented on page 77.)

9.5 Dimension expression conditionals

`\dim_compare_p:nNn` Simple comparison.

```

\dim_compare:nNnTF
4553 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
4554 {
4555   \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
4556   \prg_return_true: \else: \prg_return_false: \fi:
4557 }

```

(End definition for `\dim_compare:nNnTF`. This function is documented on page 77.)

`\dim_compare_p:n` 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 `__prg_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 `>`.

```

\__dim_compare:w
\__dim_compare:wNn
\__dim_compare:=:w
\__dim_compare!=:w
\__dim_compare<:w
\__dim_compare>:w
4558 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
4559 {
4560   \exp_after:wN \__dim_compare:w
4561   \dim_use:N \__dim_eval:w #1 \__prg_compare_error:
4562 }
4563 \cs_new:Npn \__dim_compare:w #1 \__prg_compare_error:

```

```

4564 {
4565   \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
4566   \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \q_stop
4567 }
4568 \exp_args:Nno \use:nn
4569 { \cs_new:Npn \__dim_compare:wNN #1 }
4570 { \tl_to_str:n {pt} }
4571 #2#3
4572 {
4573   \if_meaning:w = #3
4574   \use:c { \__dim_compare_#2:w }
4575   \fi:
4576   #1 pt \exp_stop_f:
4577   \prg_return_false:
4578   \exp_after:wN \use_none_delimit_by_q_stop:w
4579   \fi:
4580   \reverse_if:N \if_dim:w #1 pt #2
4581   \exp_after:wN \__dim_compare:wNN
4582   \dim_use:N \__dim_eval:w #3
4583 }
4584 \cs_new:cpn { \__dim_compare_ ! :w }
4585 #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
4586 \cs_new:cpn { \__dim_compare_ = :w }
4587 #1 \__dim_eval:w = { #1 \__dim_eval:w }
4588 \cs_new:cpn { \__dim_compare_ < :w }
4589 #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
4590 \cs_new:cpn { \__dim_compare_ > :w }
4591 #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
4592 \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \q_stop
4593 { #1 \prg_return_false: \else: \prg_return_true: \fi: }

```

(End definition for `\dim_compare:nTF` and others. These functions are documented on page 78.)

`\dim_case:nn` For dimension cases, the first task to fully expand the check condition. The over all idea
`\dim_case:nnTF` is then much the same as for `\str_case:nn(TF)` as described in l3basics.

```

\__dim_case:nnTF
\__dim_case:nw
\__dim_case_end:nw
4594 \cs_new:Npn \dim_case:nnTF #1
4595 {
4596   \exp:w
4597   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} }
4598 }
4599 \cs_new:Npn \dim_case:nnT #1#2#3
4600 {
4601   \exp:w
4602   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
4603 }
4604 \cs_new:Npn \dim_case:nnF #1#2
4605 {
4606   \exp:w
4607   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
4608 }
4609 \cs_new:Npn \dim_case:nn #1#2
4610 {
4611   \exp:w
4612   \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }

```

```

4613 }
4614 \cs_new:Npn \__dim_case:nnTF #1#2#3#4
4615 { \__dim_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
4616 \cs_new:Npn \__dim_case:nw #1#2#3
4617 {
4618   \dim_compare:nNnTF {#1} = {#2}
4619   { \__dim_case_end:nw {#3} }
4620   { \__dim_case:nw {#1} }
4621 }
4622 \cs_new_eq:NN \__dim_case_end:nw \__prg_case_end:nw

```

(End definition for `\dim_case:nnTF` and others. These functions are documented on page 79.)

9.6 Dimension expression loops

`\dim_while_do:nn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

\dim_until_do:nn
\dim_do_while:nn
\dim_do_until:nn
4623 \cs_new:Npn \dim_while_do:nn #1#2
4624 {
4625   \dim_compare:nT {#1}
4626   {
4627     #2
4628     \dim_while_do:nn {#1} {#2}
4629   }
4630 }
4631 \cs_new:Npn \dim_until_do:nn #1#2
4632 {
4633   \dim_compare:nF {#1}
4634   {
4635     #2
4636     \dim_until_do:nn {#1} {#2}
4637   }
4638 }
4639 \cs_new:Npn \dim_do_while:nn #1#2
4640 {
4641   #2
4642   \dim_compare:nT {#1}
4643   { \dim_do_while:nn {#1} {#2} }
4644 }
4645 \cs_new:Npn \dim_do_until:nn #1#2
4646 {
4647   #2
4648   \dim_compare:nF {#1}
4649   { \dim_do_until:nn {#1} {#2} }
4650 }

```

(End definition for `\dim_while_do:nn` and others. These functions are documented on page 80.)

`\dim_while_do:nNnn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

\dim_until_do:nNnn
\dim_do_while:nNnn
\dim_do_until:nNnn
4651 \cs_new:Npn \dim_while_do:nNnn #1#2#3#4
4652 {
4653   \dim_compare:nNnT {#1} #2 {#3}
4654   {

```

```

4655         #4
4656         \dim_while_do:nNnn {#1} #2 {#3} {#4}
4657     }
4658 }
4659 \cs_new:Npn \dim_until_do:nNnn #1#2#3#4
4660 {
4661     \dim_compare:nNnF {#1} #2 {#3}
4662     {
4663         #4
4664         \dim_until_do:nNnn {#1} #2 {#3} {#4}
4665     }
4666 }
4667 \cs_new:Npn \dim_do_while:nNnn #1#2#3#4
4668 {
4669     #4
4670     \dim_compare:nNnT {#1} #2 {#3}
4671     { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
4672 }
4673 \cs_new:Npn \dim_do_until:nNnn #1#2#3#4
4674 {
4675     #4
4676     \dim_compare:nNnF {#1} #2 {#3}
4677     { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
4678 }

```

(End definition for `\dim_while_do:nNnn` and others. These functions are documented on page 80.)

9.7 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

4679 \cs_new:Npn \dim_eval:n #1
4680 { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 80.)

`\dim_use:N` Accessing a $\langle dim \rangle$.

`\dim_use:c` 4681 \cs_new_eq:NN \dim_use:N \tex_the:D

We hand-code this for some speed gain:

```

4682 %\cs_generate_variant:Nn \dim_use:N { c }
4683 \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 80.)

`\dim_to_decimal:n` A function which comes up often enough to deserve a place in the kernel. Evaluate the dimension expression `#1` then remove the trailing `pt`. 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.

```

4684 \cs_new:Npn \dim_to_decimal:n #1
4685 {
4686     \exp_after:wN
4687     \__dim_to_decimal:w \dim_use:N \__dim_eval:w (#1) \__dim_eval_end:
4688 }
4689 \use:x

```

```

4690 {
4691   \cs_new:Npn \exp_not:N \__dim_to_decimal:w
4692     ##1 . ##2 \tl_to_str:n { pt }
4693 }
4694 {
4695   \int_compare:nNnTF {#2} > \c_zero
4696     { #1 . #2 }
4697     { #1 }
4698 }

```

(End definition for `\dim_to_decimal:n` and `__dim_to_decimal:w`. These functions are documented on page 81.)

`\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).

```

4699 \cs_new:Npn \dim_to_decimal_in_bp:n #1
4700 { \dim_to_decimal:n { ( #1 ) * 800 / 803 } }

```

(End definition for `\dim_to_decimal_in_bp:n`. This function is documented on page 81.)

`\dim_to_decimal_in_sp:n` Another hard-coded conversion: this one is necessary to avoid things going off-scale.

```

4701 \cs_new:Npn \dim_to_decimal_in_sp:n #1
4702 { \int_eval:n { \__dim_eval:w #1 \__dim_eval_end: } }

```

(End definition for `\dim_to_decimal_in_sp:n`. This function is documented on page 81.)

`\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.

```

4703 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
4704 {
4705   \dim_to_decimal:n
4706   {
4707     1pt *
4708     \dim_ratio:nn {#1} {#2}
4709   }
4710 }

```

(End definition for `\dim_to_decimal_in_unit:nn`. This function is documented on page 81.)

`\dim_to_fp:n` Defined in `l3fp-convert`, documented here.

(End definition for `\dim_to_fp:n`. This function is documented on page 82.)

9.8 Viewing dim variables

`\dim_show:N` Diagnostics.

```

\dim_show:c 4711 \cs_new_eq:NN \dim_show:N \__kernel_register_show:N
4712 \cs_generate_variant:Nn \dim_show:N { c }

```

(End definition for `\dim_show:N`. This function is documented on page 82.)

`\dim_show:n` Diagnostics. We don't use the TeX primitive `\showthe` to show dimension expressions: this gives a more unified output.

```

4713 \cs_new_protected:Npn \dim_show:n
4714 { \__msg_show_wrap:Nn \dim_eval:n }

```

(End definition for `\dim_show:n`. This function is documented on page 82.)

9.9 Constant dimensions

`\c_zero_dim` Constant dimensions.

```
\c_max_dim 4715 \dim_const:Nn \c_zero_dim { 0 pt }
4716 \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 82.)

9.10 Scratch dimensions

`\l_tmpa_dim` We provide two local and two global scratch registers, maybe we need more or less.

```
\l_tmpb_dim 4717 \dim_new:N \l_tmpa_dim
\g_tmpa_dim 4718 \dim_new:N \l_tmpb_dim
\g_tmpb_dim 4719 \dim_new:N \g_tmpa_dim
4720 \dim_new:N \g_tmpb_dim
```

(End definition for `\l_tmpa_dim` and others. These variables are documented on page 82.)

9.11 Creating and initialising skip variables

`\skip_new:N` Allocation of a new internal registers.

```
\skip_new:c 4721 {*\package}
4722 \cs_new_protected:Npn \skip_new:N #1
4723 {
4724     \__chk_if_free_cs:N #1
4725     \cs:w newskip \cs_end: #1
4726 }
4727 \package}
4728 \cs_generate_variant:Nn \skip_new:N { c }
```

(End definition for `\skip_new:N`. This function is documented on page 82.)

`\skip_const:Nn` Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\skip_const:cn 4729 \cs_new_protected:Npn \skip_const:Nn #1
4730 {
4731     \skip_new:N #1
4732     \skip_gset:Nn #1
4733 }
4734 \cs_generate_variant:Nn \skip_const:Nn { c }
```

(End definition for `\skip_const:Nn`. This function is documented on page 83.)

`\skip_zero:N` Reset the register to zero.

```
\skip_zero:c 4735 \cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
\skip_gzero:N 4736 \cs_new_protected:Npn \skip_gzero:N { \tex_global:D \skip_zero:N }
\skip_gzero:c 4737 \cs_generate_variant:Nn \skip_zero:N { c }
4738 \cs_generate_variant:Nn \skip_gzero:N { c }
```

(End definition for `\skip_zero:N` and `\skip_gzero:N`. These functions are documented on page 83.)

`\skip_zero_new:N` Create a register if needed, otherwise clear it.

```

\skip_zero_new:c 4739 \cs_new_protected:Npn \skip_zero_new:N #1
\skip_gzero_new:N 4740 { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\skip_gzero_new:c 4741 \cs_new_protected:Npn \skip_gzero_new:N #1
4742 { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
4743 \cs_generate_variant:Nn \skip_zero_new:N { c }
4744 \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 83.)

```

`\skip_if_exist_p:N` Copies of the cs functions defined in l3basics.

```

\skip_if_exist_p:c 4745 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N
\skip_if_exist:NTF 4746 { TF , T , F , p }
\skip_if_exist:cTF 4747 \prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
4748 { TF , T , F , p }

(End definition for \skip_if_exist:NTF. This function is documented on page 83.)

```

9.12 Setting skip variables

`\skip_set:Nn` Much the same as for dimensions.

```

\skip_set:cn 4749 \cs_new_protected:Npn \skip_set:Nn #1#2
\skip_gset:Nn 4750 { #1 ~ \etex_glueexpr:D #2 \scan_stop: }
\skip_gset:cn 4751 \cs_new_protected:Npn \skip_gset:Nn { \tex_global:D \skip_set:Nn }
4752 \cs_generate_variant:Nn \skip_set:Nn { c }
4753 \cs_generate_variant:Nn \skip_gset:Nn { c }

(End definition for \skip_set:Nn and \skip_gset:Nn. These functions are documented on page 83.)

```

`\skip_set_eq:NN` All straightforward.

```

\skip_set_eq:cn 4754 \cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\skip_set_eq:Nc 4755 \cs_generate_variant:Nn \skip_set_eq:NN { c }
\skip_set_eq:cc 4756 \cs_generate_variant:Nn \skip_set_eq:NN { Nc , cc }
\skip_gset_eq:NN 4757 \cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\skip_gset_eq:cn 4758 \cs_generate_variant:Nn \skip_gset_eq:NN { c }
\skip_gset_eq:Nc 4759 \cs_generate_variant:Nn \skip_gset_eq:NN { Nc , cc }
\skip_gset_eq:cc

(End definition for \skip_set_eq:NN and \skip_gset_eq:NN. These functions are documented on page 83.)

```

`\skip_add:Nn` Using by here deals with the (incorrect) case `\skip123`.

```

\skip_add:cn 4760 \cs_new_protected:Npn \skip_add:Nn #1#2
\skip_gadd:Nn 4761 { \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
\skip_gadd:cn 4762 \cs_new_protected:Npn \skip_gadd:Nn { \tex_global:D \skip_add:Nn }
\skip_sub:Nn 4763 \cs_generate_variant:Nn \skip_add:Nn { c }
\skip_sub:cn 4764 \cs_generate_variant:Nn \skip_gadd:Nn { c }
\skip_gsub:Nn 4765 \cs_new_protected:Npn \skip_sub:Nn #1#2
\skip_gsub:cn 4766 { \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
4767 \cs_new_protected:Npn \skip_gsub:Nn { \tex_global:D \skip_sub:Nn }
4768 \cs_generate_variant:Nn \skip_sub:Nn { c }
4769 \cs_generate_variant:Nn \skip_gsub:Nn { c }

(End definition for \skip_add:Nn and others. These functions are documented on page 83.)

```


9.13 Skip expression conditionals

`\skip_if_eq_p:nn` Comparing skips means doing two expansions to make strings, and then testing them.
`\skip_if_eq:nnTF` As a result, only equality is tested.

```

4770 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
4771 {
4772   \if_int_compare:w
4773     \__str_if_eq_x:nn { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
4774     = \c_zero
4775     \prg_return_true:
4776   \else:
4777     \prg_return_false:
4778   \fi:
4779 }

```

(End definition for `\skip_if_eq:nnTF`. This function is documented on page 84.)

`\skip_if_finite_p:n` With ε -TEX, we have an easy access to the order of infinities 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.

```

4780 \cs_set_protected:Npn \__cs_tmp:w #1
4781 {
4782   \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
4783   {
4784     \exp_after:wN \__skip_if_finite:wwNw
4785     \skip_use:N \etex_glueexpr:D ##1 ; \prg_return_false:
4786     #1 ; \prg_return_true: \q_stop
4787   }
4788   \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \q_stop {##3}
4789 }
4790 \exp_args:No \__cs_tmp:w { \tl_to_str:n { fil } }

```

(End definition for `\skip_if_finite:nTF` and `__skip_if_finite:wwNw`. These functions are documented on page 84.)

9.14 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```

4791 \cs_new:Npn \skip_eval:n #1
4792 { \skip_use:N \etex_glueexpr:D #1 \scan_stop: }

```

(End definition for `\skip_eval:n`. This function is documented on page 84.)

`\skip_use:N` Accessing a $\langle skip \rangle$.

```

\skip_use:c
4793 \cs_new_eq:NN \skip_use:N \tex_the:D
4794 %\cs_generate_variant:Nn \skip_use:N { c }
4795 \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 84.)

9.15 Inserting skips into the output

`\skip_horizontal:N` Inserting skips.
`\skip_horizontal:c` 4796 `\cs_new_eq:NN \skip_horizontal:N \tex_hskip:D`
`\skip_horizontal:n` 4797 `\cs_new:Npn \skip_horizontal:n #1`
`\skip_vertical:N` 4798 `{ \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }`
`\skip_vertical:c` 4799 `\cs_new_eq:NN \skip_vertical:N \tex_vskip:D`
`\skip_vertical:n` 4800 `\cs_new:Npn \skip_vertical:n #1`
4801 `{ \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }`
4802 `\cs_generate_variant:Nn \skip_horizontal:N { c }`
4803 `\cs_generate_variant:Nn \skip_vertical:N { c }`

(End definition for `\skip_horizontal:N` and others. These functions are documented on page 85.)

9.16 Viewing skip variables

`\skip_show:N` Diagnostics.
`\skip_show:c` 4804 `\cs_new_eq:NN \skip_show:N __kernel_register_show:N`
4805 `\cs_generate_variant:Nn \skip_show:N { c }`

(End definition for `\skip_show:N`. This function is documented on page 84.)

`\skip_show:n` Diagnostics. We don't use the T_EX primitive `\showthe` to show skip expressions: this gives a more unified output.

4806 `\cs_new_protected:Npn \skip_show:n`
4807 `{ __msg_show_wrap:Nn \skip_eval:n }`

(End definition for `\skip_show:n`. This function is documented on page 84.)

9.17 Constant skips

`\c_zero_skip` Skips with no rubber component are just dimensions but need to terminate correctly.
`\c_max_skip` 4808 `\skip_const:Nn \c_zero_skip { \c_zero_dim }`
4809 `\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 85.)

9.18 Scratch skips

`\l_tmpa_skip` We provide two local and two global scratch registers, maybe we need more or less.
`\l_tmpb_skip` 4810 `\skip_new:N \l_tmpa_skip`
`\g_tmpa_skip` 4811 `\skip_new:N \l_tmpb_skip`
`\g_tmpb_skip` 4812 `\skip_new:N \g_tmpa_skip`
4813 `\skip_new:N \g_tmpb_skip`

(End definition for `\l_tmpa_skip` and others. These variables are documented on page 85.)

9.19 Creating and initialising muskip variables

\muskip_new:N And then we add muskips.

```
\muskip_new:c 4814 (*package)
4815 \cs_new_protected:Npn \muskip_new:N #1
4816 {
4817     \__chk_if_free_cs:N #1
4818     \cs:w newmuskip \cs_end: #1
4819 }
4820 \end{package}
4821 \cs_generate_variant:Nn \muskip_new:N { c }
```

(End definition for \muskip_new:N. This function is documented on page 85.)

\muskip_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\muskip_const:cn 4822 \cs_new_protected:Npn \muskip_const:Nn #1
4823 {
4824     \muskip_new:N #1
4825     \muskip_gset:Nn #1
4826 }
4827 \cs_generate_variant:Nn \muskip_const:Nn { c }
```

(End definition for \muskip_const:Nn. This function is documented on page 86.)

\muskip_zero:N Reset the register to zero.

```
\muskip_zero:c 4828 \cs_new_protected:Npn \muskip_zero:N #1
\muskip_gzero:N 4829 { #1 \c_zero_muskip }
\muskip_gzero:c 4830 \cs_new_protected:Npn \muskip_gzero:N { \tex_global:D \muskip_zero:N }
4831 \cs_generate_variant:Nn \muskip_zero:N { c }
4832 \cs_generate_variant:Nn \muskip_gzero:N { c }
```

(End definition for \muskip_zero:N and \muskip_gzero:N. These functions are documented on page 86.)

\muskip_zero_new:N Create a register if needed, otherwise clear it.

```
\muskip_zero_new:c 4833 \cs_new_protected:Npn \muskip_zero_new:N #1
\muskip_gzero_new:N 4834 { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
\muskip_gzero_new:c 4835 \cs_new_protected:Npn \muskip_gzero_new:N #1
4836 { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
4837 \cs_generate_variant:Nn \muskip_zero_new:N { c }
4838 \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 86.)

\muskip_if_exist_p:N Copies of the cs functions defined in l3basics.

```
\muskip_if_exist_p:c 4839 \prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N
\muskip_if_exist:NTF 4840 { TF , T , F , p }
\muskip_if_exist:cTF 4841 \prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c
4842 { TF , T , F , p }
```

(End definition for \muskip_if_exist:NTF. This function is documented on page 86.)

9.20 Setting muskip variables

`\muskip_set:Nn` This should be pretty familiar.

```

\muskip_set:cn      4843 \cs_new_protected:Npn \muskip_set:Nn #1#2
\muskip_gset:Nn     4844 { #1 ~ \etex_muexpr:D #2 \scan_stop: }
\muskip_gset:cn     4845 \cs_new_protected:Npn \muskip_gset:Nn { \tex_global:D \muskip_set:Nn }
                   4846 \cs_generate_variant:Nn \muskip_set:Nn { c }
                   4847 \cs_generate_variant:Nn \muskip_gset:Nn { c }

```

(End definition for `\muskip_set:Nn` and `\muskip_gset:Nn`. These functions are documented on page 86.)

`\muskip_set_eq:NN` All straightforward.

```

\muskip_set_eq:cn   4848 \cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
\muskip_set_eq:Nc   4849 \cs_generate_variant:Nn \muskip_set_eq:NN { c }
\muskip_set_eq:cc   4850 \cs_generate_variant:Nn \muskip_set_eq:NN { Nc , cc }
\muskip_gset_eq:NN  4851 \cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\muskip_gset_eq:cn  4852 \cs_generate_variant:Nn \muskip_gset_eq:NN { c }
\muskip_gset_eq:Nc  4853 \cs_generate_variant:Nn \muskip_gset_eq:NN { Nc , cc }
\muskip_gset_eq:cc

```

(End definition for `\muskip_set_eq:NN` and `\muskip_gset_eq:NN`. These functions are documented on page 86.)

`\muskip_add:Nn` Using by here deals with the (incorrect) case `\muskip123`.

```

\muskip_add:cn      4854 \cs_new_protected:Npn \muskip_add:Nn #1#2
\muskip_gadd:Nn     4855 { \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }
\muskip_gadd:cn     4856 \cs_new_protected:Npn \muskip_gadd:Nn { \tex_global:D \muskip_add:Nn }
\muskip_sub:Nn      4857 \cs_generate_variant:Nn \muskip_add:Nn { c }
\muskip_sub:cn      4858 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
\muskip_gsub:Nn     4859 \cs_new_protected:Npn \muskip_sub:Nn #1#2
\muskip_gsub:cn     4860 { \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }
                   4861 \cs_new_protected:Npn \muskip_gsub:Nn { \tex_global:D \muskip_sub:Nn }
                   4862 \cs_generate_variant:Nn \muskip_sub:Nn { c }
                   4863 \cs_generate_variant:Nn \muskip_gsub:Nn { c }

```

(End definition for `\muskip_add:Nn` and others. These functions are documented on page 86.)

9.21 Using muskip expressions and variables

`\muskip_eval:n` Evaluating a muskip expression expandably.

```

4864 \cs_new:Npn \muskip_eval:n #1
4865 { \muskip_use:N \etex_muexpr:D #1 \scan_stop: }

```

(End definition for `\muskip_eval:n`. This function is documented on page 87.)

`\muskip_use:N` Accessing a $\langle muskip \rangle$.

```

\muskip_use:c      4866 \cs_new_eq:NN \muskip_use:N \tex_the:D
                   4867 \cs_generate_variant:Nn \muskip_use:N { c }

```

(End definition for `\muskip_use:N`. This function is documented on page 87.)

9.22 Viewing muskip variables

\muskip_show:N Diagnostics.
\muskip_show:c 4868 \cs_new_eq:NN \muskip_show:N __kernel_register_show:N
4869 \cs_generate_variant:Nn \muskip_show:N { c }
(End definition for \muskip_show:N. This function is documented on page 87.)

\muskip_show:n Diagnostics. We don't use the TeX primitive `\showthe` to show muskip expressions: this gives a more unified output.
4870 \cs_new_protected:Npn \muskip_show:n
4871 { __msg_show_wrap:Nn \muskip_eval:n }
(End definition for \muskip_show:n. This function is documented on page 87.)

9.23 Constant muskips

\c_zero_muskip Constant muskips given by their value.
\c_max_muskip 4872 \muskip_const:Nn \c_zero_muskip { 0 mu }
4873 \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 87.)

9.24 Scratch muskips

\l_tmpa_muskip We provide two local and two global scratch registers, maybe we need more or less.
\l_tmpb_muskip 4874 \muskip_new:N \l_tmpa_muskip
4875 \muskip_new:N \l_tmpb_muskip
4876 \muskip_new:N \g_tmpa_muskip
4877 \muskip_new:N \g_tmpb_muskip
(End definition for \l_tmpa_muskip and others. These variables are documented on page 87.)
4878 </initex | package>

10 l3tl implementation

4879 <*initex | package>
4880 <@@=tl>

A token list variable is a TeX macro that holds tokens. By using the ε -TeX primitive `\unexpanded` inside a TeX `\edef` it is possible to store any tokens, including #, in this way.

10.1 Functions

\tl_new:N Creating new token list variables is a case of checking for an existing definition and doing the definition.
\tl_new:c 4881 \cs_new_protected:Npn \tl_new:N #1
4882 {
4883 __chk_if_free_cs:N #1
4884 \cs_gset_eq:NN #1 \c_empty_tl
4885 }
4886 \cs_generate_variant:Nn \tl_new:N { c }

(End definition for `\tl_new:N`. This function is documented on page 89.)

`\tl_const:Nn` Constants are also easy to generate.

```

\tl_const:Nx 4887 \cs_new_protected:Npn \tl_const:Nn #1#2
\tl_const:cn 4888 {
\tl_const:cx 4889   \__chk_if_free_cs:N #1
               4890   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
               4891 }
               4892 \cs_new_protected:Npn \tl_const:Nx #1#2
               4893 {
               4894   \__chk_if_free_cs:N #1
               4895   \cs_gset_nopar:Npx #1 {#2}
               4896 }
               4897 \cs_generate_variant:Nn \tl_const:Nn { c }
               4898 \cs_generate_variant:Nn \tl_const:Nx { c }
```

(End definition for `\tl_const:Nn`. This function is documented on page 89.)

`\tl_clear:N` Clearing a token list variable means setting it to an empty value. Error checking will be sorted out by the parent function.

`\tl_gclear:N`

```

\tl_gclear:c 4899 \cs_new_protected:Npn \tl_clear:N #1
\tl_gclear:c 4900 { \tl_set_eq:NN #1 \c_empty_tl }
\tl_gclear:c 4901 \cs_new_protected:Npn \tl_gclear:N #1
\tl_gclear:c 4902 { \tl_gset_eq:NN #1 \c_empty_tl }
\tl_gclear:c 4903 \cs_generate_variant:Nn \tl_clear:N { c }
\tl_gclear:c 4904 \cs_generate_variant:Nn \tl_gclear:N { c }
```

(End definition for `\tl_clear:N` and `\tl_gclear:N`. These functions are documented on page 89.)

`\tl_clear_new:N` Clearing a token list variable means setting it to an empty value. Error checking will be sorted out by the parent function.

`\tl_gclear_new:N`

```

\tl_clear_new:c 4905 \cs_new_protected:Npn \tl_clear_new:N #1
\tl_clear_new:c 4906 { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
\tl_gclear_new:c 4907 \cs_new_protected:Npn \tl_gclear_new:N #1
\tl_gclear_new:c 4908 { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
\tl_gclear_new:c 4909 \cs_generate_variant:Nn \tl_clear_new:N { c }
\tl_gclear_new:c 4910 \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 90.)

`\tl_set_eq:NN` For setting token list variables equal to each other.

```

\tl_set_eq:Nc 4911 \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
\tl_set_eq:cN 4912 \cs_new_eq:NN \tl_set_eq:cN \cs_set_eq:cN
\tl_set_eq:cc 4913 \cs_new_eq:NN \tl_set_eq:Nc \cs_set_eq:Nc
\tl_gset_eq:NN 4914 \cs_new_eq:NN \tl_set_eq:cc \cs_set_eq:cc
\tl_gset_eq:Nc 4915 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
\tl_gset_eq:cN 4916 \cs_new_eq:NN \tl_gset_eq:cN \cs_gset_eq:cN
\tl_gset_eq:Nc 4917 \cs_new_eq:NN \tl_gset_eq:Nc \cs_gset_eq:Nc
\tl_gset_eq:cc 4918 \cs_new_eq:NN \tl_gset_eq:cc \cs_gset_eq:cc
```

(End definition for `\tl_set_eq:NN` and `\tl_gset_eq:NN`. These functions are documented on page 90.)

`\tl_concat:NNN` Concatenating token lists is easy.

```

\tl_concat:ccc 4919 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
\tl_gconcat:NNN 4920 { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
\tl_gconcat:ccc 4921 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
4922 { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
4923 \cs_generate_variant:Nn \tl_concat:NNN { ccc }
4924 \cs_generate_variant:Nn \tl_gconcat:NNN { ccc }

(End definition for \tl_concat:NNN and \tl_gconcat:NNN. These functions are documented on page 90.)

```

`\tl_if_exist_p:N` Copies of the cs functions defined in l3basics.

```

\tl_if_exist_p:c 4925 \prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
\tl_if_exist:NTF 4926 \prg_new_eq_conditional:NNn \tl_if_exist:c \cs_if_exist:c { TF , T , F , p }
\tl_if_exist:cTF

(End definition for \tl_if_exist:NTF. This function is documented on page 90.)

```

10.2 Constant token lists

`\c_empty_tl` Never full. We need to define that constant before using `\tl_new:N`.

```

4927 \tl_const:Nn \c_empty_tl { }

(End definition for \c_empty_tl. This variable is documented on page 101.)

\c_space_tl A space as a token list (as opposed to as a character).

4928 \tl_const:Nn \c_space_tl { ~ }

(End definition for \c_space_tl. This variable is documented on page 101.)

```

10.3 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 T_EX more or less redundant. The `\tl_set:No` version is done “by hand” as it is used quite a lot.

```

\tl_set:Nv 4929 \cs_new_protected:Npn \tl_set:Nn #1#2
\tl_set:Nf 4930 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
\tl_set:Nx 4931 \cs_new_protected:Npn \tl_set:No #1#2
\tl_set:cn 4932 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
\tl_set:cV 4933 \cs_new_protected:Npn \tl_set:Nx #1#2
\tl_set:cv 4934 { \cs_set_nopar:Npx #1 {#2} }
\tl_set:co 4935 \cs_new_protected:Npn \tl_gset:Nn #1#2
\tl_set:co 4936 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
\tl_set:cf 4937 \cs_new_protected:Npn \tl_gset:No #1#2
\tl_set:cx 4938 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
\tl_gset:Nn 4939 \cs_new_protected:Npn \tl_gset:Nx #1#2
\tl_gset:Nv 4940 { \cs_gset_nopar:Npx #1 {#2} }
\tl_gset:Nv 4941 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
\tl_gset:No 4942 \cs_generate_variant:Nn \tl_set:Nx { c }
\tl_gset:Nf 4943 \cs_generate_variant:Nn \tl_set:Nn { c , co , cV , cv , cf }
\tl_gset:Nx 4944 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
\tl_gset:cn 4945 \cs_generate_variant:Nn \tl_gset:Nx { c }
\tl_gset:cV 4946 \cs_generate_variant:Nn \tl_gset:Nn { c , co , cV , cv , cf }
\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 90.)

```

\tl_put_left:Nn Adding to the left is done directly to gain a little performance.

```

\tl_put_left:NV 4947 \cs_new_protected:Npn \tl_put_left:Nn #1#2
\tl_put_left:No 4948 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\tl_put_left:Nx 4949 \cs_new_protected:Npn \tl_put_left:NV #1#2
\tl_put_left:cn 4950 { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
\tl_put_left:cV 4951 \cs_new_protected:Npn \tl_put_left:No #1#2
\tl_put_left:co 4952 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
\tl_put_left:cx 4953 \cs_new_protected:Npn \tl_put_left:Nx #1#2
\tl_gput_left:Nn 4954 { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
\tl_gput_left:NV 4955 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
\tl_gput_left:No 4956 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\tl_gput_left:Nx 4957 \cs_new_protected:Npn \tl_gput_left:NV #1#2
\tl_gput_left:cn 4958 { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
\tl_gput_left:cV 4959 \cs_new_protected:Npn \tl_gput_left:No #1#2
\tl_gput_left:co 4960 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
\tl_gput_left:cx 4961 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
4962 { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
4963 \cs_generate_variant:Nn \tl_put_left:Nn { c }
4964 \cs_generate_variant:Nn \tl_put_left:NV { c }
4965 \cs_generate_variant:Nn \tl_put_left:No { c }
4966 \cs_generate_variant:Nn \tl_put_left:Nx { c }
4967 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
4968 \cs_generate_variant:Nn \tl_gput_left:NV { c }
4969 \cs_generate_variant:Nn \tl_gput_left:No { c }
4970 \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 90.)

\tl_put_right:Nn The same on the right.

```

\tl_put_right:NV 4971 \cs_new_protected:Npn \tl_put_right:Nn #1#2
\tl_put_right:No 4972 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\tl_put_right:Nx 4973 \cs_new_protected:Npn \tl_put_right:NV #1#2
\tl_put_right:cn 4974 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\tl_put_right:cV 4975 \cs_new_protected:Npn \tl_put_right:No #1#2
\tl_put_right:co 4976 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\tl_put_right:cx 4977 \cs_new_protected:Npn \tl_put_right:Nx #1#2
\tl_gput_right:Nn 4978 { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }
\tl_gput_right:NV 4979 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
\tl_gput_right:No 4980 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\tl_gput_right:Nx 4981 \cs_new_protected:Npn \tl_gput_right:NV #1#2
\tl_gput_right:cn 4982 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\tl_gput_right:cV 4983 \cs_new_protected:Npn \tl_gput_right:No #1#2
\tl_gput_right:co 4984 { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\tl_gput_right:cx 4985 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
4986 { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
4987 \cs_generate_variant:Nn \tl_put_right:Nn { c }
4988 \cs_generate_variant:Nn \tl_put_right:NV { c }
4989 \cs_generate_variant:Nn \tl_put_right:No { c }
4990 \cs_generate_variant:Nn \tl_put_right:Nx { c }
4991 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
4992 \cs_generate_variant:Nn \tl_gput_right:NV { c }
4993 \cs_generate_variant:Nn \tl_gput_right:No { c }
4994 \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 90.)

When used as a package, there is an option to be picky and to check definitions exist. This part of the process is done now, so that variable types based on `tl` (for example `clist`, `seq` and `prop`) will inherit the appropriate definitions. No `\tl_map_...` yet as the mechanisms are not fully in place. Thus instead do a more low level set up for a mapping, as in `l3basics`.

```

4995 \begin{package}
4996 \tex_ifodd:D \l@expl@check@declarations@bool
4997 \cs_set_protected:Npn \__cs_tmp:w #1
4998 {
4999   \if_meaning:w \q_recursion_tail #1
5000   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
5001   \fi:
5002   \use:x
5003   {
5004     \cs_set_protected:Npn #1 \exp_not:n { ##1 ##2 }
5005     {
5006       \__chk_if_exist_var:N \exp_not:n {##1}
5007       \exp_not:o { #1 {##1} {##2} }
5008     }
5009   }
5010   \__cs_tmp:w
5011 }
5012 \__cs_tmp:w
5013 \tl_set:Nn \tl_set:No \tl_set:Nx
5014 \tl_gset:Nn \tl_gset:No \tl_gset:Nx
5015 \tl_put_left:Nn \tl_put_left:NV
5016 \tl_put_left:No \tl_put_left:Nx
5017 \tl_gput_left:Nn \tl_gput_left:NV
5018 \tl_gput_left:No \tl_gput_left:Nx
5019 \tl_put_right:Nn \tl_put_right:NV
5020 \tl_put_right:No \tl_put_right:Nx
5021 \tl_gput_right:Nn \tl_gput_right:NV
5022 \tl_gput_right:No \tl_gput_right:Nx
5023 \q_recursion_tail \q_recursion_stop
5024 \end{package}

```

The two `set_eq` functions are done by hand as the internals there are a bit different.

```

5025 \begin{package}
5026 \cs_set_protected:Npn \tl_set_eq:NN #1#2
5027 {
5028   \__chk_if_exist_var:N #1
5029   \__chk_if_exist_var:N #2
5030   \cs_set_eq:NN #1 #2
5031 }
5032 \cs_set_protected:Npn \tl_gset_eq:NN #1#2
5033 {
5034   \__chk_if_exist_var:N #1
5035   \__chk_if_exist_var:N #2
5036   \cs_gset_eq:NN #1 #2
5037 }
5038 \end{package}

```

There is also a need to check all three arguments of the `concat` functions: a token list #2 or #3 equal to `\scan_stop:` would lead to problems later on.

```

5039 (*package)
5040 \cs_set_protected:Npn \tl_concat:NNN #1#2#3
5041 {
5042   \__chk_if_exist_var:N #1
5043   \__chk_if_exist_var:N #2
5044   \__chk_if_exist_var:N #3
5045   \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} }
5046 }
5047 \cs_set_protected:Npn \tl_gconcat:NNN #1#2#3
5048 {
5049   \__chk_if_exist_var:N #1
5050   \__chk_if_exist_var:N #2
5051   \__chk_if_exist_var:N #3
5052   \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} }
5053 }
5054 \tex_fi:D
5055 </package>

```

10.4 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.

```

5056 \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_set_rescan:NNnn
  \__tl_set_rescan_multi:n
    \__tl_rescan:w

```

These functions use a common auxiliary. After some initial setup explained below, and the user setup #3 (followed by `\scan_stop:` to be safe), the tokens are rescanned by `_tl_set_rescan:n` and stored into `\l__tl_internal_a_tl`, then passed to #1#2 outside the group after expansion. The auxiliary `__tl_set_rescan:n` is defined later: in the simplest case, this auxiliary calls `__tl_set_rescan_multi:n`, whose code is included here to help understand the approach.

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 ...
```

The standard solution is to use an x-expanding assignment and set `\everyeof` to `\exp_not:N` to suppress the error at the end of the file. Since the rescanned tokens should not be expanded, they will be taken as a delimited argument of an auxiliary which wraps them in `\exp_not:n` (in fact `\exp_not:o`, as there is a `\prg_do_nothing:` to avoid losing braces). The delimiter cannot appear within the rescanned token list because it contains twice the same character, with different catcodes.

The difference between single-line and multiple-line files complicates the story, as explained below.

```

5057 \cs_new_protected:Npn \tl_set_rescan:Nnn
5058 { \__tl_set_rescan:NNnn \tl_set:Nn }
5059 \cs_new_protected:Npn \tl_gset_rescan:Nnn
5060 { \__tl_set_rescan:NNnn \tl_gset:Nn }

```

```

5061 \cs_new_protected:Npn \tl_rescan:nn
5062 { \__tl_set_rescan:NNnn \prg_do_nothing: \use:n }
5063 \cs_new_protected:Npn \__tl_set_rescan:NNnn #1#2#3#4
5064 {
5065   \tl_if_empty:nTF {#4}
5066   {
5067     \group_begin:
5068     #3
5069     \group_end:
5070     #1 #2 { }
5071   }
5072   {
5073     \group_begin:
5074     \exp_args:No \etex_everyeof:D { \c__tl_rescan_marker_tl \exp_not:N }
5075     \int_compare:nNnT \tex_endlinechar:D = { 32 }
5076     { \tex_endlinechar:D - \c_one }
5077     \tex_newlinechar:D \tex_endlinechar:D
5078     #3 \scan_stop:
5079     \exp_args:No \__tl_set_rescan:n { \tl_to_str:n {#4} }
5080     \exp_args:NNNo
5081     \group_end:
5082     #1 #2 \l__tl_internal_a_tl
5083   }
5084 }
5085 \cs_new_protected:Npn \__tl_set_rescan_multi:n #1
5086 {
5087   \tl_set:Nx \l__tl_internal_a_tl
5088   {
5089     \exp_after:wN \__tl_rescan:w
5090     \exp_after:wN \prg_do_nothing:
5091     \etex_scantokens:D {#1}
5092   }
5093 }
5094 \exp_args:Nno \use:nn
5095 { \cs_new:Npn \__tl_rescan:w #1 } \c__tl_rescan_marker_tl
5096 { \exp_not:o {#1} }
5097 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno , Nnx }
5098 \cs_generate_variant:Nn \tl_set_rescan:Nnn { c , cno , cnx }
5099 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno , Nnx }
5100 \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 92.)

<code>__tl_set_rescan:n</code> <code>__tl_set_rescan:NnTF</code> <code>__tl_set_rescan_single:nn</code> <code>\tl_set_rescan_single_aux:nn</code>	This function calls <code>__tl_set_rescan_multiple:n</code> or <code>__tl_set_rescan_single:nn</code> { ' } depending on whether its argument is a single-line fragment of code/data or is made of multiple lines by testing for the presence of a <code>\newlinechar</code> character. If <code>\newlinechar</code> is out of range, the argument is assumed to be a single line.
---	--

The case of multiple lines is a straightforward application of `\scantokens` as described above. 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.

For a single line, no `\endlinechar` should be added, so it will be 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), 12 (other) and 13 (active) are accepted, as these are 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). Once a valid character is found, run some code very similar to `__tl_set_rescan_multi:n`, except that `__tl_rescan:w` must be redefined to also remove the additional character (with the appropriate catcode). Getting the delimiter with the right catcode requires using `\scantokens` inside an `x`-expansion, hence using the previous definition of `__tl_rescan:w` as well. The odd `\exp_not:N\use:n` ensures that the trailing `\exp_not:N` in `\everyeof` does not prevent the expansion of `\c__tl_rescan_marker_tl`, but rather of a closing brace (this does nothing). If no valid character is found, similar code is ran, and the only difference is that trailing spaces are not preserved (bear in mind that this only happens if no character between 39 and 127 has catcode letter, other or active).

There is also some work to preserve leading spaces: test whether the first character (given by `\str_head:n`, with an extra space to circumvent a limitation of `f`-expansion) has catcode 10 and add what \TeX would add in the middle of a line for any sequence of such characters: a single space with catcode 10 and character code 32.

```

5101 \group_begin:
5102   \tex_catcode:D '\^~@ = 12 \scan_stop:
5103   \cs_new_protected:Npn \__tl_set_rescan:n #1
5104     {
5105       \int_compare:nNnTF \tex_newlinechar:D < \c_zero
5106         { \use_ii:n }
5107         {
5108           \char_set_lccode:nn { 0 } { \tex_newlinechar:D }
5109           \tex_lowercase:D { \__tl_set_rescan:NnTF ^~@ } {#1}
5110         }
5111         { \__tl_set_rescan_multi:n }
5112         { \__tl_set_rescan_single:nn { ' } }
5113       {#1}
5114     }
5115   \cs_new_protected:Npn \__tl_set_rescan:NnTF #1#2
5116     { \tl_if_in:nnTF {#2} {#1} }
5117   \cs_new_protected:Npn \__tl_set_rescan_single:nn #1
5118     {
5119       \int_compare:nNnTF
5120         { \char_value_catcode:n { '#1 } / \c_three } = \c_four
5121         { \__tl_set_rescan_single_aux:nn {#1} }
5122         {
5123           \int_compare:nNnTF { '#1 } < { '\~ }
5124             {

```

```

5125         \char_set_lccode:nn { 0 } { '#1 + 1 }
5126         \tex_lowercase:D { \_tl_set_rescan_single:nn { ^~@ } }
5127     }
5128     { \_tl_set_rescan_single_aux:nn { } }
5129 }
5130 }
5131 \cs_new_protected:Npn \_tl_set_rescan_single_aux:nn #1#2
5132 {
5133     \tex_endlinechar:D - \c_one
5134     \use:x
5135     {
5136         \exp_not:N \use:n
5137         {
5138             \exp_not:n { \cs_set:Npn \_tl_rescan:w ##1 }
5139             \exp_after:wN \_tl_rescan:w
5140             \exp_after:wN \prg_do_nothing:
5141             \etex_scantokens:D {#1}
5142         }
5143         \c__tl_rescan_marker_tl
5144     }
5145     { \exp_not:o {##1} }
5146     \tl_set:Nx \l__tl_internal_a_tl
5147     {
5148         \int_compare:nNnT
5149         {
5150             \char_value_catcode:n
5151             { \exp_last_unbraced:Nf ' \str_head:n {#2} ~ }
5152         }
5153         = \c_ten { ~ }
5154         \exp_after:wN \_tl_rescan:w
5155         \exp_after:wN \prg_do_nothing:
5156         \etex_scantokens:D { #2 #1 }
5157     }
5158 }
5159 \group_end:

```

(End definition for `_tl_set_rescan:n` and others.)

10.5 Modifying token list variables

`\tl_replace_all:Nnn` All of the `replace` functions call `_tl_replace:NnnNnn` with appropriate arguments. `\tl_replace_all:cnn` 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 $\langle tl\ var \rangle$ $\{ \langle pattern \rangle \}$ $\{ \langle replacement \rangle \}$ provided by the user. When describing the auxiliary functions below, we denote the contents of the $\langle tl\ var \rangle$ by $\langle token\ list \rangle$.

```

5160 \cs_new_protected:Npn \tl_replace_once:Nnn
5161 { \_tl_replace:NnnNnn \q_mark ? \_tl_replace_wrap:w \tl_set:Nx }
5162 \cs_new_protected:Npn \tl_greplace_once:Nnn
5163 { \_tl_replace:NnnNnn \q_mark ? \_tl_replace_wrap:w \tl_gset:Nx }
5164 \cs_new_protected:Npn \tl_replace_all:Nnn
5165 { \_tl_replace:NnnNnn \q_mark ? \_tl_replace_next:w \tl_set:Nx }

```

```

5166 \cs_new_protected:Npn \tl_greplace_all:Nnn
5167   { \__tl_replace:NnnNNNnn \q_mark ? \__tl_replace_next:w \tl_gset:Nx }
5168 \cs_generate_variant:Nn \tl_replace_once:Nnn { c }
5169 \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
5170 \cs_generate_variant:Nn \tl_replace_all:Nnn { c }
5171 \cs_generate_variant:Nn \tl_greplace_all:Nnn { c }

```

(End definition for `\tl_replace_all:Nnn` and others. These functions are documented on page 91.)

```

\__tl_replace:NnnNNNnn
\__tl_replace_auxi:NnnNNNnn
\__tl_replace_auxii:NnnNNnn
\__tl_replace_next:w
\__tl_replace_wrap:w

```

To implement the actual replacement auxiliary `__tl_replace_auxii:nNNNnn` we will need a $\langle \textit{delimiter} \rangle$ with the following properties:

- all occurrences of the $\langle \textit{pattern} \rangle$ #6 in “ $\langle \textit{token list} \rangle \langle \textit{delimiter} \rangle$ ” belong to the $\langle \textit{token list} \rangle$ and have no overlap with the $\langle \textit{delimiter} \rangle$,
- the first occurrence of the $\langle \textit{delimiter} \rangle$ in “ $\langle \textit{token list} \rangle \langle \textit{delimiter} \rangle$ ” is the trailing $\langle \textit{delimiter} \rangle$.

We first find the building blocks for the $\langle \textit{delimiter} \rangle$, namely two tokens $\langle A \rangle$ and $\langle B \rangle$ such that $\langle A \rangle$ does not appear in #6 and #6 is not $\langle B \rangle$ (this condition is trivial if #6 has more than one token). Then we consider the delimiters “ $\langle A \rangle$ ” and “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ”, for $n \geq 1$, where $\langle A \rangle^n$ denotes n copies of $\langle A \rangle$, and we choose as our $\langle \textit{delimiter} \rangle$ the first one which is not in the $\langle \textit{token list} \rangle$.

Every delimiter in the set obeys the first condition: #6 does not contain $\langle A \rangle$ hence cannot be overlapping with the $\langle \textit{token list} \rangle$ and the $\langle \textit{delimiter} \rangle$, and it cannot be within the $\langle \textit{delimiter} \rangle$ since it would have to be in one of the two $\langle B \rangle$ 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 $\langle \textit{delimiter} \rangle$ we choose does not appear in the $\langle \textit{token list} \rangle$. Additionally, the set of delimiters is such that a $\langle \textit{token list} \rangle$ of n tokens can contain at most $O(n^{1/2})$ of them, hence we find a $\langle \textit{delimiter} \rangle$ 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 “ $\langle A \rangle$ ” in the list of delimiters to try, so that the $\langle \textit{delimiter} \rangle$ will simply be `\q_mark` in the most common situation where neither the $\langle \textit{token list} \rangle$ nor the $\langle \textit{pattern} \rangle$ contains `\q_mark`.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty $\langle \textit{pattern} \rangle$ #6 is an error, and if #1 is absent from both the $\langle \textit{token list} \rangle$ #5 and the $\langle \textit{pattern} \rangle$ #6 then we can use it as the $\langle \textit{delimiter} \rangle$ through `__tl_replace_auxii:nNNNnn {#1}`. Otherwise, we end up calling `__tl_replace:NnnNNNnn` repeatedly with the first two arguments `\q_mark {?}`, `\? {??}`, `\?? {???`, and so on, until #6 does not contain the control sequence #1, which we take as our $\langle A \rangle$. 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 $\langle B \rangle$ to be `\q_nil` or `\q_stop` such that it is not equal to #6.

The `__tl_replace_auxi:NnnNNNnn` auxiliary receives $\{\langle A \rangle\}$ and $\{\langle A \rangle^n \langle B \rangle\}$ as its arguments, initially with $n = 1$. If “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ” is in the $\langle \textit{token list} \rangle$ then increase n and try again. Once it is not anymore in the $\langle \textit{token list} \rangle$ we take it as our $\langle \textit{delimiter} \rangle$ and pass this to the `auxii` auxiliary.

```

5172 \cs_new_protected:Npn \__tl_replace:NnnNNNnn #1#2#3#4#5#6#7
5173   {
5174     \tl_if_empty:nTF {#6}

```

```

5175     {
5176       \_msg_kernel_error:nnx { kernel } { empty-search-pattern }
5177       { \tl_to_str:n {#7} }
5178     }
5179     {
5180       \tl_if_in:onTF { #5 #6 } {#1}
5181       {
5182         \tl_if_in:nnTF {#6} {#1}
5183         { \exp_args:Nc \_tl_replace:NnnNNnn {#2} {#2?} }
5184         {
5185           \quark_if_nil:nTF {#6}
5186           { \_tl_replace_auxi:NnnNNnn #5 {#1} { #1 \q_stop } }
5187           { \_tl_replace_auxi:NnnNNnn #5 {#1} { #1 \q_nil } }
5188         }
5189       }
5190       { \_tl_replace_auxii:nNNNnn {#1} }
5191       #3#4#5 {#6} {#7}
5192     }
5193   }
5194   \cs_new_protected:Npn \_tl_replace_auxi:NnnNNnn #1#2#3
5195   {
5196     \tl_if_in:NnTF #1 { #2 #3 #3 }
5197     { \_tl_replace_auxi:NnnNNnn #1 { #2 #3 } {#2} }
5198     { \_tl_replace_auxii:nNNNnn { #2 #3 #3 } }
5199   }

```

The auxiliary `_tl_replace_auxii:nNNNnn` receives the following arguments: $\langle\langle\textit{delimiter}\rangle\rangle$ $\langle\textit{function}\rangle$ $\langle\textit{assignment}\rangle$ $\langle\textit{tl var}\rangle$ $\langle\textit{pattern}\rangle$ $\langle\textit{replacement}\rangle$. 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 $\langle\textit{assignment}\rangle$ `#3` to the $\langle\textit{tl var}\rangle$ `#4`. The auxiliary `_tl_replace_next:w` is called, followed by the $\langle\textit{token list}\rangle$, some tokens including the $\langle\textit{delimiter}\rangle$ `#1`, followed by the $\langle\textit{pattern}\rangle$ `#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 $\langle\textit{token list}\rangle$ or is the trailing one.

If on the one hand it is found within the $\langle\textit{token list}\rangle$, then `##1` cannot contain the $\langle\textit{delimiter}\rangle$ `#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` $\langle\textit{replacement}\rangle$ 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 $\langle\textit{remaining tokens}\rangle$ in the $\langle\textit{token list}\rangle$ and `##2` is some $\langle\textit{ending code}\rangle$ 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 $\langle\textit{pattern}\rangle$ `#5`, then `##1` is “`{ } { } $\langle\textit{token list}\rangle$ $\langle\textit{delimiter}\rangle$ $\langle\textit{ending code}\rangle$ ”`, hence `_tl_replace_wrap:w` finds “`{ } { } $\langle\textit{token list}\rangle$ ”` as `##1` and the $\langle\textit{ending code}\rangle$ as `##2`. It leaves the $\langle\textit{token list}\rangle$ into the assignment and unbraces the $\langle\textit{ending code}\rangle$

which removes what remains (essentially the $\langle delimiter \rangle$ and $\langle replacement \rangle$).

```

5200 \cs_new_protected:Npn \__tl_replace_auxii:nNNNnn #1#2#3#4#5#6
5201 {
5202   \group_align_safe_begin:
5203   \cs_set:Npn \__tl_replace_wrap:w ##1 #1 ##2
5204     { \exp_not:o { \use_none:nn ##1 } ##2 }
5205   \cs_set:Npx \__tl_replace_next:w ##1 #5
5206     {
5207       \exp_not:N \__tl_replace_wrap:w ##1
5208       \exp_not:n { #1 }
5209       \exp_not:n { \exp_not:n {#6} }
5210       \exp_not:n { #2 { } { } }
5211     }
5212   #3 #4
5213   {
5214     \exp_after:wN \__tl_replace_next:w
5215     \exp_after:wN { \exp_after:wN }
5216     \exp_after:wN { \exp_after:wN }
5217     #4
5218     #1
5219     {
5220       \if_false: { \fi: }
5221       \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
5222     }
5223     #5
5224   }
5225   \group_align_safe_end:
5226 }
5227 \cs_new_eq:NN \__tl_replace_wrap:w ?
5228 \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
5229 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
5230 { \tl_replace_once:Nnn #1 {#2} { } }
5231 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
5232 { \tl_greplace_once:Nnn #1 {#2} { } }
5233 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
5234 \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 91.)

```

\tl_remove_all:Nn Removal is just a special case of replacement.
\tl_remove_all:cn
\tl_gremove_all:Nn
\tl_gremove_all:cn
5235 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
5236 { \tl_replace_all:Nnn #1 {#2} { } }
5237 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
5238 { \tl_greplace_all:Nnn #1 {#2} { } }
5239 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
5240 \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 91.)

10.6 Token list conditionals

`\tl_if_blank_p:n` TeX 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_return:o` is a fast emptiness test, converting its argument to a string (after one expansion) and using the test `\if_meaning:w \q_nil ... \q_nil`.

```

\tl_if_blank_p:n 5241 \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
\tl_if_blank_p:V 5242 { \__tl_if_empty_return:o { \use_none:n #1 ? } }
\tl_if_blank_p:o 5243 \cs_generate_variant:Nn \tl_if_blank_p:n { V }
\tl_if_blank:nTF 5244 \cs_generate_variant:Nn \tl_if_blank:nT { V }
\tl_if_blank:VTF 5245 \cs_generate_variant:Nn \tl_if_blank:nF { V }
\tl_if_blank:oTF 5246 \cs_generate_variant:Nn \tl_if_blank:nTF { V }
\__tl_if_blank_p:NNw 5247 \cs_generate_variant:Nn \tl_if_blank_p:n { o }
                    5248 \cs_generate_variant:Nn \tl_if_blank:nT { o }
                    5249 \cs_generate_variant:Nn \tl_if_blank:nF { o }
                    5250 \cs_generate_variant:Nn \tl_if_blank:nTF { o }

```

(End definition for `\tl_if_blank:nTF` and `__tl_if_blank_p:NNw`. These functions are documented on page 92.)

`\tl_if_empty_p:N` These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```

\tl_if_empty_p:c 5251 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
\tl_if_empty:nTF 5252 {
                    5253   \if_meaning:w #1 \c_empty_tl
                    5254   \prg_return_true:
                    5255   \else:
                    5256   \prg_return_false:
                    5257   \fi:
                    5258 }
\tl_if_empty:NTF 5259 \cs_generate_variant:Nn \tl_if_empty_p:N { c }
\tl_if_empty:NTF 5260 \cs_generate_variant:Nn \tl_if_empty:NT { c }
\tl_if_empty:NTF 5261 \cs_generate_variant:Nn \tl_if_empty:NF { c }
\tl_if_empty:NTF 5262 \cs_generate_variant:Nn \tl_if_empty:NTF { c }

```

(End definition for `\tl_if_empty:NTF`. This function is documented on page 93.)

`\tl_if_empty_p:n` Convert the argument to a string: this will be empty if and only if the argument is. Then `\if_meaning:w \q_nil ... \q_nil` is true if and only if the string ... is empty. It could be tempting to use `\if_meaning:w \q_nil #1 \q_nil` directly. This fails on a token list starting with `\q_nil` of course but more troubling is the case where argument is a complete conditional such as `\if_true: a \else: b \fi:` because then `\if_true:` is used by `\if_meaning:w`, the test turns out false, the `\else:` executes the false branch, the `\fi:` ends it and the `\q_nil` at the end starts executing...

```

5263 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
5264 {
5265   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
5266   \tl_to_str:n {#1} \q_nil
5267   \prg_return_true:
5268   \else:
5269   \prg_return_false:
5270   \fi:
5271 }
5272 \cs_generate_variant:Nn \tl_if_empty_p:n { V }

```

```

5273 \cs_generate_variant:Nn \tl_if_empty:nTF { V }
5274 \cs_generate_variant:Nn \tl_if_empty:nT { V }
5275 \cs_generate_variant:Nn \tl_if_empty:nF { V }

```

(End definition for `\tl_if_empty:nTF`. This function is documented on page 93.)

`\tl_if_empty_p:o` The auxiliary function `__tl_if_empty_return: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 many places. Note that this works because `\etex_detokenize:D` expands tokens that follow until reading a catcode 1 (begin-group) token.

```

5276 \cs_new:Npn \__tl_if_empty_return:o #1
5277 {
5278   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
5279   \etex_detokenize:D \exp_after:wN {#1} \q_nil
5280   \prg_return_true:
5281   \else:
5282     \prg_return_false:
5283   \fi:
5284 }
5285 \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
5286 { \__tl_if_empty_return:o {#1} }

```

(End definition for `\tl_if_empty:oTF` and `__tl_if_empty_return:o`. These functions are documented on page 93.)

`\tl_if_eq_p:NN` Returns `\c_true_bool` if and only if the two token list variables are equal.

```

\tl_if_eq_p:Nc 5287 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
\tl_if_eq_p:cN 5288 {
\tl_if_eq_p:cc 5289   \if_meaning:w #1 #2
\tl_if_eq:NNTF 5290   \prg_return_true:
\tl_if_eq:NcTF 5291   \else:
\tl_if_eq:cNTF 5292   \prg_return_false:
\tl_if_eq:ccTF 5293   \fi:
5294 }
5295 \cs_generate_variant:Nn \tl_if_eq_p:NN { Nc , c , cc }
5296 \cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
5297 \cs_generate_variant:Nn \tl_if_eq:NNT { Nc , c , cc }
5298 \cs_generate_variant:Nn \tl_if_eq:NNF { Nc , c , cc }

```

(End definition for `\tl_if_eq:NNTF`. This function is documented on page 93.)

`\tl_if_eq:nnTF` A simple store and compare routine.

```

\l__tl_internal_a_tl 5299 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
\l__tl_internal_b_tl 5300 {
5301   \group_begin:
5302     \tl_set:Nn \l__tl_internal_a_tl {#1}
5303     \tl_set:Nn \l__tl_internal_b_tl {#2}
5304     \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
5305     \group_end:
5306     \prg_return_true:
5307   \else:
5308     \group_end:

```

```

5309         \prg_return_false:
5310     \fi:
5311 }
5312 \tl_new:N \l__tl_internal_a_tl
5313 \tl_new:N \l__tl_internal_b_tl

```

(End definition for `\tl_if_eq:nnTF`, `\l__tl_internal_a_tl`, and `\l__tl_internal_b_tl`. These functions are documented on page 93.)

`\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`.

```

5314 \cs_new_protected:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
5315 \cs_new_protected:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
5316 \cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
5317 \cs_generate_variant:Nn \tl_if_in:NnT { c }
5318 \cs_generate_variant:Nn \tl_if_in:NnF { c }
5319 \cs_generate_variant:Nn \tl_if_in:NnTF { c }

```

(End definition for `\tl_if_in:NnTF`. This function is documented on page 93.)

`\tl_if_in:nnTF` Once more, the test relies on the emptiness test for robustness. The function `__tl_tmp:w` removes tokens until the first occurrence of `#2`. If this does not appear in `#1`, then the final `#2` is removed, leaving an empty token list. Otherwise some tokens remain, and 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:`.

```

5320 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
5321 {
5322     \if_false: { \fi:
5323         \cs_set:Npn \__tl_tmp:w ##1 #2 { }
5324         \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
5325         { \prg_return_false: } { \prg_return_true: }
5326     \if_false: } \fi:
5327 }
5328 \cs_generate_variant:Nn \tl_if_in:nnT { V , o , no }
5329 \cs_generate_variant:Nn \tl_if_in:nnF { V , o , no }
5330 \cs_generate_variant:Nn \tl_if_in:nnTF { V , o , no }

```

(End definition for `\tl_if_in:nnTF`. This function is documented on page 93.)

`\tl_if_single_p:N` Expand the token list and feed it to `\tl_if_single:n`.

`\tl_if_single:NnTF`

```

5331 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
5332 \cs_new:Npn \tl_if_single:NnT { \exp_args:No \tl_if_single:nT }
5333 \cs_new:Npn \tl_if_single:NnF { \exp_args:No \tl_if_single:nF }
5334 \cs_new:Npn \tl_if_single:NnTF { \exp_args:No \tl_if_single:nTF }

```

(End definition for `\tl_if_single:NnTF`. This function is documented on page 93.)

`\tl_if_single_p:n` 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, `\tl_to_str:n` 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:nw` picks the second token in front of it. If #1 is empty, this token will be the trailing ? and the catcode test yields **false**. If #1 has a single item, the token will be ^ and the catcode test yields **true**. Otherwise, it will be one of the characters resulting from `\tl_to_str:n`, and the catcode test yields **false**. Note that `\if_catcode:w` takes care of the expansions, and that `\tl_to_str:n` (the `\detokenize` primitive) actually expands tokens until finding a begin-group token.

```

5335 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
5336 {
5337   \if_catcode:w ^ \exp_after:wN \__tl_if_single:nw
5338     \tl_to_str:n \exp_after:wN { \use_none:nn #1 ?? } ^ ? \q_stop
5339   \prg_return_true:
5340   \else:
5341     \prg_return_false:
5342   \fi:
5343 }
5344 \cs_new:Npn \__tl_if_single:nw #1#2#3 \q_stop {#2}

```

(End definition for `\tl_if_single:nTF` and `__tl_if_single:nTF`. These functions are documented on page 94.)

`\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 will always be 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.

```

\__tl_case:nnTF
\__tl_case:Nw
\__prg_case_end:nw
\__tl_case_end:nw
5345 \cs_new:Npn \tl_case:Nn #1#2
5346 {
5347   \exp:w
5348   \__tl_case:NnTF #1 {#2} { } { }
5349 }
5350 \cs_new:Npn \tl_case:NnT #1#2#3
5351 {
5352   \exp:w
5353   \__tl_case:NnTF #1 {#2} {#3} { }
5354 }
5355 \cs_new:Npn \tl_case:NnF #1#2#3
5356 {
5357   \exp:w
5358   \__tl_case:NnTF #1 {#2} { } {#3}
5359 }
5360 \cs_new:Npn \tl_case:NnTF #1#2
5361 {
5362   \exp:w
5363   \__tl_case:NnTF #1 {#2}
5364 }
5365 \cs_new:Npn \__tl_case:NnTF #1#2#3#4
5366 { \__tl_case:Nw #1 #2 #1 { } \q_mark {#3} \q_mark {#4} \q_stop }
5367 \cs_new:Npn \__tl_case:Nw #1#2#3

```

```

5368 {
5369   \tl_if_eq:NNTF #1 #2
5370   { \__tl_case_end:nw {#3} }
5371   { \__tl_case:Nw #1 }
5372 }
5373 \cs_generate_variant:Nn \tl_case:Nn { c }
5374 \cs_generate_variant:Nn \tl_case:NnT { c }
5375 \cs_generate_variant:Nn \tl_case:NnF { c }
5376 \cs_generate_variant:Nn \tl_case:NnTF { c }

```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 will be the code to insert, #2 will be the *next* case to check on and #3 will be all of the rest of the cases code. That means that #4 will be the **true** branch code, and #5 will be tidy up the spare `\q_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 will be empty, #2 will be the first `\q_mark` and so #4 will be the **false** code (the **true** code is mopped up by #3).

```

5377 \cs_new:Npn \__prg_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
5378 { \exp_end: #1 #4 }
5379 \cs_new_eq:NN \__tl_case_end:nw \__prg_case_end:nw

```

(End definition for `\tl_case:NnTF` and others. These functions are documented on page 94.)

10.7 Mapping to token lists

`\tl_map_function:nN` Expandable loop macro for token lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker will be read immediately and the loop terminated.

```

\__tl_map_function:Nn
5380 \cs_new:Npn \tl_map_function:Nn #1#2
5381 {
5382   \__tl_map_function:Nn #2 #1
5383   \q_recursion_tail
5384   \__prg_break_point:Nn \tl_map_break: { }
5385 }
5386 \cs_new:Npn \tl_map_function:NN
5387 { \exp_args:No \tl_map_function:nN }
5388 \cs_new:Npn \__tl_map_function:Nn #1#2
5389 {
5390   \__quark_if_recursion_tail_break:nN {#2} \tl_map_break:
5391   #1 {#2} \__tl_map_function:Nn #1
5392 }
5393 \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 94.)

`\tl_map_inline:nn` The inline functions are straight forward by now. We use a little trick with the counter `\g__prg_map_int` to make them nestable. We can also make use of `__tl_map_function:Nn` from before.

```

5394 \cs_new_protected:Npn \tl_map_inline:nn #1#2
5395 {
5396   \int_gincr:N \g__prg_map_int
5397   \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
5398   \exp_args:Nc \__tl_map_function:Nn

```

```

5399     { __prg_map_ \int_use:N \g__prg_map_int :w }
5400     #1 \q_recursion_tail
5401     \__prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
5402   }
5403   \cs_new_protected:Npn \tl_map_inline:Nn
5404     { \exp_args:No \tl_map_inline:nn }
5405   \cs_generate_variant:Nn \tl_map_inline:Nn { c }

```

`\tl_map_variable:nNn` `\tl_map_variable:nNn` $\langle token\ list \rangle$ $\langle temp \rangle$ $\langle action \rangle$ assigns $\langle temp \rangle$ to each element and
`\tl_map_variable:NNn` executes $\langle action \rangle$.

(End definition for `\tl_map_variable:nNn`, `\tl_map_variable:NNn`, and `__tl_map_variable:Nnn`.
These functions are documented on page 95.)

[illegible]

10.8 Using token lists

(End definition for `tl_to_str:n`. This function is documented on page 96.)

```

\tl_to_str:c      5426 \cs_new:Npn \tl_to_str:N #1 { \etex_detokenize:D \exp_after:wN {#1} }
                   5427 \cs_generate_variant:Nn \tl_to_str:N { c }

```

\tl_use:N Token lists which are simply not defined will give a clear T_EX 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.

```

5428 \cs_new:Npn \tl_use:N #1
5429 {
5430   \tl_if_exist:NTF #1 {#1}
5431   {
5432     \__msg_kernel_expandable_error:nnn
5433     { kernel } { bad-variable } {#1}
5434   }
5435 }
5436 \cs_generate_variant:Nn \tl_use:N { c }

```

(End definition for `\tl_use:N`. This function is documented on page 96.)

10.9 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.

```

5437 \cs_new:Npn \tl_count:n #1
5438 {
5439   \int_eval:n
5440   { 0 \tl_map_function:nN {#1} \__tl_count:n }
5441 }
5442 \cs_new:Npn \tl_count:N #1
5443 {
5444   \int_eval:n
5445   { 0 \tl_map_function:NN #1 \__tl_count:n }
5446 }
5447 \cs_new:Npn \__tl_count:n #1 { + \c_one }
5448 \cs_generate_variant:Nn \tl_count:n { V , o }
5449 \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 96.)

\tl_reverse_items:n Reversal of a token list is done by taking one item at a time and putting it after `\q_stop`.

```

5450 \cs_new:Npn \tl_reverse_items:n #1
5451 {
5452   \__tl_reverse_items:nwNwn #1 ?
5453   \q_mark \__tl_reverse_items:nwNwn
5454   \q_mark \__tl_reverse_items:wn
5455   \q_stop { }
5456 }
5457 \cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \q_mark #3 #4 \q_stop #5
5458 {
5459   #3 #2
5460   \q_mark \__tl_reverse_items:nwNwn
5461   \q_mark \__tl_reverse_items:wn
5462   \q_stop { {#1} #5 }
5463 }
5464 \cs_new:Npn \__tl_reverse_items:wn #1 \q_stop #2
5465 { \exp_not:o { \use_none:nn #2 } }

```

(End definition for `\tl_reverse_items:n`, `__tl_reverse_items:nwNwn`, and `__tl_reverse_items:wn`. These functions are documented on page 97.)

`\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 `\q_mark`, and whose second argument is a *<continuation>*, which will receive as a braced argument `\use_none:n \q_mark <trimmed token list>`. In the case at hand, we take `\exp_not:o` as our continuation, so that space trimming will behave correctly within an x-type expansion.

```

5466 \cs_new:Npn \tl_trim_spaces:n #1
5467   { \__tl_trim_spaces:nn { \q_mark #1 } \exp_not:o }
5468 \cs_new_protected:Npn \tl_trim_spaces:N #1
5469   { \tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
5470 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
5471   { \tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
5472 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
5473 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }

```

(End definition for `\tl_trim_spaces:n`, `\tl_trim_spaces:N`, and `\tl_gtrim_spaces:N`. These functions are documented on page 97.)

`__tl_trim_spaces:nn` 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 `\q_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 `␣ \q_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 `\use_none:n` placed there to gobble a lingering `\q_mark`, and feeds this to the *<continuation>*.

```

5474 \cs_set:Npn \__tl_tmp:w #1
5475   {
5476     \cs_new:Npn \__tl_trim_spaces:nn ##1
5477       {
5478         \__tl_trim_spaces_auxi:w
5479         ##1
5480         \q_nil
5481         \q_mark #1 { }
5482         \q_mark \__tl_trim_spaces_auxii:w
5483         \__tl_trim_spaces_auxiii:w
5484         #1 \q_nil
5485         \__tl_trim_spaces_auxiv:w
5486         \q_stop
5487       }
5488     \cs_new:Npn \__tl_trim_spaces_auxi:w ##1 \q_mark #1 ##2 \q_mark ##3
5489       {
5490         ##3
5491         \__tl_trim_spaces_auxi:w
5492         \q_mark
5493         ##2
5494         \q_mark #1 {##1}
5495       }
5496     \cs_new:Npn \__tl_trim_spaces_auxii:w

```



```

5497     \_tl_trim_spaces_auxi:w \q_mark \q_mark ##1
5498     {
5499     \_tl_trim_spaces_auxiii:w
5500     ##1
5501     }
5502     \cs_new:Npn \_tl_trim_spaces_auxiii:w ##1 #1 \q_nil ##2
5503     {
5504     ##2
5505     ##1 \q_nil
5506     \_tl_trim_spaces_auxiii:w
5507     }
5508     \cs_new:Npn \_tl_trim_spaces_auxiv:w ##1 \q_nil ##2 \q_stop ##3
5509     { ##3 { \use_none:n ##1 } }
5510   }
5511   \_tl_tmp:w { ~ }

```

(End definition for `_tl_trim_spaces:nn` and others.)

`\tl_sort:Nn` Implemented in `l3sort`.

`\tl_sort:cn`

`\tl_gsort:Nn` (End definition for `\tl_sort:Nn`, `\tl_gsort:Nn`, and `\tl_sort:nN`. These functions are documented on page 98.)

`\tl_gsort:cn`

`\tl_sort:nN`

10.10 Token by token changes

`\q__tl_act_mark`

`\q__tl_act_stop`

The `\tl_act` functions may be applied to any token list. Hence, we use two private quarks, to allow any token, even quarks, in the token list. Only `\q__tl_act_mark` and `\q__tl_act_stop` may not appear in the token lists manipulated by `_tl_act:NNNnn` functions. The quarks are effectively defined in `l3quark`.

(End definition for `\q__tl_act_mark` and `\q__tl_act_stop`.)

`_tl_act:NNNnn`

`_tl_act_output:n`

`_tl_act_reverse_output:n`

`_tl_act_loop:w`

`_tl_act_normal:NwnNNN`

`_tl_act_group:nwnNNN`

`_tl_act_space:wwnNNN`

`_tl_act_end:w`

To help control the expansion, `_tl_act:NNNnn` should always be preceded by `\exp:w` and ends by producing `\exp_end:` once the result has been obtained. Then loop over tokens, groups, and spaces in #5. The marker `\q__tl_act_mark` is used both to avoid losing outer braces and to detect the end of the token list more easily. The result is stored as an argument for the dummy function `_tl_act_result:n`.

```

5512 \cs_new:Npn \_tl_act:NNNnn #1#2#3#4#5
5513 {
5514   \group_align_safe_begin:
5515   \_tl_act_loop:w #5 \q__tl_act_mark \q__tl_act_stop
5516   {#4} #1 #2 #3
5517   \_tl_act_result:n { }
5518 }

```

In the loop, we check how the token list begins and act accordingly. In the “normal” case, we may have reached `\q__tl_act_mark`, 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 a group or with a space. Some extra work is needed to make `_tl_act_space:wwnNNN` gobble the space.

```

5519 \cs_new:Npn \_tl_act_loop:w #1 \q__tl_act_stop
5520 {
5521   \tl_if_head_is_N_type:nTF {#1}

```

```

5522     { \_tl_act_normal:NwnNNN }
5523     {
5524         \tl_if_head_is_group:nTF {#1}
5525         { \_tl_act_group:nwnNNN }
5526         { \_tl_act_space:wwnNNN }
5527     }
5528     #1 \q__tl_act_stop
5529 }
5530 \cs_new:Npn \_tl_act_normal:NwnNNN #1 #2 \q__tl_act_stop #3#4
5531 {
5532     \if_meaning:w \q__tl_act_mark #1
5533     \exp_after:wN \_tl_act_end:wn
5534     \fi:
5535     #4 {#3} #1
5536     \_tl_act_loop:w #2 \q__tl_act_stop
5537     {#3} #4
5538 }
5539 \cs_new:Npn \_tl_act_end:wn #1 \_tl_act_result:n #2
5540 { \group_align_safe_end: \exp_end: #2 }
5541 \cs_new:Npn \_tl_act_group:nwnNNN #1 #2 \q__tl_act_stop #3#4#5
5542 {
5543     #5 {#3} {#1}
5544     \_tl_act_loop:w #2 \q__tl_act_stop
5545     {#3} #4 #5
5546 }
5547 \exp_last_unbraced:NNo
5548 \cs_new:Npn \_tl_act_space:wwnNNN \c_space_tl #1 \q__tl_act_stop #2#3#4#5
5549 {
5550     #5 {#2}
5551     \_tl_act_loop:w #1 \q__tl_act_stop
5552     {#2} #3 #4 #5
5553 }

```

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.

```

5554 \cs_new:Npn \_tl_act_output:n #1 #2 \_tl_act_result:n #3
5555 { #2 \_tl_act_result:n { #3 #1 } }
5556 \cs_new:Npn \_tl_act_reverse_output:n #1 #2 \_tl_act_result:n #3
5557 { #2 \_tl_act_result:n { #1 #3 } }

```

(End definition for `_tl_act:NNNnn` and others.)

<pre> \tl_reverse:n \tl_reverse:o \tl_reverse:V _tl_reverse_normal:nN _tl_reverse_group_preserve:nn _tl_reverse_space:n </pre>	<p>The goal here is to reverse without losing spaces nor braces. This is done using the general internal function <code>_tl_act:NNNnn</code>. Spaces and “normal” tokens are output on the left of the current output. Grouped tokens are output to the left but without any reversal within the group. All of the internal functions here drop one argument: this is needed by <code>_tl_act:NNNnn</code> when changing case (to record which direction the change is in), but not when reversing the tokens.</p>
---	--

```

5558 \cs_new:Npn \tl_reverse:n #1
5559 {
5560     \etex_unexpanded:D \exp_after:wN
5561     {
5562         \exp:w
5563         \_tl_act:NNNnn

```

```

5564         \_tl_reverse_normal:nN
5565         \_tl_reverse_group_preserve:nn
5566         \_tl_reverse_space:n
5567         { }
5568         {#1}
5569     }
5570 }
5571 \cs_generate_variant:Nn \tl_reverse:n { o , V }
5572 \cs_new:Npn \_tl_reverse_normal:nN #1#2
5573 { \_tl_act_reverse_output:n {#2} }
5574 \cs_new:Npn \_tl_reverse_group_preserve:nn #1#2
5575 { \_tl_act_reverse_output:n { {#2} } }
5576 \cs_new:Npn \_tl_reverse_space:n #1
5577 { \_tl_act_reverse_output:n { ~ } }

```

(End definition for `\tl_reverse:n` and others. These functions are documented on page 97.)

```

\tl_reverse:N This reverses the list, leaving \exp_stop_f: in front, which stops the f-expansion.
\tl_reverse:c 5578 \cs_new_protected:Npn \tl_reverse:N #1
\tl_greverse:N 5579 { \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
\tl_greverse:c 5580 \cs_new_protected:Npn \tl_greverse:N #1
5581 { \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
5582 \cs_generate_variant:Nn \tl_reverse:N { c }
5583 \cs_generate_variant:Nn \tl_greverse:N { c }

```

(End definition for `\tl_reverse:N` and `\tl_greverse:N`. These functions are documented on page 97.)

10.11 The first token from a token list

```

\tl_head:N Finding the head of a token list expandably will always strip braces, which is fine as
\tl_head:n this is consistent with for example mapping to a list. The empty brace groups in \tl_
\tl_head:V head:n ensure that a blank argument gives an empty result. The result is returned
\tl_head:v within the \unexpanded primitive. The approach here is to use \if_false: to allow
\tl_head:f us to use } as the closing delimiter: this is the only safe choice, as any other token
\_tl_head_auxi:nw would not be able to parse it's own code. Using a marker, we can see if what we are
\_tl_head_auxii:n grabbing is exactly the marker, or there is anything else to deal with. Is there is, there
\tl_head:w is a loop. If not, tidy up and leave the item in the output stream. More detail in
\tl_tail:N http://tex.stackexchange.com/a/70168.
\tl_tail:n 5584 \cs_new:Npn \tl_head:n #1
\tl_tail:V 5585 {
\tl_tail:v 5586 \etex_unexpanded:D
\tl_tail:f 5587 \if_false: { \fi: \_tl_head_auxi:nw #1 { } \q_stop }
5588 }
5589 \cs_new:Npn \_tl_head_auxi:nw #1#2 \q_stop
5590 {
5591 \exp_after:wN \_tl_head_auxii:n \exp_after:wN {
5592 \if_false: } \fi: {#1}
5593 }
5594 \cs_new:Npn \_tl_head_auxii:n #1
5595 {
5596 \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
5597 \tl_to_str:n \exp_after:wN { \use_none:n #1 } \q_nil
5598 \exp_after:wN \use_i:nn

```

```

5599 \else:
5600 \exp_after:wN \use_ii:nn
5601 \fi:
5602 {#1}
5603 { \if_false: { \fi: \tl_head_auxi:nw #1 } }
5604 }
5605 \cs_generate_variant:Nn \tl_head:n { V , v , f }
5606 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
5607 \cs_new:Npn \tl_head:N { \exp_args:No \tl_head:n }

```

To corrected 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

```

will give the wrong result for `\tl_tail:n { a { bc } }` (the braces will 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.

```

5608 \cs_new:Npn \tl_tail:n #1
5609 {
5610 \etex_unexpanded:D
5611 \tl_if_blank:nTF {#1}
5612 { { } }
5613 { \exp_after:wN { \use_none:n #1 } }
5614 }
5615 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
5616 \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 98.)

```

\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

```

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 will result in `\tl_head:w` leaving two tokens: ? which is taken in the `\if_charcode:w` test, and `\use_none:nn`, which ensures that `\prg_return_false:` is returned regardless of whether the charcode test was **true** or **false**.

```

5617 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
5618 {

```

```

5619 \if_charcode:w
5620 \exp_not:N #2
5621 \tl_if_head_is_N_type:nTF { #1 ? }
5622 {
5623 \exp_after:wN \exp_not:N
5624 \tl_head:w #1 { ? \use_none:nn } \q_stop
5625 }
5626 { \str_head:n {#1} }
5627 \prg_return_true:
5628 \else:
5629 \prg_return_false:
5630 \fi:
5631 }
5632 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN { f }
5633 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF { f }
5634 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT { f }
5635 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNF { 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 `\prg_return_true:` and `\else:` with `\use_none:nn` in case the catcode test with the (arbitrarily chosen) `?` is true.

```

5636 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
5637 {
5638 \if_catcode:w
5639 \exp_not:N #2
5640 \tl_if_head_is_N_type:nTF { #1 ? }
5641 {
5642 \exp_after:wN \exp_not:N
5643 \tl_head:w #1 { ? \use_none:nn } \q_stop
5644 }
5645 {
5646 \tl_if_head_is_group:nTF {#1}
5647 { \c_group_begin_token }
5648 { \c_space_token }
5649 }
5650 \prg_return_true:
5651 \else:
5652 \prg_return_false:
5653 \fi:
5654 }

```

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 the usual `\prg_return_true:` and `\else:`. 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.

```

5655 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
5656 {
5657 \tl_if_head_is_N_type:nTF { #1 ? }

```

```

5658     { \_tl\_if\_head\_eq\_meaning\_normal:nN }
5659     { \_tl\_if\_head\_eq\_meaning\_special:nN }
5660     {#1} #2
5661   }
5662 \cs_new:Npn \_tl\_if\_head\_eq\_meaning\_normal:nN #1 #2
5663 {
5664   \exp\_after:wN \if\_meaning:w
5665   \tl\_head:w #1 { ?? \use\_none:nnn } \q\_stop #2
5666   \prg\_return\_true:
5667 \else:
5668   \prg\_return\_false:
5669 \fi:
5670 }
5671 \cs_new:Npn \_tl\_if\_head\_eq\_meaning\_special:nN #1 #2
5672 {
5673   \if\_charcode:w \str\_head:n {#1} \exp\_not:N #2
5674   \exp\_after:wN \use:n
5675 \else:
5676   \prg\_return\_false:
5677   \exp\_after:wN \use\_none:n
5678 \fi:
5679 {
5680   \if\_catcode:w \exp\_not:N #2
5681       \tl\_if\_head\_is\_group:nTF {#1}
5682       { \c\_group\_begin\_token }
5683       { \c\_space\_token }
5684   \prg\_return\_true:
5685 \else:
5686   \prg\_return\_false:
5687 \fi:
5688 }
5689 }

```

(End definition for `\tl_if_head_eq_meaning:nNTF` and others. These functions are documented on page 99.)

`\tl_if_head_is_N_type_p: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:w` produces `~` (and otherwise nothing). In the third case (begin-group token), the lines involving `\exp_after:wN` 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 `*` to the beginning: if `#1` contains primitive conditionals, all of its occurrences must be dealt with before the `\if_catcode:w` tries to skip the `true` branch of the conditional.

```

5690 \prg\_new\_conditional:Npnn \tl\_if\_head\_is\_N\_type:n #1 { p , T , F , TF }
5691 {
5692   \if\_catcode:w
5693   \if\_false: { \fi: \_tl\_if\_head\_is\_N\_type:w ? #1 ~ }
5694   \exp\_after:wN \use\_none:n
5695   \exp\_after:wN { \exp\_after:wN { \token\_to\_str:N #1 ? } }
5696   * *
5697   \prg\_return\_true:
5698 \else:
5699   \prg\_return\_false:

```

```

5700     \fi:
5701   }
5702   \cs_new:Npn \__tl_if_head_is_N_type:w #1 ~
5703   {
5704     \tl_if_empty:oTF { \use_none:n #1 } { ^ } { }
5705     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
5706   }

```

(End definition for `\tl_if_head_is_N_type:nTF` and `__tl_if_head_is_N_type:w`. These functions are documented on page 100.)

`\tl_if_head_is_group:p:n` Pass the first token of #1 through `\token_to_str:N`, then check for the brace balance.
`\tl_if_head_is_group:nTF` The extra ? caters for an empty argument.⁸

```

5707   \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
5708   {
5709     \if_catcode:w
5710       \exp_after:wN \use_none:n
5711       \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
5712     * *
5713     \prg_return_false:
5714   \else:
5715     \prg_return_true:
5716   \fi:
5717 }

```

(End definition for `\tl_if_head_is_group:nTF`. This function is documented on page 100.)

`\tl_if_head_is_space:p:n` The auxiliary’s argument is all that is before the first explicit space in `?#1?~`. If that
`\tl_if_head_is_space:nTF` is a single ? the test yields true. Otherwise, that is more than one token, and the
`__tl_if_head_is_space:w` 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 `\exp:w` and `\exp_end:` ensure that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

```

5718   \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
5719   {
5720     \exp:w \if_false: { \fi:
5721       \__tl_if_head_is_space:w ? #1 ? ~ }
5722   }
5723   \cs_new:Npn \__tl_if_head_is_space:w #1 ~
5724   {
5725     \tl_if_empty:oTF { \use_none:n #1 }
5726     { \exp_after:wN \exp_end: \exp_after:wN \prg_return_true: }
5727     { \exp_after:wN \exp_end: \exp_after:wN \prg_return_false: }
5728     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
5729   }

```

(End definition for `\tl_if_head_is_space:nTF` and `__tl_if_head_is_space:w`. These functions are documented on page 100.)

⁸Bruno: this could be made faster, but we don’t: if we hope to ever have an e-type argument, we need all brace “tricks” to happen in one step of expansion, keeping the token list brace balanced at all times.

10.12 Using a single item

\tl_item:nn 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 **\quark_if_recursion_tail_stop:n** terminates the loop, and returns nothing at all.

\tl_item:Nn

\tl_item:cn

__tl_item_aux:nn

__tl_item:nn

```

5730 \cs_new:Npn \tl_item:nn #1#2
5731 {
5732   \exp_args:Nf \__tl_item:nn
5733   { \exp_args:Nf \__tl_item_aux:nn { \int_eval:n {#2} } {#1} }
5734   #1
5735   \quark_recursion_tail
5736   \__prg_break_point:
5737 }
5738 \cs_new:Npn \__tl_item_aux:nn #1#2
5739 {
5740   \int_compare:nNnTF {#1} < \c_zero
5741   { \int_eval:n { \tl_count:n {#2} + \c_one + #1 } }
5742   {#1}
5743 }
5744 \cs_new:Npn \__tl_item:nn #1#2
5745 {
5746   \__quark_if_recursion_tail_break:nN {#2} \__prg_break:
5747   \int_compare:nNnTF {#1} = \c_one
5748   { \__prg_break:n { \exp_not:n {#2} } }
5749   { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
5750 }
5751 \cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
5752 \cs_generate_variant:Nn \tl_item:Nn { c }

```

(End definition for **\tl_item:nn** and others. These functions are documented on page 100.)

10.13 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

```

5753 \cs_new_protected:Npn \tl_show:N #1
5754 {
5755   \__msg_show_variable:NNNnn #1 \tl_if_exist:NTF ? { }
5756   { > ~ \token_to_str:N #1 = \tl_to_str:N #1 }
5757 }
5758 \cs_generate_variant:Nn \tl_show:N { c }

```

(End definition for **\tl_show:N**. This function is documented on page 100.)

\tl_show:n The **__msg_show_wrap:n** internal function performs line-wrapping and shows the result using the **\etex_showtokens:D** primitive. Since **\tl_to_str:n** is expanded within the line-wrapping code, the escape character is always a backslash.

```

5759 \cs_new_protected:Npn \tl_show:n #1
5760 { \__msg_show_wrap:n { > ~ \tl_to_str:n {#1} } }

```

(End definition for **\tl_show:n**. This function is documented on page 101.)

10.14 Scratch token lists

\g_tmpa_tl **\g_tmppb_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.

```
5761 \tl_new:N \g_tmpa_tl
5762 \tl_new:N \g_tmppb_tl
```

(End definition for \g_tmpa_tl and \g_tmppb_tl. These variables are documented on page 101.)

\l_tmpa_tl **\l_tmppb_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.

```
5763 \tl_new:N \l_tmpa_tl
5764 \tl_new:N \l_tmppb_tl
```

(End definition for \l_tmpa_tl and \l_tmppb_tl. These variables are documented on page 101.)

10.15 Deprecated functions

```
\tl_to_lowercase:n For removal after 2017-12-31.
\tl_to_uppercase:n
5765 \cs_new_protected:Npn \tl_to_lowercase:n #1
5766 { \tex_lowercase:D {#1} }
5767 \cs_new_protected:Npn \tl_to_uppercase:n #1
5768 { \tex_uppercase:D {#1} }
```

(End definition for \tl_to_lowercase:n and \tl_to_uppercase:n.)

```
5769 </initex | package>
```

11 l3str implementation

```
5770 <*initex | package>
5771 <@@=str>
```

11.1 Creating and setting string variables

\str_new:N **\str_new:c** **\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**

A string is simply a token list. The full mapping system isn't set up yet so do things by hand.

```
5772 \group_begin:
5773   \cs_set_protected:Npn \__str_tmp:n #1
5774   {
5775     \tl_if_blank:nF {#1}
5776     {
5777       \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }
5778       \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }
5779       \__str_tmp:n
5780     }
5781   }
5782   \__str_tmp:n
5783   { new }
5784   { use }
5785   { clear }
5786   { gclear }
5787   { clear_new }
```

```

5788     { gclear_new }
5789     { }
5790 \group_end:
5791 \cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN
5792 \cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN
5793 \cs_generate_variant:Nn \str_set_eq:NN { c , Nc , cc }
5794 \cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }

```

```

Simply convert the token list inputs to  $\langle strings \rangle$ .
\str_set:Nn      5795 \group_begin:
\str_set:Nx      5796 \cs_set_protected:Npn \__str_tmp:n #1
\str_set:cn      5797 {
\str_set:cx      5798   \tl_if_blank:nF {#1}
\str_gset:Nn      5799   {
\str_gset:Nx      5800     \cs_new_protected:cpx { str_ #1 :Nn } ##1##2
\str_gset:cn      5801     { \exp_not:c { tl_ #1 :Nx } ##1 { \exp_not:N \tl_to_str:n {##2} } }
\str_gset:cx      5802     \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :Nn } { Nx , cn , cx }
\str_const:Nn      5803     \__str_tmp:n
\str_const:Nx      5804   }
\str_const:cn      5805 }
\str_const:cx      5806 \__str_tmp:n
\str_put_left:Nn  5807 { set }
\str_put_left:Nx  5808 { gset }
\str_put_left:cn  5809 { const }
\str_put_left:cx  5810 { put_left }
\str_gput_left:Nn 5811 { gput_left }
\str_gput_left:Nx 5812 { put_right }
\str_gput_left:cn 5813 { gput_right }
\str_gput_left:cx 5814 { }
\str_gput_left:Nx 5815 \group_end:

```

(End definition for \str set:Nn and others. These functions are documented on page 103.)

11.2 String comparisons

More copy-paste!

<code>\str_if_empty:NTF</code>	5816	<code>\prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N { p , T , F , TF }</code>
<code>\str_if_empty:N</code>	5817	<code>\prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c { p , T , F , TF }</code>
<code>\str_if_empty:cTF</code>	5818	<code>\prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N { p , T , F , TF }</code>
<code>\str_if_exist_p:N</code>	5819	<code>\prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c { p , T , F , TF }</code>
<code>\str_if_exist_p:c</code>		
<code>\str_if_exist:NTF</code>		<i>(End definition for \str_if_empty:NTF and \str_if_exist:NTF. These functions are documented on page 104.)</i>
<code>\str_if_exist:cTF</code>		
<code>__str_if_eq_x:nn</code>		String comparisons rely on the primitive <code>\(pdf)strcmp</code> if available: LuaTeX does not
<code>__str_escape_x:n</code>		have it, so emulation is required. As the net result is that we do not <i>always</i> use the

String comparisons rely on the primitive `\(pdf)strcmp` if available: LuaTeX does not have it, so emulation is required. As the net result is that we do not *always* use the primitive, the correct approach is to wrap up in a function with defined behaviour. That's done by providing a wrapper and then redefining in the LuaTeX case. Note that the necessary Lua code is covered in `l3bootstrap`: long-term this may need to go into a separate Lua file, but at present it's somewhere that spaces are not skipped for ease-of-input. The need to detokenize and force expansion of input arises from the case where

a # token is used in the input, e.g. `__str_if_eq_x:nn {#} { \tl_to_str:n {#} }`, which otherwise will fail as `\luatex_luaescapestring:D` does not double such tokens.

```

5820 \cs_new:Npn \__str_if_eq_x:nn #1#2 { \pdfTeX_strcmp:D {#1} {#2} }
5821 \cs_if_exist:NT \luatex_luaTeXversion:D
5822 {
5823   \cs_set:Npn \__str_if_eq_x:nn #1#2
5824   {
5825     \luatex_directlua:D
5826     {
5827       l3kernel_strcmp
5828       (
5829         " \__str_escape_x:n {#1} " ,
5830         " \__str_escape_x:n {#2} "
5831       )
5832     }
5833   }
5834   \cs_new:Npn \__str_escape_x:n #1
5835   {
5836     \luatex_luaescapestring:D
5837     {
5838       \etex_detokenize:D \exp_after:wN { \luatex_expanded:D {#1} }
5839     }
5840   }
5841 }

```

(End definition for `__str_if_eq_x:nn` and `__str_escape_x:n`.)

`__str_if_eq_x_return:nn` It turns out that we often need to compare a token list with the result of applying some function to it, and return with `\prg_return_true/false:`. This test is similar to `\str_if_eq:nnTF` (see `l3str`), but is hard-coded for speed.

```

5842 \cs_new:Npn \__str_if_eq_x_return:nn #1 #2
5843 {
5844   \if_int_compare:w \__str_if_eq_x:nn {#1} {#2} = \c_zero
5845   \prg_return_true:
5846   \else:
5847   \prg_return_false:
5848   \fi:
5849 }

```

(End definition for `__str_if_eq_x_return: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.

`\str_if_eq_p:Vn`

`\str_if_eq_p:on`

`\str_if_eq_p:nV`

`\str_if_eq_p:no`

`\str_if_eq_p:VV`

`\str_if_eq:nnTF`

`\str_if_eq:VnTF`

`\str_if_eq:onTF`

`\str_if_eq:nVTF`

`\str_if_eq:noTF`

`\str_if_eq:VVTF`

`\str_if_eq_x:nn`

`\str_if_eq_x:nnTF`

```

5850 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
5851 {
5852   \if_int_compare:w
5853     \__str_if_eq_x:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
5854     = \c_zero
5855   \prg_return_true: \else: \prg_return_false: \fi:
5856 }
5857 \cs_generate_variant:Nn \str_if_eq_p:nn { V , o }
5858 \cs_generate_variant:Nn \str_if_eq_p:nn { nV , no , VV }
5859 \cs_generate_variant:Nn \str_if_eq:nnT { V , o }

```

```

5860 \cs_generate_variant:Nn \str_if_eq:nnT { nV , no , VV }
5861 \cs_generate_variant:Nn \str_if_eq:nnF { V , o }
5862 \cs_generate_variant:Nn \str_if_eq:nnF { nV , no , VV }
5863 \cs_generate_variant:Nn \str_if_eq:nnTF { V , o }
5864 \cs_generate_variant:Nn \str_if_eq:nnTF { nV , no , VV }
5865 \prg_new_conditional:Npnn \str_if_eq_x:nn #1#2 { p , T , F , TF }
5866 {
5867     \if_int_compare:w \__str_if_eq_x:nn {#1} {#2} = \c_zero
5868     \prg_return_true: \else: \prg_return_false: \fi:
5869 }

```

(End definition for `\str_if_eq:nnTF` and `\str_if_eq_x:nnTF`. These functions are documented on page 104.)

`\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.

`\str_if_eq_p:Nc`

`\str_if_eq_p:cN` 5870 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }

`\str_if_eq_p:cc` 5871 {

`\str_if_eq:NNTF` 5872 \if_int_compare:w __str_if_eq_x:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }

`\str_if_eq:NcTF` 5873 = \c_zero \prg_return_true: \else: \prg_return_false: \fi:

`\str_if_eq:cNTF` 5874 }

`\str_if_eq:ccTF` 5875 \cs_generate_variant:Nn \str_if_eq:NNT { c , Nc , cc }

5876 \cs_generate_variant:Nn \str_if_eq:NNF { c , Nc , cc }

5877 \cs_generate_variant:Nn \str_if_eq:NNTF { c , Nc , cc }

5878 \cs_generate_variant:Nn \str_if_eq_p:NN { c , Nc , cc }

(End definition for `\str_if_eq:NNTF`. This function is documented on page 104.)

`\str_case:nn` Much the same as `\tl_case:nn(TF)` here: just a change in the internal comparison.

`\str_case:on` 5879 \cs_new:Npn \str_case:nn #1#2

`\str_case:nV` 5880 {

`\str_case:nv` 5881 \exp:w

`\str_case:nnTF` 5882 __str_case:nnTF {#1} {#2} { } { }

`\str_case:onTF` 5883 }

`\str_case:nVTF` 5884 \cs_new:Npn \str_case:nnT #1#2#3

`\str_case:nvTF` 5885 {

`\str_case_x:nn` 5886 \exp:w

`\str_case_x:nnTF` 5887 __str_case:nnTF {#1} {#2} {#3} { }

5888 }

`__str_case:nnTF` 5889 \cs_new:Npn \str_case:nnF #1#2

`__str_case_x:nnTF` 5890 {

`__str_case:nw` 5891 \exp:w

`__str_case_x:nw` 5892 __str_case:nnTF {#1} {#2} { }

`__str_case_end:nw` 5893 }

5894 \cs_new:Npn \str_case:nnTF #1#2

5895 {

5896 \exp:w

5897 __str_case:nnTF {#1} {#2}

5898 }

5899 \cs_new:Npn __str_case:nnTF #1#2#3#4

5900 { __str_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }

5901 \cs_generate_variant:Nn \str_case:nn { o , nV , nv }

5902 \cs_generate_variant:Nn \str_case:nnT { o , nV , nv }

5903 \cs_generate_variant:Nn \str_case:nnF { o , nV , nv }

5904 \cs_generate_variant:Nn \str_case:nnTF { o , nV , nv }

```

5905 \cs_new:Npn \__str_case:nw #1#2#3
5906 {
5907   \str_if_eq:nnTF {#1} {#2}
5908     { \__str_case_end:nw {#3} }
5909     { \__str_case:nw {#1} }
5910 }
5911 \cs_new:Npn \str_case_x:nn #1#2
5912 {
5913   \exp:w
5914   \__str_case_x:nnTF {#1} {#2} { } { }
5915 }
5916 \cs_new:Npn \str_case_x:nnT #1#2#3
5917 {
5918   \exp:w
5919   \__str_case_x:nnTF {#1} {#2} {#3} { }
5920 }
5921 \cs_new:Npn \str_case_x:nnF #1#2
5922 {
5923   \exp:w
5924   \__str_case_x:nnTF {#1} {#2} { } { }
5925 }
5926 \cs_new:Npn \str_case_x:nnTF #1#2
5927 {
5928   \exp:w
5929   \__str_case_x:nnTF {#1} {#2}
5930 }
5931 \cs_new:Npn \__str_case_x:nnTF #1#2#3#4
5932 { \__str_case_x:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
5933 \cs_new:Npn \__str_case_x:nw #1#2#3
5934 {
5935   \str_if_eq_x:nnTF {#1} {#2}
5936     { \__str_case_end:nw {#3} }
5937     { \__str_case_x:nw {#1} }
5938 }
5939 \cs_new_eq:NN \__str_case_end:nw \__prg_case_end:nw

```

(End definition for `\str_case:nnTF` and others. These functions are documented on page 104.)

11.3 Accessing specific characters in a string

`__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 `\q_mark` and `\q_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 `\q_mark` and the first A (well, there is also the need to remove a space).

```

5940 \cs_new:Npn \__str_to_other:n #1
5941 {
5942   \exp_after:wN \__str_to_other_loop:w
5943   \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_mark \q_stop
5944 }
5945 \group_begin:
5946 \tex_lccode:D ‘\* = ‘\ %
5947 \tex_lccode:D ‘\A = ‘\A

```

```

5948 \tex_lowercase:D
5949 {
5950   \group_end:
5951   \cs_new:Npn \__str_to_other_loop:w
5952     #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \q_stop
5953   {
5954     \if_meaning:w A #8
5955     \__str_to_other_end:w
5956     \fi:
5957     \__str_to_other_loop:w
5958     #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \q_stop
5959   }
5960   \cs_new:Npn \__str_to_other_end:w \fi: #1 \q_mark #2 * A #3 \q_stop
5961   { \fi: #2 }
5962 }

```

(End definition for `__str_to_other:n`, `__str_to_other_loop:w`, and `__str_to_other_end:w`.)

`\str_item:Nn` The `\str_item:nn` hands its argument with spaces escaped to `__str_item:nn`, and `\str_item:cn` 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 $\langle index \rangle$ argument #2 and count characters in the string, passing those two numbers to `__str_item:w` for further analysis. If the $\langle index \rangle$ is negative, shift it by the $\langle count \rangle$ to know the how many character to discard, and if that is still negative give an empty result. If the $\langle index \rangle$ is larger than the $\langle count \rangle$, give an empty result, and otherwise discard $\langle index \rangle - 1$ 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 $\langle index \rangle$ is zero.

```

5963 \cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }
5964 \cs_generate_variant:Nn \str_item:Nn { c }
5965 \cs_new:Npn \str_item:nn #1#2
5966 {
5967   \exp_args:Nf \tl_to_str:n
5968   {
5969     \exp_args:Nf \__str_item:nn
5970     { \__str_to_other:n {#1} } {#2}
5971   }
5972 }
5973 \cs_new:Npn \str_item_ignore_spaces:nn #1
5974 { \exp_args:No \__str_item:nn { \tl_to_str:n {#1} } }
5975 \cs_new:Npn \__str_item:nn #1#2
5976 {
5977   \exp_after:wN \__str_item:w
5978   \__int_value:w \__int_eval:w #2 \exp_after:wN ;
5979   \__int_value:w \__str_count:n {#1} ;
5980   #1 \q_stop
5981 }
5982 \cs_new:Npn \__str_item:w #1; #2;
5983 {
5984   \int_compare:nNnTF {#1} < \c_zero
5985   {
5986     \int_compare:nNnTF {#1} < {-#2}

```

```

5987         { \use_none_delimit_by_q_stop:w }
5988         {
5989             \exp_after:wN \use_i_delimit_by_q_stop:nw
5990             \exp:w \exp_after:wN \__str_skip_exp_end:w
5991             \__int_value:w \__int_eval:w #1 + #2 ;
5992         }
5993     }
5994     {
5995         \int_compare:nNnTF {#1} > {#2}
5996         { \use_none_delimit_by_q_stop:w }
5997         {
5998             \exp_after:wN \use_i_delimit_by_q_stop:nw
5999             \exp:w \__str_skip_exp_end:w #1 ; { }
6000         }
6001     }
6002 }

```

(End definition for `\str_item:Nn` and others. These functions are documented on page 106.)

```

\__str_skip_exp_end:w
\__str_skip_loop:wNNNNNNNN
\__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).

```

6003 \cs_new:Npn \__str_skip_exp_end:w #1;
6004 {
6005     \if_int_compare:w #1 > \c_eight
6006     \exp_after:wN \__str_skip_loop:wNNNNNNNN
6007     \else:
6008     \exp_after:wN \__str_skip_end:w
6009     \__int_value:w \__int_eval:w
6010     \fi:
6011     #1 ;
6012 }
6013 \cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
6014 { \exp_after:wN \__str_skip_exp_end:w \__int_value:w \__int_eval:w #1 - \c_eight ; }
6015 \cs_new:Npn \__str_skip_end:w #1 ;
6016 {
6017     \exp_after:wN \__str_skip_end:NNNNNNNN
6018     \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or:
6019 }
6020 \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.

```

6021 \cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
6022 \cs_generate_variant:Nn \str_range:Nnn { c }
6023 \cs_new:Npn \str_range:nnn #1#2#3
6024 {
6025     \exp_args:Nf \tl_to_str:n
6026     {
6027         \exp_args:Nf \__str_range:nnn
6028         { \__str_to_other:n {#1} } {#2} {#3}
6029     }
6030 }
6031 \cs_new:Npn \str_range_ignore_spaces:nnn #1
6032 { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
6033 \cs_new:Npn \__str_range:nnn #1#2#3
6034 {
6035     \exp_after:wN \__str_range:w
6036     \__int_value:w \__str_count:n {#1} \exp_after:wN ;
6037     \__int_value:w \__int_eval:w #2 - \c_one \exp_after:wN ;
6038     \__int_value:w \__int_eval:w #3 ;
6039     #1 \q_stop
6040 }
6041 \cs_new:Npn \__str_range:w #1; #2; #3;
6042 {
6043     \exp_args:Nf \__str_range:nnw
6044     { \__str_range_normalize:nn {#2} {#1} }
6045     { \__str_range_normalize:nn {#3} {#1} }
6046 }
6047 \cs_new:Npn \__str_range:nnw #1#2
6048 {
6049     \exp_after:wN \__str_collect_delimit_by_q_stop:w
6050     \__int_value:w \__int_eval:w #2 - #1 \exp_after:wN ;
6051     \exp:w \__str_skip_exp_end:w #1 ;
6052 }

```

(End definition for `\str_range:Nnn` and others. These functions are documented on page 107.)

`__str_range_normalize:nn`

This function converts an $\langle index \rangle$ argument into an explicit position in the string (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the $\langle index \rangle$ #1 and the string count #2. If #1 is negative, replace it by #1 + #2 + 1, then limit to the range [0, #2].

```

6053 \cs_new:Npn \__str_range_normalize:nn #1#2
6054 {
6055     \int_eval:n
6056     {
6057         \if_int_compare:w #1 < \c_zero
6058         \if_int_compare:w #1 < -#2 \exp_stop_f:
6059             \c_zero
6060         \else:
6061             #1 + #2 + \c_one
6062         \fi:
6063     \else:
6064         \if_int_compare:w #1 < #2 \exp_stop_f:
6065             #1

```



```

6066         \else:
6067             #2
6068         \fi:
6069     \fi:
6070 }
6071 }

```

(End definition for `_str_range_normalize:nn`.)

```

\_str_collect_delimit_by_q_stop:w
\_str_collect_loop:wn
  \_str_collect_loop:wnNNNNNNN
  \_str_collect_end:wn
\_str_collect_end:nnnnnnnnw

```

Collects `max(#1,0)` characters, and removes everything else until `\q_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:nnnnnnnnw` are some `\or:`, followed by an `\fi:`, followed by `#1` characters from the input stream. Simply leaving this in the input stream will close the conditional properly and the `\or:` disappear.

```

6072 \cs_new:Npn \_str_collect_delimit_by_q_stop:w #1;
6073 { \_str_collect_loop:wn #1 ; { } }
6074 \cs_new:Npn \_str_collect_loop:wn #1 ;
6075 {
6076   \if_int_compare:w #1 > \c_seven
6077     \exp_after:wN \_str_collect_loop:wnNNNNNNN
6078   \else:
6079     \exp_after:wN \_str_collect_end:wn
6080   \fi:
6081   #1 ;
6082 }
6083 \cs_new:Npn \_str_collect_loop:wnNNNNNNN #1; #2 #3#4#5#6#7#8#9
6084 {
6085   \exp_after:wN \_str_collect_loop:wn
6086   \_int_value:w \_int_eval:w #1 - \c_seven ;
6087   { #2 #3#4#5#6#7#8#9 }
6088 }
6089 \cs_new:Npn \_str_collect_end:wn #1 ;
6090 {
6091   \exp_after:wN \_str_collect_end:nnnnnnnnw
6092   \if_case:w \if_int_compare:w #1 > \c_zero #1 \else: 0 \fi: \exp_stop_f:
6093   \or: \or: \or: \or: \or: \or: \or: \fi:
6094 }
6095 \cs_new:Npn \_str_collect_end:nnnnnnnnw #1#2#3#4#5#6#7#8 #9 \q_stop
6096 { #1#2#3#4#5#6#7#8 }

```

(End definition for `_str_collect_delimit_by_q_stop:w` and others.)

11.4 Counting characters

```

\_str_count_spaces:N
\_str_count_spaces:c
\_str_count_spaces:n
\_str_count_spaces_loop:w

```

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⟨number⟩`, and that `⟨number⟩` is added to the sum of 9 that precedes, to adjust the result.

```

6097 \cs_new:Npn \str_count_spaces:N
6098 { \exp_args:No \str_count_spaces:n }
6099 \cs_generate_variant:Nn \str_count_spaces:N { c }
6100 \cs_new:Npn \str_count_spaces:n #1

```

```

6101 {
6102   \int_eval:n
6103   {
6104     \exp_after:wN \__str_count_spaces_loop:w
6105     \tl_to_str:n {#1} ~
6106     X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
6107     \q_stop
6108   }
6109 }
6110 \cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
6111 {
6112   \if_meaning:w X #9
6113   \use_i_delimit_by_q_stop:nw
6114   \fi:
6115   \c_nine + \__str_count_spaces_loop:w
6116 }

```

(End definition for `\str_count_spaces:N`, `\str_count_spaces:n`, and `__str_count_spaces_loop:w`. These functions are documented on page 105.)

`\str_count:N` To count characters in a string we could first escape all spaces using `__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. `\str_count:n` 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.

```

6117 \cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
6118 \cs_generate_variant:Nn \str_count:N { c }
6119 \cs_new:Npn \str_count:n #1
6120 {
6121   \__str_count_aux:n
6122   {
6123     \str_count_spaces:n {#1}
6124     + \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
6125   }
6126 }
6127 \cs_new:Npn \__str_count:n #1
6128 {
6129   \__str_count_aux:n
6130   { \__str_count_loop:NNNNNNNNN #1 }
6131 }
6132 \cs_new:Npn \str_count_ignore_spaces:n #1
6133 {
6134   \__str_count_aux:n
6135   { \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1} }
6136 }
6137 \cs_new:Npn \__str_count_aux:n #1
6138 {
6139   \int_eval:n
6140   {

```

```

6141      #1
6142      { X \c_eight } { X \c_seven } { X \c_six }
6143      { X \c_five } { X \c_four } { X \c_three }
6144      { X \c_two } { X \c_one } { X \c_zero }
6145      \q_stop
6146    }
6147  }
6148 \cs_new:Npn \__str_count_loop:NNNNNNNN #1#2#3#4#5#6#7#8#9
6149 {
6150   \if_meaning:w X #9
6151   \exp_after:wN \use_none_delimit_by_q_stop:w
6152   \fi:
6153   \c_nine + \__str_count_loop:NNNNNNNN
6154 }

```

(End definition for `\str_count:N` and others. These functions are documented on page 105.)

11.5 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`.
`\str_head:c` 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, `\use_i_delimit_by_q_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 `\use_i_delimit_by_q_stop:nw`.

```

6155 \cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
6156 \cs_generate_variant:Nn \str_head:N { c }
6157 \cs_new:Npn \str_head:n #1
6158 {
6159   \exp_after:wN \__str_head:w
6160   \tl_to_str:n {#1}
6161   { { } } ~ \q_stop
6162 }
6163 \cs_new:Npn \__str_head:w #1 ~ %
6164 { \use_i_delimit_by_q_stop:nw #1 { ~ } }
6165 \cs_new:Npn \str_head_ignore_spaces:n #1
6166 {
6167   \exp_after:wN \use_i_delimit_by_q_stop:nw
6168   \tl_to_str:n {#1} { } \q_stop
6169 }

```

(End definition for `\str_head:N` and others. These functions are documented on page 106.)

`\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 `\q_mark`. One can check that an empty (or blank) string yields an empty tail.

```

6170 \cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
6171 \cs_generate_variant:Nn \str_tail:N { c }
6172 \cs_new:Npn \str_tail:n #1
6173 {
6174   \exp_after:wN __str_tail_auxi:w
6175   \reverse_if:N \if_charcode:w
6176     \scan_stop: \tl_to_str:n {#1} X X \q_stop
6177 }
6178 \cs_new:Npn __str_tail_auxi:w #1 X #2 \q_stop { \fi: #1 }
6179 \cs_new:Npn \str_tail_ignore_spaces:n #1
6180 {
6181   \exp_after:wN __str_tail_auxii:w
6182   \tl_to_str:n {#1} \q_mark \q_mark \q_stop
6183 }
6184 \cs_new:Npn __str_tail_auxii:w #1 #2 \q_mark #3 \q_stop { #2 }

```

(End definition for `\str_tail:N` and others. These functions are documented on page 106.)

11.6 String manipulation

<pre> \str_fold_case:n \str_fold_case:V \str_lower_case:n \str_lower_case:f \str_upper_case:n \str_upper_case:f __str_change_case:nn __str_change_case_aux:nn __str_change_case_result:n __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 __str_lookup_lower:N __str_lookup_upper:N __str_lookup_fold:N </pre>	<p>Case changing for programmatic reasons is done by first detokenizing input then doing a simple loop that only has to worry about spaces and everything else. The output is detokenized to allow data sharing with text-based case changing.</p> <pre> 6185 \cs_new:Npn \str_fold_case:n #1 { __str_change_case:nn {#1} { fold } } 6186 \cs_new:Npn \str_lower_case:n #1 { __str_change_case:nn {#1} { lower } } 6187 \cs_new:Npn \str_upper_case:n #1 { __str_change_case:nn {#1} { upper } } 6188 \cs_generate_variant:Nn \str_fold_case:n { V } 6189 \cs_generate_variant:Nn \str_lower_case:n { f } 6190 \cs_generate_variant:Nn \str_upper_case:n { f } 6191 \cs_new:Npn __str_change_case:nn #1 6192 { 6193 \exp_after:wN __str_change_case_aux:nn \exp_after:wN 6194 { \tl_to_str:n {#1} } 6195 } 6196 \cs_new:Npn __str_change_case_aux:nn #1#2 6197 { 6198 __str_change_case_loop:nw {#2} #1 \q_recursion_tail \q_recursion_stop 6199 __str_change_case_result:n { } 6200 } 6201 \cs_new:Npn __str_change_case_output:nw #1#2 __str_change_case_result:n #3 6202 { #2 __str_change_case_result:n { #3 #1 } } 6203 \cs_generate_variant:Nn __str_change_case_output:nw { f } 6204 \cs_new:Npn __str_change_case_end:wn #1 __str_change_case_result:n #2 { #2 } 6205 \cs_new:Npn __str_change_case_loop:nw #1#2 \q_recursion_stop 6206 { 6207 \tl_if_head_is_space:nTF {#2} 6208 { __str_change_case_space:n } 6209 { __str_change_case_char:nN } 6210 {#1} #2 \q_recursion_stop </pre>
--	--

```

6211 }
6212 \use:x
6213 { \cs_new:Npn \exp_not:N \__str_change_case_space:n ##1 \c_space_tl }
6214 {
6215   \__str_change_case_output:nw { ~ }
6216   \__str_change_case_loop:nw {#1}
6217 }
6218 \cs_new:Npn \__str_change_case_char:nN #1#2
6219 {
6220   \quark_if_recursion_tail_stop_do:Nn #2
6221   { \__str_change_case_end:wn }
6222   \cs_if_exist:cTF { c__unicode_ #1 _ #2 _tl }
6223   {
6224     \__str_change_case_output:fw
6225     { \tl_to_str:c { c__unicode_ #1 _ #2 _tl } }
6226   }
6227   { \__str_change_case_char_aux:nN {#1} #2 }
6228   \__str_change_case_loop:nw {#1}
6229 }

```

For Unicode engines there's a look up to see if the current character has a valid one-to-one case change mapping. That's not needed for 8-bit engines: as they don't have `\utex_char:D` all of the changes they can make are hard-coded and so already picked up above.

```

6230 \cs_if_exist:NTF \utex_char:D
6231 {
6232   \cs_new:Npn \__str_change_case_char_aux:nN #1#2
6233   {
6234     \int_compare:nNnTF { \use:c { __str_lookup_ #1 :N } #2 } = { 0 }
6235     { \__str_change_case_output:nw {#2} }
6236     {
6237       \__str_change_case_output:fw
6238       { \utex_char:D \use:c { __str_lookup_ #1 :N } #2 ~ }
6239     }
6240   }
6241   \cs_new_protected:Npn \__str_lookup_lower:N #1 { \tex_lccode:D '#1 }
6242   \cs_new_protected:Npn \__str_lookup_upper:N #1 { \tex_uccode:D '#1 }
6243   \cs_new_eq:NN \__str_lookup_fold:N \__str_lookup_lower:N
6244 }
6245 {
6246   \cs_new:Npn \__str_change_case_char_aux:nN #1#2
6247   { \__str_change_case_output:nw {#2} }
6248 }

```

(End definition for `\str_fold_case:n` and others. These functions are documented on page 109.)

<code>\c_ampersand_str</code>	For all of those strings, use <code>\cs_to_str:N</code> to get characters with the correct category
<code>\c_atsign_str</code>	code without worries
<code>\c_backslash_str</code>	6249 <code>\str_const:Nx \c_ampersand_str { \cs_to_str:N \& }</code>
<code>\c_left_brace_str</code>	6250 <code>\str_const:Nx \c_atsign_str { \cs_to_str:N \@ }</code>
<code>\c_right_brace_str</code>	6251 <code>\str_const:Nx \c_backslash_str { \cs_to_str:N \\ }</code>
<code>\c_circumflex_str</code>	6252 <code>\str_const:Nx \c_left_brace_str { \cs_to_str:N \{ }</code>
<code>\c_colon_str</code>	6253 <code>\str_const:Nx \c_right_brace_str { \cs_to_str:N \} }</code>
<code>\c_dollar_str</code>	6254 <code>\str_const:Nx \c_circumflex_str { \cs_to_str:N \^ }</code>
<code>\c_hash_str</code>	
<code>\c_percent_str</code>	
<code>\c_tilde_str</code>	
<code>\c_underscore_str</code>	

```

6255 \str_const:Nx \c_colon_str      { \cs_to_str:N \: }
6256 \str_const:Nx \c_dollar_str    { \cs_to_str:N \$ }
6257 \str_const:Nx \c_hash_str      { \cs_to_str:N \# }
6258 \str_const:Nx \c_percent_str   { \cs_to_str:N \% }
6259 \str_const:Nx \c_tilde_str     { \cs_to_str:N \~ }
6260 \str_const:Nx \c_underscore_str { \cs_to_str:N \_ }

```

(End definition for `\c_ampersand_str` and others. These variables are documented on page 110.)

`\l_tmpa_str` Scratch strings.

```

\l_tmpb_str
\g_tmpa_str
\g_tmpb_str
6261 \str_new:N \l_tmpa_str
6262 \str_new:N \l_tmpb_str
6263 \str_new:N \g_tmpa_str
6264 \str_new:N \g_tmpb_str

```

(End definition for `\l_tmpa_str` and others. These variables are documented on page 110.)

11.7 Viewing strings

`\str_show:n` Displays a string on the terminal.

```

\str_show:N
\str_show:c
6265 \cs_new_eq:NN \str_show:n \tl_show:n
6266 \cs_new_eq:NN \str_show:N \tl_show:N
6267 \cs_generate_variant:Nn \str_show:N { c }

```

(End definition for `\str_show:n` and `\str_show:N`. These functions are documented on page 109.)

11.8 Unicode data for case changing

```

6268 <@@=unicode>

```

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.

The data required for cross-module manipulations is loaded here: currently this means for `str` and `tl` functions. As such, the prefix used is not `str` but rather `unicode`. For performance (as the entire data set must be read during each run) and as this code comes somewhat early in the load process, there is quite a bit of low-level code here.

As only the data needs to remain at the end of this process, everything is set up inside a group.

```

6269 \group_begin:

```

A read stream is needed. The I/O module is not yet in place *and* we do not want to use up a stream. We therefore use a known free one in format mode or look for the next free one in package mode (covers plain, L^AT_EX 2_ε and ConT_EXt MkII and MkIV).

```

6270 <*\initex>
6271 \tex_chardef:D \g__unicode_data_ior \c_zero
6272 </\initex>
6273 <*\package>
6274 \tex_chardef:D \g__unicode_data_ior
6275 \etex_numexpr:D
6276 \cs_if_exist:NTF \lastallocatedread
6277 { \lastallocatedread }
6278 {

```

```

6279         \cs_if_exist:NTF \c_syst_last_allocated_read
6280         { \c_syst_last_allocated_read }
6281         { \tex_count:D 16 ~ }
6282     }
6283     + 1
6284     \scan_stop:
6285 \endpackage

```

Set up to read each file. As they use C-style comments, there is a need to deal with #. At the same time, spaces are important so they need to be picked up as they are important. Beyond that, the current category code scheme works fine. With no I/O loop available, hard-code one that will work quickly.

```

6286 \cs_set_protected:Npn \__unicode_map_inline:n #1
6287 {
6288     \group_begin:
6289     \tex_catcode:D '# = 12 \scan_stop:
6290     \tex_catcode:D '\ = 10 \scan_stop:
6291     \tex_openin:D \g__unicode_data_ior = #1 \scan_stop:
6292     \cs_if_exist:NT \utex_char:D
6293     { \__unicode_map_loop: }
6294     \tex_closein:D \g__unicode_data_ior
6295     \group_end:
6296 }
6297 \cs_set_protected:Npn \__unicode_map_loop:
6298 {
6299     \tex_if_eof:D \g__unicode_data_ior
6300     \exp_after:wN \use_none:n
6301     \else:
6302     \exp_after:wN \use:n
6303     \fi:
6304     {
6305         \tex_read:D \g__unicode_data_ior to \l__unicode_tmp_tl
6306         \if_meaning:w \c_empty_tl \l__unicode_tmp_tl
6307         \else:
6308             \exp_after:wN \__unicode_parse:w \l__unicode_tmp_tl \q_stop
6309         \fi:
6310         \__unicode_map_loop:
6311     }
6312 }

```

The lead-off parser for each line is common for all of the files. If the line starts with a # it's a comment. There's one special comment line to look out for in `SpecialCasing.txt` as we want to ignore everything after it. As this line does not appear in any other sources and the test is quite quick (there are relatively few comment lines), it can be present in all of the passes.

```

6313 \cs_set_protected:Npn \__unicode_parse:w #1#2 \q_stop
6314 {
6315     \reverse_if:N \if:w \c_hash_str #1
6316     \__unicode_parse_auxi:w #1#2 \q_stop
6317     \else:
6318         \if_int_compare:w \__str_if_eq_x:nn
6319         { \exp_not:n {#2} } { ~Conditional~Mappings~ } = \c_zero
6320         \cs_set_protected:Npn \__unicode_parse:w ##1 \q_stop { }
6321     \fi:
6322     \fi:

```

```
6323 }
```

Storing each exception is always done in the same way: create a constant token list which expands to exactly the mapping. These will have the category codes “now” (so should be letters) but will be detokenized for string use.

```
6324 \cs_set_protected:Npn \__unicode_store:nnnnn #1#2#3#4#5
6325 {
6326   \tl_const:cx { c__unicode_ #2 _ \utex_char:D "#1 _tl }
6327   {
6328     \utex_char:D "#3 ~
6329     \utex_char:D "#4 ~
6330     \tl_if_blank:nF {#5}
6331     { \utex_char:D "#5 }
6332   }
6333 }
```

Parse the main Unicode data file for title case exceptions (the one-to-one lower and upper case mappings it contains will all be covered by the T_EX data).

```
6334 \cs_set_protected:Npn \__unicode_parse_auxi:w
6335   #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
6336   { \__unicode_parse_auxii:w #1 ; }
6337 \cs_set_protected:Npn \__unicode_parse_auxii:w
6338   #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 \q_stop
6339   {
6340     \tl_if_blank:nF {#7}
6341     {
6342       \if_int_compare:w \__str_if_eq_x:nn { #5 ~ } {#7} = \c_zero
6343       \else:
6344         \tl_const:cx
6345           { c__unicode_title_ \utex_char:D "#1 _tl }
6346           { \utex_char:D "#7 }
6347       \fi:
6348     }
6349   }
6350 \__unicode_map_inline:n { UnicodeData.txt }
```

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.

```
6351 \cs_set_protected:Npn \__unicode_parse_auxi:w #1 ; ~ #2 ; ~ #3 ; #4 \q_stop
6352 {
6353   \if_int_compare:w \__str_if_eq_x:nn {#2} { C } = \c_zero
6354   \if_int_compare:w \tex_lccode:D "#1 = "#3 \scan_stop:
6355   \else:
6356     \tl_const:cx
6357       { c__unicode_fold_ \utex_char:D "#1 _tl }
6358       { \utex_char:D "#3 ~ }
6359   \fi:
6360   \else:
6361     \if_int_compare:w \__str_if_eq_x:nn {#2} { F } = \c_zero
6362     \__unicode_parse_auxii:w #1 ~ #3 ~ \q_stop
6363   \fi:
6364   \fi:
6365 }
```



```

6366 \cs_set_protected:Npn \__unicode_parse_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
6367 { \__unicode_store:nnnnn {#1} { fold } {#2} {#3} {#4} }
6368 \__unicode_map_inline:n { CaseFolding.txt }

```

For upper and lower casing special situations, there is a bit more to do as we also have title casing to consider.

```

6369 \cs_set_protected:Npn \__unicode_parse_auxi:w #1 ;~ #2 ;~ #3 ;~ #4 ; #5 \q_stop
6370 {
6371   \use:n { \__unicode_parse_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
6372   \use:n { \__unicode_parse_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
6373   \if_int_compare:w \__str_if_eq_x:nn {#3} {#4} = \c_zero
6374   \else:
6375     \use:n { \__unicode_parse_auxii:w #1 ~ title ~ #3 ~ } ~ \q_stop
6376   \fi:
6377 }
6378 \cs_set_protected:Npn \__unicode_parse_auxii:w #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
6379 {
6380   \tl_if_empty:nF {#4}
6381   { \__unicode_store:nnnnn {#1} {#2} {#3} {#4} {#5} }
6382 }
6383 \__unicode_map_inline:n { SpecialCasing.txt }

```

For the 8-bit engines, the above does nothing but there is some set up needed. There is no expandable character generator primitive so some alternative is needed. As we've not used up hash space for the above, we can go for the fast approach here of one name per letter. Keeping folding and lower casing separate makes the use later a bit easier.

```

6384 \cs_if_exist:NF \utex_char:D
6385 {
6386   \cs_set_protected:Npn \__unicode_tmp:NN #1#2
6387   {
6388     \if_meaning:w \q_recursion_tail #2
6389     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
6390     \fi:
6391     \tl_const:cn { c__unicode_fold_ #1 _tl } {#2}
6392     \tl_const:cn { c__unicode_lower_ #1 _tl } {#2}
6393     \tl_const:cn { c__unicode_upper_ #2 _tl } {#1}
6394     \__unicode_tmp:NN
6395   }
6396   \__unicode_tmp:NN
6397   AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
6398   ? \q_recursion_tail \q_recursion_stop
6399 }

```

All done: tidy up.

```

6400 \group_end:
6401 </initex | package>

```

12 l3seq implementation

The following test files are used for this code: *m3seq002*, *m3seq003*.

```

6402 <*initex | package>
6403 <@@=seq>

```

A sequence is a control sequence whose top-level expansion is of the form “\s__seq __seq_item:n {⟨item₁⟩} ... __seq_item:n {⟨item_n⟩}”, 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 ⟨item_n⟩ \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.

\s__seq The variable is defined in the l3quark module, loaded later.

(End definition for \s__seq.)

__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.

```
6404 \cs_new:Npn \__seq_item:n
6405 {
6406   \__msg_kernel_expandable_error:nn { kernel } { misused-sequence }
6407   \use_none:n
6408 }
```

(End definition for __seq_item:n.)

\l__seq_internal_a_tl Scratch space for various internal uses.

```
\l__seq_internal_b_tl
6409 \tl_new:N \l__seq_internal_a_tl
6410 \tl_new:N \l__seq_internal_b_tl
```

(End definition for \l__seq_internal_a_tl and \l__seq_internal_b_tl.)

__seq_tmp:w Scratch function for internal use.

```
6411 \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.

```
6412 \tl_const:Nn \c_empty_seq { \s__seq }
```

(End definition for \c_empty_seq. This variable is documented on page 121.)

12.1 Allocation and initialisation

\seq_new:N Sequences are initialized to \c_empty_seq.

```
\seq_new:c
6413 \cs_new_protected:Npn \seq_new:N #1
6414 {
6415   \__chk_if_free_cs:N #1
6416   \cs_gset_eq:NN #1 \c_empty_seq
6417 }
6418 \cs_generate_variant:Nn \seq_new:N { c }
```

(End definition for \seq_new:N. This function is documented on page 112.)

\seq_clear:N Clearing a sequence is similar to setting it equal to the empty one.

```
\seq_clear:c
6419 \cs_new_protected:Npn \seq_clear:N #1
\seq_gclear:N
6420 { \seq_set_eq:NN #1 \c_empty_seq }
\seq_gclear:c
6421 \cs_generate_variant:Nn \seq_clear:N { c }
6422 \cs_new_protected:Npn \seq_gclear:N #1
6423 { \seq_gset_eq:NN #1 \c_empty_seq }
6424 \cs_generate_variant:Nn \seq_gclear:N { c }
```

(End definition for `\seq_clear:N` and `\seq_gclear:N`. These functions are documented on page 112.)

`\seq_clear_new:N` Once again we copy code from the token list functions.
`\seq_clear_new:c` 6425 `\cs_new_protected:Npn \seq_clear_new:N #1`
`\seq_gclear_new:N` 6426 `{ \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }`
`\seq_gclear_new:c` 6427 `\cs_generate_variant:Nn \seq_clear_new:N { c }`
6428 `\cs_new_protected:Npn \seq_gclear_new:N #1`
6429 `{ \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }`
6430 `\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 112.)

`\seq_set_eq:NN` Copying a sequence is the same as copying the underlying token list.
`\seq_set_eq:cN` 6431 `\cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN`
`\seq_set_eq:Nc` 6432 `\cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc`
`\seq_set_eq:cc` 6433 `\cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN`
`\seq_gset_eq:NN` 6434 `\cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc`
`\seq_gset_eq:cN` 6435 `\cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN`
`\seq_gset_eq:Nc` 6436 `\cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc`
`\seq_gset_eq:cc` 6437 `\cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN`
6438 `\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 112.)

`\seq_set_from_clist:NN` Setting a sequence from a comma-separated list is done using a simple mapping.
`\seq_set_from_clist:cN` 6439 `\cs_new_protected:Npn \seq_set_from_clist:NN #1#2`
`\seq_set_from_clist:Nc` 6440 `{`
`\seq_set_from_clist:cc` 6441 `\tl_set:Nx #1`
`\seq_set_from_clist:Nn` 6442 `{ \s__seq \clist_map_function:NN #2 __seq_wrap_item:n }`
`\seq_set_from_clist:cn` 6443 `}`
`\seq_gset_from_clist:NN` 6444 `\cs_new_protected:Npn \seq_set_from_clist:Nn #1#2`
`\seq_gset_from_clist:cN` 6445 `{`
`\seq_gset_from_clist:Nc` 6446 `\tl_set:Nx #1`
`\seq_gset_from_clist:cc` 6447 `{ \s__seq \clist_map_function:nN {#2} __seq_wrap_item:n }`
`\seq_gset_from_clist:Nn` 6448 `}`
`\seq_gset_from_clist:cn` 6449 `\cs_new_protected:Npn \seq_gset_from_clist:NN #1#2`
6450 `{`
6451 `\tl_gset:Nx #1`
6452 `{ \s__seq \clist_map_function:NN #2 __seq_wrap_item:n }`
6453 `}`
6454 `\cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2`
6455 `{`
6456 `\tl_gset:Nx #1`
6457 `{ \s__seq \clist_map_function:nN {#2} __seq_wrap_item:n }`
6458 `}`
6459 `\cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }`
6460 `\cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }`
6461 `\cs_generate_variant:Nn \seq_set_from_clist:Nn { c }`
6462 `\cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }`
6463 `\cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }`
6464 `\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 112.)

`\seq_set_split:Nnn` When the separator is empty, everything is very simple, just map `__seq_wrap_item:n`
`\seq_set_split:NnV` through the items of the last argument. For non-trivial separators, the goal is to split
`\seq_gset_split:Nnn` a given token list at the marker, strip spaces from each item, and remove one set of
`\seq_gset_split:NnV` outer braces if after removing leading and trailing spaces the item is enclosed within
`__seq_set_split:Nnnn` braces. After `\tl_replace_all:Nnn`, the token list `\l__seq_internal_a_tl` is a repe-
`__seq_set_split_auxi:w` tition of the pattern `__seq_set_split_auxi:w \prg_do_nothing: <item with spaces>`
`__seq_set_split_auxii:w` `__seq_set_split_end:.` Then, x-expansion causes `__seq_set_split_auxi:w` to trim
`__seq_set_split_end:` spaces, and leaves its result as `__seq_set_split_auxii:w <trimmed item> __seq-`
`__seq_set_split_end:` `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.

```

6465 \cs_new_protected:Npn \seq_set_split:Nnn
6466 { \__seq_set_split:Nnnn \tl_set:Nx }
6467 \cs_new_protected:Npn \seq_gset_split:Nnn
6468 { \__seq_set_split:Nnnn \tl_gset:Nx }
6469 \cs_new_protected:Npn \__seq_set_split:Nnnn #1#2#3#4
6470 {
6471   \tl_if_empty:nTF {#3}
6472   {
6473     \tl_set:Nn \l__seq_internal_a_tl
6474     { \tl_map_function:nN {#4} \__seq_wrap_item:n }
6475   }
6476   {
6477     \tl_set:Nn \l__seq_internal_a_tl
6478     {
6479       \__seq_set_split_auxi:w \prg_do_nothing:
6480       #4
6481       \__seq_set_split_end:
6482     }
6483     \tl_replace_all:Nnn \l__seq_internal_a_tl { #3 }
6484     {
6485       \__seq_set_split_end:
6486       \__seq_set_split_auxi:w \prg_do_nothing:
6487     }
6488     \tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
6489   }
6490   #1 #2 { \s__seq \l__seq_internal_a_tl }
6491 }
6492 \cs_new:Npn \__seq_set_split_auxi:w #1 \__seq_set_split_end:
6493 {
6494   \exp_not:N \__seq_set_split_auxii:w
6495   \exp_args:No \tl_trim_spaces:n {#1}
6496   \exp_not:N \__seq_set_split_end:
6497 }
6498 \cs_new:Npn \__seq_set_split_auxii:w #1 \__seq_set_split_end:
6499 { \__seq_wrap_item:n {#1} }
6500 \cs_generate_variant:Nn \seq_set_split:Nnn { NnV }
6501 \cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }

```

(End definition for `\seq_set_split:Nnn` and others. These functions are documented on page 113.)

`\seq_concat:NNN` When concatenating sequences, one must remove the leading `\s__seq` of the second
`\seq_concat:ccc`
`\seq_gconcat:NNN`
`\seq_gconcat:ccc`

sequence. The result starts with `\s__seq` (of the first sequence), which stops `f`-expansion.

```

6502 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
6503 { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
6504 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
6505 { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
6506 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
6507 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for `\seq_concat:NNN` and `\seq_gconcat:NNN`. These functions are documented on page 113.)

```

\seq_if_exist_p:N Copies of the cs functions defined in l3basics.
\seq_if_exist_p:c
\seq_if_exist:NTF
\seq_if_exist:cTF
\seq_if_exist:cTF

```

(End definition for `\seq_if_exist:NTF`. This function is documented on page 113.)

12.2 Appending data to either end

```

\seq_put_left:Nn When adding to the left of a sequence, remove \s__seq. This is done by \__seq_put_
\seq_put_left:NV left_aux:w, which also stops f-expansion.
\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

```

(End definition for `\seq_put_left:Nn`, `\seq_gput_left:Nn`, and `__seq_put_left_aux:w`. These functions are documented on page 113.)

```

\seq_put_right:Nn Since there is no trailing marker, adding an item to the right of a sequence simply means
\seq_put_right:NV wrapping it in \__seq_item:n.
\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
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn

```

```

6537 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
6538 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
6539 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
6540 \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 113.)

12.3 Modifying sequences

`__seq_wrap_item:n` This function converts its argument to a proper sequence item in an x-expansion context.

```

6541 \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.

```

6542 \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.

```

\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
\__seq_remove_duplicates:NN
6543 \cs_new_protected:Npn \seq_remove_duplicates:N
6544 { \__seq_remove_duplicates:NN \seq_set_eq:NN }
6545 \cs_new_protected:Npn \seq_gremove_duplicates:N
6546 { \__seq_remove_duplicates:NN \seq_gset_eq:NN }
6547 \cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2
6548 {
6549   \seq_clear:N \l__seq_remove_seq
6550   \seq_map_inline:Nn #2
6551   {
6552     \seq_if_in:NnF \l__seq_remove_seq {##1}
6553     { \seq_put_right:Nn \l__seq_remove_seq {##1} }
6554   }
6555   #1 #2 \l__seq_remove_seq
6556 }
6557 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
6558 \cs_generate_variant:Nn \seq_gremove_duplicates:N { c }

```

(End definition for `\seq_remove_duplicates:N`, `\seq_gremove_duplicates:N`, and `__seq_remove_duplicates:NN`. These functions are documented on page 116.)

`\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 `__seq_pop_right:NNN`, using a “flexible” x-type expansion to do most of the work. `\seq_remove_all:cn` 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 will happen repeatedly until the entire sequence has been scanned. The code is set up to avoid needing and intermediate scratch list: the lead-off x-type expansion (`#1 #2 {#2}`) will ensure that nothing is lost.

```

6559 \cs_new_protected:Npn \seq_remove_all:Nn
6560 { \__seq_remove_all_aux:NNn \tl_set:Nx }
6561 \cs_new_protected:Npn \seq_gremove_all:Nn

```

```

6562 { \_seq_remove_all_aux:Nn \tl_gset:Nx }
6563 \cs_new_protected:Npn \_seq_remove_all_aux:Nn #1#2#3
6564 {
6565   \_seq_push_item_def:n
6566   {
6567     \str_if_eq:nnT {##1} {#3}
6568     {
6569       \if_false: { \fi: }
6570       \tl_set:Nn \l__seq_internal_b_tl {##1}
6571       #1 #2
6572       { \if_false: } \fi:
6573       \exp_not:o {#2}
6574       \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
6575       { \use_none:n }
6576     }
6577     \_seq_wrap_item:n {##1}
6578   }
6579   \tl_set:Nn \l__seq_internal_a_tl {#3}
6580   #1 #2 {#2}
6581   \_seq_pop_item_def:
6582 }
6583 \cs_generate_variant:Nn \seq_remove_all:Nn { c }
6584 \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:Nn`. These functions are documented on page 116.)

```

\seq_reverse:N Previously, \seq_reverse:N was coded by collecting the items in reverse order after an
\seq_reverse:c \exp_stop_f: marker.
\seq_greverse:N
\seq_greverse:c \cs_new_protected:Npn \seq_reverse:N #1
\__seq_reverse:NN {
\__seq_reverse_item:nwn \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.

```

6585 \cs_new_protected:Npn \seq_reverse:N

```

```

6586 { \__seq_reverse:NN \tl_set:Nx }
6587 \cs_new_protected:Npn \seq_greverse:N
6588 { \__seq_reverse:NN \tl_gset:Nx }
6589 \cs_new_protected:Npn \__seq_reverse:NN #1 #2
6590 {
6591   \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
6592   \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
6593   #1 #2 { #2 \exp_not:n { } }
6594   \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
6595 }
6596 \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
6597 {
6598   #2
6599   \exp_not:n { \__seq_item:n {#1} #3 }
6600 }
6601 \cs_generate_variant:Nn \seq_reverse:N { c }
6602 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End definition for `\seq_reverse:N` and others. These functions are documented on page 116.)

`\seq_sort:Nn` Implemented in `l3sort`.

`\seq_sort:cn`

(End definition for `\seq_sort:Nn` and `\seq_gsort:Nn`. These functions are documented on page 116.)

`\seq_gsort:Nn`

`\seq_gsort:cn`

12.4 Sequence conditionals

`\seq_if_empty_p:N` Similar to token lists, we compare with the empty sequence.

`\seq_if_empty_p:c`

`\seq_if_empty:NTF`

`\seq_if_empty:cTF`

```

6603 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
6604 {
6605   \if_meaning:w #1 \c_empty_seq
6606   \prg_return_true:
6607   \else:
6608     \prg_return_false:
6609   \fi:
6610 }
6611 \cs_generate_variant:Nn \seq_if_empty_p:N { c }
6612 \cs_generate_variant:Nn \seq_if_empty:NT { c }
6613 \cs_generate_variant:Nn \seq_if_empty:NF { c }
6614 \cs_generate_variant:Nn \seq_if_empty:NTF { c }

```

(End definition for `\seq_if_empty:NTF`. This function is documented on page 116.)

`\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 will break returning `\prg_return_false:`. Everything is inside a group so that `__seq_item:n` is preserved in nested situations.

`\seq_if_in:cnTF`

`\seq_if_in:cVTF`

`\seq_if_in:cvTF`

`\seq_if_in:coTF`

`\seq_if_in:cxTF`

`__seq_if_in:`

```

6615 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
6616 { T , F , TF }
6617 {
6618   \group_begin:
6619     \tl_set:Nn \l__seq_internal_a_tl {#2}
6620     \cs_set_protected:Npn \__seq_item:n ##1
6621     {

```



```

6622         \tl_set:Nn \l__seq_internal_b_tl {##1}
6623         \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
6624             \exp_after:wN \__seq_if_in:
6625         \fi:
6626     }
6627     #1
6628     \group_end:
6629     \prg_return_false:
6630     \__prg_break_point:
6631 }
6632 \cs_new:Npn \__seq_if_in:
6633 { \__prg_break:n { \group_end: \prg_return_true: } }
6634 \cs_generate_variant:Nn \seq_if_in:NnT { c , cV , cv , co , cx }
6635 \cs_generate_variant:Nn \seq_if_in:NnT { c , cV , cv , co , cx }
6636 \cs_generate_variant:Nn \seq_if_in:NnF { c , cV , cv , co , cx }
6637 \cs_generate_variant:Nn \seq_if_in:NnF { c , cV , cv , co , cx }
6638 \cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }
6639 \cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }

```

(End definition for `\seq_if_in:NnTF` and `__seq_if_in:`. These functions are documented on page 116.)

12.5 Recovering data from sequences

`__seq_pop:NNNN` The two pop functions share their emptiness tests. We also use a common emptiness test
`__seq_pop_TF:NNNN` for all branching get and pop functions.

```

6640 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
6641 {
6642     \if_meaning:w #3 \c_empty_seq
6643         \tl_set:Nn #4 { \q_no_value }
6644     \else:
6645         #1#2#3#4
6646     \fi:
6647 }
6648 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
6649 {
6650     \if_meaning:w #3 \c_empty_seq
6651         % \tl_set:Nn #4 { \q_no_value }
6652         \prg_return_false:
6653     \else:
6654         #1#2#3#4
6655         \prg_return_true:
6656     \fi:
6657 }

```

(End definition for `__seq_pop:NNNN` and `__seq_pop_TF:NNNN`.)

`\seq_get_left:NN` Getting an item from the left of a sequence is pretty easy: just trim off the first item
`\seq_get_left:cN` after `__seq_item:n` at the start. We append a `\q_no_value` item to cover the case of
`__seq_get_left:wnw` an empty sequence

```

6658 \cs_new_protected:Npn \seq_get_left:NN #1#2
6659 {
6660     \tl_set:Nx #2
6661     {
6662         \exp_after:wN \__seq_get_left:wnw

```

```

6663         #1 \__seq_item:n { \q_no_value } \q_stop
6664     }
6665 }
6666 \cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \q_stop
6667 { \exp_not:n {#2} }
6668 \cs_generate_variant:Nn \seq_get_left:NN { c }

```

(End definition for `\seq_get_left:NN` and `__seq_get_left:wnw`. These functions are documented on page 113.)

```

\seq_pop_left:NN The approach to popping an item is pretty similar to that to get an item, with the only
\seq_pop_left:cN difference being that the sequence itself has to be redefined. This makes it more sensible
\seq_gpop_left:NN to use an auxiliary function for the local and global cases.
\seq_gpop_left:cN
\__seq_pop_left:NNN
\__seq_pop_left:wnwNNN
6669 \cs_new_protected:Npn \seq_pop_left:NN
6670 { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_set:Nn }
6671 \cs_new_protected:Npn \seq_gpop_left:NN
6672 { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_gset:Nn }
6673 \cs_new_protected:Npn \__seq_pop_left:NNN #1#2#3
6674 { \exp_after:wN \__seq_pop_left:wnwNNN #2 \q_stop #1#2#3 }
6675 \cs_new_protected:Npn \__seq_pop_left:wnwNNN
6676 #1 \__seq_item:n #2#3 \q_stop #4#5#6
6677 {
6678     #4 #5 { #1 #3 }
6679     \tl_set:Nn #6 {#2}
6680 }
6681 \cs_generate_variant:Nn \seq_pop_left:NN { c }
6682 \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End definition for `\seq_pop_left:NN` and others. These functions are documented on page 114.)

```

\seq_get_right:NN First remove \s__seq and prepend \q_no_value, then take two arguments at a time.
\seq_get_right:cN Before the right-hand end of the sequence, this is a brace group followed by \__seq_
\__seq_get_right_loop:nn item:n, both removed by \use_none:nn. At the end of the sequence, the two question
marks are taken by \use_none:nn, and the assignment is placed before the right-most
item. In the next iteration, \__seq_get_right_loop:nn receives two empty arguments,
and \use_none:nn stops the loop.

```

```

6683 \cs_new_protected:Npn \seq_get_right:NN #1#2
6684 {
6685     \exp_after:wN \use_i_ii:nnn
6686     \exp_after:wN \__seq_get_right_loop:nn
6687     \exp_after:wN \q_no_value
6688     #1
6689     { ?? \tl_set:Nn #2 }
6690     { } { }
6691 }
6692 \cs_new_protected:Npn \__seq_get_right_loop:nn #1#2
6693 {
6694     \use_none:nn #2 {#1}
6695     \__seq_get_right_loop:nn
6696 }
6697 \cs_generate_variant:Nn \seq_get_right:NN { c }

```

(End definition for `\seq_get_right:NN` and `__seq_get_right_loop:nn`. These functions are documented on page 114.)

`\seq_pop_right: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 will be 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.

```

6698 \cs_new_protected:Npn \seq_pop_right:NN
6699   { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_set:Nx }
6700 \cs_new_protected:Npn \seq_gpop_right:NN
6701   { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_gset:Nx }
6702 \cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
6703   {
6704     \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
6705     \cs_set_eq:NN \__seq_item:n \scan_stop:
6706     #1 #2
6707     { \if_false: } \fi: \s__seq
6708       \exp_after:wN \use_i:nnn
6709       \exp_after:wN \__seq_pop_right_loop:nn
6710     #2
6711     {
6712       \if_false: { \fi: }
6713       \tl_set:Nx #3
6714     }
6715     { } \use_none:nn
6716     \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
6717   }
6718 \cs_new:Npn \__seq_pop_right_loop:nn #1#2
6719   {
6720     #2 { \exp_not:n {#1} }
6721     \__seq_pop_right_loop:nn
6722   }
6723 \cs_generate_variant:Nn \seq_pop_right:NN { c }
6724 \cs_generate_variant:Nn \seq_gpop_right:NN { c }

```

(End definition for `\seq_pop_right:NN` and others. These functions are documented on page 114.)

`\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_left:cNTF
\seq_get_right:NNTF
\seq_get_right:cNTF
6725 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
6726   { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
6727 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
6728   { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
6729 \cs_generate_variant:Nn \seq_get_left:NNT { c }
6730 \cs_generate_variant:Nn \seq_get_left:NNF { c }
6731 \cs_generate_variant:Nn \seq_get_left:NNTF { c }
6732 \cs_generate_variant:Nn \seq_get_right:NNT { c }
6733 \cs_generate_variant:Nn \seq_get_right:NNF { c }
6734 \cs_generate_variant:Nn \seq_get_right:NNTF { c }

```

(End definition for \seq_get_left:NNTF and \seq_get_right:NNTF. These functions are documented on page 115.)

\seq_pop_left:NNTF More or less the same for popping.
\seq_pop_left:cNTF
\seq_gpop_left:NNTF
\seq_gpop_left:cNTF
\seq_pop_right:NNTF
\seq_pop_right:cNTF
\seq_gpop_right:NNTF
\seq_gpop_right:cNTF

```

6735 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF }
6736 { \__seq_pop_TF:NNTN \__seq_pop_left:NN \tl_set:Nn #1 #2 }
6737 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF }
6738 { \__seq_pop_TF:NNTN \__seq_pop_left:NN \tl_gset:Nn #1 #2 }
6739 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2 { T , F , TF }
6740 { \__seq_pop_TF:NNTN \__seq_pop_right:NN \tl_set:Nx #1 #2 }
6741 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2 { T , F , TF }
6742 { \__seq_pop_TF:NNTN \__seq_pop_right:NN \tl_gset:Nx #1 #2 }
6743 \cs_generate_variant:Nn \seq_pop_left:NNT { c }
6744 \cs_generate_variant:Nn \seq_pop_left:NNTF { c }
6745 \cs_generate_variant:Nn \seq_gpop_left:NNT { c }
6746 \cs_generate_variant:Nn \seq_gpop_left:NNTF { c }
6747 \cs_generate_variant:Nn \seq_gpop_left:NNTF { c }
6748 \cs_generate_variant:Nn \seq_gpop_left:NNTF { c }
6749 \cs_generate_variant:Nn \seq_pop_right:NNT { c }
6750 \cs_generate_variant:Nn \seq_pop_right:NNTF { c }
6751 \cs_generate_variant:Nn \seq_pop_right:NNTF { c }
6752 \cs_generate_variant:Nn \seq_gpop_right:NNT { c }
6753 \cs_generate_variant:Nn \seq_gpop_right:NNTF { c }
6754 \cs_generate_variant:Nn \seq_gpop_right:NNTF { c }

```

(End definition for \seq_pop_left:NNTF and others. These functions are documented on page 115.)

\seq_item:Nn The idea here is to find the offset of the item from the left, then use a loop
\seq_item:cn to grab the correct item. If the resulting offset is too large, then the stop code
__seq_item:wNn { ? __prg_break: } { } will be used by the auxiliary, terminating the loop and re-
__seq_item:nN turning nothing at all.
__seq_item:nnn

```

6755 \cs_new:Npn \seq_item:Nn #1
6756 { \exp_after:wN \__seq_item:wNn #1 \q_stop #1 }
6757 \cs_new:Npn \__seq_item:wNn \s__seq #1 \q_stop #2#3
6758 {
6759   \exp_args:Nf \__seq_item:nnn
6760   { \exp_args:Nf \__seq_item:nN { \int_eval:n {#3} } #2 }
6761   #1
6762   { ? \__prg_break: } { }
6763   \__prg_break_point:
6764 }
6765 \cs_new:Npn \__seq_item:nN #1#2
6766 {
6767   \int_compare:nNnTF {#1} < \c_zero
6768   { \int_eval:n { \seq_count:N #2 + \c_one + #1 } }
6769   {#1}
6770 }
6771 \cs_new:Npn \__seq_item:nnn #1#2#3
6772 {
6773   \use_none:n #2
6774   \int_compare:nNnTF {#1} = \c_one
6775   { \__prg_break:n { \exp_not:n {#3} } }
6776   { \exp_args:Nf \__seq_item:nnn { \int_eval:n { #1 - 1 } } }
6777 }
6778 \cs_generate_variant:Nn \seq_item:Nn { c }

```

(End definition for `\seq_item:Nn` and others. These functions are documented on page 114.)

12.6 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.

```
6779 \cs_new:Npn \seq_map_break:
6780 { \__prg_map_break:Nn \seq_map_break: { } }
6781 \cs_new:Npn \seq_map_break:n
6782 { \__prg_map_break:Nn \seq_map_break: }
```

(End definition for `\seq_map_break:` and `\seq_map_break:n`. These functions are documented on page 117.)

`\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`. This is done as by noting that every odd token in the sequence must be `__seq_item:n`, which can be gobbled by `\use_none:n`. At the end of the loop, #2 is instead `? \seq_map_break:`, which therefore breaks the loop without needing to do a (relatively-expensive) quark test.

```
6783 \cs_new:Npn \seq_map_function:NN #1#2
6784 {
6785   \exp_after:wN \use_i_ii:nnn
6786   \exp_after:wN \__seq_map_function:NNn
6787   \exp_after:wN #2
6788   #1
6789   { ? \seq_map_break: } { }
6790   \__prg_break_point:Nn \seq_map_break: { }
6791 }
6792 \cs_new:Npn \__seq_map_function:NNn #1#2#3
6793 {
6794   \use_none:n #2
6795   #1 {#3}
6796   \__seq_map_function:NNn #1
6797 }
6798 \cs_generate_variant:Nn \seq_map_function:NN { c }
```

(End definition for `\seq_map_function:NN` and `__seq_map_function:NNn`. These functions are documented on page 116.)

`__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.

```
\__seq_push_item_def:x
\__seq_push_item_def:
\__seq_pop_item_def:
6799 \cs_new_protected:Npn \__seq_push_item_def:n
6800 {
6801   \__seq_push_item_def:
6802   \cs_gset:Npn \__seq_item:n ##1
6803 }
6804 \cs_new_protected:Npn \__seq_push_item_def:x
6805 {
6806   \__seq_push_item_def:
6807   \cs_gset:Npx \__seq_item:n ##1
6808 }
6809 \cs_new_protected:Npn \__seq_push_item_def:
```

```

6810 {
6811     \int_gincr:N \g__prg_map_int
6812     \cs_gset_eq:cN { __prg_map_ \int_use:N \g__prg_map_int :w }
6813     \__seq_item:n
6814 }
6815 \cs_new_protected:Npn \__seq_pop_item_def:
6816 {
6817     \cs_gset_eq:Nc \__seq_item:n
6818     { __prg_map_ \int_use:N \g__prg_map_int :w }
6819     \int_gdecr:N \g__prg_map_int
6820 }

```

(End definition for __seq_push_item_def:n, __seq_push_item_def:, and __seq_pop_item_def:.)

\seq_map_inline:Nn 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.

\seq_map_inline:cn

```

6821 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
6822 {
6823     \__seq_push_item_def:n {#2}
6824     #1
6825     \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
6826 }
6827 \cs_generate_variant:Nn \seq_map_inline:Nn { c }

```

(End definition for \seq_map_inline:Nn. This function is documented on page 117.)

\seq_map_variable:NNn 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.

\seq_map_variable:Ncn

\seq_map_variable:cNn

\seq_map_variable:ccn

```

6828 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
6829 {
6830     \__seq_push_item_def:x
6831     {
6832         \tl_set:Nn \exp_not:N #2 {##1}
6833         \exp_not:n {#3}
6834     }
6835     #1
6836     \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
6837 }
6838 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
6839 \cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }

```

(End definition for \seq_map_variable:NNn. This function is documented on page 117.)

\seq_count:N Counting the items in a sequence 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.

\seq_count:c

__seq_count:n

```

6840 \cs_new:Npn \seq_count:N #1
6841 {
6842     \int_eval:n
6843     {
6844         0
6845         \seq_map_function:NN #1 \__seq_count:n
6846     }
6847 }
6848 \cs_new:Npn \__seq_count:n #1 { + \c_one }
6849 \cs_generate_variant:Nn \seq_count:N { c }

```

(End definition for `\seq_count:N` and `_seq_count:n`. These functions are documented on page 118.)

12.7 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`.

```

\seq_use:cnnn
\_seq_use:NNnNnn
\_seq_use_setup:w
\_seq_use:nwwwnwn
\_seq_use:nwnn
\seq_use:Nn
\seq_use:cn
6850 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
6851 {
6852   \seq_if_exist:NTF #1
6853   {
6854     \int_case:nnF { \seq_count:N #1 }
6855     {
6856       { 0 } { }
6857       { 1 } { \exp_after:wN \_seq_use:NNnNnn #1 ? { } { } }
6858       { 2 } { \exp_after:wN \_seq_use:NNnNnn #1 {#2} }
6859     }
6860     {
6861       \exp_after:wN \_seq_use_setup:w #1 \_seq_item:n
6862       \q_mark { \_seq_use:nwwwnwn {#3} }
6863       \q_mark { \_seq_use:nwnn {#4} }
6864       \q_stop { }
6865     }
6866   }
6867   {
6868     \_msg_kernel_expandable_error:nnn
6869     { kernel } { bad-variable } {#1}
6870   }
6871 }
6872 \cs_generate_variant:Nn \seq_use:Nnnn { c }
6873 \cs_new:Npn \_seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
6874 \cs_new:Npn \_seq_use_setup:w \s__seq { \_seq_use:nwwwnwn { } }
6875 \cs_new:Npn \_seq_use:nwwwnwn
6876   #1 \_seq_item:n #2 \_seq_item:n #3 \_seq_item:n #4#5
6877   \q_mark #6#7 \q_stop #8
6878   {
6879     #6 \_seq_item:n {#3} \_seq_item:n {#4} #5
6880     \q_mark {#6} #7 \q_stop { #8 #1 #2 }
6881   }
6882 \cs_new:Npn \_seq_use:nwnn #1 \_seq_item:n #2 #3 \q_stop #4
6883   { \exp_not:n { #4 #1 #2 } }
6884 \cs_new:Npn \seq_use:Nn #1#2
6885   { \seq_use:Nnnn #1 {#2} {#2} {#2} }
6886 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End definition for `\seq_use:Nnnn` and others. These functions are documented on page 118.)

12.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

`\seq_push:Nn` Pushing to a sequence is the same as adding on the left.

```

\seq_push:NV
\seq_push:Nv
\seq_push:No
\seq_push:Nx
\seq_push:cn
\seq_push:cV
\seq_push:cV
\seq_push:co
\seq_push:cx
\seq_gpush:Nn
\seq_gpush:NV
\seq_gpush:Nv

```

```

6888 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
6889 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
6890 \cs_new_eq:NN \seq_push:No \seq_put_left:No
6891 \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
6892 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
6893 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
6894 \cs_new_eq:NN \seq_push:cv \seq_put_left:cv
6895 \cs_new_eq:NN \seq_push:co \seq_put_left:co
6896 \cs_new_eq:NN \seq_push:cx \seq_put_left:cx
6897 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
6898 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
6899 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
6900 \cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
6901 \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
6902 \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
6903 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
6904 \cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
6905 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
6906 \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 120.)

```

\seq_get:NN In most cases, getting items from the stack does not need to specify that this is from the
\seq_get:cN left. So alias are provided.
\seq_pop:NN 6907 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
\seq_pop:cN 6908 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\seq_gpop:NN 6909 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
\seq_gpop:cN 6910 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
6911 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
6912 \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 119.)

```

\seq_get:NNTF More copies.
\seq_get:cNTF 6913 \prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }
\seq_pop:NNTF 6914 \prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }
\seq_pop:cNTF 6915 \prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
\seq_gpop:NNTF 6916 \prg_new_eq_conditional:NNn \seq_pop:cN \seq_pop_left:cN { T , F , TF }
\seq_gpop:cNTF 6917 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
6918 \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 119.)

12.9 Viewing sequences

`\seq_show:N` Apply the general `__msg_show_variable:NNNnn`.

```

\seq_show:c 6919 \cs_new_protected:Npn \seq_show:N #1
6920 {
6921   \__msg_show_variable:NNNnn #1
6922   \seq_if_exist:NTF \seq_if_empty:NTF { seq }
6923   { \seq_map_function:NN #1 \__msg_show_item:n }
6924 }
6925 \cs_generate_variant:Nn \seq_show:N { c }

```

(End definition for `\seq_show:N`. This function is documented on page 122.)

12.10 Scratch sequences

`\l_tmpa_seq` Temporary comma list variables.

`\l_tmpp_seq` 6926 \seq_new:N \l_tmpa_seq

`\g_tmpa_seq` 6927 \seq_new:N \l_tmpp_seq

`\g_tmpp_seq` 6928 \seq_new:N \g_tmpa_seq

6929 \seq_new:N \g_tmpp_seq

(End definition for \l_tmpa_seq and others. These variables are documented on page 122.)

6930 \</initex | package>

13 l3clist implementation

The following test files are used for this code: m3clist002.

6931 \<*initex | package>

6932 \<@@=clist>

`\c_empty_clist` An empty comma list is simply an empty token list.

6933 \cs_new_eq:NN \c_empty_clist \c_empty_tl

(End definition for \c_empty_clist. This variable is documented on page 131.)

`\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`

6934 \tl_new:N \l__clist_internal_clist

(End definition for \l__clist_internal_clist.)

`__clist_tmp:w` A temporary function for various purposes.

6935 \cs_new_protected:Npn __clist_tmp:w { }

(End definition for __clist_tmp:w.)

13.1 Allocation and initialisation

`\clist_new:N` Internally, comma lists are just token lists.

`\clist_new:c` 6936 \cs_new_eq:NN \clist_new:N \tl_new:N

6937 \cs_new_eq:NN \clist_new:c \tl_new:c

(End definition for \clist_new:N. This function is documented on page 123.)

`\clist_const:Nn` Creating and initializing a constant comma list is done in a way similar to `\clist_set:Nn` and `\clist_gset:Nn`, being careful to strip spaces.

`\clist_const:cn` 6938 \cs_new_protected:Npn \clist_const:Nn #1#2

`\clist_const:Nx` 6939 { \tl_const:Nx #1 { __clist_trim_spaces:n {#2} } }

`\clist_const:cx` 6940 \cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }

(End definition for \clist_const:Nn. This function is documented on page 123.)

`\clist_clear:N` Clearing comma lists is just the same as clearing token lists.

`\clist_clear:c` 6941 \cs_new_eq:NN \clist_clear:N \tl_clear:N

`\clist_gclear:N` 6942 \cs_new_eq:NN \clist_clear:c \tl_clear:c

`\clist_gclear:c` 6943 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N

6944 \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 123.)

```
\clist_clear_new:N Once again a copy from the token list functions.
\clist_clear_new:c 6945 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
\clist_gclear_new:N 6946 \cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c
\clist_gclear_new:c 6947 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
6948 \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 123.)

```
\clist_set_eq:NN Once again, these are simple copies from the token list functions.
\clist_set_eq:cN 6949 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
\clist_set_eq:Nc 6950 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
\clist_set_eq:cc 6951 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
\clist_gset_eq:NN 6952 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
\clist_gset_eq:cN 6953 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\clist_gset_eq:Nc 6954 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\clist_gset_eq:cN 6955 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
6956 \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 124.)

```
\clist_set_from_seq:NN Setting a comma list from a comma-separated list is done using a simple mapping. We
\clist_set_from_seq:cN wrap most items with \exp_not:n, and a comma. Items which contain a comma or a
\clist_set_from_seq:Nc space are surrounded by an extra set of braces. The first comma must be removed, except
\clist_set_from_seq:cc in the case of an empty comma-list.
\clist_gset_from_seq:NN 6957 \cs_new_protected:Npn \clist_set_from_seq:NN
\clist_gset_from_seq:cN 6958 { \__clist_set_from_seq:NNNN \clist_clear:N \tl_set:Nx }
\clist_gset_from_seq:Nc 6959 \cs_new_protected:Npn \clist_gset_from_seq:NN
\clist_gset_from_seq:cc 6960 { \__clist_set_from_seq:NNNN \clist_gclear:N \tl_gset:Nx }
\__clist_set_from_seq:NNNN 6961 \cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
6962 {
6963   \seq_if_empty:NTF #4
6964   { #1 #3 }
6965   {
6966     #2 #3
6967     {
6968       \exp_last_unbraced:Nf \use_none:n
6969       { \seq_map_function:NN #4 \__clist_wrap_item:n }
6970     }
6971   }
6972 }
6973 \cs_new:Npn \__clist_wrap_item:n #1
6974 {
6975   ,
6976   \tl_if_empty:oTF { \__clist_set_from_seq:w #1 ~ , #1 ~ }
6977   { \exp_not:n {#1} }
6978   { \exp_not:n { {#1} } }
6979 }
6980 \cs_new:Npn \__clist_set_from_seq:w #1 , #2 ~ { }
6981 \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
```

```

6982 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
6983 \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
6984 \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 124.)

```

\clist_concat:NNN Concatenating comma lists is not quite as easy as it seems, as there needs to be the
\clist_concat:ccc correct addition of a comma to the output. So a little work to do.
\clist_gconcat:NNN
\clist_gconcat:ccc
\__clist_concat:NNNN
6985 \cs_new_protected:Npn \clist_concat:NNN
6986 { \__clist_concat:NNNN \tl_set:Nx }
6987 \cs_new_protected:Npn \clist_gconcat:NNN
6988 { \__clist_concat:NNNN \tl_gset:Nx }
6989 \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
6990 {
6991     #1 #2
6992     {
6993         \exp_not:o #3
6994         \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
6995         \exp_not:o #4
6996     }
6997 }
6998 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
6999 \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 124.)

```

\clist_if_exist_p:N Copies of the cs functions defined in l3basics.
\clist_if_exist_p:c
\clist_if_exist:NTF
\clist_if_exist:cTF
7000 \prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N
7001 { TF , T , F , p }
7002 \prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c
7003 { TF , T , F , p }

```

(End definition for `\clist_if_exist:NTF`. This function is documented on page 124.)

13.2 Removing spaces around items

`__clist_trim_spaces_generic:nw` This expands to the `<code>`, followed by a brace group containing the `<item>`, with leading and trailing spaces removed. The calling function is responsible for inserting `\q_mark` in front of the `<item>`, as well as testing for the end of the list. We reuse a `\tl` internal function, whose first argument must start with `\q_mark`. That trims the item #2, then feeds the result (after having to do an o-type expansion) to `__clist_trim_spaces_generic:nn` which places the `<code>` in front of the `<trimmed item>`.

```

7004 \cs_new:Npn \__clist_trim_spaces_generic:nw #1#2 ,
7005 {
7006     \__tl_trim_spaces:nn {#2}
7007     { \exp_args:No \__clist_trim_spaces_generic:nn } {#1}
7008 }
7009 \cs_new:Npn \__clist_trim_spaces_generic:nn #1#2 { #2 {#1} }

```

(End definition for `__clist_trim_spaces_generic:nw` and `__clist_trim_spaces_generic:nn`.)

`__clist_trim_spaces:n` The first argument of `__clist_trim_spaces:nn` is initially empty, and later a comma, namely, as soon as we have added an item to the resulting list. The auxiliary tests for the end of the list, and also prevents empty arguments from finding their way into the output.

```

7010 \cs_new:Npn \__clist_trim_spaces:n #1
7011 {
7012   \__clist_trim_spaces_generic:nw
7013   { \__clist_trim_spaces:nn { } }
7014   \q_mark #1 ,
7015   \q_recursion_tail, \q_recursion_stop
7016 }
7017 \cs_new:Npn \__clist_trim_spaces:nn #1 #2
7018 {
7019   \quark_if_recursion_tail_stop:n {#2}
7020   \tl_if_empty:nTF {#2}
7021   {
7022     \__clist_trim_spaces_generic:nw
7023     { \__clist_trim_spaces:nn {#1} } \q_mark
7024   }
7025   {
7026     #1 \exp_not:n {#2}
7027     \__clist_trim_spaces_generic:nw
7028     { \__clist_trim_spaces:nn { , } } \q_mark
7029   }
7030 }

```

(End definition for `__clist_trim_spaces:n` and `__clist_trim_spaces:nn`.)

13.3 Adding data to comma lists

```

\clist_set:Nn
\clist_set:NV 7031 \cs_new_protected:Npn \clist_set:Nn #1#2
\clist_set:No 7032 { \tl_set:Nx #1 { \__clist_trim_spaces:n {#2} } }
\clist_set:Nx 7033 \cs_new_protected:Npn \clist_gset:Nn #1#2
\clist_set:cn 7034 { \tl_gset:Nx #1 { \__clist_trim_spaces:n {#2} } }
\clist_set:cV 7035 \cs_generate_variant:Nn \clist_set:Nn { NV , No , Nx , c , cV , co , cx }
\clist_set:co 7036 \cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }
\clist_set:cx

```

(End definition for `\clist_set:Nn` and `\clist_gset:Nn`. These functions are documented on page 124.)

`\clist_gset:Nn`

`\clist_put_left:Nn`

Comma lists cannot hold empty values: there are therefore a couple of sanity checks to avoid accumulating commas.

```

\clist_put_left:NV 7037 \cs_new_protected:Npn \clist_put_left:Nn
\clist_put_left:No 7038 { \__clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
\clist_put_left:Nn 7039 \cs_new_protected:Npn \clist_gput_left:Nn
\clist_put_left:cn 7040 { \__clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
\clist_put_left:cV 7041 \cs_new_protected:Npn \__clist_put_left:NNNn #1#2#3#4
\clist_put_left:co 7042 {
\clist_put_left:cx 7043   #2 \l__clist_internal_clist {#4}
7044   #1 #3 \l__clist_internal_clist #3
7045 }
7046 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
7047 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
7048 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }

```

`__clist_put_left:NNNn`

```
7049 \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 124.)

```
\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
```

```
7050 \cs_new_protected:Npn \clist_put_right:Nn
7051 { \__clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
7052 \cs_new_protected:Npn \clist_gput_right:Nn
7053 { \__clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
7054 \cs_new_protected:Npn \__clist_put_right:NNNn #1#2#3#4
7055 {
7056   #2 \l__clist_internal_clist {#4}
7057   #1 #3 #3 \l__clist_internal_clist
7058 }
7059 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
7060 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
7061 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
7062 \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 125.)

13.4 Comma lists as stacks

```
\clist_get:NN
```

```
\clist_get:cn
```

```
\__clist_get:wN
```

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma.

```
7063 \cs_new_protected:Npn \clist_get:NN #1#2
7064 {
7065   \if_meaning:w #1 \c_empty_clist
7066     \tl_set:Nn #2 { \q_no_value }
7067   \else:
7068     \exp_after:wN \__clist_get:wN #1 , \q_stop #2
7069   \fi:
7070 }
7071 \cs_new_protected:Npn \__clist_get:wN #1 , #2 \q_stop #3
7072 { \tl_set:Nn #3 {#1} }
7073 \cs_generate_variant:Nn \clist_get:NN { c }
```

(End definition for `\clist_get:NN` and `__clist_get:wN`. These functions are documented on page 129.)

```
\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 `\q_mark`, unless the original clist contained exactly one item: then the argument is just `\q_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.

```
7074 \cs_new_protected:Npn \clist_pop:NN
7075 { \__clist_pop:NNN \tl_set:Nx }
7076 \cs_new_protected:Npn \clist_gpop:NN
7077 { \__clist_pop:NNN \tl_gset:Nx }
7078 \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
7079 {
7080   \if_meaning:w #2 \c_empty_clist
7081     \tl_set:Nn #3 { \q_no_value }
```

```

7082     \else:
7083         \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
7084     \fi:
7085 }
7086 \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \q_stop #3#4#5
7087 {
7088     \tl_set:Nn #5 {#1}
7089     #3 #4
7090     {
7091         \__clist_pop:wN \prg_do_nothing:
7092         #2 \exp_not:o
7093         , \q_mark \use_none:n
7094         \q_stop
7095     }
7096 }
7097 \cs_new:Npn \__clist_pop:wN #1 , \q_mark #2 #3 \q_stop { #2 {#1} }
7098 \cs_generate_variant:Nn \clist_pop:NN { c }
7099 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End definition for \clist_pop:NN and others. These functions are documented on page 129.)

```

\clist_get:NNTF The same, as branching code: very similar to the above.
\clist_get:cNTF
\clist_pop:NNTF
\clist_pop:cNTF
\clist_gpop:NNTF
\clist_gpop:cNTF
\__clist_pop_TF:NNN
7100 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
7101 {
7102     \if_meaning:w #1 \c_empty_clist
7103     \prg_return_false:
7104     \else:
7105         \exp_after:wN \__clist_get:wN #1 , \q_stop #2
7106         \prg_return_true:
7107     \fi:
7108 }
7109 \cs_generate_variant:Nn \clist_get:NNT { c }
7110 \cs_generate_variant:Nn \clist_get:NNF { c }
7111 \cs_generate_variant:Nn \clist_get:NNTF { c }
7112 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
7113 { \__clist_pop_TF:NNN \tl_set:Nx #1 #2 }
7114 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
7115 { \__clist_pop_TF:NNN \tl_gset:Nx #1 #2 }
7116 \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
7117 {
7118     \if_meaning:w #2 \c_empty_clist
7119     \prg_return_false:
7120     \else:
7121         \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
7122         \prg_return_true:
7123     \fi:
7124 }
7125 \cs_generate_variant:Nn \clist_pop:NNT { c }
7126 \cs_generate_variant:Nn \clist_pop:NNF { c }
7127 \cs_generate_variant:Nn \clist_pop:NNTF { c }
7128 \cs_generate_variant:Nn \clist_gpop:NNT { c }
7129 \cs_generate_variant:Nn \clist_gpop:NNF { c }
7130 \cs_generate_variant:Nn \clist_gpop:NNTF { c }

```

(End definition for \clist_get:NNTF and others. These functions are documented on page 129.)

\clist_push:Nn Pushing to a comma list is the same as adding on the left.

\clist_push:Nv	7131	\cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
\clist_push:No	7132	\cs_new_eq:NN \clist_push:Nv \clist_put_left:Nv
\clist_push:Nx	7133	\cs_new_eq:NN \clist_push:No \clist_put_left:No
\clist_push:cn	7134	\cs_new_eq:NN \clist_push:Nx \clist_put_left:Nx
\clist_push:cV	7135	\cs_new_eq:NN \clist_push:cn \clist_put_left:cn
\clist_push:co	7136	\cs_new_eq:NN \clist_push:cV \clist_put_left:cV
\clist_push:cx	7137	\cs_new_eq:NN \clist_push:co \clist_put_left:co
\clist_gpush:Nn	7138	\cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
\clist_gpush:Nv	7139	\cs_new_eq:NN \clist_gpush:Nv \clist_gput_left:Nv
\clist_gpush:No	7140	\cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
\clist_gpush:Nx	7141	\cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
\clist_gpush:cn	7142	\cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
\clist_gpush:cV	7143	\cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
\clist_gpush:co	7144	\cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
\clist_gpush:cx	7145	\cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx

(End definition for `\clist_push:Nn` and `\clist_gpush:Nn`. These functions are documented on page 130.)

13.5 Modifying comma lists

`\l__clist_internal_remove_clist` An internal comma list for the removal routines.

7147 **\clist_new:N \l__clist_internal_remove_clist**

(End definition for `\l__clist_internal_remove_clist`.)

\clist_remove_duplicates:N Removing duplicates means making a new list then copying it.

\clist_remove_duplicates:c	7148	\cs_new_protected:Npn \clist_remove_duplicates:N
\clist_gremove_duplicates:N	7149	{ __clist_remove_duplicates:NN \clist_set_eq:NN }
\clist_gremove_duplicates:c	7150	\cs_new_protected:Npn \clist_gremove_duplicates:N
__clist_remove_duplicates:NN	7151	{ __clist_remove_duplicates:NN \clist_gset_eq:NN }
	7152	\cs_new_protected:Npn __clist_remove_duplicates:NN #1#2
	7153	{
	7154	\clist_clear:N \l__clist_internal_remove_clist
	7155	\clist_map_inline:Nn #2
	7156	{
	7157	\clist_if_in:NnF \l__clist_internal_remove_clist {##1}
	7158	{ \clist_put_right:Nn \l__clist_internal_remove_clist {##1} }
	7159	}
	7160	#1 #2 \l__clist_internal_remove_clist
	7161	}
	7162	\cs_generate_variant:Nn \clist_remove_duplicates:N { c }
	7163	\cs_generate_variant:Nn \clist_gremove_duplicates:N { c }

(End definition for `\clist_remove_duplicates:N`, `\clist_gremove_duplicates:N`, and `__clist_remove_duplicates:NN`. These functions are documented on page 125.)

\clist_remove_all:Nn The method used here is very similar to `\tl_replace_all:Nnn`. 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

\clist_remove_all:cn	
\clist_gremove_all:Nn	
\clist_gremove_all:cn	
__clist_remove_all:NNn	
__clist_remove_all:w	
__clist_remove_all:	

found, the `\q_mark` delimiter used is the one inserted by `__clist_tmp:w`, and `\use_none_delimit_by_q_stop:w` is deleted. At the end, the final *<item>* is grabbed, and the argument of `__clist_tmp:w` contains `\q_mark`: in that case, `__clist_remove_all:w` removes the second `\q_mark` (inserted by `__clist_tmp:w`), and lets `\use_none_delimit_by_q_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.

```

7164 \cs_new_protected:Npn \clist_remove_all:Nn
7165   { \__clist_remove_all:NNn \tl_set:Nx }
7166 \cs_new_protected:Npn \clist_gremove_all:Nn
7167   { \__clist_remove_all:NNn \tl_gset:Nx }
7168 \cs_new_protected:Npn \__clist_remove_all:NNn #1#2#3
7169   {
7170     \cs_set:Npn \__clist_tmp:w ##1 , #3 ,
7171     {
7172       ##1
7173       , \q_mark , \use_none_delimit_by_q_stop:w ,
7174       \__clist_remove_all:
7175     }
7176     #1 #2
7177     {
7178       \exp_after:wN \__clist_remove_all:
7179       #2 , \q_mark , #3 , \q_stop
7180     }
7181     \clist_if_empty:NF #2
7182     {
7183       #1 #2
7184       {
7185         \exp_args:No \exp_not:o
7186         { \exp_after:wN \use_none:n #2 }
7187       }
7188     }
7189   }
7190 \cs_new:Npn \__clist_remove_all:
7191   { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
7192 \cs_new:Npn \__clist_remove_all:w #1 , \q_mark , #2 , { \exp_not:n {#1} }
7193 \cs_generate_variant:Nn \clist_remove_all:Nn { c }
7194 \cs_generate_variant:Nn \clist_gremove_all:Nn { c }

```

(End definition for `\clist_remove_all:Nn` and others. These functions are documented on page 125.)

`\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_reverse:c`

`\clist_greverse:N`

`\clist_greverse:c`

```

7195 \cs_new_protected:Npn \clist_reverse:N #1
7196   { \tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
7197 \cs_new_protected:Npn \clist_greverse:N #1
7198   { \tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
7199 \cs_generate_variant:Nn \clist_reverse:N { c }
7200 \cs_generate_variant:Nn \clist_greverse:N { c }

```


(End definition for `\clist_reverse:N` and `\clist_greverse:N`. These functions are documented on page 125.)

`\clist_reverse:n` The reversed token list is built one item at a time, and stored between `\q_stop` and `\q_mark`, in the form of ? followed by zero or more instances of “ $\langle item \rangle$,”. We start from a comma list “ $\langle item_1 \rangle, \dots, \langle item_n \rangle$ ”. During the loop, the auxiliary `__clist_reverse:wwNww` receives “? $\langle item_i \rangle$ ” as #1, “ $\langle item_{i+1} \rangle, \dots, \langle item_n \rangle$ ” as #2, `__clist_reverse:wwNww` as #3, what remains until `\q_stop` as #4, and “ $\langle item_{i-1} \rangle, \dots, \langle item_1 \rangle$,” 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 “`\q_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 `\q_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`.

```

7201 \cs_new:Npn \clist_reverse:n #1
7202 {
7203   \__clist_reverse:wwNww ? #1 ,
7204   \q_mark \__clist_reverse:wwNww ! ,
7205   \q_mark \__clist_reverse_end:ww
7206   \q_stop ? \q_mark
7207 }
7208 \cs_new:Npn \__clist_reverse:wwNww
7209   #1 , #2 \q_mark #3 #4 \q_stop ? #5 \q_mark
7210   { #3 ? #2 \q_mark #3 #4 \q_stop #1 , #5 \q_mark }
7211 \cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \q_mark
7212   { \exp_not:o { \use_none:n #2 } }

```

(End definition for `\clist_reverse:n`, `__clist_reverse:wwNww`, and `__clist_reverse_end:ww`. These functions are documented on page 125.)

`\clist_sort:Nn` Implemented in `l3sort`.

`\clist_sort:cn`

`\clist_gsort:Nn` (End definition for `\clist_sort:Nn` and `\clist_gsort:Nn`. These functions are documented on page 126.)

`\clist_gsort:cn`

13.6 Comma list conditionals

`\clist_if_empty_p:N` Simple copies from the token list variable material.

```

7213 \prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N
7214   { p , T , F , TF }
7215 \prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c
7216   { p , T , F , TF }

```

(End definition for `\clist_if_empty:N`. This function is documented on page 126.)

`\clist_if_empty_p:n` 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 will grab `\prg_return_false:` as #2, unless every item in the comma list was blank and the loop actually got broken by the trailing `\q_mark \prg_return_false:` item.

`\clist_if_empty:nTF`
`__clist_if_empty_n:w`
`__clist_if_empty_n:wNw`

```

7217 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
7218 {
7219     \__clist_if_empty_n:w ? #1
7220     , \q_mark \prg_return_false:
7221     , \q_mark \prg_return_true:
7222     \q_stop
7223 }
7224 \cs_new:Npn \__clist_if_empty_n:w #1 ,
7225 {
7226     \tl_if_empty:oTF { \use_none:nn #1 ? }
7227     { \__clist_if_empty_n:w ? }
7228     { \__clist_if_empty_n:wNw }
7229 }
7230 \cs_new:Npn \__clist_if_empty_n:wNw #1 \q_mark #2#3 \q_stop {#2}

```

(End definition for `\clist_if_empty:nTF`, `__clist_if_empty_n:w`, and `__clist_if_empty_n:wNw`. These functions are documented on page 126.)

```

\clist_if_in:NnTF See description of the \tl_if_in:Nn function for details. We simply surround the comma
\clist_if_in:NVTF list, and the item, with commas.
\clist_if_in:NoTF
\clist_if_in:cnTF
\clist_if_in:cVTF
\clist_if_in:coTF
\clist_if_in:nnTF
\clist_if_in:nVTF
\clist_if_in:noTF
\__clist_if_in_return:nn
7231 \prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }
7232 {
7233     \exp_args:No \__clist_if_in_return:nn #1 {#2}
7234 }
7235 \prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }
7236 {
7237     \clist_set:Nn \l__clist_internal_clist {#1}
7238     \exp_args:No \__clist_if_in_return:nn \l__clist_internal_clist {#2}
7239 }
7240 \cs_new_protected:Npn \__clist_if_in_return:nn #1#2
7241 {
7242     \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
7243     \tl_if_empty:oTF
7244     { \__clist_tmp:w ,#1, {} {} ,#2, }
7245     { \prg_return_false: } { \prg_return_true: }
7246 }
7247 \cs_generate_variant:Nn \clist_if_in:NnT { NV , No }
7248 \cs_generate_variant:Nn \clist_if_in:NnT { c , cV , co }
7249 \cs_generate_variant:Nn \clist_if_in:NnF { NV , No }
7250 \cs_generate_variant:Nn \clist_if_in:NnF { c , cV , co }
7251 \cs_generate_variant:Nn \clist_if_in:NnTF { NV , No }
7252 \cs_generate_variant:Nn \clist_if_in:NnTF { c , cV , co }
7253 \cs_generate_variant:Nn \clist_if_in:nnT { nV , no }
7254 \cs_generate_variant:Nn \clist_if_in:nnF { nV , no }
7255 \cs_generate_variant:Nn \clist_if_in:nnTF { nV , no }

```

(End definition for `\clist_if_in:NnTF`, `\clist_if_in:nnTF`, and `__clist_if_in_return:nn`. These functions are documented on page 126.)

13.7 Mapping to comma lists

```

\clist_map_function:NN If the variable is empty, the mapping is skipped (otherwise, that comma-list would be
\clist_map_function:cN seen as consisting of one empty item). Then loop over the comma-list, grabbing one
\__clist_map_function:Nw

```

comma-delimited item at a time. The end is marked by `\q_recursion_tail`. The auxiliary function `__clist_map_function:Nw` is used directly in `\clist_map_inline:Nn`. Change with care.

```

7256 \cs_new:Npn \clist_map_function:NN #1#2
7257 {
7258   \clist_if_empty:NF #1
7259   {
7260     \exp_last_unbraced:NNo \__clist_map_function:Nw #2 #1
7261     , \q_recursion_tail ,
7262     \__prg_break_point:Nn \clist_map_break: { }
7263   }
7264 }
7265 \cs_new:Npn \__clist_map_function:Nw #1#2 ,
7266 {
7267   \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
7268   #1 {#2}
7269   \__clist_map_function:Nw #1
7270 }
7271 \cs_generate_variant:Nn \clist_map_function:NN { c }

```

(End definition for `\clist_map_function:NN` and `__clist_map_function:Nw`. These functions are documented on page 127.)

`\clist_map_function:nN` The `n`-type mapping function is a bit more awkward, since spaces must be trimmed from each item. Space trimming is again based on `__clist_trim_spaces_generic:nw`. The auxiliary `__clist_map_function_n:Nn` receives as arguments the function, and the result of removing leading and trailing spaces from the item which lies until the next comma. Empty items are ignored, then one level of braces is removed by `__clist_map_unbrace:Nw`.

```

7272 \cs_new:Npn \clist_map_function:nN #1#2
7273 {
7274   \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #2 }
7275   \q_mark #1, \q_recursion_tail,
7276   \__prg_break_point:Nn \clist_map_break: { }
7277 }
7278 \cs_new:Npn \__clist_map_function_n:Nn #1 #2
7279 {
7280   \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
7281   \tl_if_empty:nF {#2} { \__clist_map_unbrace:Nw #1 #2, }
7282   \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #1 }
7283   \q_mark
7284 }
7285 \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`. These functions are documented on page 127.)

`\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.

```

7286 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
7287 {
7288   \clist_if_empty:NF #1
7289   {
7290     \int_gincr:N \g__prg_map_int
7291     \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
7292     \exp_last_unbraced:Nco \__clist_map_function:Nw
7293     { __prg_map_ \int_use:N \g__prg_map_int :w }
7294     #1 , \q_recursion_tail ,
7295     \__prg_break_point:Nn \clist_map_break:
7296     { \int_gdecr:N \g__prg_map_int }
7297   }
7298 }
7299 \cs_new_protected:Npn \clist_map_inline:nn #1
7300 {
7301   \clist_set:Nn \l__clist_internal_clist {#1}
7302   \clist_map_inline:Nn \l__clist_internal_clist
7303 }
7304 \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 127.)

`\clist_map_variable:NNn` As for other comma-list mappings, filter out the case of an empty list. Same approach
`\clist_map_variable:cNn` as `\clist_map_function:Nn`, additionally we store each item in the given variable. As
`\clist_map_variable:nNn` for inline mappings, space trimming for the `n` variant is done by storing the comma list
`__clist_map_variable:Nnw` in a variable.

```

7305 \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
7306 {
7307   \clist_if_empty:NF #1
7308   {
7309     \exp_args:Nno \use:nn
7310     { \__clist_map_variable:Nnw #2 {#3} }
7311     #1
7312     , \q_recursion_tail , \q_recursion_stop
7313     \__prg_break_point:Nn \clist_map_break: { }
7314   }
7315 }
7316 \cs_new_protected:Npn \clist_map_variable:nNn #1
7317 {
7318   \clist_set:Nn \l__clist_internal_clist {#1}
7319   \clist_map_variable:NNn \l__clist_internal_clist
7320 }
7321 \cs_new_protected:Npn \__clist_map_variable:Nnw #1#2#3,
7322 {
7323   \tl_set:Nn #1 {#3}
7324   \quark_if_recursion_tail_stop:N #1
7325   \use:n {#2}
7326   \__clist_map_variable:Nnw #1 {#2}
7327 }
7328 \cs_generate_variant:Nn \clist_map_variable:NNn { c }

```

(End definition for `\clist_map_variable:NNn`, `\clist_map_variable:nNn`, and `__clist_map_variable:Nnw`. These functions are documented on page 127.)

`\clist_map_break:` The break statements use the general `__prg_map_break:Nn` mechanism.
`\clist_map_break:n`

```

7329 \cs_new:Npn \clist_map_break:
7330 { \__prg_map_break:Nn \clist_map_break: { } }
7331 \cs_new:Npn \clist_map_break:n
7332 { \__prg_map_break:Nn \clist_map_break: }

```

(End definition for `\clist_map_break:` and `\clist_map_break:n`. These functions are documented on page 127.)

`\clist_count:N` Counting the items in a comma list is done using the same approach as for other token
`\clist_count:c` count functions: turn each entry into a +1 then use integer evaluation to actually do the
`\clist_count:n` mathematics. In the case of an n-type comma-list, we could of course use `\clist_map_-`
`__clist_count:n` `function:nN`, but that is very slow, because it carefully removes spaces. Instead, we loop
`__clist_count:w` manually, and skip blank items (but not {}, hence the extra spaces).

```

7333 \cs_new:Npn \clist_count:N #1
7334 {
7335   \int_eval:n
7336   {
7337     0
7338     \clist_map_function:NN #1 \__clist_count:n
7339   }
7340 }
7341 \cs_generate_variant:Nn \clist_count:N { c }
7342 \cs_new:Npx \clist_count:n #1
7343 {
7344   \exp_not:N \int_eval:n
7345   {
7346     0
7347     \exp_not:N \__clist_count:w \c_space_tl
7348     #1 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
7349   }
7350 }
7351 \cs_new:Npn \__clist_count:n #1 { + \c_one }
7352 \cs_new:Npx \__clist_count:w #1 ,
7353 {
7354   \exp_not:n { \exp_args:Nf \quark_if_recursion_tail_stop:n } {#1}
7355   \exp_not:N \tl_if_blank:nF {#1} { + \c_one }
7356   \exp_not:N \__clist_count:w \c_space_tl
7357 }

```

(End definition for `\clist_count:N` and others. These functions are documented on page 128.)

13.8 Using comma lists

`\clist_use:Nnnn` First check that the variable exists. Then count the items in the comma list. If it has
`\clist_use:cnnn` none, output nothing. If it has one item, output that item, brace stripped (note that
`__clist_use:wwn` space-trimming has already been done when the comma list was assigned). If it has two,
`__clist_use:nwwwnwn` place the *<separator between two>* in the middle.

Otherwise, `__clist_use:nwwwnwn` takes the following arguments; 1: a *<separator>*,
`\clist_use:Nn` 2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6:
`\clist_use:cn` 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_stop`. The
<separator> and the first of the three items are placed in the result, then we use the

$\langle continuation \rangle$, placing the remaining two items after it. When we begin this loop, the three items really belong to the comma list, the first $\backslash q_mark$ 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, $\backslash q_mark$ is taken as a third item, and now the second $\backslash q_mark$ serves as a delimiter to use_ii , switching to the other $\langle continuation \rangle$, use_iii , which uses the $\langle separator\ between\ final\ two \rangle$.

```

7358 \cs_new:Npn \clist_use:Nnnn #1#2#3#4
7359 {
7360   \clist_if_exist:NTF #1
7361   {
7362     \int_case:nnF { \clist_count:N #1 }
7363     {
7364       { 0 } { }
7365       { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
7366       { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
7367     }
7368     {
7369       \exp_after:wN \__clist_use:nwwwwnwn
7370       \exp_after:wN { \exp_after:wN } #1 ,
7371       \q_mark , { \__clist_use:nwwwwnwn {#3} }
7372       \q_mark , { \__clist_use:nwnn {#4} }
7373       \q_stop { }
7374     }
7375   }
7376   {
7377     \__msg_kernel_expandable_error:nnn
7378     { kernel } { bad-variable } {#1}
7379   }
7380 }
7381 \cs_generate_variant:Nn \clist_use:Nnnn { c }
7382 \cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
7383 \cs_new:Npn \__clist_use:nwwwwnwn
7384   #1#2 , #3 , #4 , #5 \q_mark , #6#7 \q_stop #8
7385   { #6 {#3} , {#4} , #5 \q_mark , {#6} #7 \q_stop { #8 #1 #2 } }
7386 \cs_new:Npn \__clist_use:nwnn #1#2 , #3 \q_stop #4
7387   { \exp_not:n { #4 #1 #2 } }
7388 \cs_new:Npn \clist_use:Nn #1#2
7389   { \clist_use:Nnnn #1 {#2} {#2} {#2} }
7390 \cs_generate_variant:Nn \clist_use:Nn { c }

```

(End definition for $\backslash\text{clist_use:Nnnn}$ and others. These functions are documented on page 128.)

13.9 Using a single item

$\backslash\text{clist_item:Nn}$ $\backslash\text{clist_item:cn}$ $\backslash\text{__clist_item:nnnN}$ $\backslash\text{__clist_item:ffoN}$ $\backslash\text{__clist_item:ffnN}$ $\backslash\text{__clist_item_N_loop:nw}$	<p>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.</p> <pre> 7391 \cs_new:Npn \clist_item:Nn #1#2 7392 { 7393 __clist_item:ffoN 7394 { \clist_count:N #1 } </pre>
---	---

```

7395     { \int_eval:n {#2} }
7396     #1
7397     \__clist_item_N_loop:nw
7398   }
7399   \cs_new:Npn \__clist_item:nnnN #1#2#3#4
7400   {
7401     \int_compare:nNnTF {#2} < \c_zero
7402     {
7403       \int_compare:nNnTF {#2} < { - #1 }
7404       { \use_none_delimit_by_q_stop:w }
7405       { \exp_args:Nf #4 { \int_eval:n { #2 + \c_one + #1 } } }
7406     }
7407     {
7408       \int_compare:nNnTF {#2} > {#1}
7409       { \use_none_delimit_by_q_stop:w }
7410       { #4 {#2} }
7411     }
7412     { } , #3 , \q_stop
7413   }
7414   \cs_generate_variant:Nn \__clist_item:nnnN { ffo, ff }
7415   \cs_new:Npn \__clist_item_N_loop:nw #1 #2,
7416   {
7417     \int_compare:nNnTF {#1} = \c_zero
7418     { \use_i_delimit_by_q_stop:nw { \exp_not:n {#2} } }
7419     { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
7420   }
7421   \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End definition for `\clist_item:Nn`, `__clist_item:nnnN`, and `__clist_item_N_loop:nw`. These functions are documented on page [130](#).)

`\clist_item:nn` 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.

```

\__clist_item_n:nw
\__clist_item_n_loop:nw
\__clist_item_n_end:n
\__clist_item_n_strip:w
7422   \cs_new:Npn \clist_item:nn #1#2
7423   {
7424     \__clist_item:ffnN
7425     { \clist_count:n {#1} }
7426     { \int_eval:n {#2} }
7427     {#1}
7428     \__clist_item_n:nw
7429   }
7430   \cs_new:Npn \__clist_item_n:nw #1
7431   { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
7432   \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
7433   {
7434     \exp_args:No \tl_if_blank:nTF {#2}
7435     { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
7436     {
7437       \int_compare:nNnTF {#1} = \c_zero
7438       { \exp_args:No \__clist_item_n_end:n {#2} }
7439       {
7440         \exp_args:Nf \__clist_item_n_loop:nw
7441         { \int_eval:n { #1 - 1 } }

```

```

7442         \prg_do_nothing:
7443     }
7444 }
7445 }
7446 \cs_new:Npn \__clist_item_n_end:n #1 #2 \q_stop
7447 {
7448     \__tl_trim_spaces:nn { \q_mark #1 }
7449     { \exp_last_unbraced:No \__clist_item_n_strip:w } ,
7450 }
7451 \cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }

```

(End definition for `\clist_item:nn` and others. These functions are documented on page 130.)

13.10 Viewing comma lists

`\clist_show:N` Apply the general `__msg_show_variable:NNNnn`. In the case of an n-type comma-list, we must do things by hand, using the same message `show-clist` as for an N-type comma-list but with an empty name (first argument).

`\clist_show:c`

`\clist_show:n`

```

7452 \cs_new_protected:Npn \clist_show:N #1
7453 {
7454     \__msg_show_variable:NNNnn #1
7455     \clist_if_exist:NTF \clist_if_empty:NTF { clist }
7456     { \clist_map_function:NN #1 \__msg_show_item:n }
7457 }
7458 \cs_new_protected:Npn \clist_show:n #1
7459 {
7460     \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-clist }
7461     { } { \clist_if_empty:nF {#1} { ? } } { } { }
7462     \__msg_show_wrap:n
7463     { \clist_map_function:nN {#1} \__msg_show_item:n }
7464 }
7465 \cs_generate_variant:Nn \clist_show:N { c }

```

(End definition for `\clist_show:N` and `\clist_show:n`. These functions are documented on page 130.)

13.11 Scratch comma lists

`\l_tmpa_clist` Temporary comma list variables.

`\l_tmpb_clist`

`\g_tmpa_clist`

`\g_tmpb_clist`

```

7466 \clist_new:N \l_tmpa_clist
7467 \clist_new:N \l_tmpb_clist
7468 \clist_new:N \g_tmpa_clist
7469 \clist_new:N \g_tmpb_clist

```

(End definition for `\l_tmpa_clist` and others. These variables are documented on page 131.)

7470 \langle /initex | package \rangle

14 l3prop implementation

The following test files are used for this code: `m3prop001`, `m3prop002`, `m3prop003`, `m3prop004`, `m3show001`.

7471 \langle *initex | package \rangle

7472 \langle @@=prop \rangle

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 A private scan mark is used as a marker after each key, and at the very beginning of the property list.

```
7473 \__scan_new:N \s__prop
```

(End definition for `\s__prop`.)

__prop_pair:wn The delimiter is always defined, but when misused simply triggers an error and removes its argument.

```
7474 \cs_new:Npn \__prop_pair:wn #1 \s__prop #2
7475 { \__msg_kernel_expandable_error:nn { kernel } { misused-prop } }
```

(End definition for `__prop_pair:wn`.)

\l__prop_internal_tl Token list used to store the new key–value pair inserted by `\prop_put:Nnn` and friends.

```
7476 \tl_new:N \l__prop_internal_tl
```

(End definition for `\l__prop_internal_tl`.)

\c_empty_prop An empty prop.

```
7477 \tl_const:Nn \c_empty_prop { \s__prop }
```

(End definition for `\c_empty_prop`. This variable is documented on page 137.)

14.1 Allocation and initialisation

\prop_new:N Property lists are initialized with the value `\c_empty_prop`.

```
\prop_new:c 7478 \cs_new_protected:Npn \prop_new:N #1
7479 {
7480   \__chk_if_free_cs:N #1
7481   \cs_gset_eq:NN #1 \c_empty_prop
7482 }
7483 \cs_generate_variant:Nn \prop_new:N { c }
```

(End definition for `\prop_new:N`. This function is documented on page 132.)

\prop_clear:N The same idea for clearing.

```
\prop_clear:c 7484 \cs_new_protected:Npn \prop_clear:N #1
\prop_gclear:N 7485 { \prop_set_eq:NN #1 \c_empty_prop }
\prop_gclear:c 7486 \cs_generate_variant:Nn \prop_clear:N { c }
7487 \cs_new_protected:Npn \prop_gclear:N #1
7488 { \prop_gset_eq:NN #1 \c_empty_prop }
7489 \cs_generate_variant:Nn \prop_gclear:N { c }
```

(End definition for `\prop_clear:N` and `\prop_gclear:N`. These functions are documented on page 132.)

```

\prop_clear_new:N Once again a simple variation of the token list functions.
\prop_clear_new:c 7490 \cs_new_protected:Npn \prop_clear_new:N #1
\prop_gclear_new:N 7491 { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\prop_gclear_new:c 7492 \cs_generate_variant:Nn \prop_clear_new:N { c }
7493 \cs_new_protected:Npn \prop_gclear_new:N #1
7494 { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
7495 \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 132.)

```

\prop_set_eq:NN These are simply copies from the token list functions.
\prop_set_eq:cN 7496 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
\prop_set_eq:Nc 7497 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
\prop_set_eq:cc 7498 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
\prop_gset_eq:NN 7499 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
\prop_gset_eq:cN 7500 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
\prop_gset_eq:Nc 7501 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
\prop_gset_eq:cN 7502 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
7503 \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 132.)

```

\l_tmpa_prop We can now initialize the scratch variables.
\l_tmpb_prop 7504 \prop_new:N \l_tmpa_prop
\g_tmpa_prop 7505 \prop_new:N \l_tmpb_prop
\g_tmpb_prop 7506 \prop_new:N \g_tmpa_prop
7507 \prop_new:N \g_tmpb_prop

```

(End definition for `\l_tmpa_prop` and others. These variables are documented on page 136.)

14.2 Accessing data in property lists

```

\__prop_split:NnTF This function is used by most of the module, and hence must be fast. It receives a
\__prop_split_aux:NnTF <property list>, a <key>, a <true code> and a <>false code>. The aim is to split the <property
\__prop_split_aux:w list> at the given <key> into the <extract1> before the key–value pair, the <value> associated
with the <key> and the <extract2> after the key–value pair. This is done using a delimited
function, whose definition is as follows, where the <key> is turned into a string.

```

```

\cs_set:Npn \__prop_split_aux:w #1
\__prop_pair:wn <key> \s__prop #2
#3 \q_mark #4 #5 \q_stop
{ #4 {<true code>} {<>false code>}}

```

If the `<key>` is present in the property list, `__prop_split_aux:w`'s #1 is the part before the `<key>`, #2 is the `<value>`, #3 is the part after the `<key>`, #4 is `\use_i:nn`, and #5 is additional tokens that we do not care about. The `<true code>` 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 `<key>` is not there, then the `<function>` is `\use_ii:nn`, which keeps the `<>false code>`.

```

7508 \cs_new_protected:Npn \__prop_split:NnTF #1#2

```

```

7509 { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
7510 \cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
7511 {
7512   \cs_set:Npn \__prop_split_aux:w ##1
7513     \__prop_pair:wn #2 \s__prop ##2 ##3 \q_mark ##4 ##5 \q_stop
7514     { ##4 {#3} {#4} }
7515   \exp_after:wN \__prop_split_aux:w #1 \q_mark \use_i:nn
7516     \__prop_pair:wn #2 \s__prop { } \q_mark \use_ii:nn \q_stop
7517 }
7518 \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` 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.

```

\prop_remove:NV
\prop_remove:cn
\prop_remove:cV
\prop_gremove:Nn
\prop_gremove:NV
\prop_gremove:cn
\prop_gremove:cV
7519 \cs_new_protected:Npn \prop_remove:Nn #1#2
7520 {
7521   \__prop_split:NnTF #1 {#2}
7522   { \tl_set:Nn #1 { ##1 ##3 } }
7523   { }
7524 }
7525 \cs_new_protected:Npn \prop_gremove:Nn #1#2
7526 {
7527   \__prop_split:NnTF #1 {#2}
7528   { \tl_gset:Nn #1 { ##1 ##3 } }
7529   { }
7530 }
7531 \cs_generate_variant:Nn \prop_remove:Nn { NV }
7532 \cs_generate_variant:Nn \prop_remove:Nn { c , cV }
7533 \cs_generate_variant:Nn \prop_gremove:Nn { NV }
7534 \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 134.)

`\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:cnN
\prop_get:cVN
\prop_get:coN
7535 \cs_new_protected:Npn \prop_get:NnN #1#2#3
7536 {
7537   \__prop_split:NnTF #1 {#2}
7538   { \tl_set:Nn #3 {##2} }
7539   { \tl_set:Nn #3 { \q_no_value } }
7540 }
7541 \cs_generate_variant:Nn \prop_get:NnN { NV , No }
7542 \cs_generate_variant:Nn \prop_get:NnN { c , cV , co }

```

(End definition for `\prop_get:NnN`. This function is documented on page 133.)

`\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:cnN
\prop_pop:coN
\prop_gpop:NnN
\prop_gpop:NoN
\prop_gpop:cnN
\prop_gpop:coN
7543 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
7544 {
7545   \__prop_split:NnTF #1 {#2}

```

```

7546     {
7547         \tl_set:Nn #3 {##2}
7548         \tl_set:Nn #1 { ##1 ##3 }
7549     }
7550     { \tl_set:Nn #3 { \q_no_value } }
7551 }
7552 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
7553 {
7554     \__prop_split:NnTF #1 {#2}
7555     {
7556         \tl_set:Nn #3 {##2}
7557         \tl_gset:Nn #1 { ##1 ##3 }
7558     }
7559     { \tl_set:Nn #3 { \q_no_value } }
7560 }
7561 \cs_generate_variant:Nn \prop_pop:NnN { No }
7562 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
7563 \cs_generate_variant:Nn \prop_gpop:NnN { No }
7564 \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 133.)

`\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.

```

7565 \cs_new:Npn \prop_item:Nn #1#2
7566 {
7567     \exp_last_unbraced:Noo \__prop_item_Nn:nwn { \tl_to_str:n {#2} } #1
7568     \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
7569     \__prg_break_point:
7570 }
7571 \cs_new:Npn \__prop_item_Nn:nwn #1#2 \__prop_pair:wn #3 \s__prop #4
7572 {
7573     \str_if_eq_x:nnTF {#1} {#3}
7574     { \__prg_break:n { \exp_not:n {#4} } }
7575     { \__prop_item_Nn:nwn {#1} }
7576 }
7577 \cs_generate_variant:Nn \prop_item:Nn { c }

```

(End definition for `\prop_item:Nn` and `__prop_item_Nn:nwn`. These functions are documented on page 134.)

`\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_pop:cnNTF
\prop_gpop:NnNTF
\prop_gpop:cnNTF
7578 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
7579 {
7580     \__prop_split:NnTF #1 {#2}
7581     {
7582         \tl_set:Nn #3 {##2}

```

```

7583         \tl_set:Nn #1 { ##1 ##3 }
7584         \prg_return_true:
7585     }
7586     { \prg_return_false: }
7587 }
7588 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
7589 {
7590     \__prop_split:NnTF #1 {#2}
7591     {
7592         \tl_set:Nn #3 {##2}
7593         \tl_gset:Nn #1 { ##1 ##3 }
7594         \prg_return_true:
7595     }
7596     { \prg_return_false: }
7597 }
7598 \cs_generate_variant:Nn \prop_pop:NnNT { c }
7599 \cs_generate_variant:Nn \prop_pop:NnNF { c }
7600 \cs_generate_variant:Nn \prop_pop:NnNTF { c }
7601 \cs_generate_variant:Nn \prop_gpop:NnNT { c }
7602 \cs_generate_variant:Nn \prop_gpop:NnNF { c }
7603 \cs_generate_variant:Nn \prop_gpop:NnNTF { c }

```

(End definition for `\prop_pop:NnNTF` and `\prop_gpop:NnNTF`. These functions are documented on page 135.)

\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.

```

7604 \cs_new_protected:Npn \prop_put:Nnn { \__prop_put:NNnn \tl_set:Nx }
7605 \cs_new_protected:Npn \prop_gput:Nnn { \__prop_put:NNnn \tl_gset:Nx }
7606 \cs_new_protected:Npn \__prop_put:NNnn #1#2#3#4
7607 {
7608     \tl_set:Nn \l__prop_internal_tl
7609     {
7610         \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
7611         \s__prop { \exp_not:n {#4} }
7612     }
7613     \__prop_split:NnTF #2 {#3}
7614     { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
7615     { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
7616 }
7617 \cs_generate_variant:Nn \prop_put:Nnn
7618 { NnV , Nno , Nnx , NV , NVV , No , Noo }
7619 \cs_generate_variant:Nn \prop_gput:Nnn
7620 { c , cnV , cno , cnx , cV , cVV , co , coo }
7621 \cs_generate_variant:Nn \prop_gput:Nnn
7622 { NnV , Nno , Nnx , NV , NVV , No , Noo }
7623 \cs_generate_variant:Nn \prop_gput:Nnn
7624 { c , cnV , cno , cnx , cV , cVV , co , coo }

```

\prop_gput:Nnn

\prop_gput:NnV

\prop_gput:Nno

\prop_gput:Nnx

\prop_gput:NVn

\prop_gput:NVV

\prop_gput:Non

\prop_gput:Noo

\prop_gput:cnn

\prop_gput:cnV

\prop_gput:cno

\prop_gput:cnx

\prop_gput:cVn

\prop_gput:cVV

\prop_gput:con

\prop_gput:coo

__prop_put:NNnn

(End definition for `\prop_put:Nnn`, `\prop_gput:Nnn`, and `__prop_put:NNnn`. These functions are documented on page 133.)

```

\prop_put_if_new:Nnn Adding conditionally also splits. If the key is already present, the three brace groups
\prop_put_if_new:cnn given by \__prop_split:NnTF are removed. If the key is new, then the value is added,
\prop_gput_if_new:Nnn being careful to convert the key to a string using \tl_to_str:n.
\prop_gput_if_new:cnn
\__prop_put_if_new:NNnn
7625 \cs_new_protected:Npn \prop_put_if_new:Nnn
7626 { \__prop_put_if_new:NNnn \tl_set:Nx }
7627 \cs_new_protected:Npn \prop_gput_if_new:Nnn
7628 { \__prop_put_if_new:NNnn \tl_gset:Nx }
7629 \cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
7630 {
7631   \tl_set:Nn \l__prop_internal_tl
7632   {
7633     \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
7634     \s__prop \exp_not:n { {#4} }
7635   }
7636   \__prop_split:NnTF #2 {#3}
7637   { }
7638   { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
7639 }
7640 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
7641 \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 133.)

14.3 Property list conditionals

```

\prop_if_exist_p:N Copies of the cs functions defined in l3basics.
\prop_if_exist_p:c
\prop_if_exist:NTF
7642 \prg_new_eq_conditional:NNn \prop_if_exist:N \cs_if_exist:N
7643 { TF , T , F , p }
\prop_if_exist:cTF
7644 \prg_new_eq_conditional:NNn \prop_if_exist:c \cs_if_exist:c
7645 { TF , T , F , p }

```

(End definition for `\prop_if_exist:NTF`. This function is documented on page 134.)

```

\prop_if_empty_p:N Same test as for token lists.
\prop_if_empty_p:c
\prop_if_empty:NTF
7646 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
7647 {
7648   \tl_if_eq:NNTF #1 \c_empty_prop
7649   \prg_return_true: \prg_return_false:
7650 }
7651 \cs_generate_variant:Nn \prop_if_empty_p:N { c }
7652 \cs_generate_variant:Nn \prop_if_empty:N { NT }
7653 \cs_generate_variant:Nn \prop_if_empty:N { NF }
7654 \cs_generate_variant:Nn \prop_if_empty:N { NTF }

```

(End definition for `\prop_if_empty:NTF`. This function is documented on page 134.)

```

\prop_if_in_p:Nn Testing expandably if a key is in a property list requires to go through the key–value
\prop_if_in_p:Nv pairs one by one. This is rather slow, and a faster test would be
\prop_if_in_p:No
\prop_if_in_p:cn
\prop_if_in_p:cV
\prop_if_in_p:co
\prop_if_in:NnTF
\prop_if_in:NvTF
\prop_if_in:NoTF
\prop_if_in:cnTF
\prop_if_in:cVTF
\prop_if_in:coTF
\__prop_if_in:nwnn
\__prop_if_in:N

```

```

\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_x:nn`, 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_x:nn`. It cannot make the breaking mechanism choke, because the arbitrary token list material is enclosed in braces. The second argument of `__prop_if_in:nwn` is most often empty. When the *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_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.

```

7655 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
7656 {
7657   \exp_last_unbraced:Noo \__prop_if_in:nwn { \tl_to_str:n {#2} } #1
7658   \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
7659   \q_recursion_tail
7660   \__prg_break_point:
7661 }
7662 \cs_new:Npn \__prop_if_in:nwn #1#2 \__prop_pair:wn #3 \s__prop #4
7663 {
7664   \str_if_eq_x:nnTF {#1} {#3}
7665   { \__prop_if_in:N }
7666   { \__prop_if_in:nwn {#1} }
7667 }
7668 \cs_new:Npn \__prop_if_in:N #1
7669 {
7670   \if_meaning:w \q_recursion_tail #1
7671   \prg_return_false:
7672   \else:
7673     \prg_return_true:
7674   \fi:
7675   \__prg_break:
7676 }
7677 \cs_generate_variant:Nn \prop_if_in_p:Nn { NV , No }
7678 \cs_generate_variant:Nn \prop_if_in_p:Nn { c , cV , co }
7679 \cs_generate_variant:Nn \prop_if_in:NnT { NV , No }
7680 \cs_generate_variant:Nn \prop_if_in:NnT { c , cV , co }
7681 \cs_generate_variant:Nn \prop_if_in:NnF { NV , No }
7682 \cs_generate_variant:Nn \prop_if_in:NnF { c , cV , co }
7683 \cs_generate_variant:Nn \prop_if_in:NnTF { NV , No }
7684 \cs_generate_variant:Nn \prop_if_in:NnTF { c , cV , co }

```

(End definition for `\prop_if_in:NnTF`, `__prop_if_in:nwn`, and `__prop_if_in:N`. These functions are documented on page 134.)

14.4 Recovering values from property lists with branching

`\prop_get:NnTF` 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.

```

\prop_get:NnTF
\prop_get:NVNTF
\prop_get:NoNTF
\prop_get:cnNTF
\prop_get:cVNTF
\prop_get:coNTF
7685 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
7686 {
7687   \__prop_split:NnTF #1 {#2}
7688   {
7689     \tl_set:Nn #3 {##2}
7690     \prg_return_true:
7691   }
7692   { \prg_return_false: }
7693 }
7694 \cs_generate_variant:Nn \prop_get:NnNT { NV , No }
7695 \cs_generate_variant:Nn \prop_get:NnNF { NV , No }
7696 \cs_generate_variant:Nn \prop_get:NnNTF { NV , No }
7697 \cs_generate_variant:Nn \prop_get:NnNT { c , cV , co }
7698 \cs_generate_variant:Nn \prop_get:NnNF { c , cV , co }
7699 \cs_generate_variant:Nn \prop_get:NnNTF { c , cV , co }

```

(End definition for `\prop_get:NnNTF`. This function is documented on page 135.)

14.5 Mapping to property lists

`\prop_map_function:NN` The fastest way to do a recursion here is to use an `\if_meaning:w` test: the keys are strings, and thus cannot match the marker `\q_recursion_tail`. A special case to note is when the key #3 is empty: then `\q_recursion_tail` is compared to `\exp_after:wN`, also different. Note that #2 is empty, except at the first iteration, where it is `\s__prop`.

```

\__prop_map_function:Nwwn
7700 \cs_new:Npn \prop_map_function:NN #1#2
7701 {
7702   \exp_last_unbraced:NNo \__prop_map_function:Nwwn #2 #1
7703   \__prop_pair:wn \q_recursion_tail \s__prop { }
7704   \__prg_break_point:Nn \prop_map_break: { }
7705 }
7706 \cs_new:Npn \__prop_map_function:Nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
7707 {
7708   \if_meaning:w \q_recursion_tail #3
7709   \exp_after:wN \prop_map_break:
7710   \fi:
7711   #1 {#3} {#4}
7712   \__prop_map_function:Nwwn #1
7713 }
7714 \cs_generate_variant:Nn \prop_map_function:NN { Nc }
7715 \cs_generate_variant:Nn \prop_map_function:NN { c , cc }

```

(End definition for `\prop_map_function:NN` and `__prop_map_function:Nwwn`. These functions are documented on page 135.)

`\prop_map_inline:Nn` 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.


```

7716 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
7717 {
7718   \cs_gset_eq:cN
7719     { __prg_map_ \int_use:N \g__prg_map_int :wn } \__prop_pair:wn
7720   \int_gincr:N \g__prg_map_int
7721   \cs_gset:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
7722   #1
7723   \__prg_break_point:Nn \prop_map_break:
7724   {
7725     \int_gdecr:N \g__prg_map_int
7726     \cs_gset_eq:Nc \__prop_pair:wn
7727       { __prg_map_ \int_use:N \g__prg_map_int :wn }
7728   }
7729 }
7730 \cs_generate_variant:Nn \prop_map_inline:Nn { c }

```

(End definition for `\prop_map_inline:Nn`. This function is documented on page 135.)

`\prop_map_break:` The break statements are based on the general `__prg_map_break:Nn`.
`\prop_map_break:n`

```

7731 \cs_new:Npn \prop_map_break:
7732 { \__prg_map_break:Nn \prop_map_break: { } }
7733 \cs_new:Npn \prop_map_break:n
7734 { \__prg_map_break:Nn \prop_map_break: }

```

(End definition for `\prop_map_break:` and `\prop_map_break:n`. These functions are documented on page 136.)

14.6 Viewing property lists

`\prop_show:N` Apply the general `__msg_show_variable:NNNnn`. Contrarily to sequences and comma
`\prop_show:c` lists, we use `__msg_show_item:nn` to format both the key and the value for each pair.

```

7735 \cs_new_protected:Npn \prop_show:N #1
7736 {
7737   \__msg_show_variable:NNNnn #1
7738   \prop_if_exist:NTF \prop_if_empty:NTF { prop }
7739   { \prop_map_function:NN #1 \__msg_show_item:nn }
7740 }
7741 \cs_generate_variant:Nn \prop_show:N { c }

```

(End definition for `\prop_show:N`. This function is documented on page 136.)

7742 `\initex | package`

15 l3box implementation

7743 `*initex | package`

7744 `\@@=box`

The code in this module is very straight forward so I'm not going to comment it very extensively.

15.1 Creating and initialising boxes

The following test files are used for this code: `m3box001.lvt`.

\box_new:N Defining a new $\langle box \rangle$ register: remember that box 255 is not generally available.

```
\box_new:c 7745 \*package>
7746 \cs_new_protected:Npn \box_new:N #1
7747 {
7748   \__chk_if_free_cs:N #1
7749   \cs:w newbox \cs_end: #1
7750 }
7751 \</package>
7752 \cs_generate_variant:Nn \box_new:N { c }
```

Clear a $\langle box \rangle$ register.

```
7753 \cs_new_protected:Npn \box_clear:N #1
\box_clear:N 7754 { \box_set_eq:NN #1 \c_empty_box }
7755 \cs_new_protected:Npn \box_gclear:N #1
\box_gclear:N 7756 { \box_gset_eq:NN #1 \c_empty_box }
7757 \cs_generate_variant:Nn \box_clear:N { c }
7758 \cs_generate_variant:Nn \box_gclear:N { c }
```

Clear or new.

```
7759 \cs_new_protected:Npn \box_clear_new:N #1
\box_clear_new:N 7760 { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
7761 \cs_new_protected:Npn \box_gclear_new:N #1
\box_gclear_new:N 7762 { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
7763 \cs_generate_variant:Nn \box_clear_new:N { c }
7764 \cs_generate_variant:Nn \box_gclear_new:N { c }
```

Assigning the contents of a box to be another box.

```
7765 \cs_new_protected:Npn \box_set_eq:NN #1#2
\box_set_eq:cN 7766 { \tex_setbox:D #1 \tex_copy:D #2 }
7767 \cs_new_protected:Npn \box_gset_eq:NN
\box_set_eq:Nc 7768 { \tex_global:D \box_set_eq:NN }
7769 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
\box_set_eq:cc 7770 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
```

Assigning the contents of a box to be another box. This clears the second box globally (that's how T_EX does it).

```
7771 \cs_new_protected:Npn \box_set_eq_clear:NN #1#2
\box_set_eq_clear:cN 7772 { \tex_setbox:D #1 \tex_box:D #2 }
7773 \cs_new_protected:Npn \box_gset_eq_clear:NN
\box_set_eq_clear:Nc 7774 { \tex_global:D \box_set_eq_clear:NN }
7775 \cs_generate_variant:Nn \box_set_eq_clear:NN { c , Nc , cc }
\box_set_eq_clear:cc 7776 \cs_generate_variant:Nn \box_gset_eq_clear:NN { c , Nc , cc }
7777 \cs_generate_variant:Nn \box_gset_eq_clear:NN { c , Nc , cc }
```

Copies of the cs functions defined in l3basics.

```
7777 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
\box_if_exist_p:N 7778 { TF , T , F , p }
7779 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
\box_if_exist:NTF 7780 { TF , T , F , p }
\box_if_exist:cTF
```

15.2 Measuring and setting box dimensions

Accessing the height, depth, and width of a $\langle box \rangle$ register.

```

7781 \cs_new_eq:NN \box_ht:N \tex_ht:D
\box_ht:Nn 7782 \cs_new_eq:NN \box_dp:N \tex_dp:D
\box_ht:c 7783 \cs_new_eq:NN \box_wd:N \tex_wd:D
\box_dp:N 7784 \cs_generate_variant:Nn \box_ht:N { c }
\box_dp:c 7785 \cs_generate_variant:Nn \box_dp:N { c }
\box_wd:N 7786 \cs_generate_variant:Nn \box_wd:N { c }
\box_wd:c

```

Measuring is easy: all primitive work. These primitives are not expandable, so the derived functions are not either.

```

\box_set_ht:Nn 7787 \cs_new_protected:Npn \box_set_dp:Nn #1#2
\box_set_ht:cn 7788 { \box_dp:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\box_set_dp:Nn 7789 \cs_new_protected:Npn \box_set_ht:Nn #1#2
\box_set_dp:cn 7790 { \box_ht:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\box_set_wd:Nn 7791 \cs_new_protected:Npn \box_set_wd:Nn #1#2
\box_set_wd:cn 7792 { \box_wd:N #1 \__dim_eval:w #2 \__dim_eval_end: }
7793 \cs_generate_variant:Nn \box_set_ht:Nn { c }
7794 \cs_generate_variant:Nn \box_set_dp:Nn { c }
7795 \cs_generate_variant:Nn \box_set_wd:Nn { c }

```

15.3 Using boxes

Using a $\langle box \rangle$. These are just \TeX primitives with meaningful names.

```

7796 \cs_new_eq:NN \box_use_clear:N \tex_box:D
\box_use_clear:Nn 7797 \cs_new_eq:NN \box_use:N \tex_copy:D
\box_use_clear:c 7798 \cs_generate_variant:Nn \box_use_clear:N { c }
\box_use:N 7799 \cs_generate_variant:Nn \box_use:N { c }
\box_use:c

```

Move box material in different directions.

```

7800 \cs_new_protected:Npn \box_move_left:nn #1#2
\box_move_left:nn 7801 { \tex_moveleft:D \__dim_eval:w #1 \__dim_eval_end: #2 }
\box_move_right:nn 7802 \cs_new_protected:Npn \box_move_right:nn #1#2
\box_move_right:nn 7803 { \tex_moveright:D \__dim_eval:w #1 \__dim_eval_end: #2 }
\box_move_up:nn 7804 \cs_new_protected:Npn \box_move_up:nn #1#2
\box_move_down:nn 7805 { \tex_raise:D \__dim_eval:w #1 \__dim_eval_end: #2 }
7806 \cs_new_protected:Npn \box_move_down:nn #1#2
7807 { \tex_lower:D \__dim_eval:w #1 \__dim_eval_end: #2 }

```

15.4 Box conditionals

The primitives for testing if a $\langle box \rangle$ is empty/void or which type of box it is.

```

7808 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
\if_hbox:Nn 7809 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
\if_vbox:Nn 7810 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D
\if_box_empty:N

7811 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
\box_if_horizontal_p:N 7812 { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_horizontal_p:c 7813 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
\box_if_horizontal:N $\overline{TF}$  7814 { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_horizontal:c $\overline{TF}$  7815 \cs_generate_variant:Nn \box_if_horizontal_p:N { c }
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_vertical:N $\overline{TF}$ 
\box_if_vertical:c $\overline{TF}$ 

```

```

7816 \cs_generate_variant:Nn \box_if_horizontal:NT { c }
7817 \cs_generate_variant:Nn \box_if_horizontal:NF { c }
7818 \cs_generate_variant:Nn \box_if_horizontal:NTF { c }
7819 \cs_generate_variant:Nn \box_if_vertical_p:N { c }
7820 \cs_generate_variant:Nn \box_if_vertical:NT { c }
7821 \cs_generate_variant:Nn \box_if_vertical:NF { c }
7822 \cs_generate_variant:Nn \box_if_vertical:NTF { c }

```

Testing if a $\langle box \rangle$ is empty/void.

```

\box_if_empty_p:N 7823 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
\box_if_empty_p:c 7824 { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_empty:NTF 7825 \cs_generate_variant:Nn \box_if_empty_p:N { c }
\box_if_empty:cTF 7826 \cs_generate_variant:Nn \box_if_empty:NT { c }
7827 \cs_generate_variant:Nn \box_if_empty:NF { c }
7828 \cs_generate_variant:Nn \box_if_empty:NTF { c }

```

(End definition for $\backslash box_new:N$ and others. These functions are documented on page 138.)

15.5 The last box inserted

```

\box_set_to_last:N Set a box to the previous box.
\box_set_to_last:c 7829 \cs_new_protected:Npn \box_set_to_last:N #1
\box_gset_to_last:N 7830 { \tex_setbox:D #1 \tex_lastbox:D }
\box_gset_to_last:c 7831 \cs_new_protected:Npn \box_gset_to_last:N
7832 { \tex_global:D \box_set_to_last:N }
7833 \cs_generate_variant:Nn \box_set_to_last:N { c }
7834 \cs_generate_variant:Nn \box_gset_to_last:N { c }

```

(End definition for $\backslash box_set_to_last:N$ and $\backslash box_gset_to_last:N$. These functions are documented on page 140.)

15.6 Constant boxes

```

\c_empty_box A box we never use.
7835 \box_new:N \c_empty_box

```

(End definition for $\backslash c_empty_box$. This variable is documented on page 141.)

15.7 Scratch boxes

```

\l_tmpa_box Scratch boxes.
\l_tmpb_box 7836 \box_new:N \l_tmpa_box
\g_tmpa_box 7837 \box_new:N \l_tmpb_box
\g_tmpb_box 7838 \box_new:N \g_tmpa_box
7839 \box_new:N \g_tmpb_box

```

(End definition for $\backslash l_tmpa_box$ and others. These variables are documented on page 141.)

15.8 Viewing box contents

TeX's `\showbox` is not really that helpful in many cases, and it is also inconsistent with other L^AT_EX3 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.
\box_show:c    7840 \cs_new_protected:Npn \box_show:N #1
\box_show:Nnn  7841 { \box_show:Nnn #1 \c_max_int \c_max_int }
\box_show:cnn  7842 \cs_generate_variant:Nn \box_show:N { c }
                7843 \cs_new_protected:Npn \box_show:Nnn
                7844 { \__box_show:NNnn \c_one }
                7845 \cs_generate_variant:Nn \box_show:Nnn { c }
```

(End definition for `\box_show:N` and `\box_show:Nnn`. These functions are documented on page 141.)

```
\box_log:N    Getting TeX to write to the log without interruption the run is done by altering the
\box_log:c    interaction mode. For that, the  $\epsilon$ -TeX extensions are needed.
\box_log:Nnn  7846 \cs_new_protected:Npn \box_log:N #1
\box_log:cnn  7847 { \box_log:Nnn #1 \c_max_int \c_max_int }
                7848 \cs_generate_variant:Nn \box_log:N { c }
                7849 \cs_new_protected:Npn \box_log:Nnn #1#2#3
                7850 {
                7851   \use:x
                7852   {
                7853     \etex_interactionmode:D \c_zero
                7854     \__box_show:NNnn \c_zero \exp_not:N #1
                7855     { \int_eval:n {#2} } { \int_eval:n {#3} }
                7856     \etex_interactionmode:D
                7857     = \tex_the:D \etex_interactionmode:D \scan_stop:
                7858   }
                7859 }
                7860 \cs_generate_variant:Nn \box_log:Nnn { c }
```

(End definition for `\box_log:N` and `\box_log:Nnn`. These functions are documented on page 141.)

```
\__box_show:NNnn
```

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.

```
7861 \cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
7862 {
7863   \group_begin:
7864   \int_set:Nn \tex_showboxbreadth:D {#3}
7865   \int_set:Nn \tex_showboxdepth:D {#4}
7866   \int_set_eq:NN \tex_tracingonline:D #1
7867   \int_set:Nn \tex_errorcontextlines:D { - \c_one }
7868   \box_if_exist:NTF #2
7869   { \tex_showbox:D \use:n {#2} }
7870   {
7871     \__msg_kernel_error:nxx { kernel } { variable-not-defined }
7872     { \token_to_str:N #2 }
7873   }
7874   \group_end:
7875 }
```

(End definition for `__box_show:NNnn`.)

15.9 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.

```
7876 \cs_new_protected:Npn \hbox:n #1 { \tex_hbox:D \scan_stop: {#1} }
```

(End definition for `\hbox:n`. This function is documented on page 141.)

\hbox_set:Nn

\hbox_set:cn

\hbox_gset:Nn

\hbox_gset:cn

```
7877 \cs_new_protected:Npn \hbox_set:Nn #1#2
```

```
7878 { \tex_setbox:D #1 \tex_hbox:D {#2} }
```

```
7879 \cs_new_protected:Npn \hbox_gset:Nn { \tex_global:D \hbox_set:Nn }
```

```
7880 \cs_generate_variant:Nn \hbox_set:Nn { c }
```

```
7881 \cs_generate_variant:Nn \hbox_gset:Nn { c }
```

(End definition for `\hbox_set:Nn` and `\hbox_gset:Nn`. These functions are documented on page 142.)

\hbox_set_to_wd:Nnn

\hbox_set_to_wd:cnn

\hbox_gset_to_wd:Nnn

\hbox_gset_to_wd:cnn

Storing material in a horizontal box with a specified width.

```
7882 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
```

```
7883 { \tex_setbox:D #1 \tex_hbox:D to \__dim_eval:w #2 \__dim_eval_end: {#3} }
```

```
7884 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn
```

```
7885 { \tex_global:D \hbox_set_to_wd:Nnn }
```

```
7886 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
```

```
7887 \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 142.)

\hbox_set:Nw

\hbox_set:cw

\hbox_gset:Nw

\hbox_gset:cw

\hbox_set_end:

\hbox_gset_end:

Storing material in a horizontal box. This type is useful in environment definitions.

```
7888 \cs_new_protected:Npn \hbox_set:Nw #1
```

```
7889 { \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token }
```

```
7890 \cs_new_protected:Npn \hbox_gset:Nw
```

```
7891 { \tex_global:D \hbox_set:Nw }
```

```
7892 \cs_generate_variant:Nn \hbox_set:Nw { c }
```

```
7893 \cs_generate_variant:Nn \hbox_gset:Nw { c }
```

```
7894 \cs_new_eq:NN \hbox_set_end: \c_group_end_token
```

```
7895 \cs_new_eq:NN \hbox_gset_end: \c_group_end_token
```

(End definition for `\hbox_set:Nw` and others. These functions are documented on page 142.)

\hbox_to_wd:nn

\hbox_to_zero:n

Put a horizontal box directly into the input stream.

```
7896 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
```

```
7897 { \tex_hbox:D to \__dim_eval:w #1 \__dim_eval_end: {#2} }
```

```
7898 \cs_new_protected:Npn \hbox_to_zero:n #1 { \tex_hbox:D to \c_zero_dim {#1} }
```

(End definition for `\hbox_to_wd:nn` and `\hbox_to_zero:n`. These functions are documented on page 142.)

\hbox_overlap_left:n

\hbox_overlap_right: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.

```
7899 \cs_new_protected:Npn \hbox_overlap_left:n #1
```

```
7900 { \hbox_to_zero:n { \tex_hss:D #1 } }
```

```
7901 \cs_new_protected:Npn \hbox_overlap_right:n #1
```

```
7902 { \hbox_to_zero:n { #1 \tex_hss:D } }
```

(End definition for `\hbox_overlap_left:n` and `\hbox_overlap_right:n`. These functions are documented on page 142.)

\hbox_unpack:N Unpacking a box and if requested also clear it.
\hbox_unpack:c
\hbox_unpack_clear:N
\hbox_unpack_clear:c

```

7903 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
7904 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
7905 \cs_generate_variant:Nn \hbox_unpack:N { c }
7906 \cs_generate_variant:Nn \hbox_unpack_clear:N { c }

```

(End definition for `\hbox_unpack:N` and `\hbox_unpack_clear:N`. These functions are documented on page 142.)

15.10 Vertical mode boxes

T_EX 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.

\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.

```

7907 \cs_new_protected:Npn \vbox:n #1 { \tex_vbox:D { #1 \par } }
7908 \cs_new_protected:Npn \vbox_top:n #1 { \tex_vtop:D { #1 \par } }

```

(End definition for `\vbox:n` and `\vbox_top:n`. These functions are documented on page 143.)

\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

```

7909 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
7910 { \tex_vbox:D to \__dim_eval:w #1 \__dim_eval_end: { #2 \par } }
7911 \cs_new_protected:Npn \vbox_to_zero:n #1
7912 { \tex_vbox:D to \c_zero_dim { #1 \par } }

```

(End definition for `\vbox_to_ht:nn` and others. These functions are documented on page 143.)

\vbox_set:Nn Storing material in a vertical box with a natural height.
\vbox_set:cn
\vbox_gset:Nn
\vbox_gset:cn

```

7913 \cs_new_protected:Npn \vbox_set:Nn #1#2
7914 { \tex_setbox:D #1 \tex_vbox:D { #2 \par } }
7915 \cs_new_protected:Npn \vbox_gset:Nn { \tex_global:D \vbox_set:Nn }
7916 \cs_generate_variant:Nn \vbox_set:Nn { c }
7917 \cs_generate_variant:Nn \vbox_gset:Nn { c }

```

(End definition for `\vbox_set:Nn` and `\vbox_gset:Nn`. These functions are documented on page 143.)

\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

```

7918 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
7919 { \tex_setbox:D #1 \tex_vtop:D { #2 \par } }
7920 \cs_new_protected:Npn \vbox_gset_top:Nn
7921 { \tex_global:D \vbox_set_top:Nn }
7922 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
7923 \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 143.)

\vbox_set_to_ht:Nnn Storing material in a vertical box with a specified height.

\vbox_set_to_ht:cnn 7924 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3

\vbox_gset_to_ht:Nnn 7925 {

\vbox_gset_to_ht:cnn 7926 \tex_setbox:D #1 \tex_vbox:D to _dim_eval:w #2 _dim_eval_end:

7927 { #3 \par }

7928 }

7929 \cs_new_protected:Npn \vbox_gset_to_ht:Nnn

7930 { \tex_global:D \vbox_set_to_ht:Nnn }

7931 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }

7932 \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 143.)

\vbox_set:Nw Storing material in a vertical box. This type is useful in environment definitions.

\vbox_set:cw 7933 \cs_new_protected:Npn \vbox_set:Nw #1

\vbox_gset:Nw 7934 { \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }

\vbox_gset:cw 7935 \cs_new_protected:Npn \vbox_gset:Nw

\vbox_set_end: 7936 { \tex_global:D \vbox_set:Nw }

\vbox_gset_end: 7937 \cs_generate_variant:Nn \vbox_set:Nw { c }

7938 \cs_generate_variant:Nn \vbox_gset:Nw { c }

7939 \cs_new_protected:Npn \vbox_set_end:

7940 {

7941 \par

7942 \c_group_end_token

7943 }

7944 \cs_new_eq:NN \vbox_gset_end: \vbox_set_end:

(End definition for \vbox_set:Nw and others. These functions are documented on page 144.)

\vbox_unpack:N Unpacking a box and if requested also clear it.

\vbox_unpack:c 7945 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D

\vbox_unpack_clear:N 7946 \cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D

\vbox_unpack_clear:c 7947 \cs_generate_variant:Nn \vbox_unpack:N { c }

7948 \cs_generate_variant:Nn \vbox_unpack_clear:N { c }

(End definition for \vbox_unpack:N and \vbox_unpack_clear:N. These functions are documented on page 144.)

\vbox_set_split_to_ht:NNn Splitting a vertical box in two.

7949 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3

7950 { \tex_setbox:D #1 \tex_vsplit:D #2 to _dim_eval:w #3 _dim_eval_end: }

(End definition for \vbox_set_split_to_ht:NNn. This function is documented on page 144.)

7951 </initex | package>

16 l3coffins Implementation

7952 <*initex | package>

7953 <@@=coffin>

16.1 Coffins: data structures and general variables

\l_coffin_internal_box Scratch variables.

\l_coffin_internal_dim

\l_coffin_internal_tl


```

7954 \box_new:N \l__coffin_internal_box
7955 \dim_new:N \l__coffin_internal_dim
7956 \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.

```

7957 \prop_new:N \c__coffin_corners_prop
7958 \prop_put:Nnn \c__coffin_corners_prop { tl } { { 0 pt } { 0 pt } }
7959 \prop_put:Nnn \c__coffin_corners_prop { tr } { { 0 pt } { 0 pt } }
7960 \prop_put:Nnn \c__coffin_corners_prop { bl } { { 0 pt } { 0 pt } }
7961 \prop_put:Nnn \c__coffin_corners_prop { br } { { 0 pt } { 0 pt } }

```

(End definition for `\c__coffin_corners_prop`.)

`\c__coffin_poles_prop` Pole positions are given for horizontal, vertical and reference-point based values.

```

7962 \prop_new:N \c__coffin_poles_prop
7963 \tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 0 pt } { 1000 pt } }
7964 \prop_put:Nno \c__coffin_poles_prop { l } { \l__coffin_internal_tl }
7965 \prop_put:Nno \c__coffin_poles_prop { hc } { \l__coffin_internal_tl }
7966 \prop_put:Nno \c__coffin_poles_prop { r } { \l__coffin_internal_tl }
7967 \tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 1000 pt } { 0 pt } }
7968 \prop_put:Nno \c__coffin_poles_prop { b } { \l__coffin_internal_tl }
7969 \prop_put:Nno \c__coffin_poles_prop { vc } { \l__coffin_internal_tl }
7970 \prop_put:Nno \c__coffin_poles_prop { t } { \l__coffin_internal_tl }
7971 \prop_put:Nno \c__coffin_poles_prop { B } { \l__coffin_internal_tl }
7972 \prop_put:Nno \c__coffin_poles_prop { H } { \l__coffin_internal_tl }
7973 \prop_put:Nno \c__coffin_poles_prop { T } { \l__coffin_internal_tl }

```

(End definition for `\c__coffin_poles_prop`.)

`\l__coffin_slope_x_fp` Used for calculations of intersections.

```

\l__coffin_slope_y_fp
7974 \fp_new:N \l__coffin_slope_x_fp
7975 \fp_new:N \l__coffin_slope_y_fp

```

(End definition for `\l__coffin_slope_x_fp` and `\l__coffin_slope_y_fp`.)

`\l__coffin_error_bool` For propagating errors so that parts of the code can work around them.

```

7976 \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.

```

7977 \dim_new:N \l__coffin_offset_x_dim
7978 \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_tl` Needed for finding the intersection of two poles.

```

\l__coffin_pole_b_tl
7979 \tl_new:N \l__coffin_pole_a_tl
7980 \tl_new:N \l__coffin_pole_b_tl

```

(End definition for `\l__coffin_pole_a_tl` and `\l__coffin_pole_b_tl`.)

```

\l__coffin_x_dim For calculating intersections and so forth.
\l__coffin_y_dim
\l__coffin_x_prime_dim
\l__coffin_y_prime_dim

```

(End definition for \l__coffin_x_dim and others.)

16.2 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_if_exist_p:N Several of the higher-level coffin functions will 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.

```

\coffin_if_exist:NTF
\coffin_if_exist:cTF

```

```

7985 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
7986 {
7987   \cs_if_exist:NTF #1
7988   {
7989     \cs_if_exist:cTF { l__coffin_poles_ __int_value:w #1 _prop }
7990     { \prg_return_true: }
7991     { \prg_return_false: }
7992   }
7993   { \prg_return_false: }
7994 }
7995 \cs_generate_variant:Nn \coffin_if_exist_p:N { c }
7996 \cs_generate_variant:Nn \coffin_if_exist:NTF { c }
7997 \cs_generate_variant:Nn \coffin_if_exist:NTF { c }
7998 \cs_generate_variant:Nn \coffin_if_exist:NTF { c }

```

(End definition for \coffin_if_exist:N~~TF~~. This function is documented on page 146.)

__coffin_if_exist:NT Several of the higher-level coffin functions will 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.

```

7999 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
8000 {
8001   \coffin_if_exist:NTF #1
8002   { #2 }
8003   {
8004     \__msg_kernel_error:nxx { kernel } { unknown-coffin }
8005     { \token_to_str:N #1 }
8006   }
8007 }

```

(End definition for __coffin_if_exist:NT.)

\coffin_clear:N Clearing coffins means emptying the box and resetting all of the structures.

```

\coffin_clear:c

```

```

8008 \cs_new_protected:Npn \coffin_clear:N #1
8009 {
8010   \__coffin_if_exist:NT #1
8011   {
8012     \box_clear:N #1

```

```

8013         \__coffin_reset_structure:N #1
8014     }
8015 }
8016 \cs_generate_variant:Nn \coffin_clear:N { c }

```

(End definition for `\coffin_clear:N`. This function is documented on page 146.)

`\coffin_new:N` Creating a new coffin means making the underlying box and adding the data structures.
`\coffin_new:c` These are created globally, as there is a need to avoid any strange effects if the coffin is created inside a group. This means that the usual rule about `\l...` variables has to be broken.

```

8017 \cs_new_protected:Npn \coffin_new:N #1
8018 {
8019     \box_new:N #1
8020     \__chk_suspend_log:
8021     \prop_clear_new:c { l__coffin_corners_ \__int_value:w #1 _prop }
8022     \prop_clear_new:c { l__coffin_poles_ \__int_value:w #1 _prop }
8023     \prop_gset_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
8024         \c__coffin_corners_prop
8025     \prop_gset_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }
8026         \c__coffin_poles_prop
8027     \__chk_resume_log:
8028 }
8029 \cs_generate_variant:Nn \coffin_new:N { c }

```

(End definition for `\coffin_new:N`. This function is documented on page 146.)

`\hcoffin_set:Nn` Horizontal coffins are relatively easy: set the appropriate box, reset the structures then
`\hcoffin_set:cn` update the handle positions.

```

8030 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
8031 {
8032     \__coffin_if_exist:NT #1
8033     {
8034         \hbox_set:Nn #1
8035         {
8036             \color_group_begin:
8037             \color_ensure_current:
8038             #2
8039             \color_group_end:
8040         }
8041         \__coffin_reset_structure:N #1
8042         \__coffin_update_poles:N #1
8043         \__coffin_update_corners:N #1
8044     }
8045 }
8046 \cs_generate_variant:Nn \hcoffin_set:Nn { c }

```

(End definition for `\hcoffin_set:Nn`. This function is documented on page 146.)

`\vcoffin_set:Nnn` Setting vertical coffins is more complex. First, the material is typeset with a given width.
`\vcoffin_set:cnn` 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 whatsit to the start of the vertical box and mess up the location of the T pole (see *TEX by Topic* for discussion of the `\vtop` primitive, used to do the measuring).

```

8047 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
8048 {
8049   \__coffin_if_exist:NT #1
8050   {
8051     \vbox_set:Nn #1
8052     {
8053       \dim_set:Nn \tex_hsize:D {#2}
8054     }
8055     \cs_set_protected:Npn \package {
8056       \dim_set_eq:NN \linewidth \tex_hsize:D
8057       \dim_set_eq:NN \columnwidth \tex_hsize:D
8058     }
8059     \color_group_begin:
8060     #3
8061     \color_group_end:
8062   }
8063   \__coffin_reset_structure:N #1
8064   \__coffin_update_poles:N #1
8065   \__coffin_update_corners:N #1
8066   \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
8067   \__coffin_set_pole:Nnx #1 { T }
8068   {
8069     { 0 pt }
8070     {
8071       \dim_eval:n
8072       { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
8073     }
8074     { 1000 pt }
8075     { 0 pt }
8076   }
8077   \box_clear:N \l__coffin_internal_box
8078 }
8079 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }

```

(End definition for `\vcoffin_set:Nnn`. This function is documented on page 147.)

`\hcoffin_set:Nw` These are the “begin”/“end” versions of the above: watch the grouping!

`\hcoffin_set:cw`

`\hcoffin_set_end:`

```

8080 \cs_new_protected:Npn \hcoffin_set:Nw #1
8081 {
8082   \__coffin_if_exist:NT #1
8083   {
8084     \hbox_set:Nw #1 \color_group_begin: \color_ensure_current:
8085     \cs_set_protected:Npn \hcoffin_set_end:
8086     {
8087       \color_group_end:
8088       \hbox_set_end:
8089       \__coffin_reset_structure:N #1
8090       \__coffin_update_poles:N #1
8091       \__coffin_update_corners:N #1
8092     }
8093   }
8094 }
8095 \cs_new_protected:Npn \hcoffin_set_end: { }
8096 \cs_generate_variant:Nn \hcoffin_set:Nw { c }

```

(End definition for `\hcoffin_set:Nw` and `\hcoffin_set_end:`. These functions are documented on page 146.)

`\vcoffin_set:Nnw` The same for vertical coffins.

```

8097 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
8098 {
8099   \__coffin_if_exist:NT #1
8100   {
8101     \vbox_set:Nw #1
8102     \dim_set:Nn \tex_hsize:D {#2}
8103     (*package)
8104       \dim_set_eq:NN \linewidth \tex_hsize:D
8105       \dim_set_eq:NN \columnwidth \tex_hsize:D
8106     (/package)
8107     \color_group_begin:
8108     \cs_set_protected:Npn \vcoffin_set_end:
8109       {
8110         \color_group_end:
8111         \vbox_set_end:
8112         \__coffin_reset_structure:N #1
8113         \__coffin_update_poles:N #1
8114         \__coffin_update_corners:N #1
8115         \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
8116         \__coffin_set_pole:Nnx #1 { T }
8117         {
8118           { 0 pt }
8119           {
8120             \dim_eval:n
8121               { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
8122           }
8123           { 1000 pt }
8124           { 0 pt }
8125         }
8126         \box_clear:N \l__coffin_internal_box
8127       }
8128     }
8129   }
8130   \cs_new_protected:Npn \vcoffin_set_end: { }
8131   \cs_generate_variant:Nn \vcoffin_set:Nnw { c }

```

(End definition for `\vcoffin_set:Nnw` and `\vcoffin_set_end:`. These functions are documented on page 147.)

`\coffin_set_eq:NN` Setting two coffins equal is just a wrapper around other functions.

```

8132 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
8133 {
8134   \__coffin_if_exist:NT #1
8135   {
8136     \box_set_eq:NN #1 #2
8137     \__coffin_set_eq_structure:NN #1 #2
8138   }
8139 }
8140 \cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }

```

(End definition for `\coffin_set_eq:NN`. This function is documented on page 146.)

\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.

```

8141 \coffin_new:N \c_empty_coffin
8142 \hbox_set:Nn \c_empty_coffin { }
8143 \coffin_new:N \l__coffin_aligned_coffin
8144 \coffin_new:N \l__coffin_aligned_internal_coffin

```

(End definition for `\c_empty_coffin`, `\l__coffin_aligned_coffin`, and `\l__coffin_aligned_internal_coffin`. These variables are documented on page 149.)

\l_tmpa_coffin The usual scratch space.

```

8145 \coffin_new:N \l_tmpa_coffin
8146 \coffin_new:N \l_tmpb_coffin

```

(End definition for `\l_tmpa_coffin` and `\l_tmpb_coffin`. These variables are documented on page 149.)

16.3 Measuring coffins

\coffin_dp:N Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

```

8147 \cs_new_eq:NN \coffin_dp:N \box_dp:N
8148 \cs_new_eq:NN \coffin_dp:c \box_dp:c
8149 \cs_new_eq:NN \coffin_ht:N \box_ht:N
8150 \cs_new_eq:NN \coffin_ht:c \box_ht:c
8151 \cs_new_eq:NN \coffin_wd:N \box_wd:N
8152 \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 148.)

16.4 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.

```

8153 \cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
8154 {
8155   \prop_get:cnNF
8156     { l__coffin_poles_ \__int_value:w #1 _prop } {#2} #3
8157   {
8158     \_msg_kernel_error:nxx { kernel } { unknown-coffin-pole }
8159     {#2} { \token_to_str:N #1 }
8160     \tl_set:Nn #3 { { 0 pt } { 0 pt } { 0 pt } { 0 pt } }
8161   }
8162 }

```

(End definition for `__coffin_get_pole:NnN`.)

__coffin_reset_structure:N Resetting the structure is a simple copy job.

```

8163 \cs_new_protected:Npn \__coffin_reset_structure:N #1
8164 {
8165   \prop_set_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
8166   \c__coffin_corners_prop
8167   \prop_set_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }

```

```

8168     \c__coffin_poles_prop
8169 }

```

(End definition for _coffin_reset_structure:N.)

_coffin_set_eq_structure:NN Setting coffin structures equal simply means copying the property list.

```

\_coffin_gset_eq_structure:NN
8170 \cs_new_protected:Npn \_coffin_set_eq_structure:NN #1#2
8171 {
8172     \prop_set_eq:cc { l__coffin_corners_ \_int_value:w #1 _prop }
8173     { l__coffin_corners_ \_int_value:w #2 _prop }
8174     \prop_set_eq:cc { l__coffin_poles_ \_int_value:w #1 _prop }
8175     { l__coffin_poles_ \_int_value:w #2 _prop }
8176 }
8177 \cs_new_protected:Npn \_coffin_gset_eq_structure:NN #1#2
8178 {
8179     \prop_gset_eq:cc { l__coffin_corners_ \_int_value:w #1 _prop }
8180     { l__coffin_corners_ \_int_value:w #2 _prop }
8181     \prop_gset_eq:cc { l__coffin_poles_ \_int_value:w #1 _prop }
8182     { l__coffin_poles_ \_int_value:w #2 _prop }
8183 }

```

(End definition for _coffin_set_eq_structure:NN and _coffin_gset_eq_structure:NN.)

\coffin_set_horizontal_pole:Nnn

\coffin_set_horizontal_pole:cmn

\coffin_set_vertical_pole:Nnn

\coffin_set_vertical_pole:cmn

_coffin_set_pole:Nnn

_coffin_set_pole:Nnx

Setting the pole of a coffin at the user/designer level requires a bit more care. The idea here is to provide a reasonable interface to the system, then to do the setting with full expansion. The three-argument version is used internally to do a direct setting.

```

8184 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
8185 {
8186     \_coffin_if_exist:NT #1
8187     {
8188         \_coffin_set_pole:Nnx #1 {#2}
8189         {
8190             { 0 pt } { \dim_eval:n {#3} }
8191             { 1000 pt } { 0 pt }
8192         }
8193     }
8194 }
8195 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
8196 {
8197     \_coffin_if_exist:NT #1
8198     {
8199         \_coffin_set_pole:Nnx #1 {#2}
8200         {
8201             { \dim_eval:n {#3} } { 0 pt }
8202             { 0 pt } { 1000 pt }
8203         }
8204     }
8205 }
8206 \cs_new_protected:Npn \_coffin_set_pole:Nnn #1#2#3
8207 { \prop_put:cmn { l__coffin_poles_ \_int_value:w #1 _prop } {#2} {#3} }
8208 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
8209 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
8210 \cs_generate_variant:Nn \_coffin_set_pole:Nnn { Nnx }

```

(End definition for `\coffin_set_horizontal_pole:Nnn`, `\coffin_set_vertical_pole:Nnn`, and `__coffin_set_pole:Nnn`. These functions are documented on page 147.)

`__coffin_update_corners:N` 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.

```

8211 \cs_new_protected:Npn \__coffin_update_corners:N #1
8212 {
8213   \prop_put:cnx { l__coffin_corners_ } \__int_value:w #1 _prop { tl }
8214   { { 0 pt } { \dim_eval:n { \box_ht:N #1 } } }
8215   \prop_put:cnx { l__coffin_corners_ } \__int_value:w #1 _prop { tr }
8216   { { \dim_eval:n { \box_wd:N #1 } } { \dim_eval:n { \box_ht:N #1 } } }
8217   \prop_put:cnx { l__coffin_corners_ } \__int_value:w #1 _prop { bl }
8218   { { 0 pt } { \dim_eval:n { - \box_dp:N #1 } } }
8219   \prop_put:cnx { l__coffin_corners_ } \__int_value:w #1 _prop { br }
8220   { { \dim_eval:n { \box_wd:N #1 } } { \dim_eval:n { -\box_dp:N #1 } } }
8221 }

```

(End definition for `__coffin_update_corners:N`.)

`__coffin_update_poles:N` 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.

```

8222 \cs_new_protected:Npn \__coffin_update_poles:N #1
8223 {
8224   \prop_put:cnx { l__coffin_poles_ } \__int_value:w #1 _prop { hc }
8225   {
8226     { \dim_eval:n { 0.5 \box_wd:N #1 } }
8227     { 0 pt } { 0 pt } { 1000 pt }
8228   }
8229   \prop_put:cnx { l__coffin_poles_ } \__int_value:w #1 _prop { r }
8230   {
8231     { \dim_eval:n { \box_wd:N #1 } }
8232     { 0 pt } { 0 pt } { 1000 pt }
8233   }
8234   \prop_put:cnx { l__coffin_poles_ } \__int_value:w #1 _prop { vc }
8235   {
8236     { 0 pt }
8237     { \dim_eval:n { ( \box_ht:N #1 - \box_dp:N #1 ) / 2 } }
8238     { 1000 pt }
8239     { 0 pt }
8240   }
8241   \prop_put:cnx { l__coffin_poles_ } \__int_value:w #1 _prop { t }
8242   {
8243     { 0 pt }
8244     { \dim_eval:n { \box_ht:N #1 } }
8245     { 1000 pt }
8246     { 0 pt }
8247   }
8248   \prop_put:cnx { l__coffin_poles_ } \__int_value:w #1 _prop { b }
8249   {
8250     { 0 pt }
8251     { \dim_eval:n { - \box_dp:N #1 } }
8252     { 1000 pt }

```



```

8253         { 0 pt }
8254     }
8255 }

```

(End definition for `_coffin_update_poles:N`.)

16.5 Coffins: calculation of pole intersections

`_coffin_calculate_intersection:Nnn` 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.

`_coffin_calculate_intersection:nnnnnnnn`
`_coffin_calculate_intersection_aux:nnnnnN`

```

8256 \cs_new_protected:Npn \_coffin_calculate_intersection:Nnn #1#2#3
8257 {
8258     \_coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
8259     \_coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
8260     \bool_set_false:N \l__coffin_error_bool
8261     \exp_last_two_unbraced:Noo
8262     \_coffin_calculate_intersection:nnnnnnnn
8263     \l__coffin_pole_a_tl \l__coffin_pole_b_tl
8264     \bool_if:NT \l__coffin_error_bool
8265     {
8266         \_msg_kernel_error:nn { kernel } { no-pole-intersection }
8267         \dim_zero:N \l__coffin_x_dim
8268         \dim_zero:N \l__coffin_y_dim
8269     }
8270 }

```

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' will be zero and a special case is needed.

```

8271 \cs_new_protected:Npn \_coffin_calculate_intersection:nnnnnnnn
8272     #1#2#3#4#5#6#7#8
8273 {
8274     \dim_compare:nNnTF {#3} = { \c_zero_dim }

```

The case where the first pole is vertical. So the x -component of the interaction will be at a . There is then a test on the second pole: if it is also vertical then there is an error.

```

8275     {
8276         \dim_set:Nn \l__coffin_x_dim {#1}
8277         \dim_compare:nNnTF {#7} = { \c_zero_dim
8278             { \bool_set_true:N \l__coffin_error_bool }

```

The second pole may still be horizontal, in which case the y -component of the intersection will be b' . If not,

$$y = \frac{d'}{c'} (x - a') + b'$$

with the x -component already known to be $\#1$. This calculation is done as a generalised auxiliary.

```

8279     {
8280         \dim_compare:nNnTF {#8} = { \c_zero_dim
8281             { \dim_set:Nn \l__coffin_y_dim {#6} }

```

```

8282         {
8283             \__coffin_calculate_intersection_aux:nnnnnN
8284             {#1} {#5} {#6} {#7} {#8} \l__coffin_y_dim
8285         }
8286     }
8287 }

```

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.

```

8288 {
8289     \dim_compare:nNnTF {#4} = \c_zero_dim
8290     {
8291         \dim_set:Nn \l__coffin_y_dim {#2}
8292         \dim_compare:nNnTF {#8} = { \c_zero_dim }
8293         { \bool_set_true:N \l__coffin_error_bool }
8294     }
8295     \dim_compare:nNnTF {#7} = \c_zero_dim
8296     { \dim_set:Nn \l__coffin_x_dim {#5} }

```

The formula for the case where the second pole is neither horizontal nor vertical is

$$x = \frac{c'}{d'}(y - b') + a'$$

which is again handled by the same auxiliary.

```

8297 {
8298     \__coffin_calculate_intersection_aux:nnnnnN
8299     {#2} {#6} {#5} {#8} {#7} \l__coffin_x_dim
8300 }
8301 }
8302 }

```

The first pole is neither horizontal nor vertical. This still leaves the second pole, which may be a special case. For those possibilities, the calculations are the same as above with the first and second poles interchanged.

```

8303 {
8304     \dim_compare:nNnTF {#7} = \c_zero_dim
8305     {
8306         \dim_set:Nn \l__coffin_x_dim {#5}
8307         \__coffin_calculate_intersection_aux:nnnnnN
8308         {#5} {#1} {#2} {#3} {#4} \l__coffin_y_dim
8309     }
8310     {
8311         \dim_compare:nNnTF {#8} = \c_zero_dim
8312         {
8313             \dim_set:Nn \l__coffin_y_dim {#6}
8314             \__coffin_calculate_intersection_aux:nnnnnN
8315             {#6} {#2} {#1} {#4} {#3} \l__coffin_x_dim
8316         }

```

If none of the special cases apply then there is still a need to check that there is a unique intersection between the two pole. This is the case if they have different slopes.

```

8317 {
8318     \fp_set:Nn \l__coffin_slope_x_fp
8319     { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
8320     \fp_set:Nn \l__coffin_slope_y_fp

```

```

8321         { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
8322     \fp_compare:nNnTF
8323     \l__coffin_slope_x_fp = \l__coffin_slope_y_fp
8324     { \bool_set_true:N \l__coffin_error_bool }

```

All of the tests pass, so there is the full complexity of the calculation:

$$x = \frac{a(d/c) - a'(d'/c') - b + b'}{(d/c) - (d'/c')}$$

and noting that the two ratios are already worked out from the test just performed. There is quite a bit of shuffling from dimensions to floating points in order to do the work. The y -values is then worked out using the standard auxiliary starting from the x -position.

```

8325         {
8326             \dim_set:Nn \l__coffin_x_dim
8327             {
8328                 \fp_to_dim:n
8329                 {
8330                     (
8331                         \dim_to_fp:n {#1} * \l__coffin_slope_x_fp
8332                     - ( \dim_to_fp:n {#5} * \l__coffin_slope_y_fp )
8333                     - \dim_to_fp:n {#2}
8334                     + \dim_to_fp:n {#6}
8335                     )
8336                     /
8337                     ( \l__coffin_slope_x_fp - \l__coffin_slope_y_fp )
8338                 }
8339             }
8340             \__coffin_calculate_intersection_aux:nnnnnN
8341             { \l__coffin_x_dim }
8342             {#5} {#6} {#8} {#7} \l__coffin_y_dim
8343         }
8344     }
8345 }
8346 }
8347 }
8348 }

```

The formula for finding the intersection point is in most cases the same. The formula here is

$$\#6 = \#4 \cdot \left(\frac{\#1 - \#2}{\#5} \right) \#3$$

Thus #4 and #5 should be the directions of the pole while #2 and #3 are co-ordinates.

```

8349 \cs_new_protected:Npn \__coffin_calculate_intersection_aux:nnnnnN
8350     #1#2#3#4#5#6
8351     {
8352         \dim_set:Nn #6
8353         {
8354             \fp_to_dim:n
8355             {
8356                 \dim_to_fp:n {#4} *
8357                 ( \dim_to_fp:n {#1} - \dim_to_fp:n {#2} ) /
8358                 \dim_to_fp:n {#5}

```

```

8359         + \dim_to_fp:n {#3}
8360     }
8361 }
8362 }

```

(End definition for `__coffin_calculate_intersection:Nnn`, `__coffin_calculate_intersection:nnnnnnnn`, and `__coffin_calculate_intersection_aux:nnnnnN`.)

16.6 Aligning and typesetting of coffins

`\coffin_join:NnnNnnnn`
`\coffin_join:cnnNnnnn`
`\coffin_join:Nnnncnnnn`
`\coffin_join:cnnncnnnn`

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 will have all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```

8363 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
8364 {
8365     \__coffin_align:NnnNnnnnN
8366     #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 will 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.

```

8367     \hbox_set:Nn \l__coffin_aligned_coffin
8368     {
8369         \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
8370         { \tex_kern:D -\l__coffin_offset_x_dim }
8371         \hbox_unpack:N \l__coffin_aligned_coffin
8372         \dim_set:Nn \l__coffin_internal_dim
8373         { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
8374         \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
8375         { \tex_kern:D -\l__coffin_internal_dim }
8376     }

```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

8377     \__coffin_reset_structure:N \l__coffin_aligned_coffin
8378     \prop_clear:c
8379     { \l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _ prop }
8380     \__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 will then depend on whether any shift was needed.

```

8381     \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
8382     {
8383         \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
8384         \__coffin_offset_poles:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
8385         \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
8386         \__coffin_offset_corners:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
8387     }
8388     {
8389         \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
8390         \__coffin_offset_poles:Nnn #4

```

```

8391         { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
8392         \__coffin_offset_corners:Nnn #1 { 0 pt } { 0 pt }
8393         \__coffin_offset_corners:Nnn #4
8394         { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
8395     }
8396     \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
8397     \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
8398 }
8399 \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }

```

(End definition for `\coffin_join:NnnNnnnn`. This function is documented on page 148.)

`\coffin_attach:NnnNnnnn`

`\coffin_attach:cnncNnnnn`

`\coffin_attach:NnncNnnnn`

`\coffin_attach:cnncNnnnn`

`\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.

```

8400 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
8401 {
8402     \__coffin_align:NnnNnnnnN
8403     #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
8404     \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
8405     \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
8406     \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
8407     \__coffin_reset_structure:N \l__coffin_aligned_coffin
8408     \prop_set_eq:cc
8409     { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
8410     { l__coffin_corners_ \__int_value:w #1 _prop }
8411     \__coffin_update_poles:N \l__coffin_aligned_coffin
8412     \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
8413     \__coffin_offset_poles:Nnn #4
8414     { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
8415     \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
8416     \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
8417 }
8418 \cs_new_protected:Npn \coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
8419 {
8420     \__coffin_align:NnnNnnnnN
8421     #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
8422     \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
8423     \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
8424     \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
8425     \box_set_eq:NN #1 \l__coffin_aligned_coffin
8426 }
8427 \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }

```

(End definition for `\coffin_attach:NnnNnnnn` and `\coffin_attach_mark:NnnNnnnn`. These functions are documented on page 147.)

`__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 will depend on how the bounding box is being handled.

```

8428 \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
8429 {
8430   \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
8431   \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
8432   \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
8433   \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
8434   \dim_set:Nn \l__coffin_offset_x_dim
8435     { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
8436   \dim_set:Nn \l__coffin_offset_y_dim
8437     { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
8438   \hbox_set:Nn \l__coffin_aligned_internal_coffin
8439     {
8440       \box_use:N #1
8441       \tex_kern:D -\box_wd:N #1
8442       \tex_kern:D \l__coffin_offset_x_dim
8443       \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
8444     }
8445   \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
8446 }

```

(End definition for __coffin_align:NnnNnnnnN.)

__coffin_offset_poles:Nnn
 __coffin_offset_pole:Nnnnnnn

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.

```

8447 \cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
8448 {
8449   \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
8450     { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
8451 }
8452 \cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
8453 {
8454   \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
8455   \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
8456   \tl_if_in:nnTF {#2} { - }
8457     { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
8458     { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
8459   \exp_last_unbraced:NNo \__coffin_set_pole:Nnx \l__coffin_aligned_coffin
8460     { \l__coffin_internal_tl }
8461   {
8462     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
8463     {#5} {#6}
8464   }
8465 }

```

(End definition for __coffin_offset_poles:Nnn and __coffin_offset_pole:Nnnnnnn.)

`__coffin_offset_corners:Nnn` 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.

`__coffin_offset_corner:Nnnnn`

```

8466 \cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
8467 {
8468   \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
8469   { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
8470 }
8471 \cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6
8472 {
8473   \prop_put:cnx
8474   { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
8475   { #1 - #2 }
8476   {
8477     { \dim_eval:n { #3 + #5 } }
8478     { \dim_eval:n { #4 + #6 } }
8479   }
8480 }

```

(End definition for `__coffin_offset_corners:Nnn` and `__coffin_offset_corner:Nnnnn`.)

`__coffin_update_vertical_poles:NNN` The T and B poles will 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.

`__coffin_update_T:nnnnnnnnN`

`__coffin_update_B:nnnnnnnnN`

```

8481 \cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
8482 {
8483   \__coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
8484   \__coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
8485   \exp_last_two_unbraced:Noo \__coffin_update_T:nnnnnnnnN
8486   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
8487   \__coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
8488   \__coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
8489   \exp_last_two_unbraced:Noo \__coffin_update_B:nnnnnnnnN
8490   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
8491 }
8492 \cs_new_protected:Npn \__coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
8493 {
8494   \dim_compare:nNnTF {#2} < {#6}
8495   {
8496     \__coffin_set_pole:Nnx #9 { T }
8497     { { 0 pt } {#6} { 1000 pt } { 0 pt } }
8498   }
8499   {
8500     \__coffin_set_pole:Nnx #9 { T }
8501     { { 0 pt } {#2} { 1000 pt } { 0 pt } }
8502   }
8503 }
8504 \cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
8505 {
8506   \dim_compare:nNnTF {#2} < {#6}
8507   {
8508     \__coffin_set_pole:Nnx #9 { B }
8509     { { 0 pt } {#2} { 1000 pt } { 0 pt } }
8510   }
8511   {

```

```

8512         \_coffin_set_pole:Nnx #9 { B }
8513         { { 0 pt } {#6} { 1000 pt } { 0 pt } }
8514     }
8515 }

```

(End definition for `_coffin_update_vertical_poles:NNN`, `_coffin_update_T:nnnnnnnnN`, and `_coffin_update_B:nnnnnnnnN`.)

\coffin_typeset:Nnnnn 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.

\coffin_typeset:cnnnn

```

8516 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
8517 {
8518     \hbox_unpack:N \c_empty_box
8519     \_coffin_align:NnnNnnnnN \c_empty_coffin { H } { 1 }
8520     #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
8521     \box_use:N \l__coffin_aligned_coffin
8522 }
8523 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }

```

(End definition for `\coffin_typeset:Nnnnn`. This function is documented on page 148.)

16.7 Coffin diagnostics

\l__coffin_display_coffin Used for printing coffins with data structures attached.

```

\l__coffin_display_coord_coffin 8524 \coffin_new:N \l__coffin_display_coffin
\l__coffin_display_pole_coffin 8525 \coffin_new:N \l__coffin_display_coord_coffin
8526 \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.

```

8527 \prop_new:N \l__coffin_display_handles_prop
8528 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
8529 { { b } { r } { -1 } { 1 } }
8530 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
8531 { { b } { hc } { 0 } { 1 } }
8532 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
8533 { { b } { l } { 1 } { 1 } }
8534 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
8535 { { vc } { r } { -1 } { 0 } }
8536 \prop_put:Nnn \l__coffin_display_handles_prop { vhc }
8537 { { vc } { hc } { 0 } { 0 } }
8538 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
8539 { { vc } { l } { 1 } { 0 } }
8540 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
8541 { { t } { r } { -1 } { -1 } }
8542 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
8543 { { t } { hc } { 0 } { -1 } }
8544 \prop_put:Nnn \l__coffin_display_handles_prop { br }
8545 { { t } { l } { 1 } { -1 } }
8546 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }

```



```

8547 { { t } { r } { -1 } { -1 } }
8548 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
8549 { { t } { hc } { 0 } { -1 } }
8550 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
8551 { { t } { l } { 1 } { -1 } }
8552 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
8553 { { vc } { r } { -1 } { 1 } }
8554 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
8555 { { vc } { hc } { 0 } { 1 } }
8556 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
8557 { { vc } { l } { 1 } { 1 } }
8558 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
8559 { { b } { r } { -1 } { -1 } }
8560 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
8561 { { b } { hc } { 0 } { -1 } }
8562 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
8563 { { b } { l } { 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.

```

8564 \dim_new:N \l__coffin_display_offset_dim
8565 \dim_set:Nn \l__coffin_display_offset_dim { 2 pt }

```

(End definition for \l__coffin_display_offset_dim.)

`\l__coffin_display_x_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.

```

8566 \dim_new:N \l__coffin_display_x_dim
8567 \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.

```

8568 \prop_new:N \l__coffin_display_poles_prop

```

(End definition for \l__coffin_display_poles_prop.)

`\l__coffin_display_font_tl` Stores the settings used to print coffin data: this keeps things flexible.

```

8569 \tl_new:N \l__coffin_display_font_tl
8570 <*initex>
8571 \tl_set:Nn \l__coffin_display_font_tl { } % TODO
8572 </initex>
8573 <*package>
8574 \tl_set:Nn \l__coffin_display_font_tl { \sfamily \tiny }
8575 </package>

```

(End definition for \l__coffin_display_font_tl.)

`\coffin_mark_handle:Nnnn` 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.

```

8576 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
8577 {
8578   \hcoffin_set:Nn \l__coffin_display_pole_coffin
8579   {
8580     (*initex)
8581     \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
8582   }
8583   (*package)
8584   \color {#4}
8585   \rule { 1 pt } { 1 pt }
8586 }
8587
8588 \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
8589 \l__coffin_display_pole_coffin { hc } { vc } { 0 pt } { 0 pt }
8590 \hcoffin_set:Nn \l__coffin_display_coord_coffin
8591 {
8592   (*initex)
8593   % TODO
8594 }
8595 (*package)
8596 \color {#4}
8597 }
8598 \l__coffin_display_font_tl
8599 ( \tl_to_str:n { #2 , #3 } )
8600 }
8601 \prop_get:NnN \l__coffin_display_handles_prop
8602 { #2 #3 } \l__coffin_internal_tl
8603 \quark_if_no_value:NTF \l__coffin_internal_tl
8604 {
8605   \prop_get:NnN \l__coffin_display_handles_prop
8606   { #3 #2 } \l__coffin_internal_tl
8607   \quark_if_no_value:NTF \l__coffin_internal_tl
8608   {
8609     \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
8610     \l__coffin_display_coord_coffin { l } { vc }
8611     { 1 pt } { 0 pt }
8612   }
8613   {
8614     \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
8615     \l__coffin_internal_tl #1 {#2} {#3}
8616   }
8617 }
8618 {
8619   \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
8620   \l__coffin_internal_tl #1 {#2} {#3}
8621 }
8622 }
8623 \cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
8624 {
8625   \coffin_attach_mark:NnnNnnnn #5 {#6} {#7}

```

```

8626 \l__coffin_display_coord_coffin {#1} {#2}
8627 { #3 \l__coffin_display_offset_dim }
8628 { #4 \l__coffin_display_offset_dim }
8629 }
8630 \cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }

```

(End definition for `\coffin_mark_handle:Nnnn` and `__coffin_mark_handle_aux:nnnnNnn`. These functions are documented on page 149.)

`\coffin_display_handles:Nn`
`\coffin_display_handles:cn`
`__coffin_display_handles_aux:nnnnnn`
`__coffin_display_handles_aux:nnnn`
`__coffin_display_attach:Nnnnn`

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.

```

8631 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
8632 {
8633   \hcoffin_set:Nn \l__coffin_display_pole_coffin
8634   {
8635     \*initex
8636     \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
8637     \*initex
8638     \*package
8639     \color {#2}
8640     \rule { 1 pt } { 1 pt }
8641   }
8642 }
8643 \prop_set_eq:Nc \l__coffin_display_poles_prop
8644 { l__coffin_poles_ \__int_value:w #1 _prop }
8645 \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
8646 \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
8647 \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
8648 { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
8649 \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
8650 \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
8651 { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
8652 \coffin_set_eq:NN \l__coffin_display_coffin #1
8653 \prop_map_inline:Nn \l__coffin_display_poles_prop
8654 {
8655   \prop_remove:Nn \l__coffin_display_poles_prop {##1}
8656   \__coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
8657 }
8658 \box_use:N \l__coffin_display_coffin
8659 }

```

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.

```

8660 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
8661 {
8662   \prop_map_inline:Nn \l__coffin_display_poles_prop
8663   {
8664     \bool_set_false:N \l__coffin_error_bool
8665     \__coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
8666     \bool_if:NF \l__coffin_error_bool
8667     {

```

```

8668 \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
8669 \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
8670 \__coffin_display_attach:Nnnnn
8671 \l__coffin_display_pole_coffin { hc } { vc }
8672 { 0 pt } { 0 pt }
8673 \hcoffin_set:Nn \l__coffin_display_coord_coffin
8674 {
8675 \*initex
8676 % TODO
8677 \*initex
8678 \*package
8679 \color {#6}
8680 \package
8681 \l__coffin_display_font_tl
8682 ( \tl_to_str:n { #1 , ##1 } )
8683 }
8684 \prop_get:NnN \l__coffin_display_handles_prop
8685 { #1 ##1 } \l__coffin_internal_tl
8686 \quark_if_no_value:NTF \l__coffin_internal_tl
8687 {
8688 \prop_get:NnN \l__coffin_display_handles_prop
8689 { ##1 #1 } \l__coffin_internal_tl
8690 \quark_if_no_value:NTF \l__coffin_internal_tl
8691 {
8692 \__coffin_display_attach:Nnnnn
8693 \l__coffin_display_coord_coffin { 1 } { vc }
8694 { 1 pt } { 0 pt }
8695 }
8696 {
8697 \exp_last_unbraced:No
8698 \__coffin_display_handles_aux:nnnn
8699 \l__coffin_internal_tl
8700 }
8701 }
8702 {
8703 \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
8704 \l__coffin_internal_tl
8705 }
8706 }
8707 }
8708 }
8709 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
8710 {
8711 \__coffin_display_attach:Nnnnn
8712 \l__coffin_display_coord_coffin {#1} {#2}
8713 { #3 \l__coffin_display_offset_dim }
8714 { #4 \l__coffin_display_offset_dim }
8715 }
8716 \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.

```

8717 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5

```

```

8718 {
8719   \_coffin_calculate_intersection:Nnn #1 {#2} {#3}
8720   \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
8721   \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
8722   \dim_set:Nn \l__coffin_offset_x_dim
8723     { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
8724   \dim_set:Nn \l__coffin_offset_y_dim
8725     { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
8726   \hbox_set:Nn \l__coffin_aligned_coffin
8727     {
8728       \box_use:N \l__coffin_display_coffin
8729       \tex_kern:D -\box_wd:N \l__coffin_display_coffin
8730       \tex_kern:D \l__coffin_offset_x_dim
8731       \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
8732     }
8733   \box_set_ht:Nn \l__coffin_aligned_coffin
8734     { \box_ht:N \l__coffin_display_coffin }
8735   \box_set_dp:Nn \l__coffin_aligned_coffin
8736     { \box_dp:N \l__coffin_display_coffin }
8737   \box_set_wd:Nn \l__coffin_aligned_coffin
8738     { \box_wd:N \l__coffin_display_coffin }
8739   \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
8740 }

```

(End definition for `\coffin_display_handles:Nn` and others. These functions are documented on page 148.)

`\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`

```

8741 \cs_new_protected:Npn \coffin_show_structure:N #1
8742 {
8743   \_coffin_if_exist:NT #1
8744   {
8745     \_msg_show_pre:nnxxxx { LaTeX / kernel } { show-coffin }
8746     { \token_to_str:N #1 }
8747     { \dim_eval:n { \coffin_ht:N #1 } }
8748     { \dim_eval:n { \coffin_dp:N #1 } }
8749     { \dim_eval:n { \coffin_wd:N #1 } }
8750     \_msg_show_wrap:n
8751     {
8752       \prop_map_function:cN
8753       { \l__coffin_poles_ \_int_value:w #1 _prop }
8754       \_msg_show_item_unbraced:nn
8755     }
8756   }
8757 }
8758 \cs_generate_variant:Nn \coffin_show_structure:N { c }

```

(End definition for `\coffin_show_structure:N`. This function is documented on page 149.)

16.8 Messages

```

8759 \_msg_kernel_new:nnnn { kernel } { no-pole-intersection }
8760 { No~intersection~between~coffin~poles. }
8761 {

```

```

8762 \c__msg_coding_error_text_tl
8763 LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
8764 but~they~do~not~have~a~unique~meeting~point:~
8765 the~value~(0~pt,~0~pt)~will~be~used.
8766 }
8767 \__msg_kernel_new:nnnn { kernel } { unknown-coffin }
8768 { Unknown-coffin~'#1'. }
8769 { The-coffin~'#1'~was~never~defined. }
8770 \__msg_kernel_new:nnnn { kernel } { unknown-coffin-pole }
8771 { Pole~'#1'~unknown-for-coffin~'#2'. }
8772 {
8773 \c__msg_coding_error_text_tl
8774 LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
8775 but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
8776 }
8777 \__msg_kernel_new:nnn { kernel } { show-coffin }
8778 {
8779 Size-of-coffin~#1 : \\
8780 > ~ ht~=#2 \\
8781 > ~ dp~=#3 \\
8782 > ~ wd~=#4 \\
8783 Poles-of-coffin~#1 :
8784 }
8785 </initex | package>

```

17 l3color Implementation

```

8786 <*initex | package>

```

\color_group_begin: Grouping for color is almost the same as using the basic `\group_begin:` and `\group_end:` functions. However, in vertical mode the end-of-group needs a `\par`, which in horizontal mode does nothing.

```

8787 \cs_new_eq:NN \color_group_begin: \group_begin:
8788 \cs_new_protected:Npn \color_group_end:
8789 {
8790 \tex_par:D
8791 \group_end:
8792 }

```

(End definition for \color_group_begin: and \color_group_end:. These functions are documented on page 150.)

\color_ensure_current: A driver-independent wrapper for setting the foreground color to the current color “now”.

```

8793 <*initex>
8794 \cs_new_protected:Npn \color_ensure_current:
8795 { \__driver_color_ensure_current: }
8796 </initex>

```

In package mode, the driver code may not be loaded. To keep down dependencies, if there is no driver code available and no `\set@color` then color is not in use and this function can be a no-op.

```

8797 <*package>
8798 \cs_new_protected:Npn \color_ensure_current: { }
8799 \AtBeginDocument

```

```

8800 {
8801   \cs_if_exist:NTF \__driver_color_ensure_current:
8802   {
8803     \cs_set_protected:Npn \color_ensure_current:
8804     { \__driver_color_ensure_current: }
8805   }
8806   {
8807     \cs_if_exist:NT \set@color
8808     {
8809       \cs_set_protected:Npn \color_ensure_current:
8810       { \set@color }
8811     }
8812   }
8813 }
8814 \</package>

```

(End definition for \color_ensure_current:. This function is documented on page 150.)

```
8815 \</initex | package>
```

18 l3msg implementation

```
8816 \*initex | package>
```

```
8817 \@@=msg
```

\l_msg_internal_tl A general scratch for the module.

```
8818 \tl_new:N \l__msg_internal_tl
```

(End definition for \l__msg_internal_tl.)

18.1 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.

\c__msg_more_text_prefix_tl

```
8819 \tl_const:Nn \c__msg_text_prefix_tl { msg~text~>~ }
```

```
8820 \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.

\msg_if_exist:nnTF

```
8821 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
```

```
8822 {
```

```
8823   \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
```

```
8824   { \prg_return_true: } { \prg_return_false: }
```

```
8825 }
```

(End definition for \msg_if_exist:nnTF. This function is documented on page 152.)

`__chk_if_free_msg:nn` This auxiliary is similar to `__chk_if_free_cs:N`, and is used when defining messages with `\msg_new:nnnn`. It could be inlined in `\msg_new:nnnn`, but the experimental `l3trace` module needs to disable this check when reloading a package with the extra tracing information.

```

8826 \cs_new_protected:Npn \__chk_if_free_msg:nn #1#2
8827 {
8828   \msg_if_exist:nnT {#1} {#2}
8829   {
8830     \__msg_kernel_error:nxxx { kernel } { message-already-defined }
8831     {#1} {#2}
8832   }
8833 }
8834 \*package
8835 \if_bool:N \l@expl@log@functions@bool
8836   \cs_gset_protected:Npn \__chk_if_free_msg:nn #1#2
8837   {
8838     \msg_if_exist:nnT {#1} {#2}
8839     {
8840       \__msg_kernel_error:nxxx { kernel } { message-already-defined }
8841       {#1} {#2}
8842     }
8843     \__chk_log:x { Defining~message~ #1 / #2 ~\msg_line_context: }
8844   }
8845 \fi:
8846 \*package

```

(End definition for `__chk_if_free_msg:nn`.)

`\msg_new:nnnn` Setting a message simply means saving the appropriate text into two functions. A sanity check first.

```

\msg_new:nnn
\msg_gset:nnnn
\msg_gset:nnn
\msg_set:nnnn
\msg_set:nnn
8847 \cs_new_protected:Npn \msg_new:nnnn #1#2
8848 {
8849   \__chk_if_free_msg:nn {#1} {#2}
8850   \msg_gset:nnnn {#1} {#2}
8851 }
8852 \cs_new_protected:Npn \msg_new:nnn #1#2#3
8853 { \msg_new:nnnn {#1} {#2} {#3} { } }
8854 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
8855 {
8856   \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
8857   ##1##2##3##4 {#3}
8858   \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
8859   ##1##2##3##4 {#4}
8860 }
8861 \cs_new_protected:Npn \msg_set:nnn #1#2#3
8862 { \msg_set:nnnn {#1} {#2} {#3} { } }
8863 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
8864 {
8865   \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
8866   ##1##2##3##4 {#3}
8867   \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
8868   ##1##2##3##4 {#4}
8869 }
8870 \cs_new_protected:Npn \msg_gset:nnn #1#2#3

```



```
8871 { \msg_gset:nnnn {#1} {#2} {#3} { } }
```

(End definition for `\msg_new:nnnn` and others. These functions are documented on page 151.)

18.2 Messages: support functions and text

Simple pieces of text for messages.

```
\c__msg_coding_error_text_tl
\c__msg_continue_text_tl      8872 \tl_const:Nn \c__msg_coding_error_text_tl
\c__msg_critical_text_tl      8873 {
    \c__msg_fatal_text_tl      8874   This-is-a-coding-error.
    \c__msg_help_text_tl       8875   \\\
\c__msg_no_info_text_tl       8876 }
\c__msg_on_line_text_tl       8877 \tl_const:Nn \c__msg_continue_text_tl
\c__msg_return_text_tl        8878 { Type~<return>~to~continue }
\c__msg_trouble_text_tl       8879 \tl_const:Nn \c__msg_critical_text_tl
                                8880 { Reading~the~current~file~'\g_file_current_name_tl'~will~stop. }
                                8881 \tl_const:Nn \c__msg_fatal_text_tl
                                8882 { This-is-a-fatal-error:~LaTeX~will~abort. }
\c__msg_help_text_tl          8883 \tl_const:Nn \c__msg_help_text_tl
                                8884 { For~immediate~help~type~H~<return> }
\c__msg_no_info_text_tl       8885 \tl_const:Nn \c__msg_no_info_text_tl
                                8886 {
                                8887   LaTeX~does~not~know~anything~more~about~this~error,~sorry.
                                8888   \c__msg_return_text_tl
                                8889 }
\c__msg_on_line_text_tl       8890 \tl_const:Nn \c__msg_on_line_text_tl { on-line }
\c__msg_return_text_tl        8891 \tl_const:Nn \c__msg_return_text_tl
                                8892 {
                                8893   \\\
                                8894   Try~typing~<return>~to~proceed.
                                8895   \\\
                                8896   If~that~doesn't~work,~type~X~<return>~to~quit.
                                8897 }
\c__msg_trouble_text_tl       8898 \tl_const:Nn \c__msg_trouble_text_tl
                                8899 {
                                8900   \\\
                                8901   More~errors~will~almost~certainly~follow: \\\
                                8902   the~LaTeX~run~should~be~aborted.
                                8903 }
```

(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.

```
8904 \cs_new:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
8905 \cs_gset:Npn \msg_line_context:
8906 {
8907   \c__msg_on_line_text_tl
8908   \c_space_tl
8909   \msg_line_number:
8910 }
```

(End definition for `\msg_line_number:` and `\msg_line_context:`. These functions are documented on page 152.)

18.3 Showing messages: low level mechanism

`\msg_interrupt:nnn` 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 T_EX's `\errhelp` register before issuing an `\errmessage`.

```

8911 \cs_new_protected:Npn \msg_interrupt:nnn #1#2#3
8912 {
8913   \tl_if_empty:nTF {#3}
8914   {
8915     \__msg_interrupt_wrap:nn { \ \ \c__msg_no_info_text_tl }
8916     {#1 \ \ \ \ #2 \ \ \ \ \c__msg_continue_text_tl }
8917   }
8918   {
8919     \__msg_interrupt_wrap:nn { \ \ #3 }
8920     {#1 \ \ \ \ #2 \ \ \ \ \c__msg_help_text_tl }
8921   }
8922 }
```

(End definition for `\msg_interrupt:nnn`. This function is documented on page 156.)

`__msg_interrupt_wrap:nn`
`__msg_interrupt_more_text:n`

First setup T_EX'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.

```

8923 \cs_new_protected:Npn \__msg_interrupt_wrap:nn #1#2
8924 {
8925   \iow_wrap:nnnN {#1} { | ~ } { } \__msg_interrupt_more_text:n
8926   \iow_wrap:nnnN {#2} { ! ~ } { } \__msg_interrupt_text:n
8927 }
8928 \cs_new_protected:Npn \__msg_interrupt_more_text:n #1
8929 {
8930   \exp_args:Nx \tex_errhelp:D
8931   {
8932     |,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
8933     #1 \iow_newline:
8934     |,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
8935   }
8936 }
```

(End definition for `__msg_interrupt_wrap:nn` and `__msg_interrupt_more_text:n`.)

`__msg_interrupt_text: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”: T_EX'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 {<dots>}` which fills the output with dots. 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 will be inserted after the message is entirely cleaned up.

The `__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.

```

8937 \group_begin:
8938   \char_set_lccode:nn {'\} {'\ }
8939   \char_set_lccode:nn {'\} {'\ }
8940   \char_set_lccode:nn {'&} {'\!}
8941   \char_set_catcode_active:N \&
8942 \tex_lowercase:D
8943 {
8944   \group_end:
8945   \cs_new_protected:Npn \__msg_interrupt_text:n #1
8946   {
8947     \iow_term:x
8948     {
8949       \iow_newline:
8950       !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
8951       \iow_newline:
8952       !
8953     }
8954     \__iow_with:Nnn \tex_newlinechar:D { '\^^J }
8955     {
8956       \__iow_with:Nnn \tex_errorcontextlines:D { - \c_one }
8957       {
8958         \group_begin:
8959         \cs_set_protected:Npn &
8960         {
8961           \tex_errmessage:D
8962           {
8963             #1
8964             \use_none:n
8965             { ..... }
8966           }
8967         }
8968         \exp_after:wN
8969         \group_end:
8970         &
8971       }
8972     }
8973   }
8974 }
```

(End definition for `__msg_interrupt_text:n`.)

`\msg_log:n` Printing to the log or terminal without a stop is rather easier. A bit of simple visual work sets things off nicely.

```

8975 \cs_new_protected:Npn \msg_log:n #1
8976 {
8977   \iow_log:n { ..... }
8978   \iow_wrap:nnnN { . ~ #1 } { . ~ } { } \iow_log:n
8979   \iow_log:n { ..... }
```

```

8980 }
8981 \cs_new_protected:Npn \msg_term:n #1
8982 {
8983   \iow_term:n { ***** }
8984   \iow_wrap:nnnN { * ~ #1 } { * ~ } { } \iow_term:n
8985   \iow_term:n { ***** }
8986 }

```

(End definition for `\msg_log:n` and `\msg_term:n`. These functions are documented on page 157.)

18.4 Displaying messages

L^AT_EX is handling error messages and so the T_EX ones are disabled. This is already done by the L^AT_EX 2_ε kernel, so to avoid messing up any deliberate change by a user this is only set in format mode.

```

8987 <*initex>
8988 \int_gset:Nn \tex_errorcontextlines:D { - \c_one }
8989 </initex>

```

A function for issuing messages: both the text and order could in principle vary.

```

\msg_fatal_text:n
\msg_critical_text:n
\msg_error_text:n
\msg_warning_text:n
\msg_info_text:n
8990 \cs_new:Npn \msg_fatal_text:n #1 { Fatal~#1~error }
8991 \cs_new:Npn \msg_critical_text:n #1 { Critical~#1~error }
8992 \cs_new:Npn \msg_error_text:n #1 { #1~error }
8993 \cs_new:Npn \msg_warning_text:n #1 { #1~warning }
8994 \cs_new:Npn \msg_info_text:n #1 { #1~info }

```

(End definition for `\msg_fatal_text:n` and others. These functions are documented on page 152.)

`\msg_see_documentation_text:n` Contextual footer information. The L^AT_EX module only comprises L^AT_EX3 code, so we refer to the L^AT_EX3 documentation rather than simply “L^AT_EX”.

```

8995 \cs_new:Npn \msg_see_documentation_text:n #1
8996 {
8997   \\\ See~the~
8998   \str_if_eq:nnTF {#1} { LaTeX } { LaTeX3 } {#1} ~
8999   documentation~for~further~information.
9000 }

```

(End definition for `\msg_see_documentation_text:n`. This function is documented on page 153.)

`__msg_class_new:nn`

```

9001 \group_begin:
9002 \cs_set_protected:Npn \__msg_class_new:nn #1#2
9003 {
9004   \prop_new:c { l__msg_redirect_ #1 _prop }
9005   \cs_new_protected:cpn { __msg_ #1 _code:nnnnnn }
9006     ##1##2##3##4##5##6 {#2}
9007   \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
9008   {
9009     \use:x
9010     {
9011       \exp_not:n { \__msg_use:nnnnnnn {#1} {##1} {##2} }
9012       { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
9013       { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
9014     }

```

```

9015     }
9016     \cs_new_protected:cpx { msg_ #1 :nnnnn } ##1##2##3##4##5
9017     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
9018     \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
9019     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
9020     \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3
9021     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
9022     \cs_new_protected:cpx { msg_ #1 :nn } ##1##2
9023     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
9024     \cs_new_protected:cpx { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
9025     {
9026         \use:x
9027         {
9028             \exp_not:N \exp_not:n
9029             { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
9030             {##3} {##4} {##5} {##6}
9031         }
9032     }
9033     \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
9034     { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
9035     \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
9036     { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
9037     \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3
9038     { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
9039 }

```

(End definition for `_msg_class_new:nn`.)

`\msg_fatal:nnnnnn` For fatal errors, after the error message TeX bails out.

```

\msg_fatal:nnxxxx 9040 \_msg_class_new:nn { fatal }
\msg_fatal:nnnnn 9041 {
\msg_fatal:nnxxx 9042     \msg_interrupt:nnn
\msg_fatal:nnnn 9043     { \msg_fatal_text:n {#1} : ~ "#2" }
\msg_fatal:nnxx 9044     {
\msg_fatal:nnn 9045         \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
\msg_fatal:nnx 9046         \msg_see_documentation_text:n {#1}
\msg_fatal:nn 9047     }
9048     { \c_msg_fatal_text_tl }
9049     \tex_end:D
9050 }

```

(End definition for `\msg_fatal:nnnnnn` and others. These functions are documented on page 153.)

`\msg_critical:nnnnnn` Not quite so bad: just end the current file.

```

\msg_critical:nnxxxx 9051 \_msg_class_new:nn { critical }
\msg_critical:nnnnn 9052 {
\msg_critical:nnxxx 9053     \msg_interrupt:nnn
\msg_critical:nnnn 9054     { \msg_critical_text:n {#1} : ~ "#2" }
\msg_critical:nnxx 9055     {
\msg_critical:nnn 9056         \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
\msg_critical:nnx 9057         \msg_see_documentation_text:n {#1}
\msg_critical:nn 9058     }
9059     { \c_msg_critical_text_tl }
9060     \tex_endinput:D
9061 }

```

(End definition for `\msg_critical:nnnnnn` and others. These functions are documented on page 153.)

`\msg_error:nnnnnn` For an error, the interrupt routine is called. We check if there is a “more text” by
`\msg_error:nnxxxx` comparing that control sequence with a permanently empty text.
`\msg_error:nnnnn` 9062 `_msg_class_new:nn { error }`
`\msg_error:nnxxx` 9063 `{`
`\msg_error:nnnn` 9064 `_msg_error:cnnnnn`
`\msg_error:nnxx` 9065 `{ \c_msg_more_text_prefix_tl #1 / #2 }`
`\msg_error:nnn` 9066 `{#3} {#4} {#5} {#6}`
`\msg_error:nnx` 9067 `{`
`\msg_error:nn` 9068 `\msg_interrupt:nnn`
`_msg_error:cnnnnn` 9069 `{ \msg_error_text:n {#1} : ~ "#2" }`
`_msg_no_more_text:nnnn` 9070 `{`
9071 `\use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}`
9072 `\msg_see_documentation_text:n {#1}`
9073 `}`
9074 `}`
9075 `}`
9076 `\cs_new_protected:Npn _msg_error:cnnnnn #1#2#3#4#5#6`
9077 `{`
9078 `\cs_if_eq:cNTF {#1} _msg_no_more_text:nnnn`
9079 `{ #6 { } }`
9080 `{ #6 { \use:c {#1} {#2} {#3} {#4} {#5} } }`
9081 `}`
9082 `\cs_new:Npn _msg_no_more_text:nnnn #1#2#3#4 { }`

(End definition for `\msg_error:nnnnnn` and others. These functions are documented on page 154.)

`\msg_warning:nnnnnn` Warnings are printed to the terminal.
`\msg_warning:nnxxxx` 9083 `_msg_class_new:nn { warning }`
`\msg_warning:nnnnn` 9084 `{`
`\msg_warning:nnxxx` 9085 `\msg_term:n`
`\msg_warning:nnnn` 9086 `{`
`\msg_warning:nnxx` 9087 `\msg_warning_text:n {#1} : ~ "#2" \\ \\`
`\msg_warning:nnn` 9088 `\use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}`
`\msg_warning:nnx` 9089 `}`
`\msg_warning:nn` 9090 `}`

(End definition for `\msg_warning:nnnnnn` and others. These functions are documented on page 154.)

`\msg_info:nnnnnn` Information only goes into the log.
`\msg_info:nnxxxx` 9091 `_msg_class_new:nn { info }`
`\msg_info:nnnnn` 9092 `{`
`\msg_info:nnxxx` 9093 `\msg_log:n`
`\msg_info:nnnn` 9094 `{`
`\msg_info:nnxx` 9095 `\msg_info_text:n {#1} : ~ "#2" \\ \\`
`\msg_info:nnn` 9096 `\use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}`
`\msg_info:nnx` 9097 `}`
`\msg_info:nn` 9098 `}`

(End definition for `\msg_info:nnnnnn` and others. These functions are documented on page 154.)

“Log” data is very similar to information, but with no extras added.

```

\msg_log:nnnnnn
\msg_log:nnxxxx 9099 \__msg_class_new:nn { log }
\msg_log:nnnnn 9100 {
\msg_log:nnxxx 9101 \iow_wrap:nnnN
\msg_log:nnnn 9102 { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
\msg_log:nnxx 9103 { } { } \iow_log:n
\msg_log:nnn 9104 }
\msg_log:nnx
\msg_log:nn

```

(End definition for \msg_log:nnnnnn and others. These functions are documented on page 154.)

The none message type is needed so that input can be gobbled.

```

\msg_none:nnnnnn 9105 \__msg_class_new:nn { none } { }
\msg_none:nnxxxx
\msg_none:nnnnn
\msg_none:nnxxx
\msg_none:nnnn
\msg_none:nnxx 9106 \group_end:
\msg_none:nnn
\msg_none:nnx
\msg_none:nn

```

(End definition for \msg_none:nnnnnn and others. These functions are documented on page 155.)

End the group to eliminate __msg_class_new:nn.

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.

```

\__msg_class_chk_exist:nT 9107 \cs_new:Npn \__msg_class_chk_exist:nT #1
9108 {
9109 \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
9110 { \__msg_kernel_error:nnx { kernel } { message-class-unknown } {#1} }
9111 }

```

(End definition for __msg_class_chk_exist:nT.)

Support variables needed for the redirection system.

```

\l__msg_class_tl
\l__msg_current_class_tl 9112 \tl_new:N \l__msg_class_tl
9113 \tl_new:N \l__msg_current_class_tl

```

(End definition for \l__msg_class_tl and \l__msg_current_class_tl.)

For redirection of individually-named messages

```

\l__msg_redirect_prop 9114 \prop_new:N \l__msg_redirect_prop

```

(End definition for \l__msg_redirect_prop.)

During redirection, split the message name into a sequence with items {/module/submodule}, {/module}, and {}.

```

\l__msg_hierarchy_seq 9115 \seq_new:N \l__msg_hierarchy_seq

```

(End definition for \l__msg_hierarchy_seq.)

Classes encountered when following redirections to check for loops.

```

\l__msg_class_loop_seq 9116 \seq_new:N \l__msg_class_loop_seq

```

(End definition for \l__msg_class_loop_seq.)

`__msg_use:nnnnnnn` 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.

```

9117 \cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
9118 {
9119     \msg_if_exist:nnTF {#2} {#3}
9120     {
9121         \__msg_class_chk_exist:nT {#1}
9122         {
9123             \tl_set:Nn \l__msg_current_class_tl {#1}
9124             \cs_set_protected:Npx \__msg_use_code:
9125             {
9126                 \exp_not:n
9127                 {
9128                     \use:c { __msg_ \l__msg_class_tl _code:nnnnnnn }
9129                     {#2} {#3} {#4} {#5} {#6} {#7}
9130                 }
9131             }
9132             \__msg_use_redirect_name:n { #2 / #3 }
9133         }
9134     }
9135     { \__msg_kernel_error:nxx { kernel } { message-unknown } {#2} {#3} }
9136 }
9137 \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/message (with an arbitrary number of slashes), and store `{/module/submodule}`, `{/module}` and `{}` into `\l__msg_hierarchy_seq`. We will then map through this sequence, applying the most specific redirection.

```

9138 \cs_new_protected:Npn \__msg_use_redirect_name:n #1
9139 {
9140     \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
9141     { \__msg_use_code: }
9142     {
9143         \seq_clear:N \l__msg_hierarchy_seq
9144         \__msg_use_hierarchy:nwwN { }
9145         #1 \q_mark \__msg_use_hierarchy:nwwN
9146         / \q_mark \use_none_delimit_by_q_stop:w
9147         \q_stop
9148         \__msg_use_redirect_module:n { }
9149     }
9150 }
9151 \cs_new_protected:Npn \__msg_use_hierarchy:nwwN #1#2 / #3 \q_mark #4
9152 {
9153     \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
9154     #4 { #1 / #2 } #3 \q_mark #4
9155 }

```

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.

```

9156 \cs_new_protected:Npn \__msg_use_redirect_module:n #1
9157 {
9158   \seq_map_inline:Nn \l__msg_hierarchy_seq
9159   {
9160     \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
9161     {##1} \l__msg_class_tl
9162     {
9163       \seq_map_break:n
9164       {
9165         \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
9166         { \__msg_use_code: }
9167         {
9168           \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
9169           \__msg_use_redirect_module:n {##1}
9170         }
9171       }
9172     }
9173     {
9174       \str_if_eq:nnT {##1} {#1}
9175       {
9176         \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
9177         \seq_map_break:n { \__msg_use_code: }
9178       }
9179     }
9180   }
9181 }

```

(End definition for __msg_use:nnnnnnn and others.)

\msg_redirect_name:nnn Named message will always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```

9182 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
9183 {
9184   \tl_if_empty:nTF {#3}
9185   { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
9186   {
9187     \__msg_class_chk_exist:nT {#3}
9188     { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
9189   }
9190 }

```

(End definition for \msg_redirect_name:nnn. This function is documented on page 156.)

\msg_redirect_class:nn 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.

\msg_redirect_module:nnn

__msg_redirect:nnn

__msg_redirect_loop_chk:nnn

__msg_redirect_loop_list:n

```

9191 \cs_new_protected:Npn \msg_redirect_class:nn
9192 { \__msg_redirect:nnn { } }
9193 \cs_new_protected:Npn \msg_redirect_module:nnn #1
9194 { \__msg_redirect:nnn { / #1 } }
9195 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
9196 {
9197   \__msg_class_chk_exist:nT {#2}
9198   {
9199     \tl_if_empty:nTF {#3}
9200     { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
9201     {
9202       \__msg_class_chk_exist:nT {#3}
9203       {
9204         \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
9205         \tl_set:Nn \l__msg_current_class_tl {#2}
9206         \seq_clear:N \l__msg_class_loop_seq
9207         \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
9208       }
9209     }
9210   }
9211 }

```

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_tl`, 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.

```

9212 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
9213 {
9214   \seq_put_right:Nn \l__msg_class_loop_seq {#1}
9215   \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
9216   {
9217     \str_if_eq_x:nnF { \l__msg_class_tl } {#1}
9218     {
9219       \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
9220       {
9221         \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
9222         \__msg_kernel_warning:nnxxxx
9223         { kernel } { message-redirect-loop }
9224         { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
9225         { \seq_item:Nn \l__msg_class_loop_seq { \c_two } }
9226         {#3}
9227         {
9228           \seq_map_function:NN \l__msg_class_loop_seq
9229           \__msg_redirect_loop_list:n
9230           { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
9231         }
9232       }
9233     }
9234   }

```

```

9233             { \_msg_redirect_loop_chk:onn \l\_msg_class_tl {#2} {#3} }
9234         }
9235     }
9236 }
9237 \cs_generate_variant:Nn \_msg_redirect_loop_chk:nnn { o }
9238 \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 155.)

18.5 Kernel-specific functions

`_msg_kernel_new:nnnn` 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.

```

9239 \cs_new_protected:Npn \_msg_kernel_new:nnnn #1#2
9240 { \msg_new:nnnn { LaTeX } { #1 / #2 } }
9241 \cs_new_protected:Npn \_msg_kernel_new:nnn #1#2
9242 { \msg_new:nnn { LaTeX } { #1 / #2 } }
9243 \cs_new_protected:Npn \_msg_kernel_set:nnnn #1#2
9244 { \msg_set:nnnn { LaTeX } { #1 / #2 } }
9245 \cs_new_protected:Npn \_msg_kernel_set:nnn #1#2
9246 { \msg_set:nnn { LaTeX } { #1 / #2 } }

```

(End definition for `_msg_kernel_new:nnnn` and others.)

`_msg_kernel_class_new: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.

```

9247 \group_begin:
9248   \cs_set_protected:Npn \_msg_kernel_class_new:nN #1
9249   { \_msg_kernel_class_new_aux:nN { kernel_ #1 } }
9250   \cs_set_protected:Npn \_msg_kernel_class_new_aux:nN #1#2
9251   {
9252     \cs_new_protected:cpn { \_msg_ #1 :nnnnnn } ##1##2##3##4##5##6
9253     {
9254       \use:x
9255       {
9256         \exp_not:n { #2 { LaTeX } { ##1 / ##2 } }
9257         { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
9258         { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
9259       }
9260     }
9261     \cs_new_protected:cpx { \_msg_ #1 :nnnnn } ##1##2##3##4##5
9262     { \exp_not:c { \_msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
9263     \cs_new_protected:cpx { \_msg_ #1 :nnnn } ##1##2##3##4
9264     { \exp_not:c { \_msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
9265     \cs_new_protected:cpx { \_msg_ #1 :nnn } ##1##2##3
9266     { \exp_not:c { \_msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
9267     \cs_new_protected:cpx { \_msg_ #1 :nn } ##1##2
9268     { \exp_not:c { \_msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
9269     \cs_new_protected:cpx { \_msg_ #1 :nnxxx } ##1##2##3##4##5##6
9270     {
9271       \use:x

```

```

9272         {
9273             \exp_not:N \exp_not:n
9274             { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} }
9275             {##3} {##4} {##5} {##6}
9276         }
9277     }
9278     \cs_new_protected:cpx { __msg_ #1 :nnxxx } ##1##2##3##4##5
9279     { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
9280     \cs_new_protected:cpx { __msg_ #1 :nnxx } ##1##2##3##4
9281     { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
9282     \cs_new_protected:cpx { __msg_ #1 :nnx } ##1##2##3
9283     { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
9284 }

```

(End definition for `_msg_kernel_class_new:nN` and `_msg_kernel_class_new_aux:nN`.)

```

\_msg_kernel_fatal:nnnnnn
\_msg_kernel_fatal:nnxxxx
\_msg_kernel_fatal:nnnnn
\_msg_kernel_fatal:nnxxx
\_msg_kernel_fatal:nnnn
\_msg_kernel_fatal:nnxx
\_msg_kernel_fatal:nnn
\_msg_kernel_fatal:nnx
\_msg_kernel_fatal:nn
\_msg_kernel_fatal:nnnnnn
\_msg_kernel_error:nnnnnn
\_msg_kernel_warning:nnxxxx
\_msg_kernel_warning:nnnnn
\_msg_kernel_warning:nnxxx
\_msg_kernel_warning:nnxx
\_msg_kernel_warning:nnn
\_msg_kernel_warning:nnx
\_msg_kernel_warning:nn
\_msg_kernel_warning:nnnnnn
\_msg_kernel_info:nnnnnn
\_msg_kernel_info:nnxxxx
\_msg_kernel_info:nnnnn
\_msg_kernel_info:nnxxx
\_msg_kernel_info:nnnn
\_msg_kernel_info:nnxx
\_msg_kernel_info:nnn
\_msg_kernel_info:nnx
\_msg_kernel_info:nn

```

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.

```

9285 \_msg_kernel_class_new:nN { fatal } \_msg_fatal_code:nnnnnn
9286 \cs_undefine:N \_msg_kernel_error:nnxx
9287 \cs_undefine:N \_msg_kernel_error:nnx
9288 \cs_undefine:N \_msg_kernel_error:nn
9289 \_msg_kernel_class_new:nN { error } \_msg_error_code:nnnnnn

```

(End definition for `_msg_kernel_fatal:nnnnnn` and others.)

Kernel messages which can be redirected simply use the machinery for normal messages, with the module name “`LaTeX`”.

```

9290 \_msg_kernel_class_new:nN { warning } \msg_warning:nnxxxx
9291 \_msg_kernel_class_new:nN { info } \msg_info:nnxxxx

```

(End definition for `_msg_kernel_warning:nnnnnn` and others.)

End the group to eliminate `_msg_kernel_class_new:nN`.

```

9292 \group_end:

```

Error messages needed to actually implement the message system itself.

```

9293 \_msg_kernel_new:nnnn { kernel } { message-already-defined }
9294 { Message~'#2'~for~module~'#1'~already-defined. }
9295 {
9296     \c__msg_coding_error_text_tl
9297     LaTeX~was~asked~to~define~a~new~message~called~'#2'\
9298     by~the~module~'#1':~this~message~already~exists.
9299     \c__msg_return_text_tl
9300 }
9301 \_msg_kernel_new:nnnn { kernel } { message-unknown }
9302 { Unknown~message~'#2'~for~module~'#1'. }
9303 {
9304     \c__msg_coding_error_text_tl
9305     LaTeX~was~asked~to~display~a~message~called~'#2'\
9306     by~the~module~'#1':~this~message~does~not~exist.
9307     \c__msg_return_text_tl
9308 }
9309 \_msg_kernel_new:nnnn { kernel } { message-class-unknown }
9310 { Unknown~message~class~'#1'. }

```

```

9311 {
9312   LaTeX-has-been-asked-to-redirect-messages-to-a-class-~'~#1':~\\
9313   this-was-never-defined.
9314   \c__msg_return_text_tl
9315 }
9316 \__msg_kernel_new:nnnn { kernel } { message-redirect-loop }
9317 {
9318   Message-redirect-loop-caused-by~ {~#1} ~>~ {~#2}
9319   \tl_if_empty:nF {~#3} { ~for-module~' \use_none:n #3 ' } .
9320 }
9321 {
9322   Adding-the-message-redirection~ {~#1} ~>~ {~#2}
9323   \tl_if_empty:nF {~#3} { ~for-the-module~' \use_none:n #3 ' } ~
9324   created-an-infinite-loop\\
9325   \iow_indent:n { #4 \\ }
9326 }

```

Messages for earlier kernel modules.

```

9327 \__msg_kernel_new:nnnn { kernel } { bad-number-of-arguments }
9328 { Function-~'~#1'~cannot-be-defined-with~#2~arguments. }
9329 {
9330   \c__msg_coding_error_text_tl
9331   LaTeX-has-been-asked-to-define-a-function-~'~#1'~with~
9332   #2~arguments.~
9333   TeX-allows-between-0-and-9~arguments-for-a-single-function.
9334 }
9335 \__msg_kernel_new:nnn { kernel } { char-active }
9336 { Cannot-generate-active-chars. }
9337 \__msg_kernel_new:nnn { kernel } { char-invalid-catcode }
9338 { Invalid-catcode-for-char-generation. }
9339 \__msg_kernel_new:nnn { kernel } { char-null-space }
9340 { Cannot-generate-null-char-as-a-space. }
9341 \__msg_kernel_new:nnn { kernel } { char-out-of-range }
9342 { Charcode-requested-out-of-engine-range. }
9343 \__msg_kernel_new:nnn { kernel } { char-space }
9344 { Cannot-generate-space-chars. }
9345 \__msg_kernel_new:nnnn { kernel } { command-already-defined }
9346 { Control-sequence-~#1~already-defined. }
9347 {
9348   \c__msg_coding_error_text_tl
9349   LaTeX-has-been-asked-to-create-a-new-control-sequence-~'~#1'~
9350   but-this-name-has-already-been-used-elsewhere. \\ \\
9351   The-current-meaning-is:\\
9352   \ \ #2
9353 }
9354 \__msg_kernel_new:nnnn { kernel } { command-not-defined }
9355 { Control-sequence-~#1~undefined. }
9356 {
9357   \c__msg_coding_error_text_tl
9358   LaTeX-has-been-asked-to-use-a-control-sequence-~'~#1':~\\
9359   this-has-not-been-defined-yet.
9360 }
9361 \__msg_kernel_new:nnnn { kernel } { empty-search-pattern }
9362 { Empty-search-pattern. }
9363 {

```

```

9364 \c__msg_coding_error_text_tl
9365 LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'~#1~':~that~
9366 would~lead~to~an~infinite~loop!
9367 }
9368 \__msg_kernel_new:nnnn { kernel } { out-of-registers }
9369 { No~room~for~a~new~#1. }
9370 {
9371   TeX~only~supports~\int~use:N \c_max_register_int \ %
9372   of~each~type.~All~the~#1~registers~have~been~used.~
9373   This~run~will~be~aborted~now.
9374 }
9375 \__msg_kernel_new:nnnn { kernel } { non-base-function }
9376 { Function~'~#1~'~is~not~a~base~function }
9377 {
9378   \c__msg_coding_error_text_tl
9379   Functions~defined~through~\iow_char:N\cs_new:Nn~must~have~
9380   a~signature~consisting~of~only~normal~arguments~'~N~'~and~'~n~'.~
9381   To~define~variants~use~\iow_char:N\cs_generate_variant:Nn~
9382   and~to~define~other~functions~use~\iow_char:N\cs_new:Npn.
9383 }
9384 \__msg_kernel_new:nnnn { kernel } { missing-colon }
9385 { Function~'~#1~'~contains~no~':~'. }
9386 {
9387   \c__msg_coding_error_text_tl
9388   Code~level~functions~must~contain~':~'~to~separate~the~
9389   argument~specification~from~the~function~name.~This~is~
9390   needed~when~defining~conditionals~or~variants,~or~when~building~a~
9391   parameter~text~from~the~number~of~arguments~of~the~function.
9392 }
9393 \__msg_kernel_new:nnnn { kernel } { protected-predicate }
9394 { Predicate~'~#1~'~must~be~expandable. }
9395 {
9396   \c__msg_coding_error_text_tl
9397   LaTeX~has~been~asked~to~define~'~#1~'~as~a~protected~predicate.~
9398   Only~expandable~tests~can~have~a~predicate~version.
9399 }
9400 \__msg_kernel_new:nnnn { kernel } { conditional-form-unknown }
9401 { Conditional~form~'~#1~'~for~function~'~#2~'~unknown. }
9402 {
9403   \c__msg_coding_error_text_tl
9404   LaTeX~has~been~asked~to~define~the~conditional~form~'~#1~'~of~
9405   the~function~'~#2~',~but~only~'~TF~',~'~T~',~'~F~',~and~'~p~'~forms~exist.
9406 }
9407 (*package)
9408 \bool_if:NT \l@expl@check@declarations@bool
9409 {
9410   \__msg_kernel_new:nnnn { check } { non-declared-variable }
9411   { The~variable~#1~has~not~been~declared~\msg_line_context:. }
9412   {
9413     Checking~is~active,~and~you~have~tried~do~so~something~like: \\\
9414     \ \ \tl_set:Nn ~ #1 ~ \{ ~ ... ~ \} \\\
9415     without~first~having: \\\
9416     \ \ \tl_new:N ~ #1 \\\
9417     \\\

```

```

9418         LaTeX~will~create~the~variable~and~continue.
9419     }
9420 }
9421 </package>
9422 \__msg_kernel_new:nnnn { kernel } { scanmark-already-defined }
9423 { Scan~mark~#1~already~defined. }
9424 {
9425     \c__msg_coding_error_text_tl
9426     LaTeX~has~been~asked~to~create~a~new~scan~mark~'~#1'~
9427     but~this~name~has~already~been~used~for~a~scan~mark.
9428 }
9429 \__msg_kernel_new:nnnn { kernel } { variable-not-defined }
9430 { Variable~#1~undefined. }
9431 {
9432     \c__msg_coding_error_text_tl
9433     LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
9434     been~defined~yet.
9435 }
9436 \__msg_kernel_new:nnnn { kernel } { variant-too-long }
9437 { Variant~form~'~#1'~longer~than~base~signature~of~'~#2'. }
9438 {
9439     \c__msg_coding_error_text_tl
9440     LaTeX~has~been~asked~to~create~a~variant~of~the~function~'~#2'~
9441     with~a~signature~starting~with~'~#1',~but~that~is~longer~than~
9442     the~signature~(part~after~the~colon)~of~'~#2'.
9443 }
9444 \__msg_kernel_new:nnnn { kernel } { invalid-variant }
9445 { Variant~form~'~#1'~invalid~for~base~form~'~#2'. }
9446 {
9447     \c__msg_coding_error_text_tl
9448     LaTeX~has~been~asked~to~create~a~variant~of~the~function~'~#2'~
9449     with~a~signature~starting~with~'~#1',~but~cannot~change~an~argument~
9450     from~type~'~#3'~to~type~'~#4'.
9451 }

```

Some errors only appear in expandable settings, hence don't need a “more-text” argument.

```

9452 \__msg_kernel_new:nnn { kernel } { bad-variable }
9453 { Erroneous~variable~#1~used! }
9454 \__msg_kernel_new:nnn { kernel } { misused-sequence }
9455 { A~sequence~was~misused. }
9456 \__msg_kernel_new:nnn { kernel } { misused-prop }
9457 { A~property~list~was~misused. }
9458 \__msg_kernel_new:nnn { kernel } { negative-replication }
9459 { Negative~argument~for~\prg_replicate:nn. }
9460 \__msg_kernel_new:nnn { kernel } { unknown-comparison }
9461 { Relation~'~#1'~unknown:~use~=,~<,~>,~==,~!=,~<=,~>=. }
9462 \__msg_kernel_new:nnn { kernel } { zero-step }
9463 { Zero~step~size~for~step~function~#1. }

```

Messages used by the “show” functions.

```

9464 \__msg_kernel_new:nnn { kernel } { show-clist }
9465 {
9466     The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
9467     \tl_if_empty:nTF {#2}

```

```

9468     { is~empty }
9469     { contains~the~items~(without~outer~braces): }
9470   }
9471   \__msg_kernel_new:nnn { kernel } { show-prop }
9472   {
9473     The~property~list~#1~
9474     \tl_if_empty:nTF {#2}
9475       { is~empty }
9476       { contains~the~pairs~(without~outer~braces): }
9477   }
9478   \__msg_kernel_new:nnn { kernel } { show-seq }
9479   {
9480     The~sequence~#1~
9481     \tl_if_empty:nTF {#2}
9482       { is~empty }
9483       { contains~the~items~(without~outer~braces): }
9484   }
9485   \__msg_kernel_new:nnn { kernel } { show-streams }
9486   {
9487     \tl_if_empty:nTF {#2} { No~ } { The~following~ }
9488     \str_case:nn {#1}
9489       {
9490         { ior } { input ~ }
9491         { iow } { output ~ }
9492       }
9493     streams~are~
9494     \tl_if_empty:nTF {#2} { open } { in~use: }
9495   }

```

18.6 Expandable errors

`__msg_expandable_error:n` In expansion only context, we cannot use the normal means of reporting errors. Instead, we feed \TeX 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, \TeX 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 `\q_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:.`

```

9496 \group_begin:
9497 \cs_set_protected:Npn \__msg_tmp:w #1#2
9498   {
9499     \cs_new:Npn \__msg_expandable_error:n ##1
9500       {
9501         \exp:w
9502         \exp_after:wN \exp_after:wN
9503         \exp_after:wN \__msg_expandable_error:w

```



```

9504         \exp_after:wN \exp_after:wN
9505         \exp_after:wN \exp_end:
9506         \use:n { #1 #2 ##1 } #2
9507     }
9508     \cs_new:Npn \__msg_expandable_error:w ##1 #2 ##2 #2 {##1}
9509 }
9510 \exp_args:Ncx \__msg_tmp:w { LaTeX3~error: }
9511 { \char_generate:nn { '\ } { 7 } }
9512 \group_end:

```

(End definition for __msg_expandable_error:n and __msg_expandable_error:w.)

The command built from the csname \c_@@_text_prefix_tl LaTeX / #1 / #2 takes four arguments and builds the error text, which is fed to __msg_expandable_error:n.

```

\__msg_kernel_expandable_error:nnnnnn
\__msg_kernel_expandable_error:nnnnn
\__msg_kernel_expandable_error:nnnn
\__msg_kernel_expandable_error:nnn
\__msg_kernel_expandable_error:nn
9513 \cs_new:Npn \__msg_kernel_expandable_error:nnnnnn #1#2#3#4#5#6
9514 {
9515     \exp_args:Nf \__msg_expandable_error:n
9516     {
9517         \exp_args:NNc \exp_after:wN \exp_stop_f:
9518         { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
9519         {#3} {#4} {#5} {#6}
9520     }
9521 }
9522 \cs_new:Npn \__msg_kernel_expandable_error:nnnnn #1#2#3#4#5
9523 {
9524     \__msg_kernel_expandable_error:nnnnnn
9525     {#1} {#2} {#3} {#4} {#5} { }
9526 }
9527 \cs_new:Npn \__msg_kernel_expandable_error:nnnn #1#2#3#4
9528 {
9529     \__msg_kernel_expandable_error:nnnnnn
9530     {#1} {#2} {#3} {#4} { } { }
9531 }
9532 \cs_new:Npn \__msg_kernel_expandable_error:nnn #1#2#3
9533 {
9534     \__msg_kernel_expandable_error:nnnnnn
9535     {#1} {#2} {#3} { } { } { }
9536 }
9537 \cs_new:Npn \__msg_kernel_expandable_error:nn #1#2
9538 {
9539     \__msg_kernel_expandable_error:nnnnnn
9540     {#1} {#2} { } { } { } { }
9541 }

```

(End definition for __msg_kernel_expandable_error:nnnnnn and others.)

18.7 Showing variables

Functions defined in this section are used for diagnostic functions in l3clist, l3file, l3prop, l3seq, xtemplate

```

\g__msg_log_next_bool
\__msg_log_next:
9542 \bool_new:N \g__msg_log_next_bool
9543 \cs_new_protected:Npn \__msg_log_next:
9544 { \bool_gset_true:N \g__msg_log_next_bool }

```

(End definition for `\g__msg_log_next_bool` and `__msg_log_next:.`)

`__msg_show_pre:nnnnnn` Print the text of a message to the terminal or log file without formatting: short cuts around `\iow_wrap:nnnN`. The choice of terminal or log file is done by `__msg_show_pre_aux:n`.

```

9545 \cs_new_protected:Npn \__msg_show_pre:nnnnnn #1#2#3#4#5#6
9546 {
9547   \exp_args:Nx \iow_wrap:nnnN
9548   {
9549     \exp_not:c { \c__msg_text_prefix_tl #1 / #2 }
9550     { \tl_to_str:n {#3} }
9551     { \tl_to_str:n {#4} }
9552     { \tl_to_str:n {#5} }
9553     { \tl_to_str:n {#6} }
9554   }
9555   { } { } \__msg_show_pre_aux:n
9556 }
9557 \cs_new_protected:Npn \__msg_show_pre:nnxxxx #1#2#3#4#5#6
9558 {
9559   \use:x
9560   { \exp_not:n { \__msg_show_pre:nnnnnn {#1} {#2} } {#3} {#4} {#5} {#6} }
9561 }
9562 \cs_generate_variant:Nn \__msg_show_pre:nnnnnn { nnnnnV }
9563 \cs_new_protected:Npn \__msg_show_pre_aux:n
9564 { \bool_if:NTF \g__msg_log_next_bool { \iow_log:n } { \iow_term:n } }

```

(End definition for `__msg_show_pre:nnnnnn` and `__msg_show_pre_aux:n`.)

`__msg_show_variable:NNNnn` The arguments of `__msg_show_variable:NNNnn` are

- The *⟨variable⟩* to be shown as #1.
- An *⟨if-exist⟩* conditional #2 with NTF signature.
- An *⟨if-empty⟩* conditional #3 or other function with NTF signature (sometimes `\use_ii:nnn`).
- The *⟨message⟩* #4 to use.
- A construction #5 which produces the formatted string eventually passed to the `\showtokens` primitive. Typically this is a mapping of the form `\seq_map_function:NN ⟨variable⟩ __msg_show_item:n`.

If *⟨if-exist⟩* *⟨variable⟩* is false, throw an error and remember to reset `\g__msg_log_next_bool`, which is otherwise reset by `__msg_show_wrap:n`. If *⟨message⟩* is not empty, output the message `LaTeX/kernel/show-⟨message⟩` with as its arguments the *⟨variable⟩*, and either an empty second argument or ? depending on the result of *⟨if-empty⟩* *⟨variable⟩*. Afterwards, show the contents of #5 using `__msg_show_wrap:n` or `__msg_log_wrap:n`.

```

9565 \cs_new_protected:Npn \__msg_show_variable:NNNnn #1#2#3#4#5
9566 {
9567   #2 #1
9568   {
9569     \tl_if_empty:nF {#4}
9570     {

```

```

9571         \_msg_show_pre:nxxxxx { LaTeX / kernel } { show- #4 }
9572         { \token_to_str:N #1 } { #3 #1 { } { ? } } { } { }
9573     }
9574     \_msg_show_wrap:n {#5}
9575 }
9576 {
9577     \_msg_kernel_error:nxx { kernel } { variable-not-defined }
9578     { \token_to_str:N #1 }
9579     \bool_gset_false:N \g__msg_log_next_bool
9580 }
9581 }

```

(End definition for `_msg_show_variable:NNNnn`.)

`_msg_show_wrap:Nn` A short-hand used for `\int_show:n` and many other functions that passes to `_msg_show_wrap:n` the result of applying `#1` (a function such as `\int_eval:n`) to the expression `#2`. The leading `>~` is needed by `_msg_show_wrap:n`. The use of x-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`. This does not lead to double expansion because the x-expansion of `#1 {#2}` is a string in all cases where `_msg_show_wrap:Nn` is used.

```

9582 \cs_new_protected:Npn \_msg_show_wrap:Nn #1#2
9583 { \exp_args:Nx \_msg_show_wrap:n { > ~ \tl_to_str:n {#2} = #1 {#2} } }

```

(End definition for `_msg_show_wrap:Nn`.)

`_msg_show_wrap:n` The argument of `_msg_show_wrap:n` is line-wrapped using `\iow_wrap:nnnN`. Everything before the first `>` in the wrapped text is removed, as well as an optional space following it (because of f-expansion). In order for line-wrapping to give the correct result, the first `>` must in fact appear at the beginning of a line and be followed by a space (or a line-break), so in practice, the argument of `_msg_show_wrap:n` begins with `>~` or `\>~`.

The line-wrapped text is then either sent to the log file through `\iow_log:x`, or shown in the terminal using the ε -TeX primitive `\showtokens` after removing a leading `>~` and trailing dot since those are added automatically by `\showtokens`. The trailing dot was included in the first place because its presence can affect line-wrapping. Note that the space after `>` is removed through f-expansion rather than by using an argument delimited by `>~` because the space may have been replaced by a line-break when line-wrapping.

A special case is that if the line-wrapped text is a single dot (in other words if the argument of `_msg_show_wrap:n` x-expands to nothing) then no `>~` should be removed. This makes it unnecessary to check explicitly for emptiness when using for instance `\seq_map_function:NN <seq var> _msg_show_item:n` as the argument of `_msg_show_wrap:n`.

Finally, the token list `\l__msg_internal_tl` containing the result of all these manipulations is displayed to the terminal using `\etex_showtokens:D` and odd `\exp_after:wN` which expand the closing brace to improve the output slightly. The calls to `_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.

Note also that `\g__msg_log_next_bool` is only reset if that is necessary. This allows the user of an interactive prompt to insert tokens as a response to ε -TeX's `\showtokens`.

```

9584 \cs_new_protected:Npn \__msg_show_wrap:n #1
9585 { \iow_wrap:nnnN { #1 . } { } { } \__msg_show_wrap_aux:n }
9586 \cs_new_protected:Npn \__msg_show_wrap_aux:n #1
9587 {
9588   \tl_if_single:nTF {#1}
9589   { \tl_clear:N \l__msg_internal_tl }
9590   { \tl_set:Nf \l__msg_internal_tl { \__msg_show_wrap_aux:w #1 \q_stop } }
9591   \bool_if:NTF \g__msg_log_next_bool
9592   {
9593     \iow_log:x { > ~ \l__msg_internal_tl . }
9594     \bool_gset_false:N \g__msg_log_next_bool
9595   }
9596   {
9597     \__iow_with:Nnn \tex_newlinechar:D { 10 }
9598     {
9599       \__iow_with:Nnn \tex_errorcontextlines:D { - \c_one }
9600       {
9601         \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
9602         { \exp_after:wN \l__msg_internal_tl }
9603       }
9604     }
9605   }
9606 }
9607 \cs_new:Npn \__msg_show_wrap_aux:w #1 > #2 . \q_stop {#2}

```

(End definition for `__msg_show_wrap:n`, `__msg_show_wrap_aux:n`, and `__msg_show_wrap_aux:w`.)

`__msg_show_item:n` Each item in the variable is formatted using one of the following functions.

`__msg_show_item:nn`

`__msg_show_item_unbraced:nn`

```

9608 \cs_new:Npn \__msg_show_item:n #1
9609 {
9610   \ \ > \ \ \{ \tl_to_str:n {#1} \}
9611 }
9612 \cs_new:Npn \__msg_show_item:nn #1#2
9613 {
9614   \ \ > \ \ \{ \tl_to_str:n {#1} \}
9615   \ \ => \ \ \{ \tl_to_str:n {#2} \}
9616 }
9617 \cs_new:Npn \__msg_show_item_unbraced:nn #1#2
9618 {
9619   \ \ > \ \ \tl_to_str:n {#1}
9620   \ \ => \ \ \tl_to_str:n {#2}
9621 }

```

(End definition for `__msg_show_item:n`, `__msg_show_item:nn`, and `__msg_show_item_unbraced:nn`.)

9622 `\</initex | package>`

19 l3keys Implementation

9623 `*initex | package>`

19.1 Low-level interface

The low-level key parser is based heavily on `keyval`, but with a number of additional “safety” requirements and with the idea that the parsed list of key–value pairs can be

processed in a variety of ways. The net result is that this code needs around twice 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.

```
9624 <@@=keyval>
```

`\l__keyval_key_tl` The current key name and value.

```
\l__keyval_value_tl 9625 \tl_new:N \l__keyval_key_tl
9626 \tl_new:N \l__keyval_value_tl
```

(End definition for `\l__keyval_key_tl` and `\l__keyval_value_tl`.)

`\l__keyval_sanitise_tl` A token list variable for dealing with awkward category codes in the input.

```
9627 \tl_new:N \l__keyval_sanitise_tl
```

(End definition for `\l__keyval_sanitise_tl`.)

`\keyval_parse:NNn` The main function starts off by normalising category codes in package mode. That’s relatively “expensive” so is skipped (hopefully) in format mode. We then hand off to the parser. The use of `\q_mark` here prevents loss of braces from the key argument. This particular quark is chosen as it fits in with `__tl_trim_spaces:nn` and allows a performance enhancement as the token can be carried through. Notice that by passing the two processor commands along the input stack we avoid the need to track these at all.

```
9628 \cs_new_protected:Npn \keyval_parse:NNn #1#2#3
9629 {
9630   <*initex>
9631   \__keyval_loop:NNw #1#2 \q_mark #3 , \q_recursion_tail ,
9632   </initex>
9633   <*package>
9634   \tl_set:Nn \l__keyval_sanitise_tl {#3}
9635   \__keyval_sanitise_equals:
9636   \__keyval_sanitise_comma:
9637   \exp_after:wN \__keyval_loop:NNw \exp_after:wN #1 \exp_after:wN #2
9638   \exp_after:wN \q_mark \l__keyval_sanitise_tl , \q_recursion_tail ,
9639   </package>
9640   }
```

(End definition for `\keyval_parse:NNn`. This function is documented on page 172.)

`__keyval_sanitise_equals:` A reasonably fast search and replace set up specifically for the active tokens. The nature of the input is known so everything is hard-coded. With only two tokens to cover, the speed gain from using dedicated functions is worth it.

```
\__keyval_sanitise_comma:
\__keyval_sanitise_equals_auxi:w 9641 <*package>
\__keyval_sanitise_equals_auxii:w 9642 \group_begin:
\__keyval_sanitise_comma_auxi:w 9643 \char_set_catcode_active:n { '\= }
\__keyval_sanitise_comma_auxii:w 9644 \char_set_catcode_active:n { '\, }
\__keyval_sanitise_aux:w 9645 \cs_new_protected:Npn \__keyval_sanitise_equals:
9646 {
9647   \exp_after:wN \__keyval_sanitise_equals_auxi:w \l__keyval_sanitise_tl
9648   \q_mark = \q_nil =
9649   \exp_after:wN \__keyval_sanitise_aux:w \l__keyval_sanitise_tl
9650 }
```

```

9651 \cs_new_protected:Npn \__keyval_sanitise_equals_auxi:w #1 =
9652 {
9653   \tl_set:Nn \l__keyval_sanitise_tl {#1}
9654   \__keyval_sanitise_equals_auxii:w
9655 }
9656 \cs_new_protected:Npn \__keyval_sanitise_equals_auxii:w #1 =
9657 {
9658   \if_meaning:w \q_nil #1 \scan_stop:
9659   \else:
9660     \tl_set:Nx \l__keyval_sanitise_tl
9661     {
9662       \exp_not:o \l__keyval_sanitise_tl
9663       \token_to_str:N =
9664       \exp_not:n {#1}
9665     }
9666     \exp_after:wN \__keyval_sanitise_equals_auxii:w
9667   \fi:
9668 }
9669 \cs_new_protected:Npn \__keyval_sanitise_comma:
9670 {
9671   \exp_after:wN \__keyval_sanitise_comma_auxi:w \l__keyval_sanitise_tl
9672   \q_mark , \q_nil ,
9673   \exp_after:wN \__keyval_sanitise_aux:w \l__keyval_sanitise_tl
9674 }
9675 \cs_new_protected:Npn \__keyval_sanitise_comma_auxi:w #1 ,
9676 {
9677   \tl_set:Nn \l__keyval_sanitise_tl {#1}
9678   \__keyval_sanitise_comma_auxii:w
9679 }
9680 \cs_new_protected:Npn \__keyval_sanitise_comma_auxii:w #1 ,
9681 {
9682   \if_meaning:w \q_nil #1 \scan_stop:
9683   \else:
9684     \tl_set:Nx \l__keyval_sanitise_tl
9685     {
9686       \exp_not:o \l__keyval_sanitise_tl
9687       \token_to_str:N ,
9688       \exp_not:n {#1}
9689     }
9690     \exp_after:wN \__keyval_sanitise_comma_auxii:w
9691   \fi:
9692 }
9693 \group_end:
9694 \cs_new_protected:Npn \__keyval_sanitise_aux:w #1 \q_mark
9695 { \tl_set:Nn \l__keyval_sanitise_tl {#1} }
9696 \endpackage

```

(End definition for __keyval_sanitise_equals: and others.)

__keyval_loop:NNw A fast test for the end of the loop, remembering to remove the leading quark first. Assuming that is not the case, look for a key and value then loop around, re-inserting a leading quark in front of the next position.

```

9697 \cs_new_protected:Npn \__keyval_loop:NNw #1#2#3 ,
9698 {

```

```

9699 \exp_after:wN \if_meaning:w \exp_after:wN \q_recursion_tail
9700 \use_none:n #3 \prg_do_nothing:
9701 \else:
9702 \__keyval_split:NNw #1#2#3 == \q_stop
9703 \exp_after:wN \__keyval_loop:NNw \exp_after:wN #1 \exp_after:wN #2
9704 \exp_after:wN \q_mark
9705 \fi:
9706 }

```

(End definition for __keyval_loop:NNw.)

```

\__keyval_split:NNw
\__keyval_split_value:NNw
\__keyval_split_tidy:w
\__keyval_action:

```

The value is picked up separately from the key so there can be another quark inserted at the front, keeping braces and allowing both parts to share the same code paths. The key is found first then there's a check that there is something there: this is biased to the common case of there actually being a key. For the value, we first need to see if there is anything to do: if there is, extract it. The appropriate action is then inserted in front of the key and value. Doing this using an assignment is marginally faster than an expansion chain.

```

9707 \cs_new_protected:Npn \__keyval_split:NNw #1#2#3 =
9708 {
9709 \__keyval_def:Nn \l__keyval_key_tl {#3}
9710 \if_meaning:w \l__keyval_key_tl \c_empty_tl
9711 \exp_after:wN \__keyval_split_tidy:w
9712 \else:
9713 \exp_after:wN \__keyval_split_value:NNw \exp_after:wN #1 \exp_after:wN #2
9714 \exp_after:wN \q_mark
9715 \fi:
9716 }
9717 \cs_new_protected:Npn \__keyval_split_value:NNw #1#2#3 = #4 \q_stop
9718 {
9719 \if:w \scan_stop: \tl_to_str:n {#4} \scan_stop:
9720 \cs_set:Npx \__keyval_action:
9721 { \exp_not:N #1 { \exp_not:o \l__keyval_key_tl } }
9722 \else:
9723 \if:w \scan_stop: \etex_detokenize:D \exp_after:wN { \use_none:n #4 }
9724 \scan_stop:
9725 \__keyval_def:Nn \l__keyval_value_tl {#3}
9726 \cs_set:Npx \__keyval_action:
9727 {
9728 \exp_not:N #2
9729 { \exp_not:o \l__keyval_key_tl }
9730 { \exp_not:o \l__keyval_value_tl }
9731 }
9732 \else:
9733 \cs_set:Npn \__keyval_action:
9734 { \__msg_kernel_error:nn { kernel } { misplaced-equals-sign } }
9735 \fi:
9736 \fi:
9737 \__keyval_action:
9738 }
9739 \cs_new_protected:Npn \__keyval_split_tidy:w #1 \q_stop
9740 {
9741 \if:w \scan_stop: \etex_detokenize:D \exp_after:wN { \use_none:n #1 }
9742 \scan_stop:

```

```

9743 \else:
9744 \exp_after:wN \__keyval_empty_key:
9745 \fi:
9746 }
9747 \cs_new:Npn \__keyval_action: { }
9748 \cs_new_protected:Npn \__keyval_empty_key:
9749 { \__msg_kernel_error:nn { kernel } { misplaced-equals-sign } }

```

(End definition for __keyval_split:NNw and others.)

__keyval_def:Nn First trim spaces off, then potentially remove a set of braces. By using the internal interface __tl_trim_spaces:nn we can take advantage of the fact it needs a leading \q_mark in this process. The \exp_after:wN removes the quark, the delimited argument deals with any braces.

```

9750 \cs_new_protected:Npn \__keyval_def:Nn #1#2
9751 { \tl_set:Nx #1 { \__tl_trim_spaces:nn {#2} \__keyval_def_aux:n } }
9752 \cs_new:Npn \__keyval_def_aux:n #1
9753 { \exp_after:wN \__keyval_def_aux:w #1 \q_stop }
9754 \cs_new:Npn \__keyval_def_aux:w #1 \q_stop { \exp_not:n {#1} }

```

(End definition for __keyval_def:Nn, __keyval_def_aux:n, and __keyval_def_aux:w.)

One message for the low level parsing system.

```

9755 \__msg_kernel_new:nnnn { kernel } { misplaced-equals-sign }
9756 { Misplaced-equals-sign-in~key-value-input~\msg_line_number: }
9757 {
9758 LaTeX-is-attempting-to-parse-some-key-value-input-but-found~
9759 two-equals-signs-not-separated-by-a-comma.
9760 }

```

19.2 Constants and variables

```

9761 <@@=keys>

```

Various storage areas for the different data which make up keys.

```

\c__keys_code_root_tl
\c__keys_default_root_tl
\c__keys_groups_root_tl
\c__keys_inherit_root_tl
\c__keys_type_root_tl
\c__keys_validate_root_tl
9762 \tl_const:Nn \c__keys_code_root_tl { key~code~>~ }
9763 \tl_const:Nn \c__keys_default_root_tl { key~default~>~ }
9764 \tl_const:Nn \c__keys_groups_root_tl { key~groups~>~ }
9765 \tl_const:Nn \c__keys_inherit_root_tl { key~inherit~>~ }
9766 \tl_const:Nn \c__keys_type_root_tl { key~type~>~ }
9767 \tl_const:Nn \c__keys_validate_root_tl { key~validate~>~ }

```

(End definition for \c__keys_code_root_tl and others.)

\c__keys_props_root_tl The prefix for storing properties.

```

9768 \tl_const:Nn \c__keys_props_root_tl { key~prop~>~ }

```

(End definition for \c__keys_props_root_tl.)

\l_keys_choice_int Publicly accessible data on which choice is being used when several are generated as a set.

```

9769 \int_new:N \l_keys_choice_int
9770 \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 167.)

`\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.

```
9771 \clist_new:N \l__keys_groups_clist
```

(End definition for \l__keys_groups_clist.)

`\l_keys_key_tl` The name of a key itself: needed when setting keys.

```
9772 \tl_new:N \l_keys_key_tl
```

(End definition for \l_keys_key_tl. This variable is documented on page 169.)

`\l__keys_module_tl` The module for an entire set of keys.

```
9773 \tl_new:N \l__keys_module_tl
```

(End definition for \l__keys_module_tl.)

`\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.

```
9774 \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.

```
9775 \bool_new:N \l__keys_only_known_bool
```

(End definition for \l__keys_only_known_bool.)

`\l_keys_path_tl` The “path” of the current key is stored here: this is available to the programmer and so is public.

```
9776 \tl_new:N \l_keys_path_tl
```

(End definition for \l_keys_path_tl. This variable is documented on page 169.)

`\l__keys_property_tl` The “property” begin set for a key at definition time is stored here.

```
9777 \tl_new:N \l__keys_property_tl
```

(End definition for \l__keys_property_tl.)

`\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”).

```
9778 \bool_new:N \l__keys_selective_bool
```

```
9779 \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.

```
9780 \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.

```
9781 \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.
 9782 `\tl_new:N \l_keys_value_tl`
 (End definition for `\l_keys_value_tl`. This variable is documented on page 169.)

`\l__keys_tmp_bool` Scratch space.
 9783 `\bool_new:N \l__keys_tmp_bool`
 (End definition for `\l__keys_tmp_bool`.)

19.3 The key defining mechanism

`\keys_define:nn` 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).
`__keys_define:nnn`
`__keys_define:onn`

```

9784 \cs_new_protected:Npn \keys_define:nn
9785 { \__keys_define:onn \l__keys_module_tl }
9786 \cs_new_protected:Npn \__keys_define:nnn #1#2#3
9787 {
9788   \tl_set:Nx \l__keys_module_tl { \__keys_remove_spaces:n {#2} }
9789   \keyval_parse:NNn \__keys_define:n \__keys_define:nn {#3}
9790   \tl_set:Nn \l__keys_module_tl {#1}
9791 }
9792 \cs_generate_variant:Nn \__keys_define:nnn { o }

```

(End definition for `\keys_define:nn` and `__keys_define:nnn`. These functions are documented on page 162.)

`__keys_define:n` 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.
`__keys_define:nn`
`__keys_define_aux:nn`

```

9793 \cs_new_protected:Npn \__keys_define:n #1
9794 {
9795   \bool_set_true:N \l__keys_no_value_bool
9796   \__keys_define_aux:nn {#1} { }
9797 }
9798 \cs_new_protected:Npn \__keys_define:nn #1#2
9799 {
9800   \bool_set_false:N \l__keys_no_value_bool
9801   \__keys_define_aux:nn {#1} {#2}
9802 }
9803 \cs_new_protected:Npn \__keys_define_aux:nn #1#2
9804 {
9805   \__keys_property_find:n {#1}
9806   \cs_if_exist:cTF { \c__keys_props_root_tl \l__keys_property_tl }
9807   { \__keys_define_code:n {#2} }
9808   {
9809     \tl_if_empty:NF \l__keys_property_tl
9810     {
9811       \__msg_kernel_error:nxx { kernel } { property-unknown }
9812       { \l__keys_property_tl } { \l_keys_path_tl }
9813     }
9814   }
9815 }

```

(End definition for `__keys_define:n`, `__keys_define:nn`, and `__keys_define_aux:nn`.)

`__keys_property_find:n` 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.

```

9816 \cs_new_protected:Npn \__keys_property_find:n #1
9817 {
9818   \tl_set:Nx \l__keys_property_tl { \__keys_remove_spaces:n {#1} }
9819   \exp_after:wN \__keys_property_find:w \l__keys_property_tl . . \q_stop {#1}
9820 }
9821 \cs_new_protected:Npn \__keys_property_find:w #1 . #2 . #3 \q_stop #4
9822 {
9823   \tl_if_blank:nTF {#3}
9824   {
9825     \tl_clear:N \l__keys_property_tl
9826     \__msg_kernel_error:nnn { kernel } { key-no-property } {#4}
9827   }
9828   {
9829     \str_if_eq:nnTF {#3} { . }
9830     {
9831       \tl_set:Nx \l_keys_path_tl
9832       {
9833         \tl_if_empty:NF \l__keys_module_tl
9834         { \l__keys_module_tl / }
9835         #1
9836       }
9837       \tl_set:Nn \l__keys_property_tl { . #2 }
9838     }
9839     {
9840       \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / #1 . #2 }
9841       \__keys_property_search:w #3 \q_stop
9842     }
9843   }
9844 }
9845 \cs_new_protected:Npn \__keys_property_search:w #1 . #2 \q_stop
9846 {
9847   \str_if_eq:nnTF {#2} { . }
9848   {
9849     \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl }
9850     \tl_set:Nn \l__keys_property_tl { . #1 }
9851   }
9852   {
9853     \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl . #1 }
9854     \__keys_property_search:w #2 \q_stop
9855   }
9856 }

```

(End definition for `__keys_property_find:n` and `__keys_property_find:w`.)

`__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 is missing the earlier tests will have failed.

```

9857 \cs_new_protected:Npn \__keys_define_code:n #1

```

```

9858 {
9859   \bool_if:NTF \l__keys_no_value_bool
9860   {
9861     \exp_after:wN \__keys_define_code:w
9862     \l__keys_property_tl \q_stop
9863     { \use:c { \c__keys_props_root_tl \l__keys_property_tl } }
9864     {
9865       \__msg_kernel_error:nxxx { kernel }
9866       { property-requires-value } { \l__keys_property_tl }
9867       { \l_keys_path_tl }
9868     }
9869   }
9870   { \use:c { \c__keys_props_root_tl \l__keys_property_tl } {#1} }
9871 }
9872 \use:x
9873 {
9874   \cs_new:Npn \exp_not:N \__keys_define_code:w
9875     ##1 \c_colon_str ##2 \exp_not:N \q_stop
9876 }
9877 { \tl_if_empty:nTF {#2} }

```

(End definition for __keys_define_code:n and __keys_define_code:w.)

19.4 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.

```

9878 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
9879 {
9880   \bool_if_exist:NF #1 { \bool_new:N #1 }
9881   \__keys_choice_make:
9882   \__keys_cmd_set:nx { \l_keys_path_tl / true }
9883     { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
9884   \__keys_cmd_set:nx { \l_keys_path_tl / false }
9885     { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
9886   \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
9887   {
9888     \__msg_kernel_error:nxx { kernel } { boolean-values-only }
9889     { \l_keys_key_tl }
9890   }
9891   \__keys_default_set:n { true }
9892 }
9893 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }

```

(End definition for __keys_bool_set:Nn.)

__keys_bool_set_inverse:Nn Inverse boolean setting is much the same.

```

9894 \cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
9895 {
9896   \bool_if_exist:NF #1 { \bool_new:N #1 }
9897   \__keys_choice_make:
9898   \__keys_cmd_set:nx { \l_keys_path_tl / true }
9899     { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
9900   \__keys_cmd_set:nx { \l_keys_path_tl / false }

```

```

9901     { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
9902 \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
9903 {
9904     \__msg_kernel_error:nxx { kernel } { boolean-values-only }
9905     { \l_keys_key_tl }
9906 }
9907 \__keys_default_set:n { true }
9908 }
9909 \cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }

```

(End definition for `__keys_bool_set_inverse:Nn`.)

`__keys_choice_make:` To make a choice from a key, two steps: set the code, and set the unknown key. As
`__keys_multichoice_make:` multichoice and choices are essentially the same bar one function, the code is given
`__keys_choice_make:N` together.
`__keys_choice_make_aux:N`

```

9910 \cs_new_protected:Npn \__keys_choice_make:
9911 { \__keys_choice_make:N \__keys_choice_find:n }
9912 \cs_new_protected:Npn \__keys_multichoice_make:
9913 { \__keys_choice_make:N \__keys_multichoice_find:n }
9914 \cs_new_protected:Npn \__keys_choice_make:N #1
9915 {
9916     \cs_if_exist:cTF
9917     { \c__keys_type_root_tl \__keys_parent:o \l_keys_path_tl }
9918     {
9919         \str_if_eq_x:nnTF
9920         { \exp_not:v { \c__keys_type_root_tl \__keys_parent:o \l_keys_path_tl } }
9921         { choice }
9922         {
9923             \__msg_kernel_error:nxxx { kernel } { nested-choice-key }
9924             { \l_keys_path_tl } { \__keys_parent:o \l_keys_path_tl }
9925         }
9926         { \__keys_choice_make_aux:N #1 }
9927     }
9928     { \__keys_choice_make_aux:N #1 }
9929 }
9930 \cs_new_protected:Npn \__keys_choice_make_aux:N #1
9931 {
9932     \cs_set_nopar:cpn { \c__keys_type_root_tl \l_keys_path_tl } { choice }
9933     \__keys_cmd_set:nn { \l_keys_path_tl } { #1 {##1} }
9934     \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
9935     {
9936         \__msg_kernel_error:nxxx { kernel } { key-choice-unknown }
9937         { \l_keys_path_tl } {##1}
9938     }
9939 }

```

(End definition for `__keys_choice_make:` and others.)

`__keys_choices_make:nn` Auto-generating choices means setting up the root key as a choice, then defining each
`__keys_multichoice_make:nn` choice in turn.
`__keys_choices_make:Nnn`

```

9940 \cs_new_protected:Npn \__keys_choices_make:nn
9941 { \__keys_choices_make:Nnn \__keys_choice_make: }
9942 \cs_new_protected:Npn \__keys_multichoice_make:nn
9943 { \__keys_choices_make:Nnn \__keys_multichoice_make: }

```

```

9944 \cs_new_protected:Npn \__keys_choices_make:Nnn #1#2#3
9945 {
9946   #1
9947   \int_zero:N \l_keys_choice_int
9948   \clist_map_inline:nn {#2}
9949   {
9950     \int_incr:N \l_keys_choice_int
9951     \__keys_cmd_set:nx { \l_keys_path_tl / \__keys_remove_spaces:n {##1} }
9952     {
9953       \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
9954       \int_set:Nn \exp_not:N \l_keys_choice_int
9955       { \int_use:N \l_keys_choice_int }
9956       \exp_not:n {#3}
9957     }
9958   }
9959 }

```

(End definition for __keys_choices_make:nn, __keys_multichoices_make:nn, and __keys_choices_make:Nnn.)

__keys_cmd_set:nn Setting the code for a key first checks that the basic data structures exist, then saves the code.

```

\__keys_cmd_set:nx
\__keys_cmd_set:Vn
\__keys_cmd_set:Vo
9960 \cs_new_protected:Npn \__keys_cmd_set:nn #1#2
9961 {
9962   \cs_if_exist:cF { \c__keys_code_root_tl #1 }
9963   { \__chk_log:x { Defining-key-#1~\msg_line_context: } }
9964   \cs_set_protected:cpn { \c__keys_code_root_tl #1 } ##1 {#2}
9965 }
9966 \cs_generate_variant:Nn \__keys_cmd_set:nn { nx , Vn , Vo }

```

(End definition for __keys_cmd_set:nn.)

__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.

```

9967 \cs_new_protected:Npn \__keys_default_set:n #1
9968 {
9969   \tl_if_empty:nTF {#1}
9970   {
9971     \cs_set_eq:cN
9972     { \c__keys_default_root_tl \l_keys_path_tl }
9973     \tex_undefined:D
9974   }
9975   {
9976     \cs_set:cpx
9977     { \c__keys_default_root_tl \l_keys_path_tl }
9978     { \exp_not:n {#1} }
9979   }
9980 }

```

(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.

```

9981 \cs_new_protected:Npn \__keys_groups_set:n #1
9982 {
9983   \clist_set:Nn \l__keys_groups_clist {#1}
9984   \clist_if_empty:NTF \l__keys_groups_clist
9985   {
9986     \cs_set_eq:cN { \c__keys_groups_root_tl \l_keys_path_tl }
9987     \tex_undefined:D
9988   }
9989   {
9990     \clist_set_eq:cN { \c__keys_groups_root_tl \l_keys_path_tl }
9991     \l__keys_groups_clist
9992   }
9993 }

```

(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.

```

9994 \cs_new_protected:Npn \__keys_inherit:n #1
9995 {
9996   \__keys_undefine:
9997   \cs_set_nopar:cpn { \c__keys_inherit_root_tl \l_keys_path_tl } {#1}
9998 }

```

(End definition for __keys_inherit:n.)

__keys_initialise:n A set up for initialisation: just run the code if it exists.

```

9999 \cs_new_protected:Npn \__keys_initialise:n #1
10000 {
10001   \cs_if_exist_use:cT { \c__keys_code_root_tl \l_keys_path_tl } { {#1} }
10002 }

```

(End definition for __keys_initialise:n.)

__keys_meta_make:n To create a meta-key, simply set up to pass data through.

```

\__keys_meta_make:nn
10003 \cs_new_protected:Npn \__keys_meta_make:n #1
10004 {
10005   \__keys_cmd_set:Vo \l_keys_path_tl
10006   {
10007     \exp_after:wN \keys_set:nn
10008     \exp_after:wN { \l__keys_module_tl } {#1}
10009   }
10010 }
10011 \cs_new_protected:Npn \__keys_meta_make:nn #1#2
10012 { \__keys_cmd_set:Vn \l_keys_path_tl { \keys_set:nn {#1} {#2} } }

```

(End definition for __keys_meta_make:n and __keys_meta_make:nn.)

__keys_undefine: Undefined a key has to be done without `\cs_undefine:c` as that function acts globally.

```

10013 \cs_new_protected:Npn \__keys_undefine:
10014 {
10015   \clist_map_inline:nn
10016   { code , default , groups , inherit , type , validate }
10017   {
10018     \cs_set_eq:cN
10019     { \tl_use:c { c__keys_ ##1 _root_tl } \l_keys_path_tl }

```

```

10020         \tex_undefined:D
10021     }
10022 }

```

(End definition for _keys_undefine:.)

_keys_value_requirement:nn 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.

```

10023 \cs_new_protected:Npn \_keys_value_requirement:nn #1#2
10024 {
10025     \str_case:nnF {#2}
10026     {
10027         { true }
10028         {
10029             \cs_set_eq:cc
10030             { \c__keys_validate_root_tl \l_keys_path_tl }
10031             { __keys_validate_ #1 : }
10032         }
10033         { false }
10034         {
10035             \cs_if_eq:ccT
10036             { \c__keys_validate_root_tl \l_keys_path_tl }
10037             { __keys_validate_ #1 : }
10038             {
10039                 \cs_set_eq:cN
10040                 { \c__keys_validate_root_tl \l_keys_path_tl }
10041                 \tex_undefined:D
10042             }
10043         }
10044     }
10045     {
10046         \_msg_kernel_error:nnx { kernel } { property-boolean-values-only }
10047         { .value_ #1 :n }
10048     }
10049 }
10050 \cs_new_protected:Npn \_keys_validate_forbidden:
10051 {
10052     \bool_if:NF \l__keys_no_value_bool
10053     {
10054         \_msg_kernel_error:nnxx { kernel } { value-forbidden }
10055         { \l_keys_path_tl } { \l_keys_value_tl }
10056         \_keys_validate_cleanup:w
10057     }
10058 }
10059 \cs_new_protected:Npn \_keys_validate_required:
10060 {
10061     \bool_if:NT \l__keys_no_value_bool
10062     {
10063         \_msg_kernel_error:nnx { kernel } { value-required }
10064         { \l_keys_path_tl }
10065         \_keys_validate_cleanup:w
10066     }

```



```

10067     }
10068 \cs_new_protected:Npn \__keys_validate_cleanup:w #1 \cs_end: #2#3 { }

```

(End definition for `__keys_value_requirement:nn` and others.)

`__keys_variable_set:NnnN` Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed.

`__keys_variable_set:cnN`

```

10069 \cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
10070 {
10071     \use:c { #2_if_exist:NF } #1 { \use:c { #2_new:N } #1 }
10072     \__keys_cmd_set:nx { \l_keys_path_tl }
10073     {
10074         \exp_not:c { #2 _ #3 set:N #4 }
10075         \exp_not:N #1
10076         \exp_not:n { {#1} }
10077     }
10078 }
10079 \cs_generate_variant:Nn \__keys_variable_set:NnnN { c }

```

(End definition for `__keys_variable_set:NnnN`.)

19.5 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.
.bool_set:c 10080 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:N } #1
.bool_gset:N 10081 { \__keys_bool_set:Nn #1 { } }
.bool_gset:c 10082 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:c } #1
10083 { \__keys_bool_set:cn {#1} { } }
10084 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:N } #1
10085 { \__keys_bool_set:Nn #1 { g } }
10086 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:c } #1
10087 { \__keys_bool_set:cn {#1} { g } }

```

(End definition for `.bool_set:N` and `.bool_gset:N`. These functions are documented on page 163.)

```

.bool_set_inverse:N One function for this.
.bool_set_inverse:c 10088 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:N } #1
.bool_gset_inverse:N 10089 { \__keys_bool_set_inverse:Nn #1 { } }
.bool_gset_inverse:c 10090 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:c } #1
10091 { \__keys_bool_set_inverse:cn {#1} { } }
10092 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:N } #1
10093 { \__keys_bool_set_inverse:Nn #1 { g } }
10094 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:c } #1
10095 { \__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 163.)

.choice: Making a choice is handled internally, as it is also needed by **.generate_choices:n**.

```
10096 \cs_new_protected:cpn { \c__keys_props_root_tl .choice: }
10097   { \__keys_choice_make: }
```

(End definition for **.choice:**. This function is documented on page 163.)

.choices:nn For auto-generation of a series of mutually-exclusive choices. Here, #1 will consist of two separate arguments, hence the slightly odd-looking implementation.

```
.choices:Vn
.choices:on
.choices:xn
10098 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:nn } #1
10099   { \__keys_choices_make:nn #1 }
10100 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:Vn } #1
10101   { \exp_args:NV \__keys_choices_make:nn #1 }
10102 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:on } #1
10103   { \exp_args:No \__keys_choices_make:nn #1 }
10104 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:xn } #1
10105   { \exp_args:Nx \__keys_choices_make:nn #1 }
```

(End definition for **.choices:nn**. This function is documented on page 163.)

.code:n Creating code is simply a case of passing through to the underlying set function.

```
10106 \cs_new_protected:cpn { \c__keys_props_root_tl .code:n } #1
10107   { \__keys_cmd_set:nn { \l_keys_path_tl } {#1} }
```

(End definition for **.code:n**. This function is documented on page 163.)

.clist_set:N

```
.clist_set:c
.clist_gset:N
.clist_gset:c
10108 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:N } #1
10109   { \__keys_variable_set:NnnN #1 { clist } { } n }
10110 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:c } #1
10111   { \__keys_variable_set:cnN {#1} { clist } { } n }
10112 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:N } #1
10113   { \__keys_variable_set:NnnN #1 { clist } { g } n }
10114 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:c } #1
10115   { \__keys_variable_set:cnN {#1} { clist } { g } n }
```

(End definition for **.clist_set:N** and **.clist_gset:N**. These functions are documented on page 163.)

.default:n Expansion is left to the internal functions.

```
.default:V
.default:o
.default:x
10116 \cs_new_protected:cpn { \c__keys_props_root_tl .default:n } #1
10117   { \__keys_default_set:n {#1} }
10118 \cs_new_protected:cpn { \c__keys_props_root_tl .default:V } #1
10119   { \exp_args:NV \__keys_default_set:n #1 }
10120 \cs_new_protected:cpn { \c__keys_props_root_tl .default:o } #1
10121   { \exp_args:No \__keys_default_set:n {#1} }
10122 \cs_new_protected:cpn { \c__keys_props_root_tl .default:x } #1
10123   { \exp_args:Nx \__keys_default_set:n {#1} }
```

(End definition for **.default:n**. This function is documented on page 164.)

.dim_set:N Setting a variable is very easy: just pass the data along.

```
.dim_set:c
.dim_gset:N
.dim_gset:c
10124 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:N } #1
10125   { \__keys_variable_set:NnnN #1 { dim } { } n }
10126 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:c } #1
10127   { \__keys_variable_set:cnN {#1} { dim } { } n }
10128 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:N } #1
```

```

10129 { \__keys_variable_set:NnnN #1 { dim } { g } n }
10130 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
10131 { \__keys_variable_set:cnnN {#1} { dim } { g } n }

```

(End definition for `.dim_set:N` and `.dim_gset:N`. These functions are documented on page 164.)

.fp_set:N Setting a variable is very easy: just pass the data along.

```

10132 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:N } #1
10133 { \__keys_variable_set:NnnN #1 { fp } { } n }
.fp_gset:N
10134 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
10135 { \__keys_variable_set:cnnN {#1} { fp } { } n }
10136 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:N } #1
10137 { \__keys_variable_set:NnnN #1 { fp } { g } n }
10138 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
10139 { \__keys_variable_set:cnnN {#1} { fp } { g } n }

```

(End definition for `.fp_set:N` and `.fp_gset:N`. These functions are documented on page 164.)

.groups:n A single property to create groups of keys.

```

10140 \cs_new_protected:cpn { \c__keys_props_root_tl .groups:n } #1
10141 { \__keys_groups_set:n {#1} }

```

(End definition for `.groups:n`. This function is documented on page 164.)

.inherit:n Nothing complex: only one variant at the moment!

```

10142 \cs_new_protected:cpn { \c__keys_props_root_tl .inherit:n } #1
10143 { \__keys_inherit:n {#1} }

```

(End definition for `.inherit:n`. This function is documented on page 164.)

.initial:n The standard hand-off approach.

```

10144 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:n } #1
10145 { \__keys_initialise:n {#1} }
.initial:V
10146 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:V } #1
10147 { \exp_args:NV \__keys_initialise:n #1 }
.initial:o
10148 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:o } #1
10149 { \exp_args:No \__keys_initialise:n {#1} }
.initial:x
10150 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
10151 { \exp_args:Nx \__keys_initialise:n {#1} }

```

(End definition for `.initial:n`. This function is documented on page 165.)

.int_set:N Setting a variable is very easy: just pass the data along.

```

10152 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:N } #1
10153 { \__keys_variable_set:NnnN #1 { int } { } n }
.int_gset:N
10154 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
10155 { \__keys_variable_set:cnnN {#1} { int } { } n }
10156 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:N } #1
10157 { \__keys_variable_set:NnnN #1 { int } { g } n }
10158 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
10159 { \__keys_variable_set:cnnN {#1} { int } { g } n }

```

(End definition for `.int_set:N` and `.int_gset:N`. These functions are documented on page 165.)

.meta:n Making a meta is handled internally.

```
10160 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:n } #1
10161 { \__keys_meta_make:n {#1} }
```

(End definition for .meta:n. This function is documented on page 165.)

.meta:nn Meta with path: potentially lots of variants, but for the moment no so many defined.

```
10162 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:nn } #1
10163 { \__keys_meta_make:nn #1 }
```

(End definition for .meta:nn. This function is documented on page 165.)

.multichoice: The same idea as .choice: and .choices:nn, but where more than one choice is allowed.

```
.multichoices:nn 10164 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoice: }
10165 { \__keys_multichoice_make: }
.multichoices:Vn 10166 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:nn } #1
10167 { \__keys_multichoices_make:nn #1 }
.multichoices:on 10168 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:Vn } #1
10169 { \exp_args:NV \__keys_multichoices_make:nn #1 }
10170 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:on } #1
10171 { \exp_args:No \__keys_multichoices_make:nn #1 }
10172 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:xn } #1
10173 { \exp_args:Nx \__keys_multichoices_make:nn #1 }
```

(End definition for .multichoice: and .multichoices:nn. These functions are documented on page 165.)

.skip_set:N Setting a variable is very easy: just pass the data along.

```
.skip_set:c 10174 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:N } #1
10175 { \__keys_variable_set:NnnN #1 { skip } { } n }
.skip_gset:N 10176 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:c } #1
10177 { \__keys_variable_set:cnnN {#1} { skip } { } n }
10178 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:N } #1
10179 { \__keys_variable_set:NnnN #1 { skip } { g } n }
10180 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:c } #1
10181 { \__keys_variable_set:cnnN {#1} { skip } { g } n }
```

(End definition for .skip_set:N and .skip_gset:N. These functions are documented on page 165.)

.tl_set:N Setting a variable is very easy: just pass the data along.

```
.tl_set:c 10182 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:N } #1
10183 { \__keys_variable_set:NnnN #1 { tl } { } n }
.tl_gset:N 10184 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:c } #1
10185 { \__keys_variable_set:cnnN {#1} { tl } { } n }
.tl_set_x:N 10186 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:N } #1
10187 { \__keys_variable_set:NnnN #1 { tl } { } x }
.tl_set_x:c 10188 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:c } #1
10189 { \__keys_variable_set:cnnN {#1} { tl } { } x }
.tl_gset_x:N 10190 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:N } #1
10191 { \__keys_variable_set:NnnN #1 { tl } { g } n }
10192 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:c } #1
10193 { \__keys_variable_set:cnnN {#1} { tl } { g } n }
10194 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:N } #1
10195 { \__keys_variable_set:NnnN #1 { tl } { g } x }
10196 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:c } #1
10197 { \__keys_variable_set:cnnN {#1} { tl } { g } x }
```

(End definition for `.tl_set:N` and others. These functions are documented on page 165.)

`.undefine:` Another simple wrapper.

```
10198 \cs_new_protected:cpn { \c__keys_props_root_tl .undefine: }
10199   { \__keys_undefine: }
```

(End definition for `.undefine:`. This function is documented on page 166.)

`.value_forbidden:n` These are very similar, so both call the same function.

```
.value_required:n
10200 \cs_new_protected:cpn { \c__keys_props_root_tl .value_forbidden:n } #1
10201   { \__keys_value_requirement:nn { forbidden } {#1} }
10202 \cs_new_protected:cpn { \c__keys_props_root_tl .value_required:n } #1
10203   { \__keys_value_requirement:nn { required } {#1} }
```

(End definition for `.value_forbidden:n` and `.value_required:n`. These functions are documented on page 166.)

19.6 Setting keys

`\keys_set:nn` A simple wrapper again.

```
\keys_set:nV 10204 \cs_new_protected:Npn \keys_set:nn
\keys_set:nv 10205   { \__keys_set:onn { \l__keys_module_tl } }
\keys_set:no 10206 \cs_new_protected:Npn \__keys_set:nnn #1#2#3
\__keys_set:nnn 10207   {
\__keys_set:onn 10208     \tl_set:Nx \l__keys_module_tl { \__keys_remove_spaces:n {#2} }
10209     \keyval_parse:NNn \__keys_set:n \__keys_set:nn {#3}
10210     \tl_set:Nn \l__keys_module_tl {#1}
10211   }
10212 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
10213 \cs_generate_variant:Nn \__keys_set:nnn { o }
```

(End definition for `\keys_set:nn` and `__keys_set:nnn`. These functions are documented on page 169.)

`\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:nVN 10214 \cs_new_protected:Npn \keys_set_known:nnN
\keys_set_known:nvN 10215   { \__keys_set_known:onnN \l__keys_unused_clist }
\keys_set_known:noN 10216 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
\__keys_set_known:nnnN 10217 \cs_new_protected:Npn \__keys_set_known:nnnN #1#2#3#4
\__keys_set_known:onnN 10218   {
\keys_set_known:nV 10219     \clist_clear:N \l__keys_unused_clist
\keys_set_known:nv 10220     \keys_set_known:nn {#2} {#3}
\keys_set_known:no 10221     \tl_set:Nx #4 { \exp_not:o { \l__keys_unused_clist } }
10222     \tl_set:Nn \l__keys_unused_clist {#1}
10223   }
10224 \cs_generate_variant:Nn \__keys_set_known:nnnN { o }
10225 \cs_new_protected:Npn \keys_set_known:nn #1#2
10226   {
10227     \bool_set_true:N \l__keys_only_known_bool
10228     \keys_set:nn {#1} {#2}
10229     \bool_set_false:N \l__keys_only_known_bool
10230   }
10231 \cs_generate_variant:Nn \keys_set_known:nn { nV , nv , no }
```

(End definition for \keys_set_known:nnN, _keys_set_known:nnnN, and \keys_set_known:nn. These functions are documented on page 170.)

```

\keys_set_filter:nnnN
\keys_set_filter:nnVN
\keys_set_filter:nnvN
\keys_set_filter:nnoN
\_keys_set_filter:nnnnN
\_keys_set_filter:nnnnN
\keys_set_filter:nnn
\keys_set_filter:nnV
\keys_set_filter:nnv
\keys_set_filter:nno
\keys_set_groups:nnn
\keys_set_groups:nnV
\keys_set_groups:nnv
\keys_set_groups:nno
10232 \cs_new_protected:Npn \keys_set_filter:nnnN
10233 { \_keys_set_filter:nnnnN \l__keys_unused_clist }
10234 \cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
10235 \cs_new_protected:Npn \_keys_set_filter:nnnnN #1#2#3#4#5
10236 {
10237   \clist_clear:N \l__keys_unused_clist
10238   \keys_set_filter:nnn {#2} {#3} {#4}
10239   \tl_set:Nx #5 { \exp_not:o { \l__keys_unused_clist } }
10240   \tl_set:Nn \l__keys_unused_clist {#1}
10241 }
10242 \cs_generate_variant:Nn \_keys_set_filter:nnnnN { o }
10243 \cs_new_protected:Npn \keys_set_filter:nnn #1#2#3
10244 {
10245   \bool_set_true:N \l__keys_selective_bool
10246   \bool_set_true:N \l__keys_filtered_bool
10247   \seq_set_from_clist:Nn \l__keys_selective_seq {#2}
10248   \keys_set:nn {#1} {#3}
10249   \bool_set_false:N \l__keys_selective_bool
10250 }
10251 \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
10252 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
10253 {
10254   \bool_set_true:N \l__keys_selective_bool
10255   \bool_set_false:N \l__keys_filtered_bool
10256   \seq_set_from_clist:Nn \l__keys_selective_seq {#2}
10257   \keys_set:nn {#1} {#3}
10258   \bool_set_false:N \l__keys_selective_bool
10259 }
10260 \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }

```

(End definition for \keys_set_filter:nnnnN and others. These functions are documented on page 171.)

```

\_keys_set:n
\_keys_set:nn
\_keys_set_aux:nnn
\_keys_set_aux:onn
\_keys_find_key_module:w
\_keys_set_aux:
\_keys_set_selective:
10261 \cs_new_protected:Npn \_keys_set:n #1
10262 {
10263   \bool_set_true:N \l__keys_no_value_bool
10264   \_keys_set_aux:onn \l__keys_module_tl {#1} { }
10265 }
10266 \cs_new_protected:Npn \_keys_set:nn #1#2
10267 {
10268   \bool_set_false:N \l__keys_no_value_bool
10269   \_keys_set_aux:onn \l__keys_module_tl {#1} {#2}
10270 }

```

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.

```

10271 \cs_new_protected:Npn \__keys_set_aux:nnn #1#2#3
10272 {
10273   \tl_set:Nx \l_keys_path_tl
10274   {
10275     \tl_if_blank:nF {#1}
10276     { #1 / }
10277     \__keys_remove_spaces:n {#2}
10278   }
10279   \tl_clear:N \l__keys_module_tl
10280   \exp_after:wN \__keys_find_key_module:w \l_keys_path_tl / \q_stop
10281   \__keys_value_or_default:n {#3}
10282   \bool_if:NTF \l_keys_selective_bool
10283   { \__keys_set_selective: }
10284   { \__keys_execute: }
10285   \tl_set:Nn \l__keys_module_tl {#1}
10286 }
10287 \cs_generate_variant:Nn \__keys_set_aux:nnn { o }
10288 \cs_new_protected:Npn \__keys_find_key_module:w #1 / #2 \q_stop
10289 {
10290   \tl_if_blank:nTF {#2}
10291   { \tl_set:Nn \l_keys_key_tl {#1} }
10292   {
10293     \tl_put_right:Nx \l__keys_module_tl
10294     {
10295       \tl_if_empty:NF \l__keys_module_tl { / }
10296       #1
10297     }
10298     \__keys_find_key_module:w #2 \q_stop
10299   }
10300 }

```

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.

```

10301 \cs_new_protected:Npn \__keys_set_selective:
10302 {
10303   \cs_if_exist:cTF { \c__keys_groups_root_tl \l_keys_path_tl }
10304   {
10305     \clist_set_eq:Nc \l__keys_groups_clist
10306     { \c__keys_groups_root_tl \l_keys_path_tl }
10307     \__keys_check_groups:
10308   }
10309   {
10310     \bool_if:NTF \l_keys_filtered_bool
10311     { \__keys_execute: }
10312     { \__keys_store_unused: }
10313   }
10314 }

```

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.

```

10315 \cs_new_protected:Npn \__keys_check_groups:
10316 {
10317   \bool_set_false:N \l__keys_tmp_bool
10318   \seq_map_inline:Nn \l__keys_selective_seq
10319   {
10320     \clist_map_inline:Nn \l__keys_groups_clist
10321     {
10322       \str_if_eq:nnT {##1} {####1}
10323       {
10324         \bool_set_true:N \l__keys_tmp_bool
10325         \clist_map_break:n { \seq_map_break: }
10326       }
10327     }
10328   }
10329   \bool_if:NTF \l__keys_tmp_bool
10330   {
10331     \bool_if:NTF \l__keys_filtered_bool
10332     { \__keys_store_unused: }
10333     { \__keys_execute: }
10334   }
10335   {
10336     \bool_if:NTF \l__keys_filtered_bool
10337     { \__keys_execute: }
10338     { \__keys_store_unused: }
10339   }
10340 }

```

(End definition for __keys_set:n and others.)

__keys_value_or_default:n If a value is given, return it as #1, otherwise send a default if available.

```

10341 \cs_new_protected:Npn \__keys_value_or_default:n #1
10342 {
10343   \bool_if:NTF \l__keys_no_value_bool
10344   {
10345     \cs_if_exist:cTF { \c__keys_default_root_tl \l_keys_path_tl }
10346     {
10347       \tl_set_eq:Nc
10348       \l_keys_value_tl
10349       { \c__keys_default_root_tl \l_keys_path_tl }
10350     }
10351     { \tl_clear:N \l_keys_value_tl }
10352   }
10353   { \tl_set:Nn \l_keys_value_tl {#1} }
10354 }

```

(End definition for __keys_value_or_default:n.)

__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.

```

10355 \cs_new_protected:Npn \__keys_execute:
10356 {
10357   \cs_if_exist:cTF { \c__keys_code_root_tl \l_keys_path_tl }

```



```

10358     {
10359         \cs_if_exist_use:c { \c__keys_validate_root_tl \l_keys_path_tl }
10360         \cs:w \c__keys_code_root_tl \l_keys_path_tl \exp_after:wN \cs_end:
10361         \exp_after:wN { \l_keys_value_tl }
10362     }
10363     { \__keys_execute_unknown: }
10364 }
10365 \cs_new_protected:Npn \__keys_execute_unknown:
10366 {
10367     \bool_if:NTF \l__keys_only_known_bool
10368     { \__keys_store_unused: }
10369     {
10370         \cs_if_exist:cTF
10371         { \c__keys_inherit_root_tl \__keys_parent:o \l_keys_path_tl }
10372         {
10373             \clist_map_inline:cn
10374             { \c__keys_inherit_root_tl \__keys_parent:o \l_keys_path_tl }
10375             {
10376                 \cs_if_exist:cT
10377                 { \c__keys_code_root_tl ##1 / \l_keys_key_tl }
10378                 {
10379                     \cs:w \c__keys_code_root_tl ##1 / \l_keys_key_tl
10380                     \exp_after:wN \cs_end: \exp_after:wN
10381                     { \l_keys_value_tl }
10382                     \clist_map_break:
10383                 }
10384             }
10385         }
10386     }
10387     \cs_if_exist:cTF { \c__keys_code_root_tl \l__keys_module_tl / unknown }
10388     {
10389         \cs:w \c__keys_code_root_tl \l__keys_module_tl / unknown
10390         \exp_after:wN \cs_end: \exp_after:wN { \l_keys_value_tl }
10391     }
10392     {
10393         \__msg_kernel_error:nxxx { kernel } { key-unknown }
10394         { \l_keys_path_tl } { \l__keys_module_tl }
10395     }
10396 }
10397 }
10398 }
10399 \cs_new:Npn \__keys_execute:nn #1#2
10400 {
10401     \cs_if_exist:cTF { \c__keys_code_root_tl #1 }
10402     {
10403         \cs:w \c__keys_code_root_tl #1 \exp_after:wN \cs_end:
10404         \exp_after:wN { \l_keys_value_tl }
10405     }
10406     {#2}
10407 }
10408 \cs_new_protected:Npn \__keys_store_unused:
10409 {
10410     \clist_put_right:Nx \l__keys_unused_clist
10411     {

```

```

10412         \exp_not:o \l_keys_key_tl
10413         \bool_if:NF \l_keys_no_value_bool
10414         { = { \exp_not:o \l_keys_value_tl } }
10415     }
10416 }

```

(End definition for `__keys_execute:` and others.)

`__keys_choice_find:n` Executing a choice has two parts. First, try the choice given, then if that fails call the
`__keys_multichoice_find:n` unknown key. That will exist, 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.

```

10417 \cs_new:Npn \__keys_choice_find:n #1
10418 {
10419     \__keys_execute:nn { \l_keys_path_tl / \__keys_remove_spaces:n {#1} }
10420     { \__keys_execute:nn { \l_keys_path_tl / unknown } { } }
10421 }
10422 \cs_new:Npn \__keys_multichoice_find:n #1
10423 { \clist_map_function:nN {#1} \__keys_choice_find:n }

```

(End definition for `__keys_choice_find:n` and `__keys_multichoice_find:n`.)

19.7 Utilities

`__keys_parent:n` Used to strip off the ending part of the key path after the last `/`.

```

\__keys_parent:o
\__keys_parent:w
10424 \cs_new:Npn \__keys_parent:n #1
10425 { \__keys_parent:w #1 / / \q_stop { } }
10426 \cs_generate_variant:Nn \__keys_parent:n { o }
10427 \cs_new:Npn \__keys_parent:w #1 / #2 / #3 \q_stop #4
10428 {
10429     \tl_if_blank:nTF {#2}
10430     { \use_none:n #4 }
10431     {
10432         \__keys_parent:w #2 / #3 \q_stop { #4 / #1 }
10433     }
10434 }

```

(End definition for `__keys_parent:n` and `__keys_parent:w`.)

`__keys_remove_spaces:n` Removes all spaces from the input which is detokenized as a result. This function has
`__keys_remove_spaces:w` the same effect as `\zap@space` in L^AT_EX 2_ε after applying `\tl_to_str:n`. It is set up to
be fast as the use case here is tightly defined. The `?` is only there to allow for a space
after `\use_none:nn` responsible for ending the loop.

```

10435 \cs_new:Npn \__keys_remove_spaces:n #1
10436 {
10437     \exp_after:wN \__keys_remove_spaces:w \tl_to_str:n {#1}
10438     \use_none:nn ? ~
10439 }
10440 \cs_new:Npn \__keys_remove_spaces:w #1 ~
10441 { #1 \__keys_remove_spaces:w }

```

(End definition for `__keys_remove_spaces:n` and `__keys_remove_spaces:w`.)

`\keys_if_exist_p:nn` A utility for others to see if a key exists.

`\keys_if_exist:nnTF`

```

10442 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
10443 {
10444   \cs_if_exist:cTF
10445     { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 } }
10446     { \prg_return_true: }
10447     { \prg_return_false: }
10448 }

```

(End definition for `\keys_if_exist:nnTF`. This function is documented on page 171.)

`\keys_if_choice_exist_p:nnn` Just an alternative view on `\keys_if_exist:nnTF`.

`\keys_if_choice_exist:nnnTF`

```

10449 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
10450 { p , T , F , TF }
10451 {
10452   \cs_if_exist:cTF
10453     { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 / #3 } }
10454     { \prg_return_true: }
10455     { \prg_return_false: }
10456 }

```

(End definition for `\keys_if_choice_exist:nnnTF`. This function is documented on page 171.)

`\keys_show:nn` To show a key, test for its existence to issue the correct message (same message, but with a `t` or `f` argument, then build the control sequences which contain the code and other information about the key, call an intermediate auxiliary which constructs the code that will be displayed to the terminal, and finally conclude with `__msg_show_wrap:n`.

`__keys_show:N`

```

10457 \cs_new_protected:Npn \keys_show:nn #1#2
10458 {
10459   \keys_if_exist:nnTF {#1} {#2}
10460   {
10461     \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-key }
10462     { \__keys_remove_spaces:n { #1 / #2 } } { t } { } { }
10463     \exp_args:Nc \__keys_show:N
10464       { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 } }
10465   }
10466   {
10467     \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-key }
10468     { \__keys_remove_spaces:n { #1 / #2 } } { f } { } { }
10469     \__msg_show_wrap:n { }
10470   }
10471 }
10472 \cs_new_protected:Npn \__keys_show:N #1
10473 {
10474   \use:x
10475   {
10476     \__msg_show_wrap:n
10477     {
10478       \exp_not:N \__msg_show_item_unbraced:nn { code }
10479       { \token_get_replacement_spec:N #1 }
10480     }
10481   }
10482 }

```

(End definition for `\keys_show:nn` and `__keys_show:N`. These functions are documented on page 171.)

19.8 Messages

For when there is a need to complain.

```

10483 \_msg_kernel_new:nnnn { kernel } { boolean-values-only }
10484 { Key~'#1'~accepts~boolean~values~only. }
10485 { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
10486 \_msg_kernel_new:nnnn { kernel } { key-choice-unknown }
10487 { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
10488 {
10489     The~key~'#1'~only~accepts~predefined~values,~
10490     and~'#2'~is~not~one~of~these.
10491 }
10492 \_msg_kernel_new:nnnn { kernel } { key-no-property }
10493 { No~property~given~in~definition~of~key~'#1'. }
10494 {
10495     \c__msg_coding_error_text_tl
10496     Inside~\keys_define:nn  each~key~name~
10497     needs~a~property:  \ \ \
10498     \iow_indent:n { #1 .<property> } \ \ \
10499     LaTeX~did~not~find~a~'. ' ~to~indicate~the~start~of~a~property.
10500 }
10501 \_msg_kernel_new:nnnn { kernel } { key-unknown }
10502 { The~key~'#1'~is~unknown~and~is~being~ignored. }
10503 {
10504     The~module~'#2'~does~not~have~a~key~called~'#1'. \
10505     Check~that~you~have~spelled~the~key~name~correctly.
10506 }
10507 \_msg_kernel_new:nnnn { kernel } { nested-choice-key }
10508 { Attempt~to~define~'#1'~as~a~nested~choice~key. }
10509 {
10510     The~key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
10511     itself~a~choice.
10512 }
10513 \_msg_kernel_new:nnnn { kernel } { property-boolean-values-only }
10514 { The~property~'#1'~accepts~boolean~values~only. }
10515 {
10516     \c__msg_coding_error_text_tl
10517     The~property~'#1'~only~accepts~the~values~'true'~and~'false'.
10518 }
10519 \_msg_kernel_new:nnnn { kernel } { property-requires-value }
10520 { The~property~'#1'~requires~a~value. }
10521 {
10522     \c__msg_coding_error_text_tl
10523     LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'. \
10524     No~value~was~given~for~the~property,~and~one~is~required.
10525 }
10526 \_msg_kernel_new:nnnn { kernel } { property-unknown }
10527 { The~key~property~'#1'~is~unknown. }
10528 {
10529     \c__msg_coding_error_text_tl
10530     LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
10531     this~property~is~not~defined.
10532 }
10533 \_msg_kernel_new:nnnn { kernel } { value-forbidden }

```

```

10534 { The-key~'#1'~does-not~take~a~value. }
10535 {
10536   The-key~'#1'~should-be-given-without~a~value.\\
10537   The-value~'#2'~was-present:~the-key~will~be-ignored.
10538 }
10539 \__msg_kernel_new:nnnn { kernel } { value-required }
10540 { The-key~'#1'~requires~a~value. }
10541 {
10542   The-key~'#1'~must-have~a~value.\\
10543   No-value-was-present:~the-key~will~be-ignored.
10544 }
10545 \__msg_kernel_new:nnn { kernel } { show-key }
10546 {
10547   The-key~#1~
10548   \str_if_eq:nnTF {#2} { t }
10549     { has-the-properties: }
10550     { is-undefined. }
10551 }
10552 </initex | package>

```

20 l3file implementation

The following test files are used for this code: *m3file001*.

```

10553 <*initex | package>
10554 <@@=file>

```

20.1 File operations

\g_file_current_name_tl The name of the current file should be available at all times. For the format the file name needs to be picked up at the start of the file. In L^AT_EX 2_ε package mode the current file name is collected from \currname.

```

10555 \tl_new:N \g_file_current_name_tl
10556 <*initex>
10557 \tex_everyjob:D \exp_after:wN
10558 {
10559   \tex_the:D \tex_everyjob:D
10560   \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
10561 }
10562 </initex>
10563 <*package>
10564 \cs_if_exist:NT \@currname
10565 { \tl_gset_eq:NN \g_file_current_name_tl \@currname }
10566 </package>

```

(End definition for \g_file_current_name_tl. This variable is documented on page 173.)

\g__file_stack_seq The input list of files is stored as a sequence stack.

```

10567 \seq_new:N \g__file_stack_seq

```

(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! In format mode, this is done at the very start of the T_EX run. In package mode we will eventually copy the contents of `\@filelist`.

```

10568 \seq_new:N \g__file_record_seq
10569 \<*initex>
10570 \tex_everyjob:D \exp_after:wN
10571 {
10572   \tex_the:D \tex_everyjob:D
10573   \seq_gput_right:NV \g__file_record_seq \g_file_current_name_tl
10574 }
10575 \</initex>

```

(End definition for `\g__file_record_seq`.)

`\l__file_internal_tl` Used as a short-term scratch variable. It may be possible to reuse `\l__file_internal_name_tl` there.

```

10576 \tl_new:N \l__file_internal_tl

```

(End definition for `\l__file_internal_tl`.)

`\l__file_internal_name_tl` Used to return the fully-qualified name of a file.

```

10577 \tl_new:N \l__file_internal_name_tl

```

(End definition for `\l__file_internal_name_tl`.)

`\l__file_search_path_seq` The current search path.

```

10578 \seq_new:N \l__file_search_path_seq

```

(End definition for `\l__file_search_path_seq`.)

`\l__file_saved_search_path_seq` The current search path has to be saved for package use.

```

10579 \<*package>
10580 \seq_new:N \l__file_saved_search_path_seq
10581 \</package>

```

(End definition for `\l__file_saved_search_path_seq`.)

`\l__file_internal_seq` Scratch space for comma list conversion in package mode.

```

10582 \<*package>
10583 \seq_new:N \l__file_internal_seq
10584 \</package>

```

(End definition for `\l__file_internal_seq`.)

`__file_name_sanitize:nn` For converting a token list to a string where active characters are treated as strings from the start. The logic to the quoting normalisation is the same as used by `lualatexquotejobname`: check for balanced `"`, and assuming they balance strip all of them out before quoting the entire name if it contains spaces.

```

10585 \cs_new_protected:Npn \__file_name_sanitize:nn #1#2
10586 {
10587   \group_begin:
10588     \seq_map_inline:Nn \l_char_active_seq
10589     {
10590       \tl_set:Nx \l__file_internal_tl { \iow_char:N ##1 }

```

```

10591         \char_set_active_eq:NN ##1 \l__file_internal_tl
10592     }
10593     \tl_set:Nx \l__file_internal_name_tl {#1}
10594     \tl_set:Nx \l__file_internal_name_tl
10595         { \tl_to_str:N \l__file_internal_name_tl }
10596     \int_compare:nNnTF
10597     {
10598         \int_mod:nn
10599         {
10600             0 \tl_map_function:NN \l__file_internal_name_tl
10601             \__file_name_sanitize_aux:n
10602         }
10603         \c_two
10604     }
10605     = \c_zero
10606     {
10607         \tl_remove_all:Nn \l__file_internal_name_tl { " }
10608         \tl_if_in:NnT \l__file_internal_name_tl { ~ }
10609         {
10610             \tl_set:Nx \l__file_internal_name_tl
10611                 { " \exp_not:V \l__file_internal_name_tl " }
10612         }
10613     }
10614     {
10615         \__msg_kernel_error:nnx
10616             { kernel } { unbalanced-quote-in-filename }
10617             { \l__file_internal_name_tl }
10618     }
10619     \use:x
10620     {
10621         \group_end:
10622         \exp_not:n {#2} { \l__file_internal_name_tl }
10623     }
10624 }
10625 \cs_new:Npn \__file_name_sanitize_aux:n #1
10626 {
10627     \token_if_eq_charcode:NNT #1 "
10628     { + \c_one }
10629 }

```

(End definition for `__file_name_sanitize:nn` and `__file_name_sanitize_aux:n`.)

`\file_add_path:nN` The way to test if a file exists is to try to open it: if it does not exist then TeX will report end-of-file. For files which are in the current directory, this is straight-forward. `__file_add_path:nN` For other locations, a search has to be made looking at each potential path in turn. The first location is of course treated as the correct one. If nothing is found, #2 is returned empty. `__file_add_path_search:nN`

```

10630 \cs_new_protected:Npn \file_add_path:nN #1
10631 { \__file_name_sanitize:nn {#1} { \__file_add_path:nN } }
10632 \cs_new_protected:Npn \__file_add_path:nN #1#2
10633 {
10634     \__ior_open:Nn \g__file_internal_ior {#1}
10635     \ior_if_eof:NTF \g__file_internal_ior
10636     { \__file_add_path_search:nN {#1} #2 }

```

```

10637     { \tl_set:Nn #2 {#1} }
10638     \ior_close:N \g__file_internal_ior
10639   }
10640   \cs_new_protected:Npn \__file_add_path_search:nN #1#2
10641   {
10642     \tl_set:Nn #2 { \q_no_value }
10643     \*package
10644     \cs_if_exist:NT \input@path
10645     {
10646       \seq_set_eq:NN \l__file_saved_search_path_seq
10647       \l__file_search_path_seq
10648       \seq_set_split:NnV \l__file_internal_seq { , } \input@path
10649       \seq_concat:NNN \l__file_search_path_seq
10650       \l__file_search_path_seq \l__file_internal_seq
10651     }
10652   \*package
10653   \seq_map_inline:Nn \l__file_search_path_seq
10654   {
10655     \__ior_open:Nn \g__file_internal_ior { ##1 #1 }
10656     \ior_if_eof:NF \g__file_internal_ior
10657     {
10658       \tl_set:Nx #2 { ##1 #1 }
10659       \seq_map_break:
10660     }
10661   }
10662   \*package
10663   \cs_if_exist:NT \input@path
10664   {
10665     \seq_set_eq:NN \l__file_search_path_seq
10666     \l__file_saved_search_path_seq
10667   }
10668 \*package
10669 }

```

(End definition for `\file_add_path:nN`, `__file_add_path:nN`, and `__file_add_path_search:nN`. These functions are documented on page 173.)

`\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 will contain something, whereas if the file was not located then the return value will be `\q_no_value`.

```

10670 \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
10671 {
10672   \file_add_path:nN {#1} \l__file_internal_name_tl
10673   \quark_if_no_value:NTF \l__file_internal_name_tl
10674   { \prg_return_false: }
10675   { \prg_return_true: }
10676 }

```

(End definition for `\file_if_exist:nTF`. This function is documented on page 173.)

`\file_input:n` Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the `\g__file_stack_seq`, and add it to the file list, either `\g__file_record_seq`, or `\@filelist` in package mode.

```

\__file_input:n
\__file_input:V
\__file_input_aux:n
\__file_input_aux:o
10677 \cs_new_protected:Npn \file_input:n #1

```



```

10678 {
10679   \__file_if_exist:nT {#1}
10680   { \__file_input:V \l__file_internal_name_tl }
10681 }

```

This code is spun out as a separate function to encapsulate the error message into a easy-to-reuse form.

```

10682 \cs_new_protected:Npn \__file_if_exist:nT #1#2
10683 {
10684   \file_if_exist:nTF {#1}
10685     {#2}
10686     {
10687       \__file_name_sanitiz:nn {#1}
10688       { \__msg_kernel_error:nxx { kernel } { file-not-found } }
10689     }
10690 }
10691 \cs_new_protected:Npn \__file_input:n #1
10692 {
10693   \tl_if_in:nnTF {#1} { . }
10694     { \__file_input_aux:n {#1} }
10695     { \__file_input_aux:o { \tl_to_str:n { #1 . tex } } }
10696 }
10697 \cs_generate_variant:Nn \__file_input:n { V }
10698 \cs_new_protected:Npn \__file_input_aux:n #1
10699 {
10700   \*initex
10701   \seq_gput_right:Nn \g__file_record_seq {#1}
10702   \*initex
10703   \*package
10704   \clist_if_exist:NTF \@filelist
10705     { \@addtofilelist {#1} }
10706     { \seq_gput_right:Nn \g__file_record_seq {#1} }
10707   \*package
10708   \seq_gpush:No \g__file_stack_seq \g_file_current_name_tl
10709   \tl_gset:Nn \g_file_current_name_tl {#1}
10710   \tex_input:D #1 \c_space_tl
10711   \seq_gpop:NN \g__file_stack_seq \l__file_internal_tl
10712   \tl_gset_eq:NN \g_file_current_name_tl \l__file_internal_tl
10713 }
10714 \cs_generate_variant:Nn \__file_input_aux:n { o }

```

(End definition for \file_input:n and others. These functions are documented on page 173.)

```

\file_path_include:n
\file_path_remove:n
\__file_path_include:n

```

Wrapper functions to manage the search path.

```

10715 \cs_new_protected:Npn \file_path_include:n #1
10716 { \__file_name_sanitiz:nn {#1} { \__file_path_include:n } }
10717 \cs_new_protected:Npn \__file_path_include:n #1
10718 {
10719   \seq_if_in:NnF \l__file_search_path_seq {#1}
10720     { \seq_put_right:Nn \l__file_search_path_seq {#1} }
10721 }
10722 \cs_new_protected:Npn \file_path_remove:n #1
10723 {
10724   \__file_name_sanitiz:nn {#1}
10725   { \seq_remove_all:Nn \l__file_search_path_seq }

```

```
10726 }
```

(End definition for `\file_path_include:n`, `\file_path_remove:n`, and `__file_path_include:n`. These functions are documented on page 173.)

\file_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 each file name into a string.

```
10727 \cs_new_protected:Npn \file_list:
10728 {
10729   \seq_set_eq:NN \l__file_internal_seq \g__file_record_seq
10730   \*package
10731   \clist_if_exist:NT \@filelist
10732   {
10733     \clist_map_inline:Nn \@filelist
10734     {
10735       \seq_put_right:No \l__file_internal_seq
10736       { \tl_to_str:n {##1} }
10737     }
10738   }
10739   \*package
10740   \seq_remove_duplicates:N \l__file_internal_seq
10741   \iow_log:n { *File~List~* }
10742   \seq_map_inline:Nn \l__file_internal_seq { \iow_log:n {##1} }
10743   \iow_log:n { ***** }
10744 }
```

(End definition for `\file_list:`. This function is documented on page 174.)

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`.

```
10745 \*package
10746 \AtBeginDocument
10747 {
10748   \clist_map_inline:Nn \@filelist
10749   { \seq_gput_right:No \g__file_record_seq { \tl_to_str:n {##1} } }
10750 }
10751 \*package
```

20.2 Input operations

```
10752 (@@=ior)
```

20.2.1 Variables and constants

\c_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.

```
10753 \cs_new_eq:NN \c_term_ior \c_sixteen
```

(End definition for `\c_term_ior`. This variable is documented on page 179.)

`\g__ior_streams_seq` A list of the currently-available input streams to be used as a stack. In format mode, all streams (from 0 to 15) are available, while the package requests streams to L^AT_EX 2_ε as they are needed (initially none are needed), so the starting point varies!

```

10754 \seq_new:N \g__ior_streams_seq
10755 ⟨*initex⟩
10756 \seq_gset_split:Nnn \g__ior_streams_seq { , }
10757 { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }
10758 ⟨/initex⟩

```

(End definition for `\g__ior_streams_seq`.)

`\l__ior_stream_tl` Used to recover the raw stream number from the stack.

```

10759 \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^AT_EX 2_ε and plain T_EX 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_EXt, we need to look at `\count38` but there is no subtraction: like the original plain T_EX/L^AT_EX 2_ε mechanism it holds the value of the *last* stream allocated.

```

10760 \prop_new:N \g__ior_streams_prop
10761 ⟨*package⟩
10762 \int_step_inline:nnnn
10763 { \c_zero }
10764 { \c_one }
10765 {
10766   \cs_if_exist:NTF \normalend
10767   { \tex_count:D 38 \scan_stop: }
10768   {
10769     \tex_count:D 16 \scan_stop:
10770     \cs_if_exist:NT \loccount { - \c_one }
10771   }
10772 }
10773 {
10774   \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved-by~format }
10775 }
10776 ⟨/package⟩

```

(End definition for `\g__ior_streams_prop`.)

20.2.2 Stream management

`\ior_new:N` Reserving a new stream is done by defining the name as equal to using the terminal.

```

\ior_new:c 10777 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_term_ior }
10778 \cs_generate_variant:Nn \ior_new:N { c }

```

(End definition for `\ior_new:N`. This function is documented on page 174.)

`\ior_open:Nn` Opening an input stream requires a bit of pre-processing. The file name is sanitized to deal with active characters, before an auxiliary adds a path and checks that the file really exists. If those two tests pass, then pass the information on to the lower-level function which deals with streams.

`\ior_open:cn`

`__ior_open_aux:Nn`

```

10779 \cs_new_protected:Npn \ior_open:Nn #1#2
10780 { \__file_name_sanitiz:nn {#2} { \__ior_open_aux:Nn #1 } }
10781 \cs_generate_variant:Nn \ior_open:Nn { c }
10782 \cs_new_protected:Npn \__ior_open_aux:Nn #1#2
10783 {
10784   \file_add_path:nN {#2} \l__file_internal_name_tl
10785   \quark_if_no_value:NTF \l__file_internal_name_tl
10786     { \__msg_kernel_error:nxx { kernel } { file-not-found } {#2} }
10787     { \__ior_open:No #1 \l__file_internal_name_tl }
10788 }

```

(End definition for `\ior_open:Nn` and `__ior_open_aux:Nn`. These functions are documented on page 174.)

`\ior_open:NnTF` Much the same idea for opening a read with a conditional, except the auxiliary function
`\ior_open:cnTF` does not issue an error if the file is not found.
`__ior_open_aux:NnTF`

```

10789 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
10790 { \__file_name_sanitiz:nn {#2} { \__ior_open_aux:NnTF #1 } }
10791 \cs_generate_variant:Nn \ior_open:NnT { c }
10792 \cs_generate_variant:Nn \ior_open:NnF { c }
10793 \cs_generate_variant:Nn \ior_open:NnTF { c }
10794 \cs_new_protected:Npn \__ior_open_aux:NnTF #1#2
10795 {
10796   \file_add_path:nN {#2} \l__file_internal_name_tl
10797   \quark_if_no_value:NTF \l__file_internal_name_tl
10798     { \prg_return_false: }
10799     {
10800       \__ior_open:No #1 \l__file_internal_name_tl
10801       \prg_return_true:
10802     }
10803 }

```

(End definition for `\ior_open:NnTF` and `__ior_open_aux:NnTF`. These functions are documented on page 174.)

`__ior_new:N` In package mode, 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`.

```

10804 \*package
10805 \exp_args:NNf \cs_new_protected:Npn \__ior_new:N
10806 { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
10807 \*package

```

(End definition for `__ior_new:N`.)

`__ior_open:Nn` The stream allocation itself uses the fact that there is a list of all of those available, so
`__ior_open:No` allocation is simply a question of using the number at the top of the list. In package
`__ior_open_stream:Nn` mode, 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 $\text{\LaTeX}_{2\epsilon}$ for a new stream and use that number (after a bit of conversion).

```

10808 \cs_new_protected:Npn \__ior_open:Nn #1#2

```

```

10809 {
10810   \ior_close:N #1
10811   \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
10812   { \__ior_open_stream:Nn #1 {#2} }
10813   \*initex
10814   { \_msg_kernel_fatal:nn { kernel } { input-streams-exhausted } }
10815   \*initex
10816   \*package
10817   {
10818     \__ior_new:N #1
10819     \tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
10820     \__ior_open_stream:Nn #1 {#2}
10821   }
10822   \*package
10823 }
10824 \cs_generate_variant:Nn \__ior_open:Nn { No }
10825 \cs_new_protected:Npn \__ior_open_stream:Nn #1#2
10826 {
10827   \tex_global:D \tex_chardef:D #1 = \l__ior_stream_tl \scan_stop:
10828   \prop_gput:Nvn \g__ior_streams_prop #1 {#2}
10829   \tex_openin:D #1 #2 \scan_stop:
10830 }

```

(End definition for __ior_open:Nn and __ior_open_stream:Nn.)

\ior_close:N Closing a stream means getting rid of it at the T_EX 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.

\ior_close:c

```

10831 \cs_new_protected:Npn \ior_close:N #1
10832 {
10833   \int_compare:nT { - \c_one < #1 < \c_sixteen }
10834   {
10835     \tex_closein:D #1
10836     \prop_gremove:Nv \g__ior_streams_prop #1
10837     \seq_if_in:NvF \g__ior_streams_seq #1
10838     { \seq_gpush:Nv \g__ior_streams_seq #1 }
10839     \cs_gset_eq:NN #1 \c_term_ior
10840   }
10841 }
10842 \cs_generate_variant:Nn \ior_close:N { c }

```

(End definition for \ior_close:N. This function is documented on page 175.)

\ior_list_streams: 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 (in fact a question mark) otherwise. The code of the message `show-streams` takes care of translating `ior/iow` to English. The list of streams is formatted using `_msg_show_item_unbraced:nn`.

__ior_list_streams:Nn

```

10843 \cs_new_protected:Npn \ior_list_streams:
10844 { \__ior_list_streams:Nn \g__ior_streams_prop { ior } }
10845 \cs_new_protected:Npn \__ior_list_streams:Nn #1#2
10846 {
10847   \_msg_show_pre:nnxxxx { LaTeX / kernel } { show-streams }

```

```

10848     {#2} { \prop_if_empty:NF #1 { ? } } { } { }
10849     \__msg_show_wrap:n
10850     { \prop_map_function:NN #1 \__msg_show_item_unbraced:nn }
10851   }

```

(End definition for `\ior_list_streams:` and `__ior_list_streams:Nn`. These functions are documented on page 175.)

20.2.3 Reading input

`\if_eof:w` The primitive conditional

```

10852 \cs_new_eq:NN \if_eof:w \tex_ifeof:D

```

(End definition for `\if_eof:w`.)

`\ior_if_eof_p:N` To test if some particular input stream is exhausted the following conditional is provided.

`\ior_if_eof:N`**`TF`**

```

10853 \prg_new_conditional:Nnn \ior_if_eof:N { p , T , F , TF }
10854 {
10855   \cs_if_exist:NTF #1
10856   {
10857     \if_int_compare:w #1 = \c_sixteen
10858     \prg_return_true:
10859   \else:
10860     \if_eof:w #1
10861     \prg_return_true:
10862   \else:
10863     \prg_return_false:
10864   \fi:
10865   \fi:
10866 }
10867 { \prg_return_true: }
10868 }

```

(End definition for `\ior_if_eof:N`. This function is documented on page 176.)

`\ior_get:NN` And here we read from files.

```

10869 \cs_new_protected:Npn \ior_get:NN #1#2
10870 { \tex_read:D #1 to #2 }

```

(End definition for `\ior_get:NN`. This function is documented on page 175.)

`\ior_str_get:NN` Reading as strings is a more complicated wrapper, as we wish to remove the newline character.

```

10871 \cs_new_protected:Npn \ior_str_get:NN #1#2
10872 {
10873   \use:x
10874   {
10875     \int_set:Nn \tex_endlinechar:D { - \c_one }
10876     \exp_not:n { \etex_readline:D #1 to #2 }
10877     \int_set:Nn \tex_endlinechar:D { \int_use:N \tex_endlinechar:D }
10878   }
10879 }

```

(End definition for `\ior_str_get:NN`. This function is documented on page 176.)

`\g__file_internal_ior` Needed by the higher-level code, but cannot be created until here.

```
10880 \ior_new:N \g__file_internal_ior
```

(End definition for `\g__file_internal_ior`.)

20.3 Output operations

```
10881 <@@=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.

20.3.1 Variables and constants

`\c_log_iow` Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`)
`\c_term_iow` and to both the terminal and transcript file (`\c_term_iow`).

```
10882 \cs_new_eq:NN \c_log_iow \c_minus_one
```

```
10883 \int_const:Nn \c_term_iow { 128 }
```

(End definition for `\c_log_iow` and `\c_term_iow`. These variables are documented on page 179.)

`\g__iow_streams_seq` A list of the currently-available output streams to be used as a stack.

```
10884 \seq_new:N \g__iow_streams_seq
```

```
10885 <*initex>
```

```
10886 \seq_set_eq:NN \g__iow_streams_seq \g__ior_streams_seq
```

```
10887 \cs_if_exist:NT \luatex_directlua:D
```

```
10888 {
```

```
10889   \int_compare:nNnT \luatex luatexversion:D > { 80 }
```

```
10890   {
```

```
10891     \int_step_inline:nnnn { 16 } { 1 } { 127 }
```

```
10892     {
```

```
10893       \seq_gput_right:Nn \g__iow_streams_seq {#1}
```

```
10894     }
```

```
10895   }
```

```
10896 }
```

```
10897 </initex>
```

(End definition for `\g__iow_streams_seq`.)

`\l__iow_stream_tl` Used to recover the raw stream number from the stack.

```
10898 \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.

```
10899 \prop_new:N \g__iow_streams_prop
```

```
10900 <*package>
```

```
10901 \int_step_inline:nnnn
```

```
10902 { \c_zero }
```

```
10903 { \c_one }
```

```
10904 {
```

```
10905   \cs_if_exist:NTF \normalend
```

```
10906   { \tex_count:D 39 \scan_stop: }
```

```
10907   {
```

```
10908     \tex_count:D 17 \scan_stop:
```

```
10909     \cs_if_exist:NT \loccount { - \c_one }
```

```

10910     }
10911   }
10912   {
10913     \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved-by~format }
10914   }
10915 \endpackage

```

(End definition for \g__iow_streams_prop.)

20.4 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.

```

10916 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
10917 \cs_generate_variant:Nn \iow_new:N { c }

```

(End definition for \iow_new:N. This function is documented on page 174.)

__iow_new:N As for read streams, copy \newwrite in package mode, making sure that it is not \outer.

```

10918 \begin{package}
10919 \exp_args:NNf \cs_new_protected:Npn \__iow_new:N
10920   { \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }
10921 \end{package}

```

(End definition for __iow_new:N.)

\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.

```

\__iow_open:Nn
\__iow_open_stream:Nn
10922 \cs_new_protected:Npn \iow_open:Nn #1#2
10923   { \__file_name_sanitiz:nn {#2} { \__iow_open:Nn #1 } }
10924 \cs_generate_variant:Nn \iow_open:Nn { c }
10925 \cs_new_protected:Npn \__iow_open:Nn #1#2
10926   {
10927     \iow_close:N #1
10928     \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
10929     { \__iow_open_stream:Nn #1 {#2} }
10930 \*initex
10931   { \__msg_kernel_fatal:nn { kernel } { output-streams-exhausted } }
10932 \endinitex
10933 \begin{package}
10934   {
10935     \__iow_new:N #1
10936     \tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
10937     \__iow_open_stream:Nn #1 {#2}
10938   }
10939 \endpackage
10940 }
10941 \cs_generate_variant:Nn \__iow_open:Nn { No }
10942 \cs_new_protected:Npn \__iow_open_stream:Nn #1#2
10943   {
10944     \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
10945     \prop_gput:NVn \g__iow_streams_prop #1 {#2}
10946     \tex_immediate:D \tex_openout:D #1 #2 \scan_stop:
10947   }

```


(End definition for `\iow_open:Nn`, `__iow_open:Nn`, and `__iow_open_stream:Nn`. These functions are documented on page 174.)

`\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.

`\iow_close:c`

```

10948 \cs_new_protected:Npn \iow_close:N #1
10949 {
10950   \int_compare:nT { - \c_one < #1 < \c_sixteen }
10951   {
10952     \tex_immediate:D \tex_closeout:D #1
10953     \prop_gremove:NV \g__iow_streams_prop #1
10954     \seq_if_in:NVF \g__iow_streams_seq #1
10955     { \seq_gpush:NV \g__iow_streams_seq #1 }
10956     \cs_gset_eq:NN #1 \c_term_iow
10957   }
10958 }
10959 \cs_generate_variant:Nn \iow_close:N { c }

```

(End definition for `\iow_close:N`. This function is documented on page 175.)

`\iow_list_streams:` Done as for input, but with a copy of the auxiliary so the name is correct.

`__iow_list_streams:Nn`

```

10960 \cs_new_protected:Npn \iow_list_streams:
10961 { \__iow_list_streams:Nn \g__iow_streams_prop { iow } }
10962 \cs_new_eq:NN \__iow_list_streams:Nn \__ior_list_streams:Nn

```

(End definition for `\iow_list_streams:` and `__iow_list_streams:Nn`. These functions are documented on page 175.)

20.4.1 Deferred writing

`\iow_shipout_x:Nn` First the easy part, this is the primitive, which expects its argument to be braced.

`\iow_shipout_x:Nx`

`\iow_shipout_x:cn`

`\iow_shipout_x:cx`

```

10963 \cs_new_protected:Npn \iow_shipout_x:Nn #1#2
10964 { \tex_write:D #1 {#2} }
10965 \cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }

```

(End definition for `\iow_shipout_x:Nn`. This function is documented on page 177.)

`\iow_shipout:Nn` With ε -TEX available deferred writing without expansion is easy.

`\iow_shipout:Nx`

`\iow_shipout:cn`

`\iow_shipout:cx`

```

10966 \cs_new_protected:Npn \iow_shipout:Nn #1#2
10967 { \tex_write:D #1 { \exp_not:n {#2} } }
10968 \cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }

```

(End definition for `\iow_shipout:Nn`. This function is documented on page 176.)

20.4.2 Immediate writing

`__iow_with:Nnn`

`__iow_with_aux: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.

```

10969 \cs_new_protected:Npn \__iow_with:Nnn #1#2
10970 {
10971   \int_compare:nNnTF {#1} = {#2}
10972   { \use:n }

```

```

10973     { \exp_args:No \__iow_with_aux:nNnn { \int_use:N #1 } #1 {#2} }
10974   }
10975 \cs_new_protected:Npn \__iow_with_aux:nNnn #1#2#3#4
10976 {
10977   \int_set:Nn #2 {#3}
10978   #4
10979   \int_set:Nn #2 {#1}
10980 }

```

(End definition for `__iow_with:Nnn` and `__iow_with_aux:nNnn`.)

`\iow_now:Nn` This routine writes the second argument onto the output stream without expansion. If
`\iow_now:Nx` this stream isn't open, the output goes to the terminal instead. If the first argument is
`\iow_now:cn` no output stream at all, we get an internal error. We don't use the expansion done by
`\iow_now:cx` `\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 `__iow_with:Nnn` to support formats such as plain `TEX`: otherwise, `\iow_-`
`newline:` would not work. We do not do this for `\iow_shipout:Nn` or `\iow_shipout_-`
`x:Nn`, as `TEX` looks at the value of the `\newlinechar` at shipout time in those cases.

```

10981 \cs_new_protected:Npn \iow_now:Nn #1#2
10982 {
10983   \__iow_with:Nnn \tex_newlinechar:D { '\^^J }
10984   { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
10985 }
10986 \cs_generate_variant:Nn \iow_now:Nn { c, Nx, cx }

```

(End definition for `\iow_now:Nn`. This function is documented on page 176.)

`\iow_log:n` Writing to the log and the terminal directly are relatively easy.
`\iow_log:x`
`\iow_term:n`
`\iow_term:x`

```

10987 \cs_set_protected:Npn \iow_log:x { \iow_now:Nx \c_log_iow }
10988 \cs_new_protected:Npn \iow_log:n { \iow_now:Nn \c_log_iow }
10989 \cs_set_protected:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
10990 \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 176.)

20.4.3 Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written
to an output stream.

```

10991 \cs_new:Npn \iow_newline: { ^^J }

```

(End definition for `\iow_newline:`. This function is documented on page 177.)

`\iow_char:N` Function to write any escaped char to an output stream.

```

10992 \cs_new_eq:NN \iow_char:N \cs_to_str:N

```

(End definition for `\iow_char:N`. This function is documented on page 177.)

20.4.4 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 T_EXLive and MikT_EX.

```
10993 \int_new:N \l_iow_line_count_int
10994 \int_set:Nn \l_iow_line_count_int { 78 }
```

(End definition for `\l_iow_line_count_int`. This variable is documented on page 178.)

`\l__iow_target_count_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.

```
10995 \int_new:N \l__iow_target_count_int
```

(End definition for `\l__iow_target_count_int`.)

`\l__iow_current_line_int` These store the number of characters in the line and word currently being constructed, and the current indentation, respectively.

```
\l__iow_current_word_int
\l__iow_current_indentation_int
10996 \int_new:N \l__iow_current_line_int
10997 \int_new:N \l__iow_current_word_int
10998 \int_new:N \l__iow_current_indentation_int
```

(End definition for `\l__iow_current_line_int`, `\l__iow_current_word_int`, and `\l__iow_current_indentation_int`.)

`\l__iow_current_line_tl` These hold the current line of text and current word, and a number of spaces for indentation, respectively.

```
\l__iow_current_word_tl
\l__iow_current_indentation_tl
10999 \tl_new:N \l__iow_current_line_tl
11000 \tl_new:N \l__iow_current_word_tl
11001 \tl_new:N \l__iow_current_indentation_tl
```

(End definition for `\l__iow_current_line_tl`, `\l__iow_current_word_tl`, and `\l__iow_current_indentation_tl`.)

`\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.

```
11002 \tl_new:N \l__iow_wrap_tl
```

(End definition for `\l__iow_wrap_tl`.)

`\l__iow_newline_tl` The token list inserted to produce the new line, with the *⟨run-on text⟩*.

```
11003 \tl_new:N \l__iow_newline_tl
```

(End definition for `\l__iow_newline_tl`.)

`\l__iow_line_start_bool` Boolean to avoid adding a space at the beginning of forced newlines, and to know when to add the indentation.

```
11004 \bool_new:N \l__iow_line_start_bool
```

(End definition for `\l__iow_line_start_bool`.)

`\c_catcode_other_space_tl` Create a space with category code 12: an “other” space.

```
11005 \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 178.)

`\c__iow_wrap_marker_tl` Every special action of the wrapping code is preceded by the same recognizable string, `\c__iow_wrap_end_marker_tl` `\c__iow_wrap_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 nicer.

```

11006 \group_begin:
11007   \int_set:Nn \tex_escapechar:D { - \c_one }
11008   \tl_const:Nx \c__iow_wrap_marker_tl
11009     { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
11010 \group_end:
11011 \tl_map_inline:nn
11012   { { end } { newline } { indent } { unindent } }
11013   {
11014     \tl_const:cx { c__iow_wrap_ #1 _marker_tl }
11015     {
11016       \c_catcode_other_space_tl
11017       \c__iow_wrap_marker_tl
11018       \c_catcode_other_space_tl
11019       #1
11020       \c_catcode_other_space_tl
11021     }
11022   }

```

(End definition for `\c__iow_wrap_marker_tl` and others.)

`\iow_indent:n` We give a (protected) error definition to `\iow_indent:n` when outside messages. Within wrapped message, it places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. Note that there will be no forced line-break, so the indentation only changes when the next line is started.

```

11023 \cs_new:Npx \__iow_indent:n #1
11024   {
11025     \c__iow_wrap_indent_marker_tl
11026     #1
11027     \c__iow_wrap_unindent_marker_tl
11028   }
11029 \cs_new:Npn \__iow_indent_error:n #1
11030   {
11031     \__msg_kernel_expandable_error:nn { kernel } { indent-outside-wrapping-code }
11032     #1
11033   }
11034 \cs_new_protected:Npn \iow_indent:n { \__iow_indent_error:n }

```

(End definition for `\iow_indent:n`, `__iow_indent:n`, and `__iow_indent_error:n`. These functions are documented on page 178.)

`\iow_wrap:nnnN` The main wrapping function works as follows. First give `\\`, `_` and other formatting commands the correct definition for messages, before fully-expanding the input. In package mode, the expansion uses L^AT_EX 2_ε’s `\protect` mechanism. Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and its length, and initialize some registers. There is then a loop over each word in the input, which will do the actual wrapping. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The argument `#4` is available for additional set up steps for the output. The definition of `\\` and `_` use 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. The `\tl_to_str:N` step converts the “other” space back to a normal space.

```

11035 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
11036 {
11037   \group_begin:
11038     \int_set:Nn \tex_escapechar:D { - \c_one }
11039     \cs_set:Npx \{ { \token_to_str:N \{ }
11040     \cs_set:Npx \# { \token_to_str:N \# }
11041     \cs_set:Npx \} { \token_to_str:N \} }
11042     \cs_set:Npx \% { \token_to_str:N \% }
11043     \cs_set:Npx \~ { \token_to_str:N \~ }
11044     \int_set:Nn \tex_escapechar:D { 92 }
11045     \cs_set_eq:NN \ \c__iow_wrap_newline_marker_tl
11046     \cs_set_eq:NN \ \c_catcode_other_space_tl
11047     \cs_set_eq:NN \iow_indent:n \__iow_indent:n
11048     #3
11049     \*initex
11050     \tl_set:Nx \l__iow_wrap_tl {#1}
11051   \group_end:
11052   \*package
11053   \__iow_wrap_set:Nx \l__iow_wrap_tl {#1}
11054 \group_end:

```

To warn users that `\iow_indent:n` only works in the first argument of `\iow_wrap:nnnN` reset `\iow_indent:n` to its error definition. Then store a newline character and the run-on text as a string in `\l__iow_newline_tl`, and set some variables. The first line’s target count is equal to the length of the whole line. The value `\l__iow_target_count_int` is altered later on by `__iow_wrap_set_target:`.

```

11055   \cs_set_eq:NN \iow_indent:n \__iow_indent_error:n
11056   \tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
11057   \tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
11058   \int_set_eq:NN \l__iow_target_count_int \l_iow_line_count_int
11059   \tl_clear:N \l__iow_current_indentation_tl
11060   \int_zero:N \l__iow_current_line_int
11061   \tl_set:Nn \l__iow_current_line_tl { \use_none:n }
11062   \bool_set_true:N \l__iow_line_start_bool

```

After some setup above (in particular the odd setting of the current line to `\use_none:n`), a loop goes through space-delimited words in the message, recognizing special markers. To make sure that the first line behaves identically to others, start with a newline marker: the `\use_none:n` above avoids actually getting a new line in the output.

```

11063   \use:x
11064   {
11065     \exp_not:n { \tl_clear:N \l__iow_wrap_tl }
11066     \__iow_wrap_loop:w
11067     \tl_to_str:N \c__iow_wrap_newline_marker_tl
11068     \tl_to_str:N \l__iow_wrap_tl
11069     \tl_to_str:N \c__iow_wrap_end_marker_tl
11070     \c_space_tl \c_space_tl
11071     \exp_not:N \q_stop
11072   }
11073   \exp_args:NNo \group_end:
11074   #4 \l__iow_wrap_tl

```

```
11075 }
```

As using the generic loader will mean that `\protected@edef` is not available, it's not placed directly in the wrap function but is set up as an auxiliary. In the generic loader this can then be redefined.

```
11076 \*package>
11077 \cs_new_eq:NN \__iow_wrap_set:Nx \protected@edef
11078 \</package>
```

(End definition for `\iow_wrap:nnnN` and `__iow_wrap_set:Nx`. These functions are documented on page 178.)

`__iow_wrap_set_target:` This is called at the beginning of every line (both those forced by `\\` and those due to line-breaking). The initial call does nothing except redefine `__iow_wrap_set_target:` itself (within the group in which `\iow_wrap:nnnN` works). The next call (at the beginning of the second line) disables any later call and sets the `\l__iow_target_count_int` to the correct value, namely the `\l_iow_line_count_int` shortened by the length of the run-on text (the shift by 1 is due to the presence of `\iow_newline:` in `\l__iow_newline_tl`). This is a bit of a hack to measure the string length of the run on text without the `l3str` module (which is still experimental). This should be replaced once the string module is finalised with something a little cleaner.

```
11079 \cs_new_protected:Npn \__iow_wrap_set_target:
11080 {
11081   \cs_set_protected:Npn \__iow_wrap_set_target:
11082   {
11083     \cs_set_protected:Npn \__iow_wrap_set_target: { }
11084     \tl_replace_all:Nnn \l__iow_newline_tl { ~ } { \c_space_tl }
11085     \int_set:Nn \l__iow_target_count_int
11086     { \l_iow_line_count_int - \tl_count:N \l__iow_newline_tl + \c_one }
11087   }
11088 }
```

(End definition for `__iow_wrap_set_target:.`)

`__iow_wrap_loop:w` The loop grabs one word in the input, and checks whether it is the special marker, or a normal word.

```
11089 \cs_new_protected:Npn \__iow_wrap_loop:w #1 ~ %
11090 {
11091   \tl_set:Nn \l__iow_current_word_tl {#1}
11092   \tl_if_eq:NNTF \l__iow_current_word_tl \c__iow_wrap_marker_tl
11093   { \__iow_wrap_special:w }
11094   { \__iow_wrap_word: }
11095 }
```

(End definition for `__iow_wrap_loop:w.`)

`__iow_wrap_word:` For a normal word, update the line count, then test if the current word would fit in the current line, and call the appropriate function. If the word fits in the current line, add it to the line, preceded by a space unless it is the first word of the line. Otherwise, the current line is added to the result, with the run-on text. The current word (and its character count) are then put in the new line.

```
11096 \cs_new_protected:Npn \__iow_wrap_word:
11097 {
11098   \int_set:Nn \l__iow_current_word_int
```

```

11099     { \exp_args:No \str_count_ignore_spaces:n \l__iow_current_word_tl }
11100     \int_add:Nn \l__iow_current_line_int { \l__iow_current_word_int }
11101     \int_compare:nNnTF \l__iow_current_line_int < \l__iow_target_count_int
11102     { \__iow_wrap_word_fits: }
11103     { \__iow_wrap_word_newline: }
11104     \__iow_wrap_loop:w
11105   }
11106   \cs_new_protected:Npn \__iow_wrap_word_fits:
11107   {
11108     \bool_if:NTF \l__iow_line_start_bool
11109     {
11110       \bool_set_false:N \l__iow_line_start_bool
11111       \tl_put_right:Nx \l__iow_current_line_tl
11112       { \l__iow_current_indentation_tl \l__iow_current_word_tl }
11113       \int_add:Nn \l__iow_current_line_int
11114       { \l__iow_current_indentation_int }
11115     }
11116     {
11117       \tl_put_right:Nx \l__iow_current_line_tl
11118       { ~ \l__iow_current_word_tl }
11119       \int_incr:N \l__iow_current_line_int
11120     }
11121   }
11122   \cs_new_protected:Npn \__iow_wrap_word_newline:
11123   {
11124     \__iow_wrap_set_target:
11125     \tl_put_right:Nx \l__iow_wrap_tl
11126     { \l__iow_current_line_tl \l__iow_newline_tl }
11127     \int_set:Nn \l__iow_current_line_int
11128     {
11129       \l__iow_current_word_int
11130       + \l__iow_current_indentation_int
11131     }
11132     \tl_set:Nx \l__iow_current_line_tl
11133     { \l__iow_current_indentation_tl \l__iow_current_word_tl }
11134   }

```

(End definition for `__iow_wrap_word:`, `__iow_wrap_word_fits:`, and `__iow_wrap_word_newline:`.)

<pre> __iow_wrap_special:w __iow_wrap_newline:w __iow_wrap_indent:w __iow_wrap_unindent:w __iow_wrap_end:w </pre>	<p>When the “special” marker is encountered, read what operation to perform, as a space-delimited argument, perform it, and remember to loop. In fact, to avoid spurious spaces when two special actions follow each other, we look ahead for another copy of the marker. Forced newlines are almost identical to those caused by overflow, except that here the word is empty. To indent more, add four spaces to the start of the indentation token list. To reduce indentation, rebuild the indentation token list using <code>\prg_replicate:nn</code>. At the end, we simply save the last line (without the run-on text), and prevent the loop.</p>
--	---

```

11135   \cs_new_protected:Npn \__iow_wrap_special:w #1 ~ #2 ~ #3 ~ %
11136   {
11137     \use:c { __iow_wrap_#1: }
11138     \str_if_eq_x:nnTF { #2~#3 } { ~ \c__iow_wrap_marker_tl }
11139     { \__iow_wrap_special:w }
11140     { \__iow_wrap_loop:w #2 ~ #3 ~ }
11141   }
11142   \cs_new_protected:Npn \__iow_wrap_newline:

```

```

11143 {
11144   \__iow_wrap_set_target:
11145   \tl_put_right:Nx \l__iow_wrap_tl
11146   { \l__iow_current_line_tl \l__iow_newline_tl }
11147   \int_zero:N \l__iow_current_line_int
11148   \tl_clear:N \l__iow_current_line_tl
11149   \bool_set_true:N \l__iow_line_start_bool
11150 }
11151 \cs_new_protected:Npx \__iow_wrap_indent:
11152 {
11153   \int_add:Nn \l__iow_current_indentation_int \c_four
11154   \tl_put_right:Nx \exp_not:N \l__iow_current_indentation_tl
11155   { \c_space_tl \c_space_tl \c_space_tl \c_space_tl }
11156 }
11157 \cs_new_protected:Npn \__iow_wrap_unindent:
11158 {
11159   \int_sub:Nn \l__iow_current_indentation_int \c_four
11160   \tl_set:Nx \l__iow_current_indentation_tl
11161   { \prg_replicate:nn \l__iow_current_indentation_int { ~ } }
11162 }
11163 \cs_new_protected:Npn \__iow_wrap_end:
11164 {
11165   \tl_put_right:Nx \l__iow_wrap_tl
11166   { \l__iow_current_line_tl }
11167   \use_none_delimit_by_q_stop:w
11168 }

```

(End definition for __iow_wrap_special:w and others.)

20.5 Messages

```

11169 \__msg_kernel_new:nnnn { kernel } { file-not-found }
11170 { File~'#1'~not~found. }
11171 {
11172   The~requested~file~could~not~be~found~in~the~current~directory,~
11173   in~the~TeX~search~path~or~in~the~LaTeX~search~path.
11174 }
11175 \__msg_kernel_new:nnnn { kernel } { input-streams-exhausted }
11176 { Input~streams~exhausted }
11177 {
11178   TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
11179   All~16~are~currently~in~use,~and~something~wanted~to~open~
11180   another~one.
11181 }
11182 \__msg_kernel_new:nnnn { kernel } { output-streams-exhausted }
11183 { Output~streams~exhausted }
11184 {
11185   TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
11186   All~16~are~currently~in~use,~and~something~wanted~to~open~
11187   another~one.
11188 }
11189 \__msg_kernel_new:nnnn { kernel } { unbalanced-quote-in-filename }
11190 { Unbalanced~quotes~in~file~name~'#1'. }
11191 {

```



```

11192     File~names~must~contain~balanced~numbers~of~quotes~(").
11193   }
11194   \__msg_kernel_new:nnn { kernel } { indent-outside-wrapping-code }
11195   { Only~\iow_wrap:nnnN~(arg~1)~allows~\iow_indent:n }

```

20.6 Deprecatd functions

`\ior_get_str:NN` For removal after 2017-12-31.

```

11196 \cs_new_eq:NN \ior_get_str:NN \ior_str_get:NN

```

(*End definition for \ior_get_str:NN.*)

```

11197 </initex | package>

```

21 l3fp implementation

Nothing to see here: everything is in the subfiles!

22 l3fp-aux implementation

```

11198 <*initex | package>

```

```

11199 <@@=fp>

```

22.1 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 will be used in expressions, and in this context will be 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`.

Since floating point numbers are always accessed by the various operations using f-expansion, we can safely let them be protected: x-expansion will then leave them untouched. However, when used directly without an accessor function, floating points should produce an error. `\s__fp` will do nothing, and `__fp_chk:w` produces an error.

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. The various possibilities will be distinguished by their $\langle case \rangle$, which is a single digit:⁹

0 zeros: `+0` and `-0`,

1 “normal” numbers (positive and negative),

2 infinities: `+inf` and `-inf`,

3 quiet and signalling `nan`.

⁹Bruno: I need to implement subnormal numbers. Also, quiet and signalling `nan` must be better distinguished.

Table 1: 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.

The $\langle sign \rangle$ is 0 (positive) or 2 (negative), except in the case of **nan**, which have $\langle sign \rangle = 1$. This ensures that changing the $\langle sign \rangle$ digit to $2 - \langle sign \rangle$ is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

$$\backslash s_fp \backslash_fp_chk:w \langle case \rangle \langle sign \rangle \backslash s_fp_... ;$$

where $\backslash s_fp_...$ is a scan mark carrying information about how the number was formed (useful for debugging).

Normal floating point numbers ($\langle case \rangle = 1$) have the form

$$\backslash s_fp \backslash_fp_chk:w 1 \langle sign \rangle \{\langle exponent \rangle\} \{\langle X_1 \rangle\} \{\langle X_2 \rangle\} \{\langle X_3 \rangle\} \{\langle X_4 \rangle\} ;$$

Here, the $\langle exponent \rangle$ is an integer, at most $\backslash c_fp_max_exponent_int = 10000$ in absolute value. The body consists in four blocks of exactly 4 digits, $0000 \leq \langle X_i \rangle \leq 9999$, such that

$$\langle X \rangle = (-1)^{\langle sign \rangle} 10^{-\langle exponent \rangle} \sum_{i=1}^4 \langle X_i \rangle 10^{-4i}$$

and such that the $\langle exponent \rangle$ is minimal. This implies $1000 \leq \langle X_1 \rangle \leq 9999$.

22.2 Internal storage of floating points numbers

A floating point number $\langle X \rangle$ is stored as

$$\backslash s_fp \backslash_fp_chk:w \langle case \rangle \langle sign \rangle \langle body \rangle ;$$

Here, $\langle case \rangle$ is 0 for ± 0 , 1 for normal numbers, 2 for $\pm \infty$, and 3 for **nan**, and $\langle sign \rangle$ is 0 for positive numbers, 1 for **nans**, and 2 for negative numbers. The $\langle body \rangle$ of normal numbers is $\{\langle exponent \rangle\} \{\langle X_1 \rangle\} \{\langle X_2 \rangle\} \{\langle X_3 \rangle\} \{\langle X_4 \rangle\}$, with

$$\langle X \rangle = (-1)^{\langle sign \rangle} 10^{-\langle exponent \rangle} \sum_i \langle X_i \rangle 10^{-4i}.$$

Calculations are done in base 10000, *i.e.* one myriad. The $\langle exponent \rangle$ lies between $\pm \backslash c_fp_max_exponent_int = \pm 10000$ inclusive.

Additionally, positive and negative floating point numbers may only be stored with $1000 \leq \langle X_1 \rangle < 10000$. This requirement is necessary in order to preserve accuracy and speed.

22.3 Using arguments and semicolons

<code>__fp_use_none_stop_f:n</code>	<p>This function removes an argument (typically a digit) and replaces it by <code>\exp_stop_f:</code>, a marker which stops f-type expansion.</p> <pre>11200 \cs_new:Npn __fp_use_none_stop_f:n #1 { \exp_stop_f: }</pre> <p>(End definition for <code>__fp_use_none_stop_f:n</code>.)</p>
<code>__fp_use_s:n</code> <code>__fp_use_s:nn</code>	<p>Those functions place a semicolon after one or two arguments (typically digits).</p> <pre>11201 \cs_new:Npn __fp_use_s:n #1 { #1; } 11202 \cs_new:Npn __fp_use_s:nn #1#2 { #1#2; }</pre> <p>(End definition for <code>__fp_use_s:n</code> and <code>__fp_use_s:nn</code>.)</p>
<code>__fp_use_none_until_s:w</code> <code>__fp_use_i_until_s:nw</code> <code>__fp_use_ii_until_s:nnw</code>	<p>Those functions select specific arguments among a set of arguments delimited by a semicolon.</p> <pre>11203 \cs_new:Npn __fp_use_none_until_s:w #1; { } 11204 \cs_new:Npn __fp_use_i_until_s:nw #1#2; {#1} 11205 \cs_new:Npn __fp_use_ii_until_s:nnw #1#2#3; {#2}</pre> <p>(End definition for <code>__fp_use_none_until_s:w</code>, <code>__fp_use_i_until_s:nw</code>, and <code>__fp_use_ii_until_s:nnw</code>.)</p>
<code>__fp_reverse_args:Nww</code>	<p>Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.</p> <pre>11206 \cs_new:Npn __fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }</pre> <p>(End definition for <code>__fp_reverse_args:Nww</code>.)</p>
<code>__fp_rrot:www</code>	<p>Rotate three arguments delimited by semicolons. This is the inverse (or the square) of the Forth primitive ROT.</p> <pre>11207 \cs_new:Npn __fp_rrot:www #1; #2; #3; { #2; #3; #1; }</pre> <p>(End definition for <code>__fp_rrot:www</code>.)</p>
<code>__fp_use_i:ww</code> <code>__fp_use_i:www</code>	<p>Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.</p> <pre>11208 \cs_new:Npn __fp_use_i:ww #1; #2; { #1; } 11209 \cs_new:Npn __fp_use_i:www #1; #2; #3; { #1; }</pre> <p>(End definition for <code>__fp_use_i:ww</code> and <code>__fp_use_i:www</code>.)</p>

22.4 Constants, and structure of floating points

<code>\s__fp</code> <code>__fp_chk:w</code>	<p>Floating points numbers all start with <code>\s__fp __fp_chk:w</code>, where <code>\s__fp</code> is equal to the TeX primitive <code>\relax</code>, and <code>__fp_chk:w</code> is protected. The rest of the floating point number is made of characters (or <code>\relax</code>). This ensures that nothing expands under f-expansion, nor under x-expansion. However, when typeset, <code>\s__fp</code> does nothing, and <code>__fp_chk:w</code> is expanded. We define <code>__fp_chk:w</code> to produce an error.</p> <pre>11210 __scan_new:N \s__fp 11211 \cs_new_protected:Npn __fp_chk:w #1 ; 11212 { 11213 __msg_kernel_error:nnx { kernel } { misused-fp } 11214 { \fp_to_tl:n { \s__fp __fp_chk:w #1 ; } } 11215 }</pre>
---	--

(End definition for `\s__fp` and `__fp_chk:w`.)

`\s__fp_mark` Aliases of `\tex_relax:D`, used to terminate expressions.

```
\s__fp_stop 11216 \__scan_new:N \s__fp_mark
            11217 \__scan_new:N \s__fp_stop
```

(End definition for `\s__fp_mark` and `\s__fp_stop`.)

`\s__fp_invalid` A couple of scan marks used to indicate where special floating point numbers come from.

```
\s__fp_underflow 11218 \__scan_new:N \s__fp_invalid
\s__fp_overflow 11219 \__scan_new:N \s__fp_underflow
\s__fp_division 11220 \__scan_new:N \s__fp_overflow
\s__fp_exact 11221 \__scan_new:N \s__fp_division
            11222 \__scan_new:N \s__fp_exact
```

(End definition for `\s__fp_invalid` and others.)

`\c_zero_fp` The special floating points. All of them have the form

`\c_minus_zero_fp` `\s__fp __fp_chk:w <case> <sign> \s__fp... ;`

`\c_inf_fp`

`\c_minus_inf_fp`

`\c_nan_fp`

where the dots in `\s__fp...` are one of `invalid`, `underflow`, `overflow`, `division`, `exact`, describing how the floating point was created. We define the floating points here as “exact”.

```
11223 \tl_const:Nn \c_zero_fp { \s__fp \__fp_chk:w 0 0 \s__fp_exact ; }
11224 \tl_const:Nn \c_minus_zero_fp { \s__fp \__fp_chk:w 0 2 \s__fp_exact ; }
11225 \tl_const:Nn \c_inf_fp { \s__fp \__fp_chk:w 2 0 \s__fp_exact ; }
11226 \tl_const:Nn \c_minus_inf_fp { \s__fp \__fp_chk:w 2 2 \s__fp_exact ; }
11227 \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 187.)

`\c__fp_max_exponent_int` Normal floating point numbers have an exponent at most `max_exponent` in absolute value. Larger numbers are rounded to $\pm\infty$. Smaller numbers are subnormal (not implemented yet), and digits beyond $10^{-\text{max_exponent}}$ are rounded away, hence the true minimum exponent is $-\text{max_exponent} - 16$; beyond this, numbers are rounded to zero. Why this choice of limits? When computing $(a \cdot 10^n)(b \cdot 10^p)$, we need to evaluate $\log(a \cdot 10^n) = \log(a) + n \log(10)$ as a fixed point number, which we manipulate as blocks of 4 digits. Multiplying such a fixed point number by $n < 10000$ is much cheaper than larger n , because we can multiply n with each block safely.

```
11228 \int_const:Nn \c__fp_max_exponent_int { 10000 }
```

(End definition for `\c__fp_max_exponent_int`.)

`__fp_zero_fp:N` In case of overflow or underflow, we have to output a zero or infinity with a given sign.

```
\__fp_inf_fp:N 11229 \cs_new:Npn \__fp_zero_fp:N #1
                { \s__fp \__fp_chk:w 0 #1 \s__fp_underflow ; }
                11230
                11231 \cs_new:Npn \__fp_inf_fp:N #1
                { \s__fp \__fp_chk:w 2 #1 \s__fp_overflow ; }
                11232
```

(End definition for `__fp_zero_fp:N` and `__fp_inf_fp:N`.)

`__fp_max_fp:N` In some cases, we need to output the smallest or biggest positive or negative finite numbers.
`__fp_min_fp:N`

```

11233 \cs_new:Npn \__fp_min_fp:N #1
11234 {
11235   \s__fp \__fp_chk:w 1 #1
11236   { \int_eval:n { - \c__fp_max_exponent_int } }
11237   {1000} {0000} {0000} {0000} ;
11238 }
11239 \cs_new:Npn \__fp_max_fp:N #1
11240 {
11241   \s__fp \__fp_chk:w 1 #1
11242   { \int_use:N \c__fp_max_exponent_int }
11243   {9999} {9999} {9999} {9999} ;
11244 }
```

(End definition for __fp_max_fp:N and __fp_min_fp:N.)

`__fp_exponent:w` For normal numbers, the function expands to the exponent, otherwise to 0.

```

11245 \cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
11246 {
11247   \if_meaning:w 1 #1
11248     \exp_after:wN \__fp_use_ii_until_s:nnw
11249   \else:
11250     \exp_after:wN \__fp_use_i_until_s:nw
11251     \exp_after:wN 0
11252   \fi:
11253 }
```

(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.

```

11254 \cs_new:Npn \__fp_neg_sign:N #1
11255 { \__int_eval:w \c_two - #1 \__int_eval_end: }
```

(End definition for __fp_neg_sign:N.)

22.5 Overflow, underflow, and exact zero

`__fp_sanitize:Nw` 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 ± 0
`__fp_sanitize:wN` or overflowed to $\pm\infty$. The functions `__fp_underflow:w` and `__fp_overflow:w` are
`__fp_sanitize_zero:w` defined in l3fp-traps.

```

11256 \cs_new:Npn \__fp_sanitize:Nw #1 #2;
11257 {
11258   \if_case:w
11259     \if_int_compare:w #2 > \c__fp_max_exponent_int \c_one \else:
11260     \if_int_compare:w #2 < - \c__fp_max_exponent_int \c_two \else:
11261     \if_meaning:w 1 #1 \c_three \else: \c_zero \fi: \fi: \fi:
11262   \or: \exp_after:wN \__fp_overflow:w
11263   \or: \exp_after:wN \__fp_underflow:w
11264   \or: \exp_after:wN \__fp_sanitize_zero:w
11265   \fi:
```

```

11266 \s__fp \__fp_chk:w 1 #1 {#2}
11267 }
11268 \cs_new:Npn \__fp_sanitizewN #1; #2 { \__fp_sanitizewN #2 #1; }
11269 \cs_new:Npn \__fp_sanitizewZero:w \s__fp \__fp_chk:w #1 #2 #3;
11270 { \c_zero_fp }

(End definition for \__fp_sanitizewN, \__fp_sanitizewN, and \__fp_sanitizewZero:w.)

```

22.6 Expanding after a floating point number

`__fp_exp_after_o:w` Places *<tokens>* (empty in the case of `__fp_exp_after_o:w`) between the *<floating point>* and the *<more tokens>*, then hits those tokens with either o-expansion (one `\exp_after:wN`) 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.

```

11271 \cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
11272 {
11273   \if_meaning:w 1 #1
11274     \exp_after:wN \__fp_exp_after_normal:nNNw
11275   \else:
11276     \exp_after:wN \__fp_exp_after_special:nNNw
11277   \fi:
11278   { }
11279   #1
11280 }
11281 \cs_new:Npn \__fp_exp_after_o:nw #1 \s__fp \__fp_chk:w #2
11282 {
11283   \if_meaning:w 1 #2
11284     \exp_after:wN \__fp_exp_after_normal:nNNw
11285   \else:
11286     \exp_after:wN \__fp_exp_after_special:nNNw
11287   \fi:
11288   { #1 }
11289   #2
11290 }
11291 \cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
11292 {
11293   \if_meaning:w 1 #2
11294     \exp_after:wN \__fp_exp_after_normal:nNNw
11295   \else:
11296     \exp_after:wN \__fp_exp_after_special:nNNw
11297   \fi:
11298   { \exp:w \exp_end_continue_f:w #1 }
11299   #2
11300 }

```

(End definition for `__fp_exp_after_o:w`, `__fp_exp_after_o:nw`, and `__fp_exp_after_f:nw`.)

`__fp_exp_after_special:nNNw` Special floating point numbers are easy to jump over since they contain few tokens.

```

11301 \cs_new:Npn \__fp_exp_after_special:nNNw #1#2#3#4;
11302 {
11303   \exp_after:wN \s__fp
11304   \exp_after:wN \__fp_chk:w
11305   \exp_after:wN #2

```

```

11306     \exp_after:wN #3
11307     \exp_after:wN #4
11308     \exp_after:wN ;
11309     #1
11310 }

```

(End definition for `_fp_exp_after_special:nNNw`.)

`_fp_exp_after_normal: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.

```

11311 \cs_new:Npn \_fp_exp_after_normal:nNNw #1 1 #2 #3 #4#5#6#7;
11312 {
11313     \exp_after:wN \_fp_exp_after_normal:Nwwwww
11314     \exp_after:wN #2
11315     \__int_value:w #3 \exp_after:wN ;
11316     \__int_value:w 1 #4 \exp_after:wN ;
11317     \__int_value:w 1 #5 \exp_after:wN ;
11318     \__int_value:w 1 #6 \exp_after:wN ;
11319     \__int_value:w 1 #7 \exp_after:wN ; #1
11320 }
11321 \cs_new:Npn \_fp_exp_after_normal:Nwwwww
11322     #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
11323     { \s_fp \_fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }

```

(End definition for `_fp_exp_after_normal:nNNw`.)

`_fp_exp_after_array_f:w`

`_fp_exp_after_stop_f:nw`

```

11324 \cs_new:Npn \_fp_exp_after_array_f:w #1
11325 {
11326     \cs:w \_fp_exp_after \_fp_type_from_scan:N #1 _f:nw \cs_end:
11327     { \_fp_exp_after_array_f:w }
11328     #1
11329 }
11330 \cs_new_eq:NN \_fp_exp_after_stop_f:nw \use_none:nn

```

(End definition for `_fp_exp_after_array_f:w` and `_fp_exp_after_stop_f:nw`.)

22.7 Packing digits

When a positive integer `#1` is known to be less than 10^8 , the following trick will split 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
\__int_value:w \__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 $\text{T}_{\text{E}}\text{X}$'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 $1\,2345 \times 6677\,8899$. With simplified names, we would do

```

\exp_after:wN \post_processing:w
\__int_value:w \__int_eval:w - 5 0000
\exp_after:wN \pack:NNNNNw
\__int_value:w \__int_eval:w 4 9995 0000
+ 12345 * 6677
\exp_after:wN \pack:NNNNNw
\__int_value:w \__int_eval:w 5 0000 0000
+ 12345 * 8899 ;

```

The `\exp_after:wN` triggers `__int_value:w __int_eval:w`, which starts a first computation, whose initial value is $-5\,0000$ (the “leading shift”). In that computation appears an `\exp_after:wN`, which triggers the nested computation `__int_value:w __int_eval:w` with starting value $4\,9995\,0000$ (the “middle shift”). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation’s value is $5\,0000\,0000 + 12345 \times 8899$, 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 will also work 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 $5\,0000\,0000/10^4 + 4\,9995\,0000 = 5\,0000\,0000$. As long as the operands are in some range, the result of this second computation will have 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.

```

\__fp_pack:NNNNNw This set of shifts allows for computations involving results in the range  $[-4 \cdot 10^8, 5 \cdot 10^8 - 1]$ .
\c_fp_trailing_shift_int Shifted values all have exactly 9 digits.
\c_fp_middle_shift_int
\c_fp_leading_shift_int
11331 \int_const:Nn \c_fp_leading_shift_int { - 5 0000 }
11332 \int_const:Nn \c_fp_middle_shift_int { 5 0000 * 9999 }
11333 \int_const:Nn \c_fp_trailing_shift_int { 5 0000 * 10000 }
11334 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }

```

(End definition for `__fp_pack:NNNNNw` and others.)

```

\__fp_pack_big:NNNNNNw This set of shifts allows for computations involving results in the range  $[-5 \cdot 10^8, 6 \cdot 10^8 - 1]$ 
\c_fp_big_trailing_shift_int (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper
\c_fp_big_middle_shift_int bound is due to  $\text{\TeX}$ ’s limit of  $2^{31} - 1$  on integers. The shifts are chosen to be roughly
\c_fp_big_leading_shift_int the mid-point of  $10^9$  and  $2^{31}$ , the two bounds on 10-digit integers in  $\text{\TeX}$ .
11335 \int_const:Nn \c_fp_big_leading_shift_int { - 15 2374 }
11336 \int_const:Nn \c_fp_big_middle_shift_int { 15 2374 * 9999 }
11337 \int_const:Nn \c_fp_big_trailing_shift_int { 15 2374 * 10000 }
11338 \cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
11339 { + #1#2#3#4#5#6 ; {#7} }

```

(End definition for `__fp_pack_big:NNNNNNw` and others.)

`_fp_pack_Bigg:NNNNNNw` 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.

`\c_fp_Bigg_trailing_shift_int`

`\c_fp_Bigg_middle_shift_int`

`\c_fp_Bigg_leading_shift_int`

```

11340 \int_const:Nn \c\_fp\_Bigg\_leading\_shift\_int { - 20 0000 }
11341 \int_const:Nn \c\_fp\_Bigg\_middle\_shift\_int { 20 0000 * 9999 }
11342 \int_const:Nn \c\_fp\_Bigg\_trailing\_shift\_int { 20 0000 * 10000 }
11343 \cs_new:Npn \_fp\_pack\_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
11344 { + #1#2#3#4#5#6 ; {#7} }

```

(End definition for `_fp_pack_Bigg:NNNNNNw` and others.)

`_fp_pack_twice_four:wNNNNNNNN` Grabs two sets of 4 digits and places them before the semi-colon delimiter. Putting several copies of this function before a semicolon will pack more digits since each will take the digits packed by the others in its first argument.

```

11345 \cs_new:Npn \_fp\_pack\_twice\_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
11346 { #1 {#2#3#4#5} {#6#7#8#9} ; }

```

(End definition for `_fp_pack_twice_four:wNNNNNNNN`.)

`_fp_pack_eight:wNNNNNNNN` 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 will pack more digits since each will take the digits packed by the others in its first argument.

```

11347 \cs_new:Npn \_fp\_pack\_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
11348 { #1 {#2#3#4#5#6#7#8#9} ; }

```

(End definition for `_fp_pack_eight:wNNNNNNNN`.)

22.8 Decimate (dividing by a power of 10)

`_fp_decimate:nNnnnn` Each $\langle X_i \rangle$ consists in 4 digits exactly, and $1000 \leq \langle X_1 \rangle < 9999$. The first argument determines by how much we shift the digits. $\langle f_1 \rangle$ is called as follows: where $0 \leq \langle X'_i \rangle < 10^8 - 1$ are 8 digit numbers, forming the truncation of our number. In other words,

$$\left(\sum_{i=1}^4 \langle X_i \rangle \cdot 10^{-4i} \cdot 10^{-\langle shift \rangle} - \langle X'_1 \rangle \cdot 10^{-8} + \langle X'_2 \rangle \cdot 10^{-16} \right) \in [0, 10^{-16}).$$

To round properly later, we need to remember some information about the difference. The $\langle rounding \rangle$ 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, $\langle rounding \rangle$ is 1 (not 0), and $\langle X'_1 \rangle \langle X'_2 \rangle$ are both zero.

If the shift is 1, the $\langle rounding \rangle$ 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 $\langle rounding \rangle$ digit to be placed after the $\langle X_i \rangle$, but the choice we make involves less reshuffling.

Note that this function fails for negative $\langle shift \rangle$.

```

11349 \cs_new:Npn \_fp\_decimate:nNnnnn #1
11350 {
11351   \cs:w
11352   \_fp\_decimate\_
11353   \if_int_compare:w \_int\_eval:w #1 > \c\_sixteen
11354     tiny
11355   \else:

```

```

11356         \_int_to_roman:w \_int_eval:w #1
11357         \fi:
11358         :Nnnnn
11359         \cs_end:
11360     }

```

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.

(End definition for `_fp_decimate:nNnnnn`.)

```

\_fp_decimate_:Nnnnn
\_fp_decimate_tiny:Nnnnn
11361 \cs_new:Npn \_fp_decimate_:Nnnnn #1 #2#3#4#5
11362 { #1 0 {#2#3} {#4#5} ; }
11363 \cs_new:Npn \_fp_decimate_tiny:Nnnnn #1 #2#3#4#5
11364 { #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
\_fp_decimate_auxii:Nnnnn
\_fp_decimate_auxiii:Nnnnn
\_fp_decimate_auxiv:Nnnnn
\_fp_decimate_auxv:Nnnnn
\_fp_decimate_auxvi:Nnnnn
\_fp_decimate_auxvii:Nnnnn
\_fp_decimate_auxviii:Nnnnn
\_fp_decimate_auxix:Nnnnn
\_fp_decimate_auxx:Nnnnn
\_fp_decimate_auxxi:Nnnnn
\_fp_decimate_auxxii:Nnnnn
\_fp_decimate_auxxiii:Nnnnn
\_fp_decimate_auxxiv:Nnnnn
\_fp_decimate_auxxv:Nnnnn
\_fp_decimate_auxxvi:Nnnnn
11365 \cs_new:Npn \_fp_tmp:w #1 #2 #3
11366 {
11367     \cs_new:cpn { \_fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
11368     {
11369         \exp_after:wN ##1
11370         \_int_value:w
11371         \exp_after:wN \_fp_round_digit:Nw #2 ;
11372         \_fp_decimate_pack:nnnnnnnnnw #3 ;
11373     }
11374 }
11375 \_fp_tmp:w {i} {\use_none:nnn #50}{ 0{#2}#3{#4}#5 }
11376 \_fp_tmp:w {ii} {\use_none:nn #5 }{ 00{#2}#3{#4}#5 }
11377 \_fp_tmp:w {iii} {\use_none:n #5 }{ 000{#2}#3{#4}#5 }
11378 \_fp_tmp:w {iv} { #5 }{ {0000}#2{#3}#4 #5 }
11379 \_fp_tmp:w {v} {\use_none:nnn #4#5 }{ 0{0000}#2{#3}#4 #5 }
11380 \_fp_tmp:w {vi} {\use_none:nn #4#5 }{ 00{0000}#2{#3}#4 #5 }
11381 \_fp_tmp:w {vii} {\use_none:n #4#5 }{ 000{0000}#2{#3}#4 #5 }
11382 \_fp_tmp:w {viii}{ #4#5 }{ {0000}0000{#2}#3 #4 #5 }
11383 \_fp_tmp:w {ix} {\use_none:nnn #3#4+#5}{ 0{0000}0000{#2}#3 #4 #5 }
11384 \_fp_tmp:w {x} {\use_none:nn #3#4+#5}{ 00{0000}0000{#2}#3 #4 #5 }
11385 \_fp_tmp:w {xi} {\use_none:n #3#4+#5}{ 000{0000}0000{#2}#3 #4 #5 }
11386 \_fp_tmp:w {xii} { #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5 }
11387 \_fp_tmp:w {xiii}{\use_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5 }
11388 \_fp_tmp:w {xiv} {\use_none:nn #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5 }
11389 \_fp_tmp:w {xv} {\use_none:n #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5 }
11390 \_fp_tmp:w {xvi} { #2#3+#4#5}{ {0000}0000{0000}0000 #2 #3 #4 #5 }

```

¹⁰No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

(End definition for `_fp_decimate_auxi:Nnnnn` and others.)

`_fp_round_digit:Nw` `_fp_decimate_pack:nnnnnnnnnw` `_fp_round_digit:Nw` will receive the “extra digits” as its argument, and its expansion is triggered by `_int_value:w`. If the first digit is neither 0 nor 5, then it is the *rounding* digit. Otherwise, if the remaining digits are not all zero, we need to add 1 to that leading digit to get the rounding digit. Some caution is required, though, because there may be more than 10 “extra digits”, and this may overflow TeX’s integers. Instead of feeding the digits directly to `_fp_round_digit:Nw`, they come split into several blocks, separated by +. Hence the first `_int_eval:w` here.

The computation of the *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.

```
11391 \cs_new:Npn \_fp\_decimate\_pack:nnnnnnnnnw #1#2#3#4#5
11392 { \_fp\_decimate\_pack:nnnnnnw { #1#2#3#4#5 } }
11393 \cs_new:Npn \_fp\_decimate\_pack:nnnnnnw #1 #2#3#4#5#6
11394 { {#1} {#2#3#4#5#6} }
```

(End definition for `_fp_round_digit:Nw` and `_fp_decimate_pack:nnnnnnnnnw`.)

22.9 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 In this example, the case 0 will return the floating point *fp var*, expanding once after that floating point. Case 1 will do *some computation* using the *floating point* (presumably compute the operation requested by the user in that non-trivial case). Case 2 will return the *floating point* without modifying it, removing the *junk* and expanding once after. Case 3 will close the conditional, remove the *junk* and the *floating point*, and expand *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.

```
11395 \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.

```
11396 \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 T_EX conditional, removes junk and a floating point, and returns its first argument (an *<fp var>*) then expands once after it.

```
11397 \cs_new:Npn \_fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
11398 { \fi: \exp_after:wN #1 }
```

(End definition for _fp_case_return_o:Nw.)

_fp_case_return_same_o:w This function ends a T_EX conditional, removes junk, and returns the following floating point, expanding once after it.

```
11399 \cs_new:Npn \_fp_case_return_same_o:w #1 \fi: #2 \s__fp
11400 { \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.

```
11401 \cs_new:Npn \_fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
11402 { \fi: \exp_after:wN #1 }
```

(End definition for _fp_case_return_o:Nww.)

_fp_case_return_i_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.

```
11403 \cs_new:Npn \_fp_case_return_i_o:ww #1 \fi: #2 \s__fp #3 ; \s__fp #4 ;
11404 { \fi: \_fp_exp_after_o:w \s__fp #3 ; }
11405 \cs_new:Npn \_fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 ;
11406 { \fi: \_fp_exp_after_o:w }
```

(End definition for _fp_case_return_i_o:ww and _fp_case_return_ii_o:ww.)

22.10 Integer floating points

_fp_int_p:w Tests if the floating point argument is an integer. This holds if the rounding digit resulting from _fp_decimate:nNnnnn is 0.

```
\_fp_int_normal:nnnnn
\_fp_int_test:Nw
11407 \prg_new_conditional:Npnn \_fp_int:w \s__fp \_fp_chk:w #1 #2 #3; { TF , T , F , p }
11408 {
11409   \if_case:w #1 \exp_stop_f:
11410     \prg_return_true:
11411   \or: \_fp_int_normal:nnnnn #3
11412   \else: \prg_return_false:
11413   \fi:
11414 }
11415 \cs_new:Npn \_fp_int_normal:nnnnn #1 #2#3#4#5
11416 {
11417   \if_int_compare:w #1 > \c_zero
11418     \_fp_decimate:nNnnnn { \c_sixteen - #1 }
11419     \_fp_int_test:Nw
11420     {#2} {#3} {#4} {#5}
11421   \else:
11422     \prg_return_false:
11423   \fi:
11424 }
11425 \cs_new:Npn \_fp_int_test:Nw #1#2;
```

```

11426 {
11427   \if_meaning:w 0 #1
11428     \prg_return_true:
11429   \else:
11430     \prg_return_false:
11431   \fi:
11432 }

```

(End definition for `_fp_int:wTF`, `_fp_int_normal:nnnnn`, and `_fp_int_test:Nw`.)

22.11 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 converted 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: neither `nan` nor infinities are integers. Normal numbers with a non-positive exponent are never integers. When the exponent is greater than 8, the number is too large for the range. Otherwise, decimate, and test the digits after the decimal separator. The `\use_iii:nnn` remove a trailing `;` and the true branch, leaving only the false branch. The `_int_value:w` appearing in the case where the normal floating point is an integer takes care of expanding all the conditionals until the trailing `;`. That integer is fed to `_fp_small_int_true:wTF` which places it as a braced argument of the true branch. The `\use_i:nn` in `_fp_small_int_test:NnnwNTF` removes the top-level `\else:` coming from `_fp_small_int_normal:NnwTF`, hence will call the `\use_iii:nnn` which follows, taking the false branch.

```

11433 \cs_new:Npn \_fp_small_int:wTF \s_fp \_fp_chk:w #1#2
11434 {
11435   \if_case:w #1 \exp_stop_f:
11436     \_fp_case_return:nw { \_fp_small_int_true:wTF 0 ; }
11437   \or: \exp_after:wN \_fp_small_int_normal:NnwTF
11438   \or:
11439     \_fp_case_return:nw
11440     {
11441       \exp_after:wN \_fp_small_int_true:wTF \_int_value:w
11442       \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
11443     }
11444   \else: \_fp_case_return:nw \use_ii:nn
11445   \fi:
11446   #2
11447 }
11448 \cs_new:Npn \_fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
11449 \cs_new:Npn \_fp_small_int_normal:NnwTF #1#2#3;
11450 {
11451   \if_int_compare:w #2 > \c_zero
11452     \_fp_decimate:nNnnnn { \c_sixteen - #2 }
11453     \_fp_small_int_test:NnnwNnw
11454     #3 #1 {#2}
11455   \else:
11456     \exp_after:wN \use_iii:nnn
11457   \fi:
11458   ;
11459 }
11460 \cs_new:Npn \_fp_small_int_test:NnnwNTF #1#2#3#4; #5#6
11461 {

```

```

11462 \if_meaning:w 0 #1
11463 \exp_after:wN \__fp_small_int_true:wTF
11464 \__int_value:w \if_meaning:w 2 #5 - \fi:
11465 \if_int_compare:w #6 > \c_eight
11466 1 0000 0000
11467 \else:
11468 #3
11469 \fi:
11470 \else:
11471 \use_i:nn
11472 \fi:
11473 }

```

(End definition for `__fp_small_int:wTF` and others.)

22.12 Length of a floating point array

`__fp_array_count:n` Count the number of items in an array of floating points. 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.

```

11474 \cs_new:Npn \__fp_array_count:n #1
11475 {
11476 \__int_value:w \__int_eval:w \c_zero
11477 \__fp_array_count_loop:Nw #1 { ? \__prg_break: } ;
11478 \__prg_break_point:
11479 \__int_eval_end:
11480 }
11481 \cs_new:Npn \__fp_array_count_loop:Nw #1#2;
11482 { \use_none:n #1 + \c_one \__fp_array_count_loop:Nw }

```

(End definition for `__fp_array_count:n` and `__fp_array_count_loop:Nw`.)

22.13 x-like expansion expandably

`__fp_expand:n` This expandable function behaves in a way somewhat similar to `\use:x`, but much less robust. The argument is f-expanded, then the leading item (often a single character token) is moved to a storage area after `\s__fp_mark`, and f-expansion is applied again, repeating until the argument is empty. The result built one piece at a time is then inserted in the input stream. Note that spaces are ignored by this procedure, unless surrounded with braces. Multiple tokens which do not need expansion can be inserted within braces.

```

11483 \cs_new:Npn \__fp_expand:n #1
11484 {
11485 \__fp_expand_loop:nwnN { }
11486 #1 \prg_do_nothing:
11487 \s__fp_mark { } \__fp_expand_loop:nwnN
11488 \s__fp_mark { } \__fp_use_i_until_s:nw ;
11489 }
11490 \cs_new:Npn \__fp_expand_loop:nwnN #1#2 \s__fp_mark #3 #4
11491 {
11492 \exp_after:wN #4 \exp:w \exp_end_continue_f:w
11493 #2
11494 \s__fp_mark { #3 #1 } #4
11495 }

```

(End definition for `_fp_expand:n` and `_fp_expand_loop:nwnN`.)

22.14 Messages

Using a floating point directly is an error.

```

11496 \_msg_kernel_new:nnnn { kernel } { misused-fp }
11497 { A~floating~point~with~value~'#1'~was~misused. }
11498 {
11499   To~obtain~the~value~of~a~floating~point~variable,~use~
11500   '\token_to_str:N \fp_to_decimal:N',~
11501   '\token_to_str:N \fp_to_scientific:N',~or~other~
11502   conversion~functions.
11503 }
11504 </initex | package>

```

23 l3fp-traps Implementation

```

11505 <*initex | package>
11506 <@@=fp>

```

Exceptions should be accessed by an `n`-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

23.1 Flags

`\fp_flag_off:n` Function to turn a flag off. Simply undefine it.

```

11507 \cs_new_protected:Npn \fp_flag_off:n #1
11508 { \cs_set_eq:cn { l__fp_ #1 _flag_token } \tex_undefined:D }

```

(End definition for `\fp_flag_off:n`. This function is documented on page 189.)

`\fp_flag_on:n` Function to turn a flag on expandably: use $\mathrm{T}_{\mathrm{E}}\mathrm{X}$'s automatic assignment to `\scan_stop:`.

```

11509 \cs_new:Npn \fp_flag_on:n #1
11510 { \exp_args:Nc \use_none:n { l__fp_ #1 _flag_token } }

```

(End definition for `\fp_flag_on:n`. This function is documented on page 189.)

`\fp_if_flag_on_p:n` Returns true if the flag is on, false otherwise.

`\fp_if_flag_on:nTF`

```

11511 \prg_new_conditional:Npnn \fp_if_flag_on:n #1 { p , T , F , TF }
11512 {
11513   \if_cs_exist:w l__fp_ #1 _flag_token \cs_end:
11514     \prg_return_true:
11515   \else:
11516     \prg_return_false:
11517   \fi:
11518 }

```

(End definition for `\fp_if_flag_on:nTF`. This function is documented on page 189.)

`\l_fp_invalid_operation_flag_token`
`\l_fp_division_by_zero_flag_token`
`\l_fp_overflow_flag_token`
`\l_fp_underflow_flag_token`

The IEEE standard defines five exceptions. We currently don't support the “inexact” exception.

```
11519 \cs_new_eq:NN \l_fp_invalid_operation_flag_token \tex_undefined:D
11520 \cs_new_eq:NN \l_fp_division_by_zero_flag_token \tex_undefined:D
11521 \cs_new_eq:NN \l_fp_overflow_flag_token \tex_undefined:D
11522 \cs_new_eq:NN \l_fp_underflow_flag_token \tex_undefined:D
```

(End definition for `\l_fp_invalid_operation_flag_token` and others.)

23.2 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`

```
11523 \cs_new_protected:Npn \fp_trap:nn #1#2
11524 {
11525   \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
11526   {
11527     \clist_if_in:nnTF
11528     { invalid_operation , division_by_zero , overflow , underflow }
11529     {#1}
11530     {
11531       \__msg_kernel_error:nnxx { kernel }
11532       { unknown-fpu-trap-type } {#1} {#2}
```



```

11533     }
11534     {
11535         \_msg_kernel_error:nxx
11536         { kernel } { unknown-fpu-exception } {#1}
11537     }
11538 }
11539 }

```

(End definition for \fp_trap:nn. This function is documented on page 189.)

_fp_trap_invalid_operation_set_error: We provide three types of trapping for invalid operations: either produce an error and
_fp_trap_invalid_operation_set_flag: raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases,
_fp_trap_invalid_operation_set_none: the function produces as a result its first argument, possibly with post-expansion.
_fp_trap_invalid_operation_set:N

```

11540 \cs_new_protected:Npn \_fp_trap_invalid_operation_set_error:
11541 { \_fp_trap_invalid_operation_set:N \prg_do_nothing: }
11542 \cs_new_protected:Npn \_fp_trap_invalid_operation_set_flag:
11543 { \_fp_trap_invalid_operation_set:N \use_none:nnnnn }
11544 \cs_new_protected:Npn \_fp_trap_invalid_operation_set_none:
11545 { \_fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
11546 \cs_new_protected:Npn \_fp_trap_invalid_operation_set:N #1
11547 {
11548     \exp_args:Nno \use:n
11549     { \cs_set:Npn \_fp_invalid_operation:nnw ##1##2##3; }
11550     {
11551         #1
11552         \_fp_error:nmfn { invalid } {##2} { \fp_to_tl:n { ##3; } } { }
11553         \fp_flag_on:n { invalid_operation }
11554         ##1
11555     }
11556     \exp_args:Nno \use:n
11557     { \cs_set:Npn \_fp_invalid_operation_o:Nww ##1##2; ##3; }
11558     {
11559         #1
11560         \_fp_error:nffn { invalid-ii }
11561         { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
11562         \fp_flag_on:n { invalid_operation }
11563         \exp_after:wN \c_nan_fp
11564     }
11565     \exp_args:Nno \use:n
11566     { \cs_set:Npn \_fp_invalid_operation_tl_o:ff ##1##2 }
11567     {
11568         #1
11569         \_fp_error:nffn { invalid } {##1} {##2} { }
11570         \fp_flag_on:n { invalid_operation }
11571         \exp_after:wN \c_nan_fp
11572     }
11573 }

```

(End definition for _fp_trap_invalid_operation_set_error: and others.)

_fp_trap_division_by_zero_set_error: We provide three types of trapping for invalid operations and division by zero: either
_fp_trap_division_by_zero_set_flag: produce an error and raise the relevant flag; or only raise the flag; or don't even raise the
_fp_trap_division_by_zero_set_none: flag. In all cases, the function must produce a result, namely its first argument, $\pm\infty$ or
_fp_trap_division_by_zero_set:N NaN.

```

11574 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_error:
11575 { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
11576 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_flag:
11577 { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
11578 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_none:
11579 { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
11580 \cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
11581 {
11582   \exp_args:Nno \use:n
11583   { \cs_set:Npn \__fp_division_by_zero_o:NNw ##1##2##3; }
11584   {
11585     #1
11586     \__fp_error:nfn { zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
11587     \fp_flag_on:n { division_by_zero }
11588     \exp_after:wN ##1
11589   }
11590   \exp_args:Nno \use:n
11591   { \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3; ##4; }
11592   {
11593     #1
11594     \__fp_error:nfn { zero-div-ii }
11595     { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
11596     \fp_flag_on:n { division_by_zero }
11597     \exp_after:wN ##1
11598   }
11599 }

```

(End definition for `__fp_trap_division_by_zero_set_error:` and others.)

`_fp_trap_overflow_set_error:` 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 $10 \cdot 10^{9999}$, the exponent would be too large for T_EX, and `__fp_overflow:w` receives $\pm\infty$ (`__fp_underflow:w` would receive ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```

11600 \cs_new_protected:Npn \__fp_trap_overflow_set_error:
11601 { \__fp_trap_overflow_set:N \prg_do_nothing: }
11602 \cs_new_protected:Npn \__fp_trap_overflow_set_flag:
11603 { \__fp_trap_overflow_set:N \use_none:nnnnn }
11604 \cs_new_protected:Npn \__fp_trap_overflow_set_none:
11605 { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
11606 \cs_new_protected:Npn \__fp_trap_overflow_set:N #1
11607 { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
11608 \cs_new_protected:Npn \__fp_trap_underflow_set_error:
11609 { \__fp_trap_underflow_set:N \prg_do_nothing: }
11610 \cs_new_protected:Npn \__fp_trap_underflow_set_flag:
11611 { \__fp_trap_underflow_set:N \use_none:nnnnn }
11612 \cs_new_protected:Npn \__fp_trap_underflow_set_none:
11613 { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
11614 \cs_new_protected:Npn \__fp_trap_underflow_set:N #1
11615 { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }

```

```

11616 \cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
11617 {
11618   \exp_args:Nno \use:n
11619   { \cs_set:cpn { __fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3; }
11620   {
11621     #1
11622     \__fp_error:nffn
11623     { flow \if_meaning:w 1 ##1 -to \fi: }
11624     { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3; } }
11625     { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
11626     {#2}
11627     \fp_flag_on:n {#2}
11628     #3 ##2
11629   }
11630 }

```

(End definition for __fp_trap_overflow_set_error: and others.)

__fp_invalid_operation:nnw Initialize the two 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.

```

\__fp_division_by_zero_o:Nnw 11631 \cs_new:Npn \__fp_invalid_operation:nnw #1#2#3; { }
\__fp_division_by_zero_o:NNww 11632 \cs_new:Npn \__fp_invalid_operation_o:Nww #1#2; #3; { }
\__fp_overflow:w 11633 \cs_new:Npn \__fp_invalid_operation_tl_o:ff #1 #2 { }
\__fp_underflow:w 11634 \cs_new:Npn \__fp_division_by_zero_o:Nnw #1#2#3; { }
11635 \cs_new:Npn \__fp_division_by_zero_o:NNww #1#2#3; #4; { }
11636 \cs_new:Npn \__fp_overflow:w { }
11637 \cs_new:Npn \__fp_underflow:w { }
11638 \fp_trap:nn { invalid_operation } { error }
11639 \fp_trap:nn { division_by_zero } { flag }
11640 \fp_trap:nn { overflow } { flag }
11641 \fp_trap:nn { underflow } { flag }

```

(End definition for __fp_invalid_operation:nnw and others.)

__fp_invalid_operation_o:nw Convenient short-hands for returning \c_nan_fp for a unary or binary operation, and expanding after.

```

11642 \cs_new:Npn \__fp_invalid_operation_o:nw
11643 { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }
11644 \cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }

```

(End definition for __fp_invalid_operation_o:nw.)

23.3 Errors

```

\__fp_error:nnnn
\__fp_error:nnfn 11645 \cs_new:Npn \__fp_error:nnnn #1
\__fp_error:nffn 11646 { \__msg_kernel_expandable_error:nnnnn { kernel } { fp - #1 } }
11647 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff }

```

(End definition for __fp_error:nnnn.)

23.4 Messages

Some messages.

```
11648 \_msg_kernel_new:nnnn { kernel } { unknown-fpu-exception }
11649 {
11650     The~FPU~exception~'#1'~is~not~known:~
11651     that~trap~will~never~be~triggered.
11652 }
11653 {
11654     The~only~exceptions~to~which~traps~can~be~attached~are \\
11655     \iow_indent:n
11656     {
11657         * ~ invalid_operation \\
11658         * ~ division_by_zero \\
11659         * ~ overflow \\
11660         * ~ underflow
11661     }
11662 }
11663 \_msg_kernel_new:nnnn { kernel } { unknown-fpu-trap-type }
11664 { The~FPU~trap~type~'#2'~is~not~known. }
11665 {
11666     The~trap~type~must~be~one~of \\
11667     \iow_indent:n
11668     {
11669         * ~ error \\
11670         * ~ flag \\
11671         * ~ none
11672     }
11673 }
11674 \_msg_kernel_new:nnn { kernel } { fp-flow }
11675 { An ~ #3 ~ occurred. }
11676 \_msg_kernel_new:nnn { kernel } { fp-flow-to }
11677 { #1 ~ #3 ed ~ to ~ #2 . }
11678 \_msg_kernel_new:nnn { kernel } { fp-zero-div }
11679 { Division~by~zero~in~ #1 (#2) }
11680 \_msg_kernel_new:nnn { kernel } { fp-zero-div-ii }
11681 { Division~by~zero~in~ (#1) #3 (#2) }
11682 \_msg_kernel_new:nnn { kernel } { fp-invalid }
11683 { Invalid~operation~ #1 (#2) }
11684 \_msg_kernel_new:nnn { kernel } { fp-invalid-ii }
11685 { Invalid~operation~ (#1) #3 (#2) }
11686 </initex | package>
```

24 13fp-round implementation

```
11687 (*initex | package)
11688 <@@=fp>
```

24.1 Rounding tools

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 `\c_zero` or `\c_one` depending on whether the final result should be rounded up or down.

- `__fp_round:NNN <sign> <digit1> <digit2>` can expand to `\c_zero` or `\c_one`.
- `__fp_round_s:NNNw <sign> <digit1> <digit2> <more digits>;` can expand to `\c_zero` ; or `\c_one` ;.
- `__fp_round_neg:NNN <sign> <digit1> <digit2>` can expand to `\c_zero` or `\c_one`.

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 $\langle final\ sign \rangle \langle digit_1 \rangle . \langle digit_2 \rangle$ to an integer rounds it towards zero (truncates it), this function expands to `\c_zero`, and otherwise to `\c_one`. Typically used within the scope of an `__int_eval:w`, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.

It is very important that $\langle final\ sign \rangle$ be the final sign of the result. Otherwise, the result will be incorrect in the case of rounding towards $-\infty$ or towards $+\infty$. Also recall that $\langle final\ sign \rangle$ is 0 for positive, and 2 for negative.

By default, the functions below return `\c_zero`, but this is superseded by `__fp_round_return_one:`, which instead returns `\c_one`, expanding everything and removing `\c_zero` 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 $\langle digit_2 \rangle$ is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that $\langle digit_1 \rangle$ plus the result is even. In other words, round up if $\langle digit_1 \rangle$ 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 $+\infty$.

```
11689 \cs_new:Npn \__fp_round_return_one:
11690 { \exp_after:wN \c_one \exp:w }
11691 \cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
11692 {
11693   \if_meaning:w 2 #1
```

```

11694     \if_int_compare:w #3 > \c_zero
11695     \__fp_round_return_one:
11696     \fi:
11697 \fi:
11698 \c_zero
11699 }
11700 \cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { \c_zero }
11701 \cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
11702 {
11703     \if_meaning:w 0 #1
11704     \if_int_compare:w #3 > \c_zero
11705     \__fp_round_return_one:
11706     \fi:
11707 \fi:
11708 \c_zero
11709 }
11710 \cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
11711 {
11712     \if_int_compare:w #3 > \c_five
11713     \__fp_round_return_one:
11714 \else:
11715     \if_meaning:w 5 #3
11716     \if_int_odd:w #2 \exp_stop_f:
11717     \__fp_round_return_one:
11718     \fi:
11719 \fi:
11720 \fi:
11721 \c_zero
11722 }
11723 \cs_new:Npn \__fp_round_to_nearest_ninf:NNN #1 #2 #3
11724 {
11725     \if_int_compare:w #3 > \c_five
11726     \__fp_round_return_one:
11727 \else:
11728     \if_meaning:w 5 #3
11729     \if_meaning:w 2 #1
11730     \__fp_round_return_one:
11731     \fi:
11732 \fi:
11733 \fi:
11734 \c_zero
11735 }
11736 \cs_new:Npn \__fp_round_to_nearest_zero:NNN #1 #2 #3
11737 {
11738     \if_int_compare:w #3 > \c_five
11739     \__fp_round_return_one:
11740 \fi:
11741 \c_zero
11742 }
11743 \cs_new:Npn \__fp_round_to_nearest_pinf:NNN #1 #2 #3
11744 {
11745     \if_int_compare:w #3 > \c_five
11746     \__fp_round_return_one:
11747 \else:

```

```

11748     \if_meaning:w 5 #3
11749     \if_meaning:w 0 #1
11750         \__fp_round_return_one:
11751     \fi:
11752     \fi:
11753     \fi:
11754     \c_zero
11755 }
11756 \cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN

```

(End definition for __fp_round:NNN and others.)

__fp_round_s:NNNw Similar to __fp_round:NNN, but with an extra semicolon, this function expands to \c_zero ; if rounding *⟨final sign⟩⟨digit⟩.⟨more digits⟩* to an integer truncates, and to \c_one ; otherwise. The *⟨more digits⟩* part must be a digit, followed by something that does not overflow a \int_use:N __int_eval:w construction. The only relevant information about this piece is whether it is zero or not.

```

11757 \cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4;
11758 {
11759     \exp_after:wN \__fp_round:NNN
11760     \exp_after:wN #1
11761     \exp_after:wN #2
11762     \__int_value:w \__int_eval:w
11763     \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
11764         \if_meaning:w 5 #3 1 \fi:
11765         \exp_stop_f:
11766         \if_int_compare:w \__int_eval:w #4 > \c_zero
11767             1 +
11768         \fi:
11769     \fi:
11770     #3
11771 ;
11772 }

```

(End definition for __fp_round_s:NNNw.)

__fp_round_digit:Nw This function should always be called within an __int_value:w or __int_eval:w expansion; it may add an extra __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.

```

11773 \cs_new:Npn \__fp_round_digit:Nw #1 #2;
11774 {
11775     \if_int_odd:w \if_meaning:w 0 #1 \c_one \else:
11776         \if_meaning:w 5 #1 \c_one \else:
11777         \c_zero \fi: \fi:
11778     \if_int_compare:w \__int_eval:w #2 > \c_zero
11779         \__int_eval:w \c_one +
11780     \fi:
11781     \fi:
11782     #1
11783 }

```

(End definition for __fp_round_digit:Nw.)

```

    \__fp_round_neg:NNN This expands to \c_zero or \c_one after doing the following test. Starting from a
    \_fp_round_to_nearest_neg:NNN number of the form  $\langle final\ sign \rangle 0.\langle 15\ digits \rangle \langle digit_1 \rangle$  with exactly 15 (non-all-zero) digits
    \_fp_round_to_nearest_ninf_neg:NNN before  $\langle digit_1 \rangle$ , subtract from it  $\langle final\ sign \rangle 0.0 \dots 0 \langle digit_2 \rangle$ , where there are 16 zeros. If
    \_fp_round_to_nearest_zero_neg:NNN in the current rounding mode the result should be rounded down, then this function
    \_fp_round_to_nearest_pinf_neg:NNN returns \c_one. Otherwise, i.e., if the result is rounded back to the first operand, then
    \__fp_round_to_ninf_neg:NNN this function returns \c_zero.
    \__fp_round_to_zero_neg:NNN It turns out that this negative “round to nearest” is identical to the positive one.
    \__fp_round_to_pinf_neg:NNN And this is the default mode.

11784 \cs_new_eq:NN \__fp_round_to_ninf_neg:NNN \__fp_round_to_pinf:NNN
11785 \cs_new:Npn \__fp_round_to_zero_neg:NNN #1 #2 #3
11786 {
11787     \if_int_compare:w #3 > \c_zero
11788         \__fp_round_return_one:
11789     \fi:
11790     \c_zero
11791 }
11792 \cs_new_eq:NN \__fp_round_to_pinf_neg:NNN \__fp_round_to_ninf:NNN
11793 \cs_new_eq:NN \__fp_round_to_nearest_neg:NNN \__fp_round_to_nearest:NNN
11794 \cs_new_eq:NN \__fp_round_to_nearest_ninf_neg:NNN \__fp_round_to_nearest_pinf:NNN
11795 \cs_new:Npn \__fp_round_to_nearest_zero_neg:NNN #1 #2 #3
11796 {
11797     \if_int_compare:w #3 > \c_four
11798         \__fp_round_return_one:
11799     \fi:
11800     \c_zero
11801 }
11802 \cs_new_eq:NN \__fp_round_to_nearest_pinf_neg:NNN \__fp_round_to_nearest_ninf:NNN
11803 \cs_new_eq:NN \__fp_round_neg:NNN \__fp_round_to_nearest_neg:NNN

```

(End definition for __fp_round_neg:NNN and others.)

24.2 The round function

__fp_round_o:Nw 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 will change #1 from __fp_round_to_nearest:NNN to one of its analogues.

```

11804 \cs_new:Npn \__fp_round_o:Nw #1#2 @
11805 {
11806     \if_case:w
11807         \__int_eval:w \__fp_array_count:n {#2} \__int_eval_end:
11808         \__fp_round_no_arg_o:Nw #1 \exp:w
11809     \or: \__fp_round:Nwn #1 #2 {0} \exp:w
11810     \or: \__fp_round:Nww #1 #2 \exp:w
11811     \else: \__fp_round:Nwww #1 #2 @ \exp:w
11812     \fi:
11813     \exp_end_continue_f:w
11814 }

```

(End definition for __fp_round_o:Nw.)

__fp_round_no_arg_o:Nw

```

11815 \cs_new:Npn \__fp_round_no_arg_o:Nw #1
11816 {

```



```

11817 \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
11818 { \__fp_error:nnnn { num-args } { round () } { 1 } { 3 } }
11819 {
11820   \__fp_error:nffn { num-args }
11821   { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
11822 }
11823 \exp_after:wN \c_nan_fp
11824 }

```

(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.

```

11825 \cs_new:Npn \__fp_round:Nwww #1#2 ; #3 ; \s__fp \__fp_chk:w #4#5#6 ; #7 @
11826 {
11827   \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
11828   {
11829     \tl_if_empty:nTF {#7}
11830     {
11831       \exp_args:Nc \__fp_round:Nww
11832       {
11833         __fp_round_to_nearest
11834         \if_meaning:w 0 #4 _zero \else:
11835         \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
11836         :NNN
11837       }
11838       #2 ; #3 ;
11839     }
11840     {
11841       \__fp_error:nnnn { num-args } { round () } { 1 } { 3 }
11842       \exp_after:wN \c_nan_fp
11843     }
11844   }
11845   {
11846     \__fp_error:nffn { num-args }
11847     { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
11848     \exp_after:wN \c_nan_fp
11849   }
11850 }

```

(End definition for __fp_round:Nwww.)

__fp_round_name_from_cs:N

```

11851 \cs_new:Npn \__fp_round_name_from_cs:N #1
11852 {
11853   \cs_if_eq:NNTF #1 \__fp_round_to_zero:NNN { trunc }
11854   {
11855     \cs_if_eq:NNTF #1 \__fp_round_to_ninf:NNN { floor }
11856     {
11857       \cs_if_eq:NNTF #1 \__fp_round_to_pinf:NNN { ceil }
11858       { round }
11859     }

```

```

11860     }
11861 }

```

(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

11862 \cs_new:Npn \_fp_round:Nww #1#2 ; #3 ;
11863 {
11864   \_fp_small_int:wTF #3; { \_fp_round:Nwn #1#2; }
11865   {
11866     \_fp_invalid_operation_tl_o:ff
11867     { \_fp_round_name_from_cs:N #1 }
11868     { \_fp_array_to_clist:n { #2; #3; } }
11869   }
11870 }
11871 \cs_new:Npn \_fp_round:Nwn #1 \s_fp \_fp_chk:w #2#3#4; #5
11872 {
11873   \if_meaning:w 1 #2
11874     \exp_after:wN \_fp_round_normal:NwNNnw
11875     \exp_after:wN #1
11876     \__int_value:w #5
11877   \else:
11878     \exp_after:wN \_fp_exp_after_o:w
11879   \fi:
11880   \s_fp \_fp_chk:w #2#3#4;
11881 }
11882 \cs_new:Npn \_fp_round_normal:NwNNnw #1#2 \s_fp \_fp_chk:w 1#3#4#5;
11883 {
11884   \_fp_decimate:nNnnnn { \c_sixteen - #4 - #2 }
11885   \_fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
11886 }
11887 \cs_new:Npn \_fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
11888 {
11889   \exp_after:wN \_fp_round_normal:NNwNnn
11890   \__int_value:w \__int_eval:w
11891   \if_int_compare:w #2 > \c_zero
11892     1 \__int_value:w #2
11893     \exp_after:wN \_fp_round_pack:Nw
11894     \__int_value:w \__int_eval:w 1#3 +
11895   \else:
11896     \if_int_compare:w #3 > \c_zero
11897       1 \__int_value:w #3 +
11898     \fi:
11899   \fi:
11900   \exp_after:wN #5
11901   \exp_after:wN #6
11902   \use_none:nnnnnn #3
11903   #1
11904   \__int_eval_end:
11905   0000 0000 0000 0000 ; #6
11906 }
11907 \cs_new:Npn \_fp_round_pack:Nw #1
11908 { \if_meaning:w 2 #1 + \c_one \fi: \__int_eval_end: }
11909 \cs_new:Npn \_fp_round_normal:NNwNnn #1 #2

```

```

11910 {
11911   \if_meaning:w 0 #2
11912   \exp_after:wN \__fp_round_special:NwwNnn
11913   \exp_after:wN #1
11914   \fi:
11915   \__fp_pack_twice_four:wNNNNNNNN
11916   \__fp_pack_twice_four:wNNNNNNNN
11917   \__fp_round_normal_end:wwNnn
11918   ; #2
11919 }
11920 \cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
11921 {
11922   \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
11923   \__fp_sanitiz:Nw #3 #4 ; #1 ;
11924 }
11925 \cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
11926 {
11927   \if_meaning:w 0 #1
11928   \__fp_case_return:nw
11929   { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
11930   \else:
11931   \exp_after:wN \__fp_round_special_aux:Nw
11932   \exp_after:wN #4
11933   \__int_value:w \__int_eval:w \c_one
11934   \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
11935   \fi:
11936   ;
11937 }
11938 \cs_new:Npn \__fp_round_special_aux:Nw #1#2;
11939 {
11940   \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
11941   \__fp_sanitiz:Nw #1#2; {1000}{0000}{0000}{0000};
11942 }

```

(End definition for __fp_round:Nww and others.)

```

11943 </initex | package>

```

25 l3fp-parse implementation

```

11944 <*initex | package>

```

```

11945 <@@=fp>

```

25.1 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.

`__fp_parse:n` Evaluates the *<floating point expression>* and leaves the result in the input stream as an internal floating point number. This function forms the basis of almost all public l3fp functions. During evaluation, each token is fully f-expanded.

`__fp_parse_o:n` does the same but expands once after its result.

T_EXhackers note: Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as `\int_use:N`. Invalid tokens remaining after `f`-expansion will lead to unrecoverable low-level T_EX errors.

(End definition for __fp_parse:n.)

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 with multiple arguments.
- 15 Function calls expecting exactly one argument.
- 13/14 Binary `**` and `^` (right to left).
- 12 Unary `+`, `-`, `!` (right to left).
- 10 Binary `*`, `/`, and juxtaposition (implicit `*`).
- 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, and parentheses accepting commas.
- 1 Parentheses expecting exactly one argument.
- 0 Start and end of the expression.

```
\c__fp_prec_funcii_int
\c__fp_prec_func_int      11946 \int_const:Nn \c__fp_prec_funcii_int { 16 }
\c__fp_prec_hatii_int     11947 \int_const:Nn \c__fp_prec_func_int { 15 }
\c__fp_prec_hat_int       11948 \int_const:Nn \c__fp_prec_hatii_int { 14 }
\c__fp_prec_not_int       11949 \int_const:Nn \c__fp_prec_hat_int { 13 }
\c__fp_prec_times_int     11950 \int_const:Nn \c__fp_prec_not_int { 12 }
\c__fp_prec_plus_int      11951 \int_const:Nn \c__fp_prec_times_int { 10 }
\c__fp_prec_comp_int      11952 \int_const:Nn \c__fp_prec_plus_int { 9 }
\c__fp_prec_and_int       11953 \int_const:Nn \c__fp_prec_comp_int { 7 }
\c__fp_prec_or_int        11954 \int_const:Nn \c__fp_prec_and_int { 6 }
\c__fp_prec_quest_int     11955 \int_const:Nn \c__fp_prec_or_int { 5 }
\c__fp_prec_colon_int     11956 \int_const:Nn \c__fp_prec_quest_int { 4 }
\c__fp_prec_comma_int     11957 \int_const:Nn \c__fp_prec_colon_int { 3 }
\c__fp_prec_paren_int     11958 \int_const:Nn \c__fp_prec_comma_int { 2 }
\c__fp_prec_end_int       11959 \int_const:Nn \c__fp_prec_paren_int { 1 }
\c__fp_prec_end_int       11960 \int_const:Nn \c__fp_prec_end_int { 0 }
```

(End definition for \c__fp_prec_funcii_int and others.)

25.1.1 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 will grow 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 much 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 `\c_zero`, 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 \c_zero 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 `\c_zero`, 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_..._o:ww` expand what follows once. This comes at the cost of leaving tokens in the input stack, and we will 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.

25.1.2 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 the calculation $41 - 2^3 * 4 + 5$ will be done. 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 \wedge .
- Compare the precedences of $-$ and \wedge . Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw \wedge`.
- Clean up 3 and find $*$.
- Compare the precedences of \wedge and $*$. Since the former is higher, `\operand:Nw \wedge` 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 will be 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 $\langle precedence \rangle$ 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 $\langle number \rangle$, performing no computation, and finds the following binary $\langle operator \rangle$. Then it expands to

```
 $\langle number \rangle$ 
\__fp_parse_infix_ $\langle operator \rangle$ :N  $\langle precedence \rangle$ 
```

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_ $\langle operator \rangle$: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 $\langle precedence \rangle$ (of the earlier operator) to the `infix` auxiliary for the following $\langle operator \rangle$, to know whether to perform the computation of the $\langle operator \rangle$. If it should not be performed, the `infix` auxiliary expands to

```
@ \use_none:n \__fp_parse_infix_ $\langle operator \rangle$ :N
```

and otherwise it calls `__fp_parse_operand:Nw` with the precedence of the $\langle operator \rangle$ to find its second operand $\langle number_2 \rangle$ and the next $\langle operator_2 \rangle$, and expands to

```
@ \__fp_parse_apply_binary:NwNwN
 $\langle operator \rangle$   $\langle number_2 \rangle$ 
@ \__fp_parse_infix_ $\langle operator_2 \rangle$ :N
```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand $\langle number \rangle$ 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 $\langle precedence \rangle$` with some of the expansion control removed is

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN  $\langle precedence \rangle$ 
\exp:w \exp_end_continue_f:w
\__fp_parse_one:Nw  $\langle precedence \rangle$ 
```

This expands `_fp_parse_one:Nw` $\langle precedence \rangle$ completely, which finds a number, wraps the next $\langle operator \rangle$ into an infix function, feeds this function the $\langle precedence \rangle$, and expands it, yielding either

```
\_fp_parse_continue:NwN  $\langle precedence \rangle$ 
 $\langle number \rangle$  @
\use_none:n \_fp_parse_infix_ $\langle operator \rangle$ :N
```

or

```
\_fp_parse_continue:NwN  $\langle precedence \rangle$ 
 $\langle number \rangle$  @
\_fp_parse_apply_binary:NwNwN
 $\langle operator \rangle$   $\langle number_2 \rangle$ 
@ \_fp_parse_infix_ $\langle operator_2 \rangle$ :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  $\langle precedence \rangle$   $\langle number \rangle$  @
\_fp_parse_infix_ $\langle operator \rangle$ :N
```

then $\langle number \rangle$ @ `_fp_parse_infix_ $\langle operator \rangle$:N`. In the second case, #3 is `_fp_parse_apply_binary:NwNwN`, whose role is to compute $\langle number \rangle$ $\langle operator \rangle$ $\langle number_2 \rangle$ and to prepare for the next comparison of precedences: first we get

```
\_fp_parse_apply_binary:NwNwN
 $\langle precedence \rangle$   $\langle number \rangle$  @
 $\langle operator \rangle$   $\langle number_2 \rangle$ 
@ \_fp_parse_infix_ $\langle operator_2 \rangle$ :N
```

then

```
\exp_after:wN \_fp_parse_continue:NwN
\exp_after:wN  $\langle precedence \rangle$ 
\exp:w \exp_end_continue_f:w
\_fp_ $\langle operator \rangle$ _o:ww  $\langle number \rangle$   $\langle number_2 \rangle$ 
\exp:w \exp_end_continue_f:w
\_fp_parse_infix_ $\langle operator_2 \rangle$ :N  $\langle precedence \rangle$ 
```

where `_fp_ $\langle operator \rangle$ _o:ww` computes $\langle number \rangle$ $\langle operator \rangle$ $\langle number_2 \rangle$ and expands after the result, thus triggers the comparison of the precedence of the $\langle operator_2 \rangle$ and the $\langle precedence \rangle$, continuing the loop.

We have introduced the most important functions here, and the next few paragraphs will describe various subtleties.

25.1.3 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.

25.1.4 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 *significand***e***exponent*, where the *significand* is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent “**e***exponent*” 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 *significand* 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 *exponent* 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_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 *significand* 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_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_three`, 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_three 2` is disallowed.

In the above, we need to test whether a character token `#1` is a digit:

```
\if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
  is a digit
\else:
  not a digit
\fi:
```

To exclude 0, replace `\c_nine` by `\c_ten`. 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 \__int_eval:w
  ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26 = \c_three
  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 `{1, 2, 4, 10, 13}` will not work, and of course `{0, 5, 9}` cannot become tokens.

Floating point expressions should behave as much as possible like ε -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 will not be expanded if we simply perform `f`-expansion.

```

\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }

```

Of course, spaces will 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 will stop the `f`-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers will correctly be expanded to the underlying `\s__fp ...` structure. The `f`-expansion is performed by `__fp_parse_expand:w`.

25.2 Main auxiliary functions

`__fp_parse_operand:Nw` Reads the "...", performing every computation with a precedence higher than $\langle precedence \rangle$, then expands to 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` 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 Otherwise expands to A similar function exists for each infix operator.

(End definition for `__fp_parse_infix_+:N`.)

`__fp_parse_one:Nw` 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 and otherwise expands to

(End definition for `__fp_parse_one:Nw`.)

25.3 Helpers

`__fp_parse_expand:w` 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.

```
11961 \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.

```
11962 \cs_new:Npn \_fp_parse_return_semicolon:w
11963     #1 \fi: \__fp_parse_expand:w { \fi: ; #1 }
```

(End definition for `_fp_parse_return_semicolon:w`.)

`__fp_type_from_scan:N` Grabs the pieces of the stringified $\langle token \rangle$ which lies after the first `s__fp`. If the $\langle token \rangle$ does not contain that string, the result is `_?`.

`__fp_type_from_scan:w`

```
11964 \cs_new:Npx \__fp_type_from_scan:N #1
11965 {
11966     \exp_not:N \exp_after:wN \exp_not:N \__fp_type_from_scan:w
11967     \exp_not:N \token_to_str:N #1 \exp_not:N \q_mark
11968     \tl_to_str:n { s__fp _? } \exp_not:N \q_mark \exp_not:N \q_stop
11969 }
11970 \use:x
11971 {
11972     \cs_new:Npn \exp_not:N \__fp_type_from_scan:w
11973         ##1 \tl_to_str:n { s__fp } ##2 \exp_not:N \q_mark ##3 \exp_not:N \q_stop
11974         {##2}
11975 }
```

(End definition for `__fp_type_from_scan:N` and `__fp_type_from_scan:w`.)

`_fp_parse_digits_vii:N` These functions must be called within an `__int_value:w` or `__int_eval:w` construction. The first token which follows must be f-expanded prior to calling those functions. `_fp_parse_digits_vi:N` 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_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.

```
11976 \cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
11977 {
11978     \cs_new:cpn { \_fp_parse_digits_ #1 :N } ##1
11979     {
11980         \if_int_compare:w \c_nine < 1 \token_to_str:N ##1 \exp_stop_f:
11981         \token_to_str:N ##1 \exp_after:wN #2 \exp:w
11982     } \else:
11983         \_fp_parse_return_semicolon:w #3 ##1
```

```

11984         \fi:
11985         \__fp_parse_expand:w
11986     }
11987 }
11988 \__fp_tmp:w {vii} \__fp_parse_digits_vi:N { 0000000 ; 7 }
11989 \__fp_tmp:w {vi} \__fp_parse_digits_v:N { 000000 ; 6 }
11990 \__fp_tmp:w {v} \__fp_parse_digits_iv:N { 00000 ; 5 }
11991 \__fp_tmp:w {iv} \__fp_parse_digits_iii:N { 0000 ; 4 }
11992 \__fp_tmp:w {iii} \__fp_parse_digits_ii:N { 000 ; 3 }
11993 \__fp_tmp:w {ii} \__fp_parse_digits_i:N { 00 ; 2 }
11994 \__fp_tmp:w {i} \__fp_parse_digits_:N { 0 ; 1 }
11995 \cs_new:Npn \__fp_parse_digits_:N { ; ; 0 }

```

(End definition for `__fp_parse_digits_vii:N` and others.)

25.4 Parsing one number

`__fp_parse_one:Nw` This function finds one number, and packs the symbol which follows in an `\..._infix...` csname. #1 is the previous *<precedence>*, 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 will be split further). Despite the earlier f-expansion, #2 may still be expandable if it was protected by `\exp_not:N`, as may happen with the L^AT_EX 2_ε command `\protect`. Using a well placed `\reverse_if:N`, this case is sent to `__fp_parse_one_fp:NN` which deals with it robustly.

```

11996 \cs_new:Npn \__fp_parse_one:Nw #1 #2
11997 {
11998     \if_catcode:w \scan_stop: \exp_not:N #2
11999     \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
12000     \exp_after:wN \reverse_if:N
12001     \fi:
12002     \if_meaning:w \scan_stop: #2
12003     \exp_after:wN \exp_after:wN
12004     \exp_after:wN \__fp_parse_one_fp:NN
12005     \else:
12006     \exp_after:wN \exp_after:wN
12007     \exp_after:wN \__fp_parse_one_register:NN
12008     \fi:
12009     \else:
12010     \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
12011     \exp_after:wN \exp_after:wN
12012     \exp_after:wN \__fp_parse_one_digit:NN
12013     \else:
12014     \exp_after:wN \exp_after:wN
12015     \exp_after:wN \__fp_parse_one_other:NN
12016     \fi:
12017     \fi:
12018     #1 #2
12019 }

```

(End definition for `__fp_parse_one:Nw`.)

`__fp_parse_one_fp:NN` This function receives a *<precedence>* and a control sequence equal to `\scan_stop:` in meaning. There are three cases, dispatched using `__fp_type_from_scan:N`.
`__fp_exp_after_mark_f:nw`
`__fp_exp_after_?_f:nw`

- `\s__fp` starts a floating point number, and we call `__fp_exp_after_f:nw`, which f-expands after the floating point.
- `\s__fp_mark` is a premature end, we call `__fp_exp_after_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^AT_EX 2_ε 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^AT_EX 2_ε uses `\@unexpandable@protect` rather than `\protect` because `\protect` is often `\scan_stop`: hence “does not exist”.

```

12020 \cs_new:Npn \__fp_parse_one_fp:NN #1#2
12021 {
12022   \cs:w __fp_exp_after \__fp_type_from_scan:N #2 _f:nw \cs_end:
12023   {
12024     \exp_after:wN \__fp_parse_infix:NN
12025     \exp_after:wN #1 \exp:w \__fp_parse_expand:w
12026   }
12027   #2
12028 }
12029 \cs_new:Npn \__fp_exp_after_mark_f:nw #1
12030 {
12031   \__msg_kernel_expandable_error:nn { kernel } { fp-early-end }
12032   \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
12033 }
12034 \cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
12035 {
12036   \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
12037   \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
12038 }
12039 \*package)
12040 \group_begin:
12041   \char_set_catcode_letter:N \@
12042   \cs_if_exist:NT \@unexpandable@protect
12043   {
12044     \cs_gset:cpn { __fp_exp_after_?_f:nw } #1#2
12045     {
12046       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
12047       \str_if_eq:nnTF {#2} { \protect }
12048       {
12049         \cs_if_eq:NNTF #2 \@unexpandable@protect { \use_i:nn } { \use:n }
12050         { \__msg_kernel_expandable_error:nnn { kernel } { fp-robust-cmd } }
12051       }
12052       { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2} }
12053     }
12054   }
12055 \group_end:
12056 \*package)

```

(End definition for `__fp_parse_one_fp:NN`, `__fp_exp_after_mark_f:nw`, and `__fp_exp_after_?_f:nw`.)

`__fp_parse_one_register:NN` 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_eval:w` $\langle decimal value \rangle$ pt, and use an auxiliary of `\dim_to_fp:n`, which performs the multiplication by 2^{-16} , correctly rounded.

```

12057 \cs_new:Npn \__fp_parse_one_register:NN #1#2
12058 {
12059   \exp_after:wN \__fp_parse_infix_after_operand:NwN
12060   \exp_after:wN #1
12061   \exp:w \exp_end_continue_f:w
12062   \if_meaning:w \box_wd:N #2 \__fp_parse_one_register_wd:w \fi:
12063   \if_meaning:w \box_ht:N #2 \__fp_parse_one_register_wd:w \fi:
12064   \if_meaning:w \box_dp:N #2 \__fp_parse_one_register_wd:w \fi:
12065   \exp_after:wN \__fp_parse_one_register_aux:Nw
12066   \exp_after:wN #2
12067   \__int_value:w
12068   \exp_after:wN \__fp_parse_exponent:N
12069   \exp:w \__fp_parse_expand:w
12070 }
12071 \cs_new:Npx \__fp_parse_one_register_aux:Nw #1
12072 {
12073   \exp_not:n
12074   {
12075     \exp_after:wN \use:nn
12076     \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
12077   }
12078   \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
12079   ; \exp_not:N \__fp_parse_one_register_dim:ww
12080   \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_mu:www
12081   . \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_int:www
12082   \exp_not:N \q_stop
12083 }
12084 \use:x
12085 {
12086   \cs_new:Npn \exp_not:N \__fp_parse_one_register_auxii:wwwNw
12087     ##1 . ##2 \tl_to_str:n { pt } ##3 ; ##4##5 \exp_not:N \q_stop
12088     { ##4 ##1.##2; }
12089   \cs_new:Npn \exp_not:N \__fp_parse_one_register_mu:www
12090     ##1 \tl_to_str:n { mu } ; ##2 ;
12091     { \exp_not:N \__fp_parse_one_register_dim:ww ##1 ; }
12092 }
12093 \cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
12094 { \__fp_parse:n { #1 e #3 } }
12095 \cs_new:Npn \__fp_parse_one_register_dim:ww #1; #2;

```

```

12096 {
12097   \exp_after:wN \__fp_from_dim_test:ww
12098   \__int_value:w #2 \exp_after:wN ,
12099   \__int_value:w \__dim_eval:w #1 pt ;
12100 }

```

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.

```

12101 \cs_new:Npn \__fp_parse_one_register_wd:w
12102   #1#2 \exp_after:wN #3#4 \__fp_parse_expand:w
12103   {
12104     #1
12105     \exp_after:wN \__fp_parse_one_register_wd:Nw
12106     #4 \__fp_parse_expand:w e
12107   }
12108 \cs_new:Npn \__fp_parse_one_register_wd:Nw #1#2 ;
12109   {
12110     \exp_after:wN \__fp_from_dim_test:ww
12111     \exp_after:wN 0 \exp_after:wN ,
12112     \__int_value:w \__dim_eval:w
12113     \exp_after:wN \use:n \exp_after:wN { \tex_the:D #1 #2 } ;
12114   }

```

(End definition for `__fp_parse_one_register:NN` and others.)

`__fp_parse_one_digit:NN`

A digit marks the beginning of an explicit floating point number. Once the number is found, we will catch the case of overflow and underflow with `__fp_sanitizewN`, 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.

```

12115 \cs_new:Npn \__fp_parse_one_digit:NN #1
12116   {
12117     \exp_after:wN \__fp_parse_infix_after_operand:NwN
12118     \exp_after:wN #1
12119     \exp:w \exp_end_continue_f:w
12120     \exp_after:wN \__fp_sanitizewN
12121     \__int_value:w \__int_eval:w \c_zero \__fp_parse_trim_zeros:N
12122   }

```

(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 a 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`.

```

12123 \cs_new:Npn \__fp_parse_one_other:NN #1 #2
12124   {
12125     \if_int_compare:w
12126       \__int_eval:w
12127       ( '#2 \if_int_compare:w '#2 > 'Z - \c_thirty_two \fi: ) / 26
12128       = \c_three
12129     \exp_after:wN \__fp_parse_word:Nw

```



```

12130     \exp_after:wN #1
12131     \exp_after:wN #2
12132     \exp:w \exp_after:wN \_fp_parse_letters:N
12133     \exp:w
12134   \else:
12135     \exp_after:wN \_fp_parse_prefix:NNN
12136     \exp_after:wN #1
12137     \exp_after:wN #2
12138     \cs:w
12139       \_fp_parse_prefix_ \token_to_str:N #2 :Nw
12140     \exp_after:wN
12141     \cs_end:
12142     \exp:w
12143   \fi:
12144   \_fp_parse_expand:w
12145 }

```

(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.

```

12146 \cs_new:Npn \_fp_parse_word:Nw #1#2;
12147 {
12148   \cs_if_exist_use:cF { \_fp_parse_word_#2:N }
12149   {
12150     \_msg_kernel_expandable_error:nnn
12151     { kernel } { unknown-fp-word } {#2}
12152     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
12153     \_fp_parse_infix:NN
12154   }
12155   #1
12156 }
12157 \cs_new:Npn \_fp_parse_letters:N #1
12158 {
12159   \exp_end_continue_f:w
12160   \if_int_compare:w
12161     \if_catcode:w \scan_stop: \exp_not:N #1
12162     \c_zero
12163   \else:
12164     \_int_eval:w
12165     ( '#1 \if_int_compare:w '#1 > 'Z - \c_thirty_two \fi: )
12166     / 26
12167   \fi:
12168   = \c_three
12169   \exp_after:wN #1
12170   \exp:w \exp_after:wN \_fp_parse_letters:N
12171   \exp:w
12172 \else:
12173   \_fp_parse_return_semicolon:w #1
12174 \fi:

```

```

12175     \__fp_parse_expand:w
12176 }

```

(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.

```

12177 \cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
12178 {
12179   \if_meaning:w \scan_stop: #3
12180   \exp_after:wN \__fp_parse_prefix_unknown:NNN
12181   \exp_after:wN #2
12182   \fi:
12183   #3 #1
12184 }
12185 \cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
12186 {
12187   \cs_if_exist:cTF { __fp_parse_infix_ \token_to_str:N #1 :N }
12188   {
12189     \_msg_kernel_expandable_error:nnn
12190     { kernel } { fp-missing-number } {#1}
12191     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
12192     \__fp_parse_infix:NN #3 #1
12193   }
12194   {
12195     \_msg_kernel_expandable_error:nnn
12196     { kernel } { fp-unknown-symbol } {#1}
12197     \__fp_parse_one:Nw #3
12198   }
12199 }

```

(End definition for __fp_parse_prefix:NNN and __fp_parse_prefix_unknown:NNN.)

25.4.1 Numbers: trimming leading zeros

Numbers will be parsed as follows: first we trim leading zeros, then if the next character is a digit, start reading a significand ≥ 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 ≥ 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.

```

12200 \cs_new:Npn \__fp_parse_trim_zeros:N #1

```

```

12201 {
12202     \if:w 0 \exp_not:N #1
12203         \exp_after:wN \__fp_parse_trim_zeros:N
12204         \exp:w
12205     \else:
12206         \if:w . \exp_not:N #1
12207             \exp_after:wN \__fp_parse_strim_zeros:N
12208             \exp:w
12209         \else:
12210             \__fp_parse_trim_end:w #1
12211         \fi:
12212     \fi:
12213     \__fp_parse_expand:w
12214 }
12215 \cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
12216 {
12217     \fi:
12218     \fi:
12219     \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12220         \exp_after:wN \__fp_parse_large:N
12221     \else:
12222         \exp_after:wN \__fp_parse_zero:
12223     \fi:
12224     #1
12225 }

```

(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”.

```

12226 \cs_new:Npn \__fp_parse_strim_zeros:N #1
12227 {
12228     \if:w 0 \exp_not:N #1
12229         - \c_one
12230         \exp_after:wN \__fp_parse_strim_zeros:N \exp:w
12231     \else:
12232         \__fp_parse_strim_end:w #1
12233     \fi:
12234     \__fp_parse_expand:w
12235 }
12236 \cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
12237 {
12238     \fi:
12239     \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12240         \exp_after:wN \__fp_parse_small:N
12241     \else:
12242         \exp_after:wN \__fp_parse_zero:
12243     \fi:
12244     #1
12245 }

```

(End definition for __fp_parse_strim_zeros:N and __fp_parse_strim_end:w.)

`__fp_parse_zero:` After reading a significand of 0, we need to remove any exponent, then put a sign of 1 for `__fp_sanitize:wN`, small hack to denote an exact zero (rather than an underflow).

```

12246 \cs_new:Npn \__fp_parse_zero:
12247 {
12248   \exp_after:wN ; \exp_after:wN 1
12249   \__int_value:w \__fp_parse_exponent:N
12250 }

```

(End definition for `__fp_parse_zero:.`)

25.4.2 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 will leave those digits in the `__int_value:w`, and grab some more, or stop if there are no more digits. Then the `pack_leading` auxiliary puts the various parts in the appropriate order for the processing further up.

```

12251 \cs_new:Npn \__fp_parse_small:N #1
12252 {
12253   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
12254   \__int_value:w \__int_eval:w 1 \token_to_str:N #1
12255   \exp_after:wN \__fp_parse_small_leading:wwNN
12256   \__int_value:w 1
12257   \exp_after:wN \__fp_parse_digits_vii:N
12258   \exp:w \__fp_parse_expand:w
12259 }

```

(End definition for `__fp_parse_small:N`.)

`__fp_parse_small_leading:wwNN` 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 `\c_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.

```

12260 \cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
12261 {
12262   #1 #2
12263   \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
12264   \exp_after:wN \c_zero
12265   \__int_value:w \__int_eval:w 1
12266   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
12267     \token_to_str:N #4
12268     \exp_after:wN \__fp_parse_small_trailing:wwNN
12269     \__int_value:w 1
12270     \exp_after:wN \__fp_parse_digits_vi:N
12271     \exp:w
12272   \else:
12273     0000 0000 \__fp_parse_exponent:Nw #4

```

```

12274     \fi:
12275     \__fp_parse_expand:w
12276 }

```

(End definition for __fp_parse_small_leading:wwNN.)

__fp_parse_small_trailing:wwNN 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 `+\c_zero` or to `+\c_one`. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```

12277 \cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
12278 {
12279     #1 #2
12280     \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
12281     \token_to_str:N #4
12282     \exp_after:wN \__fp_parse_small_round:NN
12283     \exp_after:wN #4
12284     \exp:w
12285     \else:
12286     0 \__fp_parse_exponent:Nw #4
12287     \fi:
12288     \__fp_parse_expand:w
12289 }

```

(End definition for __fp_parse_small_trailing:wwNN.)

__fp_parse_pack_trailing:NNNNNNww 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 (`+\c_one` 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.

```

12290 \cs_new:Npn \__fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
12291 {
12292     \if_meaning:w 2 #2 + \c_one \fi:
12293     ; #8 + #1 ; {#3#4#5#6} {#7};
12294 }
12295 \cs_new:Npn \__fp_parse_pack_leading:NNNNNNww #1 #2#3#4#5 #6; #7;
12296 {
12297     + #7
12298     \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
12299     ; 0 {#2#3#4#5} {#6}
12300 }
12301 \cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
12302 { \fi: + \c_one ; 0 {1000} }

```

(End definition for __fp_parse_pack_trailing:NNNNNNww, __fp_parse_pack_leading:NNNNNNww, and __fp_parse_pack_carry:w.)

25.4.3 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 (≥ 1). It is called within an integer expression for the exponent. Grab up to 7 more digits, for a total of 8 digits.

```

12303 \cs_new:Npn \__fp_parse_large:N #1
12304 {
12305   \exp_after:wN \__fp_parse_large_leading:wwNN
12306   \__int_value:w 1 \token_to_str:N #1
12307   \exp_after:wN \__fp_parse_digits_vii:N
12308   \exp:w \__fp_parse_expand:w
12309 }
```

(End definition for `__fp_parse_large:N`.)

`_fp_parse_large_leading:wwNN` We shift the exponent by the number of digits in `#1`, namely the target number, 8, minus the *number of zeros* (number of digits missing). Then prepare to pack the 8 first digits. If the *next token* 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 *zeros* to complete the 8 first digits, insert 8 more, and look for an exponent.

```

12310 \cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
12311 {
12312   + \c_eight - #3
12313   \exp_after:wN \__fp_parse_pack_leading:NNNNww
12314   \__int_value:w \__int_eval:w 1 #1
12315   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
12316     \exp_after:wN \__fp_parse_large_trailing:wwNN
12317     \__int_value:w 1 \token_to_str:N #4
12318     \exp_after:wN \__fp_parse_digits_vi:N
12319     \exp:w
12320   \else:
12321     \if:w . \exp_not:N #4
12322       \exp_after:wN \__fp_parse_small_leading:wwNN
12323       \__int_value:w 1
12324       \cs:w
12325         \__fp_parse_digits_
12326         \__int_to_roman:w #3
12327         :N \exp_after:wN
12328       \cs_end:
12329       \exp:w
12330     \else:
12331       #2
12332       \exp_after:wN \__fp_parse_pack_trailing:NNNNww
12333       \exp_after:wN \c_zero
12334       \__int_value:w 1 0000 0000
12335       \__fp_parse_exponent:Nw #4
12336     \fi:
12337   \fi:
```

```

12338     \__fp_parse_expand:w
12339 }

```

(End definition for __fp_parse_large_leading:wwNN.)

__fp_parse_large_trailing:wwNN

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.

```

12340 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
12341 {
12342   \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
12343     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
12344     \exp_after:wN \c_eight
12345     \__int_value:w \__int_eval:w 1 #1 \token_to_str:N #4
12346     \exp_after:wN \__fp_parse_large_round:NN
12347     \exp_after:wN #4
12348     \exp:w
12349   \else:
12350     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
12351     \__int_value:w \__int_eval:w \c_seven - #3 \exp_stop_f:
12352     \__int_value:w \__int_eval:w 1 #1
12353     \if:w . \exp_not:N #4
12354       \exp_after:wN \__fp_parse_small_trailing:wwNN
12355       \__int_value:w 1
12356       \cs:w
12357         \__fp_parse_digits_
12358         \__int_to_roman:w #3
12359         :N \exp_after:wN
12360       \cs_end:
12361       \exp:w
12362     \else:
12363       #2 0 \__fp_parse_exponent:Nw #4
12364     \fi:
12365   \fi:
12366   \__fp_parse_expand:w
12367 }

```

(End definition for __fp_parse_large_trailing:wwNN.)

25.4.4 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 `;\c_zero`, otherwise by `;\c_one`. 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.

```

12368 \cs_new:Npn \__fp_parse_round_loop:N #1
12369 {
12370   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12371     + \c_one
12372     \if:w 0 \token_to_str:N #1
12373       \exp_after:wN \__fp_parse_round_loop:N
12374       \exp:w
12375     \else:
12376       \exp_after:wN \__fp_parse_round_up:N
12377       \exp:w
12378     \fi:
12379   \else:
12380     \__fp_parse_return_semicolon:w \c_zero #1
12381   \fi:
12382   \__fp_parse_expand:w
12383 }
12384 \cs_new:Npn \__fp_parse_round_up:N #1
12385 {
12386   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12387     + \c_one
12388     \exp_after:wN \__fp_parse_round_up:N
12389     \exp:w
12390   \else:
12391     \__fp_parse_return_semicolon:w \c_one #1
12392   \fi:
12393   \__fp_parse_expand:w
12394 }

```

(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 \c_zero or \c_one is added to the surrounding integer expression.

```

12395 \cs_new:Npn \__fp_parse_round_after:wN #1; #2
12396 {
12397   + #2 \exp_after:wN ;
12398   \__int_value:w \__int_eval:w #1 + \__fp_parse_exponent:N
12399 }

```

(End definition for __fp_parse_round_after:wN.)

__fp_parse_small_round:NN 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 ;<exponent> only. Otherwise, we will expand to +\c_zero or +\c_one, then ;<exponent>. 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 +\c_zero or +\c_one 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.

```

12400 \cs_new:Npn \__fp_parse_small_round:NN #1#2
12401 {
12402   \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:

```



```

12403      +
12404      \exp_after:wN \__fp_round_s:NNNw
12405      \exp_after:wN 0
12406      \exp_after:wN #1
12407      \exp_after:wN #2
12408      \__int_value:w \__int_eval:w
12409      \exp_after:wN \__fp_parse_round_after:wN
12410      \__int_value:w \__int_eval:w \c_zero * \__int_eval:w \c_zero
12411      \exp_after:wN \__fp_parse_round_loop:N
12412      \exp:w
12413    \else:
12414      \__fp_parse_exponent:Nw #2
12415    \fi:
12416    \__fp_parse_expand:w
12417  }

```

(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.

```

12418 \cs_new:Npn \__fp_parse_large_round:NN #1#2
12419 {
12420   \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
12421   +
12422   \exp_after:wN \__fp_round_s:NNNw
12423   \exp_after:wN 0
12424   \exp_after:wN #1
12425   \exp_after:wN #2
12426   \__int_value:w \__int_eval:w
12427   \exp_after:wN \__fp_parse_large_round_aux:wNN
12428   \__int_value:w \__int_eval:w \c_one
12429   \exp_after:wN \__fp_parse_round_loop:N
12430   \else: %^^A could be dot, or e, or other
12431   \exp_after:wN \__fp_parse_large_round_test:NN
12432   \exp_after:wN #1
12433   \exp_after:wN #2
12434   \fi:
12435 }
12436 \cs_new:Npn \__fp_parse_large_round_test:NN #1#2
12437 {
12438   \if:w . \exp_not:N #2
12439   \exp_after:wN \__fp_parse_small_round:NN
12440   \exp_after:wN #1
12441   \exp:w
12442   \else:
12443   \__fp_parse_exponent:Nw #2
12444   \fi:

```

```

12445     \__fp_parse_expand:w
12446   }
12447 \cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
12448 {
12449   + #2
12450   \exp_after:wN \__fp_parse_round_after:wN
12451   \__int_value:w \__int_eval:w #1
12452   \if:w . \exp_not:N #3
12453     + \c_zero * \__int_eval:w \c_zero
12454     \exp_after:wN \__fp_parse_round_loop:N
12455     \exp:w \exp_after:wN \__fp_parse_expand:w
12456   \else:
12457     \exp_after:wN ;
12458     \exp_after:wN \c_zero
12459     \exp_after:wN #3
12460   \fi:
12461 }

```

(End definition for `__fp_parse_large_round:NN`, `__fp_parse_large_round_test:NN`, and `__fp_parse_large_round_aux:wNN`.)

25.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication is implicit.

```

\@@_parse:n { 3.2 erf(0.1) }
\@@_parse:n { 3.2 e\l_my_int }
\@@_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 `__int_eval:w ...` there if needed.

```

12462 \cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w
12463 {
12464   \exp_after:wN ;
12465   \__int_value:w #2 \__fp_parse_exponent:N #1
12466 }

```

(End definition for _fp_parse_exponent:Nw.)

_fp_parse_exponent:N
_fp_parse_exponent_aux: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, leave an exponent of 0. If there is an 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.

```

12467 \cs_new:Npn \_fp_parse_exponent:N #1
12468 {
12469   \if:w e \exp_not:N #1
12470     \exp_after:wN \_fp_parse_exponent_aux:N
12471     \exp:w
12472   \else:
12473     0 \_fp_parse_return_semicolon:w #1
12474   \fi:
12475   \_fp_parse_expand:w
12476 }
12477 \cs_new:Npn \_fp_parse_exponent_aux:N #1
12478 {
12479   \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #1
12480     \c_zero \else: ' #1 \fi: > '9 \exp_stop_f:
12481     0 \exp_after:wN ; \exp_after:wN e
12482   \else:
12483     \exp_after:wN \_fp_parse_exponent_sign:N
12484   \fi:
12485   #1
12486 }

```

(End definition for _fp_parse_exponent:N and _fp_parse_exponent_aux:N.)

_fp_parse_exponent_sign:N

Read signs one by one (if there is any).

```

12487 \cs_new:Npn \_fp_parse_exponent_sign:N #1
12488 {
12489   \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
12490     \exp_after:wN \_fp_parse_exponent_sign:N
12491     \exp:w \exp_after:wN \_fp_parse_expand:w
12492   \else:
12493     \exp_after:wN \_fp_parse_exponent_body:N
12494     \exp_after:wN #1
12495   \fi:
12496 }

```

(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).

```

12497 \cs_new:Npn \_fp_parse_exponent_body:N #1
12498 {
12499   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12500     \token_to_str:N #1
12501     \exp_after:wN \_fp_parse_exponent_digits:N
12502     \exp:w

```

```

12503 \else:
12504   \__fp_parse_exponent_keep:N \NTF #1
12505   { \__fp_parse_return_semicolon:w #1 }
12506   {
12507     \exp_after:wN ;
12508     \exp:w
12509   }
12510 \fi:
12511 \__fp_parse_expand:w
12512 }

```

(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.

```

12513 \cs_new:Npn \__fp_parse_exponent_digits:N #1
12514 {
12515   \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12516   \token_to_str:N #1
12517   \exp_after:wN \__fp_parse_exponent_digits:N
12518   \exp:w
12519 \else:
12520   \__fp_parse_return_semicolon:w #1
12521 \fi:
12522 \__fp_parse_expand:w
12523 }

```

(End definition for `__fp_parse_exponent_digits:N`.)

`__fp_parse_exponent_keep:N` 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.

```

12524 \prg_new_conditional:Npnn \__fp_parse_exponent_keep:N #1 { TF }
12525 {
12526   \if_catcode:w \scan_stop: \exp_not:N #1
12527   \if_meaning:w \scan_stop: #1
12528   \if_int_compare:w
12529     \__str_if_eq:x:nn { \s__fp } { \exp_not:N #1 } = \c_zero
12530     0
12531     \__msg_kernel_expandable_error:nnn
12532     { kernel } { fp-after-e } { floating~point~ }
12533   \prg_return_true:
12534 \else:
12535   0
12536   \__msg_kernel_expandable_error:nnn
12537   { kernel } { bad-variable } { #1 }

```

```

12538     \prg_return_false:
12539     \fi:
12540   \else:
12541     \if_int_compare:w
12542       \__str_if_eq_x:nn { \__int_value:w #1 } { \tex_the:D #1 }
12543       = \c_zero
12544       \__int_value:w #1
12545     \else:
12546       0
12547       \__msg_kernel_expandable_error:nnn
12548       { kernel } { fp-after-e } { dimension~#1 }
12549     \fi:
12550   \prg_return_false:
12551   \fi:
12552 \else:
12553   0
12554   \__msg_kernel_expandable_error:nnn
12555   { kernel } { fp-missing } { exponent }
12556 \prg_return_true:
12557 \fi:
12558 }

```

(End definition for __fp_parse_exponent_keep:NTF.)

25.5 Constants, functions and prefix operators

25.5.1 Prefix operators

__fp_parse_prefix_+:Nw A unary + does nothing: we should continue looking for a number.

```

12559 \cs_new_eq:cN { \__fp_parse_prefix_+:Nw } \__fp_parse_one:Nw

```

(End definition for __fp_parse_prefix_+:Nw.)

__fp_parse_apply_unary: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.

```

12560 \cs_new:Npn \__fp_parse_apply_unary:NNNwN #1#2#3#4#5
12561 {
12562   #3 #2 #4 @
12563   \exp:w \exp_end_continue_f:w #5 #1
12564 }

```

(End definition for __fp_parse_apply_unary:NNNwN.)

__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.

__fp_parse_prefix_!:Nw

```

12565 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
12566 {
12567   \cs_new:cpn { \__fp_parse_prefix_ #1 :Nw } ##1
12568   {
12569     \exp_after:wN \__fp_parse_apply_unary:NNNwN

```

```

12570         \exp_after:wN ##1
12571         \exp_after:wN #4
12572         \exp_after:wN #3
12573         \exp:w
12574         \if_int_compare:w #2 < ##1
12575             \__fp_parse_operand:Nw ##1
12576         \else:
12577             \__fp_parse_operand:Nw #2
12578         \fi:
12579         \__fp_parse_expand:w
12580     }
12581 }
12582 \__fp_tmp:w - \c__fp_prec_not_int \__fp_set_sign_o:w 2
12583 \__fp_tmp:w ! \c__fp_prec_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.

```

12584 \cs_new:cpn { __fp_parse_prefix_:Nw } #1
12585 {
12586     \exp_after:wN \__fp_parse_infix_after_operand:NwN
12587     \exp_after:wN #1
12588     \exp:w \exp_end_continue_f:w
12589     \exp_after:wN \__fp_sanitize:wN
12590     \__int_value:w \__int_eval:w \c_zero \__fp_parse_strim_zeros:N
12591 }

```

(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. Commas will be allowed if the previous precedence is 16 (function with multiple arguments). In this case, find an operand using the precedence 1; otherwise the precedence 0. 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 the array it found as an operand, fetching the following infix operator.

```

12592 \group_begin:
12593     \char_set_catcode_letter:N (
12594     \char_set_catcode_letter:N )
12595     \cs_new:Npn \__fp_parse_prefix(:Nw #1
12596     {
12597         \exp_after:wN \__fp_parse_lparen_after:NwN
12598         \exp_after:wN #1
12599         \exp:w
12600         \if_int_compare:w #1 = \c__fp_prec_funcii_int
12601             \__fp_parse_operand:Nw \c__fp_prec_comma_int
12602         \else:
12603             \__fp_parse_operand:Nw \c__fp_prec_paren_int
12604         \fi:
12605         \__fp_parse_expand:w
12606     }

```

```

12607 \cs_new:Npn \__fp_parse_lparen_after:NwN #1#2 @ #3
12608 {
12609     \token_if_eq_meaning:NNTF #3 \__fp_parse_infix_):N
12610     {
12611         \__fp_exp_after_array_f:w #2 \s__fp_stop
12612         \exp_after:wN \__fp_parse_infix:NN
12613         \exp_after:wN #1
12614         \exp:w \__fp_parse_expand:w
12615     }
12616     {
12617         \__msg_kernel_expandable_error:nnn
12618         { kernel } { fp-missing } { } }
12619     #2 @ \use_none:n #3
12620 }
12621 }
12622 \group_end:

```

(End definition for __fp_parse_prefix_(:Nw and __fp_parse_lparen_after:NwN.)

__fp_parse_prefix_):Nw

The right parenthesis can appear as unary prefixes when arguments of a multi-argument function end with a comma, or when there is no argument, as in `max(1,2,)` or in `rand()`. In single-argument functions (precedence 0 rather than 1) forbid this.

```

12623 \cs_new:cpn { __fp_parse_prefix_):Nw } #1
12624 {
12625     \if_int_compare:w #1 = \c__fp_prec_comma_int
12626     \else:
12627         \__msg_kernel_expandable_error:nnn
12628         { kernel } { fp-missing-number } { } }
12629     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
12630 \fi:
12631 \__fp_parse_infix:NN #1 )
12632 }

```

(End definition for __fp_parse_prefix_):Nw.)

25.5.2 Constants

__fp_parse_word_inf:N
__fp_parse_word_nan:N
__fp_parse_word_pi:N
__fp_parse_word_deg:N
__fp_parse_word_true:N
__fp_parse_word_false: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.

```

12633 \cs_set_protected:Npn \__fp_tmp:w #1 #2
12634 {
12635     \cs_new:cpn { __fp_parse_word_#1:N }
12636     { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN }
12637 }
12638 \__fp_tmp:w { inf } \c_inf_fp
12639 \__fp_tmp:w { nan } \c_nan_fp
12640 \__fp_tmp:w { pi } \c_pi_fp
12641 \__fp_tmp:w { deg } \c_one_degree_fp
12642 \__fp_tmp:w { true } \c_one_fp
12643 \__fp_tmp:w { false } \c_zero_fp

```

(End definition for __fp_parse_word_inf:N and others.)

`__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.

```

12644 \cs_set_protected:Npn \__fp_tmp:w #1 #2
12645 {
12646   \cs_new:cpn { __fp_parse_word_#1:N }
12647   {
12648     \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
12649     \s__fp \__fp_chk:w 10 #2 ;
12650   }
12651 }
12652 \__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
12653 \__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
12654 \__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
12655 \__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
12656 \__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
12657 \__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
12658 \__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
12659 \__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
12660 \__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
12661 \__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
12662 \__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`.

```

12663 \tl_map_inline:nn { {em} {ex} }
12664 {
12665   \cs_new:cpn { __fp_parse_word_#1:N }
12666   {
12667     \exp_after:wN \__fp_from_dim_test:ww
12668     \exp_after:wN 0 \exp_after:wN ,
12669     \__int_value:w \__dim_eval:w 1 #1 \exp_after:wN ;
12670     \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN
12671   }
12672 }
```

(End definition for `__fp_parse_word_em:N` and `__fp_parse_word_ex:N`.)

25.5.3 Functions

```

\__fp_parse_unary_function:nnn
\__fp_parse_function:nnn
12673 \cs_new:Npn \__fp_parse_unary_function:nnn #1#2#3
12674 {
12675   \exp_after:wN \__fp_parse_apply_unary:nnnwN
12676   \exp_after:wN #3
12677   \exp_after:wN #2
12678   \cs:w \__fp_#1_o:w \exp_after:wN \cs_end:
12679   \exp:w
12680   \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
12681 }
12682 \cs_new:Npn \__fp_parse_function:nnn #1#2#3
12683 {
12684   \exp_after:wN \__fp_parse_apply_unary:nnnwN
```



```

12685     \exp_after:wN #3
12686     \exp_after:wN #2
12687     \exp_after:wN #1
12688     \exp:w
12689     \__fp_parse_operand:Nw \c__fp_prec_funcii_int \__fp_parse_expand:w
12690 }

```

(End definition for __fp_parse_unary_function:nNN and __fp_parse_function:NNN.)

```

\__fp_parse_word_acot:N
\__fp_parse_word_acotd:N
\__fp_parse_word_atan:N
\__fp_parse_word_atand:N
\__fp_parse_word_max:N
\__fp_parse_word_min:N
\__fp_parse_word_rand:N
\__fp_parse_word_randint:N

```

Those functions are also unary (not binary), but may receive a variable number of arguments. For `randint` we don't use the first argument ?.

```

12691 \cs_new:Npn \__fp_parse_word_acot:N
12692 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
12693 \cs_new:Npn \__fp_parse_word_acotd:N
12694 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
12695 \cs_new:Npn \__fp_parse_word_atan:N
12696 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
12697 \cs_new:Npn \__fp_parse_word_atand:N
12698 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }
12699 \cs_new:Npn \__fp_parse_word_max:N
12700 { \__fp_parse_function:NNN \__fp_minmax_o:Nw 2 }
12701 \cs_new:Npn \__fp_parse_word_min:N
12702 { \__fp_parse_function:NNN \__fp_minmax_o:Nw 0 }
12703 \cs_new:Npn \__fp_parse_word_rand:N
12704 { \__fp_parse_function:NNN \__fp_rand_o:Nw ? }
12705 \cs_new:Npn \__fp_parse_word_randint:N
12706 { \__fp_parse_function:NNN \__fp_randint_o:Nw ? }

```

(End definition for __fp_parse_word_acot:N and others.)

```

\__fp_parse_word_abs:N
\__fp_parse_word_exp:N
\__fp_parse_word_ln:N
\__fp_parse_word_sqrt:N

```

Unary functions.

```

12707 \cs_new:Npn \__fp_parse_word_abs:N
12708 { \__fp_parse_unary_function:nNN { set_sign } 0 }
12709 \cs_new:Npn \__fp_parse_word_exp:N
12710 { \__fp_parse_unary_function:nNN {exp} ? }
12711 \cs_new:Npn \__fp_parse_word_ln:N
12712 { \__fp_parse_unary_function:nNN {ln} ? }
12713 \cs_new:Npn \__fp_parse_word_sqrt:N
12714 { \__fp_parse_unary_function:nNN {sqrt} ? }

```

(End definition for __fp_parse_word_abs:N and others.)

```

\__fp_parse_word_acos:N
\__fp_parse_word_acosd:N
\__fp_parse_word_acsc:N
\__fp_parse_word_acscd:N
\__fp_parse_word_asec:N
\__fp_parse_word_asecd: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
\__fp_parse_word_tan:N
\__fp_parse_word_tand:N

```

Unary functions.

```

12715 \tl_map_inline:nn
12716 {
12717   {acos} {acsc} {asec} {asin}
12718   {cos} {cot} {csc} {sec} {sin} {tan}
12719 }
12720 {
12721   \cs_new:cpn { __fp_parse_word_#1:N }
12722     { \__fp_parse_unary_function:nNN {#1} \use_i:nn }
12723   \cs_new:cpn { __fp_parse_word_#1d:N }
12724     { \__fp_parse_unary_function:nNN {#1} \use_ii:nn }
12725 }

```

(End definition for `_fp_parse_word_acos:N` and others.)

```

\_fp_parse_word_trunc:N
\_fp_parse_word_floor:N
\_fp_parse_word_ceil:N
12726 \cs_new:Npn \_fp_parse_word_trunc:N
12727 { \_fp_parse_function:NNN \_fp_round_o:Nw \_fp_round_to_zero:NNN }
12728 \cs_new:Npn \_fp_parse_word_floor:N
12729 { \_fp_parse_function:NNN \_fp_round_o:Nw \_fp_round_to_ninf:NNN }
12730 \cs_new:Npn \_fp_parse_word_ceil:N
12731 { \_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
12732 \cs_new:Npn \_fp_parse_word_round:N #1#2
12733 {
12734   \if_meaning:w + #2
12735     \_fp_parse_round:Nw \_fp_round_to_pinf:NNN
12736   \else:
12737     \if_meaning:w 0 #2
12738       \_fp_parse_round:Nw \_fp_round_to_zero:NNN
12739     \else:
12740       \if_meaning:w - #2
12741         \_fp_parse_round:Nw \_fp_round_to_ninf:NNN
12742       \fi:
12743     \fi:
12744   \fi:
12745   \_fp_parse_function:NNN
12746   \_fp_round_o:Nw \_fp_round_to_nearest:NNN #1
12747   #2
12748 }
12749 \cs_new:Npn \_fp_parse_round:Nw
12750 #1 #2 \_fp_round_to_nearest:NNN #3#4 { #2 #1 #3 }

```

(End definition for `_fp_parse_word_round:N` and `_fp_parse_round:Nw`.)

25.6 Main functions

`_fp_parse:n` Start an `\exp:w` expansion so that `_fp_parse:n` expands in two steps. The `_fp_parse_operand:Nw` function will perform computations until reaching an operation with precedence `\c_fp_prec_end_int` or less, namely, the end of the expression. The marker `\s_fp_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:.`

```

12751 \cs_new:Npn \_fp_parse:n #1
12752 {
12753   \exp:w
12754   \exp_after:wN \_fp_parse_after:ww
12755   \exp:w
12756   \_fp_parse_operand:Nw \c_fp_prec_end_int
12757   \_fp_parse_expand:w #1
12758   \s_fp_mark \_fp_parse_infix_end:N
12759   \s_fp_stop

```

```

12760 }
12761 \cs_new:Npn \__fp_parse_after:ww
12762   #1@ \__fp_parse_infix_end:N \s__fp_stop
12763   { \exp_end: #1 }

```

(End definition for __fp_parse:n and __fp_parse_after:ww.)

__fp_parse_o:n

```

12764 \cs_new:Npn \__fp_parse_o:n #1
12765 {
12766   \exp_after:wN \exp_after:wN
12767   \exp_after:wN \__fp_exp_after_o:w
12768   \__fp_parse:n {#1}
12769 }

```

(End definition for __fp_parse_o:n.)

__fp_parse_operand:Nw
 __fp_parse_continue:NwN

The __fp_parse_operand This is just a shorthand which sets up both __fp_parse_continue and __fp_parse_one with the same precedence. Note the trailing \exp:w. This function should be used with much care.

```

12770 \cs_new:Npn \__fp_parse_operand:Nw #1
12771 {
12772   \exp_end_continue_f:w
12773   \exp_after:wN \__fp_parse_continue:NwN
12774   \exp_after:wN #1
12775   \exp:w \exp_end_continue_f:w
12776   \exp_after:wN \__fp_parse_one:Nw
12777   \exp_after:wN #1
12778   \exp:w
12779 }
12780 \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

Receives $\langle precedence \rangle \langle operand_1 \rangle @ \langle operation \rangle \langle operand_2 \rangle @ \langle infix command \rangle$. Builds the appropriate call to the $\langle operation \rangle$ #3.

```

12781 \cs_new:Npn \__fp_parse_apply_binary:NwNwN #1 #2@ #3 #4@ #5
12782 {
12783   \exp_after:wN \__fp_parse_continue:NwN
12784   \exp_after:wN #1
12785   \exp:w \exp_end_continue_f:w \cs:w __fp_#3_o:ww \cs_end: #2 #4
12786   \exp:w \exp_end_continue_f:w #5 #1
12787 }

```

(End definition for _fp_parse_apply_binary:NwNwN.)

25.7 Infix operators

_fp_parse_infix_after_operand:NwN

```

12788 \cs_new:Npn \__fp_parse_infix_after_operand:NwN #1 #2;
12789 {
12790   \__fp_exp_after_f:nw { \__fp_parse_infix:NN #1 }
12791   #2;
12792 }

```

```

12793 \group_begin:
12794   \char_set_catcode_letter:N \*
12795   \cs_new:Npn \__fp_parse_infix:NN #1 #2
12796   {
12797     \if_catcode:w \scan_stop: \exp_not:N #2
12798     \if_int_compare:w
12799       \__str_if_eq_x:nn { \s__fp_mark } { \exp_not:N #2 }
12800       = \c_zero
12801       \exp_after:wN \exp_after:wN
12802       \exp_after:wN \__fp_parse_infix_mark:NNN
12803     \else:
12804       \exp_after:wN \exp_after:wN
12805       \exp_after:wN \__fp_parse_infix_juxtapose:N
12806     \fi:
12807   \else:
12808     \if_int_compare:w
12809       \__int_eval:w
12810       ( '#2 \if_int_compare:w '#2 > 'Z - \c_thirty_two \fi: )
12811       / 26
12812       = \c_three
12813       \exp_after:wN \exp_after:wN
12814       \exp_after:wN \__fp_parse_infix_juxtapose:N
12815     \else:
12816       \exp_after:wN \__fp_parse_infix_check:NNN
12817       \cs:w
12818         \__fp_parse_infix_ \token_to_str:N #2 :N
12819       \exp_after:wN \exp_after:wN \exp_after:wN
12820     \cs_end:
12821     \fi:
12822   \fi:
12823   #1
12824   #2
12825 }
12826 \cs_new:Npn \__fp_parse_infix_check:NNN #1#2#3
12827 {
12828   \if_meaning:w \scan_stop: #1
12829     \__msg_kernel_expandable_error:nnn
12830     { kernel } { fp-missing } { * }
12831     \exp_after:wN \__fp_parse_infix_*:N
12832     \exp_after:wN #2
12833     \exp_after:wN #3
12834   \else:
12835     \exp_after:wN #1
12836     \exp_after:wN #2
12837     \exp:w \exp_after:wN \__fp_parse_expand:w
12838   \fi:
12839 }
12840 \group_end:

```

(End definition for __fp_parse_infix_after_operand:NwN.)

25.7.1 Closing parentheses and commas

`__fp_parse_infix_mark:NNN` As an infix operator, `\s__fp_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.

```
12841 \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.

```
12842 \cs_new:Npn \__fp_parse_infix_end:N #1
12843 { @ \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.

```
12844 \group_begin:
12845   \char_set_catcode_letter:N \)
12846   \cs_new:Npn \__fp_parse_infix_):N #1
12847   {
12848     \if_int_compare:w #1 < \c__fp_prec_paren_int
12849       \_msg_kernel_expandable_error:nnn { kernel } { fp-extra } { ) }
12850       \exp_after:wN \__fp_parse_infix:NN
12851       \exp_after:wN #1
12852       \exp:w \exp_after:wN \__fp_parse_expand:w
12853     \else:
12854       \exp_after:wN @
12855       \exp_after:wN \use_none:n
12856       \exp_after:wN \__fp_parse_infix_):N
12857     \fi:
12858   }
12859 \group_end:
```

(End definition for `__fp_parse_infix_):N`.)

`__fp_parse_infix_,:N` `__fp_,_o:ww` is a complicated way of replacing any number of floating point arguments by `nan`.

```
\__fp_parse_infix_comma:w
  \__fp_parse_infix_comma_error:w
  \__fp_,_o:ww
12860 \group_begin:
12861   \char_set_catcode_letter:N \,
12862   \cs_new:Npn \__fp_parse_infix_,:N #1
12863   {
12864     \if_int_compare:w #1 > \c__fp_prec_comma_int
12865       \exp_after:wN @
12866       \exp_after:wN \use_none:n
12867       \exp_after:wN \__fp_parse_infix_,:N
12868     \else:
12869       \if_int_compare:w #1 < \c__fp_prec_comma_int
12870         \__fp_parse_infix_comma_error:w
12871       \fi:
12872       \exp_after:wN \__fp_parse_infix_comma:w
12873       \exp:w \__fp_parse_operand:Nw \c__fp_prec_comma_int
12874       \exp_after:wN \__fp_parse_expand:w
```

```

12875     \fi:
12876   }
12877   \cs_new:Npn \__fp_parse_infix_comma:w #1 @
12878     { #1 @ \use_none:n }
12879   \cs_new:Npn \__fp_parse_infix_comma_error:w #1 \exp:w
12880     {
12881       \fi:
12882       \__msg_kernel_expandable_error:nn { kernel } { fp-extra-comma }
12883       \exp_after:wN @
12884       \exp_after:wN \__fp_parse_apply_binary:NwNwN
12885       \exp_after:wN ,
12886       \exp:w
12887     }
12888   \cs_new:Npn \__fp_,_o:ww #1
12889     {
12890       \if_meaning:w \s_fp #1
12891         \exp_after:wN \__fp_use_i_until_s:nw
12892         \exp_after:wN \__fp_,_o:ww
12893       \fi:
12894       \exp_after:wN \c_nan_fp
12895       #1
12896     }
12897   \group_end:

```

(End definition for `__fp_parse_infix_,:N` and others.)

25.7.2 Usual infix operators

`__fp_parse_infix_+:N` As described in the “work plan”, each infix operator has an associated `\..._infix_...`
`__fp_parse_infix_-:N` function, a computing function, and precedence, given as arguments to `__fp_tmp:w`.
`__fp_parse_infix/:N` Using the general mechanism for arithmetic operations. The power operation must be
`__fp_parse_infix_mul:N` associative in the opposite order from all others. For this, we use two distinct precedences.
`__fp_parse_infix_and:N` The odd requirement to set `\+` here is to cover the case where `expl3` is loaded by
`__fp_parse_infix_or:N` plain `TEX`: `\+` is an `\outer` macro there, and so the following code would otherwise give
`__fp_parse_infix^:N` an error in that case.

```

12898 \group_begin:
12899 (*package)
12900   \cs_set:Npn \+ { }
12901 </package>
12902   \char_set_catcode_other:N \&
12903   \char_set_catcode_letter:N \^
12904   \char_set_catcode_letter:N \ /
12905   \char_set_catcode_letter:N \-
12906   \char_set_catcode_letter:N \+
12907   \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
12908     {
12909       \cs_new:Npn #1 ##1
12910         {
12911           \if_int_compare:w ##1 < #3
12912             \exp_after:wN @
12913             \exp_after:wN \__fp_parse_apply_binary:NwNwN
12914             \exp_after:wN #2
12915             \exp:w
12916             \__fp_parse_operand:Nw #4

```

```

12917         \exp_after:wN \__fp_parse_expand:w
12918     \else:
12919         \exp_after:wN @
12920         \exp_after:wN \use_none:n
12921         \exp_after:wN #1
12922     \fi:
12923 }
12924 }
12925 \__fp_tmp:w \__fp_parse_infix_^:N ^ \c__fp_prec_hatii_int \c__fp_prec_hat_int
12926 \__fp_tmp:w \__fp_parse_infix_/:N / \c__fp_prec_times_int \c__fp_prec_times_int
12927 \__fp_tmp:w \__fp_parse_infix_mul:N * \c__fp_prec_times_int \c__fp_prec_times_int
12928 \__fp_tmp:w \__fp_parse_infix -:N - \c__fp_prec_plus_int \c__fp_prec_plus_int
12929 \__fp_tmp:w \__fp_parse_infix_+:N + \c__fp_prec_plus_int \c__fp_prec_plus_int
12930 \__fp_tmp:w \__fp_parse_infix_and:N & \c__fp_prec_and_int \c__fp_prec_and_int
12931 \__fp_tmp:w \__fp_parse_infix_or:N | \c__fp_prec_or_int \c__fp_prec_or_int
12932 \group_end:

```

(End definition for __fp_parse_infix_+:N and others.)

25.7.3 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_juxtapose:N.

```

12933 \cs_new:cpn { __fp_parse_infix_(:N } #1
12934 { \__fp_parse_infix_juxtapose:N #1 ( }

```

(End definition for __fp_parse_infix_(:N)

__fp_parse_infix_juxtapose:N Juxtaposition follows the same scheme as other binary operations, but calls __fp_parse_apply_juxtapose:NwwN rather than directly calling __fp_parse_apply_binary:NwwN. This lets us catch errors such as ... (1,2,3)pt where one operand of the juxtaposition is not a single number: both #3 and #5 of the apply auxiliary must be empty.

```

12935 \cs_new:Npn \__fp_parse_infix_juxtapose:N #1
12936 {
12937     \if_int_compare:w #1 < \c__fp_prec_times_int
12938         \exp_after:wN @
12939         \exp_after:wN \__fp_parse_apply_juxtapose:NwwN
12940         \exp:w
12941         \__fp_parse_operand:Nw \c__fp_prec_times_int
12942         \exp_after:wN \__fp_parse_expand:w
12943     \else:
12944         \exp_after:wN @
12945         \exp_after:wN \use_none:n
12946         \exp_after:wN \__fp_parse_infix_juxtapose:N
12947     \fi:
12948 }
12949 \cs_new:Npn \__fp_parse_apply_juxtapose:NwwN #1 #2;#3@ #4;#5@
12950 {
12951     \if_catcode:w ^ \tl_to_str:n { #3 #5 } ^
12952     \else:
12953         \__fp_error:nffn { invalid-ii }
12954         { \__fp_array_to_clist:n { #2; #3 } }

```

```

12955         { \_fp_array_to_clist:n { #4; #5 } }
12956     { }
12957 \fi:
12958 \_fp_parse_apply_binary:NwNwN #1 #2;@ * #4;@
12959 }

```

(End definition for _fp_parse_infix_juxtapose:N and _fp_parse_apply_juxtapose:NwwN.)

25.7.4 Multi-character cases

_fp_parse_infix_*:N

```

12960 \group_begin:
12961   \char_set_catcode_letter:N ^
12962   \cs_new:cpn { \_fp_parse_infix_*:N } #1#2
12963   {
12964     \if:w * \exp_not:N #2
12965       \exp_after:wN \_fp_parse_infix_~:N
12966       \exp_after:wN #1
12967     \else:
12968       \exp_after:wN \_fp_parse_infix_mul:N
12969       \exp_after:wN #1
12970       \exp_after:wN #2
12971     \fi:
12972   }
12973 \group_end:

```

(End definition for _fp_parse_infix_*:N.)

_fp_parse_infix_|:Nw

_fp_parse_infix_&:Nw

```

12974 \group_begin:
12975   \char_set_catcode_letter:N \
12976   \char_set_catcode_letter:N &
12977   \cs_new:Npn \_fp_parse_infix_|:N #1#2
12978   {
12979     \if:w | \exp_not:N #2
12980       \exp_after:wN \_fp_parse_infix_|:N
12981       \exp_after:wN #1
12982       \exp:w \exp_after:wN \_fp_parse_expand:w
12983     \else:
12984       \exp_after:wN \_fp_parse_infix_or:N
12985       \exp_after:wN #1
12986       \exp_after:wN #2
12987     \fi:
12988   }
12989   \cs_new:Npn \_fp_parse_infix_&:N #1#2
12990   {
12991     \if:w & \exp_not:N #2
12992       \exp_after:wN \_fp_parse_infix_&:N
12993       \exp_after:wN #1
12994       \exp:w \exp_after:wN \_fp_parse_expand:w
12995     \else:
12996       \exp_after:wN \_fp_parse_infix_and:N
12997       \exp_after:wN #1
12998       \exp_after:wN #2

```



```

12999     \fi:
13000   }
13001 \group_end:

```

(End definition for _fp_parse_infix_!:Nw and _fp_parse_infix_&:Nw.)

25.7.5 Ternary operator

```

\_fp_parse_infix_?:N
\_fp_parse_infix_:N

```

```

13002 \group_begin:
13003   \char_set_catcode_letter:N \?
13004   \cs_new:Npn \_fp_parse_infix_?:N #1
13005     {
13006       \if_int_compare:w #1 < \c__fp_prec_quest_int
13007         \exp_after:wN @
13008         \exp_after:wN \_fp_ternary:NwwN
13009         \exp:w
13010         \_fp_parse_operand:Nw \c__fp_prec_quest_int
13011         \exp_after:wN \_fp_parse_expand:w
13012       \else:
13013         \exp_after:wN @
13014         \exp_after:wN \use_none:n
13015         \exp_after:wN \_fp_parse_infix_?:N
13016       \fi:
13017     }
13018   \cs_new:Npn \_fp_parse_infix_:N #1
13019     {
13020       \if_int_compare:w #1 < \c__fp_prec_quest_int
13021         \_msg_kernel_expandable_error:nnnn
13022         { kernel } { fp-missing } { ? } { ~for~?: }
13023         \exp_after:wN @
13024         \exp_after:wN \_fp_ternary_auxii:NwwN
13025         \exp:w
13026         \_fp_parse_operand:Nw \c__fp_prec_colon_int
13027         \exp_after:wN \_fp_parse_expand:w
13028       \else:
13029         \exp_after:wN @
13030         \exp_after:wN \use_none:n
13031         \exp_after:wN \_fp_parse_infix_:N
13032       \fi:
13033     }
13034 \group_end:

```

(End definition for _fp_parse_infix_?:N and _fp_parse_infix_:N.)

25.7.6 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:NNNNN
\_fp_parse_compare_end:NNNNw
\_fp_compare:wNNNNw

```

```

13035 \cs_new:cpn { \_fp_parse_infix_<:N } #1
13036   {
13037     \_fp_parse_compare:NNNNNNN #1 \c_one
13038     \c_zero \c_zero \c_zero \c_zero <
13039   }
13040 \cs_new:cpn { \_fp_parse_infix_=:N } #1

```

```

13041 {
13042     \__fp_parse_compare:NNNNNNN #1 \c_one
13043     \c_zero \c_zero \c_zero \c_zero =
13044 }
13045 \cs_new:cpn { __fp_parse_infix_>:N } #1
13046 {
13047     \__fp_parse_compare:NNNNNNN #1 \c_one
13048     \c_zero \c_zero \c_zero \c_zero >
13049 }
13050 \cs_new:cpn { __fp_parse_infix_!:N } #1
13051 {
13052     \exp_after:wN \__fp_parse_compare:NNNNNNN
13053     \exp_after:wN #1
13054     \exp_after:wN \c_zero
13055     \exp_after:wN \c_one
13056     \exp_after:wN \c_one
13057     \exp_after:wN \c_one
13058     \exp_after:wN \c_one
13059 }
13060 \cs_new:Npn \__fp_parse_excl_error:
13061 {
13062     \__msg_kernel_expandable_error:nnnn
13063     { kernel } { fp-missing } { = } { ~after~!. }
13064 }
13065 \cs_new:Npn \__fp_parse_compare:NNNNNNN #1
13066 {
13067     \if_int_compare:w #1 < \c__fp_prec_comp_int
13068     \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
13069     \exp_after:wN \__fp_parse_excl_error:
13070     \else:
13071     \exp_after:wN @
13072     \exp_after:wN \use_none:n
13073     \exp_after:wN \__fp_parse_compare:NNNNNNN
13074     \fi:
13075 }
13076 \cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
13077 {
13078     \if_case:w
13079         \if_catcode:w \scan_stop: \exp_not:N #7
13080         \c_four
13081     \else:
13082         \__int_eval:w '#7 - '< \__int_eval_end:
13083     \fi:
13084     \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
13085     \or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
13086     \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
13087     \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#5#2
13088     \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
13089     \fi:
13090 }
13091 \cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
13092 {
13093     \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
13094     \exp_after:wN \prg_do_nothing:

```

```

13095     \exp_after:wN #1
13096     \exp_after:wN #2
13097     \exp_after:wN #3
13098     \exp_after:wN #4
13099     \exp_after:wN #5
13100     \exp:w \exp_after:wN \__fp_parse_expand:w
13101   }
13102 \cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
13103 {
13104   \fi:
13105   \exp_after:wN @
13106   \exp_after:wN \__fp_parse_apply_compare:NwNNNNNwN
13107   \exp_after:wN \c_one_fp
13108   \exp_after:wN #1
13109   \exp_after:wN #2
13110   \exp_after:wN #3
13111   \exp_after:wN #4
13112   \exp:w
13113   \__fp_parse_operand:Nw \c__fp_prec_comp_int \__fp_parse_expand:w #5
13114 }
13115 \cs_new:Npn \__fp_parse_apply_compare:NwNNNNNwN
13116   #1 #2@ #3 #4#5#6#7 #8@ #9
13117 {
13118   \if_int_odd:w
13119     \if_meaning:w \c_zero_fp #3
13120     \c_zero
13121   \else:
13122     \if_case:w \__fp_compare_back:ww #8 #2 \exp_stop_f:
13123       #5 \or: #6 \or: #7 \else: #4
13124     \fi:
13125     \fi:
13126     \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
13127     \exp_after:wN \c_one_fp
13128   \else:
13129     \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
13130     \exp_after:wN \c_zero_fp
13131   \fi:
13132   #1 #8 #9
13133 }
13134 \cs_new:Npn \__fp_parse_apply_compare_aux:NNwN #1 #2 #3; #4
13135 {
13136   \if_meaning:w \__fp_parse_compare:NNNNNNN #4
13137     \exp_after:wN \__fp_parse_continue_compare:NNwNN
13138     \exp_after:wN #1
13139     \exp_after:wN #2
13140     \exp:w \exp_end_continue_f:w
13141     \__fp_exp_after_o:w #3;
13142     \exp:w \exp_end_continue_f:w
13143   \else:
13144     \exp_after:wN \__fp_parse_continue:NwN
13145     \exp_after:wN #2
13146     \exp:w \exp_end_continue_f:w
13147     \exp_after:wN #1
13148     \exp:w \exp_end_continue_f:w

```

```

13149     \fi:
13150     #4 #2
13151   }
13152 \cs_new:Npn \__fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
13153   { #4 #2 #3@ #1 }

```

(End definition for __fp_parse_infix_<:N and others.)

25.8 Candidate: defining new l3fp functions

\fp_function:Nw Parse the argument of the function #1 using __fp_parse_operand:Nw with a precedence of 16, and pass the function and argument to __fp_function_apply:nw.

```

13154 \cs_new:Npn \fp_function:Nw #1
13155   {
13156     \exp_after:wN \__fp_function_apply:nw
13157     \exp_after:wN #1
13158     \exp:w
13159     \__fp_parse_operand:Nw \c__fp_prec_funcii_int \__fp_parse_expand:w
13160   }

```

(End definition for \fp_function:Nw.)

\fp_new_function:Npn Save the code provided by the user in the control sequence __fp_user_#1. Define __fp_new_function:NNnnn #1 to call __fp_function_apply:nw after parsing one operand using __fp_parse_operand:Nw with precedence 16. The auxiliary __fp_function_args:Nwn receives the user function and the number of arguments (half of the number of tokens in the parameter text #2), followed by the operand (as a token list of floating points). It checks the number of arguments, and applies the user function to the arguments (without the outer brace group).

```

13161 \cs_new_protected:Npn \fp_new_function:Npn #1#2#
13162   {
13163     \__fp_new_function:Ncfnn #1
13164     { \__fp_user_ \cs_to_str:N #1 }
13165     { \int_eval:n { \tl_count:n {#2} / \c_two } }
13166     {#2}
13167   }
13168 \cs_new_protected:Npn \__fp_new_function:NNnnn #1#2#3#4#5
13169   {
13170     \cs_new:Npn #1
13171     {
13172       \exp_after:wN \__fp_function_apply:nw \exp_after:wN
13173       {
13174         \exp_after:wN \__fp_function_args:Nwn
13175         \exp_after:wN #2
13176         \__int_value:w #3 \exp_after:wN ; \exp_after:wN
13177       }
13178       \exp:w
13179       \__fp_parse_operand:Nw \c__fp_prec_funcii_int \__fp_parse_expand:w
13180     }
13181     \cs_new:Npn #2 #4 {#5}
13182   }
13183 \cs_generate_variant:Nn \__fp_new_function:NNnnn { Ncf }
13184 \cs_new:Npn \__fp_function_args:Nwn #1#2; #3
13185   {

```

```

13186 \int_compare:nNnTF { \tl_count:n {#3} } = {#2}
13187 { #1 #3 }
13188 {
13189   \__msg_kernel_expandable_error:nnnnn
13190   { kernel } { fp-num-args } { #1() } {#2} {#2}
13191   \c_nan_fp
13192 }
13193 }

```

(End definition for \fp_new_function:Npn, __fp_new_function:NNnnn, and __fp_function_args:Nwn.)

```

\__fp_function_apply:nw
\__fp_function_store:wwNwnn
\__fp_function_store_end:wnnn

```

The auxiliary __fp_function_apply:nw is called after parsing an operand, so it receives some code #1, then the operand ending with @, then a function such as __fp_parse_infix_+:N (but not always of this form, see comparisons for instance). Package the operand (an array) into a token list with floating point items: this is the role of __fp_function_store:wwNwnn and __fp_function_store_end:wnnn. Then apply __fp_parse:n to the code #1 followed by a brace group with this token list. This results in a floating point result, which will correctly be parsed as the next operand of whatever was looking for one. The trailing \s__fp_mark is used as a special infix operator to indicate that the next token has already gone through __fp_parse_infix:NN.

```

13194 \cs_new:Npn \__fp_function_apply:nw #1#2 @
13195 {
13196   \__fp_parse:n
13197   {
13198     \__fp_function_store:wwNwnn #2
13199     \s__fp_mark \__fp_function_store:wwNwnn ;
13200     \s__fp_mark \__fp_function_store_end:wnnn
13201     \s__fp_stop { } { } {#1}
13202   }
13203   \s__fp_mark
13204 }
13205 \cs_new:Npn \__fp_function_store:wwNwnn
13206   #1; #2 \s__fp_mark #3#4 \s__fp_stop #5#6
13207   { #3 #2 \s__fp_mark #3#4 \s__fp_stop { #5 #6 } { { #1; } } }
13208 \cs_new:Npn \__fp_function_store_end:wnnn
13209   #1 \s__fp_stop #2#3#4
13210   { #4 {#2} }

```

(End definition for __fp_function_apply:nw, __fp_function_store:wwNwnn, and __fp_function_store_end:wnnn.)

25.9 Messages

```

13211 \__msg_kernel_new:nnn { kernel } { unknown-fp-word }
13212 { Unknown~fp-word~#1. }
13213 \__msg_kernel_new:nnn { kernel } { fp-missing }
13214 { Missing~#1~inserted #2. }
13215 \__msg_kernel_new:nnn { kernel } { fp-extra }
13216 { Extra~#1~ignored. }
13217 \__msg_kernel_new:nnn { kernel } { fp-early-end }
13218 { Premature~end~in~fp-expression. }
13219 \__msg_kernel_new:nnn { kernel } { fp-after-e }
13220 { Cannot~use~#1 after~'e'. }
13221 \__msg_kernel_new:nnn { kernel } { fp-missing-number }

```

```

13222 { Missing~number~before~'~#1'. }
13223 \__msg_kernel_new:nnn { kernel } { fp-unknown-symbol }
13224 { Unknown~symbol~#1~ignored. }
13225 \__msg_kernel_new:nnn { kernel } { fp-extra-comma }
13226 { Unexpected~comma:~extra~arguments~ignored. }
13227 \__msg_kernel_new:nnn { kernel } { fp-num-args }
13228 { #1~expects~between~#2~and~#3~arguments. }
13229 \*package
13230 \cs_if_exist:cT { @unexpandable@protect }
13231 {
13232   \__msg_kernel_new:nnn { kernel } { fp-robust-cmd }
13233   { Robust~command~#1~invalid~in~fp~expression! }
13234 }
13235 \</package>
13236 \</initex | package>

```

26 l3fp-logic Implementation

```

13237 \*initex | package>
13238 \<@@=fp>

```

26.1 Syntax of internal functions

- __fp_compare_npos:nwnw {<exp₀>} <body₁> ; {<exp₀>} <body₂> ;
- __fp_minmax_o:Nw <sign> <floating point array>
- __fp_not_o:w ? <floating point array> (with one floating point number only)
- __fp_&_o:ww <floating point> <floating point>
- __fp_|_o:ww <floating point> <floating point>
- __fp_ternary:NwN, __fp_ternary_auxi:NwN, __fp_ternary_auxii:NwN have to be understood.

26.2 Existence test

\fp_if_exist_p:N Copies of the cs functions defined in l3basics.

```

\fp_if_exist_p:c 13239 \prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
\fp_if_exist:NTF 13240 \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:N~~TF~~. This function is documented on page 184.)

26.3 Comparison

\fp_compare_p:n Within floating point expressions, comparison operators are treated as operations, so we evaluate #1, then compare with 0.

\fp_compare:n~~TF~~

```

\__fp_compare_return:w 13241 \prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
13242 {
13243   \exp_after:wN \__fp_compare_return:w
13244   \exp:w \exp_end_continue_f:w \__fp_parse:n {#1}
13245 }
13246 \cs_new:Npn \__fp_compare_return:w \s__fp \__fp_chk:w #1#2;

```

```

13247 {
13248   \if_meaning:w 0 #1
13249   \prg_return_false:
13250   \else:
13251     \prg_return_true:
13252   \fi:
13253 }

```

(End definition for `\fp_compare:nTF` and `__fp_compare_return:w`. These functions are documented on page 185.)

`\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:ww`, defined below. Compare the result with `\fp_compare:nNnTF` `'#2-'`, which is -1 for $<$, 0 for $=$, 1 for $>$ and 2 for $?$.
`__fp_compare_aux:wn`

```

13254 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
13255 {
13256   \if_int_compare:w
13257     \exp_after:wN \__fp_compare_aux:wn
13258     \exp:w \exp_end_continue_f:w \__fp_parse:n {#1} {#3}
13259     = \__int_eval:w '#2 - ' = \__int_eval_end:
13260     \prg_return_true:
13261   \else:
13262     \prg_return_false:
13263   \fi:
13264 }
13265 \cs_new:Npn \__fp_compare_aux:wn #1; #2
13266 {
13267   \exp_after:wN \__fp_compare_back:ww
13268   \exp:w \exp_end_continue_f:w \__fp_parse:n {#2} #1;
13269 }

```

(End definition for `\fp_compare:nNnTF` and `__fp_compare_aux:wn`. These functions are documented on page 185.)

`__fp_compare_back:ww` `__fp_compare_back:ww` $\langle y \rangle$; $\langle x \rangle$;
`__fp_compare_nan:w`

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.

```

13270 \cs_new:Npn \__fp_compare_back:ww
13271   \s__fp \__fp_chk:w #1 #2 #3;
13272   \s__fp \__fp_chk:w #4 #5 #6;
13273 {
13274   \__int_value:w
13275   \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
13276   \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
13277   \if_meaning:w 2 #5 - \fi:
13278   \if_meaning:w #2 #5
13279     \if_meaning:w #1 #4
13280       \if_meaning:w 1 #1
13281         \__fp_compare_npos:nwnw #6; #3;

```

```

13282         \else:
13283             0
13284         \fi:
13285     \else:
13286         \if_int_compare:w #4 < #1 - \fi: 1
13287     \fi:
13288 \else:
13289     \if_int_compare:w #1#4 = \c_zero
13290     0
13291 \else:
13292     1
13293 \fi:
13294 \fi:
13295 \exp_stop_f:
13296 }
13297 \cs_new:Npn \__fp_compare_nan:w #1 \exp_stop_f: { \c_two }

```

(End definition for __fp_compare_back:ww and __fp_compare_nan:w.)

```

\__fp_compare_npos:nwnw \__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.

```

13298 \cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
13299 {
13300     \if_int_compare:w #1 = #3 \exp_stop_f:
13301         \__fp_compare_significand:nnnnnnnn #2 #4
13302     \else:
13303         \if_int_compare:w #1 < #3 - \fi: 1
13304     \fi:
13305 }
13306 \cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
13307 {
13308     \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
13309     \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
13310     0
13311     \else:
13312         \if_int_compare:w #3#4 < #7#8 - \fi: 1
13313     \fi:
13314 \else:
13315     \if_int_compare:w #1#2 < #5#6 - \fi: 1
13316 \fi:
13317 }

```

(End definition for __fp_compare_npos:nwnw and __fp_compare_significand:nnnnnnnn.)

26.4 Floating point expression loops

\fp_do_until: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.

\fp_do_while:nn

\fp_until_do:nn

\fp_while_do:nn


```

13318 \cs_new:Npn \fp_do_until:nn #1#2
13319 {
13320     #2
13321     \fp_compare:nF {#1}
13322     { \fp_do_until:nn {#1} {#2} }
13323 }
13324 \cs_new:Npn \fp_do_while:nn #1#2
13325 {
13326     #2
13327     \fp_compare:nT {#1}
13328     { \fp_do_while:nn {#1} {#2} }
13329 }
13330 \cs_new:Npn \fp_until_do:nn #1#2
13331 {
13332     \fp_compare:nF {#1}
13333     {
13334         #2
13335         \fp_until_do:nn {#1} {#2}
13336     }
13337 }
13338 \cs_new:Npn \fp_while_do:nn #1#2
13339 {
13340     \fp_compare:nT {#1}
13341     {
13342         #2
13343         \fp_while_do:nn {#1} {#2}
13344     }
13345 }

```

(End definition for \fp_do_until:nn and others. These functions are documented on page 186.)

```

\fp_do_until:nNnn As above but not using the nNn syntax.
\fp_do_while:nNnn
\fp_until_do:nNnn
\fp_while_do:nNnn
13346 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
13347 {
13348     #4
13349     \fp_compare:nNnF {#1} #2 {#3}
13350     { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
13351 }
13352 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
13353 {
13354     #4
13355     \fp_compare:nNnT {#1} #2 {#3}
13356     { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
13357 }
13358 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
13359 {
13360     \fp_compare:nNnF {#1} #2 {#3}
13361     {
13362         #4
13363         \fp_until_do:nNnn {#1} #2 {#3} {#4}
13364     }
13365 }
13366 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
13367 {

```

```

13368 \fp_compare:nNnT {#1} #2 {#3}
13369 {
13370     #4
13371     \fp_while_do:nNnn {#1} #2 {#3} {#4}
13372 }
13373 }

```

(End definition for `\fp_do_until:nNnn` and others. These functions are documented on page 186.)

\fp_step_function:nnnN

\fp_step_function:nnnc

`__fp_step: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 will increase.

```

13374 \cs_new:Npn \fp_step_function:nnnN #1#2#3
13375 {
13376     \exp_after:wN \__fp_step:wwwN
13377     \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#1}
13378     \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#2}
13379     \exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
13380 }
13381 \cs_generate_variant:Nn \fp_step_function:nnnN { nnnc }
13382 % \end{macrocode}
13383 % Only \enquote{normal} floating points (not $\pm 0$,
13384 % $\pm\texttt{inf}$, $\texttt{nan}$) can be used as step; if positive,
13385 % call \cs{__fp_step:NnnnnN} with argument |>| otherwise~|<|. This
13386 % function has one more argument than its integer counterpart, namely
13387 % the previous value, to catch the case where the loop has made no
13388 % progress. Conversion to decimal is done just before calling the
13389 % user's function.
13390 % \begin{macrocode}
13391 \cs_new:Npn \__fp_step:wwwN #1 ; \s__fp \__fp_chk:w #2#3#4 ; #5 ; #6
13392 {
13393     \token_if_eq_meaning:NNTF #2 1
13394     {
13395         \token_if_eq_meaning:NNTF #3 0
13396         { \__fp_step:NnnnnN > }
13397         { \__fp_step:NnnnnN < }
13398     }
13399     {
13400         \token_if_eq_meaning:NNTF #2 0
13401         { \_msg_kernel_expandable_error:nnn { kernel } { zero-step } {#6} }
13402         {
13403             \__fp_error:nfn { bad-step } { }
13404             { \fp_to_tl:n { \s__fp \__fp_chk:w #2#3#4 ; } } {#6}
13405         }
13406         \use_none:nnnnn
13407     }
13408     { #1 ; } { \c_nan_fp } { \s__fp \__fp_chk:w #2#3#4 ; } { #5 ; } #6
13409 }
13410 \cs_new:Npn \__fp_step:NnnnnN #1#2#3#4#5#6
13411 {
13412     \fp_compare:nNnTF {#2} = {#3}
13413     {
13414         \__fp_error:nfn { tiny-step }

```

```

13415         { \fp_to_tl:n {#3} } { \fp_to_tl:n {#4} } {#6}
13416     }
13417     {
13418         \fp_compare:nNnF {#2} #1 {#5}
13419         {
13420             \exp_args:Nf #6 { \__fp_to_decimal_dispatch:w #2 }
13421             \__fp_step:NfnnnN
13422             #1 { \__fp_parse:n { #2 + #4 } } {#2} {#4} {#5} #6
13423         }
13424     }
13425 }
13426 \cs_generate_variant:Nn \__fp_step:NnnnnN { Nf }

```

(End definition for `\fp_step_function:nnnN`, `__fp_step:wwwN`, and `__fp_step:NnnnnN`. These functions are documented on page 187.)

\fp_step_inline:nnnn As for `\int_step_inline:nnnn`, create a global function and apply it, following up with a break point.

```

13427 \cs_new_protected:Npn \fp_step_inline:nnnn #1#2#3#4
13428 {
13429     \int_gincr:N \g__prg_map_int
13430     \cs_gset_protected:cpn { __prg_map_ \int_use:N \g__prg_map_int :w }
13431     ##1 {#4}
13432     \fp_step_function:nnnc {#1} {#2} {#3}
13433     { __prg_map_ \int_use:N \g__prg_map_int :w }
13434     \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
13435 }

```

(End definition for `\fp_step_inline:nnnn`. This function is documented on page 187.)

```

13436 \__msg_kernel_new:nnn { kernel } { fp-bad-step }
13437 { Invalid~step~size~#2~in~step~function~#3. }
13438 \__msg_kernel_new:nnn { kernel } { fp-tiny-step }
13439 { Tiny~step~size~(#1+#2=#1)~in~step~function~#3. }

```

26.5 Extrema

__fp_minmax_o:Nw 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 ± 0), the first is kept. We append $-\infty$ (∞), for the case of an empty array, currently impossible. Since no number is smaller (larger) than that, it will never alter the maximum (minimum). The weird fp-like trailing marker breaks the loop correctly: see the precise definition of `__fp_minmax_loop:Nww`.

```

13440 \cs_new:Npn \__fp_minmax_o:Nw #1#2 @
13441 {
13442     \if_meaning:w 0 #1
13443     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN +
13444     \else:
13445     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN -
13446     \fi:
13447     #2
13448     \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
13449     \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
13450 }

```

(End definition for _fp_minmax_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.

```

13451 \cs_new:Npn \_fp_minmax_loop:Nww
13452   #1 \s__fp \_fp_chk:w #2#3; \s__fp \_fp_chk:w #4#5;
13453   {
13454     \if_meaning:w 3 #4
13455     \if_meaning:w 3 #2
13456       \_fp_minmax_auxi:ww
13457     \else:
13458       \_fp_minmax_auxii:ww
13459     \fi:
13460   \else:
13461     \if_int_compare:w
13462       \_fp_compare_back:ww
13463       \s__fp \_fp_chk:w #4#5;
13464       \s__fp \_fp_chk:w #2#3;
13465       = #1 \c_one
13466       \_fp_minmax_auxii:ww
13467     \else:
13468       \_fp_minmax_auxi:ww
13469     \fi:
13470   \fi:
13471   \_fp_minmax_loop:Nww #1
13472   \s__fp \_fp_chk:w #2#3;
13473   \s__fp \_fp_chk:w #4#5;
13474   }

```

(End definition for _fp_minmax_loop:Nww.)

_fp_minmax_auxi:ww Keep the first/second number, and remove the other.

```

13475 \cs_new:Npn \_fp_minmax_auxi:ww #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
13476   { \fi: \fi: #2 \s__fp #3 ; }
13477 \cs_new:Npn \_fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
13478   { \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.

```

13479 \cs_new:Npn \_fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
13480   { \fi: \_fp_exp_after_o:w \s__fp #3; }

```

(End definition for _fp_minmax_break_o:w.)

26.6 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.

```

13481 \cs_new:cpn { __fp_not_o:w } #1 \s__fp \__fp_chk:w #2#3; @
13482 {
13483     \if_meaning:w 0 #2
13484     \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
13485     \else:
13486     \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
13487     \fi:
13488 }
```

(End definition for `__fp_not_o:w`.)

`__fp_&_o:ww` For and, if the first number is zero, return it (with the same sign). Otherwise, return
`__fp_|_o:ww` the second one. For or, the logic is reversed: if the first number is non-zero, return
`__fp_and_return:wNw` 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.

```

13489 \group_begin:
13490     \char_set_catcode_letter:N &
13491     \char_set_catcode_letter:N |
13492     \cs_new:Npn \__fp_&_o:ww #1 \s__fp \__fp_chk:w #2#3;
13493     {
13494         \if_meaning:w 0 #2 #1
13495         \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
13496         \fi:
13497         \__fp_exp_after_o:w
13498     }
13499     \cs_new:Npn \__fp_|_o:ww { \__fp_&_o:ww \else: }
13500 \group_end:
13501 \cs_new:Npn \__fp_and_return:wNw #1; \fi: #2#3; { \fi: #2 #1; }
```

(End definition for `__fp_&_o:ww`, `__fp_|_o:ww`, and `__fp_and_return:wNw`.)

26.7 Ternary operator

`__fp_ternary:NwwN` The first function receives the test and the true branch of the `?:` ternary operator. It
`__fp_ternary_auxi:NwwN` returns the true branch, unless the test branch is zero. In that case, the function returns
`__fp_ternary_auxii:NwwN` a very specific nan. The second function receives the output of the first function, and the
`__fp_ternary_loop_break:w` false branch. It returns the previous input, unless that is the special nan, in which case
`__fp_ternary_loop:Nw` we return the false branch.

```

13502 \cs_new:Npn \__fp_ternary:NwwN #1 #2@ #3@ #4
13503 {
13504     \if_meaning:w \__fp_parse_infix_::N #4
13505     \__fp_ternary_loop:Nw
13506     #2
13507     \s__fp \__fp_chk:w { \__fp_ternary_loop_break:w } ;
13508     \__fp_ternary_break_point:n { \exp_after:wN \__fp_ternary_auxi:NwwN }
13509     \exp_after:wN #1
13510     \exp:w \exp_end_continue_f:w
13511     \__fp_exp_after_array_f:w #3 \s__fp_stop
```

```

13512     \exp_after:wN @
13513     \exp:w
13514     \__fp_parse_operand:Nw \c__fp_prec_colon_int
13515     \__fp_parse_expand:w
13516   \else:
13517     \__msg_kernel_expandable_error:nnnn
13518     { kernel } { fp-missing } { : } { ~for~?: }
13519     \exp_after:wN \__fp_parse_continue:NwN
13520     \exp_after:wN #1
13521     \exp:w \exp_end_continue_f:w
13522     \__fp_exp_after_array_f:w #3 \s__fp_stop
13523     \exp_after:wN #4
13524     \exp_after:wN #1
13525   \fi:
13526 }
13527 \cs_new:Npn \__fp_ternary_loop_break:w
13528   #1 \fi: #2 \__fp_ternary_break_point:n #3
13529 {
13530   \c_zero = \c_zero \fi:
13531   \exp_after:wN \__fp_ternary_auxii:NwwN
13532 }
13533 \cs_new:Npn \__fp_ternary_loop:Nw \s__fp \__fp_chk:w #1#2;
13534 {
13535   \if_int_compare:w #1 > \c_zero
13536     \exp_after:wN \__fp_ternary_map_break:
13537     \fi:
13538   \__fp_ternary_loop:Nw
13539 }
13540 \cs_new:Npn \__fp_ternary_map_break: #1 \__fp_ternary_break_point:n #2 {#2}
13541 \cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
13542 {
13543   \exp_after:wN \__fp_parse_continue:NwN
13544   \exp_after:wN #1
13545   \exp:w \exp_end_continue_f:w
13546   \__fp_exp_after_array_f:w #2 \s__fp_stop
13547   #4 #1
13548 }
13549 \cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
13550 {
13551   \exp_after:wN \__fp_parse_continue:NwN
13552   \exp_after:wN #1
13553   \exp:w \exp_end_continue_f:w
13554   \__fp_exp_after_array_f:w #3 \s__fp_stop
13555   #4 #1
13556 }

```

(End definition for __fp_ternary:NwwN and others.)

```

13557 </initex | package>

```

27 l3fp-basics Implementation

```

13558 <*initex | package>
13559 <@@=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>.

27.1 Common to several operations

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}`.

```

13560 \cs_new:Npn \__fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6;
13561   { + #1 - \c_one ; {#2#3#4#5} {#6} ; }
13562 \cs_new:Npn \__fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6;
13563   {
13564     \if_meaning:w 2 #1
13565       \__fp_basics_pack_high_carry:w
13566     \fi:
13567     ; {#2#3#4#5} {#6}
13568   }
13569 \cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
13570   { \fi: + \c_one ; {1000} }

```

(End definition for `__fp_basics_pack_low:NNNNNw`, `__fp_basics_pack_high:NNNNNw`, and `__fp_basics_pack_high_carry:w`.)

I don't fully understand those functions, used for additions and divisions. Hence the name.

```

13571 \cs_new:Npn \__fp_basics_pack_weird_low:NNNNNw #1 #2#3#4 #5;
13572   {
13573     \if_meaning:w 2 #1
13574       + \c_one
13575     \fi:
13576     \__int_eval_end:
13577     #2#3#4; {#5} ;
13578   }
13579 \cs_new:Npn \__fp_basics_pack_weird_high:NNNNNNNNw
13580   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:NNNNNw` and `__fp_basics_pack_weird_high:NNNNNNNNw`.)

27.2 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.

27.2.1 Sign, exponent, and special numbers

`__fp_-_o:ww` A previous version of this function grabbed its two operands, changed the sign of the second, and called `__fp+_o:ww`. However, for efficiency reasons, the operands were swapped in the process, which means that error messages ended up wrong. Now, 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 check that it was followed by `\s__fp` and not arbitrary junk.

```

13581 \cs_new:cpx { __fp_-_o:ww } \s__fp
13582 {
13583   \exp_not:c { __fp+_o:ww }
13584   \exp_not:n { \s__fp \__fp_neg_sign:N }
13585 }
```

(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 (equivalent to changing the $\langle sign_2 \rangle$ of the second operand). If the $\langle types \rangle$ `#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 $\langle sign_2 \rangle$ (expansion of `#1#5`) as an argument. If the $\langle types \rangle$ are distinct, the result is simply the floating point number with the highest $\langle type \rangle$. Since case 3 (used for two `nan`) also picks the first operand, we can also use it when $\langle type_1 \rangle$ is greater than $\langle type_2 \rangle$. Also note that we don't need to worry about $\langle sign_2 \rangle$ in that case since the second operand is discarded.

```

13586 \cs_new:cpn { __fp+_o:ww }
13587   \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
13588 {
13589   \if_case:w
13590     \if_meaning:w #2 #4
13591       #2 \exp_stop_f:
13592   \else:
```



```

13593         \if_int_compare:w #2 > #4 \exp_stop_f:
13594         \c_three
13595     \else:
13596         \c_four
13597     \fi:
13598 \fi:
13599     \exp_after:wN \__fp_add_zeros_o:Nww \__int_value:w
13600 \or:   \exp_after:wN \__fp_add_normal_o:Nww \__int_value:w
13601 \or:   \exp_after:wN \__fp_add_inf_o:Nww \__int_value:w
13602 \or:   \__fp_case_return_i_o:ww
13603 \else: \exp_after:wN \__fp_add_return_ii_o:Nww \__int_value:w
13604 \fi:
13605 #1 #5
13606 \s__fp \__fp_chk:w #2 #3 ;
13607 \s__fp \__fp_chk:w #4 #5
13608 }

```

(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.

```

13609 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
13610 { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }

```

(End definition for __fp_add_return_ii_o:Nww.)

__fp_add_zeros_o:Nww Adding two zeros yields \c_zero_fp, except if both zeros were -0 .

```

13611 \cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
13612 {
13613     \if_int_compare:w #2 #1 = 20 \exp_stop_f:
13614     \exp_after:wN \__fp_add_return_ii_o:Nww
13615 \else:
13616     \__fp_case_return_i_o:ww
13617 \fi:
13618 #1
13619 \s__fp \__fp_chk:w 0 #2
13620 }

```

(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.

```

13621 \cs_new:Npn \__fp_add_inf_o:Nww
13622 #1 \s__fp \__fp_chk:w 2 #2 #3; \s__fp \__fp_chk:w 2 #4
13623 {
13624     \if_meaning:w #1 #2
13625     \__fp_case_return_i_o:ww
13626 \else:
13627     \__fp_case_use:nw
13628     {
13629         \if_meaning:w #1 #4
13630         \exp_after:wN \__fp_invalid_operation_o:Nww
13631         \exp_after:wN +

```

```

13632         \else:
13633             \exp_after:wN \__fp_invalid_operation_o:Nww
13634             \exp_after:wN -
13635         \fi:
13636     }
13637 \fi:
13638 \s__fp \__fp_chk:w 2 #2 #3;
13639 \s__fp \__fp_chk:w 2 #4
13640 }

```

(End definition for __fp_add_inf_o:Nww.)

```

\__fp_add_normal_o:Nww \__fp_add_normal_o:Nww <sign2> \s__fp \__fp_chk:w 1 <sign1> <exp1>
<body1> ; \s__fp \__fp_chk:w 1 <initial sign2> <exp2> <body2> ;

```

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```

13641 \cs_new:Npn \__fp_add_normal_o:Nww #1 \s__fp \__fp_chk:w 1 #2
13642 {
13643     \if_meaning:w #1#2
13644         \exp_after:wN \__fp_add_npos_o:NnwNnw
13645     \else:
13646         \exp_after:wN \__fp_sub_npos_o:NnwNnw
13647     \fi:
13648     #2
13649 }

```

(End definition for __fp_add_normal_o:Nww.)

27.2.2 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 `__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.

```

13650 \cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \__fp_chk:w 1 #4 #5
13651 {
13652     \exp_after:wN \__fp_sanitize:Nw
13653     \exp_after:wN #1
13654     \__int_value:w \__int_eval:w
13655     \if_int_compare:w #2 > #5 \exp_stop_f:
13656         #2
13657         \exp_after:wN \__fp_add_big_i_o:wNww \__int_value:w -
13658     \else:
13659         #5
13660         \exp_after:wN \__fp_add_big_ii_o:wNww \__int_value:w
13661     \fi:

```

```

13662     \__int_eval:w #5 - #2 ; #1 #3;
13663 }

```

(End definition for __fp_add_npos_o:NnwNnw.)

```

\__fp_add_big_i_o:wNww     \__fp_add_big_i_o:wNww <shift> ; <final sign> <body1> ; <body2> ;
\__fp_add_big_ii_o:wNww    Shift the significand of the small number, then add with \__fp_add_significand_
o:NnnwnnnnN.

```

```

13664 \cs_new:Npn \__fp_add_big_i_o:wNww #1; #2 #3; #4;
13665 {
13666     \__fp_decimate:nNnnnn {#1}
13667     \__fp_add_significand_o:NnnwnnnnN
13668     #4
13669     #3
13670     #2
13671 }
13672 \cs_new:Npn \__fp_add_big_ii_o:wNww #1; #2 #3; #4;
13673 {
13674     \__fp_decimate:nNnnnn {#1}
13675     \__fp_add_significand_o:NnnwnnnnN
13676     #3
13677     #4
13678     #2
13679 }

```

(End definition for __fp_add_big_i_o:wNww and __fp_add_big_ii_o:wNww.)

```

\__fp_add_significand_o:NnnwnnnnN     \__fp_add_significand_o:NnnwnnnnN <rounding digit> {\langle Y'_1 \rangle} {\langle Y'_2 \rangle}
\__fp_add_significand_pack:NNNNNNN    <extra-digits> ; {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} <final sign>
\__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.

```

13680 \cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
13681 {
13682     \exp_after:wN \__fp_add_significand_test_o:N
13683     \__int_value:w \__int_eval:w 1#5#6 + #2
13684     \exp_after:wN \__fp_add_significand_pack:NNNNNNN
13685     \__int_value:w \__int_eval:w 1#7#8 + #3 ; #1
13686 }
13687 \cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
13688 {
13689     \if_meaning:w 2 #1
13690     + \c_one
13691     \fi:
13692     ; #2 #3 #4 #5 #6 #7 ;
13693 }
13694 \cs_new:Npn \__fp_add_significand_test_o:N #1
13695 {
13696     \if_meaning:w 2 #1
13697     \exp_after:wN \__fp_add_significand_carry_o:wwwNN
13698     \else:
13699     \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN

```

```

13700     \fi:
13701 }

```

(End definition for `__fp_add_significand_o:NnnwnnnN`, `__fp_add_significand_pack:NNNNNN`, 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.

```

13702 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
13703     #1; #2; #3#4 ; #5#6
13704 {
13705     \exp_after:wN \__fp_basics_pack_high:NNNNNw
13706     \__int_value:w \__int_eval:w 1 #1
13707     \exp_after:wN \__fp_basics_pack_low:NNNNNw
13708     \__int_value:w \__int_eval:w 1 #2 #3#4
13709     + \__fp_round:NNN #6 #4 #5
13710     \exp_after:wN ;
13711 }

```

(End definition for `__fp_add_significand_no_carry_o:wwwNN`.)

```

\__fp_add_significand_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.

```

13712 \cs_new:Npn \__fp_add_significand_carry_o:wwwNN
13713     #1; #2; #3#4; #5#6
13714 {
13715     + \c_one
13716     \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
13717     \__int_value:w \__int_eval:w 1 1 #1
13718     \exp_after:wN \__fp_basics_pack_weird_low:NNNNNw
13719     \__int_value:w \__int_eval:w 1 #2#3 +
13720     \exp_after:wN \__fp_round:NNN
13721     \exp_after:wN #6
13722     \exp_after:wN #3
13723     \__int_value:w \__fp_round_digit:Nw #4 #5 ;
13724     \exp_after:wN ;
13725 }

```

(End definition for `__fp_add_significand_carry_o:wwwNN`.)

27.2.3 Absolute subtraction

```

\__fp_sub_npos_o:NnwNnw    \__fp_sub_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s_fp \__fp_chk:w 1
\__fp_sub_eq_o:Nnwnw    <initial sign2> <exp2> <body2> ;
\__fp_sub_npos_ii_o:Nnwnw

```

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:Nnwnw` with the opposite of $\langle sign_1 \rangle$.

```

13726 \cs_new:Npn \__fp_sub_npos_o:NnwNnw #1#2#3; \s_fp \__fp_chk:w 1 #4#5#6;
13727 {

```

```

13728 \if_case:w \_fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
13729 \exp_after:wN \_fp_sub_eq_o:Nnwnw
13730 \or:
13731 \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
13732 \else:
13733 \exp_after:wN \_fp_sub_npos_ii_o:Nnwnw
13734 \fi:
13735 #1 {#2} #3; {#5} #6;
13736 }
13737 \cs_new:Npn \_fp_sub_eq_o:Nnwnw #1#2; #3; { \exp_after:wN \c_zero_fp }
13738 \cs_new:Npn \_fp_sub_npos_ii_o:Nnwnw #1 #2; #3;
13739 {
13740 \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
13741 \_int_value:w \_int_eval:w \c_two - #1 \_int_eval_end:
13742 #3; #2;
13743 }

```

(End definition for _fp_sub_npos_o:NnwNnw, _fp_sub_eq_o:Nnwnw, and _fp_sub_npos_ii_o:Nnwnw.)

_fp_sub_npos_i_o:Nnwnw

After the computation is done, _fp_sanitize:Nw checks for overflow/underflow. It expects the $\langle final\ sign \rangle$ and the $\langle exponent \rangle$ (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.

```

13744 \cs_new:Npn \_fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
13745 {
13746 \exp_after:wN \_fp_sanitize:Nw
13747 \exp_after:wN #1
13748 \_int_value:w \_int_eval:w
13749 #2
13750 \if_int_compare:w #2 = #4 \exp_stop_f:
13751 \exp_after:wN \_fp_sub_back_near_o:nnnnnnnnN
13752 \else:
13753 \exp_after:wN \_fp_decimate:nNnnnn \exp_after:wN
13754 { \_int_value:w \_int_eval:w #2 - #4 - \c_one \exp_after:wN }
13755 \exp_after:wN \_fp_sub_back_far_o:NnnwnnnnnN
13756 \fi:
13757 #5
13758 #3
13759 #1
13760 }

```

(End definition for _fp_sub_npos_i_o:Nnwnw.)

_fp_sub_back_near_o:nnnnnnnnN
_fp_sub_back_near_pack:NNNNNNw
_fp_sub_back_near_after:wNNNNw

_fp_sub_back_near_o:nnnnnnnnN { $\langle Y_1 \rangle$ } { $\langle Y_2 \rangle$ } { $\langle Y_3 \rangle$ } { $\langle Y_4 \rangle$ } { $\langle X_1 \rangle$ }
{ $\langle X_2 \rangle$ } { $\langle X_3 \rangle$ } { $\langle X_4 \rangle$ } $\langle final\ sign \rangle$

In this case, the subtraction is exact, so we discard the $\langle final\ sign \rangle$ #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.

```

13761 \cs_new:Npn \_fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
13762 {

```

```

13763 \exp_after:wN \_fp_sub_back_near_after:wNNNNw
13764 \_int_value:w \_int_eval:w 10#5#6 - #1#2 - \c_eleven
13765 \exp_after:wN \_fp_sub_back_near_pack:NNNNNNw
13766 \_int_value:w \_int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
13767 }
13768 \cs_new:Npn \_fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
13769 { + #1#2 ; {#3#4#5#6} {#7} ; }
13770 \cs_new:Npn \_fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
13771 {
13772 \if_meaning:w 0 #1
13773 \exp_after:wN \_fp_sub_back_shift:wnnnn
13774 \fi:
13775 ; {#1#2#3#4} {#5}
13776 }

```

(End definition for `_fp_sub_back_near_o:nnnnnnnnN`, `_fp_sub_back_near_pack:NNNNNNw`, and `_fp_sub_back_near_after:wNNNNw`.)

```

\_fp_sub_back_shift:wnnnn
\_fp_sub_back_shift_ii:ww
\_fp_sub_back_shift_iii:NNNNNNNNw
\_fp_sub_back_shift_iv:nnnnw

```

```

\_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 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.

```

13777 \cs_new:Npn \_fp_sub_back_shift:wnnnn ; #1#2
13778 {
13779 \exp_after:wN \_fp_sub_back_shift_ii:ww
13780 \_int_value:w #1 #2 0 ;
13781 }
13782 \cs_new:Npn \_fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
13783 {
13784 \if_meaning:w @ #1 @
13785 - \c_seven
13786 - \exp_after:wN \use_i:nnn
13787 \exp_after:wN \_fp_sub_back_shift_iii:NNNNNNNNw
13788 \_int_value:w #2#3 0 ~ 123456789;
13789 \else:
13790 - \_fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
13791 \fi:
13792 \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
13793 \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
13794 \exp_after:wN \_fp_sub_back_shift_iv:nnnnw
13795 \exp_after:wN ;
13796 \_int_value:w
13797 #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
13798 }
13799 \cs_new:Npn \_fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
13800 \cs_new:Npn \_fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }

```

(End definition for `_fp_sub_back_shift:wnnnn` and others.)

_fp_sub_back_far_o:NnnwnnnnN

_fp_sub_back_far_o:NnnwnnnnN $\langle \text{rounding} \rangle$ $\{ \langle Y'_1 \rangle \} \{ \langle Y'_2 \rangle \}$
 $\langle \text{extra-digits} \rangle$; $\{ \langle X_1 \rangle \} \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \}$ $\langle \text{final sign} \rangle$

If the difference is greater than $10^{\langle \text{expo}_x \rangle}$, call the **very_far** auxiliary. If the result is less than $10^{\langle \text{expo}_x \rangle}$, 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).

```

13801 \cs_new:Npn \_fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
13802 {
13803   \if_case:w
13804     \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
13805     \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
13806       \c_zero
13807     \else:
13808       \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: \c_one
13809     \fi:
13810   \else:
13811     \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: \c_one
13812   \fi:
13813     \exp_after:wN \_fp_sub_back_quite_far_o:wwNN
13814   \or:   \exp_after:wN \_fp_sub_back_very_far_o:wwwNN
13815   \else: \exp_after:wN \_fp_sub_back_not_far_o:wwwNN
13816   \fi:
13817   #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
13818 }

```

(End definition for _fp_sub_back_far_o:NnnwnnnnN.)

_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 $\langle \text{rounding} \rangle$ #3 and the $\langle \text{final sign} \rangle$ #4 control whether we get 1 or 0.9999999999999999. In the usual round-to-nearest mode, we will get 1 whenever the $\langle \text{rounding} \rangle$ digit is less than or equal to 5 (remember that the $\langle \text{rounding} \rangle$ digit is only equal to 5 if there was no further non-zero digit).

```

13819 \cs_new:Npn \_fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
13820 {
13821   \exp_after:wN \_fp_sub_back_quite_far_ii:NN
13822   \exp_after:wN #3
13823   \exp_after:wN #4
13824 }
13825 \cs_new:Npn \_fp_sub_back_quite_far_ii:NN #1#2
13826 {
13827   \if_case:w \_fp_round_neg:NNN #2 0 #1
13828     \exp_after:wN \use_i:nn
13829   \else:
13830     \exp_after:wN \use_ii:nn
13831   \fi:
13832   { ; {1000} {0000} {0000} {0000} ; }
13833   { - \c_one ; {9999} {9999} {9999} {9999} ; }
13834 }

```

(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 $- \backslash c_one$). 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.

```
13835 \cs_new:Npn \_fp_sub_back_not_far_o:wwwNN #1 ~ #2; #3 ~ #4; #5#6
13836 {
13837   - \c_one
13838   \exp_after:wN \_fp_sub_back_near_after:wNNNNw
13839   \_int_value:w \_int_eval:w 1#30 - #1 - \c_eleven
13840   \exp_after:wN \_fp_sub_back_near_pack:NNNNNNw
13841   \_int_value:w \_int_eval:w 11 0000 0000 + #40 - #2
13842   - \exp_after:wN \_fp_round_neg:NNN
13843   \exp_after:wN #6
13844   \use_none:nnnnnnn #2 #5
13845   \exp_after:wN ;
13846 }
```

(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.

```
13847 \cs_new:Npn \_fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
13848 {
13849   \_fp_pack_eight:wNNNNNNNN
13850   \_fp_sub_back_very_far_ii_o:nnNwwNN
13851   { 0 #1#2#3 #4#5#6#7 }
13852   ;
13853 }
13854 \cs_new:Npn \_fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
13855 {
13856   \exp_after:wN \_fp_basics_pack_high:NNNNNw
13857   \_int_value:w \_int_eval:w 1#4 - #1 - \c_one
13858   \exp_after:wN \_fp_basics_pack_low:NNNNNw
13859   \_int_value:w \_int_eval:w 2#5 - #2
13860   - \exp_after:wN \_fp_round_neg:NNN
13861   \exp_after:wN #7
13862   \_int_value:w
13863   \if_int_odd:w \_int_eval:w #5 - #2 \_int_eval_end:
13864     1 \else: 2 \fi:
13865   \_int_value:w \_fp_round_digit:Nw #3 #6 ;
13866   \exp_after:wN ;
13867 }
```

(End definition for _fp_sub_back_very_far_o:wwwNN and _fp_sub_back_very_far_ii_o:nnNwwNN.)

27.3 Multiplication

27.3.1 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`.

```

13868 \cs_new:cpn { __fp*_o:ww }
13869 {
13870     \__fp_mul_cases_o:NnNnw
13871     *
13872     { - \c_two + }
13873     \__fp_mul_npos_o:Nww
13874     { }
13875 }
```

(End definition for `__fp*_o:ww`.)

`__fp_mul_cases_o:NnNnw` 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 $0 \times \infty$ 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.

```

13876 \cs_new:Npn \__fp_mul_cases_o:NnNnw
13877 #1#2#3#4 \s__fp \__fp_chk:w #5#6#7; \s__fp \__fp_chk:w #8#9
13878 {
13879     \if_case:w \__int_eval:w
13880         \if_int_compare:w #5 #8 = \c_eleven
13881         \c_one
13882     \else:
13883         \if_meaning:w 3 #8
13884         \c_three
13885     \else:
13886         \if_meaning:w 3 #5
13887         \c_two
13888     \else:
13889         \if_int_compare:w #5 #8 = \c_ten
13890         \c_nine #2 - \c_two
13891     \else:
13892         (#5 #2 #8) / \c_two * \c_two + \c_seven
13893     \fi:
13894     \fi:
13895     \fi:
13896     \fi:
13897     \if_meaning:w #6 #9 - \c_one \fi:
```

```

13898         \__int_eval_end:
13899         \__fp_case_use:nw { #3 0 }
13900     \or: \__fp_case_use:nw { #3 2 }
13901     \or: \__fp_case_return_i_o:ww
13902     \or: \__fp_case_return_ii_o:ww
13903     \or: \__fp_case_return_o:Nww \c_zero_fp
13904     \or: \__fp_case_return_o:Nww \c_minus_zero_fp
13905     \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
13906     \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
13907     \or: \__fp_case_return_o:Nww \c_inf_fp
13908     \or: \__fp_case_return_o:Nww \c_minus_inf_fp
13909     #4
13910     \fi:
13911     \s__fp \__fp_chk:w #5 #6 #7;
13912     \s__fp \__fp_chk:w #8 #9
13913 }

```

(End definition for __fp_mul_cases_o:nNnnww.)

27.3.2 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, __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.

```

13914 \cs_new:Npn \__fp_mul_npos_o:Nww
13915     #1 \s__fp \__fp_chk:w #2 #3 #4 #5 ; \s__fp \__fp_chk:w #6 #7 #8 #9 ;
13916     {
13917     \exp_after:wN \__fp_sanitize:Nw
13918     \exp_after:wN #1
13919     \__int_value:w \__int_eval:w
13920     #4 + #8
13921     \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
13922     }

```

(End definition for __fp_mul_npos_o:Nww.)

```

\__fp_mul_significand_o:nnnnNnnnn \__fp_mul_significand_o:nnnnNnnnn {\<X1>} {\<X2>} {\<X3>} {\<X4>} <sign>
\__fp_mul_significand_drop:NNNNNw {\<Y1>} {\<Y2>} {\<Y3>} {\<Y4>}
\__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 __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 `__int_eval:w`.

```

13923 \cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
13924 {
13925   \exp_after:wN \__fp_mul_significand_test_f:NNN
13926   \exp_after:wN #5
13927   \__int_value:w \__int_eval:w 99990000 + #1*#6 +
13928   \exp_after:wN \__fp_mul_significand_keep:NNNNNw
13929   \__int_value:w \__int_eval:w 99990000 + #1*#7 + #2*#6 +
13930   \exp_after:wN \__fp_mul_significand_keep:NNNNNw
13931   \__int_value:w \__int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
13932   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
13933   \__int_value:w \__int_eval:w 99990000 + #1*#9 + #2*#8 + #3*#7 + #4*#6 +
13934   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
13935   \__int_value:w \__int_eval:w 99990000 + #2*#9 + #3*#8 + #4*#7 +
13936   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
13937   \__int_value:w \__int_eval:w 99990000 + #3*#9 + #4*#8 +
13938   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
13939   \__int_value:w \__int_eval:w 100000000 + #4*#9 ;
13940   ; \exp_after:wN ;
13941 }
13942 \cs_new:Npn \__fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
13943 { #1#2#3#4#5 ; + #6 }
13944 \cs_new:Npn \__fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
13945 { #1#2#3#4#5 ; #6 ; }

```

(End definition for `__fp_mul_significand_o:nnnnNnnnn`, `__fp_mul_significand_drop:NNNNNw`, and `__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.

```

13946 \cs_new:Npn \__fp_mul_significand_test_f:NNN #1 #2 #3
13947 {
13948   \if_meaning:w 0 #3
13949   \exp_after:wN \__fp_mul_significand_small_f:NNwwwN
13950   \else:
13951   \exp_after:wN \__fp_mul_significand_large_f:NwwNNNN
13952   \fi:
13953   #1 #3
13954 }

```

(End definition for `__fp_mul_significand_test_f:NNN`.)

`__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, `__fp_round_digit:Nw` takes digits 17 and further (as an integer expression), and replaces it by a *<rounding digit>*, suitable for `__fp_round:NNN`.

```

13955 \cs_new:Npn \__fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
13956 {
13957   \exp_after:wN \__fp_basics_pack_high:NNNNNw

```

```

13958     \__int_value:w \__int_eval:w 1#2
13959     \exp_after:wN \__fp_basics_pack_low:NNNNNw
13960     \__int_value:w \__int_eval:w 1#3#4#5#6#7
13961     + \exp_after:wN \__fp_round:NNN
13962     \exp_after:wN #1
13963     \exp_after:wN #7
13964     \__int_value:w \__fp_round_digit:Nw
13965 }

```

(End definition for __fp_mul_significand_large_f:NwwNNNN.)

__fp_mul_significand_small_f:NNwwwN In this branch, $\langle digit\ 1 \rangle$ is zero. Our result will thus be $\langle digits\ 2-17 \rangle$, 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.

```

13966 \cs_new:Npn \__fp_mul_significand_small_f:NNwwwN #1 #2#3; #4#5; #6; + #7
13967 {
13968   - \c_one
13969   \exp_after:wN \__fp_basics_pack_high:NNNNNw
13970   \__int_value:w \__int_eval:w 1#3#4
13971   \exp_after:wN \__fp_basics_pack_low:NNNNNw
13972   \__int_value:w \__int_eval:w 1#5#6#7
13973   + \exp_after:wN \__fp_round:NNN
13974   \exp_after:wN #1
13975   \exp_after:wN #7
13976   \__int_value:w \__fp_round_digit:Nw
13977 }

```

(End definition for __fp_mul_significand_small_f:NNwwwN.)

27.4 Division

27.4.1 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 `- \c_two +` 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:NnNww` are provided as the fourth argument here.

```

13978 \cs_new:cpn { __fp/_o:ww }
13979 {
13980   \__fp_mul_cases_o:NnNww
13981   /
13982   { - }
13983   \__fp_div_npos_o:Nww
13984   {
13985     \or:
13986     \__fp_case_use:nw
13987     { \__fp_division_by_zero_o:NNww \c_inf_fp / }
13988     \or:

```

```

13989         \__fp_case_use:nw
13990         { \__fp_division_by_zero_o:NNnw \c_minus_inf_fp / }
13991     }
13992 }

```

(End definition for __fp_/_o:ww.)

```

\__fp_div_npos_o:Nnw      \__fp_div_npos_o:Nnw <final sign> \s__fp \__fp_chk:w 1 <sign_A> {\exp A}\
                          {\langle A_1 \rangle} {\langle A_2 \rangle} {\langle A_3 \rangle} {\langle A_4 \rangle} ; \s__fp \__fp_chk:w 1 <sign_Z> {\exp Z}\
                          {\langle Z_1 \rangle} {\langle Z_2 \rangle} {\langle Z_3 \rangle} {\langle Z_4 \rangle} ;

```

We want to compute A/Z . As for multiplication, `__fp_sanitize:Nw` checks for overflow or underflow; we provide it with the $\langle final\ sign \rangle$, 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 $\langle y \rangle$ obtained by adding 1 to the first 5 digits of Z (explanation given soon below), then the four $\{\langle A_i \rangle\}$, then the four $\{\langle Z_i \rangle\}$, a semi-colon, and the $\langle final\ sign \rangle$, used for rounding at the end.

```

13993 \cs_new:Npn \__fp_div_npos_o:Nnw
13994     #1 \s__fp \__fp_chk:w 1 #2 #3 #4 ; \s__fp \__fp_chk:w 1 #5 #6 #7#8#9;
13995 {
13996     \exp_after:wN \__fp_sanitize:Nw
13997     \exp_after:wN #1
13998     \__int_value:w \__int_eval:w
13999     #3 - #6
14000     \exp_after:wN \__fp_div_significand_i_o:wnnw
14001     \__int_value:w \__int_eval:w #7 \use_i:nnnn #8 + \c_one ;
14002     #4
14003     {\#7}{\#8}\#9 ;
14004     #1
14005 }

```

(End definition for __fp_div_npos_o:Nnw.)

27.4.2 Work plan

In this subsection, we explain how to avoid overflowing $\text{T}_{\text{E}}\text{X}$'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 will be 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

$$\backslash\text{int_eval:n}\left\{\frac{A_1A_2}{Z_1+1}-1\right\}\leq 10^4\frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that $\varepsilon\text{-TeX}$'s $\backslash_int_eval:w$ 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

$$Q_A = \backslash\text{int_eval:n}\left\{\frac{10 \cdot A_1A_2}{\lfloor 10^{-3} \cdot Z_1Z_2 \rfloor + 1} - 1\right\}.$$

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

$$y = \lfloor 10^{-3} \cdot Z_1Z_2 \rfloor + 1,$$

so that, taking into account the fact that $\varepsilon\text{-TeX}$ rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1A_20}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1A_20}{y} - \frac{3}{2}. \end{aligned}$$

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 :

$$\begin{aligned} 10^5 B &= A_1A_20 + 10 \cdot 0.A_3A_4 - 10 \cdot Z_1.Z_2Z_3Z_4 \cdot Q_A \\ &< A_1A_20 \cdot \left(1 - 10 \cdot \frac{Z_1.Z_2Z_3Z_4}{y}\right) + \frac{3}{2} \cdot 10 \cdot Z_1.Z_2Z_3Z_4 + 10 \\ &\leq \frac{A_1A_20 \cdot (y - 10 \cdot Z_1.Z_2Z_3Z_4)}{y} + \frac{3}{2}y + 10 \\ &\leq \frac{A_1A_20 \cdot 1}{y} + \frac{3}{2}y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y. \end{aligned}$$

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 \TeX 's bounds in all cases!

We will 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 will be 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 $\varepsilon\text{-TeX}$ 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 $\varepsilon\text{-TeX}$'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.

27.4.3 Implementing the significand division

`_fp_div_significand_i_o:wnnw`

`_fp_div_significand_i_o:wnnw` $\langle y \rangle$; $\{\langle A_1 \rangle\}$ $\{\langle A_2 \rangle\}$ $\{\langle A_3 \rangle\}$ $\{\langle A_4 \rangle\}$
 $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$; $\langle sign \rangle$

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 $\langle continuation \rangle$ arguments for 4 consecutive calls to `_fp_div_significand_calc:wnnnnnnnn`. Each of these calls will need $\langle y \rangle$ (#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.

```

14006 \cs_new:Npn \_fp\_div\_significand\_i\_o:wnnw #1 ; #2#3 #4 ;
14007 {
14008   \exp\_after:wN \_fp\_div\_significand\_test\_o:w
14009   \_int\_value:w \_int\_eval:w
14010   \exp\_after:wN \_fp\_div\_significand\_calc:wnnnnnnnn
14011   \_int\_value:w \_int\_eval:w 999999 + #2 #3 0 / #1 ;
14012   #2 #3 ;
14013   #4
14014   { \exp\_after:wN \_fp\_div\_significand\_ii:wN \_int\_value:w #1 }
14015   { \exp\_after:wN \_fp\_div\_significand\_ii:wN \_int\_value:w #1 }
14016   { \exp\_after:wN \_fp\_div\_significand\_ii:wN \_int\_value:w #1 }
14017   { \exp\_after:wN \_fp\_div\_significand\_iii:wnnnnnn \_int\_value:w #1 }
14018 }

```

(End definition for `_fp_div_significand_i_o:wnnw`.)

`_fp_div_significand_calc:wnnnnnnnn`
`_fp_div_significand_calc_i:wnnnnnnnn`
`_fp_div_significand_calc_ii:wnnnnnnnn`

`_fp_div_significand_calc:wnnnnnnnn` $\langle 10^6 + Q_A \rangle$; $\langle A_1 \rangle$ $\langle A_2 \rangle$; $\{\langle A_3 \rangle\}$
 $\{\langle A_4 \rangle\}$ $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$ $\{\langle continuation \rangle\}$

expands to

$\langle 10^6 + Q_A \rangle$ $\langle continuation \rangle$; $\langle B_1 \rangle$ $\langle B_2 \rangle$; $\{\langle B_3 \rangle\}$ $\{\langle B_4 \rangle\}$ $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$
 $\{\langle Z_4 \rangle\}$

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 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 *continuation*, this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

$$10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7) \\ + 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),$$

where $\langle i \rangle$ stands for the 10^5 digit of Q_A , which is 0 or 1, and $\#1$, $\#2$, *etc.* 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 TeX's limits once more.

```

14019 \cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn #1#1
14020 {
14021   \if_meaning:w 1 #1
14022     \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
14023   \else:
14024     \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnn
14025   \fi:
14026 }
14027 \cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
14028 {
14029   1 1 #1
14030   #9 \exp_after:wN ;
14031   \__int_value:w \__int_eval:w \c__fp_Bigg_leading_shift_int
14032   + #2 - #1 * #5 - #5#60
14033   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
14034   \__int_value:w \__int_eval:w \c__fp_Bigg_middle_shift_int
14035   + #3 - #1 * #6 - #70
14036   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
14037   \__int_value:w \__int_eval:w \c__fp_Bigg_middle_shift_int
14038   + #4 - #1 * #7 - #80
14039   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
14040   \__int_value:w \__int_eval:w \c__fp_Bigg_trailing_shift_int
14041   - #1 * #8 ;
14042   {#5}{#6}{#7}{#8}
14043 }
14044 \cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
14045 {
14046   1 0 #1
14047   #9 \exp_after:wN ;
14048   \__int_value:w \__int_eval:w \c__fp_Bigg_leading_shift_int

```

```

14049     + #2 - #1 * #5
14050     \exp_after:wN \__fp_pack_Bigg:NNNNNNw
14051     \__int_value:w \__int_eval:w \c__fp_Bigg_middle_shift_int
14052     + #3 - #1 * #6
14053     \exp_after:wN \__fp_pack_Bigg:NNNNNNw
14054     \__int_value:w \__int_eval:w \c__fp_Bigg_middle_shift_int
14055     + #4 - #1 * #7
14056     \exp_after:wN \__fp_pack_Bigg:NNNNNNw
14057     \__int_value:w \__int_eval:w \c__fp_Bigg_trailing_shift_int
14058     - #1 * #8 ;
14059     {#5}{#6}{#7}{#8}
14060 }

```

(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:wnn

```

\__fp_div_significand_ii:wnn <y> ; <B1> ; {<B2>} {<B3>} {<B4>} {<Z1>}
{<Z2>} {<Z3>} {<Z4>} <continuations> <sign>

```

Compute Q_B by evaluating $\langle B_1 \rangle \langle B_2 \rangle 0 / y - 1$. The result will be output to the left, in an `__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 .

```

14061 \cs_new:Npn \__fp_div_significand_ii:wnn #1; #2;#3
14062 {
14063   \exp_after:wN \__fp_div_significand_pack:NNN
14064   \__int_value:w \__int_eval:w
14065   \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
14066   \__int_value:w \__int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
14067 }

```

(End definition for __fp_div_significand_ii:wnn.)

__fp_div_significand_iii:wwnnnnn

```

\__fp_div_significand_iii:wwnnnnn <y> ; <E1> ; {<E2>} {<E3>} {<E4>}
{<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>

```

We compute $P \simeq 2E/Z$ by rounding $2E_1E_2/Z_1Z_2$. Note the first 0, which multiplies Q_D by 10: we will 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 .

```

14068 \cs_new:Npn \__fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
14069 {
14070   0
14071   \exp_after:wN \__fp_div_significand_iv:wwnnnnnnn
14072   \__int_value:w \__int_eval:w (\c_two * #2 #3) / #6 #7 ; % <- P
14073   #2 ; {#3} {#4} {#5}
14074   {#6} {#7}
14075 }

```

(End definition for __fp_div_significand_iii:wwnnnnn.)

__fp_div_significand_iv:wwnnnnnnn

```

\__fp_div_significand_iv:wwnnnnnnn <P> ; <E1> ; {<E2>} {<E3>} {<E4>}
{<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>

```

__fp_div_significand_v:NNw

__fp_div_significand_vi:Nw

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: $T = 0$.

```

14076 \cs_new:Npn \__fp_div_significand_iv:wwnnnnnnn #1; #2;#3#4#5 #6#7#8#9
14077 {
14078   + \c_five * #1
14079   \exp_after:wN \__fp_div_significand_vi:Nw
14080   \__int_value:w \__int_eval:w -20 + 2*#2#3 - #1*#6#7 +
14081   \exp_after:wN \__fp_div_significand_v:NN
14082   \__int_value:w \__int_eval:w 199980 + 2*#4 - #1*#8 +
14083   \exp_after:wN \__fp_div_significand_v:NN
14084   \__int_value:w \__int_eval:w 200000 + 2*#5 - #1*#9 ;
14085 }
14086 \cs_new:Npn \__fp_div_significand_v:NN #1#2 { #1#2 \__int_eval_end: + }
14087 \cs_new:Npn \__fp_div_significand_vi:Nw #1#2;
14088 {
14089   \if_meaning:w 0 #1
14090     \if_int_compare:w \__int_eval:w #2 > \c_zero + \c_one \fi:
14091   \else:
14092     \if_meaning:w - #1 - \else: + \fi: \c_one
14093   \fi:
14094   ;
14095 }
```

(End definition for `__fp_div_significand_iv:wwnnnnnnn`, `__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 106 + QA \__fp_div_significand_
pack:NNN 106 + QB \__fp_div_significand_pack:NNN 106 + QC \__fp_
div_significand_pack:NNN 107 + 10 · QD + 5 · P + ε ; <sign>
```

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).

```
14096 \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 \__fp_div_significand_test_o:w 1 0 <5d> ; <4d> ; <4d> ; <5d> ; <sign>
```

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.

```
14097 \cs_new:Npn \__fp_div_significand_test_o:w 10 #1
14098 {
14099   \if_meaning:w 0 #1
14100     \exp_after:wN \__fp_div_significand_small_o:wwwNNNNwN
14101   \else:
14102     \exp_after:wN \__fp_div_significand_large_o:wwwNNNNwN
14103   \fi:
14104   #1
14105 }
```

(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:NNNNw` and `__fp_basics_pack_high:NNNNw`. We finally get to use the `<final sign>` which has been sitting there for a while.

```
14106 \cs_new:Npn \__fp_div_significand_small_o:wwwNNNNwN
14107   0 #1; #2; #3; #4#5#6#7#8; #9
14108 {
14109   \exp_after:wN \__fp_basics_pack_high:NNNNw
14110   \__int_value:w \__int_eval:w 1 #1#2
14111   \exp_after:wN \__fp_basics_pack_low:NNNNw
14112   \__int_value:w \__int_eval:w 1 #3#4#5#6#7
14113   + \__fp_round:NNN #9 #7 #8
14114   \exp_after:wN ;
14115 }
```

(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.

```
14116 \cs_new:Npn \__fp_div_significand_large_o:wwwNNNNwN
14117   #1; #2; #3; #4#5#6#7#8; #9
14118 {
14119   + \c_one
14120   \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNwN
```

```

14121     \__int_value:w \__int_eval:w 1 #1 #2
14122     \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
14123     \__int_value:w \__int_eval:w 1 #3 #4 #5 #6 +
14124     \exp_after:wN \__fp_round:NNN
14125     \exp_after:wN #9
14126     \exp_after:wN #6
14127     \__int_value:w \__fp_round_digit:Nw #7 #8 ;
14128     \exp_after:wN ;
14129 }

```

(End definition for __fp_div_significand_large_o:wwwNNNNwN.)

27.5 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.

```

14130 \cs_new:Npn \__fp_sqrt_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14131 {
14132     \if_meaning:w 0 #2 \__fp_case_return_same_o:w \fi:
14133     \if_meaning:w 2 #3
14134         \__fp_case_use:nw { \__fp_invalid_operation_o:nw { sqrt } }
14135     \fi:
14136     \if_meaning:w 1 #2 \else: \__fp_case_return_same_o:w \fi:
14137     \__fp_sqrt_npos_o:w
14138     \s__fp \__fp_chk:w #2 #3 #4;
14139 }

```

(End definition for __fp_sqrt_o:w.)

__fp_sqrt_npos_o:w 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 $x = 57234133 \simeq 10^{7.75}$. 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 $17782794 \simeq 10^{7.25}$.

```

14140 \cs_new:Npn \__fp_sqrt_npos_o:w \s__fp \__fp_chk:w 1 0 #1#2#3#4#5;
14141 {
14142     \exp_after:wN \__fp_sanitize:Nw
14143     \exp_after:wN 0
14144     \__int_value:w \__int_eval:w
14145     \if_int_odd:w #1 \exp_stop_f:
14146         \exp_after:wN \__fp_sqrt_npos_auxi_o:wwnnN
14147     \fi:
14148     #1 / \c_two
14149     \__fp_sqrt_Newton_o:wwn 56234133; 0; {#2#3} {#4#5} 0
14150 }
14151 \cs_new:Npn \__fp_sqrt_npos_auxi_o:wwnnN #1 / \c_two #2; 0; #3#4#5
14152 {
14153     ( #1 + \c_one ) / \c_two
14154     \__fp_pack_eight:wNNNNNNNNN
14155     \__fp_sqrt_npos_auxii_o:wwnnNNNNN
14156     ;
14157     0 #3 #4

```

```

14158 }
14159 \cs_new:Npn \__fp_sqrt_npos_auxii_o:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
14160 { \__fp_sqrt_Newton_o:wnn 17782794; 0; {#1} {#2#3#4#5#6#7#8#9} }

```

(End definition for `__fp_sqrt_npos_o:w`, `__fp_sqrt_npos_auxi_o:wnnnN`, and `__fp_sqrt_npos_auxii_o:wNNNNNNNN`.)

`__fp_sqrt_Newton_o:wnn`

Newton's method maps $x \mapsto [(x + [10^8 a_1/x])/2]$ in each iteration, where $[b/c]$ denotes ε -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 ε -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

$$x' = \left\lceil \frac{x + [10^8 a_1/x]}{2} \right\rceil \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}.$$

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 δ decreases at each step: since all x are integers, δ 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:wnn` receives the newly computed result as `#1`, the previous result as `#2`, and a_1 as `#3`. Note that ε -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 $[10^7, 10^8]$.

```

14161 \cs_new:Npn \__fp_sqrt_Newton_o:wnn #1; #2; #3
14162 {
14163   \if_int_compare:w #1 = #2 \exp_stop_f:
14164     \exp_after:wN \__fp_sqrt_auxi_o:NNNNwnnnN
14165     \__int_value:w \__int_eval:w 9999 9999 +

```

```

14166         \exp_after:wN \__fp_use_none_until_s:w
14167     \fi:
14168     \exp_after:wN \__fp_sqrt_Newton_o:wnn
14169     \__int_value:w \__int_eval:w (#1 + #3 * 1 0000 0000 / #1) / \c_two ;
14170     #1; {#3}
14171 }

```

(End definition for __fp_sqrt_Newton_o:wnn.)

__fp_sqrt_auxi_o:NNNNwnnnN

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 will be 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, $y < 1$.

```

14172 \cs_new:Npn \__fp_sqrt_auxi_o:NNNNwnnnN 1 #1#2#3#4#5;
14173 {
14174     \__fp_sqrt_auxii_o:NnnnnnnnnN
14175     \__fp_sqrt_auxiii_o:wnnnnnnnnn
14176     {#1#2#3#4} {#5} {2499} {9988} {7500}
14177 }

```

(End definition for __fp_sqrt_auxi_o:NNNNwnnnN.)

__fp_sqrt_auxii_o:NnnnnnnnnN

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 ε -TeX's rounding division)

$$10^{4j}z = \left[([10^{4j}(a - y^2)] - 257) \cdot (0.5 \cdot 10^8) \right] / [10^8 y + 1].$$

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 TeX'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

$$10^{4j}(\sqrt{a} - y) = \frac{10^{4j}(a - y^2 - (\sqrt{a} - y)^2)}{2y} \geq \frac{[10^{4j}(a - y^2)] - 257}{2 \cdot 10^{-8} [10^8 y + 1]} + \frac{1}{2}$$

where we have replaced the bound $10^{4j}(16 \cdot 10^{-2j}) = 256$ by 257 and extracted the corresponding term $1/(2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor) \geq 1/2$. Given that $\varepsilon\text{-TeX}$'s integer division obeys $\lfloor b/c \rfloor \leq b/c + 1/2$, 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.

```

14178 \cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnN #1 #2#3#4#5#6 #7#8#9
14179 {
14180   \exp_after:wN #1
14181   \__int_value:w \__int_eval:w \c__fp_big_leading_shift_int
14182   + #7 - #2 * #2
14183   \exp_after:wN \__fp_pack_big:NNNNNNw
14184   \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14185   - 2 * #2 * #3
14186   \exp_after:wN \__fp_pack_big:NNNNNNw
14187   \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14188   + #8 - #3 * #3 - 2 * #2 * #4
14189   \exp_after:wN \__fp_pack_big:NNNNNNw
14190   \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14191   - 2 * #3 * #4 - 2 * #2 * #5
14192   \exp_after:wN \__fp_pack_big:NNNNNNw
14193   \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14194   + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
14195   \exp_after:wN \__fp_pack_big:NNNNNNw
14196   \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14197   - 2 * #4 * #5 - 2 * #3 * #6
14198   \exp_after:wN \__fp_pack_big:NNNNNNw
14199   \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14200   - #5 * #5 - 2 * #4 * #6
14201   \exp_after:wN \__fp_pack_big:NNNNNNw
14202   \__int_value:w \__int_eval:w
14203   \c__fp_big_middle_shift_int
14204   - 2 * #5 * #6
14205   \exp_after:wN \__fp_pack_big:NNNNNNw
14206   \__int_value:w \__int_eval:w
14207   \c__fp_big_trailing_shift_int
14208   - #6 * #6 ;
14209   % (
14210   - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
14211   {#2}{#3}{#4}{#5}{#6} {#7}{#8}{#9}
14212 }

```

(End definition for `__fp_sqrt_auxii_o:NnnnnnnN`.)

```

\__fp_sqrt_auxiii_o:wnnnnnnnN
\__fp_sqrt_auxiv_o:NNNNNw
\__fp_sqrt_auxv_o:NNNNNw
\__fp_sqrt_auxvi_o:NNNNNw
\__fp_sqrt_auxvii_o:NNNNNw

```

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:NnnnnnnN`, which completes the expression

$$10^{4j}z = \left[(\lfloor 10^{4j}(a - y^2) \rfloor - 257) \cdot (0.5 \cdot 10^8) \right] / \lfloor 10^8 y + 1 \rfloor$$

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 $10^4 d_2 + d_3 \geq 2$, $j = 4$ 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

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{[10^8y + 1]} \geq (20000 - 257) \cdot 0.5 > 0.$$

```

14213 \cs_new:Npn \_fp_sqrt_auxiii_o:wnnnnnnnn
14214   #1; #2#3#4#5#6#7#8#9
14215   {
14216     \if_int_compare:w #1 > \c_one
14217       \exp_after:wN \_fp_sqrt_auxiv_o:NNNNNw
14218       \__int_value:w \__int_eval:w (#1#2 %)
14219     \else:
14220       \if_int_compare:w #1#2 > \c_one
14221         \exp_after:wN \_fp_sqrt_auxv_o:NNNNNw
14222         \__int_value:w \__int_eval:w (#1#2#3 %)
14223       \else:
14224         \if_int_compare:w #1#2#3 > \c_one
14225           \exp_after:wN \_fp_sqrt_auxvi_o:NNNNNw
14226           \__int_value:w \__int_eval:w (#1#2#3#4 %)
14227         \else:
14228           \exp_after:wN \_fp_sqrt_auxvii_o:NNNNNw
14229           \__int_value:w \__int_eval:w (#1#2#3#4#5 %)
14230         \fi:
14231       \fi:
14232     \fi:
14233   }
14234 \cs_new:Npn \_fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6;
14235   { \_fp_sqrt_auxviii_o:nnnnnnn {#1#2#3#4#5#6} {00000000} }
14236 \cs_new:Npn \_fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
14237   { \_fp_sqrt_auxviii_o:nnnnnnn {000#1#2#3#4#5} {#60000} }
14238 \cs_new:Npn \_fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
14239   { \_fp_sqrt_auxviii_o:nnnnnnn {0000000#1} {#2#3#4#5#6} }
14240 \cs_new:Npn \_fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
14241   {
14242     \if_int_compare:w #1#2 = \c_zero
14243       \exp_after:wN \_fp_sqrt_auxx_o:Nnnnnnnn
14244     \fi:
14245     \_fp_sqrt_auxviii_o:nnnnnnn {00000000} {000#1#2#3#4#5}
14246   }

```

(End definition for `_fp_sqrt_auxiii_o:wnnnnnnnn` and others.)

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 $y'^2 = (y + z)^2$.

```

14247 \cs_new:Npn \_fp_sqrt_auxviii_o:nnnnnnn #1#2 #3#4#5#6#7
14248   {
14249     \exp_after:wN \_fp_sqrt_auxix_o:wnwnw
14250     \__int_value:w \__int_eval:w #3

```

```

14251     \exp_after:wN \__fp_basics_pack_low:NNNNNw
14252     \__int_value:w \__int_eval:w #1 + 1#4#5
14253     \exp_after:wN \__fp_basics_pack_low:NNNNNw
14254     \__int_value:w \__int_eval:w #2 + 1#6#7 ;
14255 }
14256 \cs_new:Npn \__fp_sqrt_auxix_o:wnnnw #1; #2#3; #4#5;
14257 {
14258     \__fp_sqrt_auxii_o:NnnnnnnnN
14259     \__fp_sqrt_auxiii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5}
14260 }

```

(End definition for __fp_sqrt_auxviii_o:nnnnnnnn and __fp_sqrt_auxix_o:wnnnw.)

```

\__fp_sqrt_auxx_o:Nnnnnnnnn
\__fp_sqrt_auxxi_o:wnnnN

```

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 $a \leq 1 - 10^{-16}$.

```

14261 \cs_new:Npn \__fp_sqrt_auxx_o:Nnnnnnnnn #1#2#3 #4#5#6#7#8
14262 {
14263     \exp_after:wN \__fp_sqrt_auxxi_o:wnnnN
14264     \__int_value:w \__int_eval:w
14265     (#8 + 2499) / 5000 * 5000 ;
14266     {#4} {#5} {#6} {#7} ;
14267 }
14268 \cs_new:Npn \__fp_sqrt_auxxi_o:wnnnN #1; #2; #3#4#5
14269 {
14270     \__fp_sqrt_auxii_o:NnnnnnnnnN
14271     \__fp_sqrt_auxxii_o:nnnnnnnnnw
14272     #2 {#1}
14273     {#3} { #4 + \c_one } #5
14274 }

```

(End definition for __fp_sqrt_auxx_o:Nnnnnnnnn and __fp_sqrt_auxxi_o:wnnnN.)

```

\__fp_sqrt_auxxii_o:nnnnnnnnnw
\__fp_sqrt_auxxiii_o:w

```

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.

```

14275 \cs_new:Npn \__fp_sqrt_auxxii_o:nnnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
14276 {
14277   \if_int_compare:w #1#2 > \c_zero
14278     \if_int_compare:w #1#2 = \c_one
14279       \if_int_compare:w #3#4 = \c_zero
14280         \if_int_compare:w #5#6 = \c_zero
14281           \if_int_compare:w #7#8 = \c_zero
14282             \__fp_sqrt_auxxiii_o:w
14283           \fi:
14284         \fi:
14285       \fi:
14286     \fi:
14287     \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnnN
14288     \__int_value:w 9998
14289   \else:
14290     \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnnN
14291     \__int_value:w 10000
14292   \fi:
14293 ;
14294 }
14295 \cs_new:Npn \__fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
14296 {
14297   \fi: \fi: \fi: \fi: \fi:
14298   \__fp_sqrt_auxxiv_o:wnnnnnnnnN 9999 ;
14299 }

```

(End definition for `__fp_sqrt_auxxii_o:nnnnnnnnw` and `__fp_sqrt_auxxiii_o:w`.)

`__fp_sqrt_auxxiv_o:wnnnnnnnnN`

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.

```

14300 \cs_new:Npn \__fp_sqrt_auxxiv_o:wnnnnnnnnN #1; #2#3#4#5#6 #7#8#9
14301 {
14302   \exp_after:wN \__fp_basics_pack_high:NNNNNw
14303   \__int_value:w \__int_eval:w 1 0000 0000 + #2#3
14304   \exp_after:wN \__fp_basics_pack_low:NNNNNw
14305   \__int_value:w \__int_eval:w 1 0000 0000
14306   + #4#5
14307   \if_int_compare:w #6 > #1 \exp_stop_f: + \c_one \fi:
14308   + \exp_after:wN \__fp_round:NNN
14309   \exp_after:wN 0

```

```

14310         \exp_after:wN 0
14311         \__int_value:w
14312         \exp_after:wN \use_i:nn
14313         \exp_after:wN \__fp_round_digit:Nw
14314         \__int_value:w \__int_eval:w #6 + 19999 - #1 ;
14315     \exp_after:wN ;
14316 }

```

(End definition for `__fp_sqrt_auxxiv_o:wnnnnnnnN`.)

27.6 Setting the sign

`__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`.

```

14317 \cs_new:Npn \__fp_set_sign_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14318 {
14319     \exp_after:wN \__fp_exp_after_o:w
14320     \exp_after:wN \s__fp
14321     \exp_after:wN \__fp_chk:w
14322     \exp_after:wN #2
14323     \__int_value:w
14324     \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
14325     #4;
14326 }

```

(End definition for `__fp_set_sign_o:w`.)

```
14327 </initex | package>
```

28 l3fp-extended implementation

```
14328 <*initex | package>
```

```
14329 <@@=fp>
```

28.1 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

$$\{\langle a_1 \rangle\} \{\langle a_2 \rangle\} \{\langle a_3 \rangle\} \{\langle a_4 \rangle\} \{\langle a_5 \rangle\} \{\langle a_6 \rangle\} ;$$

where each $\langle a_i \rangle$ is exactly 4 digits (ranging from 0000 to 9999), except $\langle a_1 \rangle$, 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 They perform the $\langle calculation \rangle$ on the two $\langle operands \rangle$, then feed the result (6 brace groups followed by a semicolon) to the $\langle continuation \rangle$, responsible for the next step of the calculation. Some functions only accept an N-type $\langle continuation \rangle$. This allows constructions such as

```

\__fp_fixed_add:wn  $\langle X_1 \rangle$  ;  $\langle X_2 \rangle$  ;
\__fp_fixed_mul:wn  $\langle X_3 \rangle$  ;
\__fp_fixed_add:wn  $\langle X_4 \rangle$  ;

```

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: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.

28.2 Helpers for numbers with extended precision

`\c__fp_one_fixed_tl` The fixed-point number 1, used in `l3fp-expo`.

```

14330 \tl_const:Nn \c__fp_one_fixed_tl
14331 { {10000} {0000} {0000} {0000} {0000} {0000} }

```

(End definition for `\c__fp_one_fixed_tl`.)

`__fp_fixed_continue:wn` This function does nothing. Of course, there is no bound on a_1 (except $\text{T}_{\text{E}}\text{X}$'s own $2^{31} - 1$).

```

14332 \cs_new:Npn \__fp_fixed_continue:wn #1; #2 { #2 #1; }

```

(End definition for `__fp_fixed_continue:wn`.)

`__fp_fixed_add_one:wn` 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 $a_1 \leq 2^{31} - 10001$.

```

14333 \cs_new:Npn \__fp_fixed_add_one:wn #1#2; #3
14334 {
14335   \exp_after:wn #3 \exp_after:wn
14336   { \__int_value:w \__int_eval:w \c_ten_thousand + #1 } #2 ;
14337 }

```

(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 $[0, 5 \cdot 10^8 - 1]$.

```

14338 \cs_new:Npn \__fp_fixed_div_myriad:wn #1#2#3#4#5#6;
14339 {
14340   \exp_after:wn \__fp_fixed_mul_after:wn
14341   \__int_value:w \__int_eval:w \c__fp_leading_shift_int
14342   \exp_after:wn \__fp_pack:NNNNNw
14343   \__int_value:w \__int_eval:w \c__fp_trailing_shift_int
14344   + #1 ; {#2}{#3}{#4}{#5};
14345 }

```

(End definition for `__fp_fixed_div_myriad:wn`.)

`__fp_fixed_mul_after:wnn` The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the $\langle continuation \rangle$ #2 in front. The $\langle continuation \rangle$ was brought up through the expansions by the packing functions.

```
14346 \cs_new:Npn \__fp_fixed_mul_after:wnn #1; #2; #3 { #3 {#1} #2; }
```

(End definition for `__fp_fixed_mul_after:wnn`.)

28.3 Multiplying a fixed point number by a short one

`__fp_fixed_mul_short:wnn` 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: a second operand of $\{0001\}\{0000\}\{0000\}$ will leave the first operand unchanged (rather than dividing it by 10^4 , as `__fp_fixed_mul:wnn` would).

```
14347 \cs_new:Npn \__fp_fixed_mul_short:wnn #1#2#3#4#5#6; #7#8#9;
14348 {
14349   \exp_after:wN \__fp_fixed_mul_after:wnn
14350   \__int_value:w \__int_eval:w \c__fp_leading_shift_int
14351   + #1*#7
14352   \exp_after:wN \__fp_pack:NNNNNw
14353   \__int_value:w \__int_eval:w \c__fp_middle_shift_int
14354   + #1*#8 + #2*#7
14355   \exp_after:wN \__fp_pack:NNNNNw
14356   \__int_value:w \__int_eval:w \c__fp_middle_shift_int
14357   + #1*#9 + #2*#8 + #3*#7
14358   \exp_after:wN \__fp_pack:NNNNNw
14359   \__int_value:w \__int_eval:w \c__fp_middle_shift_int
14360   + #2*#9 + #3*#8 + #4*#7
14361   \exp_after:wN \__fp_pack:NNNNNw
14362   \__int_value:w \__int_eval:w \c__fp_middle_shift_int
14363   + #3*#9 + #4*#8 + #5*#7
14364   \exp_after:wN \__fp_pack:NNNNNw
14365   \__int_value:w \__int_eval:w \c__fp_trailing_shift_int
14366   + #4*#9 + #5*#8 + #6*#7
14367   + ( #5*#9 + #6*#8 + #6*#9 / \c_ten_thousand )
14368   / \c_ten_thousand ; ;
14369 }
```

(End definition for `__fp_fixed_mul_short:wnn`.)

28.4 Dividing a fixed point number by a small integer

`__fp_fixed_div_int:wnN` Divides the fixed point number $\langle a \rangle$ by the (small) integer $0 < \langle n \rangle < 10^4$ and feeds the result to the $\langle continuation \rangle$. There is no bound on a_1 .

`__fp_fixed_div_int:wnN` 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 .

`__fp_fixed_div_int_auxii:wnn` 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 will have 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.

`__fp_fixed_div_int_pack:Nw` When the `iii` auxiliary is called, the situation looks like this:

`__fp_fixed_div_int_after:Nw`

```

    \__fp_fixed_div_int_after:Nw <continuation>
    -1 + Q1
    \__fp_fixed_div_int_pack:Nw 9999 + Q2
    \__fp_fixed_div_int_pack:Nw 9999 + Q3
    \__fp_fixed_div_int_pack:Nw 9999 + Q4
    \__fp_fixed_div_int_pack:Nw 9999 + Q5
    \__fp_fixed_div_int_pack:Nw 9999
    \__fp_fixed_div_int_auxii:wnn Q6 ; {<n>} {<a6>}

```

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 `<continuation>` as appropriate.

```

14370 \cs_new:Npn \__fp_fixed_div_int:wwN #1#2#3#4#5#6 ; #7 ; #8
14371 {
14372   \exp_after:wN \__fp_fixed_div_int_after:Nw
14373   \exp_after:wN #8
14374   \__int_value:w \__int_eval:w - \c_one
14375   \__fp_fixed_div_int:wnN
14376   #1; {#7} \__fp_fixed_div_int_auxi:wnn
14377   #2; {#7} \__fp_fixed_div_int_auxi:wnn
14378   #3; {#7} \__fp_fixed_div_int_auxi:wnn
14379   #4; {#7} \__fp_fixed_div_int_auxi:wnn
14380   #5; {#7} \__fp_fixed_div_int_auxi:wnn
14381   #6; {#7} \__fp_fixed_div_int_auxii:wnn ;
14382 }
14383 \cs_new:Npn \__fp_fixed_div_int:wnN #1; #2 #3
14384 {
14385   \exp_after:wN #3
14386   \__int_value:w \__int_eval:w #1 / #2 - \c_one ;
14387   {#2}
14388   {#1}
14389 }
14390 \cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
14391 {
14392   + #1
14393   \exp_after:wN \__fp_fixed_div_int_pack:Nw
14394   \__int_value:w \__int_eval:w 9999
14395   \exp_after:wN \__fp_fixed_div_int:wnN
14396   \__int_value:w \__int_eval:w #3 - #1*#2 \__int_eval_end:
14397 }
14398 \cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + \c_two ; }
14399 \cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
14400 \cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }

```

(End definition for `__fp_fixed_div_int:wwN` and others.)

28.5 Adding and subtracting fixed points

`__fp_fixed_add:wnn` Computes $a + b$ (resp. $a - b$) and feeds the result to the `<continuation>`. This function requires $0 \leq a_1, b_1 \leq 114748$, its result must be positive (this happens automatically for

```

\__fp_fixed_add:Nnnnnwnn
\__fp_fixed_add:nnNnnnwn
\__fp_fixed_add_pack:NNNNNwn
\__fp_fixed_add_after:NNNNNwn

```

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.

```

14401 \cs_new:Npn \__fp_fixed_add:wnn { \__fp_fixed_add:Nnnnnwnn + }
14402 \cs_new:Npn \__fp_fixed_sub:wnn { \__fp_fixed_add:Nnnnnwnn - }
14403 \cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
14404 {
14405   \exp_after:wN \__fp_fixed_add_after:NNNNNwn
14406   \__int_value:w \__int_eval:w 9 9999 9998 + #2#3 #1 #7#8
14407   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
14408   \__int_value:w \__int_eval:w 1 9999 9998 + #4#5
14409   \__fp_fixed_add:nnNnnwn #6 #1
14410 }
14411 \cs_new:Npn \__fp_fixed_add:nnNnnwn #1#2 #3 #4#5 #6#7 ; #8
14412 {
14413   #3 #4#5
14414   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
14415   \__int_value:w \__int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
14416 }
14417 \cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
14418 { + #1 ; {#7} {#2#3#4#5} {#6} }
14419 \cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
14420 { #7 {#1#2#3#4#5} {#6} }

```

(End definition for `__fp_fixed_add:wnn` and others.)

28.6 Multiplying fixed points

`__fp_fixed_mul:wnn` Computes $a \times b$ and feeds the result to $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 < 10000$. Once more, we need to play around the limit of 9 arguments for \TeX 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 $\langle continuation \rangle$ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The $\langle continuation \rangle$ is finally placed in front of the 6 brace groups by `_fp_fixed_mul_after:wwn`.

```

14421 \cs_new:Npn \_fp\_fixed\_mul:wwn #1#2#3#4 #5; #6#7#8#9
14422 {
14423   \exp\_after:wN \_fp\_fixed\_mul\_after:wwn
14424   \_int\_value:w \_int\_eval:w \c\_fp\_leading\_shift\_int
14425   \exp\_after:wN \_fp\_pack:NNNNNw
14426   \_int\_value:w \_int\_eval:w \c\_fp\_middle\_shift\_int
14427   + #1*#6
14428   \exp\_after:wN \_fp\_pack:NNNNNw
14429   \_int\_value:w \_int\_eval:w \c\_fp\_middle\_shift\_int
14430   + #1*#7 + #2*#6
14431   \exp\_after:wN \_fp\_pack:NNNNNw
14432   \_int\_value:w \_int\_eval:w \c\_fp\_middle\_shift\_int
14433   + #1*#8 + #2*#7 + #3*#6
14434   \exp\_after:wN \_fp\_pack:NNNNNw
14435   \_int\_value:w \_int\_eval:w \c\_fp\_middle\_shift\_int
14436   + #1*#9 + #2*#8 + #3*#7 + #4*#6
14437   \exp\_after:wN \_fp\_pack:NNNNNw
14438   \_int\_value:w \_int\_eval:w \c\_fp\_trailing\_shift\_int
14439   + #2*#9 + #3*#8 + #4*#7
14440   + ( #3*#9 + #4*#8
14441     + \_fp\_fixed\_mul:nnnnnnnw #5 {#6}{#7} {#1}{#2}
14442   )
14443 \cs\_new:Npn \_fp\_fixed\_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
14444 {
14445   #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c\_ten\_thousand
14446   + #1*#3 + #5*#7 ; ;
14447 }

```

(End definition for `_fp_fixed_mul:wwn` and `_fp_fixed_mul:nnnnnnnw`.)

28.7 Combining product and sum of fixed points

`_fp_fixed_mul_add:wwwn` Compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the $\langle continuation \rangle$.
`_fp_fixed_mul_sub_back:wwwn` Those functions require $0 \leq a_1, b_1, c_1 \leq 10000$. Since those functions are at the heart of
`_fp_fixed_mul_one_minus_mul:wwn` 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 will 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 $+ c_5 c_6$; $\{\langle continuation \rangle\}$; . The $+ c_5 c_6$ piece, which is omitted for `__fp_fixed_one_minus_mul:wnn`, will be taken in the integer expression for the 10^{-24} level.

```

14448 \cs_new:Npn \__fp_fixed_mul_add:wwn #1; #2; #3#4#5#6#7#8;
14449 {
14450   \exp_after:wN \__fp_fixed_mul_after:wnn
14451   \__int_value:w \__int_eval:w \c__fp_big_leading_shift_int
14452   \exp_after:wN \__fp_pack_big:NNNNNNw
14453   \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int + #3 #4
14454   \__fp_fixed_mul_add:Nwnnnwnnn +
14455   + #5 #6 ; #2 ; #1 ; #2 ; +
14456   + #7 #8 ; ;
14457 }
14458 \cs_new:Npn \__fp_fixed_mul_sub_back:wwn #1; #2; #3#4#5#6#7#8;
14459 {
14460   \exp_after:wN \__fp_fixed_mul_after:wnn
14461   \__int_value:w \__int_eval:w \c__fp_big_leading_shift_int
14462   \exp_after:wN \__fp_pack_big:NNNNNNw
14463   \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int + #3 #4
14464   \__fp_fixed_mul_add:Nwnnnwnnn -
14465   + #5 #6 ; #2 ; #1 ; #2 ; -
14466   + #7 #8 ; ;
14467 }
14468 \cs_new:Npn \__fp_fixed_one_minus_mul:wnn #1; #2;
14469 {
14470   \exp_after:wN \__fp_fixed_mul_after:wnn
14471   \__int_value:w \__int_eval:w \c__fp_big_leading_shift_int
14472   \exp_after:wN \__fp_pack_big:NNNNNNw
14473   \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int + 1 0000 0000
14474   \__fp_fixed_mul_add:Nwnnnwnnn -
14475   ; #2 ; #1 ; #2 ; -
14476   ; ;
14477 }

```

(End definition for `_fp_fixed_mul_add:www`, `_fp_fixed_mul_sub_back:www`, and `_fp_fixed_mul_one_minus_mul:wwn`.)

`_fp_fixed_mul_add:Nwnnnwnnn` 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 - 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.

```

14478 \cs_new:Npn \_fp\_fixed\_mul\_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
14479 {
14480   #1 #7*#3
14481   \exp_after:wN \_fp\_pack\_big:NNNNNNw
14482   \_int_value:w \_int_eval:w \c\_fp\_big\_middle\_shift\_int
14483   #1 #7*#4 #1 #8*#3
14484   \exp_after:wN \_fp\_pack\_big:NNNNNNw
14485   \_int_value:w \_int_eval:w \c\_fp\_big\_middle\_shift\_int
14486   #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
14487   \exp_after:wN \_fp\_pack\_big:NNNNNNw
14488   \_int_value:w \_int_eval:w \c\_fp\_big\_middle\_shift\_int
14489   #1 \_fp\_fixed\_mul\_add:nnnnwnnnn {#7}{#8}{#9}
14490 }

```

(End definition for `_fp_fixed_mul_add:Nwnnnwnnn`.)

`_fp_fixed_mul_add:nnnnwnnnN` 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

$$b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1$$

$$b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2.$$

Obviously, those expressions make no mathematical sense: we will 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 $\langle b \rangle$.

```

14491 \cs_new:Npn \_fp\_fixed\_mul\_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
14492 {
14493   ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
14494   \exp_after:wN \_fp\_pack\_big:NNNNNNw
14495   \_int_value:w \_int_eval:w \c\_fp\_big\_trailing\_shift\_int
14496   \_fp\_fixed\_mul\_add:nnnnwnnnN
14497   { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
14498   { #7 + #4*#8 + #3*#9 + #2 }
14499   {#1} #5;
14500   {#6}
14501 }

```

(End definition for `_fp_fixed_mul_add:nnnnwnnnn`.)

`_fp_fixed_mul_add:nnnnwnnnN` 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.

```

14502 \cs_new:Npn \__fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
14503 {
14504     #9 (#4* #1 *#7)
14505     #9 (#5*#6+#4* #2 *#7+#3*#8) / \c_ten_thousand
14506 }

```

(End definition for __fp_fixed_mul_add:nnnnwnnwN.)

28.8 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.

```

\__fp_ep_to_fixed:wwN  Converts an extended-precision number with an exponent at most 4 to a fixed point
\__fp_ep_to_fixed_auxi:www number whose first block will have 12 digits, most often starting with many zeros.
\__fp_ep_to_fixed_auxii:nnnnnnwnwN
14507 \cs_new:Npn \__fp_ep_to_fixed:wwN #1,#2
14508 {
14509     \exp_after:wN \__fp_ep_to_fixed_auxi:www
14510     \__int_value:w \__int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
14511     \exp:w \exp_end_continue_f:w
14512     \prg_replicate:nn { \c_four - \int_max:nn {#1} { -32 } } { 0 } ;
14513 }
14514 \cs_new:Npn \__fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
14515 {
14516     \__fp_pack_eight:wNNNNNNNN
14517     \__fp_pack_twice_four:wNNNNNNNN
14518     \__fp_pack_twice_four:wNNNNNNNN
14519     \__fp_pack_twice_four:wNNNNNNNN
14520     \__fp_ep_to_fixed_auxii:nnnnnnwnwN ;
14521     #2 #1#3#4#5#6#7 0000 !
14522 }
14523 \cs_new:Npn \__fp_ep_to_fixed_auxii:nnnnnnwnwN #1#2#3#4#5#6#7; #8! #9
14524 { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }

```

(End definition for __fp_ep_to_fixed:wwN, __fp_ep_to_fixed_auxi:www, and __fp_ep_to_fixed_auxii:nnnnnnwnwN.)

```

\__fp_ep_to_ep:wwN  Normalize an extended-precision number. More precisely, leading zeros are removed from
\__fp_ep_to_ep_loop:N the mantissa of the argument, decreasing its exponent as appropriate. Then the digits
\__fp_ep_to_ep_end:www are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally,
\__fp_ep_to_ep_zero:ww 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).

```

14525 \cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
14526 {
14527   \exp_after:wN #8
14528   \__int_value:w \__int_eval:w #1 + \c_four
14529   \exp_after:wN \use_i:nn
14530   \exp_after:wN \__fp_ep_to_ep_loop:N
14531   \__int_value:w \__int_eval:w 1 0000 0000 + #2 \__int_eval_end:
14532   #3#4#5#6#7 ; ; !
14533 }
14534 \cs_new:Npn \__fp_ep_to_ep_loop:N #1
14535 {
14536   \if_meaning:w 0 #1
14537   - \c_one
14538   \else:
14539     \__fp_ep_to_ep_end:www #1
14540   \fi:
14541   \__fp_ep_to_ep_loop:N
14542 }
14543 \cs_new:Npn \__fp_ep_to_ep_end:www
14544 #1 \fi: \__fp_ep_to_ep_loop:N #2; #3!
14545 {
14546   \fi:
14547   \if_meaning:w ; #1
14548   - \c_two * \c_fp_max_exponent_int
14549   \__fp_ep_to_ep_zero:ww
14550   \fi:
14551   \__fp_pack_twice_four:wNNNNNNNN
14552   \__fp_pack_twice_four:wNNNNNNNN
14553   \__fp_pack_twice_four:wNNNNNNNN
14554   \__fp_use_i:ww , ;
14555   #1 #2 0000 0000 0000 0000 0000 0000 ;
14556 }
14557 \cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
14558 { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }

```

(End definition for __fp_ep_to_ep:wwN and others.)

__fp_ep_compare:www
 __fp_ep_compare_aux:www

In l3fp-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].

```

14559 \cs_new:Npn \__fp_ep_compare:www #1,#2#3#4#5#6#7;
14560 { \__fp_ep_compare_aux:www {#1}{#2}{#3}{#4}{#5}; #6#7; }
14561 \cs_new:Npn \__fp_ep_compare_aux:www #1;#2;#3;#4#5#6#7#8#9;
14562 {
14563   \if_case:w
14564     \__fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
14565     \if_int_compare:w #2 = #8#9 \exp_stop_f:
14566     0
14567   \else:
14568     \if_int_compare:w #2 < #8#9 - \fi: 1
14569   \fi:

```

```

14570     \or:      1
14571     \else: -1
14572     \fi:
14573   }

```

(End definition for `__fp_ep_compare:www` and `__fp_ep_compare_aux:www`.)

`__fp_ep_mul:www`
`__fp_ep_mul_raw:www`

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]`.

```

14574 \cs_new:Npn \__fp_ep_mul:www #1,#2; #3,#4;
14575 {
14576   \__fp_ep_to_ep:wwN #3,#4;
14577   \__fp_fixed_continue:wn
14578   {
14579     \__fp_ep_to_ep:wwN #1,#2;
14580     \__fp_ep_mul_raw:wwwN
14581   }
14582   \__fp_fixed_continue:wn
14583 }
14584 \cs_new:Npn \__fp_ep_mul_raw:wwwN #1,#2; #3,#4; #5
14585 {
14586   \__fp_fixed_mul:wn #2; #4;
14587   { \exp_after:wN #5 \__int_value:w \__int_eval:w #1 + #3 , }
14588 }

```

(End definition for `__fp_ep_mul:www` and `__fp_ep_mul_raw:www`.)

28.9 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 will first find an integer estimate $a \simeq 10^8 / \langle d \rangle$ by computing

$$\begin{aligned}
 \alpha &= \left\lceil \frac{10^9}{\langle d_1 \rangle + 1} \right\rceil \\
 \beta &= \left\lceil \frac{10^9}{\langle d_1 \rangle} \right\rceil \\
 a &= 10^3 \alpha + (\beta - \alpha) \cdot \left(10^3 - \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil \right) - 1250,
 \end{aligned}$$

where $\left\lceil \frac{\cdot}{\cdot} \right\rceil$ denotes ε -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 will 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 $\varepsilon\text{-TeX}$'s division $\left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor$ will at most underestimate $10^{-1}(\langle d_2 \rangle + 1)$ by 0.5, as can be checked for each possible last digit of $\langle d_2 \rangle$. Then,

$$10^7 \langle d \rangle a < \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \beta + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \alpha - 1250 \right) \quad (1)$$

$$< \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \quad (2)$$

$$\left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \left(\frac{10^9}{\langle d_1 \rangle} + \frac{1}{2} \right) + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \left(\frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2} \right) - 1250 \right) \quad (3)$$

$$< \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 750 \right) \quad (4)$$

We recognize a quadratic polynomial in $\left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor$ with a negative leading coefficient: this polynomial is bounded above, according to $(\left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + a)(b - c\left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor) \leq (b + ca)^2 / (4c)$. 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

$$10^7 \langle d \rangle a > \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor - \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 1750 \right)$$

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 will give 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` *<denominator>* *<numerator>*, responsible for estimating the inverse of the denominator.

```

14589 \cs_new:Npn \__fp_ep_div:wwwn #1,#2; #3,#4;
14590 {
14591   \__fp_ep_to_ep:wwN #1,#2;
14592   \__fp_fixed_continue:wn
14593   {
14594     \__fp_ep_to_ep:wwN #3,#4;
14595     \__fp_ep_div_esti:wwwn
14596   }
14597 }

```

(End definition for __fp_ep_div:wwwn.)

```

\__fp_ep_div_esti:wwwn
\__fp_ep_div_estii:wwnnwwn
\__fp_ep_div_estiii:NNNNNwwwn

```

The **esti** function evaluates $\alpha = 10^9 / (\langle d_1 \rangle + 1)$, which is used twice in the expression for a , and combines the exponents #1 and #4 (with a shift by 1 because we will 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_epsilon:wnNNNNn, which computes $10^{-9}a / (1 - \epsilon)$, that is, $1 / (10 \langle d \rangle)$ and we finally multiply this by the numerator #8.

```

14598 \cs_new:Npn \__fp_ep_div_esti:wwwn #1,#2#3; #4,
14599 {
14600   \exp_after:wN \__fp_ep_div_estii:wwnnwwn
14601   \__int_value:w \__int_eval:w 10 0000 0000 / ( #2 + \c_one )
14602   \exp_after:wN ;
14603   \__int_value:w \__int_eval:w #4 - #1 + \c_one ,
14604   {#2} #3;
14605 }
14606 \cs_new:Npn \__fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
14607 {
14608   \exp_after:wN \__fp_ep_div_estiii:NNNNNwwwn
14609   \__int_value:w \__int_eval:w 10 0000 0000 - 1750
14610   + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
14611   {#3}{#4}#5; #6; { #7 #2, }
14612 }
14613 \cs_new:Npn \__fp_ep_div_estiii:NNNNNwwwn 1#1#2#3#4#5#6; #7;
14614 {
14615   \__fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
14616   \__fp_ep_div_epsilon:wnNNNNn {#1#2#3#4}#5#6
14617   \__fp_fixed_mul:wwn
14618 }

```

(End definition for __fp_ep_div_esti:wwwn, __fp_ep_div_estii:wwnnwwn, and __fp_ep_div_estiii:NNNNNwwwn.)

```

\__fp_ep_div_epsilon:wnNNNNn
\__fp_ep_div_eps_pack:NNNNNw
\__fp_ep_div_epsii:wwnnNNNNn

```

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 ϵ 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 ϵ 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**.


```

14619 \cs_new:Npn \__fp_ep_div_epsilon:wnNNNNNn #1#2#3#4#5#6;
14620 {
14621   \exp_after:wN \__fp_ep_div_epsilon:wnNNNNNn
14622   \__int_value:w \__int_eval:w 1 9998 - #2
14623   \exp_after:wN \__fp_ep_div_epsilon_pack:NNNNNw
14624   \__int_value:w \__int_eval:w 1 9999 9998 - #3#4
14625   \exp_after:wN \__fp_ep_div_epsilon_pack:NNNNNw
14626   \__int_value:w \__int_eval:w 2 0000 0000 - #5#6 ; ;
14627 }
14628 \cs_new:Npn \__fp_ep_div_epsilon_pack:NNNNNw #1#2#3#4#5#6;
14629 { + #1 ; {#2#3#4#5} {#6} }
14630 \cs_new:Npn \__fp_ep_div_epsilon:wnNNNNNn 1#1; #2; #3#4#5#6#7#8
14631 {
14632   \__fp_fixed_mul:wnn {0000}{#1}#2; {0000}{#1}#2;
14633   \__fp_fixed_add_one:wN
14634   \__fp_fixed_mul:wnn {10000} {#1} #2 ;
14635   {
14636     \__fp_fixed_mul_short:wnn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
14637     \__fp_fixed_div_myriad:wn
14638     \__fp_fixed_mul:wnn
14639   }
14640   \__fp_fixed_add:wnn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
14641 }

```

(End definition for `__fp_ep_div_epsilon:wnNNNNNn`, `__fp_ep_div_epsilon_pack:NNNNNw`, and `__fp_ep_div_epsilon:wnNNNNNn`.)

28.10 Inverse square root of extended precision numbers

The idea here is similar to division. Normalize the input, multiplying by powers of 100 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 will replace 10^8 by a slightly larger number which will ensure 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 $y = 10^{-4} r^2 x$ (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 $x^{-1/2} = 10^{-2} r y^{-1/2}$.

```

\__fp_ep_isqrt:wnn
\__fp_ep_isqrt_aux:wnn
\__fp_ep_isqrt_auxii:wnnnwn

```

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 $10^4 x$ (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.

```

14642 \cs_new:Npn \__fp_ep_isqrt:wnn #1,#2;
14643 {
14644   \__fp_ep_to_ep:wwN #1,#2;
14645   \__fp_ep_isqrt_auxi:wnn
14646 }
14647 \cs_new:Npn \__fp_ep_isqrt_auxi:wnn #1,
14648 {
14649   \exp_after:wN \__fp_ep_isqrt_auxii:wwnnwn
14650   \__int_value:w \__int_eval:w
14651   \int_if_odd:nTF {#1}
14652     { (\c_one - #1) / \c_two , 535 , { 0 } { } }
14653     { \c_one - #1 / \c_two , 168 , { } { 0 } }
14654 }
14655 \cs_new:Npn \__fp_ep_isqrt_auxii:wwnnwn #1, #2, #3#4 #5#6; #7
14656 {
14657   \__fp_ep_isqrt_esti:wwnnwn #2, 0, #5, {#3} {#4}
14658   {#5} #6 ; { #7 #1 , }
14659 }

```

(End definition for __fp_ep_isqrt:wnn, __fp_ep_isqrt_aux:wnn, and __fp_ep_isqrt_auxii:wwnnwn.)

```

\__fp_ep_isqrt_esti:wwnnwn
\__fp_ep_isqrt_estii:wwnnwn
\__fp_ep_isqrt_estiii:NNNNNwwnn

```

If the last two approximations gave the same result, we are done: call the `esti` 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_epsilon:wN`, which computes $1/\sqrt{y}$, and we finally multiply the result by r .

```

14660 \cs_new:Npn \__fp_ep_isqrt_esti:wwnnwn #1, #2, #3, #4
14661 {
14662   \if_int_compare:w #1 = #2 \exp_stop_f:
14663     \exp_after:wN \__fp_ep_isqrt_estii:wwnnwn
14664   \fi:
14665   \exp_after:wN \__fp_ep_isqrt_esti:wwnnwn
14666   \__int_value:w \__int_eval:w
14667   (#1 + 1 0050 0000 #4 / (#1 * #3)) / \c_two ,
14668   #1, #3, {#4}
14669 }
14670 \cs_new:Npn \__fp_ep_isqrt_estii:wwnnwn #1, #2, #3, #4#5
14671 {
14672   \exp_after:wN \__fp_ep_isqrt_estiii:NNNNNwwnn
14673   \__int_value:w \__int_eval:w 1000 0000 + #2 * #2 #5 * \c_five
14674   \exp_after:wN , \__int_value:w \__int_eval:w 10000 + #2 ;
14675 }
14676 \cs_new:Npn \__fp_ep_isqrt_estiii:NNNNNwwnn 1#1#2#3#4#5#6, 1#7#8; #9;
14677 {
14678   \__fp_fixed_mul_short:wnn #9; {#1} {#2#3#4#5} {#600} ;

```

```

14679     \__fp_ep_isqrt_epsilon:wwN
14680     \__fp_fixed_mul_short:wwn {#7} {#80} {0000} ;
14681 }

```

(End definition for __fp_ep_isqrt_esti:wwnnwn, __fp_ep_isqrt_estii:wwnnwn, and __fp_ep_isqrt_estiii:NNNNNwwnn.)

```

\__fp_ep_isqrt_epsilon:wwN
\__fp_ep_isqrt_epsilonii:wwN

```

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`.

```

14682 \cs_new:Npn \__fp_ep_isqrt_epsilon:wwN #1;
14683 {
14684     \__fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}{0000}; #1;
14685     \__fp_ep_isqrt_epsilonii:wwN #1;
14686     \__fp_ep_isqrt_epsilonii:wwN #1;
14687     \__fp_ep_isqrt_epsilonii:wwN #1;
14688 }
14689 \cs_new:Npn \__fp_ep_isqrt_epsilonii:wwN #1; #2;
14690 {
14691     \__fp_fixed_mul:wwn #1; #1;
14692     \__fp_fixed_mul_sub_back:wwnn #2;
14693     {15000}{0000}{0000}{0000}{0000}{0000};
14694     \__fp_fixed_mul:wwn #1;
14695 }

```

(End definition for __fp_ep_isqrt_epsilon:wwN and __fp_ep_isqrt_epsilonii:wwN.)

28.11 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:wwN
\__fp_ep_inv_to_float: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.

```

14696 \cs_new:Npn \__fp_ep_to_float:wwN #1,
14697 { + \__int_eval:w #1 \__fp_fixed_to_float:wwN }
14698 \cs_new:Npn \__fp_ep_inv_to_float:wwN #1,#2;
14699 {
14700     \__fp_ep_div:wwnn 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1,#2;
14701     \__fp_ep_to_float:wwN
14702 }

```

(End definition for __fp_ep_to_float:wwN and __fp_ep_inv_to_float:wwN.)

```

\__fp_fixed_inv_to_float:wwN

```

Another function which reduces to converting an extended precision number to a float.

```

14703 \cs_new:Npn \__fp_fixed_inv_to_float:wwN
14704 { \__fp_ep_inv_to_float:wwN 0, }

```

(End definition for __fp_fixed_inv_to_float:wwN.)

`_fp_fixed_to_float_rad:wN` Converts the fixed point number #1 from degrees to radians then to a floating point number. This could perhaps remain in `l3fp-trig`.

```

14705 \cs_new:Npn \_fp_fixed_to_float_rad:wN #1;
14706 {
14707   \_fp_fixed_mul:wwN #1; {5729}{5779}{5130}{8232}{0876}{7981};
14708   { \_fp_ep_to_float:wwN 2, }
14709 }

```

(End definition for `_fp_fixed_to_float_rad:wN`.)

`_fp_fixed_to_float:wN` yields

`_fp_fixed_to_float:Nw` $\langle exponent \rangle$; $\{\langle a'_1 \rangle\} \{\langle a'_2 \rangle\} \{\langle a'_3 \rangle\} \{\langle a'_4 \rangle\}$;

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 .¹¹

```

14710 \cs_new:Npn \_fp_fixed_to_float:Nw #1#2; { \_fp_fixed_to_float:wN #2; #1 }
14711 \cs_new:Npn \_fp_fixed_to_float:wN #1#2#3#4#5#6; #7
14712 {
14713   + \_int_eval:w \c_four % for the 8-digit-at-the-start thing.
14714   \exp_after:wN \exp_after:wN
14715   \exp_after:wN \_fp_fixed_to_loop:N
14716   \exp_after:wN \use_none:n
14717   \_int_value:w \_int_eval:w
14718   1 0000 0000 + #1 \exp_after:wN \_fp_use_none_stop_f:n
14719   \_int_value:w 1#2 \exp_after:wN \_fp_use_none_stop_f:n
14720   \_int_value:w 1#3#4 \exp_after:wN \_fp_use_none_stop_f:n
14721   \_int_value:w 1#5#6
14722   \exp_after:wN ;
14723   \exp_after:wN ;
14724 }
14725 \cs_new:Npn \_fp_fixed_to_loop:N #1
14726 {
14727   \if_meaning:w 0 #1
14728   - \c_one
14729   \exp_after:wN \_fp_fixed_to_loop:N
14730   \else:
14731   \exp_after:wN \_fp_fixed_to_loop_end:w
14732   \exp_after:wN #1
14733   \fi:
14734 }
14735 \cs_new:Npn \_fp_fixed_to_loop_end:w #1 #2 ;
14736 {
14737   \if_meaning:w ; #1
14738   \exp_after:wN \_fp_fixed_to_float_zero:w
14739   \else:
14740   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
14741   \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
14742   \exp_after:wN \_fp_fixed_to_float_pack:ww
14743   \exp_after:wN ;
14744   \fi:

```

¹¹Bruno: I must double check this assumption.

```

14745     #1 #2 0000 0000 0000 0000 ;
14746   }
14747   \cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
14748   {
14749     - \c_two * \c__fp_max_exponent_int ;
14750     {0000} {0000} {0000} {0000} ;
14751   }
14752   \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
14753   {
14754     \if_int_compare:w #2 > \c_four
14755       \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
14756       \fi:
14757     ; #1 ;
14758   }
14759   \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
14760   {
14761     \exp_after:wN \__fp_basics_pack_high:NNNNNw
14762     \__int_value:w \__int_eval:w 1 #1#2
14763     \exp_after:wN \__fp_basics_pack_low:NNNNNw
14764     \__int_value:w \__int_eval:w 1 #3#4 + \c_one ;
14765   }

```

(End definition for __fp_fixed_to_float:wN and __fp_fixed_to_float:Nw.)

```

14766 \</initex | package>

```

29 l3fp-expo implementation

```

14767 \*initex | package>
14768 \@@=fp>

```

29.1 Logarithm

29.1.1 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, $c \in [1, 10]$ will be 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.

29.1.2 Some constants

A few values of the logarithm as extended fixed point numbers. Those are needed in the implementation. It turns out that we don't need the value of $\ln(5)$.

```
\c__fp_ln_i_fixed_t1
\c__fp_ln_ii_fixed_t1
\c__fp_ln_iii_fixed_t1
\c__fp_ln_iv_fixed_t1
\c__fp_ln_vi_fixed_t1
\c__fp_ln_vii_fixed_t1
\c__fp_ln_viii_fixed_t1
\c__fp_ln_ix_fixed_t1
\c__fp_ln_x_fixed_t1
14769 \tl_const:Nn \c__fp_ln_i_fixed_t1 { {0000}{0000}{0000}{0000}{0000} }
14770 \tl_const:Nn \c__fp_ln_ii_fixed_t1 { {6931}{4718}{0559}{9453}{0941}{7232} }
14771 \tl_const:Nn \c__fp_ln_iii_fixed_t1 { {10986}{1228}{8668}{1096}{9139}{5245} }
14772 \tl_const:Nn \c__fp_ln_iv_fixed_t1 { {13862}{9436}{1119}{8906}{1883}{4464} }
14773 \tl_const:Nn \c__fp_ln_vi_fixed_t1 { {17917}{5946}{9228}{0550}{0081}{2477} }
14774 \tl_const:Nn \c__fp_ln_vii_fixed_t1 { {19459}{1014}{9055}{3133}{0510}{5353} }
14775 \tl_const:Nn \c__fp_ln_viii_fixed_t1 { {20794}{4154}{1679}{8359}{2825}{1696} }
14776 \tl_const:Nn \c__fp_ln_ix_fixed_t1 { {21972}{2457}{7336}{2193}{8279}{0490} }
14777 \tl_const:Nn \c__fp_ln_x_fixed_t1 { {23025}{8509}{2994}{0456}{8401}{7991} }
```

(End definition for `\c__fp_ln_i_fixed_t1` and others.)

29.1.3 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`.

```
14778 \cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14779 {
14780   \if_meaning:w 2 #3
14781     \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
14782   \fi:
14783   \if_case:w #2 \exp_stop_f:
14784     \__fp_case_use:nw
14785     { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
14786   \or:
14787   \else:
14788     \__fp_case_return_same_o:w
14789   \fi:
14790   \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4;
14791 }
```

(End definition for `__fp_ln_o:w`.)

29.1.4 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.

```
14792 \cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;
14793 { %^A todo: ln(1) should be "exact zero", not "underflow"
14794   \exp_after:wN \__fp_sanitize:Nw
14795   \__int_value:w % for the overall sign
```

```

14796     \if_int_compare:w #1 < \c_one
14797         2
14798     \else:
14799         0
14800     \fi:
14801     \exp_after:wN \exp_stop_f:
14802     \__int_value:w \__int_eval:w % for the exponent
14803     \__fp_ln_significand:NNNNnnnnN #2#3
14804     \__fp_ln_exponent:wn {#1}
14805 }

```

(End definition for __fp_ln_npos_o:w.)

__fp_ln_significand:NNNNnnnnN $\langle X_1 \rangle \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \} \langle continuation \rangle$
This function expands to

$\langle continuation \rangle \{ \langle Y_1 \rangle \} \{ \langle Y_2 \rangle \} \{ \langle Y_3 \rangle \} \{ \langle Y_4 \rangle \} \{ \langle Y_5 \rangle \} \{ \langle Y_6 \rangle \} ;$

where $Y = -\ln(X)$ as an extended fixed point.

```

14806 \cs_new:Npn \__fp_ln_significand:NNNNnnnnN #1#2#3#4
14807 {
14808     \exp_after:wN \__fp_ln_x_ii:wnnnnn
14809     \__int_value:w
14810     \if_case:w #1 \exp_stop_f:
14811     \or:
14812         \if_int_compare:w #2 < \c_four
14813             \__int_eval:w \c_ten - #2
14814         \else:
14815             6
14816         \fi:
14817     \or: 4
14818     \or: 3
14819     \or: 2
14820     \or: 2
14821     \or: 2
14822     \else: 1
14823     \fi:
14824     ; { #1 #2 #3 #4 }
14825 }

```

(End definition for __fp_ln_significand:NNNNnnnnN.)

__fp_ln_x_ii:wnnnnn We have thus found $c \in [1, 10]$ such that $0.7 \leq ac < 1.4$ in all cases. Compute $1 + x = 1 + ac \in [1.7, 2.4)$.

```

14826 \cs_new:Npn \__fp_ln_x_ii:wnnnnn #1; #2#3#4#5
14827 {
14828     \exp_after:wN \__fp_ln_div_after:Nw
14829     \cs:w c__fp_ln_ \__int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
14830     \__int_value:w
14831     \exp_after:wN \__fp_ln_x_iv:wnnnnnnnnn
14832     \__int_value:w \__int_eval:w
14833     \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
14834     \__int_value:w \__int_eval:w 9999 9990 + #1*#2#3 +
14835     \exp_after:wN \__fp_ln_x_iii:NNNNNNw
14836     \__int_value:w \__int_eval:w 10 0000 0000 + #1*#4#5 ;

```

```

14837     {20000} {0000} {0000} {0000}
14838   } %^A todo: reoptimize (a generalization attempt failed).
14839 \cs_new:Npn \__fp_ln_x_iii:NNNNNw #1#2 #3#4#5#6 #7;
14840   { #1#2; {#3#4#5#6} {#7} }
14841 \cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
14842   {
14843     #1#2#3#4#5 + \c_one ;
14844     {#1#2#3#4#5} {#6}
14845   }

```

The Taylor series will be 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 `eTeX` 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).¹²

`__fp_ln_x_iv:wnnnnnnnn <1 or 2> <8d> ; {<4d>} {<4d>} <fixed-tl>`

The number is x . Compute y by adding 1 to the five first digits.

```

14846 \cs_new:Npn \__fp_ln_x_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
14847 {
14848   \exp_after:wN \__fp_div_significand_pack:NNN
14849   \__int_value:w \__int_eval:w
14850   \__fp_ln_div_i:w #1 ;
14851   #6 #7 ; {#8} {#9}
14852   {#2} {#3} {#4} {#5}
14853   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
14854   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
14855   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
14856   { \exp_after:wN \__fp_ln_div_ii:wnn \__int_value:w #1 }
14857   { \exp_after:wN \__fp_ln_div_vi:wnn \__int_value:w #1 }
14858 }
14859 \cs_new:Npn \__fp_ln_div_i:w #1;
14860 {
14861   \exp_after:wN \__fp_div_significand_calc:wnnnnnnnn
14862   \__int_value:w \__int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
14863 }
14864 \cs_new:Npn \__fp_ln_div_ii:wnn #1; #2;#3 % y; B1;B2 <- for k=1
14865 {
14866   \exp_after:wN \__fp_div_significand_pack:NNN
14867   \__int_value:w \__int_eval:w
14868   \exp_after:wN \__fp_div_significand_calc:wnnnnnnnn
14869   \__int_value:w \__int_eval:w 999999 + #2 #3 / #1 ; % Q2
14870   #2 #3 ;
14871 }
14872 \cs_new:Npn \__fp_ln_div_vi:wnn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
14873 {

```

¹²Bruno: to be completed.

```

14874 \exp_after:wN \_fp_div_significand_pack:NNN
14875 \_int_value:w \_int_eval:w 1000000 + #2 #3 / #1 ; % Q6
14876 }

```

We now have essentially¹³

$$\begin{aligned} & _fp_ln_div_after:Nw \langle fixed\ tl \rangle _fp_div_significand_pack:NNN 10^6 + \\ & Q_1 _fp_div_significand_pack:NNN 10^6 + Q_2 _fp_div_significand_ \\ & pack:NNN 10^6 + Q_3 _fp_div_significand_pack:NNN 10^6 + Q_4 _fp_ \\ & div_significand_pack:NNN 10^6 + Q_5 _fp_div_significand_pack:NNN \\ & 10^6 + Q_6 ; \langle exponent \rangle ; \langle continuation \rangle \end{aligned}$$

where $\langle fixed\ tl \rangle$ holds the logarithm of a number in $[1, 10]$, and $\langle exponent \rangle$ 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

$$\begin{aligned} & _fp_ln_div_after:Nw \langle fixed\ tl \rangle \langle 1d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \\ & \langle 4d \rangle ; \langle exponent \rangle ; \end{aligned}$$

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.

```

14877 \cs_new:Npn \_fp_ln_div_after:Nw #1#2;
14878 {
14879   \if_meaning:w 0 #2
14880     \exp_after:wN \_fp_ln_t_small:Nw
14881   \else:
14882     \exp_after:wN \_fp_ln_t_large:NNw
14883     \exp_after:wN -
14884   \fi:
14885   #1
14886 }
14887 \cs_new:Npn \_fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
14888 {
14889   \exp_after:wN \_fp_ln_t_large:NNw
14890   \exp_after:wN + % <sign>
14891   \exp_after:wN #1
14892   \_int_value:w \_int_eval:w 9999 - #2 \exp_after:wN ;
14893   \_int_value:w \_int_eval:w 9999 - #3 \exp_after:wN ;
14894   \_int_value:w \_int_eval:w 9999 - #4 \exp_after:wN ;
14895   \_int_value:w \_int_eval:w 9999 - #5 \exp_after:wN ;
14896   \_int_value:w \_int_eval:w 9999 - #6 \exp_after:wN ;
14897   \_int_value:w \_int_eval:w 1 0000 - #7 ;
14898 }

```

$$\begin{aligned} & _fp_ln_t_large:NNw \langle sign \rangle \langle fixed\ tl \rangle \langle t_1 \rangle ; \langle t_2 \rangle ; \langle t_3 \rangle ; \langle t_4 \rangle ; \langle t_5 \rangle ; \langle t_6 \rangle ; \\ & \langle exponent \rangle ; \langle continuation \rangle \end{aligned}$$

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.

¹³Bruno: add a mention that the error on Q_6 is bounded by 10 (probably 6.7), and thus corresponds to an error of 10^{-23} on the final result, small enough in all cases.

```

14899 \cs_new:Npn \__fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
14900 {
14901   \exp_after:wN \__fp_ln_square_t_after:w
14902   \__int_value:w \__int_eval:w 9999 0000 + #3*#3
14903   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
14904   \__int_value:w \__int_eval:w 9999 0000 + 2*#3*#4
14905   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
14906   \__int_value:w \__int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
14907   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
14908   \__int_value:w \__int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
14909   \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
14910   \__int_value:w \__int_eval:w 1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
14911   + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
14912   % ; ; ;
14913   \exp_after:wN \__fp_ln_twice_t_after:w
14914   \__int_value:w \__int_eval:w -1 + 2*#3
14915   \exp_after:wN \__fp_ln_twice_t_pack:Nw
14916   \__int_value:w \__int_eval:w 9999 + 2*#4
14917   \exp_after:wN \__fp_ln_twice_t_pack:Nw
14918   \__int_value:w \__int_eval:w 9999 + 2*#5
14919   \exp_after:wN \__fp_ln_twice_t_pack:Nw
14920   \__int_value:w \__int_eval:w 9999 + 2*#6
14921   \exp_after:wN \__fp_ln_twice_t_pack:Nw
14922   \__int_value:w \__int_eval:w 9999 + 2*#7
14923   \exp_after:wN \__fp_ln_twice_t_pack:Nw
14924   \__int_value:w \__int_eval:w 10000 + 2*#8 ; ;
14925   { \__fp_ln_c:NwNw #1 }
14926   #2
14927 }
14928 \cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
14929 \cs_new:Npn \__fp_ln_twice_t_after:w #1; { ; ; ; {#1} }
14930 \cs_new:Npn \__fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6;
14931   { + #1#2#3#4#5 ; {#6} }
14932 \cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4;
14933   { \__fp_ln_Taylor:wwNw {0#1#2#3} {#4} }

```

(End definition for __fp_ln_x_ii:wnnnn.)

__fp_ln_Taylor:wwNw Denoting $T = t^2$, we get

__fp_ln_Taylor:wwNw $\{\langle T_1 \rangle\} \{\langle T_2 \rangle\} \{\langle T_3 \rangle\} \{\langle T_4 \rangle\} \{\langle T_5 \rangle\} \{\langle T_6 \rangle\} ; ;$
 $\{\langle (2t)_1 \rangle\} \{\langle (2t)_2 \rangle\} \{\langle (2t)_3 \rangle\} \{\langle (2t)_4 \rangle\} \{\langle (2t)_5 \rangle\} \{\langle (2t)_6 \rangle\} ; \{__fp_ln_c:NwNn \langle sign \rangle \langle fixed\ tl \rangle \langle exponent \rangle \langle continuation \rangle$

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + T \left(\frac{1}{3} + T \left(\frac{1}{5} + T \left(\frac{1}{7} + T \left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$

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;

```

14

This uses the routine for dividing a number by a small integer ($< 10^4$).

```

14934 \cs_new:Npn \__fp_ln_Taylor:wwNw
14935 { \__fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
14936 \cs_new:Npn \__fp_ln_Taylor_loop:www #1; #2; #3;
14937 {
14938   \if_int_compare:w #1 = \c_one
14939     \__fp_ln_Taylor_break:w
14940   \fi:
14941   \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl ; #1;
14942   \__fp_fixed_add:wwn #2;
14943   \__fp_fixed_mul:wwn #3;
14944   {
14945     \exp_after:wN \__fp_ln_Taylor_loop:www
14946     \__int_value:w \__int_eval:w #1 - \c_two ;
14947   }
14948   #3;
14949 }
14950 \cs_new:Npn \__fp_ln_Taylor_break:w \fi: #1 \__fp_fixed_add:wwn #2#3; #4 ;;
14951 {
14952   \fi:
14953   \exp_after:wN \__fp_fixed_mul:wwn
14954   \exp_after:wN { \__int_value:w \__int_eval:w 10000 + #2 } #3;
14955 }

```

(End definition for `__fp_ln_Taylor:wwNw`.)

`__fp_ln_c:NwNw` $\langle sign \rangle \{ \langle r_1 \rangle \} \{ \langle r_2 \rangle \} \{ \langle r_3 \rangle \} \{ \langle r_4 \rangle \} \{ \langle r_5 \rangle \} \{ \langle r_6 \rangle \} ; \langle fixed\ tl \rangle$
 $\langle exponent \rangle ; \langle continuation \rangle$

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 will be at least $\ln(10/7) \simeq 0.35$.¹⁵

```

14956 \cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
14957 {
14958   \if_meaning:w + #1
14959     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wwn
14960   \else:
14961     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wwn
14962   \fi:
14963   #3 ; #2 ;
14964 }

```

16

(End definition for `__fp_ln_c:NwNw`.)

`__fp_ln_exponent:wn` $__fp_ln_exponent:wn \{ \langle s_1 \rangle \} \{ \langle s_2 \rangle \} \{ \langle s_3 \rangle \} \{ \langle s_4 \rangle \} \{ \langle s_5 \rangle \} \{ \langle s_6 \rangle \} ;$
 $\{ \langle exponent \rangle \}$

¹⁴Bruno: add explanations.

¹⁵Bruno: that was wrong at some point, I must check.

¹⁶Bruno: this *must* be updated with correct values!

Compute $\langle exponent \rangle$ times $\ln(10)$. Apart from the cases where $\langle exponent \rangle$ is 0 or 1, the result will necessarily be 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.

```

14965 \cs_new:Npn \__fp_ln_exponent:wn #1; #2
14966 {
14967   \if_case:w #2 \exp_stop_f:
14968     \c_zero \__fp_case_return:nw { \__fp_fixed_to_float:Nw 2 }
14969   \or:
14970     \exp_after:wN \__fp_ln_exponent_one:ww \__int_value:w
14971   \else:
14972     \if_int_compare:w #2 > \c_zero
14973       \exp_after:wN \__fp_ln_exponent_small:NNww
14974       \exp_after:wN 0
14975       \exp_after:wN \__fp_fixed_sub:wwn \__int_value:w
14976     \else:
14977       \exp_after:wN \__fp_ln_exponent_small:NNww
14978       \exp_after:wN 2
14979       \exp_after:wN \__fp_fixed_add:wwn \__int_value:w -
14980     \fi:
14981   \fi:
14982   #2; #1;
14983 }

```

Now we painfully write all the cases.¹⁷ No overflow nor underflow can happen, except when computing $\ln(1)$.

```

14984 \cs_new:Npn \__fp_ln_exponent_one:ww 1; #1;
14985 {
14986   \c_zero
14987   \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_t1 ; #1;
14988   \__fp_fixed_to_float:wN 0
14989 }

```

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.

```

14990 \cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
14991 {
14992   \c_four
14993   \exp_after:wN \__fp_fixed_mul:wwn
14994     \c__fp_ln_x_fixed_t1 ;
14995     {#3}{0000}{0000}{0000}{0000}{0000} ;
14996   #2
14997     {0000}{#4}{#5}{#6}{#7}{#8};
14998   \__fp_fixed_to_float:wN #1
14999 }

```

(End definition for $\backslash_fp_ln_exponent:wn$.)

¹⁷Bruno: do rounding.

29.2 Exponential

29.2.1 Sign, exponent, and special numbers

`__fp_exp_o:w`

```
15000 \cs_new:Npn \__fp_exp_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15001 {
15002   \if_case:w #2 \exp_stop_f:
15003     \__fp_case_return_o:Nw \c_one_fp
15004   \or:
15005     \exp_after:wN \__fp_exp_normal:w
15006   \or:
15007     \if_meaning:w 0 #3
15008       \exp_after:wN \__fp_case_return_o:Nw
15009       \exp_after:wN \c_inf_fp
15010     \else:
15011       \exp_after:wN \__fp_case_return_o:Nw
15012       \exp_after:wN \c_zero_fp
15013     \fi:
15014   \or:
15015     \__fp_case_return_same_o:w
15016   \fi:
15017   \s__fp \__fp_chk:w #2#3#4;
15018 }
```

(End definition for __fp_exp_o:w.)

`__fp_exp_normal:w`

`__fp_exp_pos:Nnwnw`

```
15019 \cs_new:Npn \__fp_exp_normal:w \s__fp \__fp_chk:w 1#1
15020 {
15021   \if_meaning:w 0 #1
15022     \__fp_exp_pos:Nnwnw + \__fp_fixed_to_float:wN
15023   \else:
15024     \__fp_exp_pos:Nnwnw - \__fp_fixed_inv_to_float:wN
15025   \fi:
15026 }
15027 \cs_new:Npn \__fp_exp_pos:Nnwnw #1#2#3 \fi: #4#5;
15028 {
15029   \fi:
15030   \exp_after:wN \__fp_sanitize:Nw
15031   \exp_after:wN 0
15032   \__int_value:w #1 \__int_eval:w
15033   \if_int_compare:w #4 < - \c_eight
15034     \c_one
15035     \exp_after:wN \__fp_add_big_i_o:wNww
15036     \__int_value:w \__int_eval:w \c_one - #4 ;
15037     0 {1000}{0000}{0000}{0000} ; #5;
15038     \exp:w
15039   \else:
15040     \if_int_compare:w #4 > \c_five % cf \c__fp_max_exponent_int
15041       \exp_after:wN \__fp_exp_overflow:
15042       \exp:w
15043     \else:
15044       \if_int_compare:w #4 < \c_zero
15045         \exp_after:wN \use_i:nn
```

```

15046         \else:
15047             \exp_after:wN \use_ii:nn
15048         \fi:
15049         {
15050             \c_zero
15051             \__fp_decimate:nNnnnn { - #4 }
15052             \__fp_exp_Taylor:Nnnwn
15053         }
15054         {
15055             \__fp_decimate:nNnnnn { \c_sixteen - #4 }
15056             \__fp_exp_pos_large:NnnNwn
15057         }
15058         #5
15059         {#4}
15060         #1 #2 0
15061         \exp:w
15062     \fi:
15063     \fi:
15064     \exp_after:wN \c_zero
15065 }
15066 \cs_new:Npn \__fp_exp_overflow:
15067 { + \c_two * \c__fp_max_exponent_int ; {1000} {0000} {0000} {0000} ; }

```

(End definition for __fp_exp_normal:w and __fp_exp_pos:Nnnwn.)

__fp_exp_Taylor:Nnnwn This function is called for numbers in the range $[10^{-9}, 10^{-1}]$. Our only task is to compute 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.

```

\__fp_exp_Taylor_loop:www
\__fp_exp_Taylor_break:Nww

15068 \cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
15069 {
15070     #6
15071     \__fp_pack_twice_four:wNNNNNNNN
15072     \__fp_pack_twice_four:wNNNNNNNN
15073     \__fp_pack_twice_four:wNNNNNNNN
15074     \__fp_exp_Taylor_ii:ww
15075     ; #2#3#4 0000 0000 ;
15076 }
15077 \cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;
15078 { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s_stop }
15079 \cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
15080 {
15081     \if_int_compare:w #1 = \c_one
15082     \exp_after:wN \__fp_exp_Taylor_break:Nww
15083     \fi:
15084     \__fp_fixed_div_int:wwN #3 ; #1 ;
15085     \__fp_fixed_add_one:wN
15086     \__fp_fixed_mul:wwn #2 ;
15087     {
15088         \exp_after:wN \__fp_exp_Taylor_loop:www
15089         \__int_value:w \__int_eval:w #1 - 1 ;
15090         #2 ;
15091     }
15092 }

```

```

15093 \cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s__stop
15094 { \__fp_fixed_add_one:wN #2 ; }

```

(End definition for __fp_exp_Taylor:Nnnwn, __fp_exp_Taylor_loop:www, and __fp_exp_Taylor-break:Nww.)

```

\__fp_exp_pos_large:NnnNwn
\__fp_exp_large_after:wwn
  \__fp_exp_large:w
    \__fp_exp_large_v:wN
    \__fp_exp_large_iv:wN
    \__fp_exp_large_iii:wN
    \__fp_exp_large_ii:wN
    \__fp_exp_large_i:wN
    \__fp_exp_large:wN

```

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 having the auxiliary for one exponent call the auxiliary for the next exponent. The current total is expressed by leaving the exponent behind in the input stream (we are currently within an __int_eval:w), and keeping track of a fixed point number, #1 for the numbered auxiliaries. Our usage of \if_case:w is somewhat dirty for optimization: T_EX jumps to the appropriate case, but we then close the \if_case:w “by hand”, using \or: and \fi: as delimiters.

```

15095 \cs_new:Npn \__fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
15096 {
15097   \exp_after:wN \exp_after:wN
15098   \cs:w \__fp_exp_large_ \__int_to_roman:w #6 :wN \exp_after:wN \cs_end:
15099   \exp_after:wN \c__fp_one_fixed_tl
15100   \exp_after:wN ;
15101   \__int_value:w #3 #4 \exp_stop_f:
15102   #5 00000 ;
15103 }
15104 \cs_new:Npn \__fp_exp_large:w #1 \or: #2 \fi:
15105 { \fi: \__fp_fixed_mul:wwn #1; }
15106 \cs_new:Npn \__fp_exp_large_v:wN #1; #2
15107 {
15108   \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wn \or:
15109   + 4343 \__fp_exp_large:w {8806}{8182}{2566}{2921}{5872}{6150} \or:
15110   + 8686 \__fp_exp_large:w {7756}{0047}{2598}{6861}{0458}{3204} \or:
15111   + 13029 \__fp_exp_large:w {6830}{5723}{7791}{4884}{1932}{7351} \or:
15112   + 17372 \__fp_exp_large:w {6015}{5609}{3095}{3052}{3494}{7574} \or:
15113   + 21715 \__fp_exp_large:w {5297}{7951}{6443}{0315}{3251}{3576} \or:
15114   + 26058 \__fp_exp_large:w {4665}{6719}{0099}{3379}{5527}{2929} \or:
15115   + 30401 \__fp_exp_large:w {4108}{9724}{3326}{3186}{5271}{5665} \or:
15116   + 34744 \__fp_exp_large:w {3618}{6973}{3140}{0875}{3856}{4102} \or:
15117   + 39087 \__fp_exp_large:w {3186}{9209}{6113}{3900}{6705}{9685} \or:
15118   \fi:
15119   #1;
15120   \__fp_exp_large_iv:wN
15121 }
15122 \cs_new:Npn \__fp_exp_large_iv:wN #1; #2
15123 {
15124   \if_case:w #2 ~ \exp_after:wN \__fp_fixed_continue:wn \or:
15125   + 435 \__fp_exp_large:w {1970}{0711}{1401}{7046}{9938}{8888} \or:
15126   + 869 \__fp_exp_large:w {3881}{1801}{9428}{4368}{5764}{8232} \or:
15127   + 1303 \__fp_exp_large:w {7646}{2009}{8905}{4704}{8893}{1073} \or:
15128   + 1738 \__fp_exp_large:w {1506}{3559}{7005}{0524}{9009}{7592} \or:
15129   + 2172 \__fp_exp_large:w {2967}{6283}{8402}{3667}{0689}{6630} \or:
15130   + 2606 \__fp_exp_large:w {5846}{4389}{5650}{2114}{7278}{5046} \or:
15131   + 3041 \__fp_exp_large:w {1151}{7900}{5080}{6878}{2914}{4154} \or:

```



```

15132     + 3475 \_fp_exp_large:w {2269}{1083}{0850}{6857}{8724}{4002} \or:
15133     + 3909 \_fp_exp_large:w {4470}{3047}{3316}{5442}{6408}{6591} \or:
15134     \fi:
15135     #1;
15136     \_fp_exp_large_iii:wN
15137 }
15138 \cs_new:Npn \_fp_exp_large_iii:wN #1; #2
15139 {
15140     \if_case:w #2 ~          \exp_after:wN \_fp_fixed_continue:wn \or:
15141     + 44 \_fp_exp_large:w {2688}{1171}{4181}{6135}{4484}{1263} \or:
15142     + 87 \_fp_exp_large:w {7225}{9737}{6812}{5749}{2581}{7748} \or:
15143     + 131 \_fp_exp_large:w {1942}{4263}{9524}{1255}{9365}{8421} \or:
15144     + 174 \_fp_exp_large:w {5221}{4696}{8976}{4143}{9505}{8876} \or:
15145     + 218 \_fp_exp_large:w {1403}{5922}{1785}{2837}{4107}{3977} \or:
15146     + 261 \_fp_exp_large:w {3773}{0203}{0092}{9939}{8234}{0143} \or:
15147     + 305 \_fp_exp_large:w {1014}{2320}{5473}{5004}{5094}{5533} \or:
15148     + 348 \_fp_exp_large:w {2726}{3745}{7211}{2566}{5673}{6478} \or:
15149     + 391 \_fp_exp_large:w {7328}{8142}{2230}{7421}{7051}{8866} \or:
15150     \fi:
15151     #1;
15152     \_fp_exp_large_ii:wN
15153 }
15154 \cs_new:Npn \_fp_exp_large_ii:wN #1; #2
15155 {
15156     \if_case:w #2 ~          \exp_after:wN \_fp_fixed_continue:wn \or:
15157     + 5 \_fp_exp_large:w {2202}{6465}{7948}{0671}{6516}{9579} \or:
15158     + 9 \_fp_exp_large:w {4851}{6519}{5409}{7902}{7796}{9107} \or:
15159     + 14 \_fp_exp_large:w {1068}{6474}{5815}{2446}{2146}{9905} \or:
15160     + 18 \_fp_exp_large:w {2353}{8526}{6837}{0199}{8540}{7900} \or:
15161     + 22 \_fp_exp_large:w {5184}{7055}{2858}{7072}{4640}{8745} \or:
15162     + 27 \_fp_exp_large:w {1142}{0073}{8981}{5684}{2836}{6296} \or:
15163     + 31 \_fp_exp_large:w {2515}{4386}{7091}{9167}{0062}{6578} \or:
15164     + 35 \_fp_exp_large:w {5540}{6223}{8439}{3510}{0525}{7117} \or:
15165     + 40 \_fp_exp_large:w {1220}{4032}{9431}{7840}{8020}{0271} \or:
15166     \fi:
15167     #1;
15168     \_fp_exp_large_i:wN
15169 }
15170 \cs_new:Npn \_fp_exp_large_i:wN #1; #2
15171 {
15172     \if_case:w #2 ~          \exp_after:wN \_fp_fixed_continue:wn \or:
15173     + 1 \_fp_exp_large:w {2718}{2818}{2845}{9045}{2353}{6029} \or:
15174     + 1 \_fp_exp_large:w {7389}{0560}{9893}{0650}{2272}{3043} \or:
15175     + 2 \_fp_exp_large:w {2008}{5536}{9231}{8766}{7740}{9285} \or:
15176     + 2 \_fp_exp_large:w {5459}{8150}{0331}{4423}{9078}{1103} \or:
15177     + 3 \_fp_exp_large:w {1484}{1315}{9102}{5766}{0342}{1116} \or:
15178     + 3 \_fp_exp_large:w {4034}{2879}{3492}{7351}{2260}{8387} \or:
15179     + 4 \_fp_exp_large:w {1096}{6331}{5842}{8458}{5992}{6372} \or:
15180     + 4 \_fp_exp_large:w {2980}{9579}{8704}{1728}{2747}{4359} \or:
15181     + 4 \_fp_exp_large:w {8103}{0839}{2757}{5384}{0077}{1000} \or:
15182     \fi:
15183     #1;
15184     \_fp_exp_large_:wN
15185 }

```

```

15186 \cs_new:Npn \__fp_exp_large:wN #1; #2
15187 {
15188   \if_case:w #2 ~      \exp_after:wN \__fp_fixed_continue:wn \or:
15189     + 1 \__fp_exp_large:w {1105}{1709}{1807}{5647}{6248}{1171} \or:
15190     + 1 \__fp_exp_large:w {1221}{4027}{5816}{0169}{8339}{2107} \or:
15191     + 1 \__fp_exp_large:w {1349}{8588}{0757}{6003}{1039}{8374} \or:
15192     + 1 \__fp_exp_large:w {1491}{8246}{9764}{1270}{3178}{2485} \or:
15193     + 1 \__fp_exp_large:w {1648}{7212}{7070}{0128}{1468}{4865} \or:
15194     + 1 \__fp_exp_large:w {1822}{1188}{0039}{0508}{9748}{7537} \or:
15195     + 1 \__fp_exp_large:w {2013}{7527}{0747}{0476}{5216}{2455} \or:
15196     + 1 \__fp_exp_large:w {2225}{5409}{2849}{2467}{6045}{7954} \or:
15197     + 1 \__fp_exp_large:w {2459}{6031}{1115}{6949}{6638}{0013} \or:
15198   \fi:
15199   #1;
15200   \__fp_exp_large_after:wnn
15201 }
15202 \cs_new:Npn \__fp_exp_large_after:wnn #1; #2; #3
15203 {
15204   \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
15205   \__fp_fixed_mul:wnn #1;
15206 }

```

(End definition for `__fp_exp_pos_large:NnnNwn` and others.)

29.3 Power

Raising a number a to a power b leads to many distinct situations.

a^b	$-\infty$	$-y$	$-n$	± 0	$+n$	$+y$	$+\infty$	NaN
$+\infty$	+0	+0	+0	+1	$+\infty$	$+\infty$	$+\infty$	NaN
$1 < x$	+0	$+x^{-y}$	$+x^{-n}$	+1	$+x^n$	$+x^y$	$+\infty$	NaN
+1	+1	+1	+1	+1	+1	+1	+1	+1
$0 < x < 1$	$+\infty$	$+x^{-y}$	$+x^{-n}$	+1	$+x^n$	$+x^y$	+0	NaN
+0	$+\infty$	$+\infty$	$+\infty$	+1	+0	+0	+0	NaN
-0	NaN	NaN	$\pm\infty$	+1	± 0	+0	+0	NaN
$-1 < -x < 0$	NaN	NaN	$\pm x^{-n}$	+1	$\pm x^n$	NaN	+0	NaN
-1	NaN	NaN	± 1	+1	± 1	NaN	NaN	NaN
$-x < -1$	+0	NaN	$\pm x^{-n}$	+1	$\pm x^n$	NaN	NaN	NaN
$-\infty$	+0	+0	± 0	+1	$\pm\infty$	NaN	NaN	NaN
NaN	NaN	NaN	NaN	+1	NaN	NaN	NaN	NaN

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 a 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:ww` followed by the two `fp` a and b . For $a = +0$ or $+\text{inf}$, 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: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.

```

15207 \cs_new:cpn { __fp_ \iow_char:N \^_o:ww }
15208   \s__fp \__fp_chk:w #1#2#3; \s__fp \__fp_chk:w #4#5#6;
15209   {
15210     \if_meaning:w 0 #4
15211       \__fp_case_return_o:Nw \c_one_fp
15212     \fi:
15213     \if_case:w #2 \exp_stop_f:
15214       \exp_after:wN \use_i:nn
15215     \or:
15216       \__fp_case_return_o:Nw \c_nan_fp
15217     \else:
15218       \exp_after:wN \__fp_pow_neg:www
15219       \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
15220     \fi:
15221     {
15222       \if_meaning:w 1 #1
15223         \exp_after:wN \__fp_pow_normal:ww
15224       \else:
15225         \exp_after:wN \__fp_pow_zero_or_inf:ww
15226       \fi:
15227       \s__fp \__fp_chk:w #1#2#3;
15228     }
15229     { \s__fp \__fp_chk:w #4#5#6; \s__fp \__fp_chk:w #1#2#3; }
15230     \s__fp \__fp_chk:w #4#5#6;
15231   }

```

(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 ∞ 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 $a = \pm 0$ with $b < 0$ and we have a division by zero, or $a = \pm \infty$ and $b > 0$ and the result is also $+\infty$, but without any exception.

```

15232 \cs_new:Npn \__fp_pow_zero_or_inf:ww
15233   \s__fp \__fp_chk:w #1#2; \s__fp \__fp_chk:w #3#4
15234   {
15235     \if_meaning:w 1 #4
15236       \__fp_case_return_same_o:w
15237     \fi:
15238     \if_meaning:w #1 #4
15239       \__fp_case_return_o:Nw \c_zero_fp
15240     \fi:
15241     \if_meaning:w 0 #1
15242       \__fp_case_use:nw
15243       {
15244         \__fp_division_by_zero_o:NNww \c_inf_fp \^_
15245         \s__fp \__fp_chk:w #1 #2 ;
15246       }
15247     \else:

```

```

15248     \__fp_case_return_o:Nw \c_inf_fp
15249     \fi:
15250     \s__fp \__fp_chk:w #3#4
15251 }

```

(End definition for __fp_pow_zero_or_inf:ww.)

__fp_pow_normal: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:ww.
- 2 Return $+\infty$ or $+0$ depending on the sign of b and whether the exponent of a is positive or not.
- 3 Return b .

```

15252 \cs_new:Npn \__fp_pow_normal:ww
15253   \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
15254   {
15255     \if_int_compare:w \__str_if_eq_x:nn { #2 #3 }
15256       { 1 {1000} {0000} {0000} {0000} } = \c_zero
15257       \if_int_compare:w #4 #1 = 32 \exp_stop_f:
15258       \exp_after:wN \__fp_case_return_ii_o:ww
15259       \fi:
15260       \__fp_case_return_o:Nww \c_one_fp
15261     \fi:
15262     \if_case:w #4 \exp_stop_f:
15263     \or:
15264       \exp_after:wN \__fp_pow_npos:Nww
15265       \exp_after:wN #5
15266     \or:
15267       \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
15268       \if_int_compare:w #2 > \c_zero
15269         \exp_after:wN \__fp_case_return_o:Nww
15270         \exp_after:wN \c_inf_fp
15271       \else:
15272         \exp_after:wN \__fp_case_return_o:Nww
15273         \exp_after:wN \c_zero_fp
15274       \fi:
15275     \or:
15276       \__fp_case_return_ii_o:ww
15277     \fi:
15278     \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
15279     \s__fp \__fp_chk:w #4 #5
15280   }

```

(End definition for __fp_pow_normal:ww.)

__fp_pow_npos: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.

```

15281 \cs_new:Npn \__fp_pow_npos:Nww #1 \s__fp \__fp_chk:w 1#2#3
15282 {
15283   \exp_after:wN \__fp_sanitize:Nw
15284   \exp_after:wN 0
15285   \__int_value:w
15286   \if:w #1 \if_int_compare:w #3 > \c_zero 0 \else: 2 \fi:
15287     \exp_after:wN \__fp_pow_npos_aux:NNnw
15288     \exp_after:wN +
15289     \exp_after:wN \__fp_fixed_to_float:wN
15290   \else:
15291     \exp_after:wN \__fp_pow_npos_aux:NNnw
15292     \exp_after:wN -
15293     \exp_after:wN \__fp_fixed_inv_to_float:wN
15294   \fi:
15295   {#3}
15296 }

```

(End definition for `__fp_pow_npos:Nww`.)

`__fp_pow_npos_aux:NNnw`

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 $-\ln(x)$.

```

15297 \cs_new:Npn \__fp_pow_npos_aux:NNnw #1#2#3#4#5; \s__fp \__fp_chk:w 1#6#7#8;
15298 {
15299   #1
15300   \__int_eval:w
15301   \__fp_ln_significand:NNNNnnnN #4#5
15302   \__fp_pow_exponent:wnN {#3}
15303   \__fp_fixed_mul:wwN #8 {0000}{0000} ;
15304   \__fp_pow_B:wwN #7;
15305   #1 #2 0 % fixed_to_float:wN
15306 }
15307 \cs_new:Npn \__fp_pow_exponent:wnN #1; #2
15308 {
15309   \if_int_compare:w #2 > \c_zero
15310     \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
15311     \exp_after:wN +
15312   \else:
15313     \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % -(\ln|\ln(10) + (-\ln(x)))
15314     \exp_after:wN -
15315   \fi:
15316   #2; #1;
15317 }
15318 \cs_new:Npn \__fp_pow_exponent:Nwnnnnnw #1#2; #3#4#5#6#7#8;
15319 { %^^A todo: use that in ln.
15320   \exp_after:wN \__fp_fixed_mul_after:wwn
15321   \__int_value:w \__int_eval:w \c__fp_leading_shift_int
15322   \exp_after:wN \__fp_pack:NNNNNw
15323   \__int_value:w \__int_eval:w \c__fp_middle_shift_int
15324   #1#2*23025 - #1 #3

```

```

15325         \exp_after:wN \__fp_pack:NNNNNw
15326         \__int_value:w \__int_eval:w \c__fp_middle_shift_int
15327         #1 #2*8509 - #1 #4
15328         \exp_after:wN \__fp_pack:NNNNNw
15329         \__int_value:w \__int_eval:w \c__fp_middle_shift_int
15330         #1 #2*2994 - #1 #5
15331         \exp_after:wN \__fp_pack:NNNNNw
15332         \__int_value:w \__int_eval:w \c__fp_middle_shift_int
15333         #1 #2*0456 - #1 #6
15334         \exp_after:wN \__fp_pack:NNNNNw
15335         \__int_value:w \__int_eval:w \c__fp_trailing_shift_int
15336         #1 #2*8401 - #1 #7
15337         #1 ( #2*7991 - #8 ) / 1 0000 ; ;
15338     }
15339 \cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
15340 {
15341     \if_int_compare:w #7 < \c_zero
15342         \exp_after:wN \__fp_pow_C_neg:w \__int_value:w -
15343     \else:
15344         \if_int_compare:w #7 < 22 \exp_stop_f:
15345         \exp_after:wN \__fp_pow_C_pos:w \__int_value:w
15346     \else:
15347         \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
15348     \fi:
15349     \fi:
15350     #7 \exp_after:wN ;
15351     \__int_value:w \__int_eval:w 10 0000 + #1 \__int_eval_end:
15352     #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^A todo: how many 0?
15353 }
15354 \cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
15355 {
15356     + \c_two * \c__fp_max_exponent_int
15357     \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl ;
15358 }
15359 \cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
15360 {
15361     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
15362     \prg_replicate:nn {#1} {0}
15363 }
15364 \cs_new:Npn \__fp_pow_C_pos:w #1; 1
15365 { \__fp_pow_C_pos_loop:wN #1; }
15366 \cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
15367 {
15368     \if_meaning:w 0 #1
15369         \exp_after:wN \__fp_pow_C_pack:w
15370         \exp_after:wN #2
15371     \else:
15372         \if_meaning:w 0 #2
15373         \exp_after:wN \__fp_pow_C_pos_loop:wN \__int_value:w
15374     \else:
15375         \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
15376     \fi:
15377     \__int_eval:w #1 - \c_one \exp_after:wN ;
15378     \fi:

```

```

15379     }
15380 \cs_new:Npn \__fp_pow_C_pack:w
15381 { \exp_after:wN \__fp_exp_large_v:wN \c__fp_one_fixed_tl ; }

```

(End definition for __fp_pow_npos_aux:NNnw.)

__fp_pow_neg:www
__fp_pow_neg_aux:wNN

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, $(-0.1)**(12345.6)$ will give $+0$ rather than complaining that the sign is not defined.

```

15382 \cs_new:Npn \__fp_pow_neg:www \s__fp \__fp_chk:w #1#2; #3; #4;
15383 {
15384     \if_case:w \__fp_pow_neg_case:w #4 ;
15385     \exp_after:wN \__fp_pow_neg_aux:wNN
15386     \or:
15387     \if_int_compare:w \__int_eval:w #1 / \c_two = \c_one
15388     \__fp_invalid_operation_o:Nww ^ #3; #4;
15389     \exp:w \exp_end_continue_f:w
15390     \exp_after:wN \exp_after:wN
15391     \exp_after:wN \__fp_use_none_until_s:w
15392     \fi:
15393     \fi:
15394     \__fp_exp_after_o:w
15395     \s__fp \__fp_chk:w #1#2;
15396 }
15397 \cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s__fp \__fp_chk:w #2#3
15398 {
15399     \exp_after:wN \__fp_exp_after_o:w
15400     \exp_after:wN \s__fp
15401     \exp_after:wN \__fp_chk:w
15402     \exp_after:wN #2
15403     \__int_value:w \__int_eval:w \c_two - #3 \__int_eval_end:
15404 }

```

(End definition for __fp_pow_neg:www and __fp_pow_neg_aux:wNN.)

__fp_pow_neg_case:w
__fp_pow_neg_case_aux:nnnnn
__fp_pow_neg_case_aux:NNNNNNNNw

This function expects a floating point number, and “returns” -1 if it is an even integer, 0 if it is an odd integer, and 1 if it is not an integer. Zeros are even, $\pm\infty$ and nan are non-integers. The sign of normal numbers is irrelevant to parity. If the exponent is greater than sixteen, then the number is even. If the exponent is non-positive, the number cannot be an integer. We also separate the ranges of exponent $[1, 8]$ and $[9, 16]$. In the former case, check that the last 8 digits are zero (otherwise we don’t have an integer). In both cases, consider the appropriate 8 digits, either $\#4\#5$ or $\#2\#3$, remove the first few: we are then left with $\langle digit \rangle \langle digits \rangle$; which would be the digits surrounding the decimal period. If the $\langle digits \rangle$ are non-zero, the number is not an integer. Otherwise, check the parity of the $\langle digit \rangle$ and return c_zero or $-\text{c_one}$.

```

15405 \cs_new:Npn \__fp_pow_neg_case:w \s__fp \__fp_chk:w #1#2#3;
15406 {
15407     \if_case:w #1 \exp_stop_f:
15408     -\c_one
15409     \or: \__fp_pow_neg_case_aux:nnnnn #3

```

```

15410     \else: \c_one
15411     \fi:
15412 }
15413 \cs_new:Npn \__fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
15414 {
15415     \if_int_compare:w #1 > \c_eight
15416     \if_int_compare:w #1 > \c_sixteen
15417     -\c_one
15418     \else:
15419     \exp_after:wN \exp_after:wN
15420     \exp_after:wN \__fp_pow_neg_case_aux:NNNNNNNNw
15421     \prg_replicate:nn { \c_sixteen - #1 } { 0 } #4#5 ;
15422     \fi:
15423     \else:
15424     \if_int_compare:w #1 > \c_zero
15425     \if_int_compare:w #4#5 = \c_zero
15426     \exp_after:wN \exp_after:wN
15427     \exp_after:wN \__fp_pow_neg_case_aux:NNNNNNNNw
15428     \prg_replicate:nn { \c_eight - #1 } { 0 } #2#3 ;
15429     \else:
15430     \c_one
15431     \fi:
15432     \else:
15433     \c_one
15434     \fi:
15435     \fi:
15436 }
15437 \cs_new:Npn \__fp_pow_neg_case_aux:NNNNNNNNw #1#2#3#4#5#6#7#8#9;
15438 {
15439     \if_int_compare:w 0 #9 = \c_zero
15440     \if_int_odd:w #8 \exp_stop_f:
15441     \c_zero
15442     \else:
15443     -\c_one
15444     \fi:
15445     \else:
15446     \c_one
15447     \fi:
15448 }

```

(End definition for __fp_pow_neg_case:w, __fp_pow_neg_case_aux:nnnnn, and __fp_pow_neg_case_aux:NNNNNNNNw.)

```

15449 </initex | package>

```

30 13fp-trig Implementation

```

15450 <*initex | package>

```

```

15451 <@@=fp>

```

30.1 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 (± 0 , $\pm \text{inf}$ 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.

30.1.1 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 ± 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` will be called to compute the Taylor series: this function receives a sign `#3`, an initial octant of 0, and the function `__fp_ep_to_float:wwN` which converts the result of the series to a floating point directly rather than taking its inverse, since $\sin(x) = \#3 \sin|x|$.

```

15452 \cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15453 {
15454   \if_case:w #2 \exp_stop_f:
15455     \__fp_case_return_same_o:w
15456   \or: \__fp_case_use:nw
15457     {
15458       \__fp_trig:NNNNwn #1 \__fp_sin_series_o:NNwww
15459       \__fp_ep_to_float:wwN #3 \c_zero
15460     }
15461   \or: \__fp_case_use:nw
15462     { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
15463   \else: \__fp_case_return_same_o:w
15464   \fi:
15465   \s__fp \__fp_chk:w #2 #3 #4;
15466 }

```

(End definition for `__fp_sin_o:w`.)

`__fp_cos_o:w` The cosine of ± 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 will 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 $\cos(x) = +\sin(\pi/2 + |x|)$.

```

15467 \cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
15468 {
15469   \if_case:w #2 \exp_stop_f:

```

```

15470     \__fp_case_return_o:Nw \c_one_fp
15471 \or:   \__fp_case_use:nw
15472       {
15473         \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
15474         \__fp_ep_to_float:wwN 0 \c_two
15475       }
15476 \or:   \__fp_case_use:nw
15477       { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
15478 \else: \__fp_case_return_same_o:w
15479 \fi:
15480 \s__fp \__fp_chk:w #2 #3;
15481 }

```

(End definition for __fp_cos_o:w.)

__fp_csc_o:w The cosecant of ± 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 $\csc(x) = \#3(\sin|x|)^{-1}$.

```

15482 \cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15483 {
15484   \if_case:w #2 \exp_stop_f:
15485     \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
15486   \or:   \__fp_case_use:nw
15487         {
15488           \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
15489           \__fp_ep_inv_to_float:wwN #3 \c_zero
15490         }
15491   \or:   \__fp_case_use:nw
15492         { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
15493   \else: \__fp_case_return_same_o:w
15494   \fi:
15495   \s__fp \__fp_chk:w #2 #3 #4;
15496 }

```

(End definition for __fp_csc_o:w.)

__fp_sec_o:w The secant of ± 0 is 1. The secant of $\pm\infty$ raises an invalid operation exception. The secant of 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 $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```

15497 \cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
15498 {
15499   \if_case:w #2 \exp_stop_f:
15500     \__fp_case_return_o:Nw \c_one_fp
15501   \or:   \__fp_case_use:nw
15502         {
15503           \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
15504           \__fp_ep_inv_to_float:wwN 0 \c_two
15505         }
15506   \or:   \__fp_case_use:nw

```

```

15507         { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
15508     \else: \__fp_case_return_same_o:w
15509     \fi:
15510     \s__fp \__fp_chk:w #2 #3;
15511 }

```

(End definition for __fp_sec_o:w.)

__fp_tan_o:w The tangent of ± 0 or 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:NNwww`, 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.

```

15512 \cs_new:Npn \__fp_tan_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15513 {
15514     \if_case:w #2 \exp_stop_f:
15515         \__fp_case_return_same_o:w
15516     \or: \__fp_case_use:nw
15517         {
15518             \__fp_trig:NNNNNwn #1
15519             \__fp_tan_series_o:NNwww 0 #3 \c_one
15520         }
15521     \or: \__fp_case_use:nw
15522         { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } }
15523     \else: \__fp_case_return_same_o:w
15524     \fi:
15525     \s__fp \__fp_chk:w #2 #3 #4;
15526 }

```

(End definition for __fp_tan_o:w.)

__fp_cot_o:w The cotangent of ± 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.

```

15527 \cs_new:Npn \__fp_cot_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15528 {
15529     \if_case:w #2 \exp_stop_f:
15530         \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
15531     \or: \__fp_case_use:nw
15532         {
15533             \__fp_trig:NNNNNwn #1
15534             \__fp_tan_series_o:NNwww 2 #3 \c_three
15535         }
15536     \or: \__fp_case_use:nw
15537         { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
15538     \else: \__fp_case_return_same_o:w
15539     \fi:
15540     \s__fp \__fp_chk:w #2 #3 #4;
15541 }
15542 \cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:

```

```

15543 {
15544   \fi:
15545   \token_if_eq_meaning:NNTF 0 #1
15546   { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_inf_fp }
15547   { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
15548   {#2}
15549 }

```

(End definition for __fp_cot_o:w and __fp_cot_zero_o:Nfw.)

30.1.2 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:wN or __fp_ep_inv_to_float: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 ≥ 1 in radians or ≥ 10 in degrees, and `small` otherwise. All four `trig`/`trigd` auxiliaries receive the operand as an extended-precision number.

```

15550 \cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
15551 {
15552   \exp_after:wN #2
15553   \exp_after:wN #3
15554   \exp_after:wN #4
15555   \__int_value:w \__int_eval:w #5
15556   \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
15557   \if_int_compare:w #7 > #1 \c_zero \c_one
15558   #1 \__fp_trig_large:ww \__fp_trigd_large:ww
15559   \else:
15560     #1 \__fp_trig_small:ww \__fp_trigd_small:ww
15561   \fi:
15562   #7,#8{0000}{0000};
15563 }

```

(End definition for __fp_trig:NNNNNwn.)

30.1.3 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 will be 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.

```

15564 \cs_new:Npn \__fp_trig_small:ww #1,#2;
15565 { \__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.

```

15566 \cs_new:Npn \__fp_trigd_small:ww #1,#2;
15567 {
15568   \__fp_ep_mul_raw:wwwN
15569   -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
15570   \__fp_trig_small:ww
15571 }

```

(End definition for __fp_trigd_small:ww.)

30.1.4 Argument reduction in degrees

__fp_trigd_large:ww 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 $\langle block_1 \rangle + \langle block_2 \rangle \pmod{9}$, 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 __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).

```

15572 \cs_new:Npn \__fp_trigd_large:ww #1, #2#3#4#5#6#7;
15573 {
15574   \exp_after:wN \__fp_pack_eight:wNNNNNNNN
15575   \exp_after:wN \__fp_pack_eight:wNNNNNNNN
15576   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
15577   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
15578   \exp_after:wN \__fp_trigd_large_auxi:nnnnwNNNN
15579   \exp_after:wN ;
15580   \exp:w \exp_end_continue_f:w
15581   \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
15582   #2#3#4#5#6#7 0000 0000 0000 !
15583 }
15584 \cs_new:Npn \__fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
15585 {
15586   \exp_after:wN \__fp_trigd_large_auxii:wNw
15587   \__int_value:w \__int_eval:w #1 + #2
15588   - (#1 + #2 - \c_four) / \c_nine * \c_nine \__int_eval_end:
15589   #3;

```

```

15590     #4; #5{#6#7#8#9};
15591   }
15592   \cs_new:Npn \__fp_trigd_large_auxii:wNw #1; #2#3;
15593   {
15594     + (#1#2 - \c_four) / \c_nine * \c_two
15595     \exp_after:wN \__fp_trigd_large_auxiii:www
15596     \__int_value:w \__int_eval:w #1#2
15597     - (#1#2 - \c_four) / \c_nine * \c_nine \__int_eval_end: #3 ;
15598   }
15599   \cs_new:Npn \__fp_trigd_large_auxiii:www #1; #2; #3!
15600   {
15601     \if_int_compare:w #1 < 4500 \exp_stop_f:
15602     \exp_after:wN \__fp_use_i_until_s:nw
15603     \exp_after:wN \__fp_fixed_continue:wn
15604     \else:
15605       + \c_one
15606     \fi:
15607     \__fp_fixed_sub:wnw {9000}{0000}{0000}{0000}{0000}{0000};
15608     {#1}#2{0000}{0000};
15609     { \__fp_trigd_small:ww 2, }
15610   }

```

(End definition for `__fp_trigd_large:ww` and others.)

30.1.5 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 2π , 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 2π must be subtracted, hence we must work with a very good approximation of 2π in order to get a sensible remainder modulo 2π .

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.

`__fp_trig_inverse_two_pi:` This macro expands to `,,!` or `,!` followed by 10112 decimals of $10^{-16}/(2\pi)$. The number of decimals we really need is the maximum exponent plus the number of digits we will need later, 52, plus 12 ($4 - 1$ groups of 4 digits). We store the decimals as a control sequence name, and convert it to a token list when required: strings take up less memory than their token list representation.

```

15611   \cs_new:Npx \__fp_trig_inverse_two_pi:
15612   {

```

15613 \exp_not:n { \exp_after:wN \use_none:n \token_to_str:N }
15614 \cs:w , , !
15615 0000000000000000159154943091895335768883763372514362034459645740 ~
15616 4564487476673440588967976342265350901138027662530859560728427267 ~
15617 5795803689291184611457865287796741073169983922923996693740907757 ~
15618 3077746396925307688717392896217397661693362390241723629011832380 ~
15619 1142226997557159404618900869026739561204894109369378440855287230 ~
15620 9994644340024867234773945961089832309678307490616698646280469944 ~
15621 8652187881574786566964241038995874139348609983868099199962442875 ~
15622 5851711788584311175187671605465475369880097394603647593337680593 ~
15623 0249449663530532715677550322032477781639716602294674811959816584 ~
15624 0606016803035998133911987498832786654435279755070016240677564388 ~
15625 849571310880122199376147681377647378906330680464579784817613124 ~
15626 2731406996077502450029775985708905690279678513152521001631774602 ~
15627 0924811606240561456203146484089248459191435211575407556200871526 ~
15628 6068022171591407574745827225977462853998751553293908139817724093 ~
15629 5825479707332871904069997590765770784934703935898280871734256403 ~
15630 6689511662545705943327631268650026122717971153211259950438667945 ~
15631 0376255608363171169525975812822494162333431451061235368785631136 ~
15632 3669216714206974696012925057833605311960859450983955671870995474 ~
15633 6510431623815517580839442979970999505254387566129445883306846050 ~
15634 7852915151410404892988506388160776196993073410389995786918905980 ~
15635 9373777206187543222718930136625526123878038753888110681406765434 ~
15636 0828278526933426799556070790386060352738996245125995749276297023 ~
15637 5940955843011648296411855777124057544494570217897697924094903272 ~
15638 9477021664960356531815354400384068987471769158876319096650696440 ~
15639 4776970687683656778104779795450353395758301881838687937766124814 ~
15640 9530599655802190835987510351271290432315804987196868777594656634 ~
15641 6221034204440855497850379273869429353661937782928735937843470323 ~
15642 0237145837923557118636341929460183182291964165008783079331353497 ~
15643 7909974586492902674506098936890945883050337030538054731232158094 ~
15644 3197676032283131418980974982243833517435698984750103950068388003 ~
15645 9786723599608024002739010874954854787923568261139948903268997427 ~
15646 0834961149208289037767847430355045684560836714793084567233270354 ~
15647 8539255620208683932409956221175331839402097079357077496549880868 ~
15648 6066360968661967037474542102831219251846224834991161149566556037 ~
15649 9696761399312829960776082779901007830360023382729879085402387615 ~
15650 5744543092601191005433799838904654921248295160707285300522721023 ~
15651 6017523313173179759311050328155109373913639645305792607180083617 ~
15652 9548767246459804739772924481092009371257869183328958862839904358 ~
15653 6866663975673445140950363732719174311388066383072592302759734506 ~
15654 0548212778037065337783032170987734966568490800326988506741791464 ~
15655 6835082816168533143361607309951498531198197337584442098416559541 ~
15656 5225064339431286444038388356150879771645017064706751877456059160 ~
15657 8716857857939226234756331711132998655941596890719850688744230057 ~
15658 5191977056900382183925622033874235362568083541565172971088117217 ~
15659 9593683256488518749974870855311659830610139214454460161488452770 ~
15660 2511411070248521739745103866736403872860099674893173561812071174 ~
15661 0478899368886556923078485023057057144063638632023685201074100574 ~
15662 8592281115721968003978247595300166958522123034641877365043546764 ~
15663 6456565971901123084767099309708591283646669191776938791433315566 ~
15664 5066981321641521008957117286238426070678451760111345080069947684 ~
15665 2235698962488051577598095339708085475059753626564903439445420581 ~
15666 7886435683042000315095594743439252544850674914290864751442303321 ~

15667 3324569511634945677539394240360905438335528292434220349484366151 ~
15668 4663228602477666660495314065734357553014090827988091478669343492 ~
15669 2737602634997829957018161964321233140475762897484082891174097478 ~
15670 2637899181699939487497715198981872666294601830539583275209236350 ~
15671 6853889228468247259972528300766856937583659722919824429747406163 ~
15672 8183113958306744348516928597383237392662402434501997809940402189 ~
15673 6134834273613676449913827154166063424829363741850612261086132119 ~
15674 9863346284709941839942742955915628333990480382117501161211667205 ~
15675 1912579303552929241134403116134112495318385926958490443846807849 ~
15676 0973982808855297045153053991400988698840883654836652224668624087 ~
15677 2540140400911787421220452307533473972538149403884190586842311594 ~
15678 6322744339066125162393106283195323883392131534556381511752035108 ~
15679 7459558201123754359768155340187407394340363397803881721004531691 ~
15680 8295194879591767395417787924352761740724605939160273228287946819 ~
15681 3649128949714953432552723591659298072479985806126900733218844526 ~
15682 7943350455801952492566306204876616134365339920287545208555344144 ~
15683 0990512982727454659118132223284051166615650709837557433729548631 ~
15684 2041121716380915606161165732000083306114606181280326258695951602 ~
15685 4632166138576614804719932707771316441201594960110632830520759583 ~
15686 4850305079095584982982186740289838551383239570208076397550429225 ~
15687 9847647071016426974384504309165864528360324933604354657237557916 ~
15688 1366324120457809969715663402215880545794313282780055246132088901 ~
15689 8742121092448910410052154968097113720754005710963406643135745439 ~
15690 9159769435788920793425617783022237011486424925239248728713132021 ~
15691 7667360756645598272609574156602343787436291321097485897150713073 ~
15692 9104072643541417970572226547980381512759579124002534468048220261 ~
15693 7342299001020483062463033796474678190501811830375153802879523433 ~
15694 4195502135689770912905614317878792086205744999257897569018492103 ~
15695 2420647138519113881475640209760554895793785141404145305151583964 ~
15696 2823265406020603311891586570272086250269916393751527887360608114 ~
15697 5569484210322407772727421651364234366992716340309405307480652685 ~
15698 0930165892136921414312937134106157153714062039784761842650297807 ~
15699 8606266969960809184223476335047746719017450451446166382846208240 ~
15700 8673595102371302904443779408535034454426334130626307459513830310 ~
15701 2293146934466832851766328241515210179422644395718121717021756492 ~
15702 1964449396532222187658488244511909401340504432139858628621083179 ~
15703 3939608443898019147873897723310286310131486955212620518278063494 ~
15704 5711866277825659883100535155231665984394090221806314454521212978 ~
15705 9734471488741258268223860236027109981191520568823472398358013366 ~
15706 0683786328867928619732367253606685216856320119489780733958419190 ~
15707 6659583867852941241871821727987506103946064819585745620060892122 ~
15708 8416394373846549589932028481236433466119707324309545859073361878 ~
15709 6290631850165106267576851216357588696307451999220010776676830946 ~
15710 9814975622682434793671310841210219520899481912444048751171059184 ~
15711 4139907889455775184621619041530934543802808938628073237578615267 ~
15712 7971143323241969857805637630180884386640607175368321362629671224 ~
15713 2609428540110963218262765120117022552929289655594608204938409069 ~
15714 0760692003954646191640021567336017909631872891998634341086903200 ~
15715 5796637103128612356988817640364252540837098108148351903121318624 ~
15716 7228181050845123690190646632235938872454630737272808789830041018 ~
15717 9485913673742589418124056729191238003306344998219631580386381054 ~
15718 2457893450084553280313511884341007373060595654437362488771292628 ~
15719 9807423539074061786905784443105274262641767830058221486462289361 ~
15720 9296692992033046693328438158053564864073184440599549689353773183 ~

15721 6726613130108623588021288043289344562140479789454233736058506327 ~
15722 0439981932635916687341943656783901281912202816229500333012236091 ~
15723 8587559201959081224153679499095448881099758919890811581163538891 ~
15724 6339402923722049848375224236209100834097566791710084167957022331 ~
15725 7897107102928884897013099533995424415335060625843921452433864640 ~
15726 3432440657317477553405404481006177612569084746461432976543900008 ~
15727 3826521145210162366431119798731902751191441213616962045693602633 ~
15728 6102355962140467029012156796418735746835873172331004745963339773 ~
15729 2477044918885134415363760091537564267438450166221393719306748706 ~
15730 2881595464819775192207710236743289062690709117919412776212245117 ~
15731 2354677115640433357720616661564674474627305622913332030953340551 ~
15732 3841718194605321501426328000879551813296754972846701883657425342 ~
15733 5016994231069156343106626043412205213831587971115075454063290657 ~
15734 0248488648697402872037259869281149360627403842332874942332178578 ~
15735 7750735571857043787379693402336902911446961448649769719434527467 ~
15736 4429603089437192540526658890710662062575509930379976658367936112 ~
15737 8137451104971506153783743579555867972129358764463093757203221320 ~
15738 2460565661129971310275869112846043251843432691552928458573495971 ~
15739 5042565399302112184947232132380516549802909919676815118022483192 ~
15740 5127372199792134331067642187484426215985121676396779352982985195 ~
15741 8545392106957880586853123277545433229161989053189053725391582222 ~
15742 9232597278133427818256064882333760719681014481453198336237910767 ~
15743 1255017528826351836492103572587410356573894694875444694018175923 ~
15744 0609370828146501857425324969212764624247832210765473750568198834 ~
15745 5641035458027261252285503154325039591848918982630498759115406321 ~
15746 0354263890012837426155187877318375862355175378506956599570028011 ~
15747 5841258870150030170259167463020842412449128392380525772514737141 ~
15748 2310230172563968305553583262840383638157686828464330456805994018 ~
15749 7001071952092970177990583216417579868116586547147748964716547948 ~
15750 8312140431836079844314055731179349677763739898930227765607058530 ~
15751 4083747752640947435070395214524701683884070908706147194437225650 ~
15752 2823145872995869738316897126851939042297110721350756978037262545 ~
15753 8141095038270388987364516284820180468288205829135339013835649144 ~
15754 3004015706509887926715417450706686888783438055583501196745862340 ~
15755 8059532724727843829259395771584036885940989939255241688378793572 ~
15756 7967951654076673927031256418760962190243046993485989199060012977 ~
15757 7469214532970421677817261517850653008552559997940209969455431545 ~
15758 2745856704403686680428648404512881182309793496962721836492935516 ~
15759 2029872469583299481932978335803459023227052612542114437084359584 ~
15760 9443383638388317751841160881711251279233374577219339820819005406 ~
15761 3292937775306906607415304997682647124407768817248673421685881509 ~
15762 9133422075930947173855159340808957124410634720893194912880783576 ~
15763 3115829400549708918023366596077070927599010527028150868897828549 ~
15764 4340372642729262103487013992868853550062061514343078665396085995 ~
15765 0058714939141652065302070085265624074703660736605333805263766757 ~
15766 2018839497277047222153633851135483463624619855425993871933367482 ~
15767 0422097449956672702505446423243957506869591330193746919142980999 ~
15768 3424230550172665212092414559625960554427590951996824313084279693 ~
15769 7113207021049823238195747175985519501864630940297594363194450091 ~
15770 9150616049228764323192129703446093584259267276386814363309856853 ~
15771 2786024332141052330760658841495858718197071242995959226781172796 ~
15772 4438853796763139274314227953114500064922126500133268623021550837 ~
15773 \cs_end:
15774 }

(End definition for `_fp_trig_inverse_two_pi:`.)

The exponent #1 is between 1 and 10000. We discard the integer part of $10^{\#1-16}/(2\pi)$, that is, the first #1 digits of $10^{-16}/(2\pi)$, because it yields an integer contribution to $x/(2\pi)$. The `auxii` auxiliary discards 64 digits at a time thanks to spaces inserted in the result of `_fp_trig_inverse_two_pi:`, while `auxiii` discards 8 digits at a time, and `auxiv` discards digits one at a time. Then 64 digits are packed into groups of 4 and the `auxv` auxiliary is called.

```

15775 \cs_new:Npn \_fp_trig_large:ww #1, #2#3#4#5#6;
15776 {
15777   \exp_after:wN \_fp_trig_large_auxi:wwwww
15778   \__int_value:w \__int_eval:w (#1 - 32) / 64 \exp_after:wN ,
15779   \__int_value:w \__int_eval:w (#1 - 4) / 8 \exp_after:wN ,
15780   \__int_value:w #1 \_fp_trig_inverse_two_pi: ;
15781   {#2}{#3}{#4}{#5} ;
15782 }
15783 \cs_new:Npn \_fp_trig_large_auxi:wwwww #1, #2, #3, #4!
15784 {
15785   \prg_replicate:nn {#1} { \_fp_trig_large_auxii:ww }
15786   \prg_replicate:nn { #2 - #1 * \c_eight }
15787   { \_fp_trig_large_auxiii:wNNNNNNNN }
15788   \prg_replicate:nn { #3 - #2 * \c_eight }
15789   { \_fp_trig_large_auxiv:wN }
15790   \prg_replicate:nn { \c_eight } { \_fp_pack_twice_four:wNNNNNNNN }
15791   \_fp_trig_large_auxv:www
15792   ;
15793 }
15794 \cs_new:Npn \_fp_trig_large_auxii:ww #1; #2 ~ { #1; }
15795 \cs_new:Npn \_fp_trig_large_auxiii:wNNNNNNNN
15796   #1; #2#3#4#5#6#7#8#9 { #1; }
15797 \cs_new:Npn \_fp_trig_large_auxiv:wN #1; #2 { #1; }

```

(End definition for `_fp_trig_large:ww` and others.)

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 some 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.

```

15798 \cs_new:Npn \_fp_trig_large_auxv:www #1; #2; #3;
15799 {
15800   \exp_after:wN \_fp_use_i_until_s:nw
15801   \exp_after:wN \_fp_trig_large_auxvii:w
15802   \__int_value:w \__int_eval:w \c_fp_leading_shift_int
15803   \prg_replicate:nn { \c_thirteen }
15804   { \_fp_trig_large_auxvi:wNNNNNNNN }

```

```

15805         + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
15806         \__fp_use_i_until_s:nw
15807         ; #3 #1 ; ;
15808     }
15809 \cs_new:Npn \__fp_trig_large_auxvi:wnnnnnnnn #1; #2#3#4#5#6#7#8#9
15810 {
15811     \exp_after:wN \__fp_trig_large_pack:NNNNNw
15812     \__int_value:w \__int_eval:w \c__fp_middle_shift_int
15813     + #2*#9 + #3*#8 + #4*#7 + #5*#6
15814     #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
15815 }
15816 \cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
15817 { + #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.)

```

\__fp_trig_large_auxvii:w
\__fp_trig_large_auxviii:w
\__fp_trig_large_auxix:Nw
\__fp_trig_large_auxx:wNNNNN
\__fp_trig_large_auxxi:w

```

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.

```

15818 \cs_new:Npn \__fp_trig_large_auxvii:w #1#2#3
15819 {
15820     \exp_after:wN \__fp_trig_large_auxviii:ww
15821     \__int_value:w \__int_eval:w (#1#2#3 - 62) / 125 ;
15822     #1#2#3
15823 }
15824 \cs_new:Npn \__fp_trig_large_auxviii:ww #1;
15825 {
15826     + #1
15827     \if_int_odd:w #1 \exp_stop_f:
15828     \exp_after:wN \__fp_trig_large_auxix:Nw
15829     \exp_after:wN -
15830     \else:
15831     \exp_after:wN \__fp_trig_large_auxix:Nw
15832     \exp_after:wN +
15833     \fi:
15834 }
15835 \cs_new:Npn \__fp_trig_large_auxix:Nw
15836 {
15837     \exp_after:wN \__fp_use_i_until_s:nw
15838     \exp_after:wN \__fp_trig_large_auxxi:w
15839     \__int_value:w \__int_eval:w \c__fp_leading_shift_int
15840     \prg_replicate:nn { \c_thirteen }
15841     { \__fp_trig_large_auxx:wNNNNN }
15842     + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
15843     ;
15844 }

```

```

15845 \cs_new:Npn \__fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
15846 {
15847   \exp_after:wN \__fp_trig_large_pack:NNNNNw
15848   \__int_value:w \__int_eval:w \c__fp_middle_shift_int
15849   #2 \c_eight * #3#4#5#6
15850   #1; #2
15851 }
15852 \cs_new:Npn \__fp_trig_large_auxxi:w #1;
15853 {
15854   \exp_after:wN \__fp_ep_mul_raw:wwwN
15855   \__int_value:w \__int_eval:w \c_zero \__fp_ep_to_ep_loop:N #1 ; ; !
15856   0,{7853}{9816}{3397}{4483}{0961}{5661};
15857   \__fp_trig_small:ww
15858 }

```

(End definition for __fp_trig_large_auxvii:w and others.)

30.1.6 Computing the power series

```

\__fp_sin_series_o:NNwww
\__fp_sin_series_aux_o:NNwww

```

Here we receive a conversion function __fp_ep_to_float:wwN or __fp_ep_inv_to_float:wwN, a *sign* (0 or 2), a (non-negative) *octant* delimited by a dot, a *fixed point* 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 will control 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.

```

15859 \cs_new:Npn \__fp_sin_series_o:NNwww #1#2#3. #4;
15860 {
15861   \__fp_fixed_mul:wwn #4; #4;
15862   {
15863     \exp_after:wN \__fp_sin_series_aux_o:NNwww
15864     \exp_after:wN #1
15865     \__int_value:w
15866     \if_int_odd:w \__int_eval:w (#3 + \c_two) / \c_four \__int_eval_end:
15867     #2

```

```

15868         \else:
15869             \if_meaning:w #2 0 2 \else: 0 \fi:
15870         \fi:
15871     {#3}
15872 }
15873 }
15874 \cs_new:Npn \__fp_sin_series_aux_o:NNwww #1#2#3 #4; #5,#6;
15875 {
15876     \if_int_odd:w \__int_eval:w #3 / \c_two \__int_eval_end:
15877     \exp_after:wN \use_i:nn
15878     \else:
15879     \exp_after:wN \use_ii:nn
15880     \fi:
15881     { % 1/18!
15882         \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0001}{5619}{2070};
15883         #4;{0000}{0000}{0000}{0477}{9477}{3324};
15884         \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0011}{4707}{4559}{7730};
15885         \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{2087}{6756}{9878}{6810};
15886         \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0027}{5573}{1922}{3985}{8907};
15887         \__fp_fixed_mul_sub_back:wwwn #4;{0000}{2480}{1587}{3015}{8730}{1587};
15888         \__fp_fixed_mul_sub_back:wwwn #4;{0013}{8888}{8888}{8888}{8888}{8889};
15889         \__fp_fixed_mul_sub_back:wwwn #4;{0416}{6666}{6666}{6666}{6666}{6667};
15890         \__fp_fixed_mul_sub_back:wwwn #4;{5000}{0000}{0000}{0000}{0000}{0000};
15891         \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
15892         { \__fp_fixed_continue:wn 0, }
15893     }
15894     { % 1/17!
15895         \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0028}{1145}{7254};
15896         #4;{0000}{0000}{0000}{7647}{1637}{3182};
15897         \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
15898         \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
15899         \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
15900         \__fp_fixed_mul_sub_back:wwwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
15901         \__fp_fixed_mul_sub_back:wwwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
15902         \__fp_fixed_mul_sub_back:wwwn #4;{1666}{6666}{6666}{6666}{6666}{6667};
15903         \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
15904         { \__fp_ep_mul:wwwn 0, } #5,#6;
15905     }
15906     {
15907         \exp_after:wN \__fp_sanitiz:Nw
15908         \exp_after:wN #2
15909         \__int_value:w \__int_eval:w #1
15910     }
15911     #2
15912 }

```

(End definition for __fp_sin_series_o:NNwww and __fp_sin_series_aux_o:NNwww.)

__fp_tan_series_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

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2 a_5))))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2 b_5))))}.$$

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.

```

15913 \cs_new:Npn \__fp_tan_series_o:NNwww #1#2#3. #4;
15914 {
15915   \__fp_fixed_mul:wwn #4; #4;
15916   {
15917     \exp_after:wN \__fp_tan_series_aux_o:Nnwww
15918     \__int_value:w
15919     \if_int_odd:w \__int_eval:w #3 / \c_two \__int_eval_end:
15920     \exp_after:wN \reverse_if:N
15921     \fi:
15922     \if_meaning:w #1#2 2 \else: 0 \fi:
15923   }#3}
15924 }
15925 }
15926 \cs_new:Npn \__fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
15927 {
15928   \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{1527}{3493}{0856}{7059};
15929   #3; {0000}{0159}{6080}{0274}{5257}{6472};
15930   \__fp_fixed_mul_sub_back:wwwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
15931   \__fp_fixed_mul_sub_back:wwwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
15932   \__fp_fixed_mul_sub_back:wwwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
15933   \__fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
15934   { \__fp_ep_mul:wwwn 0, } #4,#5;
15935   {
15936     \__fp_fixed_mul_sub_back:wwwn {0000}{0007}{0258}{0681}{9408}{4706};
15937     #3; {0000}{2343}{7175}{1399}{6151}{7670};
15938     \__fp_fixed_mul_sub_back:wwwn #3; {0019}{2638}{4588}{9232}{8861}{3691};
15939     \__fp_fixed_mul_sub_back:wwwn #3; {0536}{6357}{0691}{4344}{6852}{4252};
15940     \__fp_fixed_mul_sub_back:wwwn #3; {5263}{1578}{9473}{6842}{1052}{6315};
15941     \__fp_fixed_mul_sub_back:wwwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
15942     {
15943       \reverse_if:N \if_int_odd:w
15944       \__int_eval:w (#2 - \c_one) / \c_two \__int_eval_end:
15945       \exp_after:wN \__fp_reverse_args:Nww
15946       \fi:
15947       \__fp_ep_div:wwwn 0,
15948     }
15949   }
15950   {
15951     \exp_after:wN \__fp_sanitize:Nw
15952     \exp_after:wN #1
15953     \__int_value:w \__int_eval:w \__fp_ep_to_float:wwN

```

```

15954     }
15955     #1
15956 }

```

(End definition for `_fp_tan_series_o:Nnwww` and `_fp_tan_series_aux_o:Nnwww`.)

30.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arc-cosecant, 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

$$\arccos x = \text{atan}(\sqrt{1 - x^2}, x) \quad (5)$$

$$\arcsin x = \text{atan}(x, \sqrt{1 - x^2}) \quad (6)$$

$$\text{asec } x = \text{atan}(\sqrt{x^2 - 1}, 1) \quad (7)$$

$$\text{acsc } x = \text{atan}(1, \sqrt{x^2 - 1}) \quad (8)$$

$$\text{atan } x = \text{atan}(x, 1) \quad (9)$$

$$\text{acot } x = \text{atan}(1, x). \quad (10)$$

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 will give 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 $\text{atan } \frac{|y|}{x} = \frac{\pi}{4} + \text{atan } \frac{-x+|y|}{x+|y|}$;

3 $0 < x < 0.41421|y|$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{2} - \text{atan } \frac{x}{|y|}$;

4 $0 < -x < 0.41421|y|$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{2} + \text{atan } \frac{-x}{|y|}$;

5 $0 < 0.41421|y| < -x < |y|$, then $\text{atan } \frac{|y|}{x} = \frac{3\pi}{4} - \text{atan } \frac{x+|y|}{-x+|y|}$;

6 $0 < -0.41421x < |y| < -x$, then $\text{atan } \frac{|y|}{x} = \frac{3\pi}{4} + \text{atan } \frac{-x-|y|}{-x+|y|}$;

7 $0 < |y| < -0.41421x$, then $\operatorname{atan} \frac{|y|}{x} = \pi - \operatorname{atan} \frac{|y|}{-x}$.

In the following, we will 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.

30.2.1 Arctangent and arccotangent

`__fp_atan_o:Nw`
`__fp_acot_o:Nw`
`__fp_atan_dispatch_o:NNnNw`

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. Here, we dispatch according to the number of arguments. The one-argument versions of arctangent and arccotangent are special cases of the two-argument ones: $\operatorname{atan}(y) = \operatorname{atan}(y, 1) = \operatorname{acot}(1, y)$ and $\operatorname{acot}(x) = \operatorname{atan}(1, x) = \operatorname{acot}(x, 1)$.

```

15957 \cs_new:Npn \__fp_atan_o:Nw
15958 {
15959   \__fp_atan_dispatch_o:NNnNw
15960   \__fp_acotii_o:Nww \__fp_atanii_o:Nww { atan }
15961 }
15962 \cs_new:Npn \__fp_acot_o:Nw
15963 {
15964   \__fp_atan_dispatch_o:NNnNw
15965   \__fp_atanii_o:Nww \__fp_acotii_o:Nww { acot }
15966 }
15967 \cs_new:Npn \__fp_atan_dispatch_o:NNnNw #1#2#3#4#5@
15968 {
15969   \if_case:w
15970     \__int_eval:w \__fp_array_count:n {#5} - \c_one \__int_eval_end:
15971     \exp_after:wN #1 \exp_after:wN #4 \c_one_fp #5
15972     \exp:w
15973   \or: #2 #4 #5 \exp:w
15974   \else:
15975     \__msg_kernel_expandable_error:nnnnn
15976     { kernel } { fp-num-args } { #3() } { 1 } { 2 }
15977     \exp_after:wN \c_nan_fp \exp:w
15978   \fi:
15979   \exp_after:wN \exp_end:
15980 }

```

(End definition for `__fp_atan_o:Nw`, `__fp_acot_o:Nw`, and `__fp_atan_dispatch_o:NNnNw`.)

`__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 $\{\pm\pi/4, \pm3\pi/4\}$ (in degrees, $\{\pm45, \pm135\}$). 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, ±90), or 0, leading to $\{\pm0, \pm\pi\}$ (in degrees, $\{\pm0, \pm180\}$). Since $\operatorname{acot}(x, y) = \operatorname{atan}(y, x)$, `__fp_acotii_o:ww` simply reverses its two arguments.

```

15981 \cs_new:Npn \__fp_atanii_o:Nww
15982   #1 \s__fp \__fp_chk:w #2#3#4; \s__fp \__fp_chk:w #5
15983 {
15984   \if_meaning:w 3 #2 \__fp_case_return_i_o:ww \fi:
15985   \if_meaning:w 3 #5 \__fp_case_return_ii_o:ww \fi:
15986   \if_case:w

```



```

15987     \if_meaning:w #2 #5
15988     \if_meaning:w 1 #2 \c_ten \else: \c_zero \fi:
15989     \else:
15990     \if_int_compare:w #2 > #5 \c_one \else: \c_two \fi:
15991     \fi:
15992     \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 \c_two }
15993     \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 \c_four }
15994     \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 \c_zero }
15995     \fi:
15996     \__fp_atan_normal_o:NNnwNnw #1
15997     \s__fp \__fp_chk:w #2#3#4;
15998     \s__fp \__fp_chk:w #5
15999   }
16000   \cs_new:Npn \__fp_acotii_o:Nww #1#2; #3;
16001   { \__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 ± 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:NwwwwwN, with arguments the final sign #2; the octant #3; $\operatorname{atan} z/z = 1$ 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, $\operatorname{atan} z$ will be computed to be 0, and the result will be $[\#3/2] \cdot \pi/4$ if the sign #5 of x is positive, and $[(7 - \#3)/2] \cdot \pi/4$ for negative x , where the divisions are rounded up.

```

16002   \cs_new:Npn \__fp_atan_inf_o:NNNw #1#2#3 \s__fp \__fp_chk:w #4#5#6;
16003   {
16004     \exp_after:wN \__fp_atan_combine_o:NwwwwwN
16005     \exp_after:wN #2
16006     \__int_value:w \__int_eval:w
16007     \if_meaning:w 2 #5 \c_seven - \fi: #3 \exp_after:wN ;
16008     \c__fp_one_fixed_tl ;
16009     {0000}{0000}{0000}{0000}{0000}{0000};
16010     0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
16011   }

```

(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 $\operatorname{atan}(x, \sqrt{1 - x^2})$ without intermediate rounding errors.

```

16012   \cs_new_protected:Npn \__fp_atan_normal_o:NNnwNnw
16013   #1 \s__fp \__fp_chk:w 1#2#3#4; \s__fp \__fp_chk:w 1#5#6#7;
16014   {
16015     \__fp_atan_test_o:NwwNwwN
16016     #2 #3, #4{0000}{0000};
16017     #5 #6, #7{0000}{0000}; #1
16018   }

```

(End definition for __fp_atan_normal_o:NNnwNnw.)

`__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 $(\operatorname{atan} z)/z = 1 - \dots$, 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 what z will be, 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.

```

16019 \cs_new:Npn \__fp_atan_test_o:NwwNwwN #1#2,#3; #4#5,#6;
16020 {
16021   \exp_after:wN \__fp_atan_combine_o:NwwwwwN
16022   \exp_after:wN #1
16023   \__int_value:w \__int_eval:w
16024   \if_meaning:w 2 #4
16025     \c_seven - \__int_eval:w
16026   \fi:
16027   \if_int_compare:w
16028     \__fp_ep_compare:www #2,#3; #5,#6; > \c_zero
16029     \c_three -
16030   \exp_after:wN \__fp_reverse_args:Nww
16031   \fi:
16032   \__fp_atan_div:wnwwnw #2,#3; #5,#6;
16033 }

```

(End definition for `__fp_atan_test_o:NwwNwwN`.)

`__fp_atan_div:wnwwnw`
`__fp_atan_near:wwwn`
`__fp_atan_near_aux:wwn`

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 $\operatorname{atan} \frac{b-a}{a+b}$. Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute $\operatorname{atan} \frac{a}{b}$. In any case, call `__fp_atan_auxi:ww` followed by z , as a comma-delimited exponent and a fixed point number.

```

16034 \cs_new:Npn \__fp_atan_div:wnwwnw #1,#2#3; #4,#5#6;
16035 {
16036   \if_int_compare:w
16037     \__int_eval:w 41421 * #5 < #2 000
16038     \if_case:w \__int_eval:w #4 - #1 \__int_eval_end: 00 \or: 0 \fi:
16039     \exp_stop_f:
16040     \exp_after:wN \__fp_atan_near:wwwn
16041   \fi:
16042   \c_zero
16043   \__fp_ep_div:wwwn #1,{#2}#3; #4,{#5}#6;
16044   \__fp_atan_auxi:ww
16045 }
16046 \cs_new:Npn \__fp_atan_near:wwwn
16047   \c_zero \__fp_ep_div:wwwn #1,#2; #3,
16048   {
16049     \c_one
16050     \__fp_ep_to_fixed:wwn #1 - #3, #2;

```

```

16051     \__fp_atan_near_aux:wwn
16052   }
16053   \cs_new:Npn \__fp_atan_near_aux:wwn #1; #2;
16054   {
16055     \__fp_fixed_add:wwn #1; #2;
16056     { \__fp_fixed_sub:wwn #2; #1; { \__fp_ep_div:wwwn 0, } 0, }
16057   }

```

(End definition for __fp_atan_div:wwwn, __fp_atan_near:wwwn, and __fp_atan_near_aux:wwn.)

__fp_atan_auxi:ww Convert z from a representation as an exponent and a fixed point number in $[0.01, 1)$ to a
 __fp_atan_auxii:w 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.

```

16058   \cs_new:Npn \__fp_atan_auxi:ww #1,#2;
16059   { \__fp_ep_to_fixed:wwn #1,#2; \__fp_atan_auxii:w #1,#2; }
16060   \cs_new:Npn \__fp_atan_auxii:w #1;
16061   {
16062     \__fp_fixed_mul:wwn #1; #1;
16063     {
16064       \__fp_atan_Taylor_loop:www 39 ;
16065       {0000}{0000}{0000}{0000}{0000}{0000} ;
16066     }
16067     ! #1;
16068   }

```

(End definition for __fp_atan_auxi:ww and __fp_atan_auxii:w.)

__fp_atan_Taylor_loop:www We compute the series of $(\operatorname{atan} z)/z$. A typical intermediate stage has $\#1 = 2k - 1$,
 __fp_atan_Taylor_break:w $\#2 = \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\dots - z^2\frac{1}{39}))$, 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 $(\operatorname{atan} z)/z$, 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.

```

16069   \cs_new:Npn \__fp_atan_Taylor_loop:www #1; #2; #3;
16070   {
16071     \if_int_compare:w #1 = - \c_one
16072     \__fp_atan_Taylor_break:w
16073     \fi:
16074     \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_t1 ; #1;
16075     \__fp_rrot:www \__fp_fixed_mul_sub_back:wwwn #2; #3;
16076     {
16077       \exp_after:wN \__fp_atan_Taylor_loop:www
16078       \__int_value:w \__int_eval:w #1 - \c_two ;
16079     }
16080     #3;
16081   }
16082   \cs_new:Npn \__fp_atan_Taylor_break:w
16083   \fi: #1 \__fp_fixed_mul_sub_back:wwwn #2; #3 !
16084   { \fi: ; #2 ; }

```

(End definition for __fp_atan_Taylor_loop:www and __fp_atan_Taylor_break:w.)

__fp_atan_combine_o:NwwwwN This receives a $\langle sign \rangle$, an $\langle octant \rangle$, a fixed point value of $(\operatorname{atan} z)/z$, 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 `\use_i:nn` (when working in radians) or `\use_ii:nn` (when working in degrees). The function computes the floating point result

$$\langle sign \rangle \left(\left\lceil \frac{\langle octant \rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant \rangle} \frac{\operatorname{atan} z}{z} \cdot z \right), \quad (11)$$

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 `__fp_sanitize:Nw`, which checks for overflow or underflow. If the octant is 0, leave the exponent `#5` for `__fp_sanitize:Nw`, and multiply `#3 = $\frac{\operatorname{atan} z}{z}$` with `#6`, the adjusted z . Otherwise, multiply `#3 = $\frac{\operatorname{atan} z}{z}$` 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 `__fp_fixed_to_float:wN`.

```

16085 \cs_new:Npn \__fp_atan_combine_o:NwwwwN #1 #2; #3; #4; #5,#6; #7
16086 {
16087   \exp_after:wN \__fp_sanitize:Nw
16088   \exp_after:wN #1
16089   \__int_value:w \__int_eval:w
16090   \if_meaning:w 0 #2
16091     \exp_after:wN \use_i:nn
16092   \else:
16093     \exp_after:wN \use_ii:nn
16094   \fi:
16095   { #5 \__fp_fixed_mul:wwn #3; #6; }
16096   {
16097     \__fp_fixed_mul:wwn #3; #4;
16098     {
16099       \exp_after:wN \__fp_atan_combine_aux:ww
16100       \__int_value:w \__int_eval:w #2 / \c_two ; #2;
16101     }
16102   }
16103   { #7 \__fp_fixed_to_float:wN \__fp_fixed_to_float_rad:wN }
16104   #1
16105 }
16106 \cs_new:Npn \__fp_atan_combine_aux:ww #1; #2;
16107 {
16108   \__fp_fixed_mul_short:wwn
16109   {7853}{9816}{3397}{4483}{0961}{5661};
16110   {#1}{0000}{0000};
16111   {
16112     \if_int_odd:w #2 \exp_stop_f:
16113     \exp_after:wN \__fp_fixed_sub:wwn
16114   \else:
16115     \exp_after:wN \__fp_fixed_add:wwn
16116   \fi:
16117 }
16118 }
```

(End definition for `__fp_atan_combine_o:NwwwwN` and `__fp_atan_combine_aux:ww`.)

30.2.2 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 ± 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).

```

16119 \cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3; @
16120 {
16121   \if_case:w #2 \exp_stop_f:
16122     \__fp_case_return_same_o:w
16123   \or:
16124     \__fp_case_use:nw
16125     { \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
16126   \or:
16127     \__fp_case_use:nw
16128     { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }
16129   \else:
16130     \__fp_case_return_same_o:w
16131   \fi:
16132   \s__fp \__fp_chk:w #2 #3;
16133 }

```

(End definition for `__fp_asin_o:w`.)

`__fp_acos_o:w` The arccosine of ± 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.

```

16134 \cs_new:Npn \__fp_acos_o:w #1 \s__fp \__fp_chk:w #2#3; @
16135 {
16136   \if_case:w #2 \exp_stop_f:
16137     \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 \c_four }
16138   \or:
16139     \__fp_case_use:nw
16140     {
16141       \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
16142       \__fp_reverse_args:Nww
16143     }
16144   \or:
16145     \__fp_case_use:nw
16146     { \__fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
16147   \else:
16148     \__fp_case_return_same_o:w
16149   \fi:
16150   \s__fp \__fp_chk:w #2 #3;
16151 }

```

(End definition for `__fp_acos_o:w`.)

`__fp_asin_normal_o:NfwNnnnnw` If the exponent #5 is strictly less than 1, the operand lies within $(-1, 1)$ and the operation is permitted: call `__fp_asin_auxi_o:nNww` with the appropriate arguments. If the number is exactly ± 1 (the test works because we know that $\#5 \geq 1$, $\#6\#7 \geq 10000000$, $\#8\#9 \geq 0$, with equality only for ± 1), we also call `__fp_asin_auxi_o:nNww`. Otherwise, `__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.

```

16152 \cs_new:Npn \__fp_asin_normal_o:NfwNnnnnw

```

```

16153     #1#2#3 \s__fp \__fp_chk:w 1#4#5#6#7#8#9;
16154 {
16155     \if_int_compare:w #5 < \c_one
16156         \exp_after:wN \__fp_use_none_until_s:w
16157     \fi:
16158     \if_int_compare:w \__int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
16159         \exp_after:wN \__fp_use_none_until_s:w
16160     \fi:
16161     \__fp_use_i:ww
16162     \__fp_invalid_operation_o:fw {#2}
16163     \s__fp \__fp_chk:w 1#4#{#5}{#6}{#7}{#8}{#9};
16164     \__fp_asin_auxi_o:NnNww
16165     #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
16166 }

```

(End definition for `__fp_asin_normal_o:NfwNnnnnw`.)

`__fp_asin_auxi_o:NnNww`
`__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` (`__fp_reverse_args:Nww` in that case) before `__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`.

```

16167 \cs_new:Npn \__fp_asin_auxi_o:NnNww #1#2#3#4,#5;
16168 {
16169     \__fp_ep_to_fixed:wwn #4,#5;
16170     \__fp_asin_isqrt:wn
16171     \__fp_ep_mul:wwwwn #4,#5;
16172     \__fp_ep_to_ep:wwN
16173     \__fp_fixed_continue:wn
16174     { #2 \__fp_atan_test_o:NwwNwwN #3 }
16175     0 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1
16176 }
16177 \cs_new:Npn \__fp_asin_isqrt:wn #1;
16178 {
16179     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl ; #1;
16180     {
16181         \__fp_fixed_add_one:wn #1;
16182         \__fp_fixed_continue:wn { \__fp_ep_mul:wwwwn 0, } 0,
16183     }
16184     \__fp_ep_isqrt:wwn
16185 }

```

(End definition for `__fp_asin_auxi_o:NnNww` and `__fp_asin_isqrt:wn`.)

30.2.3 Arccosecant 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 ± 0 raises

an invalid operation exception. The arccosecant of $\pm\infty$ is ± 0 with the same sign. The arccosecant 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.

```

16186 \cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
16187 {
16188   \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
16189     \__fp_case_use:nw
16190     { \__fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
16191   \or: \__fp_case_use:nw
16192     { \__fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
16193   \or: \__fp_case_return_o:Nw \c_zero_fp
16194   \or: \__fp_case_return_same_o:w
16195   \else: \__fp_case_return_o:Nw \c_minus_zero_fp
16196   \fi:
16197   \s__fp \__fp_chk:w #2 #3 #4;
16198 }

```

(End definition for `__fp_acsc_o:w`.)

`__fp_asec_o:w` The arcsecant of ± 0 raises an invalid operation exception. The arcsecant of $\pm\infty$ is $\pi/2$ (in degrees, 90). The arccosecant 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`.

```

16199 \cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3; @
16200 {
16201   \if_case:w #2 \exp_stop_f:
16202     \__fp_case_use:nw
16203     { \__fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
16204   \or:
16205     \__fp_case_use:nw
16206     {
16207       \__fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
16208       \__fp_reverse_args:Nww
16209     }
16210   \or: \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 \c_four }
16211   \else: \__fp_case_return_same_o:w
16212   \fi:
16213   \s__fp \__fp_chk:w #2 #3;
16214 }

```

(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:nNww` (with all the appropriate arguments). This computes what we want thanks to $\text{acsc}(x) = \text{asin}(1/x)$ and $\text{asec}(x) = \text{acos}(1/x)$.

```

16215 \cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \__fp_chk:w 1#4#5#6;
16216 {
16217   \int_compare:nNnTF {#5} < \c_one
16218   {
16219     \__fp_invalid_operation_o:fw {#2}
16220     \s__fp \__fp_chk:w 1#4{#5}#6;

```

```

16221     }
16222     {
16223         \__fp_ep_div:wwwn
16224         1,{1000}{0000}{0000}{0000}{0000}{0000};
16225         #5,#6{0000}{0000};
16226         { \__fp_asin_auxi_o:NnNww #1 {#3} #4 }
16227     }
16228 }

```

(End definition for __fp_acsc_normal_o:NfwNnw.)

```
16229 </initex | package>
```

31 13fp-convert implementation

```
16230 (*initex | package)
```

```
16231 <@@=fp>
```

31.1 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 will be the dot auxiliary, which removes a trailing dot if any. We then clean-up with the end auxiliary, keeping only the number.

```

16232 \cs_new:Npn \__fp_trim_zeros:w #1 ;
16233 {
16234     \__fp_trim_zeros_loop:w #1
16235     ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__stop
16236 }
16237 \cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
16238 \cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
16239 \cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__stop { #1 }

```

(End definition for __fp_trim_zeros:w and others.)

31.2 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.

```

16240 \cs_new:Npn \fp_to_scientific:N #1
16241 { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
16242 \cs_generate_variant:Nn \fp_to_scientific:N { c }
16243 \cs_new:Npn \fp_to_scientific:n
16244 {
16245     \exp_after:wN \__fp_to_scientific_dispatch:w
16246     \exp:w \exp_end_continue_f:w \__fp_parse:n
16247 }

```

(End definition for \fp_to_scientific:N and \fp_to_scientific:n. These functions are documented on page 184.)


```

\__fp_to_scientific_dispatch: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: ± 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. Finally trim zeros.

```

16248 \cs_new:Npn \__fp_to_scientific_dispatch:w \s__fp \__fp_chk:w #1#2
16249 {
16250   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
16251   \if_case:w #1 \exp_stop_f:
16252     \__fp_case_return:nw { 0 }
16253   \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
16254   \or:
16255     \__fp_case_use:nw
16256     {
16257       \__fp_invalid_operation:nnw
16258       {
16259         \exp_after:wN 1
16260         \exp_after:wN e
16261         \int_use:N \c__fp_max_exponent_int
16262       }
16263       { fp_to_scientific }
16264     }
16265   \or:
16266     \__fp_case_use:nw
16267     {
16268       \__fp_invalid_operation:nnw
16269       { 0 }
16270       { fp_to_scientific }
16271     }
16272   \fi:
16273   \s__fp \__fp_chk:w #1 #2
16274 }
16275 \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
16276 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
16277 {
16278   \if_int_compare:w #2 = \c_one
16279     \exp_after:wN \__fp_to_scientific_normal:wNw
16280   \else:
16281     \exp_after:wN \__fp_to_scientific_normal:wNw
16282     \exp_after:wN e
16283     \__int_value:w \__int_eval:w #2 - \c_one
16284   \fi:
16285   ; #3 #4 #5 #6 ;
16286 }
16287 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
16288 { \__fp_trim_zeros:w #2.#3 ; #1 }

```

(End definition for `__fp_to_scientific_dispatch:w`, `__fp_to_scientific_normal:wnnnnn`, and `__fp_to_scientific_normal:wNw`.)

31.3 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`
`\fp_to_decimal:n`

```
16289 \cs_new:Npn \fp_to_decimal:N #1
16290 { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
16291 \cs_generate_variant:Nn \fp_to_decimal:N { c }
16292 \cs_new:Npn \fp_to_decimal:n
16293 {
16294   \exp_after:wN \__fp_to_decimal_dispatch:w
16295   \exp:w \exp_end_continue_f:w \__fp_parse:n
16296 }
```

(End definition for `\fp_to_decimal:N` and `\fp_to_decimal:n`. These functions are documented on page 183.)

`__fp_to_decimal_dispatch:w` The structure is similar to `__fp_to_scientific_dispatch: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.

```
16297 \cs_new:Npn \__fp_to_decimal_dispatch:w \s__fp \__fp_chk:w #1#2
16298 {
16299   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
16300   \if_case:w #1 \exp_stop_f:
16301     \__fp_case_return:nw { 0 }
16302   \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
16303   \or:
16304     \__fp_case_use:nw
16305     {
16306       \__fp_invalid_operation:nnw
16307       {
16308         \exp_after:wN \exp_after:wN \exp_after:wN 1
16309         \prg_replicate:nn \c__fp_max_exponent_int 0
16310       }
16311       { fp_to_decimal }
16312     }
16313   \or:
16314     \__fp_case_use:nw
16315     {
16316       \__fp_invalid_operation:nnw
16317       { 0 }
16318       { fp_to_decimal }
16319     }
16320   \fi:
16321   \s__fp \__fp_chk:w #1 #2
16322 }
16323 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
16324 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
16325 {
16326   \int_compare:nNnTF {#2} > \c_zero
```

```

16327     {
16328         \int_compare:nNnTF {#2} < \c_sixteen
16329         {
16330             \__fp_decimate:nNnnnn { \c_sixteen - #2 }
16331             \__fp_to_decimal_large:Nnnw
16332         }
16333         {
16334             \exp_after:wN \exp_after:wN
16335             \exp_after:wN \__fp_to_decimal_huge:wnnnn
16336             \prg_replicate:nn { #2 - \c_sixteen } { 0 } ;
16337         }
16338     {#3} {#4} {#5} {#6}
16339 }
16340 {
16341     \exp_after:wN \__fp_trim_zeros:w
16342     \exp_after:wN 0
16343     \exp_after:wN .
16344     \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
16345     #3#4#5#6 ;
16346 }
16347 }
16348 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
16349 {
16350     \exp_after:wN \__fp_trim_zeros:w \__int_value:w
16351     \if_int_compare:w #2 > \c_zero
16352     #2
16353     \fi:
16354     \exp_stop_f:
16355     #3.#4 ;
16356 }
16357 \cs_new:Npn \__fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }

```

(End definition for `__fp_to_decimal_dispatch:w` and others.)

31.4 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:c` dispatch:w.

```

\fp_to_tl:n 16358 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
16359 \cs_generate_variant:Nn \fp_to_tl:N { c }
16360 \cs_new:Npn \fp_to_tl:n
16361 {
16362     \exp_after:wN \__fp_to_tl_dispatch:w
16363     \exp:w \exp_end_continue_f:w \__fp_parse:n
16364 }

```

(End definition for `\fp_to_tl:N` and `\fp_to_tl:n`. These functions are documented on page 184.)

`__fp_to_tl_dispatch:w` 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.

```

16365 \cs_new:Npn \__fp_to_tl_dispatch:w \s__fp \__fp_chk:w #1#2
16366 {

```

```

16367 \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
16368 \if_case:w #1 \exp_stop_f:
16369     \__fp_case_return:nw { 0 }
16370 \or: \exp_after:wN \__fp_to_tl_normal:nnnnn
16371 \or: \__fp_case_return:nw { inf }
16372 \else: \__fp_case_return:nw { nan }
16373 \fi:
16374 }
16375 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
16376 {
16377     \if_int_compare:w #1 > \c_sixteen
16378         \exp_after:wN \__fp_to_scientific_normal:wnnnnn
16379     \else:
16380         \if_int_compare:w #1 < - \c_two
16381             \exp_after:wN \exp_after:wN
16382             \exp_after:wN \__fp_to_scientific_normal:wnnnnn
16383         \else:
16384             \exp_after:wN \exp_after:wN
16385             \exp_after:wN \__fp_to_decimal_normal:wnnnnn
16386         \fi:
16387     \fi:
16388     \s__fp \__fp_chk:w 1 0 {#1}
16389 }

```

(End definition for __fp_to_tl_dispatch:w and __fp_to_tl_normal:nnnnn.)

31.5 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.

31.6 Convert to dimension or integer

\fp_to_dim:N These three public functions rely on \fp_to_decimal:n internally. We make sure to produce pt with category other.

```

\fp_to_dim:c
\fp_to_dim:n
16390 \cs_new:Npn \fp_to_dim:N #1
16391 { \fp_to_decimal:N #1 pt }
16392 \cs_generate_variant:Nn \fp_to_dim:N { c }
16393 \cs_new:Npn \fp_to_dim:n #1
16394 { \fp_to_decimal:n {#1} pt }

```

(End definition for \fp_to_dim:N and \fp_to_dim:n. These functions are documented on page 183.)

\fp_to_int:N These three public functions evaluate their argument, then pass it to \fp_to_int_dispatch:w.

```

\fp_to_int:c
\fp_to_int:n
16395 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
16396 \cs_generate_variant:Nn \fp_to_int:N { c }
16397 \cs_new:Npn \fp_to_int:n
16398 {
16399     \exp_after:wN \__fp_to_int_dispatch:w
16400     \exp:w \exp_end_continue_f:w \__fp_parse:n
16401 }

```

(End definition for `\fp_to_int:N` and `\fp_to_int:n`. These functions are documented on page 184.)

`__fp_to_int_dispatch:w` 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 will be no trailing dot nor zero.

```

16402 \cs_new:Npn \__fp_to_int_dispatch:w #1;
16403 {
16404   \exp_after:wN \__fp_to_decimal_dispatch:w \exp:w \exp_end_continue_f:w
16405   \__fp_round:Nwn \__fp_round_to_nearest:NNN #1; { 0 }
16406 }

```

(End definition for `__fp_to_int_dispatch:w`.)

31.7 Convert from a dimension

`\dim_to_fp:n` 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:` in `\dim_to_fp:n`.

```

16407 \cs_new:Npn \dim_to_fp:n #1
16408 {
16409   \exp_after:wN \__fp_from_dim_test:ww
16410   \exp_after:wN 0
16411   \exp_after:wN ,
16412   \__int_value:w \etex_glueexpr:D #1 ;
16413 }
16414 \cs_new:Npn \__fp_from_dim_test:ww #1, #2
16415 {
16416   \if_meaning:w 0 #2
16417   \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
16418   \else:
16419   \exp_after:wN \__fp_from_dim:wNw
16420   \__int_value:w \__int_eval:w #1 - \c_four
16421   \if_meaning:w - #2
16422   \exp_after:wN , \exp_after:wN 2 \__int_value:w
16423   \else:
16424   \exp_after:wN , \exp_after:wN 0 \__int_value:w #2
16425   \fi:
16426   \fi:
16427 }
16428 \cs_new:Npn \__fp_from_dim:wNw #1,#2#3;
16429 {
16430   \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
16431   #3 000 0000 00 {10}987654321; #2 {#1}
16432 }
16433 \cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
16434 { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
16435 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8

```

```

16436 {
16437   \__fp_mul_npos_o:Nww #7
16438   \s__fp \__fp_chk:w 1 #7 {#5} #1 ;
16439   \s__fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
16440   \prg_do_nothing:
16441 }

```

(End definition for `\dim_to_fp:n` and others. These functions are documented on page 82.)

31.8 Use and eval

\fp_use:N Those public functions are simple copies of the decimal conversions.

```

\fp_use:c 16442 \cs_new_eq:NN \fp_use:N \fp_to_decimal:N
\fp_eval:n 16443 \cs_generate_variant:Nn \fp_use:N { c }
16444 \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 184.)

\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.

```

16445 \cs_new:Npn \fp_abs:n #1
16446 { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }

```

(End definition for `\fp_abs:n`. This function is documented on page 197.)

\fp_max:nn Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, etc.

```

\fp_min:nn 16447 \cs_new:Npn \fp_max:nn #1#2
16448 { \fp_to_decimal:n { max ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
16449 \cs_new:Npn \fp_min:nn #1#2
16450 { \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 197.)

31.9 Convert an array of floating points to a comma list

__fp_array_to_clist:n 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.

```

16451 \cs_new:Npn \__fp_array_to_clist:n #1
16452 {
16453   \tl_if_empty:nF {#1}
16454   {
16455     \__fp_expand:n
16456     {
16457       { \use_ii:nn }

```

```

16458         \_fp_array_to_clist_loop:Nw #1 { ? \_prg_break: } ;
16459         \_prg_break_point:
16460     }
16461 }
16462 }
16463 \cs_new:Npx \_fp_array_to_clist_loop:Nw #1#2;
16464 {
16465     \exp_not:N \use_none:n #1
16466     \exp_not:N \exp_after:wN
16467     {
16468         \exp_not:N \exp_after:wN ,
16469         \exp_not:N \exp_after:wN \c_space_tl
16470         \exp_not:N \exp:w
16471         \exp_not:N \exp_end_continue_f:w
16472         \exp_not:N \_fp_to_tl_dispatch:w #1 #2 ;
16473     }
16474     \exp_not:N \_fp_array_to_clist_loop:Nw
16475 }

```

(End definition for _fp_array_to_clist:n and _fp_array_to_clist_loop:Nw.)

```

16476 </initex | package>

```

32 l3fp-random Implementation

```

16477 <*initex | package>
16478 <@@=fp>

```

32.1 Engine support

At present, X_YTeX, pTeX and upTeX do not provide random numbers, while LuaTeX and pdfTeX provide the primitive `\pdfutex_uniformdeviate:D` (`\pdfuniformdeviate` in pdfTeX and `\uniformdeviate` in LuaTeX). We write the test twice simply in order to write the false branch first.

```

16479 \cs_if_exist:NF \pdfutex_uniformdeviate:D
16480 {
16481     \_msg_kernel_new:nnn { kernel } { fp-no-random }
16482     { Random-numbers-unavailable }
16483     \cs_new:Npn \_fp_rand_o:Nw ? #1 @
16484     {
16485         \_msg_kernel_expandable_error:nn { kernel } { fp-no-random }
16486         \exp_after:wN \c_nan_fp
16487     }
16488     \cs_new_eq:NN \_fp_randint_o:Nw \_fp_rand_o:Nw
16489 }
16490 \cs_if_exist:NT \pdfutex_uniformdeviate:D
16491 {

```

<pre> _fp_rand_uniform: _c_fp_rand_size_int _c_fp_rand_four_int _c_fp_rand_eight_int </pre>	<p>The <code>\pdfutex_uniformdeviate:D</code> primitive gives a pseudo-random integer in a range $[0, n - 1]$ of the user's choice. This number is meant to be uniformly distributed, but is produced by rescaling a uniform pseudo-random integer in $[0, 2^{28} - 1]$. For instance, setting n to (any multiple of) 2^{29} gives only even values. Thus it is only safe to call</p>
--	---

`\pdfTeXuniformdeviate:D` with argument 2^{28} . This integer is also used in the implementation of `\int_rand:nn`. We will also use variants of this number rounded down to multiples of 10^4 and 10^8 .

```

16492 \cs_new:Npn \__fp_rand_uniform:
16493 { \pdfTeXuniformdeviate:D \c__fp_rand_size_int }
16494 \int_const:Nn \c__fp_rand_size_int { 268 435 456 }
16495 \int_const:Nn \c__fp_rand_four_int { 268 430 000 }
16496 \int_const:Nn \c__fp_rand_eight_int { 200 000 000 }

```

(End definition for `__fp_rand_uniform:` and others.)

`__fp_rand_myriads:n` Used as `__fp_rand_myriads:n {XXX}` with one input character per block of four digit
`__fp_rand_myriads_loop:nn` we want. Given a pseudo-random integer from the primitive, we extract 2 blocks of digits
`__fp_rand_myriads_get:w` if possible, namely if the integer is less than 2×10^8 . If that's not possible, we try to
`__fp_rand_myriads_last:` extract 1 block, which succeeds in the range $[2 \times 10^8, 26843 \times 10^4)$. For the 5456 remaining
`__fp_rand_myriads_last:w` possible values we just throw away the random integer and get a new one. Depending on
whether we got 2, 1, or 0 blocks, remove the same number of characters from the input
stream with `\use_i:nnn`, `\use_i:nn` or nothing.

```

16497 \cs_new:Npn \__fp_rand_myriads:n #1
16498 {
16499   \__fp_rand_myriads_loop:nn #1
16500   { ? \use_i_delimit_by_q_stop:nw \__fp_rand_myriads_last: }
16501   { ? \use_none_delimit_by_q_stop:w } \q_stop
16502 }
16503 \cs_new:Npn \__fp_rand_myriads_loop:nn #1#2
16504 {
16505   \use_none:n #2
16506   \exp_after:wN \__fp_rand_myriads_get:w
16507   \__int_value:w \__fp_rand_uniform: ; {#1}{#2}
16508 }
16509 \cs_new:Npn \__fp_rand_myriads_get:w #1 ;
16510 {
16511   \if_int_compare:w #1 < \c__fp_rand_eight_int
16512     \exp_after:wN \use_none:n
16513     \__int_value:w \__int_eval:w
16514     \c__fp_rand_eight_int + #1 \__int_eval_end:
16515     \exp_after:wN \use_i:nnn
16516   \else:
16517     \if_int_compare:w #1 < \c__fp_rand_four_int
16518       \exp_after:wN \use_none:nnnnn
16519       \__int_value:w \__int_eval:w
16520       \c__fp_rand_four_int + #1 \__int_eval_end:
16521       \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
16522     \fi:
16523   \fi:
16524   \__fp_rand_myriads_loop:nn
16525 }
16526 \cs_new:Npn \__fp_rand_myriads_last:
16527 {
16528   \exp_after:wN \__fp_rand_myriads_last:w
16529   \__int_value:w \__fp_rand_uniform: ;
16530 }
16531 \cs_new:Npn \__fp_rand_myriads_last:w #1 ;

```



```

16532 {
16533   \if_int_compare:w #1 < \c__fp_rand_four_int
16534     \exp_after:wN \use_none:nnnnn
16535     \__int_value:w \__int_eval:w
16536     \c__fp_rand_four_int + #1 \__int_eval_end:
16537   \else:
16538     \exp_after:wN \__fp_rand_myriads_last:
16539   \fi:
16540 }

```

(End definition for `__fp_rand_myriads:n` and others.)

32.2 Random floating point

`__fp_rand_o:Nw` First we check that random was called without argument. Then get four blocks of four digits.

```

\__fp_rand_o:
\__fp_rand_o:w
16541 \cs_new:Npn \__fp_rand_o:Nw ? #1 @
16542 {
16543   \tl_if_empty:nTF {#1}
16544     { \__fp_rand_o: }
16545     {
16546       \__msg_kernel_expandable_error:nnnnn
16547       { kernel } { fp-num-args } { rand() } { 0 } { 0 }
16548       \exp_after:wN \c_nan_fp
16549     }
16550   }
16551   \cs_new:Npn \__fp_rand_o:
16552     { \__fp_parse_o:n { . \__fp_rand_myriads:n { xxxx } } }

```

(End definition for `__fp_rand_o:Nw`, `__fp_rand_o:`, and `__fp_rand_o:w`.)

32.3 Random integer

`__fp_randint_o:Nw` Enforce that there is one argument (then add first argument 1) or two arguments. Enforce that they are integers in $(-10^{16}, 10^{16})$ and ordered. We distinguish narrow ranges (less than 2^{28}) from wider ones.

`__fp_randint_e:w` For narrow ranges, compute the number n of possible outputs as an integer using `\fp_to_int:n`, and reduce a pseudo-random 28-bit integer r modulo n . On its own, this is not uniform when $[0, 2^{28} - 1]$ does not divide evenly into intervals of size n . The auxiliary `__fp_randint_e:wwwNnn` discards the pseudo-random integer if it lies in an incomplete interval, and repeats.

`__fp_randint_e:w` For wide ranges we use the same code except for the last eight digits which use `__fp_rand_myriads:n`. It is not safe to combine the first digits with the last eight as a single string of digits, as this may exceed 16 digits and be rounded. Instead, we first add the first few digits (times 10^8) to the lower bound. The result is compared to the upper bound and the process repeats if needed.

```

16553 \cs_new:Npn \__fp_randint_o:Nw ? #1 @
16554 {
16555   \if_case:w
16556     \__int_eval:w \__fp_array_count:n {#1} - \c_one \__int_eval_end:
16557     \exp_after:wN \__fp_randint_e:w \c_one_fp #1
16558   \or: \__fp_randint_e:w #1
16559   \else:

```

```

16560     \__msg_kernel_expandable_error:nnnnn
16561     { kernel } { fp-num-args } { randint() } { 1 } { 2 }
16562     \exp_after:wN \c_nan_fp \exp:w
16563     \fi:
16564     \exp_after:wN \exp_end:
16565 }
16566 \cs_new:Npn \__fp_randint_badarg:w \s__fp \__fp_chk:w #1#2#3;
16567 {
16568     \__fp_int:wTF \s__fp \__fp_chk:w #1#2#3;
16569     {
16570         \if_meaning:w 1 #1
16571         \if_int_compare:w
16572             \use_i_delimit_by_q_stop:nw #3 \q_stop > \c_sixteen
16573             \exp_after:wN \c_one
16574             \fi:
16575         \fi:
16576     }
16577     { \c_one }
16578 }
16579 \cs_new:Npn \__fp_randint_e:w #1; #2;
16580 {
16581     \if_case:w
16582         \__fp_randint_badarg:w #1;
16583         \__fp_randint_badarg:w #2;
16584         \fp_compare:nNnTF { #1; } > { #2; } { \c_one } { \c_zero }
16585         \exp_after:wN \exp_after:wN \exp_after:wN \__fp_randint_e:wnn
16586         \__fp_parse:n { #2; - #1; } { #1; } { #2; }
16587     \or:
16588         \__fp_invalid_operation_tl_o:ff
16589         { randint } { \__fp_array_to_clist:n { #1; #2; } }
16590         \exp:w
16591     \fi:
16592 }
16593 \cs_new:Npn \__fp_randint_e:wnn #1;
16594 {
16595     \exp_after:wN \__fp_randint_e:wwNnn
16596     \__int_value:w \__fp_rand_uniform: \exp_after:wN ;
16597     \exp:w \exp_end_continue_f:w
16598     \fp_compare:nNnTF { #1 ; } < \c__fp_rand_size_int
16599     { \fp_to_int:n { #1 ; + 1 } ; \__fp_randint_narrow_e:nnnn }
16600     { \fp_to_int:n { floor(#1 ; * 1e-8 + 1) } ; \__fp_randint_wide_e:nnnn }
16601 }
16602 \cs_new:Npn \__fp_randint_e:wwNnn #1 ; #2 ;
16603 {
16604     \exp_after:wN \__fp_randint_e:wwwNnn
16605     \__int_value:w \int_mod:nn {#1} {#2} ; #1 ; #2 ;
16606 }
16607 \cs_new:Npn \__fp_randint_e:wwwNnn #1 ; #2 ; #3 ; #4
16608 {
16609     \int_compare:nNnTF { #2 - #1 + #3 } > \c__fp_rand_size_int
16610     {
16611         \exp_after:wN \__fp_randint_e:wwNnn
16612         \__int_value:w \__fp_rand_uniform: ; #3 ; #4
16613     }

```

```

16614         { #4 {#1} {#3} }
16615     }
16616 \cs_new:Npn \__fp_randint_narrow_e:nnnn #1#2#3#4
16617 { \__fp_parse_o:n { #3 + #1 } \exp:w }
16618 \cs_new:Npn \__fp_randint_wide_e:nnnn #1#2#3#4
16619 {
16620     \exp_after:wN \exp_after:wN
16621     \exp_after:wN \__fp_randint_wide_e:wnnn
16622     \__fp_parse:n { #3 + #1e8 + \__fp_rand_myriads:n { xx } }
16623     {#2} {#3} {#4}
16624 }
16625 \cs_new:Npn \__fp_randint_wide_e:wnnn #1 ; #2#3#4
16626 {
16627     \fp_compare:nNnTF { #1 ; } > {#4}
16628     {
16629         \exp_after:wN \__fp_randint_e:wwNnn
16630         \__int_value:w \__fp_rand_uniform: ; #2 ;
16631         \__fp_randint_wide_e:nnnn {#3} {#4}
16632     }
16633     { \__fp_exp_after_o:w #1 ; \exp:w }
16634 }

```

(End definition for `__fp_randint_o:Nw` and others.)

End the initial conditional that ensures these commands are only defined in pdfTeX and LuaTeX.

```

16635     }
16636 </initex | package>

```

33 l3fp-assign implementation

```

16637 (*initex | package)
16638 <@@=fp>

```

33.1 Assigning values

\fp_new:N Floating point variables are initialized to be +0.

```

16639 \cs_new_protected:Npn \fp_new:N #1
16640 { \cs_new_eq:NN #1 \c_zero_fp }
16641 \cs_generate_variant:Nn \fp_new:N {c}

```

(End definition for `\fp_new:N`. This function is documented on page 182.)

\fp_set:Nn Simply use `__fp_parse:n` within various f-expanding assignments.

```

16642 \cs_new_protected:Npn \fp_set:Nn #1#2
16643 { \tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_gset:Nn 16644 \cs_new_protected:Npn \fp_gset:Nn #1#2
\fp_gset:cn 16645 { \tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_const:Nn 16646 \cs_new_protected:Npn \fp_const:Nn #1#2
\fp_const:cn 16647 { \tl_const:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
16648 \cs_generate_variant:Nn \fp_set:Nn {c}
16649 \cs_generate_variant:Nn \fp_gset:Nn {c}
16650 \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 182.)

`\fp_set_eq:NN` Copying a floating point is the same as copying the underlying token list.
`\fp_set_eq:cN` 16651 `\cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN`
`\fp_set_eq:Nc` 16652 `\cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN`
`\fp_set_eq:cc` 16653 `\cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }`
`\fp_gset_eq:NN` 16654 `\cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }`
`\fp_gset_eq:cN` (End definition for `\fp_set_eq:NN` and `\fp_gset_eq:NN`. These functions are documented on page 183.)
`\fp_gset_eq:Nc`
`\fp_gset_eq:cc`
`\fp_zero:N` Setting a floating point to zero: copy `\c_zero_fp`.
`\fp_zero:c` 16655 `\cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }`
`\fp_gzero:N` 16656 `\cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }`
`\fp_gzero:c` 16657 `\cs_generate_variant:Nn \fp_zero:N { c }`
16658 `\cs_generate_variant:Nn \fp_gzero:N { c }`
(End definition for `\fp_zero:N` and `\fp_gzero:N`. These functions are documented on page 182.)

`\fp_zero_new:N` Set the floating point to zero, or define it if needed.
`\fp_zero_new:c` 16659 `\cs_new_protected:Npn \fp_zero_new:N #1`
`\fp_gzero_new:N` 16660 `{ \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }`
`\fp_gzero_new:c` 16661 `\cs_new_protected:Npn \fp_gzero_new:N #1`
16662 `{ \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }`
16663 `\cs_generate_variant:Nn \fp_zero_new:N { c }`
16664 `\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 182.)

33.2 Updating values

These match the equivalent functions in `l3int` and `l3skip`.

`\fp_add:Nn` For the sake of error recovery we should not simply set `#1` to `#1 ± (#2)`: for instance, if `#2`
`\fp_add:cN` is 0)+2, the parsing error would be raised at the last closing parenthesis rather than at
`\fp_gadd:Nn` the closing parenthesis in the user argument. Thus we evaluate `#2` instead of just putting
`\fp_gadd:cN` parentheses. As an optimization we use `__fp_parse:n` rather than `\fp_eval:n`, which
`\fp_sub:Nn` would convert the result away from the internal representation and back.
`\fp_sub:cN` 16665 `\cs_new_protected:Npn \fp_add:Nn { __fp_add:NNNn \fp_set:Nn + }`
`\fp_gsub:Nn` 16666 `\cs_new_protected:Npn \fp_gadd:Nn { __fp_add:NNNn \fp_gset:Nn + }`
`\fp_gsub:cN` 16667 `\cs_new_protected:Npn \fp_sub:Nn { __fp_add:NNNn \fp_set:Nn - }`
`__fp_add:NNNn` 16668 `\cs_new_protected:Npn \fp_gsub:Nn { __fp_add:NNNn \fp_gset:Nn - }`
16669 `\cs_new_protected:Npn __fp_add:NNNn #1#2#3#4`
16670 `{ #1 #3 { #3 #2 __fp_parse:n {#4} } }`
16671 `\cs_generate_variant:Nn \fp_add:Nn { c }`
16672 `\cs_generate_variant:Nn \fp_gadd:Nn { c }`
16673 `\cs_generate_variant:Nn \fp_sub:Nn { c }`
16674 `\cs_generate_variant:Nn \fp_gsub:Nn { c }`
(End definition for `\fp_add:Nn` and others. These functions are documented on page 183.)

33.3 Showing values

`\fp_show:N` This shows the result of computing its argument. The input of `__msg_show_variable:NNNnn` must start with `>~` (or be empty).

```
\fp_show:c
\fp_show:n
16675 \cs_new_protected:Npn \fp_show:N #1
16676 {
16677   \__msg_show_variable:NNNnn #1 \fp_if_exist:NTF ? { }
16678   { > ~ \token_to_str:N #1 = \fp_to_tl:N #1 }
16679 }
16680 \cs_new_protected:Npn \fp_show:n
16681 { \__msg_show_wrap:Nn \fp_to_tl:n }
16682 \cs_generate_variant:Nn \fp_show:N { c }
```

(End definition for `\fp_show:N` and `\fp_show:n`. These functions are documented on page 189.)

33.4 Some useful constants and scratch variables

`\c_one_fp` Some constants.

```
\c_e_fp
16683 \fp_const:Nn \c_e_fp { 2.718 2818 2845 9045 }
16684 \fp_const:Nn \c_one_fp { 1 }
```

(End definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 187.)

`\c_pi_fp` We simply round π to the closest multiple of 10^{-15} .

```
\c_one_degree_fp
16685 \fp_const:Nn \c_pi_fp { 3.141 5926 5358 9793 }
16686 \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 188.)

`\l_tmpa_fp` Scratch variables are simply initialized there.

```
\l_tmpb_fp
\g_tmpa_fp
\g_tmpb_fp
16687 \fp_new:N \l_tmpa_fp
16688 \fp_new:N \l_tmpb_fp
16689 \fp_new:N \g_tmpa_fp
16690 \fp_new:N \g_tmpb_fp
```

(End definition for `\l_tmpa_fp` and others. These variables are documented on page 188.)

```
16691 </initex | package>
```

34 l3sort implementation

```
16692 < *initex | package>
```

```
16693 < @@=sort>
```

34.1 Variables

`\l__sort_length_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 `__sort_compute_range:.` That bound is such that the merge sort will only use `\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.

```
16694 \int_new:N \l__sort_length_int
16695 \int_new:N \l__sort_min_int
```

```

16696 \int_new:N \l__sort_top_int
16697 \int_new:N \l__sort_max_int
16698 \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.

```

16699 \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.

```

16700 \int_new:N \l__sort_begin_int
16701 \int_new:N \l__sort_end_int

```

(End definition for `\l__sort_begin_int` and `\l__sort_end_int`.)

`\l__sort_A_int` `\l__sort_B_int` `\l__sort_C_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.

```

16702 \int_new:N \l__sort_A_int
16703 \int_new:N \l__sort_B_int
16704 \int_new:N \l__sort_C_int

```

(End definition for `\l__sort_A_int`, `\l__sort_B_int`, and `\l__sort_C_int`.)

34.2 Finding available `\toks` registers

`__sort_shrink_range:` `__sort_shrink_range_loop:` 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`.

```

16705 \cs_new_protected:Npn \__sort_shrink_range:
16706 {
16707   \int_set:Nn \l__sort_A_int
16708     { \l__sort_true_max_int - \l__sort_min_int + \c_one }
16709   \int_set:Nn \l__sort_block_int { \c_max_register_int / \c_two }
16710   \__sort_shrink_range_loop:
16711   \int_set:Nn \l__sort_max_int
16712     {
16713     \int_compare:nNnTF
16714       { \l__sort_block_int * \c_three / \c_two } > \l__sort_A_int
16715       {
16716         \l__sort_min_int

```

```

16717         + ( \l__sort_A_int - \c_one ) / \c_two
16718         + \l__sort_block_int / \c_four
16719         - \c_one
16720     }
16721     { \l__sort_true_max_int - \l__sort_block_int / \c_two }
16722 }
16723 }
16724 \cs_new_protected:Npn \__sort_shrink_range_loop:
16725 {
16726     \if_int_compare:w \l__sort_A_int < \l__sort_block_int
16727         \tex_divide:D \l__sort_block_int \c_two
16728         \exp_after:wN \__sort_shrink_range_loop:
16729     \fi:
16730 }

```

(End definition for `__sort_shrink_range:` and `__sort_shrink_range_loop:`.)

`__sort_compute_range:` First find out what `\toks` have not yet been assigned. There are many cases. In L^AT_EX 2_ε 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_EX, or when the package `etex` is loaded in L^AT_EX 2_ε, redefine `__sort_compute_range:` to use the range `\count265` to `\count275 - 1`. The `elocalloc` 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 `elocalloc` could be loaded.

In ConT_EXt 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:`. The L^AT_EX3 format mode is easiest: no `\toks` are ever allocated so available `\toks` range from 0 to `\c_max_register_int` and we precompute the result of `__sort_shrink_range:`.

```

16731 (*package)
16732 \cs_new_protected:Npn \__sort_compute_range:
16733 {
16734     \int_set:Nn \l__sort_min_int { \tex_count:D 15 + \c_one }
16735     \int_set:Nn \l__sort_true_max_int { \c_max_register_int + \c_one }
16736     \__sort_shrink_range:
16737     \if_meaning:w \loctoks \tex_undefined:D \else:
16738         \if_meaning:w \loctoks \scan_stop: \else:
16739             \__sort_redefine_compute_range:
16740             \__sort_compute_range:
16741         \fi:
16742     \fi:
16743 }
16744 \cs_new_protected:Npn \__sort_redefine_compute_range:
16745 {
16746     \cs_if_exist:cTF { ver@elocalloc.sty }
16747     {
16748         \cs_gset_protected:Npn \__sort_compute_range:
16749         {
16750             \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
16751             \int_set_eq:NN \l__sort_true_max_int \e@alloc@top
16752             \__sort_shrink_range:
16753         }

```

```

16754     }
16755     {
16756         \cs_gset_protected:Npn \__sort_compute_range:
16757         {
16758             \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
16759             \int_set:Nn \l__sort_true_max_int { \tex_count:D 275 }
16760             \__sort_shrink_range:
16761         }
16762     }
16763 }
16764 \cs_if_exist:NT \loctoks { \__sort_redefine_compute_range: }
16765 \tl_map_inline:nn { \lastallocatedtoks \c_syst_last_allocated_toks }
16766 {
16767     \cs_if_exist:NT #1
16768     {
16769         \cs_gset_protected:Npn \__sort_compute_range:
16770         {
16771             \int_set:Nn \l__sort_min_int { #1 + \c_one }
16772             \int_set:Nn \l__sort_true_max_int { \c_max_register_int + \c_one }
16773             \__sort_shrink_range:
16774         }
16775     }
16776 }
16777 </package>
16778 <*initex>
16779 \int_const:Nn \c__sort_max_length_int
16780 { ( \c_max_register_int + 1 ) * 3 / 4 }
16781 \cs_new_protected:Npn \__sort_compute_range:
16782 {
16783     \int_set_eq:NN \l__sort_min_int \c_zero
16784     \int_set:Nn \l__sort_true_max_int { \c_max_register_int + \c_one }
16785     \int_set_eq:NN \l__sort_max_int \c__sort_max_length_int
16786 }
16787 </initex>

```

(End definition for __sort_compute_range:, __sort_redefine_compute_range:, and \c__sort_max_length_int.)

34.3 Protected user commands

__sort_main:NNNnNn 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, all further items are entirely ignored after raising an error. Secondly, sort the array of \toks registers, using the user-defined sorting function, #6. Finally, unpack the \toks registers (now sorted) into a variable of the right type, by x-expanding the code in #4, specific to each type of list.

```

16788 \cs_new_protected:Npn \__sort_main:NNNnNn #1#2#3#4#5#6
16789 {
16790     \group_begin:
16791     <package> \__sort_disable_toksdef:
16792     \__sort_compute_range:
16793     \int_set_eq:NN \l__sort_top_int \l__sort_min_int
16794     #2 #5
16795     {

```



```

16796 \if_int_compare:w \l__sort_top_int = \l__sort_max_int
16797 \__sort_too_long_error:NNw #3 #5
16798 \fi:
16799 \tex_toks:D \l__sort_top_int {##1}
16800 \tex_advance:D \l__sort_top_int \c_one
16801 }
16802 \int_set:Nn \l__sort_length_int
16803 { \l__sort_top_int - \l__sort_min_int }
16804 \cs_set:Npn \__sort_compare:nn ##1 ##2 { #6 }
16805 \int_set_eq:NN \l__sort_block_int \c_one
16806 \__sort_level:
16807 \use:x
16808 {
16809 \group_end:
16810 #1 \exp_not:N #5 {#4}
16811 }
16812 }

```

`\seq_sort:Nn` The first argument to `__sort_main:NNNnNn` is the final assignment function used, either `\tl_set:Nn` or `\tl_gset:Nn` to control local versus global results. The second argument is what mapping function is used when storing items to `\toks` registers, and the third breaks away from the loop. The fourth is used to build back the correct kind of list from the contents of the `\toks` registers, including the leading `\s__seq`. Fifth and sixth arguments are the variable to sort, and the sorting method as inline code.

(End definition for `\seq_sort:Nn` and `\seq_gsort:Nn`. These functions are documented on page 116.)

```

\tl_gsort:cn      16827 \cs_new_protected:Npn \tl_sort:Nn
                   16828 {
                   16829     \__sort_main:NNNnNn \tl_set:Nn
                   16830     \tl_map_inline:Nn \tl_map_break:n
                   16831     { \__sort_toks:NN \prg_do_nothing: \prg_do_nothing: }
                   16832 }
                   16833 \cs_generate_variant:Nn \tl_sort:Nn { c }
                   16834 \cs_new_protected:Npn \tl_gsort:Nn

```

```

16835 {
16836   \__sort_main:NNNnNn \tl_gset:Nn
16837   \tl_map_inline:Nn \tl_map_break:n
16838   { \__sort_toks:NN \prg_do_nothing: \prg_do_nothing: }
16839 }
16840 \cs_generate_variant:Nn \tl_gsort:Nn { c }

```

(End definition for `\tl_sort:Nn` and `\tl_gsort:Nn`. These functions are documented on page 98.)

`\clist_sort:Nn` The case of empty comma-lists is a little bit special as usual, and filtered out: there is nothing to sort in that case. Otherwise, the input is done with `\clist_map_inline:Nn`, and the output requires some more elaborate processing than for sequences and token lists. The first comma must be removed. An item must be wrapped in an extra set of braces if it contains either the space or the comma characters. This is taken care of by `\clist_wrap_item:n`, but `__sort_toks:NN` would simply feed `\tex_the:D \tex_toks:D <number>` as an argument to that function; hence we need to expand this argument once to unpack the register.

```

16841 \cs_new_protected:Npn \clist_sort:Nn
16842 { \__sort_clist:NNn \tl_set:Nn }
16843 \cs_new_protected:Npn \clist_gsort:Nn
16844 { \__sort_clist:NNn \tl_gset:Nn }
16845 \cs_generate_variant:Nn \clist_sort:Nn { c }
16846 \cs_generate_variant:Nn \clist_gsort:Nn { c }
16847 \cs_new_protected:Npn \__sort_clist:NNn #1#2#3
16848 {
16849   \clist_if_empty:NF #2
16850   {
16851     \__sort_main:NNNnNn #1
16852     \clist_map_inline:Nn \clist_map_break:n
16853     {
16854       \exp_last_unbraced:Nf \use_none:n
16855       { \__sort_toks:NN \exp_args:No \__clist_wrap_item:n }
16856     }
16857     #2 {#3}
16858   }
16859 }

```

(End definition for `\clist_sort:Nn`, `\clist_gsort:Nn`, and `__sort_clist:NNn`. These functions are documented on page 126.)

`__sort_toks:NN` Unpack the various `\toks` registers, from `\l__sort_min_int` to `\l__sort_top_int - 1`.
`__sort_toks:NNw` The functions #1 and #2 allow us to treat the three data structures in a unified way:

- for sequences, they are `\exp_not:N __seq_item:n`, expanding to the `__seq_item:n` separator, as expected;
- for token lists, they expand to nothing;
- for comma lists, they expand to `\exp_args:No \clist_wrap_item:n`, taking care of unpacking the register before letting the undocumented internal `clist` function `\clist_wrap_item:n` do the work of putting a comma and possibly braces.

```

16860 \cs_new:Npn \__sort_toks:NN #1#2
16861 { \__sort_toks:NNw #1 #2 \l__sort_min_int ; }
16862 \cs_new:Npn \__sort_toks:NNw #1#2#3 ;

```

```

16863 {
16864   \if_int_compare:w #3 < \l__sort_top_int
16865     #1 #2 { \tex_the:D \tex_toks:D #3 }
16866     \exp_after:wN \__sort_toks:NNw \exp_after:wN #1 \exp_after:wN #2
16867     \__int_value:w \__int_eval:w #3 + \c_one \exp_after:wN ;
16868   \fi:
16869 }

```

(End definition for `__sort_toks:NN` and `__sort_toks:NNw`.)

34.4 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.

```

16870 \cs_new_protected:Npn \__sort_level:
16871 {
16872   \if_int_compare:w \l__sort_block_int < \l__sort_length_int
16873     \l__sort_end_int \l__sort_min_int
16874     \__sort_merge_blocks:
16875     \tex_advance:D \l__sort_block_int \l__sort_block_int
16876     \exp_after:wN \__sort_level:
16877   \fi:
16878 }

```

(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 will index 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 `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.

```

16879 \cs_new_protected:Npn \__sort_merge_blocks:
16880 {
16881   \l__sort_begin_int \l__sort_end_int
16882   \tex_advance:D \l__sort_end_int \l__sort_block_int
16883   \if_int_compare:w \l__sort_end_int < \l__sort_top_int
16884     \l__sort_A_int \l__sort_end_int
16885     \tex_advance:D \l__sort_end_int \l__sort_block_int
16886     \if_int_compare:w \l__sort_end_int > \l__sort_top_int
16887       \l__sort_end_int \l__sort_top_int
16888     \fi:
16889     \l__sort_B_int \l__sort_A_int
16890     \l__sort_C_int \l__sort_top_int
16891     \__sort_copy_block:
16892     \tex_advance:D \l__sort_A_int - \c_one

```

```

16893     \tex_advance:D \l__sort_B_int - \c_one
16894     \tex_advance:D \l__sort_C_int - \c_one
16895     \exp_after:wN \__sort_merge_blocks_aux:
16896     \exp_after:wN \__sort_merge_blocks:
16897   \fi:
16898 }

```

(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.

```

16899 \cs_new_protected:Npn \__sort_copy_block:
16900 {
16901   \tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
16902   \tex_advance:D \l__sort_C_int \c_one
16903   \tex_advance:D \l__sort_B_int \c_one
16904   \if_int_compare:w \l__sort_B_int = \l__sort_end_int
16905     \use_i:nn
16906   \fi:
16907   \__sort_copy_block:
16908 }

```

(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.

```

16909 \cs_new_protected:Npn \__sort_merge_blocks_aux:
16910 {
16911   \exp_after:wN \__sort_compare:nn \exp_after:wN
16912   { \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN }
16913   \exp_after:wN { \tex_the:D \tex_toks:D \l__sort_C_int }
16914   \prg_do_nothing:
16915   \__sort_return_mark:N
16916   \__sort_return_mark:N
16917   \__sort_return_none_error:
16918 }

```

(End definition for __sort_merge_blocks_aux:.)

\sort_return_same: The marker removes one token. Each comparison should call \sort_return_same: or **\sort_return_swapped:** exactly once. If neither is called, **__sort_return_none_error:** is called.

```

\__sort_return_mark:N
\__sort_return_none_error:
\__sort_return_two_error:w
16919 \cs_new_protected:Npn \sort_return_same: #1 \__sort_return_mark:N
16920 { #1 \__sort_return_mark:N \__sort_return_two_error:w \__sort_return_same: }
16921 \cs_new_protected:Npn \sort_return_swapped: #1 \__sort_return_mark:N
16922 { #1 \__sort_return_mark:N \__sort_return_two_error:w \__sort_return_swapped: }
16923 \cs_new_protected:Npn \__sort_return_mark:N #1 { }

```

```

16924 \cs_new_protected:Npn \__sort_return_none_error:
16925 {
16926   \__msg_kernel_error:nnxx { sort } { return-none }
16927   { \tex_the:D \tex_toks:D \l__sort_A_int }
16928   { \tex_the:D \tex_toks:D \l__sort_C_int }
16929   \__sort_return_same:
16930 }
16931 \cs_new_protected:Npn \__sort_return_two_error:w
16932 #1 \__sort_return_none_error:
16933 { \__msg_kernel_error:nn { sort } { return-two } }

```

(End definition for \sort_return_same: and others. These functions are documented on page ??.)

__sort_return_same: 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.

```

16934 \cs_new_protected:Npn \__sort_return_same:
16935 {
16936   \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
16937   \tex_advance:D \l__sort_B_int - \c_one
16938   \tex_advance:D \l__sort_C_int - \c_one
16939   \if_int_compare:w \l__sort_C_int < \l__sort_top_int
16940     \use_i:nn
16941   \fi:
16942   \__sort_merge_blocks_aux:
16943 }

```

(End definition for __sort_return_same:.)

__sort_return_swapped: 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:**.

```

16944 \cs_new_protected:Npn \__sort_return_swapped:
16945 {
16946   \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_A_int
16947   \tex_advance:D \l__sort_B_int - \c_one
16948   \tex_advance:D \l__sort_A_int - \c_one
16949   \if_int_compare:w \l__sort_A_int < \l__sort_begin_int
16950     \__sort_merge_blocks_end: \use_i:nn
16951   \fi:
16952   \__sort_merge_blocks_aux:
16953 }

```

(End definition for __sort_return_swapped:.)

__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**.

```

16954 \cs_new_protected:Npn \__sort_merge_blocks_end:

```

```

16955 {
16956   \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
16957   \tex_advance:D \l__sort_B_int - \c_one
16958   \tex_advance:D \l__sort_C_int - \c_one
16959   \if_int_compare:w \l__sort_B_int < \l__sort_begin_int
16960     \use_i:nn
16961   \fi:
16962   \__sort_merge_blocks_end:
16963 }

```

(End definition for `__sort_merge_blocks_end:`.)

34.5 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_recursion_tail \q_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 will be 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 *<item>* 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 \} \q_stop
```

In this example, which matches the structure of $\backslash_sort_quick_split_i:NnnnnNn$ and a few other functions below, the $\backslash_sort_loop:wNn$ 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 $\backslash_sort_loop:wNn$, 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 $\backslash_sort:nnnn$, 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 $\backslash_sort_loop:wNn$ 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 $\langle conditional \rangle \{ \langle item \rangle \}$, 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 $\backslash use:ff$ using a continuation-passing style: $\backslash_sort_quick_split:NnNn$ expects a list followed by $\backslash q_mark \{ \langle code \rangle \}$, 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 ... \q_mark
{
  \__sort_quick_split:NnNn #1 ... \q_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 *conditional* $\{\langle item \rangle\}$ 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 *end-loop* 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 *end-loop* 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 `\q_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 will sort the prepared list and perform the post action placed after `\q_mark`, namely removing the trailing `\s__stop` and `\q_stop` and leaving `\exp_stop_f:` to stop `f`-expansion.

```
16964 \cs_new:Npn \tl_sort:nN #1#2
16965 {
16966   \exp_not:f
16967   {
16968     \tl_if_blank:nF {#1}
16969     {
16970       \__sort_quick_prepare:Nnnn #2 { } { }
16971       #1
16972       { \__prg_break_point: \__sort_quick_prepare_end:NNNnw }
16973       \q_stop
16974     }
16975   }
16976 }
16977 \cs_new:Npn \__sort_quick_prepare:Nnnn #1#2#3#4
16978 {
16979   \__prg_break: #4 \__prg_break_point:
16980   \__sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
16981 }
```



```

16982 \cs_new:Npn \__sort_quick_prepare_end:NNNnw #1#2#3#4#5 \q_stop
16983 {
16984   \__sort_quick_split:NnNn #4 \__sort_quick_end:nnTFNn { }
16985   \q_mark { \__sort_quick_cleanup:w \exp_stop_f: }
16986   \s__stop \q_stop
16987 }
16988 \cs_new:Npn \__sort_quick_cleanup:w #1 \s__stop \q_stop {#1}

```

(End definition for `\tl_sort:nN` and others. These functions are documented on page 98.)

```

\__sort_quick_split:NnNn
\__sort_quick_only_i:NnnnnNn
\__sort_quick_only_ii:NnnnnNn
\__sort_quick_split_i:NnnnnNn
\__sort_quick_split_ii:NnnnnNn

```

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.

```

16989 \cs_new:Npn \__sort_quick_split:NnNn #1#2#3#4
16990 {
16991   #3 {#2} {#4} \__sort_quick_only_ii:NnnnnNn \__sort_quick_only_i:NnnnnNn
16992   \__sort_quick_single_end:nnnwnw
16993   { #3 {#4} } { } { } {#2}
16994 }
16995 \cs_new:Npn \__sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
16996 {
16997   #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn \__sort_quick_only_i:NnnnnNn
16998   \__sort_quick_only_i_end:nnnwnw
16999   { #6 {#7} } { #3 #2 } { } {#5}
17000 }
17001 \cs_new:Npn \__sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
17002 {
17003   #6 {#5} {#7} \__sort_quick_only_ii:NnnnnNn \__sort_quick_split_i:NnnnnNn
17004   \__sort_quick_only_ii_end:nnnwnw
17005   { #6 {#7} } { } { #4 #2 } {#5}
17006 }
17007 \cs_new:Npn \__sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
17008 {
17009   #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn \__sort_quick_split_i:NnnnnNn
17010   \__sort_quick_split_end:nnnwnw
17011   { #6 {#7} } { #3 #2 } {#4} {#5}
17012 }
17013 \cs_new:Npn \__sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
17014 {
17015   #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn \__sort_quick_split_i:NnnnnNn
17016   \__sort_quick_split_end:nnnwnw
17017   { #6 {#7} } {#3} { #4 #2 } {#5}
17018 }

```

(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 `\q_mark`. To avoid a memory problem described earlier, all of the ending functions read #6 until `\q_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.

```

17019 \cs_new:Npn \__sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}
17020 \cs_new:Npn \__sort_quick_single_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
17021 { #5 {#3} #6 \q_stop }
17022 \cs_new:Npn \__sort_quick_only_i_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
17023 {
17024   \__sort_quick_split:NnNn #1
17025   \__sort_quick_end:nnTFNn { } \q_mark {#5}
17026   {#3}
17027   #6 \q_stop
17028 }
17029 \cs_new:Npn \__sort_quick_only_ii_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
17030 {
17031   \__sort_quick_split:NnNn #2
17032   \__sort_quick_end:nnTFNn { } \q_mark { #5 {#3} }
17033   #6 \q_stop
17034 }
17035 \cs_new:Npn \__sort_quick_split_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
17036 {
17037   \__sort_quick_split:NnNn #2 \__sort_quick_end:nnTFNn { } \q_mark
17038   {
17039     \__sort_quick_split:NnNn #1
17040     \__sort_quick_end:nnTFNn { } \q_mark {#5}
17041     {#3}
17042   }
17043   #6 \q_stop
17044 }

```

(End definition for `__sort_quick_end:nnTFNn` and others.)

34.6 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 `l3sort` commands to be trivial, with `__sort_level:` getting rid of the final assignment. This error recovery won't work in a group.

```

17045 \cs_new_protected:Npn \__sort_error:
17046 {
17047   \cs_set_eq:NN \__sort_merge_blocks_aux: \prg_do_nothing:
17048   \cs_set_eq:NN \__sort_merge_blocks: \prg_do_nothing:

```

```

17049 \cs_set_protected:Npn \__sort_level: \use:x ##1 { \group_end: }
17050 }

```

(End definition for __sort_error:.)

__sort_disable_toksdef: While sorting, \toksdef is locally disabled to prevent users from using \newtoks or
__sort_disabled_toksdef:n similar commands in their comparison code: the \toks registers that would be assigned
are in use by l3sort. In format mode, none of this is needed since there is no \toks
allocator.

```

17051 \*package>
17052 \cs_new_protected:Npn \__sort_disable_toksdef:
17053 { \cs_set_eq:NN \toksdef \__sort_disabled_toksdef:n }
17054 \cs_new_protected:Npn \__sort_disabled_toksdef:n #1
17055 {
17056   \__msg_kernel_error:nnx { sort } { toksdef }
17057   { \token_to_str:N #1 }
17058   \__sort_error:
17059   \tex_toksdef:D #1
17060 }
17061 \__msg_kernel_new:nnnn { sort } { toksdef }
17062 { Allocation~of~\iow_char:N\toks~registers~impossible~while~sorting. }
17063 {
17064   The~comparison~code~used~for~sorting~a~list~has~attempted~to~
17065   define~#1~as~a~new~\iow_char:N\toks~register~using~\iow_char:N\newtoks~
17066   or~a~similar~command.~The~list~will~not~be~sorted.
17067 }
17068 \*package>

```

(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.

```

17069 \cs_new_protected:Npn \__sort_too_long_error:NNw #1#2 \fi:
17070 {
17071   \fi:
17072   \__msg_kernel_error:nnxxx { sort } { too-large }
17073   { \token_to_str:N #2 }
17074   { \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
17075   { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
17076   #1 \__sort_error:
17077 }
17078 \__msg_kernel_new:nnnn { sort } { too-large }
17079 { The~list~#1~is~too~long~to~be~sorted~by~TeX. }
17080 {
17081   TeX~has~#2~toks~registers~still~available:~
17082   this~only~allows~to~sort~with~up~to~#3~
17083   items.~All~extra~items~will~be~deleted.
17084 }

```

(End definition for __sort_too_long_error:NNw.)

```

17085 \__msg_kernel_new:nnnn { sort } { return-none }
17086 { The~comparison~code~did~not~return. }
17087 {
17088   When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~

```

```

17089      did-not-call~
17090      \iow_char:N\sort_return_same: ~nor~
17091      \iow_char:N\sort_return_swapped: .~
17092      Exactly~one~of~these~should-be~called.
17093    }
17094    \__msg_kernel_new:nnnn { sort } { return-two }
17095    { The~comparison~code~returned~multiple~times. }
17096    {
17097      When~sorting~a~list,~the~code~to~compare~items~called~
17098      \iow_char:N\sort_return_same: ~or~
17099      \iow_char:N\sort_return_swapped: ~multiple~times.~
17100      Exactly~one~of~these~should-be~called.
17101    }

```

34.7 Deprecated functions

`\sort_ordered:` These functions were renamed for consistency.
`\sort_reversed:`

```

17102 \cs_new_eq:NN \sort_ordered: \sort_return_same:
17103 \cs_new_eq:NN \sort_reversed: \sort_return_swapped:

```

(End definition for `\sort_ordered:` and `\sort_reversed:.`)

```

17104 </initex | package>

```

35 l3candidates Implementation

```

17105 (*initex | package)

```

35.1 Additions to l3basics

```

17106 <@@=cs>

```

`\cs_log:N` Use `\cs_show:N` or `\cs_show:c` after calling `__msg_log_next:` to redirect their output to the log file only. Note that `\cs_log:c` is not just a variant of `\cs_log:N` as the csname should be turned to a control sequence within a group (see `\cs_show:c`).

```

17107 \cs_new_protected:Npn \cs_log:N
17108   { \__msg_log_next: \cs_show:N }
17109 \cs_new_protected:Npn \cs_log:c
17110   { \__msg_log_next: \cs_show:c }

```

(End definition for `\cs_log:N`. This function is documented on page [201](#).)

`__kernel_register_log:N` Redirect the output of `__kernel_register_show:N` to the log.

```

\__kernel_register_log:c
17111 \cs_new_protected:Npn \__kernel_register_log:N
17112   { \__msg_log_next: \__kernel_register_show:N }
17113 \cs_generate_variant:Nn \__kernel_register_log:N { c }

```

(End definition for `__kernel_register_log:N`.)

35.2 Additions to l3box

17114 <@@=box>

35.3 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.

17115 `\fp_new:N \l__box_angle_fp`

(End definition for `\l__box_angle_fp`.)

`\l__box_cos_fp` These are used to hold the calculated sine and cosine values while carrying out a rotation.

`\l__box_sin_fp` 17116 `\fp_new:N \l__box_cos_fp`

17117 `\fp_new:N \l__box_sin_fp`

(End definition for `\l__box_cos_fp` and `\l__box_sin_fp`.)

`\l__box_top_dim` These are the positions of the four edges of a box before manipulation.

`\l__box_bottom_dim` 17118 `\dim_new:N \l__box_top_dim`

`\l__box_left_dim` 17119 `\dim_new:N \l__box_bottom_dim`

`\l__box_right_dim` 17120 `\dim_new:N \l__box_left_dim`

17121 `\dim_new:N \l__box_right_dim`

(End definition for `\l__box_top_dim` and others.)

`\l__box_top_new_dim` These are the positions of the four edges of a box after manipulation.

`\l__box_bottom_new_dim` 17122 `\dim_new:N \l__box_top_new_dim`

`\l__box_left_new_dim` 17123 `\dim_new:N \l__box_bottom_new_dim`

`\l__box_right_new_dim` 17124 `\dim_new:N \l__box_left_new_dim`

17125 `\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.

17126 `\box_new:N \l__box_internal_box`

(End definition for `\l__box_internal_box`.)

`\box_rotate:Nn` 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

`__box_rotate:N`

`__box_rotate_x:nnN` 17127 `\cs_new_protected:Npn \box_rotate:Nn #1#2`

`__box_rotate_y:nnN` 17128 {

`__box_rotate_quadrant_one:` 17129 `\hbox_set:Nn #1`

`__box_rotate_quadrant_two:` 17130 {

`__box_rotate_quadrant_three:` 17131 `\group_begin:`

`__box_rotate_quadrant_four:` 17132 `\fp_set:Nn \l__box_angle_fp {#2}`

17133 `\fp_set:Nn \l__box_sin_fp { sind (\l__box_angle_fp) }`

17134 `\fp_set:Nn \l__box_cos_fp { cosd (\l__box_angle_fp) }`

17135 `__box_rotate:N #1`

17136 `\group_end:`

17137 }

17138 }

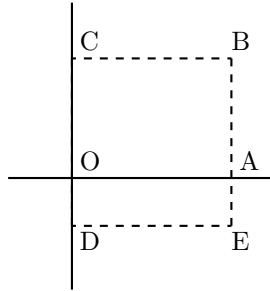


Figure 1: Co-ordinates of a box prior to rotation.

The edges of the box are then recorded: the left edge will always be at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.

```

17139 \cs_new_protected:Npn \_box_rotate:N #1
17140 {
17141   \dim_set:Nn \l__box_top_dim    { \box_ht:N #1 }
17142   \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
17143   \dim_set:Nn \l__box_right_dim   { \box_wd:N #1 }
17144   \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 α :

$$\begin{aligned}
 P'_x &= P_x - O_x \\
 P'_y &= P_y - O_y \\
 P''_x &= (P'_x \cos(\alpha)) - (P'_y \sin(\alpha)) \\
 P''_y &= (P'_x \sin(\alpha)) + (P'_y \cos(\alpha)) \\
 P'''_x &= P''_x + O_x + L_x \\
 P'''_y &= P''_y + O_y
 \end{aligned}$$

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.)

```

17145   \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
17146   {
17147     \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
17148     { \_box_rotate_quadrant_one: }
17149     { \_box_rotate_quadrant_two: }
17150   }
17151   {
17152     \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
17153     { \_box_rotate_quadrant_three: }
17154     { \_box_rotate_quadrant_four: }
17155   }

```

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 will be in the same place as the original.

```

17156 \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
17157 \hbox_set:Nn \l__box_internal_box
17158 {
17159   \tex_kern:D -\l__box_left_new_dim
17160   \hbox:n
17161   {
17162     \__driver_box_use_rotate:Nn
17163     \l__box_internal_box
17164     \l__box_angle_fp
17165   }
17166 }

```

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.

```

17167 \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
17168 \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
17169 \box_set_wd:Nn \l__box_internal_box
17170 { \l__box_right_new_dim - \l__box_left_new_dim }
17171 \box_use:N \l__box_internal_box
17172 }

```

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.

```

17173 \cs_new_protected:Npn \__box_rotate_x:nnN #1#2#3
17174 {
17175   \dim_set:Nn #3
17176   {
17177     \fp_to_dim:n
17178     {
17179       \l__box_cos_fp * \dim_to_fp:n {#1}
17180       - \l__box_sin_fp * \dim_to_fp:n {#2}
17181     }
17182   }
17183 }
17184 \cs_new_protected:Npn \__box_rotate_y:nnN #1#2#3
17185 {
17186   \dim_set:Nn #3
17187   {
17188     \fp_to_dim:n
17189     {
17190       \l__box_sin_fp * \dim_to_fp:n {#1}
17191       + \l__box_cos_fp * \dim_to_fp:n {#2}
17192     }
17193   }
17194 }

```

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.

```

17195 \cs_new_protected:Npn \__box_rotate_quadrant_one:
17196 {
17197   \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
17198   \l__box_top_new_dim
17199   \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
17200   \l__box_bottom_new_dim
17201   \__box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
17202   \l__box_left_new_dim
17203   \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
17204   \l__box_right_new_dim
17205 }
17206 \cs_new_protected:Npn \__box_rotate_quadrant_two:
17207 {
17208   \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
17209   \l__box_top_new_dim
17210   \__box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
17211   \l__box_bottom_new_dim
17212   \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
17213   \l__box_left_new_dim
17214   \__box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
17215   \l__box_right_new_dim
17216 }
17217 \cs_new_protected:Npn \__box_rotate_quadrant_three:
17218 {
17219   \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
17220   \l__box_top_new_dim
17221   \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
17222   \l__box_bottom_new_dim
17223   \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
17224   \l__box_left_new_dim
17225   \__box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
17226   \l__box_right_new_dim
17227 }
17228 \cs_new_protected:Npn \__box_rotate_quadrant_four:
17229 {
17230   \__box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
17231   \l__box_top_new_dim
17232   \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
17233   \l__box_bottom_new_dim
17234   \__box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
17235   \l__box_left_new_dim
17236   \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
17237   \l__box_right_new_dim
17238 }

```

(End definition for `\box_rotate:Nn` and others. These functions are documented on page 203.)

`\l__box_scale_x_fp` Scaling is potentially-different in the two axes.
`\l__box_scale_y_fp`

```

17239 \fp_new:N \l__box_scale_x_fp
17240 \fp_new:N \l__box_scale_y_fp

```

(End definition for `\l__box_scale_x_fp` and `\l__box_scale_y_fp`.)

`\box_resize:Nnn` Resizing a box starts by working out the various dimensions of the existing box.

`\box_resize:cnn` 17241 `\cs_new_protected:Npn \box_resize:Nnn #1#2#3`

`__box_resize_set_corners:N` 17242 `{`

`__box_resize:N` 17243 `\hbox_set:Nn #1`

`__box_resize:NNN` 17244 `{`

`\group_begin:`

`__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.

17247 `\fp_set:Nn \l__box_scale_x_fp`

17248 `{ \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.

17249 `\fp_set:Nn \l__box_scale_y_fp`

17250 `{`

17251 `\dim_to_fp:n {#3}`

17252 `/ \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }`

17253 `}`

Hand off to the auxiliary which does the rest of the work.

17254 `__box_resize:N #1`

17255 `\group_end:`

17256 `}`

17257 `}`

17258 `\cs_generate_variant:Nn \box_resize:NNn { c }`

17259 `\cs_new_protected:Npn __box_resize_set_corners:N #1`

17260 `{`

17261 `\dim_set:Nn \l__box_top_dim { \box_ht:N #1 }`

17262 `\dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }`

17263 `\dim_set:Nn \l__box_right_dim { \box_wd:N #1 }`

17264 `\dim_zero:N \l__box_left_dim`

17265 `}`

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.

17266 `\cs_new_protected:Npn __box_resize:N #1`

17267 `{`

17268 `__box_resize:NNN \l__box_right_new_dim`

17269 `\l__box_scale_x_fp \l__box_right_dim`

17270 `__box_resize:NNN \l__box_bottom_new_dim`

17271 `\l__box_scale_y_fp \l__box_bottom_dim`

17272 `__box_resize:NNN \l__box_top_new_dim`

17273 `\l__box_scale_y_fp \l__box_top_dim`

17274 `__box_resize_common:N #1`

17275 `}`

17276 `\cs_new_protected:Npn __box_resize:NNN #1#2#3`

17277 `{`

17278 `\dim_set:Nn #1`

17279 `{ \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }`

17280 `}`

(End definition for `\box_resize:Nnn` and others. These functions are documented on page 202.)

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).

```

\box_resize_to_ht:Nn \box_resize_to_ht:cn
\box_resize_to_ht_plus_dp:Nn \box_resize_to_ht_plus_dp:cn
\box_resize_to_wd:Nn \box_resize_to_wd:cn
\box_resize_to_wd_and_ht:Nnn \box_resize_to_wd_and_ht:cnn
17281 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2
17282 {
17283   \hbox_set:Nn #1
17284   {
17285     \group_begin:
17286       \__box_resize_set_corners:N #1
17287       \fp_set:Nn \l__box_scale_y_fp
17288       {
17289         \dim_to_fp:n {#2}
17290         / \dim_to_fp:n { \l__box_top_dim }
17291       }
17292       \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
17293       \__box_resize:N #1
17294     \group_end:
17295   }
17296 }
17297 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c }
17298 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2
17299 {
17300   \hbox_set:Nn #1
17301   {
17302     \group_begin:
17303       \__box_resize_set_corners:N #1
17304       \fp_set:Nn \l__box_scale_y_fp
17305       {
17306         \dim_to_fp:n {#2}
17307         / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
17308       }
17309       \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
17310       \__box_resize:N #1
17311     \group_end:
17312   }
17313 }
17314 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
17315 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
17316 {
17317   \hbox_set:Nn #1
17318   {
17319     \group_begin:
17320       \__box_resize_set_corners:N #1
17321       \fp_set:Nn \l__box_scale_x_fp
17322       { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
17323       \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
17324       \__box_resize:N #1
17325     \group_end:
17326   }
17327 }
17328 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
17329 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
17330 {

```

```

17331 \hbox_set:Nn #1
17332 {
17333   \group_begin:
17334     \__box_resize_set_corners:N #1
17335     \fp_set:Nn \l__box_scale_x_fp
17336       { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
17337     \fp_set:Nn \l__box_scale_y_fp
17338       {
17339         \dim_to_fp:n {#3}
17340         / \dim_to_fp:n { \l__box_top_dim }
17341       }
17342     \__box_resize:N #1
17343   \group_end:
17344 }
17345 }
17346 \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }

```

(End definition for `\box_resize_to_ht:Nn` and others. These functions are documented on page 202.)

`\box_scale:Nnn` 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 dimension scaling operations are carried out using the T_EX mechanism as it avoids needing to use too many fp operations.

`\box_scale:cnn`

```

17347 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
17348 {
17349   \hbox_set:Nn #1
17350   {
17351     \group_begin:
17352       \fp_set:Nn \l__box_scale_x_fp {#2}
17353       \fp_set:Nn \l__box_scale_y_fp {#3}
17354       \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
17355       \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
17356       \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
17357       \dim_zero:N \l__box_left_dim
17358       \dim_set:Nn \l__box_top_new_dim
17359         { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
17360       \dim_set:Nn \l__box_bottom_new_dim
17361         { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
17362       \dim_set:Nn \l__box_right_new_dim
17363         { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
17364       \__box_resize_common:N #1
17365     \group_end:
17366   }
17367 }
17368 \cs_generate_variant:Nn \box_scale:Nnn { c }

```

(End definition for `\box_scale:Nnn`. This function is documented on page 203.)

`__box_resize_common:N` The main resize function places in input into a box which will start of with zero width, and includes the handles for engine rescaling.

```

17369 \cs_new_protected:Npn \__box_resize_common:N #1
17370 {
17371   \hbox_set:Nn \l__box_internal_box

```

```

17372 {
17373   \_driver_box_use_scale:Nnn
17374   #1
17375   \l__box_scale_x_fp
17376   \l__box_scale_y_fp
17377 }

```

The new height and depth can be applied directly.

```

17378 \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
17379 {
17380   \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
17381   \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
17382 }
17383 {
17384   \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
17385   \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
17386 }

```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling 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.

```

17387 \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
17388 {
17389   \hbox_to_wd:nn { \l__box_right_new_dim }
17390   {
17391     \tex_kern:D \l__box_right_new_dim
17392     \box_use:N \l__box_internal_box
17393     \tex_hss:D
17394   }
17395 }
17396 {
17397   \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
17398   \hbox:n
17399   {
17400     \tex_kern:D \c_zero_dim
17401     \box_use:N \l__box_internal_box
17402     \tex_hss:D
17403   }
17404 }
17405 }

```

(End definition for _box_resize_common:N.)

35.4 Viewing part of a box

\box_clip:N A wrapper around the driver-dependent code.

```

\box_clip:c 17406 \cs_new_protected:Npn \box_clip:N #1
17407 { \hbox_set:Nn #1 { \_driver_box_use_clip:N #1 } }
17408 \cs_generate_variant:Nn \box_clip:N { c }

```

(End definition for \box_clip:N. This function is documented on page 203.)

`\box_trim:Nnnnn` Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

```

17409 \cs_new_protected:Npn \box_trim:Nnnnn #1#2#3#4#5
17410 {
17411   \hbox_set:Nn \l__box_internal_box
17412   {
17413     \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
17414     \box_use:N #1
17415     \tex_kern:D -\__dim_eval:w #4 \__dim_eval_end:
17416   }

```

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`.

```

17417   \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
17418   {
17419     \hbox_set:Nn \l__box_internal_box
17420     {
17421       \box_move_down:nn \c_zero_dim
17422       { \box_use:N \l__box_internal_box }
17423     }
17424     \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
17425   }
17426   {
17427     \hbox_set:Nn \l__box_internal_box
17428     {
17429       \box_move_down:nn { #3 - \box_dp:N #1 }
17430       { \box_use:N \l__box_internal_box }
17431     }
17432     \box_set_dp:Nn \l__box_internal_box \c_zero_dim
17433   }

```

Same thing, this time from the top of the box.

```

17434   \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
17435   {
17436     \hbox_set:Nn \l__box_internal_box
17437     {
17438       \box_move_up:nn \c_zero_dim
17439       { \box_use:N \l__box_internal_box }
17440     }
17441     \box_set_ht:Nn \l__box_internal_box
17442     { \box_ht:N \l__box_internal_box - (#5) }
17443   }
17444   {
17445     \hbox_set:Nn \l__box_internal_box
17446     {
17447       \box_move_up:nn { #5 - \box_ht:N \l__box_internal_box }
17448       { \box_use:N \l__box_internal_box }
17449     }
17450     \box_set_ht:Nn \l__box_internal_box \c_zero_dim
17451   }

```

```

17452 \box_set_eq:NN #1 \l__box_internal_box
17453 }
17454 \cs_generate_variant:Nn \box_trim:Nnnnn { c }

```

(End definition for `\box_trim:Nnnnn`. This function is documented on page 203.)

`\box_viewport:Nnnnn` 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.

`\box_viewport:cnnnn`

```

17455 \cs_new_protected:Npn \box_viewport:Nnnnn #1#2#3#4#5
17456 {
17457   \hbox_set:Nn \l__box_internal_box
17458   {
17459     \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
17460     \box_use:N #1
17461     \tex_kern:D \__dim_eval:w #4 - \box_wd:N #1 \__dim_eval_end:
17462   }
17463   \dim_compare:nNnTF {#3} < \c_zero_dim
17464   {
17465     \hbox_set:Nn \l__box_internal_box
17466     {
17467       \box_move_down:nn \c_zero_dim
17468       { \box_use:N \l__box_internal_box }
17469     }
17470     \box_set_dp:Nn \l__box_internal_box { -\dim_eval:n {#3} }
17471   }
17472   {
17473     \hbox_set:Nn \l__box_internal_box
17474     { \box_move_down:nn {#3} { \box_use:N \l__box_internal_box } }
17475     \box_set_dp:Nn \l__box_internal_box \c_zero_dim
17476   }
17477   \dim_compare:nNnTF {#5} > \c_zero_dim
17478   {
17479     \hbox_set:Nn \l__box_internal_box
17480     {
17481       \box_move_up:nn \c_zero_dim
17482       { \box_use:N \l__box_internal_box }
17483     }
17484     \box_set_ht:Nn \l__box_internal_box
17485     {
17486       #5
17487       \dim_compare:nNnT {#3} > \c_zero_dim
17488       { - (#3) }
17489     }
17490   }
17491   {
17492     \hbox_set:Nn \l__box_internal_box
17493     {
17494       \box_move_up:nn { -\dim_eval:n {#5} }
17495       { \box_use:N \l__box_internal_box }
17496     }
17497     \box_set_ht:Nn \l__box_internal_box \c_zero_dim
17498   }
17499   \box_set_eq:NN #1 \l__box_internal_box
17500 }
17501 \cs_generate_variant:Nn \box_viewport:Nnnnn { c }

```

(End definition for `\box_viewport:Nnnnn`. This function is documented on page 204.)

35.5 Additions to `l3clist`

17502 `<@@=clist>`

`\clist_log:N` Redirect output of `\clist_show:N` to the log.

`\clist_log:c` 17503 `\cs_new_protected:Npn \clist_log:N`

`\clist_log:n` 17504 `{ __msg_log_next: \clist_show:N }`

17505 `\cs_new_protected:Npn \clist_log:n`

17506 `{ __msg_log_next: \clist_show:n }`

17507 `\cs_generate_variant:Nn \clist_log:N { c }`

(End definition for `\clist_log:N` and `\clist_log:n`. These functions are documented on page 204.)

`\clist_rand_item:n` The N-type function is not implemented through the n-type function for efficiency: for

`\clist_rand_item:N` instance comma-list variables do not require space-trimming of their items. Even testing

`\clist_rand_item:c` for emptiness of an n-type comma-list is slow, so we count items first and use that both

`__clist_rand_item:nn` for the emptiness test and the pseudo-random integer. Importantly, `\clist_item:Nn` and `\clist_item:nn` only evaluate their argument once.

17508 `\cs_new:Npn \clist_rand_item:n #1`

17509 `{ \exp_args:Nf __clist_rand_item:nn { \clist_count:n {#1} } {#1} }`

17510 `\cs_new:Npn __clist_rand_item:nn #1#2`

17511 `{`

17512 `\int_compare:nNf {#1} = \c_zero`

17513 `{ \clist_item:nn {#2} { \int_rand:nn { 1 } {#1} } }`

17514 `}`

17515 `\cs_new:Npn \clist_rand_item:N #1`

17516 `{`

17517 `\clist_if_empty:NF #1`

17518 `{ \clist_item:Nn #1 { \int_rand:nn { 1 } { \clist_count:N #1 } } }`

17519 `}`

17520 `\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 204.)

35.6 Additions to `l3coffins`

17521 `<@@=coffin>`

35.7 Rotating coffins

`\l__coffin_sin_fp` Used for rotations to get the sine and cosine values.

`\l__coffin_cos_fp` 17522 `\fp_new:N \l__coffin_sin_fp`

17523 `\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.

17524 `\prop_new:N \l__coffin_bounding_prop`

(End definition for `\l__coffin_bounding_prop`.)

`\l__coffin_bounding_shift_dim` The shift of the bounding box of a coffin from the real content.

```
17525 \dim_new:N \l__coffin_bounding_shift_dim
```

(End definition for \l__coffin_bounding_shift_dim.)

`\l__coffin_left_corner_dim` These are used to hold maxima for the various corner values: these thus define the
`\l__coffin_right_corner_dim` minimum size of the bounding box after rotation.

```
\l__coffin_bottom_corner_dim 17526 \dim_new:N \l__coffin_left_corner_dim
\l__coffin_top_corner_dim     17527 \dim_new:N \l__coffin_right_corner_dim
                               17528 \dim_new:N \l__coffin_bottom_corner_dim
                               17529 \dim_new:N \l__coffin_top_corner_dim
```

(End definition for \l__coffin_left_corner_dim and others.)

`\coffin_rotate:Nn` Rotating a coffin requires several steps which can be conveniently run together. The sine
`\coffin_rotate:cn` and cosine of the angle in degrees are computed. This is then used to set `\l__coffin_sin_fp`
and `\l__coffin_cos_fp`, which are carried through unchanged for the rest of the procedure.

```
17530 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
17531 {
17532   \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
17533   \fp_set:Nn \l__coffin_cos_fp { cosd ( #2 ) }
```

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```
17534   \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
17535   { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
17536   \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
17537   { \__coffin_rotate_pole:Nnnnnn #1 {##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.

```
17538   \__coffin_set_bounding:N #1
17539   \prop_map_inline:Nn \l__coffin_bounding_prop
17540   { \__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.

```
17541   \__coffin_find_corner_maxima:N #1
17542   \__coffin_find_bounding_shift:
17543   \box_rotate:Nn #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.

```
17544   \hbox_set:Nn \l__coffin_internal_box
17545   {
17546     \tex_kern:D
17547     \__dim_eval:w
17548     \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim
17549     \__dim_eval_end:
```



```

17550         \box_move_down:nn { \l__coffin_bottom_corner_dim }
17551         { \box_use:N #1 }
17552     }

```

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.

```

17553     \box_set_ht:Nn \l__coffin_internal_box
17554     { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
17555     \box_set_dp:Nn \l__coffin_internal_box { 0 pt }
17556     \box_set_wd:Nn \l__coffin_internal_box
17557     { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
17558     \hbox_set:Nn #1 { \box_use: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.

```

17559     \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
17560     { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
17561     \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
17562     { \__coffin_shift_pole:Nnnnnn #1 {##1} ##2 }
17563 }
17564 \cs_generate_variant:Nn \coffin_rotate:Nn { c }

```

(End definition for \coffin_rotate:Nn. This function is documented on page 204.)

`__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.

```

17565 \cs_new_protected:Npn \__coffin_set_bounding:N #1
17566 {
17567     \prop_put:Nnx \l__coffin_bounding_prop { tl }
17568     { { 0 pt } { \dim_eval:n { \box_ht:N #1 } } }
17569     \prop_put:Nnx \l__coffin_bounding_prop { tr }
17570     { { \dim_eval:n { \box_wd:N #1 } } { \dim_eval:n { \box_ht:N #1 } } }
17571     \dim_set:Nn \l__coffin_internal_dim { -\box_dp:N #1 }
17572     \prop_put:Nnx \l__coffin_bounding_prop { bl }
17573     { { 0 pt } { \dim_use:N \l__coffin_internal_dim } }
17574     \prop_put:Nnx \l__coffin_bounding_prop { br }
17575     { { \dim_eval:n { \box_wd:N #1 } } { \dim_use:N \l__coffin_internal_dim } }
17576 }

```

(End definition for __coffin_set_bounding:N.)

`__coffin_rotate_bounding:nnn` 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.

```

17577 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
17578 {
17579     \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
17580     \prop_put:Nnx \l__coffin_bounding_prop {#1}
17581     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
17582 }
17583 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
17584 {

```

```

17585     \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
17586     \prop_put:cnx { \l__coffin_corners_ \__int_value:w #1 _prop } {#2}
17587     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
17588 }

```

(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.

```

17589 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
17590 {
17591     \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
17592     \__coffin_rotate_vector:nnNN {#5} {#6}
17593     \l__coffin_x_prime_dim \l__coffin_y_prime_dim
17594     \__coffin_set_pole:Nnx #1 {#2}
17595     {
17596         { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
17597         { \dim_use:N \l__coffin_x_prime_dim }
17598         { \dim_use:N \l__coffin_y_prime_dim }
17599     }
17600 }

```

(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.

```

17601 \cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
17602 {
17603     \dim_set:Nn #3
17604     {
17605         \fp_to_dim:n
17606         {
17607             \dim_to_fp:n {#1} * \l__coffin_cos_fp
17608             - \dim_to_fp:n {#2} * \l__coffin_sin_fp
17609         }
17610     }
17611     \dim_set:Nn #4
17612     {
17613         \fp_to_dim:n
17614         {
17615             \dim_to_fp:n {#1} * \l__coffin_sin_fp
17616             + \dim_to_fp:n {#2} * \l__coffin_cos_fp
17617         }
17618     }
17619 }

```

(End definition for __coffin_rotate_vector:nnNN.)

__coffin_find_corner_maxima:N 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.

__coffin_find_corner_maxima_aux:nn

```

17620 \cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
17621 {
17622   \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
17623   \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
17624   \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
17625   \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
17626   \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
17627     { \__coffin_find_corner_maxima_aux:nn ##2 }
17628 }
17629 \cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
17630 {
17631   \dim_set:Nn \l__coffin_left_corner_dim
17632     { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
17633   \dim_set:Nn \l__coffin_right_corner_dim
17634     { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
17635   \dim_set:Nn \l__coffin_bottom_corner_dim
17636     { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
17637   \dim_set:Nn \l__coffin_top_corner_dim
17638     { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
17639 }

```

(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.

```

17640 \cs_new_protected:Npn \__coffin_find_bounding_shift:
17641 {
17642   \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
17643   \prop_map_inline:Nn \l__coffin_bounding_prop
17644     { \__coffin_find_bounding_shift_aux:nn ##2 }
17645 }
17646 \cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
17647 {
17648   \dim_set:Nn \l__coffin_bounding_shift_dim
17649     { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
17650 }

```

(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.

```

17651 \cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
17652 {
17653   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
17654   {
17655     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
17656     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
17657   }
17658 }
17659 \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
17660 {
17661   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } {#2}

```

```

17662     {
17663         { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
17664         { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } } }
17665     {#5} {#6}
17666 }
17667 }

```

(End definition for __coffin_shift_corner:Nnnn and __coffin_shift_pole:Nnnnnn.)

35.8 Resizing coffins

\l__coffin_scale_x_fp Storage for the scaling factors in x and y , respectively.
\l__coffin_scale_y_fp

```

17668 \fp_new:N \l__coffin_scale_x_fp
17669 \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 When scaling, the values given have to be turned into absolute values.
\l__coffin_scaled_width_dim

```

17670 \dim_new:N \l__coffin_scaled_total_height_dim
17671 \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 Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.
\coffin_resize:cnn

```

17672 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
17673 {
17674     \fp_set:Nn \l__coffin_scale_x_fp
17675     { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
17676     \fp_set:Nn \l__coffin_scale_y_fp
17677     {
17678         \dim_to_fp:n {#3}
17679         / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
17680     }
17681     \box_resize:Nnn #1 {#2} {#3}
17682     \__coffin_resize_common:Nnn #1 {#2} {#3}
17683 }
17684 \cs_generate_variant:Nn \coffin_resize:Nnn { c }

```

(End definition for \coffin_resize:Nnn. This function is documented on page 204.)

__coffin_resize_common:Nnn The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```

17685 \cs_new_protected:Npn \__coffin_resize_common:Nnn #1#2#3
17686 {
17687     \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
17688     { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
17689     \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
17690     { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }

```

Negative x -scaling values will place the poles in the wrong location: this is corrected here.

```

17691 \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
17692 {
17693   \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
17694   { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
17695   \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
17696   { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
17697 }
17698 }

```

(End definition for `__coffin_resize_common:Nnn`.)

`\coffin_scale:Nnn` For scaling, the opposite calculation is done to find the new dimensions for the coffin.
`\coffin_scale:cnm` Only the total height is needed, as this is the shift required for corners and poles. The scaling is done the T_EX way as this works properly with floating point values without needing to use the `fp` module.

```

17699 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
17700 {
17701   \fp_set:Nn \l__coffin_scale_x_fp {#2}
17702   \fp_set:Nn \l__coffin_scale_y_fp {#3}
17703   \box_scale:Nnn #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
17704   \dim_set:Nn \l__coffin_internal_dim
17705   { \coffin_ht:N #1 + \coffin_dp:N #1 }
17706   \dim_set:Nn \l__coffin_scaled_total_height_dim
17707   { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
17708   \dim_set:Nn \l__coffin_scaled_width_dim
17709   { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
17710   \__coffin_resize_common:Nnn #1
17711   { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
17712 }
17713 \cs_generate_variant:Nn \coffin_scale:Nnn { c }

```

(End definition for `\coffin_scale:Nnn`. This function is documented on page 204.)

`__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.

```

17714 \cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
17715 {
17716   \dim_set:Nn #3
17717   { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
17718   \dim_set:Nn #4
17719   { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
17720 }

```

(End definition for `__coffin_scale_vector:nnNN`.)

`__coffin_scale_corner:Nnnn` Scaling both corners and poles is a simple calculation using the preceding vector scaling.
`__coffin_scale_pole:Nnnnnn`

```

17721 \cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
17722 {
17723   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
17724   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
17725   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }

```

```

17726 }
17727 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
17728 {
17729   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
17730   \__coffin_set_pole:Nnx #1 {#2}
17731   {
17732     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
17733     {#5} {#6}
17734   }
17735 }

```

(End definition for __coffin_scale_corner:Nnnn and __coffin_scale_pole:Nnnnnn.)

__coffin_x_shift_corner:Nnnn These functions correct for the x displacement that takes place with a negative horizontal
 __coffin_x_shift_pole:Nnnnnn scaling.

```

17736 \cs_new_protected:Npn \__coffin_x_shift_corner:Nnnn #1#2#3#4
17737 {
17738   \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
17739   {
17740     { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
17741   }
17742 }
17743 \cs_new_protected:Npn \__coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
17744 {
17745   \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } {#2}
17746   {
17747     { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
17748     {#5} {#6}
17749   }
17750 }

```

(End definition for __coffin_x_shift_corner:Nnnn and __coffin_x_shift_pole:Nnnnnn.)

35.9 Coffin diagnostics

\coffin_log_structure:N Redirect output of \coffin_show_structure:N to the log.

```

\coffin_log_structure:c
17751 \cs_new_protected:Npn \coffin_log_structure:N
17752 { \__msg_log_next: \coffin_show_structure:N }
17753 \cs_generate_variant:Nn \coffin_log_structure:N { c }

```

(End definition for \coffin_log_structure:N. This function is documented on page 205.)

35.10 Additions to l3file

```

17754 <@@=file>

```

\file_if_exist_input:nTF Input of a file with a test for existence cannot be done the usual way as the tokens to insert are in an odd place.

```

17755 \cs_new_protected:Npn \file_if_exist_input:n #1
17756 {
17757   \file_if_exist:nT {#1}
17758   { \__file_input:V \l__file_internal_name_tl }
17759 }
17760 \cs_new_protected:Npn \file_if_exist_input:nT #1#2

```

```

17761 {
17762     \file_if_exist:nT {#1}
17763     {
17764         #2
17765         \__file_input:V \l__file_internal_name_tl
17766     }
17767 }
17768 \cs_new_protected:Npn \file_if_exist_input:nF #1
17769 {
17770     \file_if_exist:nTF {#1}
17771     { \__file_input:V \l__file_internal_name_tl }
17772 }
17773 \cs_new_protected:Npn \file_if_exist_input:nTF #1#2
17774 {
17775     \file_if_exist:nTF {#1}
17776     {
17777         #2
17778         \__file_input:V \l__file_internal_name_tl
17779     }
17780 }

```

(End definition for \file_if_exist_input:nTF. This function is documented on page 205.)

```

17781 <@@=ior>

```

\ior_map_break: Usual map breaking functions. Those are not yet in l3kernel proper since the mapping below is the first of its kind.

\ior_map_break:n

```

17782 \cs_new:Npn \ior_map_break:
17783 { \__prg_map_break:Nn \ior_map_break: { } }
17784 \cs_new:Npn \ior_map_break:n
17785 { \__prg_map_break:Nn \ior_map_break: }

```

(End definition for \ior_map_break: and \ior_map_break:n. These functions are documented on page 206.)

\ior_map_inline:Nn

\ior_str_map_inline:Nn

__ior_map_inline:NNn

__ior_map_inline:NNNn

__ior_map_inline_loop:NNN

\l__ior_internal_tl

Mapping to an input stream can be done on either a token or a string basis, hence the set up. Within that, there is a check to avoid reading past the end of a file, hence the two applications of \ior_if_eof:N. This mapping cannot be nested as the stream has only one “current line”.

```

17786 \cs_new_protected:Npn \ior_map_inline:Nn
17787 { \__ior_map_inline:NNn \ior_get:NN }
17788 \cs_new_protected:Npn \ior_str_map_inline:Nn
17789 { \__ior_map_inline:NNn \ior_str_get:NN }
17790 \cs_new_protected:Npn \__ior_map_inline:NNn
17791 {
17792     \int_gincr:N \g__prg_map_int
17793     \exp_args:Nc \__ior_map_inline:NNNn
17794     { __prg_map_ \int_use:N \g__prg_map_int :n }
17795 }
17796 \cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
17797 {
17798     \cs_set:Npn #1 ##1 {#4}
17799     \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
17800     \__prg_break_point:Nn \ior_map_break:
17801     { \int_gdecr:N \g__prg_map_int }

```

```

17802     }
17803     \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
17804     {
17805         #2 #3 \l__ior_internal_tl
17806         \ior_if_eof:NF #3
17807         {
17808             \exp_args:No #1 \l__ior_internal_tl
17809             \__ior_map_inline_loop:NNN #1#2#3
17810         }
17811     }
17812     \tl_new:N \l__ior_internal_tl

```

(End definition for \ior_map_inline:Nn and others. These functions are documented on page 205.)

\ior_log_streams: Redirect output of \ior_list_streams: to the log.

```

17813     \cs_new_protected:Npn \ior_log_streams:
17814     { \__msg_log_next: \ior_list_streams: }

```

(End definition for \ior_log_streams:. This function is documented on page 206.)

```

17815     <@@=iow>

```

\iow_log_streams: Redirect output of \iow_list_streams: to the log.

```

17816     \cs_new_protected:Npn \iow_log_streams:
17817     { \__msg_log_next: \iow_list_streams: }

```

(End definition for \iow_log_streams:. This function is documented on page 206.)

35.11 Additions to l3fp-assign

```

17818     <@@=fp>

```

\fp_log:N Redirect output of \fp_show:N to the log.

```

\fp_log:c 17819     \cs_new_protected:Npn \fp_log:N
\fp_log:n 17820     { \__msg_log_next: \fp_show:N }
17821     \cs_new_protected:Npn \fp_log:n
17822     { \__msg_log_next: \fp_show:n }
17823     \cs_generate_variant:Nn \fp_log:N { c }

```

(End definition for \fp_log:N and \fp_log:n. These functions are documented on page 207.)

35.12 Additions to l3int

```

17824     <@@=int>

```

\int_log:N Redirect output of \int_show:N to the log. This is not just a copy of __kernel_register_log:N because of subtleties involving \currentgrouplevel and \currentgrouptype. See \int_show:N for details.

```

17825     \cs_new_protected:Npn \int_log:N
17826     { \__msg_log_next: \int_show:N }
17827     \cs_generate_variant:Nn \int_log:N { c }

```

(End definition for \int_log:N. This function is documented on page 207.)

\int_log:n Redirect output of \int_show:n to the log.

```

17828     \cs_new_protected:Npn \int_log:n
17829     { \__msg_log_next: \int_show:n }

```


(End definition for `\int_log:n`. This function is documented on page 207.)

`\int_rand:nn` 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_fp_rand_size_int`; #2-#1 may overflow for very large positive #2 and negative #1. If the range is wide, use slower code from `l3fp`. If the range is narrow, call `__int_rand_narrow:nn` $\langle\textit{choices}\rangle$ {#1} where $\langle\textit{choices}\rangle$ is the number of possible outcomes. Then `__int_rand_narrow:nnnn` receives a random number reduced modulo $\langle\textit{choices}\rangle$, the random number itself, $\langle\textit{choices}\rangle$ and #1. To avoid bias, throw away the random number if it lies in the last, incomplete, interval of size $\langle\textit{choices}\rangle$ in $[0, \text{c_fp_rand_size_int} - 1]$, and try again.

```

17830 \cs_if_exist:NTF \pdfTeX_uniformdeviate:D
17831 {
17832   \cs_new:Npn \int_rand:nn #1#2
17833   {
17834     \exp_after:wN \__int_rand:ww
17835     \__int_value:w \__int_eval:w #1 \exp_after:wN ;
17836     \__int_value:w \__int_eval:w #2 ;
17837   }
17838   \cs_new:Npn \__int_rand:ww #1; #2;
17839   {
17840     \int_compare:nNnTF {#1} > {#2}
17841     {
17842       \__msg_kernel_expandable_error:nnnn
17843       { kernel } { backward-range } {#1} {#2}
17844       \__int_rand:ww #2; #1;
17845     }
17846     {
17847       \int_compare:nNnTF {#1} > \c_zero
17848       { \int_compare:nNnTF { #2 - #1 } < \c_fp_rand_size_int }
17849       { \int_compare:nNnTF {#2} < { #1 + \c_fp_rand_size_int } }
17850       {
17851         \exp_args:Nf \__int_rand_narrow:nn
17852         { \int_eval:n { #2 - #1 + \c_one } } {#1}
17853       }
17854       { \fp_to_int:n { randint(#1,#2) } }
17855     }
17856   }
17857   \cs_new:Npn \__int_rand_narrow:nn
17858   {
17859     \exp_args:No \__int_rand_narrow:nnn
17860     { \pdfTeX_uniformdeviate:D \c_fp_rand_size_int }
17861   }
17862   \cs_new:Npn \__int_rand_narrow:nnn #1#2
17863   {
17864     \exp_args:Nf \__int_rand_narrow:nnnn
17865     { \int_mod:nn {#1} {#2} } {#1} {#2}
17866   }
17867   \cs_new:Npn \__int_rand_narrow:nnnn #1#2#3#4
17868   {
17869     \int_compare:nNnTF { #2 - #1 + #3 } > \c_fp_rand_size_int
17870     { \__int_rand_narrow:nn {#3} {#4} }
17871     { \int_eval:n { #4 + #1 } }

```

```

17872     }
17873   }
17874   {
17875     \cs_new:Npn \int_rand:nn #1#2
17876     {
17877       \__msg_kernel_expandable_error:nn { kernel } { fp-no-random }
17878       \int_eval:n {#1}
17879     }
17880   }

```

(End definition for \int_rand:nn and others. These functions are documented on page 207.)

The following must be added to l3msg.

```

17881 \cs_if_exist:NT \pdfTeX_uniformdeviate:D
17882   {
17883     \__msg_kernel_new:nnn { kernel } { backward-range }
17884     { Bounds~ordered~backwards~in~\int_rand:nn {#1}~{#2}. }
17885   }

```

35.13 Additions to l3keys

```

17886 <@@=keys>

```

\keys_log:nn Redirect output of \keys_show:nn to the log.

```

17887 \cs_new_protected:Npn \keys_log:nn
17888   { \__msg_log_next: \keys_show:nn }

```

(End definition for \keys_log:nn. This function is documented on page 207.)

35.14 Additions to l3msg

```

17889 <@@=msg>

```

Pass to an auxiliary the message to display and the module name

```

\msg_expandable_error:nnnnnn \msg_expandable_error:nnffff
\msg_expandable_error:nnnnnn \msg_expandable_error:nnfff
\msg_expandable_error:nnfff \msg_expandable_error:nnnn
\msg_expandable_error:nnnn \msg_expandable_error:nnff
\msg_expandable_error:nnff \msg_expandable_error:nnn
\msg_expandable_error:nnn \msg_expandable_error:nn
\__msg_expandable_error_module:nn
17890 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
17891   {
17892     \exp_args:Nf \__msg_expandable_error_module:nn
17893     {
17894       \exp_args:Nf \tl_to_str:n
17895       { \use:c { \c_msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
17896     }
17897     {#1}
17898   }
17899 \cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
17900   { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
17901 \cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
17902   { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
17903 \cs_new:Npn \msg_expandable_error:nnn #1#2#3
17904   { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
17905 \cs_new:Npn \msg_expandable_error:nn #1#2
17906   { \msg_expandable_error:nnnnnn {#1} {#2} { } { } { } { } }
17907 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnffff }
17908 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnfff }
17909 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnff }
17910 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnf }
17911 \cs_new:Npn \__msg_expandable_error_module:nn #1#2

```

```

17912 {
17913   \exp_after:wN \exp_after:wN
17914   \exp_after:wN \use_none_delimit_by_q_stop:w
17915   \use:n { \::error ! ~ #2 : ~ #1 } \q_stop
17916 }

```

(End definition for \msg_expandable_error:nnnnnn and others. These functions are documented on page 208.)

35.15 Additions to l3prg

```

17917 <@@=bool>

```

\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:nTF

__bool_lazy_all:n

```

17918 \prg_new_conditional:Npnn \bool_lazy_all:n #1 { p , T , F , TF }
17919 { \__bool_lazy_all:n #1 \q_recursion_tail \q_recursion_stop }
17920 \cs_new:Npn \__bool_lazy_all:n #1
17921 {
17922   \quark_if_recursion_tail_stop_do:nn {#1} { \prg_return_true: }
17923   \bool_if:nF {#1}
17924   { \use_i_delimit_by_q_recursion_stop:nw { \prg_return_false: } }
17925   \__bool_lazy_all:n
17926 }

```

(End definition for \bool_lazy_all:nTF and __bool_lazy_all:n. These functions are documented on page 208.)

\bool_lazy_and_p:n Only evaluate the second expression if the first is **true**.

\bool_lazy_and:nnTF

```

17927 \prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }
17928 {
17929   \bool_if:nTF {#1}
17930   { \bool_if:nTF {#2} { \prg_return_true: } { \prg_return_false: } }
17931   { \prg_return_false: }
17932 }

```

(End definition for \bool_lazy_and:nnTF. This function is documented on page 209.)

\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:nTF

__bool_lazy_any:n

```

17933 \prg_new_conditional:Npnn \bool_lazy_any:n #1 { p , T , F , TF }
17934 { \__bool_lazy_any:n #1 \q_recursion_tail \q_recursion_stop }
17935 \cs_new:Npn \__bool_lazy_any:n #1
17936 {
17937   \quark_if_recursion_tail_stop_do:nn {#1} { \prg_return_false: }
17938   \bool_if:nT {#1}
17939   { \use_i_delimit_by_q_recursion_stop:nw { \prg_return_true: } }
17940   \__bool_lazy_any:n
17941 }

```

(End definition for \bool_lazy_any:nTF and __bool_lazy_any:n. These functions are documented on page 209.)

\bool_lazy_or_p:nn Only evaluate the second expression if the first is false.
\bool_lazy_or:nnTF

```

17942 \prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
17943 {
17944   \bool_if:nTF {#1}
17945     { \prg_return_true: }
17946     { \bool_if:nTF {#2} { \prg_return_true: } { \prg_return_false: } }
17947 }

```

(End definition for `\bool_lazy_or:nnTF`. This function is documented on page 209.)

\bool_log:N Redirect output of `\bool_show:N` to the log.
\bool_log:c
\bool_log:n

```

17948 \cs_new_protected:Npn \bool_log:N
17949 { \__msg_log_next: \bool_show:N }
17950 \cs_new_protected:Npn \bool_log:n
17951 { \__msg_log_next: \bool_show:n }
17952 \cs_generate_variant:Nn \bool_log:N { c }

```

(End definition for `\bool_log:N` and `\bool_log:n`. These functions are documented on page 209.)

35.16 Additions to `l3prop`

17953 `<@@=prop>`

\prop_count:N Counting the key–value pairs in a property list is done using the same approach as for
\prop_count:c other count functions: turn each entry into a +1 then use integer evaluation to actually
__prop_count:nn do the mathematics.

```

17954 \cs_new:Npn \prop_count:N #1
17955 {
17956   \int_eval:n
17957     {
17958       \c_zero
17959       \prop_map_function:NN #1 \__prop_count:nn
17960     }
17961 }
17962 \cs_new:Npn \__prop_count:nn #1#2 { + \c_one }
17963 \cs_generate_variant:Nn \prop_count:N { c }

```

(End definition for `\prop_count:N` and `__prop_count:nn`. These functions are documented on page 209.)

\prop_map_tokens:Nn The mapping is very similar to `\prop_map_function:NN`. It grabs one key–value pair
\prop_map_tokens:cn at a time, and stops when reaching the marker key `\q_recursion_tail`, which cannot
__prop_map_tokens:nwn appear in normal keys since those are strings. The odd construction `\use:n {#1}` allows
#1 to contain any token without interfering with `\prop_map_break:.` Argument **#2** of
__prop_map_tokens:nwn is `\s__prop` the first time, and is otherwise empty.

```

17964 \cs_new:Npn \prop_map_tokens:Nn #1#2
17965 {
17966   \exp_last_unbraced:Nno \__prop_map_tokens:nwn {#2} #1
17967   \__prop_pair:wn \q_recursion_tail \s__prop { }
17968   \__prg_break_point:Nn \prop_map_break: { }
17969 }
17970 \cs_new:Npn \__prop_map_tokens:nwn #1#2 \__prop_pair:wn #3 \s__prop #4
17971 {
17972   \if_meaning:w \q_recursion_tail #3

```

```

17973     \exp_after:wN \prop_map_break:
17974     \fi:
17975     \use:n {#1} {#3} {#4}
17976     \__prop_map_tokens:nwn {#1}
17977   }
17978   \cs_generate_variant:Nn \prop_map_tokens:Nn { c }

```

(End definition for `\prop_map_tokens:Nn` and `__prop_map_tokens:nwn`. These functions are documented on page 209.)

`\prop_log:N` Redirect output of `\prop_show:N` to the log.

```

\prop_log:c 17979 \cs_new_protected:Npn \prop_log:N
17980   { \__msg_log_next: \prop_show:N }
17981 \cs_generate_variant:Nn \prop_log:N { c }

```

(End definition for `\prop_log:N`. This function is documented on page 210.)

`\prop_rand_key_value:N` Contrarily to `clist`, `seq` and `tl`, 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. At the end, leave either the key #3 or the value #4 in the input stream.

```

\prop_rand_key_value:c 17982 \cs_new:Npn \prop_rand_key_value:N { \__prop_rand:NN \__prop_rand:nNn }
17983 \cs_new:Npn \__prop_rand:nNn #1#2#3 { \exp_not:n { {#1} {#3} } }
17984 \cs_new:Npn \__prop_rand:NN #1#2
17985   {
17986     \prop_if_empty:NTF #2 { }
17987     {
17988       \exp_after:wN \__prop_rand_item:Nw \exp_after:wN #1
17989       \__int_value:w \int_rand:nn { 1 } { \prop_count:N #2 } #2
17990       \q_stop
17991     }
17992   }
17993 \cs_new:Npn \__prop_rand_item:Nw #1#2 \s__prop \__prop_pair:wn #3 \s__prop #4
17994   {
17995     \int_compare:nNnF {#2} > \c_one
17996     { \use_i_delimit_by_q_stop:nw { #1 {#3} \exp_not:n {#4} } }
17997     \exp_after:wN \__prop_rand_item:Nw \exp_after:wN #1
17998     \__int_value:w \int_eval:n { #2 - \c_one } \s__prop
17999   }
18000 \cs_generate_variant:Nn \prop_rand_key_value:N { c }

```

(End definition for `\prop_rand_key_value:N`, `__prop_rand:NN`, and `__prop_rand_item:Nw`. These functions are documented on page 210.)

35.17 Additions to `l3seq`

```

18001 <@@=seq>

```

`\seq_mapthread_function:NNN` 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 the sequences will both be `\s__seq __seq_item:n`. The function to be mapped will then be applied to the two entries. When the code hits the end of one of the sequences, the break material will stop the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```

18002 \cs_new:Npn \seq_mapthread_function:NNN #1#2#3
18003 { \exp_after:wN \__seq_mapthread_function:wNN #2 \q_stop #1 #3 }
18004 \cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \q_stop #2#3
18005 {
18006   \exp_after:wN \__seq_mapthread_function:wNw #2 \q_stop #3
18007   #1 { ? \__prg_break: } { }
18008   \__prg_break_point:
18009 }
18010 \cs_new:Npn \__seq_mapthread_function:wNw \s__seq #1 \q_stop #2
18011 {
18012   \__seq_mapthread_function:Nnnwnn #2
18013   #1 { ? \__prg_break: } { }
18014   \q_stop
18015 }
18016 \cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \q_stop #5#6
18017 {
18018   \use_none:n #2
18019   \use_none:n #5
18020   #1 {#3} {#6}
18021   \__seq_mapthread_function:Nnnwnn #1 #4 \q_stop
18022 }
18023 \cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc }
18024 \cs_generate_variant:Nn \seq_mapthread_function:NNN { c , cc }

```

(End definition for `\seq_mapthread_function:NNN` and others. These functions are documented on page 210.)

`\seq_set_filter:NNn` 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.

`\seq_gset_filter:NNn`
`__seq_set_filter:NNNn`

```

18025 \cs_new_protected:Npn \seq_set_filter:NNn
18026 { \__seq_set_filter:NNNn \tl_set:Nx }
18027 \cs_new_protected:Npn \seq_gset_filter:NNn
18028 { \__seq_set_filter:NNNn \tl_gset:Nx }
18029 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
18030 {
18031   \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
18032   #1 #2 { #3 }
18033   \__seq_pop_item_def:
18034 }

```

(End definition for `\seq_set_filter:NNn`, `\seq_gset_filter:NNn`, and `__seq_set_filter:NNNn`. These functions are documented on page 210.)

`\seq_set_map:NNn` Very similar to `\seq_set_filter:NNn`. We could actually merge the two within a single function, but it would have weird semantics.

`\seq_gset_map:NNn`
`__seq_set_map:NNNn`

```

18035 \cs_new_protected:Npn \seq_set_map:NNn
18036 { \__seq_set_map:NNNn \tl_set:Nx }
18037 \cs_new_protected:Npn \seq_gset_map:NNn
18038 { \__seq_set_map:NNNn \tl_gset:Nx }
18039 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
18040 {
18041   \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }

```

```

18042     #1 #2 { #3 }
18043     \__seq_pop_item_def:
18044 }

```

(End definition for `\seq_set_map:NNn`, `\seq_gset_map:NNn`, and `__seq_set_map:NNNn`. These functions are documented on page 210.)

`\seq_log:N` Redirect output of `\seq_show:N` to the log.

```

\seq_log:c
18045 \cs_new_protected:Npn \seq_log:N
18046 { \__msg_log_next: \seq_show:N }
18047 \cs_generate_variant:Nn \seq_log:N { c }

```

(End definition for `\seq_log:N`. This function is documented on page 210.)

`\seq_rand_item:N` Importantly, `\seq_item:Nn` only evaluates its argument once.

```

\seq_rand_item:c
18048 \cs_new:Npn \seq_rand_item:N #1
18049 {
18050     \seq_if_empty:NF #1
18051     { \seq_item:Nn #1 { \int_rand:nn { 1 } { \seq_count:N #1 } } }
18052 }
18053 \cs_generate_variant:Nn \seq_rand_item:N { c }

```

(End definition for `\seq_rand_item:N`. This function is documented on page 211.)

35.18 Additions to `l3skip`

```

18054 <@@=skip>

```

`\skip_split_finite_else_action:nnNN` This macro is useful when performing error checking in certain circumstances. If the `<skip>` register holds finite glue it sets `#3` and `#4` to the stretch and shrink component, resp. If it holds infinite glue set `#3` and `#4` to zero and issue the special action `#2` which is probably an error message. Assignments are local.

```

18055 \cs_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
18056 {
18057     \skip_if_finite:nTF {#1}
18058     {
18059         #3 = \etex_gluestretch:D #1 \scan_stop:
18060         #4 = \etex_glueshrink:D #1 \scan_stop:
18061     }
18062     {
18063         #3 = \c_zero_skip
18064         #4 = \c_zero_skip
18065         #2
18066     }
18067 }

```

(End definition for `\skip_split_finite_else_action:nnNN`. This function is documented on page 211.)

`\dim_log:N` Diagnostics. Redirect output of `\dim_show:n` to the log.

```

\dim_log:c
18068 \cs_new_eq:NN \dim_log:N \__kernel_register_log:N
\dim_log:n
18069 \cs_new_eq:NN \dim_log:c \__kernel_register_log:c
18070 \cs_new_protected:Npn \dim_log:n
18071 { \__msg_log_next: \dim_show:n }

```

(End definition for `\dim_log:N` and `\dim_log:n`. These functions are documented on page 211.)

\skip_log:N Diagnostics. Redirect output of \skip_show:n to the log.
\skip_log:c 18072 \cs_new_eq:NN \skip_log:N __kernel_register_log:N
\skip_log:n 18073 \cs_new_eq:NN \skip_log:c __kernel_register_log:c
18074 \cs_new_protected:Npn \skip_log:n
18075 { __msg_log_next: \skip_show:n }

(End definition for \skip_log:N and \skip_log:n. These functions are documented on page 211.)

\muskip_log:N Diagnostics. Redirect output of \muskip_show:n to the log.
\muskip_log:c 18076 \cs_new_eq:NN \muskip_log:N __kernel_register_log:N
\muskip_log:n 18077 \cs_new_eq:NN \muskip_log:c __kernel_register_log:c
18078 \cs_new_protected:Npn \muskip_log:n
18079 { __msg_log_next: \muskip_show:n }

(End definition for \muskip_log:N and \muskip_log:n. These functions are documented on page 211.)

35.19 Additions to l3tl

18080 <@@=tl>

\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.

\tl_if_single_token:nTF

18081 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
18082 {
18083 \tl_if_head_is_N_type:nTF {#1}
18084 { __tl_if_empty_return:o { \use_none:n #1 } }
18085 {
18086 \tl_if_empty:nTF {#1}
18087 { \prg_return_false: }
18088 { __tl_if_empty_return:o { \exp:w \exp_end_continue_f:w #1 } }
18089 }
18090 }

(End definition for \tl_if_single_token:nTF. This function is documented on page 212.)

\tl_reverse_tokens:n The same as \tl_reverse:n but with recursion within brace groups.
__tl_reverse_group:nn

18091 \cs_new:Npn \tl_reverse_tokens:n #1
18092 {
18093 \etex_unexpanded:D \exp_after:wN
18094 {
18095 \exp:w
18096 __tl_act:NNNnn
18097 __tl_reverse_normal:nN
18098 __tl_reverse_group:nn
18099 __tl_reverse_space:n
18100 { }
18101 {#1}
18102 }
18103 }
18104 \cs_new:Npn __tl_reverse_group:nn #1
18105 {


```

18106 \tl_act_group_recurse:Nnn
18107 \tl_act_reverse_output:n
18108 { \tl_reverse_tokens:n }
18109 }

```

In many applications of `\tl_act:NNNnn`, we need to recursively apply some transformation within brace groups, then output. In this code, #1 is the output function, #2 is the transformation, which should expand in two steps, and #3 is the group.

```

18110 \cs_new:Npn \tl_act_group_recurse:Nnn #1#2#3
18111 {
18112   \exp_args:Nf #1
18113   { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
18114 }

```

(End definition for `\tl_reverse_tokens:n`, `\tl_reverse_group:nn`, and `\tl_act_group_recurse:Nnn`. These functions are documented on page 212.)

```

\tl_count_tokens:n
\tl_act_count_normal:nN
\tl_act_count_group:nn
\tl_act_count_space: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`). Somewhat a hack!

```

18115 \cs_new:Npn \tl_count_tokens:n #1
18116 {
18117   \int_eval:n
18118   {
18119     \tl_act:NNNnn
18120     \tl_act_count_normal:nN
18121     \tl_act_count_group:nn
18122     \tl_act_count_space:n
18123     { }
18124     {#1}
18125   }
18126 }
18127 \cs_new:Npn \tl_act_count_normal:nN #1 #2 { 1 + }
18128 \cs_new:Npn \tl_act_count_space:n #1 { 1 + }
18129 \cs_new:Npn \tl_act_count_group:nn #1 #2
18130 { 2 + \tl_count_tokens:n {#2} + }

```

(End definition for `\tl_count_tokens:n` and others. These functions are documented on page 212.)

```

\tl_set_from_file:Nnn
\tl_set_from_file:cnn
\tl_gset_from_file:Nnn
\tl_gset_from_file:cnn
\tl_set_from_file:NNnn
\tl_from_file_do:w

```

The approach here is similar to that for doing a rescan, and so the same internals can be reused. Thus the plan is to insert a pair of tokens of the same charcode but different catcodes after the file has been read. This plus `\exp_not:N` allows the primitive to be used to carry out a set operation.

```

18131 \cs_new_protected:Npn \tl_set_from_file:Nnn
18132 { \tl_set_from_file:NNnn \tl_set:Nn }
18133 \cs_new_protected:Npn \tl_gset_from_file:Nnn
18134 { \tl_set_from_file:NNnn \tl_gset:Nn }
18135 \cs_generate_variant:Nn \tl_set_from_file:Nnn { c }
18136 \cs_generate_variant:Nn \tl_gset_from_file:Nnn { c }
18137 \cs_new_protected:Npn \tl_set_from_file:NNnn #1#2#3#4
18138 {
18139   \__file_if_exist:nT {#4}
18140   {
18141     \group_begin:

```

```

18142         \exp_args:No \etex_everyeof:D
18143         { \c__tl_rescan_marker_tl \exp_not:N }
18144     #3 \scan_stop:
18145     \exp_after:wN \__tl_from_file_do:w
18146     \exp_after:wN \prg_do_nothing:
18147     \tex_input:D \l__file_internal_name_tl \scan_stop:
18148     \exp_args:NNNo \group_end:
18149     #1 #2 \l__tl_internal_a_tl
18150 }
18151 }
18152 \exp_args:Nno \use:nn
18153 { \cs_new_protected:Npn \__tl_from_file_do:w #1 }
18154 { \c__tl_rescan_marker_tl }
18155 { \tl_set:No \l__tl_internal_a_tl {#1} }

```

(End definition for `\tl_set_from_file:Nnn` and others. These functions are documented on page 215.)

`\tl_set_from_file_x:Nnn` When reading a file and allowing expansion of the content, the set up only needs to prevent \TeX complaining about the end of the file. That is done simply, with a group then used to trap the definition needed. Once the business is done using some scratch space, the tokens can be transferred to the real target.

```

\__tl_set_from_file_x:NNnn
18156 \cs_new_protected:Npn \tl_set_from_file_x:Nnn
18157 { \__tl_set_from_file_x:NNnn \tl_set:Nn }
18158 \cs_new_protected:Npn \tl_gset_from_file_x:Nnn
18159 { \__tl_set_from_file_x:NNnn \tl_gset:Nn }
18160 \cs_generate_variant:Nn \tl_set_from_file_x:Nnn { c }
18161 \cs_generate_variant:Nn \tl_gset_from_file_x:Nnn { c }
18162 \cs_new_protected:Npn \__tl_set_from_file_x:NNnn #1#2#3#4
18163 {
18164     \__file_if_exist:nT {#4}
18165     {
18166         \group_begin:
18167         \etex_everyeof:D { \exp_not:N }
18168         #3 \scan_stop:
18169         \tl_set:Nx \l__tl_internal_a_tl
18170         { \tex_input:D \l__file_internal_name_tl \c_space_token }
18171         \exp_args:NNNo \group_end:
18172         #1 #2 \l__tl_internal_a_tl
18173     }
18174 }

```

(End definition for `\tl_set_from_file_x:Nnn`, `\tl_gset_from_file_x:Nnn`, and `__tl_set_from_file_x:NNnn`. These functions are documented on page 215.)

35.19.1 Unicode case changing

The mechanisms needed for case changing are somewhat involved, particularly to allow for all of the special cases. These functions also require the appropriate data extracted from the Unicode documentation (either manually or automatically).

`\tl_if_head_eq_catcode:oNTF` Extra variants.

```

18175 \cs_generate_variant:Nn \tl_if_head_eq_catcode:nNTF { o }

```

(End definition for `\tl_if_head_eq_catcode:oNTF`. This function is documented on page 99.)

`\tl_lower_case:n` The user level functions here are all wrappers around the internal functions for case changing. Note that `\tl_mixed_case:nn` could be done without an internal, but this way the logic is slightly clearer as everything essentially follows the same path.

`\tl_upper_case:n`

`\tl_mixed_case:n`

`\tl_lower_case:nn` 18176 `\cs_new:Npn \tl_lower_case:n { \tl_change_case:nnn { lower } { } }`

`\tl_upper_case:nn` 18177 `\cs_new:Npn \tl_upper_case:n { \tl_change_case:nnn { upper } { } }`

`\tl_mixed_case:nn` 18178 `\cs_new:Npn \tl_mixed_case:n { \tl_mixed_case:nn { } }`

18179 `\cs_new:Npn \tl_lower_case:nn { \tl_change_case:nnn { lower } }`

18180 `\cs_new:Npn \tl_upper_case:nn { \tl_change_case:nnn { upper } }`

18181 `\cs_new:Npn \tl_mixed_case:nn { \tl_mixed_case:nn }`

(End definition for `\tl_lower_case:n` and others. These functions are documented on page 212.)

`\tl_change_case:nnn` The mechanism for the core conversion of case is based on the idea that we can use a loop to grab the entire token list plus a quark: the latter is used as an end marker and to avoid any brace stripping. Depending on the nature of the first item in the grabbed argument, it can either processed as a single token, treated as a group or treated as a space. These different cases all work by re-reading #1 in the appropriate way, hence the repetition of #1 `\q_recursion_stop`.

`\tl_change_case_aux:nnn`

`\tl_change_case_loop:wnn`

`\tl_change_case_output:nwn`

`\tl_change_case_output:Vwn`

`\tl_change_case_output:own`

`\tl_change_case_output:vwn`

`\tl_change_case_output:fw`

`\tl_change_case_end:wn`

`\tl_change_case_group:nwnn`

`\tl_change_case_space:wnn`

`\tl_change_case_N_type:Nwnn`

`\tl_change_case_N_type:NNNnnn`

`\tl_change_case_math:NNNnnn`

`\tl_change_case_math_loop:wNNnn`

`\tl_change_case_math:NwNNnn`

`\tl_change_case_math_group:nwNNnn`

`\tl_change_case_math_space:wNNnn`

`\tl_change_case_N_type:NNnn`

`\tl_change_case_char:Nnn`

`\tl_change_case_char:nN`

`\tl_change_case_char_auxi:nN`

`\tl_change_case_char_auxii:nN`

`\tl_lookup_lower:N`

`\tl_lookup_upper:N`

`\tl_lookup_title:N`

`\tl_change_case_char_UTFviii:nNN`

`\tl_change_case_char_UTFviii:nNNN`

`\tl_change_case_char_UTFviii:nNNNN`

`\tl_change_case_char_UTFviii:nn`

`\tl_change_case_cs_letterlike:Nnn`

`\tl_change_case_cs_accents:NN`

`\tl_change_case_cs:N`

`\tl_change_case_cs:NN`

`\tl_change_case_cs:NNn`

`\tl_change_case_protect:wNN`

`\tl_change_case_if_expandable:NTF`

`\tl_change_case_cs_expand:Nnw`

`\tl_change_case_cs_expand:NN`

18182 `\cs_new:Npn \tl_change_case:nnn #1#2#3`

18183 `{`

18184 `\etex_unexpanded:D \exp_after:wN`

18185 `{`

18186 `\exp:w`

18187 `\tl_change_case_aux:nnn {#1} {#2} {#3}`

18188 `}`

18189 `}`

18190 `\cs_new:Npn \tl_change_case_aux:nnn #1#2#3`

18191 `{`

18192 `\group_align_safe_begin:`

18193 `\tl_change_case_loop:wnn`

18194 `#3 \q_recursion_tail \q_recursion_stop {#1} {#2}`

18195 `\tl_change_case_result:n { }`

18196 `}`

18197 `\cs_new:Npn \tl_change_case_loop:wnn #1 \q_recursion_stop`

18198 `{`

18199 `\tl_if_head_is_N_type:NTF {#1}`

18200 `{ \tl_change_case_N_type:Nwnn }`

18201 `{`

18202 `\tl_if_head_is_group:NTF {#1}`

18203 `{ \tl_change_case_group:nwnn }`

18204 `{ \tl_change_case_space:wnn }`

18205 `}`

18206 `#1 \q_recursion_stop`

18207 `}`

Earlier versions of the code where only x-type expandable rather than f-type: this causes issues with nesting and so the slight performance hit is taken for a better outcome in usability terms. Setting up for f-type expandability has two requirements: a marker token after the main loop (see above) and a mechanism to “load” and finalise the result. That is handled in the code below, which includes the necessary material to end the `\exp:w` expansion.

18208 `\cs_new:Npn \tl_change_case_output:nwn #1#2 \tl_change_case_result:n #3`

18209 `{ #2 \tl_change_case_result:n { #3 #1 } }`

```

18210 \cs_generate_variant:Nn \__tl_change_case_output:nwn { V , o , v , f }
18211 \cs_new:Npn \__tl_change_case_end:wn #1 \__tl_change_case_result:n #2
18212 {
18213   \group_align_safe_end:
18214   \exp_end:
18215   #2
18216 }

```

Handling for the cases where the current argument is a brace group or a space is relatively easy. For the brace case, the routine works recursively, using the expandability of the mechanism to ensure that the result is finalised before storage. For the space case it is simply a question of removing the space in the input and storing it in the output. In both cases, and indeed for the N-type grabber, after removing the current item from the input `__tl_change_case_loop:wnn` is inserted in front of the remaining tokens.

```

18217 \cs_new:Npn \__tl_change_case_group:nwnn #1#2 \q_recursion_stop #3#4
18218 {
18219   \__tl_change_case_output:own
18220   {
18221     \exp_after:wN
18222     {
18223       \exp:w
18224       \__tl_change_case_aux:nnn {#3} {#4} {#1}
18225     }
18226   }
18227   \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
18228 }
18229 \exp_last_unbraced:NNo \cs_new:Npn \__tl_change_case_space:wnn \c_space_tl
18230 {
18231   \__tl_change_case_output:nwn { ~ }
18232   \__tl_change_case_loop:wnn
18233 }

```

For N-type arguments there are several stages to the approach. First, a simply check for the end-of-input marker, which if found triggers the final clean up and output step. Assuming that is not the case, the first check is for math-mode escaping: this test can encompass control sequences or other N-type tokens so is handled up front.

```

18234 \cs_new:Npn \__tl_change_case_N_type:Nwnn #1#2 \q_recursion_stop
18235 {
18236   \quark_if_recursion_tail_stop_do:Nn #1
18237   { \__tl_change_case_end:wn }
18238   \exp_after:wN \__tl_change_case_N_type:NNNnnn
18239   \exp_after:wN #1 \l_tl_change_case_math_tl
18240   \q_recursion_tail ? \q_recursion_stop {#2}
18241 }

```

Looking for math mode escape first requires a loop over the possible token pairs to see if the current input (`#1`) matches an open-math case (`#2`). If it does then this test loop is ended and a new input-gathering one is begun. The latter simply transfers material from the input to the output without any expansion, testing each N-type token to see if it matches the close-math case required. If that is the situation then the “math loop” stops and resumes the main loop: as that might be either the standard case-changing one or the mixed-case alternative, it is not hard-coded into the math loop but is rather passed as argument `#3` to `__tl_change_case_math:NNNnnn`. If no close-math token is found

then the final clean-up will be forced (*i.e.* there is no assumption of “well-behaved” code in terms of math mode).

```

18242 \cs_new:Npn \__tl_change_case_N_type:NNNnnn #1#2#3
18243 {
18244   \quark_if_recursion_tail_stop_do:Nn #2
18245   { \__tl_change_case_N_type:Nnnn #1 }
18246   \token_if_eq_meaning:NNTF #1 #2
18247   {
18248     \use_i_delimit_by_q_recursion_stop:nw
18249     {
18250       \__tl_change_case_math:NNNnnn
18251       #1 #3 \__tl_change_case_loop:wnn
18252     }
18253   }
18254   { \__tl_change_case_N_type:NNNnnn #1 }
18255 }
18256 \cs_new:Npn \__tl_change_case_math:NNNnnn #1#2#3#4
18257 {
18258   \__tl_change_case_output:nwn {#1}
18259   \__tl_change_case_math_loop:wNNnn #4 \q_recursion_stop #2 #3
18260 }
18261 \cs_new:Npn \__tl_change_case_math_loop:wNNnn #1 \q_recursion_stop
18262 {
18263   \tl_if_head_is_N_type:nTF {#1}
18264   { \__tl_change_case_math:NwNNnn }
18265   {
18266     \tl_if_head_is_group:nTF {#1}
18267     { \__tl_change_case_math_group:nwNNnn }
18268     { \__tl_change_case_math_space:wNNnn }
18269   }
18270   #1 \q_recursion_stop
18271 }
18272 \cs_new:Npn \__tl_change_case_math:NwNNnn #1#2 \q_recursion_stop #3#4
18273 {
18274   \token_if_eq_meaning:NNTF \q_recursion_tail #1
18275   { \__tl_change_case_end:wn }
18276   {
18277     \__tl_change_case_output:nwn {#1}
18278     \token_if_eq_meaning:NNTF #1 #3
18279     { #4 #2 \q_recursion_stop }
18280     { \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop #3#4 }
18281   }
18282 }
18283 \cs_new:Npn \__tl_change_case_math_group:nwNNnn #1#2 \q_recursion_stop
18284 {
18285   \__tl_change_case_output:nwn { {#1} }
18286   \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop
18287 }
18288 \exp_last_unbraced:NNo
18289 \cs_new:Npn \__tl_change_case_math_space:wNNnn \c_space_tl
18290 {
18291   \__tl_change_case_output:nwn { ~ }
18292   \__tl_change_case_math_loop:wNNnn
18293 }

```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: they cannot be used in the lookup table and also may require expansion. At this stage the loop code starting `__tl_change_case_loop:wnn` is inserted: all subsequent steps in the code which need a look-ahead are coded to rely on this and thus have w-type arguments if they may do a look-ahead.

```

18294 \cs_new:Npn \__tl_change_case_N_type:Nnnn #1#2#3#4
18295 {
18296   \token_if_cs:NTF #1
18297     { \__tl_change_case_cs_letterlike:Nnn #1 {#3} { } }
18298     { \__tl_change_case_char:Nnn #1 {#3} {#4} }
18299   \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
18300 }

```

For character tokens there are some special cases to deal with then the majority of changes are covered by using the \TeX data as a lookup along with expandable character generation. This avoids needing a very large number of macros or (as seen in earlier versions) a somewhat tricky split of the characters into various blocks. Notice that the special case code may do a look-ahead so requires a final w-type argument whereas the core lookup table does not and also guarantees an output so f-type expansion may be used to obtain the case-changed result.

```

18301 \cs_new:Npn \__tl_change_case_char:Nnn #1#2#3
18302 {
18303   \cs_if_exist_use:cF { __tl_change_case_ #2 _ #3 :Nnw }
18304   { \use_ii:nn }
18305   #1
18306   {
18307     \use:c { __tl_change_case_ #2 _ sigma:Nnw } #1
18308     { \__tl_change_case_char:nN {#2} #1 }
18309   }
18310 }

```

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.)

```

18311 \cs_if_exist:NTF \utex_char:D
18312 {
18313   \cs_new:Npn \__tl_change_case_char:nN #1#2
18314     { \__tl_change_case_char_auxi:nN {#1} #2 }
18315 }
18316 {
18317   \cs_new:Npn \__tl_change_case_char:nN #1#2
18318   {
18319     \int_compare:nNnTF { '#2 } > { "80 }
18320     {
18321       \int_compare:nNnTF { '#2 } < { "E0 }
18322       { \__tl_change_case_char_UTFviii:nNNN {#1} #2 }
18323       {
18324         \int_compare:nNnTF { '#2 } < { "F0 }
18325         { \__tl_change_case_char_UTFviii:nNNNN {#1} #2 }
18326         { \__tl_change_case_char_UTFviii:nNNNNN {#1} #2 }

```

```

18327     }
18328   }
18329   { \_tl_change_case_char_auxi:nN {#1} #2 }
18330 }
18331 }
18332 \cs_new:Npn \_tl_change_case_char_auxi:nN #1#2
18333 {
18334   \_tl_change_case_output:fwn
18335   {
18336     \cs_if_exist_use:cF { c__unicode_ #1 _ \token_to_str:N #2 _tl }
18337     { \_tl_change_case_char_auxii:nN {#1} #2 }
18338   }
18339 }
18340 \cs_if_exist:NTF \utex_char:D
18341 {
18342   \cs_new:Npn \_tl_change_case_char_auxii:nN #1#2
18343   {
18344     \int_compare:nNnTF { \use:c { __tl_lookup_ #1 :N } #2 } = { 0 }
18345     { \exp_stop_f: #2 }
18346     {
18347       \char_generate:nn
18348       { \use:c { __tl_lookup_ #1 :N } #2 }
18349       { \char_value_catcode:n { \use:c { __tl_lookup_ #1 :N } #2 } }
18350     }
18351   }
18352   \cs_new_protected:Npn \_tl_lookup_lower:N #1 { \tex_lccode:D ‘#1 }
18353   \cs_new_protected:Npn \_tl_lookup_upper:N #1 { \tex_uccode:D ‘#1 }
18354   \cs_new_eq:NN \_tl_lookup_title:N \_tl_lookup_upper:N
18355 }
18356 {
18357   \cs_new:Npn \_tl_change_case_char_auxii:nN #1#2 { \exp_stop_f: #2 }
18358   \cs_new:Npn \_tl_change_case_char_UTFviii:nNNN #1#2#3#4
18359   { \_tl_change_case_char_UTFviii:nnN {#1} {#2#4} #3 }
18360   \cs_new:Npn \_tl_change_case_char_UTFviii:nNNNN #1#2#3#4#5
18361   { \_tl_change_case_char_UTFviii:nnN {#1} {#2#4#5} #3 }
18362   \cs_new:Npn \_tl_change_case_char_UTFviii:nNNNNN #1#2#3#4#5#6
18363   { \_tl_change_case_char_UTFviii:nnN {#1} {#2#4#5#6} #3 }
18364   \cs_new:Npn \_tl_change_case_char_UTFviii:nnN #1#2#3
18365   {
18366     \cs_if_exist:cTF { c__unicode_ #1 _ \tl_to_str:n {#2} _tl }
18367     {
18368       \_tl_change_case_output:vnw
18369       { c__unicode_ #1 _ \tl_to_str:n {#2} _tl }
18370     }
18371     { \_tl_change_case_output:nwn {#2} }
18372   } #3
18373 }
18374 }

```

Before dealing with general control sequences there are the special ones to deal with. Letter-like control sequences are a simple look-up, while for accents the loop is much as done elsewhere. Notice that we have a no-op test to make sure there is no unexpected expansion of letter-like input. The third argument here is needed for mixed casing, where it if there is a hit there has to be a change-of-path.

```

18375 \cs_new:Npn \__tl_change_case_cs_letterlike:Nnn #1#2#3
18376 {
18377   \cs_if_exist:cTF { c__tl_change_case_ #2 _ \token_to_str:N #1 _tl }
18378   {
18379     \__tl_change_case_output:vwN
18380     { c__tl_change_case_ #2 _ \token_to_str:N #1 _tl }
18381     #3
18382   }
18383   {
18384     \cs_if_exist:cTF
18385     {
18386       c__tl_change_case_
18387       \str_if_eq:nnTF {#2} { lower } { upper } { lower }
18388       _ \token_to_str:N #1 _tl
18389     }
18390     {
18391       \__tl_change_case_output:nwn {#1}
18392       #3
18393     }
18394     {
18395       \exp_after:wN \__tl_change_case_cs_accents:NN
18396       \exp_after:wN #1 \l_tl_case_change_accents_tl
18397       \q_recursion_tail \q_recursion_stop
18398     }
18399   }
18400 }
18401 \cs_new:Npn \__tl_change_case_cs_accents:NN #1#2
18402 {
18403   \quark_if_recursion_tail_stop_do:Nn #2
18404   { \__tl_change_case_cs:N #1 }
18405   \str_if_eq:nnTF {#1} {#2}
18406   {
18407     \use_i_delimit_by_q_recursion_stop:nw
18408     { \__tl_change_case_output:nwn {#1} }
18409   }
18410   { \__tl_change_case_cs_accents:NN #1 }
18411 }

```

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: that comes *after* the loop function which is therefore rearranged. In a $\text{\LaTeX} 2_\epsilon$ context, `\protect` needs to be treated specially, to prevent expansion of the next token but output it without braces.

```

18412 \cs_new:Npn \__tl_change_case_cs:N #1
18413 {
18414   <*package>
18415   \str_if_eq:nnTF {#1} { \protect } { \__tl_change_case_protect:wNN }
18416   </package>
18417   \exp_after:wN \__tl_change_case_cs:NN
18418   \exp_after:wN #1 \l_tl_case_change_exclude_tl
18419   \q_recursion_tail \q_recursion_stop
18420 }
18421 \cs_new:Npn \__tl_change_case_cs:NN #1#2
18422 {

```



```

18423 \quark_if_recursion_tail_stop_do:Nn #2
18424 {
18425   \__tl_change_case_cs_expand:Nnw #1
18426   { \__tl_change_case_output:nwn {#1} }
18427 }
18428 \str_if_eq:nnTF {#1} {#2}
18429 {
18430   \use_i_delimit_by_q_recursion_stop:nw
18431   { \__tl_change_case_cs:NNn #1 }
18432 }
18433 { \__tl_change_case_cs:NN #1 }
18434 }
18435 \cs_new:Npn \__tl_change_case_cs:NNn #1#2#3
18436 {
18437   \__tl_change_case_output:nwn { #1 {#3} }
18438   #2
18439 }
18440 \*package>
18441 \cs_new:Npn \__tl_change_case_protect:wNN #1 \q_recursion_stop #2 #3
18442 { \__tl_change_case_output:nwn { \protect #3 } #2 }
18443 \</package>

```

When a control sequence is not on the exclude list the other test if to see if it is expandable. Once again, if there is a hit then the loop function is grabbed as part of the clean-up and reinserted before the now expanded material. The test for expandability has to check for end-of-recursion as it is needed by the look-ahead code which might hit the end of the input. The test is done in two parts as `\bool_if:nTF` will choke if #1 is (!

```

18444 \cs_new:Npn \__tl_change_case_if_expandable:NTF #1
18445 {
18446   \token_if_expandable:NTF #1
18447   {
18448     \bool_if:nTF
18449     {
18450       \token_if_protected_macro_p:N      #1
18451       || \token_if_protected_long_macro_p:N #1
18452       || \token_if_eq_meaning_p:NN \q_recursion_tail #1
18453     }
18454     { \use_ii:nn }
18455     { \use_i:nn }
18456   }
18457   { \use_ii:nn }
18458 }
18459 \cs_new:Npn \__tl_change_case_cs_expand:Nnw #1#2
18460 {
18461   \__tl_change_case_if_expandable:NTF #1
18462   { \__tl_change_case_cs_expand:NN #1 }
18463   { #2 }
18464 }
18465 \cs_new:Npn \__tl_change_case_cs_expand:NN #1#2
18466 { \exp_after:wN #2 #1 }

```

(End definition for `__tl_change_case:nnn` and others.)

```

\__tl_change_case_lower_sigma:Nnw
\__tl_change_case_lower_sigma:w
\__tl_change_case_lower_sigma:Nw
\__tl_change_case_upper_sigma:Nnw

```

If the current char is an upper case 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.

```

18467 \cs_new:Npn \__tl_change_case_lower_sigma:Nnw #1#2#3#4 \q_recursion_stop
18468 {
18469   \int_compare:nNnTF { '#1 } = { "03A3 }
18470   {
18471     \__tl_change_case_output:fwN
18472     { \__tl_change_case_lower_sigma:w #4 \q_recursion_stop }
18473   }
18474   {#2}
18475   #3 #4 \q_recursion_stop
18476 }
18477 \cs_new:Npn \__tl_change_case_lower_sigma:w #1 \q_recursion_stop
18478 {
18479   \tl_if_head_is_N_type:nTF {#1}
18480   { \__tl_change_case_lower_sigma:Nw #1 \q_recursion_stop }
18481   { \c__unicode_final_sigma_tl }
18482 }
18483 \cs_new:Npn \__tl_change_case_lower_sigma:Nw #1#2 \q_recursion_stop
18484 {
18485   \__tl_change_case_if_expandable:NTF #1
18486   {
18487     \exp_after:wN \__tl_change_case_lower_sigma:w #1
18488     #2 \q_recursion_stop
18489   }
18490   {
18491     \token_if_letter:NTF #1
18492     { \c__unicode_std_sigma_tl }
18493     { \c__unicode_final_sigma_tl }
18494   }
18495 }

```

Simply skip to the final step for upper casing.

```

18496 \cs_new_eq:NN \__tl_change_case_upper_sigma:Nnw \use_ii:nn

```

(End definition for __tl_change_case_lower_sigma:Nnw and others.)

```

\__tl_change_case_lower_tr:Nnw
\__tl_change_case_lower_tr_auxi:Nw
\__tl_change_case_lower_tr_auxii:Nw
\__tl_change_case_upper_tr:Nnw
\__tl_change_case_lower_az:Nnw
\__tl_change_case_upper_az:Nnw

```

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.

```

18497 \cs_if_exist:NTF \utex_char:D
18498 {
18499   \cs_new:Npn \__tl_change_case_lower_tr:Nnw #1#2
18500   {
18501     \int_compare:nNnTF { '#1 } = { "0049 }
18502     { \__tl_change_case_lower_tr_auxi:Nw }
18503     {
18504       \int_compare:nNnTF { '#1 } = { "0130 }
18505       { \__tl_change_case_output:nwn { i } }
18506       {#2}
18507     }
18508   }

```

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, which is done by the `\use_i:nn` (it grabs `__tl_change_case_loop:wN` and the dot-above char and discards the latter).

```

18509 \cs_new:Npn \__tl_change_case_lower_tr_auxi:Nw #1#2 \q_recursion_stop
18510 {
18511   \tl_if_head_is_N_type:nTF {#2}
18512   { \__tl_change_case_lower_tr_auxii:Nw #2 \q_recursion_stop }
18513   { \__tl_change_case_output:Vwn \c__unicode_dotless_i_tl }
18514   #1 #2 \q_recursion_stop
18515 }
18516 \cs_new:Npn \__tl_change_case_lower_tr_auxii:Nw #1#2 \q_recursion_stop
18517 {
18518   \__tl_change_case_if_expandable:NTF #1
18519   {
18520     \exp_after:wN \__tl_change_case_lower_tr_auxi:Nw #1
18521     #2 \q_recursion_stop
18522   }
18523   {
18524     \bool_if:nTF
18525     {
18526       \token_if_cs_p:N #1
18527       || ! ( \int_compare_p:nNn { '#1 } = { "0307 } )
18528     }
18529     { \__tl_change_case_output:Vwn \c__unicode_dotless_i_tl }
18530     {
18531       \__tl_change_case_output:nwn { i }
18532       \use_i:nn
18533     }
18534   }
18535 }
18536 }

```

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.

```

18537 {
18538   \cs_new:Npn \__tl_change_case_lower_tr:Nnw #1#2
18539   {
18540     \int_compare:nNnTF { '#1 } = { "0049 }
18541     { \__tl_change_case_output:Vwn \c__unicode_dotless_i_tl }
18542     {
18543       \int_compare:nNnTF { '#1 } = { 196 }
18544       { \__tl_change_case_lower_tr_auxi:Nw #1 {#2} }
18545       {#2}
18546     }
18547   }
18548   \cs_new:Npn \__tl_change_case_lower_tr_auxi:Nw #1#2#3#4
18549   {
18550     \int_compare:nNnTF { '#4 } = { 176 }
18551     {
18552       \__tl_change_case_output:nwn { i }
18553       #3
18554     }
18555     {

```

```

18556         #2
18557         #3 #4
18558     }
18559 }
18560 }

```

Upper casing is easier: just one exception with no context.

```

18561 \cs_new:Npn \__tl_change_case_upper_tr:Nnw #1#2
18562 {
18563     \int_compare:nNnTF { '#1 } = { "0069 }
18564     { \__tl_change_case_output:Vwn \c__unicode_dotted_I_tl }
18565     {#2}
18566 }

```

Straight copies.

```

18567 \cs_new_eq:NN \__tl_change_case_lower_az:Nnw \__tl_change_case_lower_tr:Nnw
18568 \cs_new_eq:NN \__tl_change_case_upper_az:Nnw \__tl_change_case_upper_tr:Nnw

```

(End definition for __tl_change_case_lower_tr:Nnw and others.)

_tl_change_case_lower_lt:Nnw For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. That means that there is some work to do when lower casing I and J. The first step is a simple match attempt: \c__tl_accents_lt_tl contains accented upper case letters which should gain a dot-above char in their lower case form. This is done using f-type expansion so only one pass is needed to find if it works or not. If there was no hit, the second stage is to check for I, J and I-ogonek, and if the current char is a match to look for a following accent.

```

18569 \cs_new:Npn \__tl_change_case_lower_lt:Nnw #1
18570 {
18571     \exp_args:Nf \__tl_change_case_lower_lt:nNnw
18572     { \str_case:nVF #1 \c__unicode_accents_lt_tl \exp_stop_f: }
18573     #1
18574 }
18575 \cs_new:Npn \__tl_change_case_lower_lt:nNnw #1#2
18576 {
18577     \tl_if_blank:nTF {#1}
18578     {
18579         \exp_args:Nf \__tl_change_case_lower_lt:nnw
18580         {
18581             \int_case:nnF { '#2 }
18582             {
18583                 { "0049 } i
18584                 { "004A } j
18585                 { "012E } \c__unicode_i_ogonek_tl
18586             }
18587             \exp_stop_f:
18588         }
18589     }
18590     {
18591         \__tl_change_case_output:wnw {#1}
18592         \use_none:n
18593     }
18594 }
18595 \cs_new:Npn \__tl_change_case_lower_lt:nnw #1#2
18596 {

```

```

18597 \tl_if_blank:nTF {#1}
18598   {#2}
18599   {
18600     \__tl_change_case_output:nwn {#1}
18601     \__tl_change_case_lower_lt:Nw
18602   }
18603 }

```

Grab the next char and see if it is one of the accents used in Lithuanian: if it is, add the dot-above char into the output.

```

18604 \cs_new:Npn \__tl_change_case_lower_lt:Nw #1#2 \q_recursion_stop
18605 {
18606   \tl_if_head_is_N_type:nT {#2}
18607   { \__tl_change_case_lower_lt:NNw }
18608   #1 #2 \q_recursion_stop
18609 }
18610 \cs_new:Npn \__tl_change_case_lower_lt:NNw #1#2#3 \q_recursion_stop
18611 {
18612   \__tl_change_case_if_expandable:NTF #2
18613   {
18614     \exp_after:wN \__tl_change_case_lower_lt:Nw \exp_after:wN #1 #2
18615     #3 \q_recursion_stop
18616   }
18617   {
18618     \bool_if:nT
18619     {
18620       ! \token_if_cs_p:N #2
18621       &&
18622       (
18623         \int_compare_p:nNn { '#2 } = { "0300 }
18624         || \int_compare_p:nNn { '#2 } = { "0301 }
18625         || \int_compare_p:nNn { '#2 } = { "0303 }
18626       )
18627     }
18628     { \__tl_change_case_output:Vwn \c__unicode_dot_above_tl }
18629     #1 #2#3 \q_recursion_stop
18630   }
18631 }

```

For upper casing, the test required is for a dot-above char after an I, J or I-ogonek. First a test for the appropriate letter, and if found a look-ahead and potentially one token dropped.

```

18632 \cs_new:Npn \__tl_change_case_upper_lt:Nnw #1
18633 {
18634   \exp_args:Nf \__tl_change_case_upper_lt:nnw
18635   {
18636     \int_case:nnF { '#1 }
18637     {
18638       { "0069 } I
18639       { "006A } J
18640       { "012F } \c__unicode_I_ogonek_tl
18641     }
18642     \exp_stop_f:
18643   }
18644 }

```

```

18645 \cs_new:Npn \__tl_change_case_upper_lt:nw #1#2
18646 {
18647   \tl_if_blank:nTF {#1}
18648     {#2}
18649     {
18650       \__tl_change_case_output:nwn {#1}
18651       \__tl_change_case_upper_lt:Nw
18652     }
18653   }
18654 \cs_new:Npn \__tl_change_case_upper_lt:Nw #1#2 \q_recursion_stop
18655 {
18656   \tl_if_head_is_N_type:nT {#2}
18657     { \__tl_change_case_upper_lt:NNw }
18658     #1 #2 \q_recursion_stop
18659   }
18660 \cs_new:Npn \__tl_change_case_upper_lt:NNw #1#2#3 \q_recursion_stop
18661 {
18662   \__tl_change_case_if_expandable:NTF #2
18663   {
18664     \exp_after:wN \__tl_change_case_upper_lt:Nw \exp_after:wN #1 #2
18665     #3 \q_recursion_stop
18666   }
18667   {
18668     \bool_if:nTF
18669       {
18670         ! \token_if_cs_p:N #2
18671         && \int_compare_p:nNn { '#2 } = { "0307 }
18672       }
18673       { #1 }
18674       { #1 #2 }
18675     #3 \q_recursion_stop
18676   }
18677 }

```

(End definition for __tl_change_case_lower_lt:Nnw and others.)

_tl_change_case_upper_de-alt:Nnw A simple alternative version for German.

```

18678 \cs_new:cpn { \__tl_change_case_upper_de-alt:Nnw } #1#2
18679 {
18680   \int_compare:nNnTF { '#1 } = { 223 }
18681     { \__tl_change_case_output:Vwn \c__unicode_upper_Eszett_tl }
18682     {#2}
18683 }

```

(End definition for __tl_change_case_upper_de-alt:Nnw.)

_unicode_codepoint_to_UTFviii:n This code will convert a codepoint into the correct UTF-8 representation. As there are a variable number of octets, the result starts with the numeral 1–4 to indicate the nature of the returned value. Note that this code will cover the full range even though at this stage it is not required here. Also note that longer-term this is likely to need a public interface and/or moving to l3str (see experimental string conversions). In terms of the algorithm itself, see <https://en.wikipedia.org/wiki/UTF-8> for the octet pattern.

```

18684 \cs_new:Npn \__unicode_codepoint_to_UTFviii:n #1
18685 {

```

```

18686     \exp_args:Nf \__unicode_codepoint_to_UTFviii_auxi:n
18687     { \int_eval:n {#1} }
18688   }
18689   \cs_new:Npn \__unicode_codepoint_to_UTFviii_auxi:n #1
18690   {
18691     \if_int_compare:w #1 > "80 ~
18692     \if_int_compare:w #1 < "800 ~
18693       2
18694       \__unicode_codepoint_to_UTFviii_auxii:Nnn C {#1} { 64 }
18695       \__unicode_codepoint_to_UTFviii_auxiii:n {#1}
18696     \else:
18697       \if_int_compare:w #1 < "10000 ~
18698         3
18699         \__unicode_codepoint_to_UTFviii_auxii:Nnn E {#1} { 64 * 64 }
18700         \__unicode_codepoint_to_UTFviii_auxiii:n {#1}
18701         \__unicode_codepoint_to_UTFviii_auxiii:n
18702         { \int_div_truncate:nn {#1} { 64 } }
18703       \else:
18704         4
18705         \__unicode_codepoint_to_UTFviii_auxii:Nnn F
18706         {#1} { 64 * 64 * 64 }
18707         \__unicode_codepoint_to_UTFviii_auxiii:n
18708         { \int_div_truncate:nn {#1} { 64 * 64 } }
18709         \__unicode_codepoint_to_UTFviii_auxiii:n
18710         { \int_div_truncate:nn {#1} { 64 } }
18711         \__unicode_codepoint_to_UTFviii_auxiii:n {#1}
18712
18713       \fi:
18714     \fi:
18715   \else:
18716     1 {#1}
18717   \fi:
18718 }
18719 \cs_new:Npn \__unicode_codepoint_to_UTFviii_auxii:Nnn #1#2#3
18720 { { \int_eval:n { "#10 + \int_div_truncate:nn {#2} {#3} } } }
18721 \cs_new:Npn \__unicode_codepoint_to_UTFviii_auxiii:n #1
18722 { { \int_eval:n { \int_mod:nn {#1} { 64 } + 128 } } }

```

(End definition for __unicode_codepoint_to_UTFviii:n and others.)

```

\c__unicode_std_sigma_tl
\c__unicode_final_sigma_tl
\c__unicode_accents_lt_tl
\c__unicode_dot_above_tl
\c__unicode_upper_Eszett_tl

```

The above needs various special token lists containing pre-formed characters. This set are only available in Unicode engines, with no-op definitions for 8-bit use.

```

18723 \cs_if_exist:NTF \utex_char:D
18724 {
18725   \tl_const:Nx \c__unicode_std_sigma_tl { \utex_char:D "03C3 ~ }
18726   \tl_const:Nx \c__unicode_final_sigma_tl { \utex_char:D "03C2 ~ }
18727   \tl_const:Nx \c__unicode_accents_lt_tl
18728   {
18729     \utex_char:D "00CC ~
18730     { \utex_char:D "0069 ~ \utex_char:D "0307 ~ \utex_char:D "0300 ~ }
18731     \utex_char:D "00CD ~
18732     { \utex_char:D "0069 ~ \utex_char:D "0307 ~ \utex_char:D "0301 ~ }
18733     \utex_char:D "0128 ~
18734     { \utex_char:D "0069 ~ \utex_char:D "0307 ~ \utex_char:D "0303 ~ }

```

```

18735     }
18736     \tl_const:Nx \c__unicode_dot_above_tl { \utex_char:D "0307 ~ }
18737     \tl_const:Nx \c__unicode_upper_Eszett_tl { \utex_char:D "1E9E ~ }
18738   }
18739   {
18740     \tl_const:Nn \c__unicode_std_sigma_tl { }
18741     \tl_const:Nn \c__unicode_final_sigma_tl { }
18742     \tl_const:Nn \c__unicode_accents_lt_tl { }
18743     \tl_const:Nn \c__unicode_dot_above_tl { }
18744     \tl_const:Nn \c__unicode_upper_Eszett_tl { }
18745   }

```

(End definition for \c__unicode_std_sigma_tl and others.)

\c__unicode_dotless_i_tl For cases where there is an 8-bit option in the T1 font set up, a variant is provided in both cases.

```

\c__unicode_dotted_I_tl
\c__unicode_i_ogonek_tl
\c__unicode_I_ogonek_tl
18746 \group_begin:
18747   \cs_if_exist:NTF \utex_char:D
18748   {
18749     \cs_set_protected:Npn \__tl_tmp:w #1#2
18750     { \tl_const:Nx #1 { \utex_char:D "#2 ~ } }
18751   }
18752   {
18753     \cs_set_protected:Npn \__tl_tmp:w #1#2
18754     {
18755       \group_begin:
18756       \cs_set_protected:Npn \__tl_tmp:w ##1##2##3
18757       {
18758         \tl_const:Nx #1
18759         {
18760           \exp_after:wN \exp_after:wN \exp_after:wN
18761           \exp_not:N \__char_generate:nn {##2} { 13 }
18762           \exp_after:wN \exp_after:wN \exp_after:wN
18763           \exp_not:N \__char_generate:nn {##3} { 13 }
18764         }
18765       }
18766       \tl_set:Nx \l__tl_internal_a_tl
18767       { \__unicode_codepoint_to_UTFviii:n {"#2} }
18768       \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
18769     \group_end:
18770   }
18771 }
18772 \__tl_tmp:w \c__unicode_dotless_i_tl { 0131 }
18773 \__tl_tmp:w \c__unicode_dotted_I_tl { 0130 }
18774 \__tl_tmp:w \c__unicode_i_ogonek_tl { 012F }
18775 \__tl_tmp:w \c__unicode_I_ogonek_tl { 012E }
18776 \group_end:

```

(End definition for \c__unicode_dotless_i_tl and others.)

For 8-bit engines we now need to define the case-change data for the multi-octet mappings. 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.

```

18777 \group_begin:

```



```

18778 \bool_if:nT
18779 {
18780   \sys_if_engine_pdftex_p: || \sys_if_engine_uptex_p:
18781 }
18782 {
18783   \cs_set_protected:Npn \__tl_loop:nn #1#2
18784   {
18785     \quark_if_recursion_tail_stop:n {#1}
18786     \tl_set:Nx \l__tl_internal_a_tl
18787     {
18788       \__unicode_codepoint_to_UTFviii:n {"#1}
18789       \__unicode_codepoint_to_UTFviii:n {"#2}
18790     }
18791     \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
18792     \__tl_loop:nn
18793   }
18794   \cs_set_protected:Npn \__tl_tmp:w #1#2#3#4#5#6
18795   {
18796     \tl_const:cx
18797     {
18798       c__unicode_lower_
18799       \char_generate:nn {#2} { 12 }
18800       \char_generate:nn {#3} { 12 }
18801       _tl
18802     }
18803     {
18804       \exp_after:wN \exp_after:wN \exp_after:wN
18805       \exp_not:N \__char_generate:nn {#5} { 13 }
18806       \exp_after:wN \exp_after:wN \exp_after:wN
18807       \exp_not:N \__char_generate:nn {#6} { 13 }
18808     }
18809     \tl_const:cx
18810     {
18811       c__unicode_upper_
18812       \char_generate:nn {#5} { 12 }
18813       \char_generate:nn {#6} { 12 }
18814       _tl
18815     }
18816     {
18817       \exp_after:wN \exp_after:wN \exp_after:wN
18818       \exp_not:N \__char_generate:nn {#2} { 13 }
18819       \exp_after:wN \exp_after:wN \exp_after:wN
18820       \exp_not:N \__char_generate:nn {#3} { 13 }
18821     }
18822   }
18823   \__tl_loop:nn
18824   { 00C0 } { 00E0 }
18825   { 00C2 } { 00E2 }
18826   { 00C3 } { 00E3 }
18827   { 00C4 } { 00E4 }
18828   { 00C5 } { 00E5 }
18829   { 00C6 } { 00E6 }
18830   { 00C7 } { 00E7 }
18831   { 00C8 } { 00E8 }

```

18832	{ 00C9 }	{ 00E9 }
18833	{ 00CA }	{ 00EA }
18834	{ 00CB }	{ 00EB }
18835	{ 00CC }	{ 00EC }
18836	{ 00CD }	{ 00ED }
18837	{ 00CE }	{ 00EE }
18838	{ 00CF }	{ 00EF }
18839	{ 00D0 }	{ 00F0 }
18840	{ 00D1 }	{ 00F1 }
18841	{ 00D2 }	{ 00F2 }
18842	{ 00D3 }	{ 00F3 }
18843	{ 00D4 }	{ 00F4 }
18844	{ 00D5 }	{ 00F5 }
18845	{ 00D6 }	{ 00F6 }
18846	{ 00D8 }	{ 00F8 }
18847	{ 00D9 }	{ 00F9 }
18848	{ 00DA }	{ 00FA }
18849	{ 00DB }	{ 00FB }
18850	{ 00DC }	{ 00FC }
18851	{ 00DD }	{ 00FD }
18852	{ 00DE }	{ 00FE }
18853	{ 0100 }	{ 0101 }
18854	{ 0102 }	{ 0103 }
18855	{ 0104 }	{ 0105 }
18856	{ 0106 }	{ 0107 }
18857	{ 0108 }	{ 0109 }
18858	{ 010A }	{ 010B }
18859	{ 010C }	{ 010D }
18860	{ 010E }	{ 010F }
18861	{ 0110 }	{ 0111 }
18862	{ 0112 }	{ 0113 }
18863	{ 0114 }	{ 0115 }
18864	{ 0116 }	{ 0117 }
18865	{ 0118 }	{ 0119 }
18866	{ 011A }	{ 011B }
18867	{ 011C }	{ 011D }
18868	{ 011E }	{ 011F }
18869	{ 0120 }	{ 0121 }
18870	{ 0122 }	{ 0123 }
18871	{ 0124 }	{ 0125 }
18872	{ 0128 }	{ 0129 }
18873	{ 012A }	{ 012B }
18874	{ 012C }	{ 012D }
18875	{ 012E }	{ 012F }
18876	{ 0132 }	{ 0133 }
18877	{ 0134 }	{ 0135 }
18878	{ 0136 }	{ 0137 }
18879	{ 0139 }	{ 013A }
18880	{ 013B }	{ 013C }
18881	{ 013E }	{ 013F }
18882	{ 0141 }	{ 0142 }
18883	{ 0143 }	{ 0144 }
18884	{ 0145 }	{ 0146 }
18885	{ 0147 }	{ 0148 }

```

18886 { 014A } { 014B }
18887 { 014C } { 014D }
18888 { 014E } { 014F }
18889 { 0150 } { 0151 }
18890 { 0152 } { 0153 }
18891 { 0154 } { 0155 }
18892 { 0156 } { 0157 }
18893 { 0158 } { 0159 }
18894 { 015A } { 015B }
18895 { 015C } { 015D }
18896 { 015E } { 015F }
18897 { 0160 } { 0161 }
18898 { 0162 } { 0163 }
18899 { 0164 } { 0165 }
18900 { 0168 } { 0169 }
18901 { 016A } { 016B }
18902 { 016C } { 016D }
18903 { 016E } { 016F }
18904 { 0170 } { 0171 }
18905 { 0172 } { 0173 }
18906 { 0174 } { 0175 }
18907 { 0176 } { 0177 }
18908 { 0178 } { 00FF }
18909 { 0179 } { 017A }
18910 { 017B } { 017C }
18911 { 017D } { 017E }
18912 { 01CD } { 01CE }
18913 { 01CF } { 01D0 }
18914 { 01D1 } { 01D2 }
18915 { 01D3 } { 01D4 }
18916 { 01E2 } { 01E3 }
18917 { 01E6 } { 01E7 }
18918 { 01E8 } { 01E9 }
18919 { 01EA } { 01EB }
18920 { 01F4 } { 01F5 }
18921 { 0218 } { 0219 }
18922 { 021A } { 021B }
18923 \q_recursion_tail ?
18924 \q_recursion_stop
18925 \cs_set_protected:Npn \__tl_tmp:w #1#2#3
18926 {
18927   \group_begin:
18928     \cs_set_protected:Npn \__tl_tmp:w ##1##2##3
18929     {
18930       \tl_const:cx
18931       {
18932         c__unicode_ #3 _
18933         \char_generate:nn {##2} { 12 }
18934         \char_generate:nn {##3} { 12 }
18935         _tl
18936       }
18937       {#2}
18938     }
18939     \tl_set:Nx \l__tl_internal_a_tl

```

```

18940         { \__unicode_codepoint_to_UTFviii:n { "#1 } }
18941         \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
18942         \group_end:
18943     }
18944     \__tl_tmp:w { 00DF } { SS } { upper }
18945     \__tl_tmp:w { 00DF } { Ss } { title }
18946     \__tl_tmp:w { 0131 } { I } { upper }
18947 }
18948 \group_end:

```

The (fixed) look-up mappings for letter-like control sequences.

```

18949 \group_begin:
18950 \cs_set_protected:Npn \__tl_change_case_setup:NN #1#2
18951 {
18952     \quark_if_recursion_tail_stop:N #1
18953     \tl_const:cn { c__tl_change_case_lower_ \token_to_str:N #1 _tl } { #2 }
18954     \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N #2 _tl } { #1 }
18955     \__tl_change_case_setup:NN
18956 }
18957 \__tl_change_case_setup:NN
18958 \AA \aa
18959 \AE \ae
18960 \DH \dh
18961 \DJ \dj
18962 \IJ \ij
18963 \L \l
18964 \NG \ng
18965 \O \o
18966 \OE \oe
18967 \SS \ss
18968 \TH \th
18969 \q_recursion_tail ?
18970 \q_recursion_stop
18971 \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N \i _tl } { I }
18972 \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N \j _tl } { J }
18973 \group_end:

```

\l_tl_case_change_accents_tl A list of accents to leave alone.

```

18974 \tl_new:N \l_tl_case_change_accents_tl
18975 \tl_set:Nn \l_tl_case_change_accents_tl
18976 { \" \' \. \^ \' \~ \c \H \k \r \t \u \v }

```

(End definition for `\l_tl_case_change_accents_tl`. This variable is documented on page 214.)

```

\__tl_mixed_case:nn
\__tl_mixed_case_aux:nn
\__tl_mixed_case_loop:wN
\__tl_mixed_case_group:nwN
\__tl_mixed_case_space:wN
\__tl_mixed_case_N_type:NwN
\__tl_mixed_case_N_type:NNwNn
\__tl_mixed_case_N_type:Nnn
\__tl_mixed_case_letterlike:Nw
\__tl_mixed_case_char:N
\__tl_mixed_case_skip:N
\__tl_mixed_case_skip:NN
\__tl_mixed_case_skip_tidy:NwN
\__tl_mixed_case_char:nN

```

Mixed (title) casing requires some custom handling of the case changing of the first letter in the input followed by a switch to the normal lower casing routine. That could be covered by passing a set of functions to generic routines, but at the cost of making the process rather opaque. Instead, the approach taken here is to use a dedicated set of functions which keep the different loop requirements clearly separate.

The main loop looks for the first “real” char in the input (skipping any pre-letter chars). Once one is found, it is case changed to upper case but first checking that there is not an entry in the exceptions list. Note that simply grabbing the first token in the input is no good here: it can’t handle pre-letter tokens or any special treatment of the first letter found (*e.g.* words starting with *i* in Turkish). Spaces at the start of the input

are passed through without counting as being the “start” of the first word, while a brace group is assumed to be contain the first char with everything after the brace therefore lower cased.

```

18977 \cs_new:Npn \__tl_mixed_case:nn #1#2
18978 {
18979   \etex_unexpanded:D \exp_after:wN
18980   {
18981     \exp:w
18982     \__tl_mixed_case_aux:nn {#1} {#2}
18983   }
18984 }
18985 \cs_new:Npn \__tl_mixed_case_aux:nn #1#2
18986 {
18987   \group_align_safe_begin:
18988   \__tl_mixed_case_loop:wn
18989   #2 \q_recursion_tail \q_recursion_stop {#1}
18990   \__tl_change_case_result:n { }
18991 }
18992 \cs_new:Npn \__tl_mixed_case_loop:wn #1 \q_recursion_stop
18993 {
18994   \tl_if_head_is_N_type:nTF {#1}
18995   { \__tl_mixed_case_N_type:Nwn }
18996   {
18997     \tl_if_head_is_group:nTF {#1}
18998     { \__tl_mixed_case_group:nwn }
18999     { \__tl_mixed_case_space:wn }
19000   }
19001   #1 \q_recursion_stop
19002 }
19003 \cs_new:Npn \__tl_mixed_case_group:nwn #1#2 \q_recursion_stop #3
19004 {
19005   \__tl_change_case_output:own
19006   {
19007     \exp_after:wN
19008     {
19009       \exp:w
19010       \__tl_mixed_case_aux:nn {#3} {#1}
19011     }
19012   }
19013   \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } {#3}
19014 }
19015 \exp_last_unbraced:NNo \cs_new:Npn \__tl_mixed_case_space:wn \c_space_tl
19016 {
19017   \__tl_change_case_output:nwn { ~ }
19018   \__tl_mixed_case_loop:wn
19019 }
19020 \cs_new:Npn \__tl_mixed_case_N_type:Nwn #1#2 \q_recursion_stop
19021 {
19022   \quark_if_recursion_tail_stop_do:Nn #1
19023   { \__tl_change_case_end:wn }
19024   \exp_after:wN \__tl_mixed_case_N_type:NNNnn
19025   \exp_after:wN #1 \l_tl_case_change_math_tl
19026   \q_recursion_tail ? \q_recursion_stop {#2}
19027 }

```

```

19028 \cs_new:Npn \__tl_mixed_case_N_type:NNNnn #1#2#3
19029 {
19030   \quark_if_recursion_tail_stop_do:Nn #2
19031   { \__tl_mixed_case_N_type:Nnn #1 }
19032   \token_if_eq_meaning:NNTF #1 #2
19033   {
19034     \use_i_delimit_by_q_recursion_stop:nw
19035     {
19036       \__tl_change_case_math:NNNnnn
19037       #1 #3 \__tl_mixed_case_loop:wn
19038     }
19039   }
19040   { \__tl_mixed_case_N_type:NNNnn #1 }
19041 }

```

The business end of the loop is here: there is first a need to deal with any control sequence cases before looking for characters to skip. If there is a hit for a letter-like control sequence, switch to lower casing.

```

19042 \cs_new:Npn \__tl_mixed_case_N_type:Nnn #1#2#3
19043 {
19044   \token_if_cs:NNTF #1
19045   {
19046     \__tl_change_case_cs_letterlike:Nnn #1 { upper }
19047     { \__tl_mixed_case_letterlike:Nw }
19048     \__tl_mixed_case_loop:wn #2 \q_recursion_stop {#3}
19049   }
19050   {
19051     \__tl_mixed_case_char:Nn #1 {#3}
19052     \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } {#3}
19053   }
19054 }
19055 \cs_new:Npn \__tl_mixed_case_letterlike:Nw #1#2 \q_recursion_stop
19056 { \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } }

```

As detailed above, handling a mixed case char means first looking for exceptions then treating as an upper cased letter, but with a list of tokens to skip over too.

```

19057 \cs_new:Npn \__tl_mixed_case_char:Nn #1#2
19058 {
19059   \cs_if_exist_use:cF { __tl_change_case_mixed_ #2 :Nnw }
19060   {
19061     \cs_if_exist_use:cF { __tl_change_case_upper_ #2 :Nnw }
19062     { \use_ii:nn }
19063   }
19064   #1
19065   { \__tl_mixed_case_skip:N #1 }
19066 }
19067 \cs_new:Npn \__tl_mixed_case_skip:N #1
19068 {
19069   \exp_after:wN \__tl_mixed_case_skip:NN
19070   \exp_after:wN #1 \l_tl_mixed_case_ignore_tl
19071   \q_recursion_tail \q_recursion_stop
19072 }
19073 \cs_new:Npn \__tl_mixed_case_skip:NN #1#2
19074 {
19075   \quark_if_recursion_tail_stop_do:nn {#2}

```

```

19076     { \_tl_mixed_case_char:N #1 }
19077 \int_compare:nNnT { '#1 } = { '#2 }
19078     {
19079         \use_i_delimit_by_q_recursion_stop:nw
19080         {
19081             \_tl_change_case_output:nwn {#1}
19082             \_tl_mixed_case_skip_tidy:Nwn
19083         }
19084     }
19085     \_tl_mixed_case_skip:NN #1
19086 }
19087 \cs_new:Npn \_tl_mixed_case_skip_tidy:Nwn #1#2 \q_recursion_stop #3
19088 {
19089     \_tl_mixed_case_loop:wn #2 \q_recursion_stop
19090 }
19091 \cs_new:Npn \_tl_mixed_case_char:N #1
19092 {
19093     \cs_if_exist:cTF { c__unicode_title_ #1 _tl }
19094     {
19095         \_tl_change_case_output:fwn
19096         { \tl_use:c { c__unicode_title_ #1 _tl } }
19097     }
19098     { \_tl_change_case_char:nN { upper } #1 }
19099 }

```

(End definition for _tl_mixed_case:nn and others.)

_tl_change_case_mixed_n1:Nnw 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.

```

19100 \cs_new:Npn \_tl_change_case_mixed_n1:Nnw #1
19101 {
19102     \bool_if:nTF
19103     {
19104         \int_compare_p:nNn { '#1 } = { 'i }
19105         || \int_compare_p:nNn { '#1 } = { 'I }
19106     }
19107     {
19108         \_tl_change_case_output:nwn { I }
19109         \_tl_change_case_mixed_n1:Nw
19110     }
19111 }
19112 \cs_new:Npn \_tl_change_case_mixed_n1:Nw #1#2 \q_recursion_stop
19113 {
19114     \tl_if_head_is_N_type:nT {#2}
19115     { \_tl_change_case_mixed_n1:NNw }
19116     #1 #2 \q_recursion_stop
19117 }
19118 \cs_new:Npn \_tl_change_case_mixed_n1:NNw #1#2#3 \q_recursion_stop
19119 {
19120     \_tl_change_case_if_expandable:NTF #2
19121     {
19122         \exp_after:wN \_tl_change_case_mixed_n1:Nw \exp_after:wN #1 #2
19123         #3 \q_recursion_stop
19124     }

```

```

19125 {
19126   \bool_if:nTF
19127   {
19128     ! ( \token_if_cs_p:N #2 )
19129     &&
19130     (
19131       \int_compare_p:nNn { '#2 } = { 'j }
19132       || \int_compare_p:nNn { '#2 } = { 'J }
19133     )
19134   }
19135   {
19136     \__tl_change_case_output:nwn { J }
19137     #1
19138   }
19139   { #1 #2 }
19140   #3 \q_recursion_stop
19141 }
19142 }

```

(End definition for `__tl_change_case_mixed_n1:Nnw`, `__tl_change_case_mixed_n1:Nw`, and `__tl_change_case_mixed_n1:NNw`.)

`\l_tl_case_change_math_tl` The list of token pairs which are treated as math mode and so not case changed.

```

19143 \tl_new:N \l_tl_case_change_math_tl
19144 <*package>
19145 \tl_set:Nn \l_tl_case_change_math_tl
19146 { $ $ \ ( \ ) }
19147 </package>

```

(End definition for `\l_tl_case_change_math_tl`. This variable is documented on page 213.)

`\l_tl_case_change_exclude_tl` The list of commands for which an argument is not case changed.

```

19148 \tl_new:N \l_tl_case_change_exclude_tl
19149 <*package>
19150 \tl_set:Nn \l_tl_case_change_exclude_tl
19151 { \cite \ensuremath \label \ref }
19152 </package>

```

(End definition for `\l_tl_case_change_exclude_tl`. This variable is documented on page 213.)

`\l_tl_mixed_case_ignore_tl` Characters to skip over when finding the first letter in a word to be mixed cased.

```

19153 \tl_new:N \l_tl_mixed_case_ignore_tl
19154 \tl_set:Nx \l_tl_mixed_case_ignore_tl
19155 {
19156   ( % )
19157   [ % ]
19158   \cs_to_str:N \{ % \% }
19159   '
19160   -
19161 }

```

(End definition for `\l_tl_mixed_case_ignore_tl`. This variable is documented on page 214.)

\tl_log:N Redirect output of \tl_show:N to the log.

```
\tl_log:c 19162 \cs_new_protected:Npn \tl_log:N
19163 { \__msg_log_next: \tl_show:N }
19164 \cs_generate_variant:Nn \tl_log:N { c }
```

(End definition for \tl_log:N. This function is documented on page 215.)

\tl_log:n Redirect output of \tl_show:n to the log.

```
19165 \cs_new_protected:Npn \tl_log:n
19166 { \__msg_log_next: \tl_show:n }
```

(End definition for \tl_log:n. This function is documented on page 215.)

\tl_rand_item:n Importantly \tl_item:nn only evaluates its argument once.

```
\tl_rand_item:N 19167 \cs_new:Npn \tl_rand_item:n #1
\tl_rand_item:c 19168 {
19169     \tl_if_blank:nF {#1}
19170     { \tl_item:nn {#1} { \int_rand:nn { 1 } { \tl_count:n {#1} } } }
19171 }
19172 \cs_new:Npn \tl_rand_item:N { \exp_args:No \tl_rand_item:n }
19173 \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 216.)

35.20 Additions to l3tokens

```
19174 <@@=peek>
```

\peek_N_type:TF

__peek_execute_branches_N_type:

__peek_N_type:w

__peek_N_type_aux:nnw

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, \use_none_delimit_by_q_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.

```
19175 \group_begin:
19176 \cs_set_protected:Npn \__peek_tmp:w #1 \q_stop
19177 {
19178     \cs_new_protected:Npn \__peek_execute_branches_N_type:
19179     {
19180         \if_int_odd:w
19181             \if_catcode:w \exp_not:N \l_peek_token { \c_two \fi:
19182             \if_catcode:w \exp_not:N \l_peek_token } \c_two \fi:
19183             \if_meaning:w \l_peek_token \c_space_token \c_two \fi:
19184             \c_one
```

```

19185         \exp_after:wN \__peek_N_type:w
19186         \token_to_meaning:N \l_peek_token
19187         \q_mark \__peek_N_type_aux:nw
19188         #1 \q_mark \use_none_delimit_by_q_stop:w
19189         \q_stop
19190         \exp_after:wN \__peek_true:w
19191     \else:
19192         \exp_after:wN \__peek_false:w
19193     \fi:
19194 }
19195 \cs_new_protected:Npn \__peek_N_type:w ##1 #1 ##2 \q_mark ##3
19196 { ##3 {##1} {##2} }
19197 }
19198 \exp_after:wN \__peek_tmp:w \tl_to_str:n { outer } \q_stop
19199 \group_end:
19200 \cs_new_protected:Npn \__peek_N_type_aux:nw #1 #2 #3 \fi:
19201 {
19202     \fi:
19203     \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
19204     { \__peek_true:w }
19205     { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
19206 }
19207 \cs_new_protected:Npn \peek_N_type:TF
19208 { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }
19209 \cs_new_protected:Npn \peek_N_type:T
19210 { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
19211 \cs_new_protected:Npn \peek_N_type:F
19212 { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }

(End definition for \peek_N_type:TF and others. These functions are documented on page 216.)

19213 </initex | package>

```

36 l3sys implementation

```

19214 <*initex | package>

```

36.1 The name of the job

\c_sys_jobname_str Inherited from the L^AT_EX3 name for the primitive: this needs to actually contain the text of the job name rather than the name of the primitive, of course.

```

19215 <*initex>
19216 \tex_everyjob:D \exp_after:wN
19217 {
19218     \tex_the:D \tex_everyjob:D
19219     \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
19220 }
19221 </initex>
19222 <*package>
19223 \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
19224 </package>

```

(End definition for \c_sys_jobname_str. This variable is documented on page 217.)

36.2 Time and date

Copies of the information provided by T_EX

```

\c_sys_minute_int 19225 \int_const:Nn \c_sys_minute_int
\c_sys_hour_int    19226 { \int_mod:nn { \tex_time:D } { 60 } }
\c_sys_day_int     19227 \int_const:Nn \c_sys_hour_int
\c_sys_month_int   19228 { \int_div_truncate:nn { \tex_time:D } { 60 } }
\c_sys_year_int    19229 \int_const:Nn \c_sys_day_int { \tex_day:D }
                  19230 \int_const:Nn \c_sys_month_int { \tex_month:D }
                  19231 \int_const:Nn \c_sys_year_int { \tex_year:D }

```

(End definition for `\c_sys_minute_int` and others. These variables are documented on page 217.)

36.3 Detecting the engine

Set up the engine tests on the basis exactly one test should be true. Mainly a case of looking for the appropriate marker primitive. For upT_EX, there is a complexity in that setting `-kanji-internal=sjis` or `-kanji-internal=euc` effective makes it more like pT_EX. In those cases we therefore report pT_EX rather than upT_EX.

```

\sys_if_engine luatex_p: 19232 \clist_map_inline:nn { lua , pdf , p , up , xe }
\sys_if_engine luatex:TF 19233 {
\sys_if_engine pdftex_p: 19234   \cs_new_eq:cN { sys_if_engine_ #1 tex:T } \use_none:n
\sys_if_engine pdftex:TF 19235   \cs_new_eq:cN { sys_if_engine_ #1 tex:F } \use:n
\sys_if_engine ptex_p:   19236   \cs_new_eq:cN { sys_if_engine_ #1 tex:TF } \use_ii:nn
\sys_if_engine ptex:TF   19237   \cs_new_eq:cN { sys_if_engine_ #1 tex_p: } \c_false_bool
\sys_if_engine xetex_p:  19238 }
\sys_if_engine xetex:TF  19239 \cs_if_exist:NT \luatex luatexversion:D
\c_sys_engine_str        19240 {
                        19241   \cs_gset_eq:NN \sys_if_engine luatex:T \use:n
                        19242   \cs_gset_eq:NN \sys_if_engine luatex:F \use_none:n
                        19243   \cs_gset_eq:NN \sys_if_engine luatex:TF \use_i:nn
                        19244   \cs_gset_eq:NN \sys_if_engine luatex_p: \c_true_bool
                        19245   \str_const:Nn \c_sys_engine_str { luatex }
                        19246 }
                        19247 \cs_if_exist:NT \pdftex pdftexversion:D
                        19248 {
                        19249   \cs_gset_eq:NN \sys_if_engine pdftex:T \use:n
                        19250   \cs_gset_eq:NN \sys_if_engine pdftex:F \use_none:n
                        19251   \cs_gset_eq:NN \sys_if_engine pdftex:TF \use_i:nn
                        19252   \cs_gset_eq:NN \sys_if_engine pdftex_p: \c_true_bool
                        19253   \str_const:Nn \c_sys_engine_str { pdftex }
                        19254 }
                        19255 \cs_if_exist:NT \ptex kanjiskip:D
                        19256 {
                        19257   \bool_if:nTF
                        19258   {
                        19259     \cs_if_exist_p:N \uptex disablecjktoken:D &&
                        19260     \int_compare_p:nNn { \ptex_jis:D "2121 } = { "3000 }
                        19261   }
                        19262   {
                        19263     \cs_gset_eq:NN \sys_if_engine uptex:T \use:n
                        19264     \cs_gset_eq:NN \sys_if_engine uptex:F \use_none:n
                        19265     \cs_gset_eq:NN \sys_if_engine uptex:TF \use_i:nn
                        19266     \cs_gset_eq:NN \sys_if_engine uptex_p: \c_true_bool

```

```

19267     \str_const:Nn \c_sys_engine_str { uptex }
19268   }
19269   {
19270     \cs_gset_eq:NN \sys_if_engine_ptex:T \use:n
19271     \cs_gset_eq:NN \sys_if_engine_ptex:F \use_none:n
19272     \cs_gset_eq:NN \sys_if_engine_ptex:TF \use_i:nn
19273     \cs_gset_eq:NN \sys_if_engine_ptex_p: \c_true_bool
19274     \str_const:Nn \c_sys_engine_str { ptex }
19275   }
19276 }
19277 \cs_if_exist:NT \xetex_XeTeXversion:D
19278 {
19279   \cs_gset_eq:NN \sys_if_engine_xetex:T \use:n
19280   \cs_gset_eq:NN \sys_if_engine_xetex:F \use_none:n
19281   \cs_gset_eq:NN \sys_if_engine_xetex:TF \use_i:nn
19282   \cs_gset_eq:NN \sys_if_engine_xetex_p: \c_true_bool
19283   \str_const:Nn \c_sys_engine_str { xetex }
19284 }

```

(End definition for `\sys_if_engine luatex:TF` and others. These functions are documented on page 217.)

36.4 Detecting the output

`\sys_if_output_dvi_p:` This is a simple enough concept: the two views here are complementary.

```

\sys_if_output_dvi:TF 19285 \bool_if:nTF
\sys_if_output_dvi:TF 19286 {
\sys_if_output_pdf_p: 19287   \cs_if_exist_p:N \pdfTeX_pdfoutput:D
\sys_if_output_pdf:TF 19288   && \int_compare_p:nNn \pdfTeX_pdfoutput:D > \c_zero
\c_sys_output_str      19289 }
19290 {
19291   \cs_new_eq:NN \sys_if_output_dvi:T \use_none:n
19292   \cs_new_eq:NN \sys_if_output_dvi:F \use:n
19293   \cs_new_eq:NN \sys_if_output_dvi:TF \use_i:nn
19294   \cs_new_eq:NN \sys_if_output_dvi_p: \c_false_bool
19295   \cs_new_eq:NN \sys_if_output_pdf:T \use:n
19296   \cs_new_eq:NN \sys_if_output_pdf:F \use_none:n
19297   \cs_new_eq:NN \sys_if_output_pdf:TF \use_i:nn
19298   \cs_new_eq:NN \sys_if_output_pdf_p: \c_true_bool
19299   \str_const:Nn \c_sys_output_str { pdf }
19300 }
19301 {
19302   \cs_new_eq:NN \sys_if_output_dvi:T \use:n
19303   \cs_new_eq:NN \sys_if_output_dvi:F \use_none:n
19304   \cs_new_eq:NN \sys_if_output_dvi:TF \use_i:nn
19305   \cs_new_eq:NN \sys_if_output_dvi_p: \c_true_bool
19306   \cs_new_eq:NN \sys_if_output_pdf:T \use_none:n
19307   \cs_new_eq:NN \sys_if_output_pdf:F \use:n
19308   \cs_new_eq:NN \sys_if_output_pdf:TF \use_i:nn
19309   \cs_new_eq:NN \sys_if_output_pdf_p: \c_false_bool
19310   \str_const:Nn \c_sys_output_str { dvi }
19311 }

```

(End definition for `\sys_if_output_dvi:TF`, `\sys_if_output_pdf:TF`, and `\c_sys_output_str`. These functions are documented on page 218.)

19312 \langle /initex | package \rangle

37 l3luatex implementation

19313 \langle *initex | package \rangle

37.1 Breaking out to Lua

19314 \langle *tex \rangle

`\lua_now_x:n` Wrappers around the primitives. As with engines other than LuaTeX these have to be macros, we give them the same status in all cases. When LuaTeX is not in use, simply give an error message/
`\lua_now:n`
`\lua_shipout_x:n`
`\lua_shipout:n`
`\lua_escape_x:n`
`\lua_escape:n`

```
19315 \cs_new:Npn \lua_now_x:n #1 { \luatex_directlua:D {#1} }
19316 \cs_new:Npn \lua_now:n #1 { \lua_now_x:n { \exp_not:n {#1} } }
19317 \cs_new_protected:Npn \lua_shipout_x:n #1 { \luatex_latelua:D {#1} }
19318 \cs_new_protected:Npn \lua_shipout:n #1
19319 { \lua_shipout_x:n { \exp_not:n {#1} } }
19320 \cs_new:Npn \lua_escape_x:n #1 { \luatex_luaescapestring:D {#1} }
19321 \cs_new:Npn \lua_escape:n #1 { \lua_escape_x:n { \exp_not:n {#1} } }
19322 \sys_if_engine luatex:F
19323 {
19324   \clist_map_inline:nn
19325   { \lua_now_x:n , \lua_now:n , \lua_escape_x:n , \lua_escape:n }
19326   {
19327     \cs_set:Npn #1 ##1
19328     {
19329       \__msg_kernel_expandable_error:nnn
19330       { kernel } { luatex-required } { #1 }
19331     }
19332   }
19333   \clist_map_inline:nn
19334   { \lua_shipout_x :n , \lua_shipout:n }
19335   {
19336     \cs_set_protected:Npn #1 ##1
19337     {
19338       \__msg_kernel_error:nnn
19339       { kernel } { luatex-required } { #1 }
19340     }
19341   }
19342 }
```

(End definition for `\lua_now_x:n` and others. These functions are documented on page 219.)

37.2 Messages

```
19343 \__msg_kernel_new:nnnn { kernel } { luatex-required }
19344 { LuaTeX~engine~not~in~use!~Ignoring~#1. }
19345 {
19346   The~feature~you~are~using~is~only~available~
19347   with~the~LuaTeX~engine.~LaTeX3~ignored~'~#1~'.
19348 }
19349  $\langle$ /tex $\rangle$ 
```

37.3 Lua functions for internal use

19350 $\langle *lua \rangle$

l3kernel Create a table for the kernel's own use.

19351 `l3kernel = l3kernel or { }`

(End definition for l3kernel.)

Various local copies of standard functions: naming convention is to retain the full text but replace all `.` by `_`.

19352 `local tex_setcatcode = tex.setcatcode`

19353 `local tex_sprint = tex.sprint`

19354 `local tex_write = tex.write`

19355 `local unicode_utf8_char = unicode.utf8.char`

l3kernel.strcmp String comparison which gives the same results as pdfTeX's `\pdfstrcmp`, although the ordering should likely not be relied upon!

19356 `local function strcmp (A, B)`

19357 `if A == B then`

19358 `tex_write("0")`

19359 `elseif A < B then`

19360 `tex_write("-1")`

19361 `else`

19362 `tex_write("1")`

19363 `end`

19364 `end`

19365 `l3kernel.strcmp = strcmp`

(End definition for l3kernel.strcmp.)

l3kernel.charcat Creating arbitrary chars needs a category code table. As set up here, one may have been assigned earlier (see `l3bootstrap`) or a hard-coded one is used. The latter is intended for format mode and should be adjusted to match an eventual allocator.

19366 `local charcat_table = l3kernel.charcat_table or 1`

19367 `local function charcat (charcode, catcode)`

19368 `tex_setcatcode(charcat_table, charcode, catcode)`

19369 `tex_sprint(charcat_table, unicode_utf8_char(charcode))`

19370 `end`

19371 `l3kernel.charcat = charcat`

(End definition for l3kernel.charcat.)

19372 $\langle /lua \rangle$

19373 $\langle /initex | package \rangle$

37.4 Format mode code: font loader

19374 $\langle *fontloader \rangle$

In format mode, there needs to be a font loader available to let us use OpenType fonts. For testing, this is provided by `fontloader.lua` from the Speedata Publisher system (<https://github.com/speedata/publisher>). The code there is designed to be self-contained and has a certain number of build-in assumptions, so there is a small amount of compatibility required.

The code we load looks up `texmf` tree files using `kpse.filelist`, which isn't part of the standard `kpse` library. The interface is emulated using `metatable`.

```

19375 kpse.filelist = setmetatable({}, {
19376   __index = function (t, key)
19377     return kpse.lookup(key)
19378   end
19379 })

```

There is a built-in assumption in `fontloader.lua` that various environmental variables are set. We deal with that by intercepting the relevant names and returning something sane.

```

19380 local os_getenv = os.getenv
19381 function os.getenv (var)
19382   if var == "SP_FONT_PATH" then return "" end
19383   return os.getenv(var)
19384 end

```

As detailed in <https://github.com/speedata/publisher/blob/develop/COPYING>, the current license for Speedata Publisher is AGPLv3. We therefore only load the file and use its public interfaces rather than copying/modifying the code itself. Note though that we do have permission to use `fontloader.lua` as a public domain work (<http://chat.stackexchange.com/transcript/message/27273687#27273687>): if we want to develop a richer loader we may want to take advantage of that (which also applies to the simple shaper in the related `fonts.lua` file).

```

19385 local fontloader = require("fontloader.lua")

```

That done, register a callback which at present simply passes everything through. There's no attempt to pick up font settings (which presumably will be needed). Syntax is coerced to the same as for \TeX .

```

19386 callback.register("define_font",
19387   function (name, size, id)
19388     local opts, opttab, otfeatures = "", { }, { }
19389     if string.match(name, "^%[" then
19390       name, opts = string.match(name, "^%[[^%]]*)%["^:]*:?(.*)")
19391     end
19392     if opts ~= "" then
19393       for _,kv in ipairs(string.explode(opts,";")) do
19394         if string.match(kv, "=") then
19395           local k, v = string.match(kv, "[^=]*=?(.*)")
19396           opttab[k] = v
19397         else
19398           if string.match(kv, "^+") then
19399             otfeatures[string.sub(kv,2,-1)] = "true"
19400           elseif string.match(kv, "^-") then
19401             otfeatures[string.sub(kv,2,-1)] = "false"
19402           else
19403             otfeatures[kv] = "true"
19404           end
19405         end
19406       end
19407     end
19408     if next(otfeatures) then
19409       opttab["otfeatures"] = otfeatures
19410     end
19411     return select(2, fontloader.define_font(name, size, opttab))

```

```

19412   end
19413 )
19414 </fontloader>

```

38 l3drivers Implementation

```

19415 (*initex | package)
19416 (@@=driver)

```

Whilst there is a reasonable amount of code overlap between drivers, it is much clearer to have the blocks more-or-less separated than run in together and DocStripped out in parts. As such, most of the following is set up on a per-driver basis, though there is some common code (again given in blocks not interspersed with other material).

All the file identifiers are up-front so that they come out in the right place in the files.

```

19417 <*package>
19418 \ProvidesExplFile
19419 <*dvipdfmx>
19420 {l3dvidpfmx.def}{\ExplFileDate}{\ExplFileVersion}
19421 {L3 Experimental driver: dvipdfmx}
19422 </dvipdfmx>
19423 <*dvips>
19424 {l3dvips.def}{\ExplFileDate}{\ExplFileVersion}
19425 {L3 Experimental driver: dvips}
19426 </dvips>
19427 <*dvisvgm>
19428 {l3dvisvgm.def}{\ExplFileDate}{\ExplFileVersion}
19429 {L3 Experimental driver: dvisvgm}
19430 </dvisvgm>
19431 <*pdfmode>
19432 {l3pdfmode.def}{\ExplFileDate}{\ExplFileVersion}
19433 {L3 Experimental driver: PDF mode}
19434 </pdfmode>
19435 <*xdvipdfmx>
19436 {l3xdvidpfmx.def}{\ExplFileDate}{\ExplFileVersion}
19437 {L3 Experimental driver: xdvipdfmx}
19438 </xdvipdfmx>
19439 </package>

```

38.1 pdfmode driver

```

19440 <*pdfmode>

```

The direct PDF driver covers both pdf_{TEX} and Lua_{TEX}. The latter renames/restructures the driver primitives but this can be handled at one level of abstraction. As such, we avoid using two separate drivers for this material at the cost of some x-type definitions to get everything expanded up-front.

38.1.1 Basics

`_driver_literal:n` This is equivalent to `\special{pdf:}` but the engine can track it. Without the `direct` keyword everything is kept in sync: the transformation matrix is set to the current point automatically. Note that this is still inside the text (BT ...ET block).

```

19441 \cs_new_protected:Npx \_driver_literal:n #1
19442 {
19443   \cs_if_exist:NTF \luatex_pdfextension:D

```



```

19444     { \luatex_pdfextension:D literal }
19445     { \pdftex_pdfliteral:D }
19446     {#1}
19447   }

```

(End definition for _driver_literal:n.)

_driver_scope_begin: Higher-level interfaces for saving and restoring the graphic state.

```

\_driver_scope_end:
19448 \cs_new_protected:Npx \_driver_scope_begin:
19449 {
19450   \cs_if_exist:NTF \luatex_pdfextension:D
19451   { \luatex_pdfextension:D save \scan_stop: }
19452   { \pdftex_pdfsave:D }
19453 }
19454 \cs_new_protected:Npx \_driver_scope_end:
19455 {
19456   \cs_if_exist:NTF \luatex_pdfextension:D
19457   { \luatex_pdfextension:D restore \scan_stop: }
19458   { \pdftex_pdfrestore:D }
19459 }

```

(End definition for _driver_scope_begin: and _driver_scope_end:.)

_driver_matrix:n Here the appropriate function is set up to insert an affine matrix into the PDF. With pdfTeX and LuaTeX in direct PDF output mode there is a primitive for this, which only needs the rotation/scaling/skew part.

```

19460 \cs_new_protected:Npx \_driver_matrix:n #1
19461 {
19462   \cs_if_exist:NTF \luatex_pdfextension:D
19463   { \luatex_pdfextension:D setmatrix }
19464   { \pdftex_pdfsetmatrix:D }
19465   {#1}
19466 }

```

(End definition for _driver_matrix:n.)

38.1.2 Color

\l_driver_current_color_tl The current color in driver-dependent format: pick up the package-mode data if available.

```

19467 \tl_new:N \l_driver_current_color_tl
19468 \tl_set:Nn \l_driver_current_color_tl { 0~g~0~G }
19469 <*package>
19470 \AtBeginDocument
19471 {
19472   \@ifpackageloaded { color }
19473   { \tl_set:Nn \l_driver_current_color_tl { \current@color } }
19474   { }
19475 }
19476 </package>

```

(End definition for \l_driver_current_color_tl.)

\l_driver_color_stack_int pdfTeX and LuaTeX have multiple stacks available, and to track which one is in use a variable is required.

```

19477 \int_new:N \l_driver_color_stack_int

```

(End definition for \l__driver_color_stack_int.)

`_driver_color_ensure_current:` There is a dedicated primitive/primitive interface for setting colors. As with scoping, `_driver_color_reset:` this approach is not suitable for cached operations.

```

19478 \cs_new_protected:Npx \_driver_color_ensure_current:
19479 {
19480   \cs_if_exist:NTF \luatex_pdfextension:D
19481     { \luatex_pdfextension:D colorstack }
19482     { \pdfTEX_pdfcolorstack:D }
19483     \exp_not:N \l__driver_color_stack_int push
19484     { \exp_not:N \l__driver_current_color_tl }
19485   \group_insert_after:N \exp_not:N \_driver_color_reset:
19486 }
19487 \cs_new_protected:Npx \_driver_color_reset:
19488 {
19489   \cs_if_exist:NTF \luatex_pdfextension:D
19490     { \luatex_pdfextension:D colorstack }
19491     { \pdfTEX_pdfcolorstack:D }
19492     \exp_not:N \l__driver_color_stack_int pop \scan_stop:
19493 }

```

(End definition for _driver_color_ensure_current: and _driver_color_reset:.)

19494 `\pdfmode`

38.2 dvipdfmx driver

19495 `*dvipdfmx |xdvipdfmx`

The `dvipdfmx` shares code with the PDF mode one (using the common section to this file) but also with `xdvipdfmx`. The latter is close to identical to `dvipdfmx` and so all of the code here is extracted for both drivers, with some `clean up` for `xdvipdfmx` as required.

38.2.1 Basics

`_driver_literal:n` Equivalent to `pdf:content` but favoured as the link to the pdfTeX primitive approach is clearer.

```

19496 \cs_new_protected:Npn \_driver_literal:n #1
19497 { \tex_special:D { pdf:literal~ #1 } }

```

(End definition for _driver_literal:n.)

`_driver_scope_begin:` Scoping is done using direct PDF operations here.

```

\_driver_scope_end:
19498 \cs_new_protected:Npn \_driver_scope_begin:
19499 { \_driver_literal:n { q } }
19500 \cs_new_protected:Npn \_driver_scope_end:
19501 { \_driver_literal:n { Q } }

```

(End definition for _driver_scope_begin: and _driver_scope_end:.)

`_driver_matrix:n` With (x)dvipdfmx the matrix has to include a translation part: that is always zero and so is built in here so that the same internal interface works for all PDF-related drivers.

```

19502 \cs_new_protected:Npn \_driver_matrix:n #1
19503 { \_driver_literal:n { #1 \c_space_tl 0~0~cm } }

```

(End definition for _driver_matrix:n.)

38.2.2 Color

`\l__driver_current_color_tl` The current color in driver-dependent format.

```

19504 \tl_new:N \l__driver_current_color_tl
19505 \tl_set:Nn \l__driver_current_color_tl { [ 0 ] }
19506 \*package
19507 \AtBeginDocument
19508 {
19509   \@ifpackageloaded { color }
19510     { \tl_set:Nn \l__driver_current_color_tl { \current@color } }
19511     { }
19512 }
19513 \*package

```

(End definition for `\l__driver_current_color_tl`.)

`__driver_color_ensure_current:` Directly set the color using the specials with optimisation support.

```

\__driver_color_reset:
19514 \cs_new_protected:Npn \__driver_color_ensure_current:
19515 {
19516   \tex_special:D { pdf:bcolor~\l__driver_current_color_tl }
19517   \group_insert_after:N \__driver_color_reset:
19518 }
19519 \cs_new_protected:Npn \__driver_color_reset:
19520 { \tex_special:D { pdf:ecolor } }

```

(End definition for `__driver_color_ensure_current:` and `__driver_color_reset:.`)

```

19521 \*xdvipdfmx | xdvipdfmx

```

38.3 xdvipdfmx driver

```

19522 \*xdvipdfmx

```

38.3.1 Color

`__driver_color_ensure_current:` The L^AT_EX 2_ε driver uses dvips-like specials so there has to be a change of set up if color is loaded.

```

\__driver_color_reset:
19523 \*package
19524 \AtBeginDocument
19525 {
19526   \@ifpackageloaded { color }
19527     {
19528       \cs_set_protected:Npn \__driver_color_ensure_current:
19529         {
19530           \tex_special:D { color~push~\l__driver_current_color_tl }
19531           \group_insert_after:N \__driver_color_reset:
19532         }
19533       \cs_set_protected:Npn \__driver_color_reset:
19534         { \tex_special:D { color~pop } }
19535     }
19536   { }
19537 }
19538 \*package

```

(End definition for `__driver_color_ensure_current:` and `__driver_color_reset:.`)

```

19539 \*xdvipdfmx

```

38.4 Common code for PDF production

As all of the drivers which understand PDF-targeted specials act in much the same way there is a lot of shared code. Rather than try to DocStrip it interspersed with the above, we collect all of it here.

```
19540 <*dvipdfmx | pdfmode | xdvi pdfmx>
```

38.4.1 Box operations

_driver_box_use_clip:N The general method is to save the current location, define a clipping path equivalent to the bounding box, then insert the content at the current position and in a zero width box. The “real” width is then made up using a horizontal skip before tidying up. There are other approaches that can be taken (for example using XForm objects), but the logic here shares as much code as possible and uses the same conversions (and so same rounding errors) in all cases.

```
19541 \cs_new_protected:Npn \_driver_box_use_clip:N #1
19542 {
19543   \_driver_scope_begin:
19544   \_driver_literal:n
19545   {
19546     0~
19547     \dim_to_decimal_in_bp:n { -\box_dp:N #1 } ~
19548     \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
19549     \dim_to_decimal_in_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~
19550     re~W~n
19551   }
19552   \hbox_overlap_right:n { \box_use:N #1 }
19553   \_driver_scope_end:
19554   \skip_horizontal:n { \box_wd:N #1 }
19555 }
```

(End definition for _driver_box_use_clip:N.)

_driver_box_use_rotate:Nn Rotations are set using an affine transformation matrix which therefore requires sine/cosine values not the angle itself. We store the rounded values to avoid rounding twice. There are also a couple of comparisons to ensure that -0 is not written to the output, as this avoids any issues with problematic display programs. Note that numbers are compared to 0 after rounding.

```
19556 \cs_new_protected:Npn \_driver_box_use_rotate:Nn #1#2
19557 {
19558   \_driver_scope_begin:
19559   \box_set_wd:Nn #1 \c_zero_dim
19560   \fp_set:Nn \l__driver_cos_fp { round ( cosd ( #2 ) , 5 ) }
19561   \fp_compare:nNnT \l__driver_cos_fp = \c_zero_fp
19562     { \fp_zero:N \l__driver_cos_fp }
19563   \fp_set:Nn \l__driver_sin_fp { round ( sind ( #2 ) , 5 ) }
19564   \_driver_matrix:n
19565   {
19566     \fp_use:N \l__driver_cos_fp \c_space_tl
19567     \fp_compare:nNnTF \l__driver_sin_fp = \c_zero_fp
19568       { 0~0 }
19569     {
19570       \fp_use:N \l__driver_sin_fp
```

```

19571         \c_space_tl
19572         \fp_eval:n { -\l__driver_sin_fp }
19573     }
19574     \c_space_tl
19575     \fp_use:N \l__driver_cos_fp
19576 }
19577 \box_use:N #1
19578 \__driver_scope_end:
19579 }
19580 \fp_new:N \l__driver_cos_fp
19581 \fp_new:N \l__driver_sin_fp

```

(End definition for `__driver_box_use_rotate:Nn`, `\l__driver_cos_fp`, and `\l__driver_sin_fp`.)

`__driver_box_use_scale:Nnn` The same idea as for rotation but without the complexity of signs and cosines.

```

19582 \cs_new_protected:Npn \__driver_box_use_scale:Nnn #1#2#3
19583 {
19584     \__driver_scope_begin:
19585     \__driver_matrix:n
19586     {
19587         \fp_eval:n { round ( #2 , 5 ) } ~
19588         0~0~
19589         \fp_eval:n { round ( #3 , 5 ) }
19590     }
19591     \hbox_overlap_right:n { \box_use:N #1 }
19592     \__driver_scope_end:
19593 }

```

(End definition for `__driver_box_use_scale:Nnn`.)

38.5 Drawing

`__driver_draw_literal:n` Pass data through using a dedicated interface.

```

\__driver_draw_literal:x
19594 \cs_new_eq:NN \__driver_draw_literal:n \__driver_literal:n
19595 \cs_generate_variant:Nn \__driver_draw_literal:n { x }

```

(End definition for `__driver_draw_literal:n`.)

`__driver_draw_begin:` No special requirements here, so simply set up a drawing scope.

```

\__driver_draw_end:
19596 \cs_new_protected:Npn \__driver_draw_begin:
19597 { \__driver_draw_scope_begin: }
19598 \cs_new_protected:Npn \__driver_draw_end:
19599 { \__driver_draw_scope_end: }

```

(End definition for `__driver_draw_begin:` and `__driver_draw_end:.`)

`__driver_draw_scope_begin:` In contrast to a general scope, a drawing scope is always done using the PDF operators
`__driver_draw_scope_end:` so is the same for all relevant drivers.

```

19600 \cs_new_protected:Npn \__driver_draw_scope_begin:
19601 { \__driver_draw_literal:n { q } }
19602 \cs_new_protected:Npn \__driver_draw_scope_end:
19603 { \__driver_draw_literal:n { Q } }

```

(End definition for `__driver_draw_scope_begin:` and `__driver_draw_scope_end:.`)

`__driver_draw_moveto:nn` Path creation operations all resolve directly to PDF primitive steps, with only the need to convert to bp. Notice that x-type expansion is included here to ensure that any variable values are forced to literals before any possible caching.

```

19604 \cs_new_protected:Npn \__driver_draw_moveto:nn #1#2
19605 {
19606   \__driver_draw_literal:x
19607   { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ m }
19608 }
19609 \cs_new_protected:Npn \__driver_draw_lineto:nn #1#2
19610 {
19611   \__driver_draw_literal:x
19612   { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ l }
19613 }
19614 \cs_new_protected:Npn \__driver_draw_curveto:nnnnnn #1#2#3#4#5#6
19615 {
19616   \__driver_draw_literal:x
19617   {
19618     \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
19619     \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
19620     \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
19621     c
19622   }
19623 }
19624 \cs_new_protected:Npn \__driver_draw_rectangle:nnnn #1#2#3#4
19625 {
19626   \__driver_draw_literal:x
19627   {
19628     \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
19629     \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
19630     re
19631   }
19632 }

```

(End definition for `__driver_draw_moveto:nn` and others.)

`__driver_draw_evenodd_rule:` The even-odd rule here can be implemented as a simply switch.

```

19633 \cs_new_protected:Npn \__driver_draw_evenodd_rule:
19634 { \bool_gset_true:N \g__driver_draw_eor_bool }
19635 \cs_new_protected:Npn \__driver_draw_nonzero_rule:
19636 { \bool_gset_false:N \g__driver_draw_eor_bool }
19637 \bool_new:N \g__driver_draw_eor_bool

```

(End definition for `__driver_draw_evenodd_rule:`, `__driver_draw_nonzero_rule:`, and `\g__driver_draw_eor_bool`.)

`__driver_draw_closepath:` Converting paths to output is again a case of mapping directly to PDF operations.

```

19638 \cs_new_protected:Npn \__driver_draw_closepath:
19639 { \__driver_draw_literal:n { h } }
19640 \cs_new_protected:Npn \__driver_draw_stroke:
19641 { \__driver_draw_literal:n { S } }
19642 \cs_new_protected:Npn \__driver_draw_closestroke:
19643 { \__driver_draw_literal:n { s } }
19644 \cs_new_protected:Npn \__driver_draw_fill:
19645 {

```

```

19646     \__driver_draw_literal:x
19647     { f \bool_if:NT \g__driver_draw_eor_bool * }
19648   }
19649   \cs_new_protected:Npn \__driver_draw_fillstroke:
19650   {
19651     \__driver_draw_literal:x
19652     { B \bool_if:NT \g__driver_draw_eor_bool * }
19653   }
19654   \cs_new_protected:Npn \__driver_draw_clip:
19655   {
19656     \__driver_draw_literal:x
19657     { W \bool_if:NT \g__driver_draw_eor_bool * }
19658   }
19659   \cs_new_protected:Npn \__driver_draw_discardpath:
19660   { \__driver_draw_literal:n { n } }

```

(End definition for __driver_draw_closepath: and others.)

```

\__driver_draw_dash:nn
\__driver_draw_dash:n
\__driver_draw_linewidth:n
\__driver_draw_miterlimit:n
\__driver_draw_cap_butt:
\__driver_draw_cap_round:
\__driver_draw_cap_rectangle:
\__driver_draw_join_miter:
\__driver_draw_join_round:
\__driver_draw_join_bevel:

```

Converting paths to output is again a case of mapping directly to PDF operations.

```

19661 \cs_new_protected:Npn \__driver_draw_dash:nn #1#2
19662 {
19663   \__driver_draw_literal:x
19664   {
19665     [ ~
19666       \clist_map_function:nN {#1} \__driver_draw_dash:n
19667     ] ~
19668     \dim_to_decimal_in_bp:n {#2} ~ d
19669   }
19670 }
19671 \cs_new:Npn \__driver_draw_dash:n #1
19672 { \dim_to_decimal_in_bp:n {#1} ~ }
19673 \cs_new_protected:Npn \__driver_draw_linewidth:n #1
19674 {
19675   \__driver_draw_literal:x
19676   { \dim_to_decimal_in_bp:n {#1} ~ w }
19677 }
19678 \cs_new_protected:Npn \__driver_draw_miterlimit:n #1
19679 { \__driver_draw_literal:x { \fp_eval:n {#1} ~ M } }
19680 \cs_new_protected:Npn \__driver_draw_cap_butt:
19681 { \__driver_draw_literal:n { 0 ~ J } }
19682 \cs_new_protected:Npn \__driver_draw_cap_round:
19683 { \__driver_draw_literal:n { 1 ~ J } }
19684 \cs_new_protected:Npn \__driver_draw_cap_rectangle:
19685 { \__driver_draw_literal:n { 2 ~ J } }
19686 \cs_new_protected:Npn \__driver_draw_join_miter:
19687 { \__driver_draw_literal:n { 0 ~ j } }
19688 \cs_new_protected:Npn \__driver_draw_join_round:
19689 { \__driver_draw_literal:n { 1 ~ j } }
19690 \cs_new_protected:Npn \__driver_draw_join_bevel:
19691 { \__driver_draw_literal:n { 2 ~ j } }

```

(End definition for __driver_draw_dash:nn and others.)

```

\__driver_draw_color_cmyk:nnnn
\__driver_draw_color_cmyk_fill:nnnn
\__driver_draw_color_cmyk_stroke:nnnn
\__driver_draw_color_cmyk_aux:nnnn
\__driver_draw_color_gray:n
\__driver_draw_color_gray_fill:n
\__driver_draw_color_gray_stroke:n
\__driver_draw_color_gray_aux:n
\__driver_draw_color_rgb:nnn
\__driver_draw_color_rgb_fill:nnn
\__driver_draw_color_rgb_stroke:nnn
\__driver_draw_color_rgb_aux:nnn

```

Yet more fast conversion, all using the FPU to allow for expressions in numerical input.

```

19692 \cs_new_protected:Npn \__driver_draw_color_cmyk:nnnn #1#2#3#4
19693 {
19694     \use:x
19695     {
19696         \__driver_draw_color_cmyk_aux:nnnn
19697         { \fp_eval:n {#1} }
19698         { \fp_eval:n {#2} }
19699         { \fp_eval:n {#3} }
19700         { \fp_eval:n {#4} }
19701     }
19702 }
19703 \cs_new_protected:Npn \__driver_draw_color_cmyk_aux:nnnn #1#2#3#4
19704 {
19705     \__driver_draw_literal:n
19706     { #1 ~ #2 ~ #3 ~ #4 ~ k ~ #1 ~ #2 ~ #3 ~ #4 ~ K }
19707 }
19708 \cs_new_protected:Npn \__driver_draw_color_cmyk_fill:nnnn #1#2#3#4
19709 {
19710     \__driver_draw_literal:x
19711     {
19712         \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
19713         \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
19714         k
19715     }
19716 }
19717 \cs_new_protected:Npn \__driver_draw_color_cmyk_stroke:nnnn #1#2#3#4
19718 {
19719     \__driver_draw_literal:x
19720     {
19721         \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
19722         \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
19723         K
19724     }
19725 }
19726 \cs_new_protected:Npn \__driver_draw_color_gray:n #1
19727 {
19728     \use:x
19729     { \__driver_draw_color_gray_aux:n { \fp_eval:n {#1} } }
19730 }
19731 \cs_new_protected:Npn \__driver_draw_color_gray_aux:n #1
19732 {
19733     \__driver_draw_literal:n { #1 ~ g ~ #1 ~ G }
19734 }
19735 \cs_new_protected:Npn \__driver_draw_color_gray_fill:n #1
19736 { \__driver_draw_literal:x { \fp_eval:n {#1} ~ g } }
19737 \cs_new_protected:Npn \__driver_draw_color_gray_stroke:n #1
19738 { \__driver_draw_literal:x { \fp_eval:n {#1} ~ G } }
19739 \cs_new_protected:Npn \__driver_draw_color_rgb:nnn #1#2#3
19740 {
19741     \use:x
19742     {
19743         \__driver_draw_color_rgb_aux:nnn
19744         { \fp_eval:n {#1} }
19745         { \fp_eval:n {#2} }

```



```

19746         { \fp_eval:n {#3} }
19747     }
19748 }
19749 \cs_new_protected:Npn \__driver_draw_color_rgb_aux:nnn #1#2#3
19750 {
19751     \__driver_draw_literal:n
19752     { #1 ~ #2 ~ #3 ~ rg ~ #1 ~ #2 ~ #3 ~ RG }
19753 }
19754 \cs_new_protected:Npn \__driver_draw_color_rgb_fill:nnn #1#2#3
19755 {
19756     \__driver_draw_literal:x
19757     { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ rg }
19758 }
19759 \cs_new_protected:Npn \__driver_draw_color_rgb_stroke:nnn #1#2#3
19760 {
19761     \__driver_draw_literal:x
19762     { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ RG }
19763 }

```

(End definition for __driver_draw_color_cmyk:nnnn and others.)

`__driver_draw_transformcm:nnnnnn` The first four arguments here are floats (the affine matrix), the last two are a displacement vector. Once again, force evaluation to allow for caching.

```

19764 \cs_new_protected:Npn \__driver_draw_transformcm:nnnnnn #1#2#3#4#5#6
19765 {
19766     \__driver_draw_literal:x
19767     {
19768         \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
19769         \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
19770         \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
19771         cm
19772     }
19773 }

```

(End definition for __driver_draw_transformcm:nnnnnn.)

`__driver_draw_hbox:Nnnnnnn`
`\l__driver_tmp_box` Inserting a \TeX box transformed to the requested position and using the current matrix is done using a mixture of \TeX and low-level manipulation. The offset can be handled by \TeX , so only any rotation/skew/scaling component needs to be done using the matrix operation. As this operation can never be cached, the scope is set directly not using the `draw` version.

```

19774 \cs_new_protected:Npn \__driver_draw_hbox:Nnnnnnn #1#2#3#4#5#6#7
19775 {
19776     \hbox_set:Nn \l__driver_tmp_box
19777     {
19778         \tex_kern:D \__dim_eval:w #6 \__dim_eval_end:
19779         \__driver_scope_begin:
19780         \__driver_draw_transformcm:nnnnnn {#2} {#3} {#4} {#5}
19781         { Opt } { Opt }
19782         \box_move_up:nn {#7} { \box_use:N #1 }
19783         \__driver_scope_end:
19784     }
19785     \box_set_wd:Nn \l__driver_tmp_box { Opt }
19786     \box_set_ht:Nn \l__driver_tmp_box { Opt }

```

```

19787 \box_set_dp:Nn \l__driver_tmp_box { Opt }
19788 \box_use:N \l__driver_tmp_box
19789 }
19790 \box_new:N \l__driver_tmp_box

(End definition for \__driver_draw_hbox:Nnnnnnn and \l__driver_tmp_box.)

19791 </dvipdfmx | pdfmode | xdvipdfmx>

```

38.6 dvips driver

```

19792 <*dvips>

```

38.6.1 Basics

`__driver_literal:n` In the case of `dvips` there is no build-in saving of the current position, and so some additional PostScript is required to set up the transformation matrix and also to restore it afterwards. Notice the use of the stack to save the current position “up front” and to move back to it at the end of the process.

```

19793 \cs_new_protected:Npn \__driver_literal:n #1
19794 {
19795   \tex_special:D
19796   {
19797     ps:
19798     currentpoint~
19799     currentpoint~translate~
19800     #1 ~
19801     neg~exch~neg~exch~translate
19802   }
19803 }

```

(End definition for `__driver_literal:n`.)

`__driver_scope_begin:` Scope saving/restoring is done directly with no need to worry about the transformation matrix. General scoping is only for the graphics stack so the lower-cost `gsave`/`grestore` pair are used.

```

19804 \cs_new_protected:Npn \__driver_scope_begin:
19805 { \tex_special:D { ps:gsave } }
19806 \cs_new_protected:Npn \__driver_scope_end:
19807 { \tex_special:D { ps:grestore } }

```

(End definition for `__driver_scope_begin:` and `__driver_scope_end:.`)

38.7 Driver-specific auxiliaries

`__driver_absolute_lengths:n` The `dvips` driver scales all absolute dimensions based on the output resolution selected and any T_EX magnification. Thus for any operation involving absolute lengths there is a correction to make. This is based on `normalscale` from `special.pro` but using the stack rather than a definition to save the current matrix.

```

19808 \cs_new:Npn \__driver_absolute_lengths:n #1
19809 {
19810   matrix~currentmatrix~
19811   Resolution~72~div~VResolution~72~div~scale~
19812   DVImag~dup~scale~
19813   #1 ~

```

```

19814     setmatrix
19815 }

```

(End definition for `_driver_absolute_lengths:n`.)

38.7.1 Box operations

`_driver_box_use_clip:N` Much the same idea as for the PDF mode version but with a slightly different syntax for creating the clip path. To avoid any scaling issues we need the absolute length auxiliary here.

```

19816 \cs_new_protected:Npn \_driver_box_use_clip:N #1
19817 {
19818   \_driver_scope_begin:
19819   \_driver_literal:n
19820   {
19821     \_driver_absolute_lengths:n
19822     {
19823       0 ~
19824       \dim_to_decimal_in_bp:n { \box_dp:N #1 } ~
19825       \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
19826       \dim_to_decimal_in_bp:n { -\box_ht:N #1 - \box_dp:N #1 } ~
19827       rectclip
19828     }
19829   }
19830   \hbox_overlap_right:n { \box_use:N #1 }
19831   \_driver_scope_end:
19832   \skip_horizontal:n { \box_wd:N #1 }
19833 }

```

(End definition for `_driver_box_use_clip:N`.)

`_driver_box_use_rotate:Nn` Rotating using dvips does not require that the box dimensions are altered and has a very convenient built-in operation. Zero rotation must be written as 0 not -0 so there is a quick test.

```

19834 \cs_new_protected:Npn \_driver_box_use_rotate:Nn #1#2
19835 {
19836   \_driver_scope_begin:
19837   \_driver_literal:n
19838   {
19839     \fp_compare:nNnTF {#2} = \c_zero_fp
19840     { 0 }
19841     { \fp_eval:n { round ( -#2 , 5 ) } } ~
19842     rotate
19843   }
19844   \box_use:N #1
19845   \_driver_scope_end:
19846 }
19847 % \end{macro}
19848 %
19849 % \begin{macro}{\_driver_box_use_scale:Nnn}
19850 %   The \texttt{dvips} driver once again has a dedicated operation we can
19851 %   use here.
19852 %   \begin{macrocode}
19853 \cs_new_protected:Npn \_driver_box_use_scale:Nnn #1#2#3

```

```

19854 {
19855   \__driver_scope_begin:
19856   \__driver_literal:n
19857   {
19858     \fp_eval:n { round ( #2 , 5 ) } ~
19859     \fp_eval:n { round ( #3 , 5 ) } ~
19860     scale
19861   }
19862   \hbox_overlap_right:n { \box_use:N #1 }
19863   \__driver_scope_end:
19864 }

```

(End definition for __driver_box_use_rotate:Nn.)

38.7.2 Color

`\l__driver_current_color_tl` The current color in driver-dependent format.

```

19865 \tl_new:N \l__driver_current_color_tl
19866 \tl_set:Nn \l__driver_current_color_tl { gray~0 }
19867 \*package)
19868 \AtBeginDocument
19869 {
19870   \ifpackageloaded { color }
19871   { \tl_set:Nn \l__driver_current_color_tl { \current@color } }
19872   { }
19873 }
19874 \*package)

```

(End definition for \l__driver_current_color_tl.)

`__driver_color_ensure_current:` Directly set the color using the specials: no optimisation here.

```

\__driver_color_reset:
19875 \cs_new_protected:Npn \__driver_color_ensure_current:
19876 {
19877   \tex_special:D { color~push~\l__driver_current_color_tl }
19878   \group_insert_after:N \__driver_color_reset:
19879 }
19880 \cs_new_protected:Npn \__driver_color_reset:
19881 { \tex_special:D { color~pop } }

```

(End definition for __driver_color_ensure_current: and __driver_color_reset:.)

38.8 Drawing

`__driver_draw_literal:n` Literals with no positioning (using `ps`: each one is positioned but cut of from everything else, so no good for the stepwise approach needed here).

`__driver_draw_literal:x`

```

19882 \cs_new_protected:Npn \__driver_draw_literal:n #1
19883 { \tex_special:D { ps:: ~ #1 } }
19884 \cs_generate_variant:Nn \__driver_draw_literal:n { x }

```

(End definition for __driver_draw_literal:n.)

`__driver_draw_begin:` The `ps::[begin]` special here deals with positioning but allows us to continue on to a matching `ps::[end]`: contrast with `ps:`, which positions but where we can't split material between separate calls. The `@beginspecial/@endspecial` pair are from `special.pro` and correct the scale and *y*-axis direction. The reference point at the start of the box is saved (as `13x/13y`) as it is needed when inserting various items.

```

19885 \cs_new_protected:Npn \__driver_draw_begin:
19886 {
19887   \tex_special:D { ps::[begin] }
19888   \tex_special:D { ps::~save }
19889   \tex_special:D { ps::~/13x~currentpoint~/13y~exch~def~def }
19890   \tex_special:D { ps::~@beginspecial }
19891 }
19892 \cs_new_protected:Npn \__driver_draw_end:
19893 {
19894   \tex_special:D { ps::~@endspecial }
19895   \tex_special:D { ps::~restore }
19896   \tex_special:D { ps::[end] }
19897 }

```

(End definition for `__driver_draw_begin:` and `__driver_draw_end:.`)

`__driver_draw_scope_begin:` Scope here may need to contain saved definitions, so the entire memory rather than just the graphic state has to be sent to the stack.

```

19898 \cs_new_protected:Npn \__driver_draw_scope_begin:
19899 { \__driver_draw_literal:n { save } }
19900 \cs_new_protected:Npn \__driver_draw_scope_end:
19901 { \__driver_draw_literal:n { restore } }

```

(End definition for `__driver_draw_scope_begin:` and `__driver_draw_scope_end:.`)

`__driver_draw_moveto:nn` Path creation operations mainly resolve directly to PostScript primitive steps, with only the need to convert to `bp`. Notice that `x`-type expansion is included here to ensure that any variable values are forced to literals before any possible caching. There is no native rectangular path command (without also clipping, filling or stroking), so that task is done using a small amount of PostScript.

```

19902 \cs_new_protected:Npn \__driver_draw_moveto:nn #1#2
19903 {
19904   \__driver_draw_literal:x
19905   { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ moveto }
19906 }
19907 \cs_new_protected:Npn \__driver_draw_lineto:nn #1#2
19908 {
19909   \__driver_draw_literal:x
19910   { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ lineto }
19911 }
19912 \cs_new_protected:Npn \__driver_draw_rectangle:nnnn #1#2#3#4
19913 {
19914   \__driver_draw_literal:x
19915   {
19916     \dim_to_decimal_in_bp:n {#4} ~ \dim_to_decimal_in_bp:n {#3} ~
19917     \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
19918     moveto~dup~0~rlineto~exch~0~exch~rlineto~neg~0~rlineto~clospath
19919   }

```

```

19920 }
19921 \cs_new_protected:Npn \__driver_draw_curveto:nnnnnn #1#2#3#4#5#6
19922 {
19923   \__driver_draw_literal:x
19924   {
19925     \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
19926     \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
19927     \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
19928     curveto
19929   }
19930 }

```

(End definition for __driver_draw_moveto:nn and others.)

__driver_draw_evenodd_rule: The even-odd rule here can be implemented as a simply switch.

```

\__driver_draw_nonzero_rule:
\g__driver_draw_eor_bool
19931 \cs_new_protected:Npn \__driver_draw_evenodd_rule:
19932 { \bool_gset_true:N \g__driver_draw_eor_bool }
19933 \cs_new_protected:Npn \__driver_draw_nonzero_rule:
19934 { \bool_gset_false:N \g__driver_draw_eor_bool }
19935 \bool_new:N \g__driver_draw_eor_bool

```

(End definition for __driver_draw_evenodd_rule:, __driver_draw_nonzero_rule:, and \g__driver_draw_eor_bool.)

__driver_draw_closepath: Unlike PDF, PostScript doesn't track separate colors for strokes and other elements. It is also desirable to have the clip keyword after a stroke or fill. To achieve those outcomes, there is some work to do. For color, if a stroke or fill color is defined it is used for the relevant operation, with a graphic scope inserted as required. That does mean that once such a color is set all further uses inside the same scope have to use scoping: see also **__driver_draw_fillstroke:** the color set up functions. For clipping, the required ordering is achieved using a T_EX switch. All of the operations end with a new path instruction as they do not terminate (again in contrast to PDF).

```

\__driver_draw_clip:
\g__driver_draw_clip_bool
19936 \cs_new_protected:Npn \__driver_draw_closepath:
19937 { \__driver_draw_literal:n { closepath } }
19938 \cs_new_protected:Npn \__driver_draw_stroke:
19939 {
19940   \__driver_draw_literal:n { currentdict~/l3sc-known~{gsave~l3sc}-if }
19941   \__driver_draw_literal:n { stroke }
19942   \__driver_draw_literal:n { currentdict~/l3sc-known~{grestore}-if }
19943   \bool_if:NT \g__driver_draw_clip_bool
19944   {
19945     \__driver_draw_literal:x
19946     {
19947       \bool_if:NT \g__driver_draw_eor_bool { eo }
19948       clip
19949     }
19950   }
19951   \__driver_draw_literal:n { newpath }
19952   \bool_gset_false:N \g__driver_draw_clip_bool
19953 }
19954 \cs_new_protected:Npn \__driver_draw_closestroke:
19955 {
19956   \__driver_draw_closepath:
19957   \__driver_draw_stroke:

```

```

19958     }
19959     \cs_new_protected:Npn \__driver_draw_fill:
19960     {
19961         \__driver_draw_literal:n { currentdict~/l3fc-known~{gsave~l3fc}~if }
19962         \__driver_draw_literal:x
19963         {
19964             \bool_if:NT \g__driver_draw_eor_bool { eo }
19965             fill
19966         }
19967         \__driver_draw_literal:n { currentdict~/l3fc-known~{grestore}~if }
19968         \bool_if:NT \g__driver_draw_clip_bool
19969         {
19970             \__driver_draw_literal:x
19971             {
19972                 \bool_if:NT \g__driver_draw_eor_bool { eo }
19973                 clip
19974             }
19975         }
19976         \__driver_draw_literal:n { newpath }
19977         \bool_gset_false:N \g__driver_draw_clip_bool
19978     }
19979     \cs_new_protected:Npn \__driver_draw_fillstroke:
19980     {
19981         \__driver_draw_literal:n { currentdict~/l3fc-known~{gsave~l3fc}~if }
19982         \__driver_draw_literal:x
19983         {
19984             \bool_if:NT \g__driver_draw_eor_bool { eo }
19985             fill
19986         }
19987         \__driver_draw_literal:n { currentdict~/l3fc-known~{grestore}~if }
19988         \__driver_draw_literal:n { currentdict~/l3sc-known~{gsave~l3sc}~if }
19989         \__driver_draw_literal:n { stroke }
19990         \__driver_draw_literal:n { currentdict~/l3sc-known~{grestore}~if }
19991         \bool_if:NT \g__driver_draw_clip_bool
19992         {
19993             \__driver_draw_literal:x
19994             {
19995                 \bool_if:NT \g__driver_draw_eor_bool { eo }
19996                 clip
19997             }
19998         }
19999         \__driver_draw_literal:n { newpath }
20000         \bool_gset_false:N \g__driver_draw_clip_bool
20001     }
20002     \cs_new_protected:Npn \__driver_draw_clip:
20003     { \bool_gset_true:N \g__driver_draw_clip_bool }
20004     \bool_new:N \g__driver_draw_clip_bool
20005     \cs_new_protected:Npn \__driver_draw_discardpath:
20006     {
20007         \bool_if:NT \g__driver_draw_clip_bool
20008         {
20009             \__driver_draw_literal:x
20010             {
20011                 \bool_if:NT \g__driver_draw_eor_bool { eo }

```

```

20012         clip
20013     }
20014 }
20015 \__driver_draw_literal:n { newpath }
20016 \bool_gset_false:N \g__driver_draw_clip_bool
20017 }

```

(End definition for __driver_draw_closepath: and others.)

__driver_draw_dash:nn Converting paths to output is again a case of mapping directly to PostScript operations.

```

\__driver_draw_dash:nn
\__driver_draw_dash:n
\__driver_draw_linewidth:n
\__driver_draw_miterlimit:n
\__driver_draw_cap_but:
\__driver_draw_cap_round:
\__driver_draw_cap_rectangle:
\__driver_draw_join_miter:
\__driver_draw_join_round:
\__driver_draw_join_bevel:
20018 \cs_new_protected:Npn \__driver_draw_dash:nn #1#2
20019 {
20020     \__driver_draw_literal:x
20021     {
20022         [ ~
20023         \clist_map_function:nN {#1} \__driver_draw_dash:n
20024         ] ~
20025         \dim_to_decimal_in_bp:n {#2} ~ setdash
20026     }
20027 }
20028 \cs_new:Npn \__driver_draw_dash:n #1
20029 { \dim_to_decimal_in_bp:n {#1} ~ }
20030 \cs_new_protected:Npn \__driver_draw_linewidth:n #1
20031 {
20032     \__driver_draw_literal:x
20033     { \dim_to_decimal_in_bp:n {#1} ~ setlinewidth }
20034 }
20035 \cs_new_protected:Npn \__driver_draw_miterlimit:n #1
20036 { \__driver_draw_literal:x { \fp_eval:n {#1} ~ setmiterlimit } }
20037 \cs_new_protected:Npn \__driver_draw_cap_but:
20038 { \__driver_draw_literal:n { 0 ~ setlinecap } }
20039 \cs_new_protected:Npn \__driver_draw_cap_round:
20040 { \__driver_draw_literal:n { 1 ~ setlinecap } }
20041 \cs_new_protected:Npn \__driver_draw_cap_rectangle:
20042 { \__driver_draw_literal:n { 2 ~ setlinecap } }
20043 \cs_new_protected:Npn \__driver_draw_join_miter:
20044 { \__driver_draw_literal:n { 0 ~ setlinejoin } }
20045 \cs_new_protected:Npn \__driver_draw_join_round:
20046 { \__driver_draw_literal:n { 1 ~ setlinejoin } }
20047 \cs_new_protected:Npn \__driver_draw_join_bevel:
20048 { \__driver_draw_literal:n { 2 ~ setlinejoin } }

```

(End definition for __driver_draw_dash:nn and others.)

__driver_draw_color_reset: To allow color to be defined for strokes and fills separately and to respect scoping, the data needs to be stored at the PostScript level. We cannot undefine (local) fill/stroke colors once set up but we can set them blank to improve performance slightly.

```

\__driver_draw_color_cmyk:nnnn
\__driver_draw_color_cmyk fill:nnnn
\__driver_draw_color_cmyk stroke:nnnn
\__driver_draw_color_gray:n
\__driver_draw_color_gray fill:n
\__driver_draw_color_gray stroke:n
\__driver_draw_color_rgb:nnn
\__driver_draw_color_rgb fill:nnn
\__driver_draw_color_rgb stroke:nnn
20049 \cs_new_protected:Npn \__driver_draw_color_reset:
20050 {
20051     \__driver_draw_literal:n { currentdic~/l3fc-known~{ /l3fc~ { } ~def }~if }
20052     \__driver_draw_literal:n { currentdic~/l3sc-known~{ /l3sc~ { } ~def }~if }
20053 }
20054 \cs_new_protected:Npn \__driver_draw_color_cmyk:nnnn #1#2#3#4
20055 {

```



```

20056     \__driver_draw_literal:x
20057     {
20058         \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
20059         \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
20060         setcmykcolor ~
20061     }
20062     \__driver_draw_color_reset:
20063 }
20064 \cs_new_protected:Npn \__driver_draw_color_cmyk_fill:nnnn #1#2#3#4
20065 {
20066     \__driver_draw_literal:x
20067     {
20068         /l3fc ~
20069         {
20070             \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
20071             \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
20072             setcmykcolor
20073         } ~
20074         def
20075     }
20076 }
20077 \cs_new_protected:Npn \__driver_draw_color_cmyk_stroke:nnnn #1#2#3#4
20078 {
20079     \__driver_draw_literal:x
20080     {
20081         /l3sc ~
20082         {
20083             \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
20084             \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
20085             setcmykcolor
20086         } ~
20087         def
20088     }
20089 }
20090 \cs_new_protected:Npn \__driver_draw_color_gray:n #1
20091 {
20092     \__driver_draw_literal:x { \fp_eval:n {#1} ~ setgray }
20093     \__driver_draw_color_reset:
20094 }
20095 \cs_new_protected:Npn \__driver_draw_color_gray_fill:n #1
20096 { \__driver_draw_literal:x { /l3fc ~ { \fp_eval:n {#1} ~ setgray } ~ def } }
20097 \cs_new_protected:Npn \__driver_draw_color_gray_stroke:n #1
20098 { \__driver_draw_literal:x { /l3sc ~ { \fp_eval:n {#1} ~ setgray } ~ def } }
20099 \cs_new_protected:Npn \__driver_draw_color_rgb:nnn #1#2#3
20100 {
20101     \__driver_draw_literal:x
20102     {
20103         \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
20104         setrgbcolor
20105     }
20106     \__driver_draw_color_reset:
20107 }
20108 \cs_new_protected:Npn \__driver_draw_color_rgb_fill:nnn #1#2#3
20109 {

```

```

20110 \__driver_draw_literal:x
20111 {
20112   /l3fc ~
20113   {
20114     \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
20115     setrgbcolor
20116   } ~
20117   def
20118 }
20119 }
20120 \cs_new_protected:Npn \__driver_draw_color_rgb_stroke:nnn #1#2#3
20121 {
20122   \__driver_draw_literal:x
20123   {
20124     /l3sc ~
20125     {
20126       \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
20127       setrgbcolor
20128     } ~
20129     def
20130   }
20131 }

```

(End definition for __driver_draw_color_reset: and others.)

__driver_draw_transformcm:nnnnnn The first four arguments here are floats (the affine matrix), the last two are a displacement vector. Once again, force evaluation to allow for caching.

```

20132 \cs_new_protected:Npn \__driver_draw_transformcm:nnnnnn #1#2#3#4#5#6
20133 {
20134   \__driver_draw_literal:x
20135   {
20136     [
20137       \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
20138       \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
20139       \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
20140     ] ~
20141     concat
20142   }
20143 }

```

(End definition for __driver_draw_transformcm:nnnnnn.)

__driver_draw_hbox:Nnnnnnn Inside a picture @beginspecial/@endspecial are active, which is normally a good thing but means that the position and scaling will be off if the box is inserted directly. Instead, we need to reverse the effect of the (normally desirable) shift/scaling within the box. That requires knowing where the reference point for the drawing is: saved as l3x/l3y at the start of the picture. Transformation here is relative to the drawing origin so has to be done purely in driver code not using T_EX offsets.

```

20144 \cs_new_protected:Npn \__driver_draw_hbox:Nnnnnnn #1#2#3#4#5#6#7
20145 {
20146   \__driver_scope_begin:
20147   \tex_special:D { ps:[end] }
20148   \__driver_draw_transformcm:nnnnnn {#2} {#3} {#4} {#5} {#6} {#7}
20149   \tex_special:D { ps::~72~Resolution~div~72~VResolution~div~neg~scale }

```

```

20150 \tex_special:D { ps::~magscale~{1~DVImag~div~dup~scale}~if }
20151 \tex_special:D { ps::~l3x~neg~l3y~neg~translate }
20152 \group_begin:
20153   \box_set_wd:Nn #1 { Opt }
20154   \box_set_ht:Nn #1 { Opt }
20155   \box_set_dp:Nn #1 { Opt }
20156   \box_use:N #1
20157 \group_end:
20158 \tex_special:D { ps::[begin] }
20159 \__driver_scope_end:
20160 }

```

(End definition for __driver_draw_hbox:Nnnnnnn.)

```
20161 </dvips>
```

38.9 dvisvgm driver

```
20162 <*dvisvgm>
```

38.9.1 Basics

__driver_literal:n Unlike the other drivers, the requirements for making SVG files mean that we can’t conveniently transform all operations to the current point. That makes life a bit more tricky later as that needs to be accounted for. A new line is added after each call to help to keep the output readable for debugging.

```

20163 \cs_new_protected:Npn \__driver_literal:n #1
20164 { \tex_special:D { dvisvgm:raw~ #1 { ?nl } } }

```

(End definition for __driver_literal:n.)

__driver_scope_begin: A scope in SVG terms is slightly different to the other drivers as operations have to be “tied” to these not simply inside them.

__driver_scope_end:

```

20165 \cs_new_protected:Npn \__driver_scope_begin:
20166 { \__driver_literal:n { <g> } }
20167 \cs_new_protected:Npn \__driver_scope_end:
20168 { \__driver_literal:n { </g> } }

```

(End definition for __driver_scope_begin: and __driver_scope_end:.)

38.10 Driver-specific auxiliaries

__driver_scope_begin:n In SVG transformations, clips and so on are attached directly to scopes so we need a way or allowing for that. This is rather more useful that **__driver_scope_begin:** as a result. No assumptions are made about the nature of the scoped operation(s).

```

20169 \cs_new_protected:Npn \__driver_scope_begin:n #1
20170 { \__driver_literal:n { <g~ #1 > } }

```

(End definition for __driver_scope_begin:n.)

38.10.1 Box operations

`__driver_box_use_clip:N`
`\g__driver_clip_path_int`

Clipping in SVG is more involved than with other drivers. The first issue is that the clipping path must be defined separately from where it is used, so we need to track how many paths have applied. The naming here uses `l3cp` as the namespace with a number following. Rather than use a rectangular operation, we define the path manually as this allows it to have a depth: easier than the alternative approach of shifting content up and down using scopes to allow for the depth of the `TEX` box and keep the reference point the same!

```

20171 \cs_new_protected:Npn \__driver_box_use_clip:N #1
20172 {
20173   \int_gincr:N \g__driver_clip_path_int
20174   \__driver_literal:n
20175   { < clipPath-id = " l3cp \int_use:N \g__driver_clip_path_int " > }
20176   \__driver_literal:n
20177   {
20178     <
20179     path ~ d =
20180     "
20181       M ~ 0 ~
20182       \dim_to_decimal:n { -\box_dp:N #1 } ~
20183       L ~ \dim_to_decimal:n { \box_wd:N #1 } ~
20184       \dim_to_decimal:n { -\box_dp:N #1 } ~
20185       L ~ \dim_to_decimal:n { \box_wd:N #1 } ~
20186       \dim_to_decimal:n { \box_ht:N #1 + \box_dp:N #1 } ~
20187       L ~ 0 ~
20188       \dim_to_decimal:n { \box_ht:N #1 + \box_dp:N #1 } ~
20189       Z
20190     "
20191     />
20192   }
20193   \__driver_literal:n
20194   { < /clipPath > }

```

In general the SVG set up does not try to transform coordinates to the current point. For clipping we need to do that, so have a transformation here to get us to the right place, and a matching one just before the `TEX` box is inserted to get things back on track. The clip path needs to come between those two such that if lines up with the current point, as does the `TEX` box.

```

20195 \__driver_scope_begin:n
20196 {
20197   transform =
20198   "
20199     translate ( { ?x } , { ?y } ) ~
20200     scale ( 1 , -1 )
20201   "
20202 }
20203 \__driver_scope__begin:n
20204 {
20205   clip-path = "url ( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int ) "
20206 }
20207 \__driver_scope_begin:n
20208 {
20209   transform =

```

```

20210         "
20211             scale ( -1 , 1 ) ~
20212             translate ( { ?x } , { ?y } ) ~
20213             scale ( -1 , -1 )
20214         "
20215     }
20216     \box_use:N #1
20217     \__driver_scope_end:
20218     \__driver_scope_end:
20219     \__driver_scope_end:
20220 %     \skip_horizontal:n { \box_wd:N #1 }
20221 }
20222 \int_new:N \g__driver_clip_path_int

```

(End definition for __driver_box_use_clip:N and \g__driver_clip_path_int.)

__driver_box_use_rotate:Nn Rotation has a dedicated operation which includes a centre-of-rotation optional pair. That can be picked up from the driver syntax, so there is no need to worry about the transformation matrix.

```

20223 \cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
20224 {
20225     \__driver_scope_begin:n
20226     {
20227         transform =
20228         "
20229             rotate
20230             ( \fp_eval:n { round ( -#2 , 5 ) } , ~ { ?x } , ~ { ?y } )
20231         "
20232     }
20233     \box_use:N #1
20234     \__driver_scope_end:
20235 }

```

(End definition for __driver_box_use_rotate:Nn.)

__driver_box_use_scale:Nnn In contrast to rotation, we have to account for the current position in this case. That is done using a couple of translations in addition to the scaling (which is therefore done backward with a flip).

```

20236 \cs_new_protected:Npn \__driver_box_use_scale:Nnn #1#2#3
20237 {
20238     \__driver_scope_begin:n
20239     {
20240         transform =
20241         "
20242             translate ( { ?x } , { ?y } ) ~
20243             scale
20244             (
20245                 \fp_eval:n { round ( -#2 , 5 ) } ,
20246                 \fp_eval:n { round ( -#3 , 5 ) }
20247             ) ~
20248             translate ( { ?x } , { ?y } ) ~
20249             scale ( -1 )
20250         "
20251     }

```

```

20252 \hbox_overlap_right:n { \box_use:N #1 }
20253 \__driver_scope_end:
20254 }

```

(End definition for __driver_box_use_scale:Nnn.)

38.10.2 Color

`\l__driver_current_color_tl` The current color in driver-dependent format: the same as for dvips.

```

20255 \tl_new:N \l__driver_current_color_tl
20256 \tl_set:Nn \l__driver_current_color_tl { gray~0 }
20257 \*package
20258 \AtBeginDocument
20259 {
20260 \ifpackageloaded { color }
20261 { \tl_set:Nn \l__driver_current_color_tl { \current@color } }
20262 { }
20263 }
20264 \*package

```

(End definition for \l__driver_current_color_tl.)

`__driver_color_ensure_current:` Directly set the color: same as dvips.

```

\__driver_color_reset:
20265 \cs_new_protected:Npn \__driver_color_ensure_current:
20266 {
20267 \tex_special:D { color~push~\l__driver_current_color_tl }
20268 \group_insert_after:N \__driver_color_reset:
20269 }
20270 \cs_new_protected:Npn \__driver_color_reset:
20271 { \tex_special:D { color~pop } }

```

(End definition for __driver_color_ensure_current: and __driver_color_reset:.)

38.11 Drawing

`__driver_draw_literal:n` The same as the more general literal call.

```

\__driver_draw_literal:x
20272 \cs_new_eq:NN \__driver_draw_literal:n \__driver_literal:n
20273 \cs_generate_variant:Nn \__driver_draw_literal:n { x }

```

(End definition for __driver_draw_literal:n.)

`__driver_draw_begin:` A drawing needs to be set up such that the co-ordinate system is translated. That is done inside a scope, which as described below

```

\__driver_draw_end:
20274 \cs_new_protected:Npn \__driver_draw_begin:
20275 {
20276 \__driver_draw_scope_begin:
20277 \__driver_draw_scope:n { transform="translate({?x},{?y})~scale(1,-1)" }
20278 }
20279 \cs_new_protected:Npn \__driver_draw_end:
20280 { \__driver_draw_scope_end: }

```

(End definition for __driver_draw_begin: and __driver_draw_end:.)

`__driver_draw_scope_begin:` Several settings that with other drivers are “stand alone” have to be given as part of
`__driver_draw_scope_end:` a scope in SVG. As a result, there is a need to provide a mechanism to automatically
`__driver_draw_scope:n` close these extra scopes. That is done using a dedicated function and a pair of tracking
`__driver_draw_scope:x` variables. Within each graphics scope we use a global variable to do the work, with a
`\g__driver_draw_scope_int` group used to save the value between scopes. The result is that no direct action is needed
`\l__driver_draw_scope_int` when creating a scope.

```

20281 \cs_new_protected:Npn \__driver_draw_scope_begin:
20282 {
20283   \int_set_eq:NN
20284     \l__driver_draw_scope_int
20285     \g__driver_draw_scope_int
20286   \group_begin:
20287     \int_gset:Nn \g__driver_draw_scope_int { 0 }
20288 }
20289 \cs_new_protected:Npn \__driver_draw_scope_end:
20290 {
20291   \prg_replicate:nn
20292     { \g__driver_draw_scope_int }
20293     { \__driver_draw_literal:n { </g> } }
20294   \group_end:
20295   \int_gset_eq:NN
20296     \g__driver_draw_scope_int
20297     \l__driver_draw_scope_int
20298 }
20299 \cs_new_protected:Npn \__driver_draw_scope:n #1
20300 {
20301   \__driver_draw_literal:n { <g~ #1 > }
20302   \int_gincr:N \g__driver_draw_scope_int
20303 }
20304 \cs_generate_variant:Nn \__driver_draw_scope:n { x }
20305 \int_new:N \g__driver_draw_scope_int
20306 \int_new:N \l__driver_draw_scope_int

```

(End definition for `__driver_draw_scope_begin:` and others.)

`__driver_draw_moveto:nn` Once again, some work is needed to get path constructs correct. Rather than write the
`__driver_draw_lineto:nn` values as they are given, the entire path needs to be collected up before being output in
`__driver_draw_rectangle:nnnn` one go. For that we use a dedicated storage routine, which will add spaces as required.
`__driver_draw_curveto:nnnnnn` Since paths should be fully expanded there is no need to worry about the internal x-type
`__driver_draw_add_to_path:n` expansion.
`\g__driver_draw_path_tl`

```

20307 \cs_new_protected:Npn \__driver_draw_moveto:nn #1#2
20308 {
20309   \__driver_draw_add_to_path:n
20310     { M ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} }
20311 }
20312 \cs_new_protected:Npn \__driver_draw_lineto:nn #1#2
20313 {
20314   \__driver_draw_add_to_path:n
20315     { L ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} }
20316 }
20317 \cs_new_protected:Npn \__driver_draw_rectangle:nnnn #1#2#3#4
20318 {
20319   \__driver_draw_add_to_path:n

```

```

20320     {
20321         M ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2}
20322         h ~ \dim_to_decimal:n {#3} ~
20323         v ~ \dim_to_decimal:n {#4} ~
20324         h ~ \dim_to_decimal:n { -#3 } ~
20325         Z
20326     }
20327 }
20328 \cs_new_protected:Npn \__driver_draw_curveto:nnnnnn #1#2#3#4#5#6
20329 {
20330     \__driver_draw_add_to_path:n
20331     {
20332         C ~
20333         \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} ~
20334         \dim_to_decimal:n {#3} ~ \dim_to_decimal:n {#4} ~
20335         \dim_to_decimal:n {#5} ~ \dim_to_decimal:n {#6}
20336     }
20337 }
20338 \cs_new_protected:Npn \__driver_draw_add_to_path:n #1
20339 {
20340     \tl_gset:Nx \g__driver_draw_path_tl
20341     {
20342         \g__driver_draw_path_tl
20343         \tl_if_empty:NF \g__driver_draw_path_tl { \c_space_tl }
20344         #1
20345     }
20346 }
20347 \tl_new:N \g__driver_draw_path_tl

```

(End definition for __driver_draw_moveto:nn and others.)

__driver_draw_evenodd_rule: The fill rules here have to be handled as scopes.

```

\__driver_draw_nonzero_rule:
20348 \cs_new_protected:Npn \__driver_draw_evenodd_rule:
20349 { \__driver_draw_scope:n { fill-rule="evenodd" } }
20350 \cs_new_protected:Npn \__driver_draw_nonzero_rule:
20351 { \__driver_draw_scope:n { fill-rule="nonzero" } }

```

(End definition for __driver_draw_evenodd_rule: and __driver_draw_nonzero_rule:.)

__driver_draw_path:n Setting fill and stroke effects and doing clipping all has to be done using scopes. This means setting up the various requirements in a shared auxiliary which deals with the bits and pieces. Clipping paths are reused for path drawing: not essential but avoids constructing them twice. Discarding a path needs a separate function as it's not quite the same.

```

\__driver_draw_closepath:
\__driver_draw_stroke:
\__driver_draw_closestroke:
\__driver_draw_fill:
\__driver_draw_fillstroke:
\__driver_draw_clip:
\__driver_draw_discardpath:
\g__driver_draw_clip_bool
\g__driver_draw_path_int
20352 \cs_new_protected:Npn \__driver_draw_closepath:
20353 { \__driver_draw_add_to_path:n { Z } }
20354 \cs_new_protected:Npn \__driver_draw_path:n #1
20355 {
20356     \bool_if:NTF \g__driver_draw_clip_bool
20357     {
20358         \int_gincr:N \g__driver_clip_path_int
20359         \__driver_draw_literal:x
20360         {
20361             < clipPath-id = " l3cp \int_use:N \g__driver_clip_path_int " >

```



```

20362         { ?nl }
20363         <path-d=" \g__driver_draw_path_tl "/> { ?nl }
20364         < /clipPath > { ? nl }
20365         <
20366             use-xlink:href =
20367                 "\c_hash_str l3path \int_use:N \g__driver_path_int " ~
20368                 #1
20369         />
20370     }
20371     \__driver_draw_scope:x
20372     {
20373         clip-path =
20374             "url( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int)"
20375     }
20376 }
20377 {
20378     \__driver_draw_literal:x
20379     { <path ~ d=" \g__driver_draw_path_tl " ~ #1 /> }
20380 }
20381 \tl_gclear:N \g__driver_draw_path_tl
20382 \bool_gset_false:N \g__driver_draw_clip_bool
20383 }
20384 \int_new:N \g__driver_path_int
20385 \cs_new_protected:Npn \__driver_draw_stroke:
20386 { \__driver_draw_path:n { style="fill:none" } }
20387 \cs_new_protected:Npn \__driver_draw_closestroke:
20388 {
20389     \__driver_draw_closepath:
20390     \__driver_draw_stroke:
20391 }
20392 \cs_new_protected:Npn \__driver_draw_fill:
20393 { \__driver_draw_path:n { style="stroke:none" } }
20394 \cs_new_protected:Npn \__driver_draw_fillstroke:
20395 { \__driver_draw_path:n { } }
20396 \cs_new_protected:Npn \__driver_draw_clip:
20397 { \bool_gset_true:N \g__driver_draw_clip_bool }
20398 \bool_new:N \g__driver_draw_clip_bool
20399 \cs_new_protected:Npn \__driver_draw_discardpath:
20400 {
20401     \bool_if:NT \g__driver_draw_clip_bool
20402     {
20403         \int_gincr:N \g__driver_clip_path_int
20404         \__driver_draw_literal:x
20405         {
20406             < clipPath-id = " l3cp \int_use:N \g__driver_clip_path_int " >
20407             { ?nl }
20408             <path-d=" \g__driver_draw_path_tl "/> { ?nl }
20409             < /clipPath >
20410         }
20411         \__driver_draw_scope:x
20412         {
20413             clip-path =
20414                 "url( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int)"
20415         }

```

```

20416     }
20417     \tl_gclear:N \g__driver_draw_path_tl
20418     \bool_gset_false:N \g__driver_draw_clip_bool
20419 }

```

(End definition for __driver_draw_path:n and others.)

__driver_draw_dash:nn All of these ideas are properties of scopes in SVG. The only slight complexity is converting the dash array properly (doing any required maths).

```

\__driver_draw_dash:nn
\__driver_draw_dash:nn
\__driver_draw_dash_aux:nn
\__driver_draw_linewidth:nn
\__driver_draw_miterlimit:nn
\__driver_draw_cap_but:nn
\__driver_draw_cap_round:nn
\__driver_draw_cap_rectangle:nn
\__driver_draw_join_miter:nn
\__driver_draw_join_round:nn
\__driver_draw_join_bevel:nn
20420 \cs_new_protected:Npn \__driver_draw_dash:nn #1#2
20421 {
20422     \use:x
20423     {
20424         \__driver_draw_dash_aux:nn
20425         { \clist_map_function:nn {#1} \__driver_draw_dash:n }
20426         { \dim_to_decimal:n {#2} }
20427     }
20428 }
20429 \cs_new:Npn \__driver_draw_dash:n #1
20430 { , \dim_to_decimal_in_bp:n {#1} }
20431 \cs_new_protected:Npn \__driver_draw_dash_aux:nn #1#2
20432 {
20433     \__driver_draw_scope:x
20434     {
20435         stroke-dasharray =
20436         "
20437         \tl_if_empty:oTF { \use_none:n #1 }
20438         { none }
20439         { \use_none:n #1 }
20440         " ~
20441         stroke-offset=" #2 "
20442     }
20443 }
20444 \cs_new_protected:Npn \__driver_draw_linewidth:n #1
20445 { \__driver_draw_scope:x { stroke-width=" \dim_to_decimal:n {#1} " } }
20446 \cs_new_protected:Npn \__driver_draw_miterlimit:n #1
20447 { \__driver_draw_scope:x { stroke-miterlimit=" \fp_eval:n {#1} " } }
20448 \cs_new_protected:Npn \__driver_draw_cap_but:nn
20449 { \__driver_draw_scope:n { stroke-linecap="butt" } }
20450 \cs_new_protected:Npn \__driver_draw_cap_round:nn
20451 { \__driver_draw_scope:n { stroke-linecap="round" } }
20452 \cs_new_protected:Npn \__driver_draw_cap_rectangle:nn
20453 { \__driver_draw_scope:n { stroke-linecap="square" } }
20454 \cs_new_protected:Npn \__driver_draw_join_miter:nn
20455 { \__driver_draw_scope:n { stroke-linejoin="miter" } }
20456 \cs_new_protected:Npn \__driver_draw_join_round:nn
20457 { \__driver_draw_scope:n { stroke-linejoin="round" } }
20458 \cs_new_protected:Npn \__driver_draw_join_bevel:nn
20459 { \__driver_draw_scope:n { stroke-linejoin="bevel" } }

```

(End definition for __driver_draw_dash:nn and others.)

_driver_draw_color_cmyk:nnnn SVG only works with RGB colors, so there is some conversion to do. The values also need to be given as percentages, which means a little more maths.

```

\_driver_draw_color_cmyk:nnnn
\_driver_draw_color_cmyk_fill:nnnn
\_driver_draw_color_cmyk_stroke:nnnn
\_driver_draw_color_gray:n
\_driver_draw_color_gray_fill:n
\_driver_draw_color_gray_stroke:n
\_driver_draw_color_rgb:nnn
\_driver_draw_color_rgb_fill:nnn
\_driver_draw_color_rgb_stroke:nnn

```

```

20460 \cs_new_protected:Npn \__driver_draw_color_cmyk_aux:NNnnnnn #1#2#3#4#5#6
20461 {
20462     \use:x
20463     {
20464         \__driver_draw_color_rgb_auxii:nnn
20465         { \fp_eval:n { -100 * ( (#3) * ( 1 - (#6) ) - 1 ) } }
20466         { \fp_eval:n { -100 * ( (#4) * ( 1 - (#6) ) + #6 - 1 ) } }
20467         { \fp_eval:n { -100 * ( (#5) * ( 1 - (#6) ) + #6 - 1 ) } }
20468     }
20469     #1 #2
20470 }
20471 \cs_new_protected:Npn \__driver_draw_color_cmyk:nnnn
20472 { \__driver_draw_color_cmyk_aux:NNnnnnn \c_true_bool \c_true_bool }
20473 \cs_new_protected:Npn \__driver_draw_color_cmyk_fill:nnnn
20474 { \__driver_draw_color_cmyk_aux:NNnnnnn \c_false_bool \c_true_bool }
20475 \cs_new_protected:Npn \__driver_draw_color_cmyk_stroke:nnnn
20476 { \__driver_draw_color_cmyk_aux:NNnnnnn \c_true_bool \c_false_bool }
20477 \cs_new_protected:Npn \__driver_draw_color_gray_aux:NNn #1#2#3
20478 {
20479     \use:x
20480     {
20481         \__driver_draw_color_gray_aux:nNN
20482         { \fp_eval:n { 100 * (#3) } }
20483     }
20484     #1 #2
20485 }
20486 \cs_new_protected:Npn \__driver_draw_color_gray_aux:nNN #1
20487 { \__driver_draw_color_rgb_auxii:nnnNN {#1} {#1} {#1} }
20488 \cs_generate_variant:Nn \__driver_draw_color_gray_aux:nNN { x }
20489 \cs_new_protected:Npn \__driver_draw_color_gray:n
20490 { \__driver_draw_color_gray_aux:NNn \c_true_bool \c_true_bool }
20491 \cs_new_protected:Npn \__driver_draw_color_gray_fill:n
20492 { \__driver_draw_color_gray_aux:NNn \c_false_bool \c_true_bool }
20493 \cs_new_protected:Npn \__driver_draw_color_gray_stroke:n
20494 { \__driver_draw_color_gray_aux:NNn \c_true_bool \c_false_bool }
20495 \cs_new_protected:Npn \__driver_draw_color_rgb_auxi:NNnnn #1#2#3#4#5
20496 {
20497     \use:x
20498     {
20499         \__driver_draw_color_rgb_auxii:nnnNN
20500         { \fp_eval:n { 100 * (#3) } }
20501         { \fp_eval:n { 100 * (#4) } }
20502         { \fp_eval:n { 100 * (#5) } }
20503     }
20504     #1 #2
20505 }
20506 \cs_new_protected:Npn \__driver_draw_color_rgb_auxii:nnnNN #1#2#3#4#5
20507 {
20508     \__driver_draw_scope:x
20509     {
20510         \bool_if:NT #4
20511         {
20512             fill =
20513             "

```

```

20514         rgb
20515         (
20516             #1 \c_percent_str ,
20517             #2 \c_percent_str ,
20518             #3 \c_percent_str
20519         )
20520         "
20521         \bool_if:NT #5 { ~ }
20522     }
20523     \bool_if:NT #5
20524     {
20525         stroke =
20526         "
20527         rgb
20528         (
20529             #1 \c_percent_str ,
20530             #2 \c_percent_str ,
20531             #3 \c_percent_str
20532         )
20533         "
20534     }
20535 }
20536 }
20537 \cs_new_protected:Npn \__driver_draw_color_rgb:nnn
20538 { \__driver_draw_color_rgb_auxi:NNnnn \c_true_bool \c_true_bool }
20539 \cs_new_protected:Npn \__driver_draw_color_rgb_fill:nnn
20540 { \__driver_draw_color_rgb_auxi:NNnnn \c_false_bool \c_true_bool }
20541 \cs_new_protected:Npn \__driver_draw_color_rgb_stroke:nnn
20542 { \__driver_draw_color_rgb_auxi:NNnnn \c_true_bool \c_false_bool }

```

(End definition for __driver_draw_color_cmyk:nnnn and others.)

__driver_draw_transformcm:nnnnnn The first four arguments here are floats (the affine matrix), the last two are a displacement vector. Once again, force evaluation to allow for caching.

```

20543 \cs_new_protected:Npn \__driver_draw_transformcm:nnnnnn #1#2#3#4#5#6
20544 {
20545     \__driver_draw_scope:x
20546     {
20547         transform =
20548         "
20549         matrix
20550         (
20551             \fp_eval:n {#1} , \fp_eval:n {#2} ,
20552             \fp_eval:n {#3} , \fp_eval:n {#4} ,
20553             \dim_to_decimal:n {#5} , \dim_to_decimal:n {#6}
20554         )
20555         "
20556     }
20557 }

```

(End definition for __driver_draw_transformcm:nnnnnn.)

__driver_draw_hbox:Nnnnnnn No special savings can be made here: simply displace the box inside a scope. As there is nothing to re-box, just make the box passed of zero size.

```

20558 \cs_new_protected:Npn \__driver_draw_hbox:Nnnnnnn #1#2#3#4#5#6#7
20559 {
20560   \__driver_scope_begin:
20561   \__driver_draw_transformcm:nnnnnn {#2} {#3} {#4} {#5} {#6} {#7}
20562   \__driver_literal:n
20563   {
20564     < g~
20565     stroke="none"~
20566     transform="scale(-1,1)~translate({?x},{?y})~scale(-1,-1)"
20567     >
20568   }
20569   \group_begin:
20570   \box_set_wd:Nn #1 { Opt }
20571   \box_set_ht:Nn #1 { Opt }
20572   \box_set_dp:Nn #1 { Opt }
20573   \box_use:N #1
20574   \group_end:
20575   \__driver_literal:n { </g> }
20576   \__driver_scope_end:
20577 }

(End definition for \__driver_draw_hbox:Nnnnnnn.)

20578 </dvisvgm>
20579 </initex | 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	
!	<i>192</i>
\!	<i>8940</i>
\"	<i>3015, 3018, 18976</i>
\#	<i>6, 3015, 6257, 6289, 11040</i>
\\$	<i>3015, 3018, 6256</i>
\%	<i>3015, 6258, 11042</i>
\&	<i>3015, 3018, 6249, 8940, 8941, 12902, 12976</i>
&&	<i>191</i>
\'	<i>18976</i>
\(<i>19146</i>
\)	<i>12845, 19146</i>
*	<i>192</i>
*	<i>3188, 3190, 3194, 3202, 5946, 12794</i>
**	<i>192</i>
+	<i>192, 192</i>
\+	<i>608, 608, 12900, 12906</i>
\,	<i>9644, 12861</i>
-	<i>192, 192</i>
\-	<i>205, 286, 12905</i>
\.	<i>18976</i>
/	<i>192</i>
\/	<i>285, 12904</i>
\:	<i>6255</i>
\::	<i>33, 279, 295, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2053, 2055, 2062, 2067, 2073, 2192, 2193, 2194, 2195, 2196, 2197, 2198, 2199, 2200, 2201, 2202, 2203, 2204, 2205, 2206, 2207, 2208, 2209, 2210, 2211, 2212, 2213, 2214, 2215, 2216, 2217, 2218, 2219, 2220, 2222, 2227, 2229, 2234, 2239, 2276, 2277, 2278, 2279, 2280</i>
\:N	<i>33, 2049, 2202, 2208, 2209, 2213, 2214, 2215, 2279</i>
\:V	<i>33, 2067, 2198</i>
\:V_unbraced	<i>2221</i>
\:c	<i>33, 2051, 2193, 2201, 2203, 2210, 2218, 2219</i>
\:error	<i>208, 17915</i>
\:f	<i>33, 2055, 2194, 2195, 2196, 2200, 2278</i>
\:f_unbraced	<i>2221</i>
\:n	<i>33, 2048, 2193, 2196, 2197, 2198, 2204, 2208, 2210, 2211, 2214, 2216, 2217, 2219, 2276, 2279</i>
\:o	<i>33, 2053, 2194, 2197, 2199, 2200, 2201, 2205, 2206, 2208, 2209, 2211, 2212, 2215, 2217, 2220, 2277</i>
\:o_unbraced	<i>2221, 2276, 2277, 2278, 2279</i>
\:p	<i>33, 280, 2050</i>
\:v	<i>33, 2067</i>
\:v_unbraced	<i>2221</i>
\:x	<i>33, 2062, 2192, 2202, 2203, 2204, 2205, 2206, 2207, 2213, 2214, 2215, 2216, 2217, 2218, 2219, 2220</i>
\:x_unbraced	<i>2221, 2280</i>
<	<i>192</i>
=	<i>192</i>
>	<i>192</i>
?	<i>192</i>
\?	<i>13003</i>
?:	<i>191</i>
\	<i>3015, 6251, 8779, 8780, 8781, 8782, 8875, 8893, 8895, 8900, 8901, 8915, 8916, 8919, 8920, 8997, 9087, 9095, 9297, 9305, 9312, 9324, 9325, 9350, 9351, 9358, 9379, 9381, 9382, 9413, 9414, 9415, 9416, 9417, 9610, 9614, 9619, 10497, 10498, 10504, 10523, 10536, 10542, 11045, 11178, 11185, 11654, 11657, 11658, 11659, 11666, 11669, 11670, 17062, 17065, 17090, 17091, 17098, 17099</i>
\{	<i>4, 3015, 6252, 8938, 9414, 9610, 9614, 9615, 11039, 19158</i>
\}	<i>5, 3015, 6253, 8939, 9414, 9610, 9614, 9615, 11041, 19158</i>
\~	<i>7, 10, 103, 194, 195, 196, 197, 1722, 2301, 2305, 3015, 3018, 3020, 3026, 3088, 5102, 6254, 8954, 10983, 11009, 12903, 15207, 18976</i>
^	<i>192</i>
\	<i>3015, 3018, 6260</i>
\'	<i>18976</i>
	<i>191</i>
\~	<i>3015, 3018, 5123, 6259, 11043, 18976</i>
_	<i>284, 1592, 3015, 5946, 6290, 8938, 8939, 9352, 9371, 9414, 9416, 9511, 9610, 9614, 9615, 9619, 9620, 11005, 11046</i>
A	
\A	<i>5947</i>
\AA	<i>18958</i>

- \aa 18958
 - \above 287
 - \abovedisplayshortskip 288
 - \abovedisplayskip 289
 - \abovewithdelims 290
 - abs 192
 - \accent 291
 - acos 194
 - acosd 194
 - acot 195
 - acotd 195
 - acsc 194
 - acscd 194
 - \adjdemerits 292
 - \adjustspacing 945
 - \advance 168, 184, 293
 - \AE 18959
 - \ae 18959
 - \afterassignment 294
 - \aftergroup 295
 - \alignmark 859
 - \alignat 860
 - asec 194
 - asecd 194
 - asin 194
 - asind 194
 - atan 195
 - atand 195
 - \AtBeginDocument ... 532, 8799, 10746,
19470, 19507, 19524, 19868, 20258
 - \atop 296
 - \atopwithdelims 297
 - \attribute 861
 - \attributedef 862
 - \autoscaling 1118
 - \autoxspacing 1119
- B**
- \badness 298
 - \baselineskip 299
 - \batchmode 300
 - \begin 13390, 19849, 19852
 - \begincsname 863
 - \begingroup 13,
20, 38, 42, 48, 73, 142, 162, 274, 301
 - \beginL 609
 - \beginR 610
 - \belowdisplayshortskip 302
 - \belowdisplayskip 303
 - \binoppenalty 304
 - \bodydir 930
 - bool commands:
 - \bool_do_until:Nn 38, 38, 2697
 - \bool_do_until:nn 39, 39, 2703
 - \bool_do_while:Nn 38, 38, 2697
 - \bool_do_while:nn 39, 39, 2703
 - .bool_gset:N 163, 10080
 - \bool_gset:Nn 37, 2521, 2564
 - \bool_gset_eq:NN 37, 2513, 2554
 - \bool_gset_false:N 36, 2501, 2544,
9579, 9594, 19636, 19934, 19952,
19977, 20000, 20016, 20382, 20418
 - .bool_gset_inverse:N 163, 10088
 - \bool_gset_true:N 36, 2501,
2539, 9544, 19634, 19932, 20003, 20397
 - \bool_if:NTF ... 37, 37, 238, 1924,
2571, 2580, 2581, 2692, 2694, 2698,
2700, 8264, 8666, 9408, 9564, 9591,
9859, 10052, 10061, 10282, 10310,
10329, 10331, 10336, 10343, 10367,
10413, 11108, 19647, 19652, 19657,
19943, 19947, 19964, 19968, 19972,
19984, 19991, 19995, 20007, 20011,
20356, 20401, 20510, 20521, 20523
 - \bool_if:nTF 37, 38,
38, 39, 39, 39, 39, 779, 795, 2591,
2601, 2705, 2714, 2718, 2727, 17923,
17929, 17930, 17938, 17944, 17946,
18031, 18448, 18524, 18618, 18668,
18778, 19102, 19126, 19257, 19285
 - \bool_if_exist:NTF
..... 37, 37, 2585, 2597, 9880, 9896
 - \bool_if_exist_p:N 37, 37, 2597
 - \bool_if_p:N 37, 37, 2571
 - \bool_if_p:n 38, 38, 303, 2522, 2524,
2562, 2567, 2601, 2609, 2684, 2687
 - \bool_lazy_all:nTF
..... 208, 208, 208, 208, 209, 17918
 - \bool_lazy_all_p:n .. 208, 208, 17918
 - \bool_lazy_and:nnTF
..... 208, 208, 209, 209, 209, 17927
 - \bool_lazy_and_p:nn
..... 208, 209, 209, 17927
 - \bool_lazy_any:nTF
..... 208, 209, 209, 209, 209, 17933
 - \bool_lazy_any_p:n 208, 209, 209, 17933
 - \bool_lazy_or:nnTF
..... 208, 209, 209, 209, 209, 17942
 - \bool_lazy_or_p:nn .. 209, 209, 17942
 - \bool_log:N 209, 209, 17948
 - \bool_log:n 209, 209, 17948
 - \bool_new:N ... 36, 36, 2499, 2593,
2594, 2595, 2596, 7976, 9542, 9774,
9775, 9778, 9779, 9783, 9880, 9896,
11004, 19637, 19935, 20004, 20398
 - \bool_not_p:n 38, 38, 2684
 - .bool_set:N 163, 10080
 - \bool_set:Nn 37, 37, 2521, 2559

- \bool_set_eq:NN . . . [37](#), [37](#), [2513](#), [2549](#)
- \bool_set_false:N [36](#), [36](#), [253](#), [2501](#), [2534](#),
[8260](#), [8664](#), [9800](#), [10229](#), [10249](#),
[10255](#), [10258](#), [10268](#), [10317](#), [11110](#)
- .bool_set_inverse:N [163](#), [10088](#)
- \bool_set_true:N [36](#),
[36](#), [267](#), [2501](#), [2529](#), [8278](#), [8293](#),
[8324](#), [9795](#), [10227](#), [10245](#), [10246](#),
[10254](#), [10263](#), [10324](#), [11062](#), [11149](#)
- \bool_show:N . [37](#), [37](#), [766](#), [2583](#), [17949](#)
- \bool_show:n [37](#), [37](#), [2583](#), [17951](#)
- \bool_until_do:Nn [38](#), [38](#), [2691](#)
- \bool_until_do:nn [39](#), [39](#), [2703](#)
- \bool_while_do:Nn [39](#), [39](#), [2691](#)
- \bool_while_do:nn [39](#), [39](#), [2703](#)
- \bool_xor_p:nn [38](#), [38](#), [2685](#)
- \c_false_bool [20](#), [36](#), [266](#), [290](#), [296](#), [300](#),
[300](#), [1455](#), [1495](#), [1538](#), [1543](#), [1587](#),
[1603](#), [1869](#), [1876](#), [2347](#), [2499](#), [2504](#),
[2508](#), [2537](#), [2547](#), [2657](#), [2659](#), [2664](#),
[2688](#), [19237](#), [19294](#), [19309](#), [20474](#),
[20476](#), [20492](#), [20494](#), [20540](#), [20542](#)
- \g_tmpa_bool [37](#), [2593](#)
- \l_tmpa_bool [37](#), [2593](#)
- \g_tmpb_bool [37](#), [2593](#)
- \l_tmpb_bool [37](#), [2593](#)
- \c_true_bool [20](#), [36](#), [266](#),
[266](#), [296](#), [300](#), [300](#), [380](#), [1495](#), [1587](#),
[1602](#), [1887](#), [2502](#), [2506](#), [2532](#), [2542](#),
[2573](#), [2658](#), [2660](#), [2666](#), [2689](#), [19244](#),
[19252](#), [19266](#), [19273](#), [19282](#), [19298](#),
[19305](#), [20472](#), [20474](#), [20476](#), [20490](#),
[20492](#), [20494](#), [20538](#), [20540](#), [20542](#)
- bool internal commands:
 - __bool_!:Nw [2639](#)
 - __bool_&_0:w [2663](#)
 - __bool_&_1:w [2661](#)
 - __bool_(:Nw [2641](#)
 - __bool_)_0:w [2657](#)
 - __bool_)_1:w [2657](#)
 - __bool_choose:NNN . . [2643](#), [2647](#), [2648](#)
 - __bool_eval_skip_to_end_auxi:Nw
. [2663](#)
 - __bool_eval_skip_to_end_-
auxii:Nw [2663](#)
 - __bool_eval_skip_to_end_-
auxiii:Nw [2663](#)
 - __bool_get_next:NN
. . [2627](#), [2629](#), [2640](#), [2644](#), [2661](#), [2662](#)
 - __bool_if_left_parentheses:wwwn
. [2609](#)
 - __bool_if_or:wwwn [2609](#)
 - __bool_if_parse:NNNww [299](#), [2616](#), [2625](#)
 - __bool_if_right_parentheses:wwwn
. [2609](#)
 - __bool_lazy_all:n [17918](#)
 - __bool_lazy_any:n [17933](#)
 - __bool_p:Nw [2646](#)
 - __bool_S_0:w [2657](#)
 - __bool_S_1:w [2657](#)
 - __bool_to_str:n [2583](#)
 - __bool_|_0:w [2661](#)
 - __bool_|_1:w [2663](#)
- \bookmark [305](#)
- \botmarks [611](#)
- \box [306](#)
- box commands:
 - \box_clear:N
[138](#), [138](#), [7753](#), [7760](#), [8012](#), [8076](#), [8126](#)
 - box_clear:N [138](#)
 - \box_clear_new:N [138](#), [138](#), [7759](#)
 - \box_clip:N . [203](#), [203](#), [203](#), [204](#), [17406](#)
 - \box_dp:N [139](#), [139](#),
[7781](#), [7788](#), [8147](#), [8148](#), [8218](#), [8220](#),
[8237](#), [8251](#), [8405](#), [8423](#), [8736](#), [12064](#),
[17142](#), [17262](#), [17355](#), [17417](#), [17424](#),
[17429](#), [17571](#), [19547](#), [19549](#), [19824](#),
[19826](#), [20182](#), [20184](#), [20186](#), [20188](#)
 - \box_gclear:N [138](#), [7753](#), [7762](#)
 - \box_gclear_new:N [138](#), [7759](#)
 - \box_gset_eq:NN [138](#), [7756](#), [7765](#)
 - \box_gset_eq_clear:NN [138](#), [138](#), [7771](#)
 - \box_gset_to_last:N [140](#), [7829](#)
 - \box_ht:N [139](#),
[139](#), [7781](#), [7790](#), [8071](#), [8121](#), [8149](#),
[8150](#), [8214](#), [8216](#), [8237](#), [8244](#), [8404](#),
[8422](#), [8734](#), [12063](#), [17141](#), [17261](#),
[17354](#), [17434](#), [17442](#), [17447](#), [17568](#),
[17570](#), [19549](#), [19826](#), [20186](#), [20188](#)
 - \box_if_empty:NTF
. [140](#), [140](#), [7823](#), [7826](#), [7827](#)
 - \box_if_empty_p:N [140](#), [140](#), [7823](#)
 - \box_if_exist:NTF
. . . . [138](#), [138](#), [7760](#), [7762](#), [7777](#), [7868](#)
 - \box_if_exist_p:N [138](#), [138](#), [7777](#)
 - \box_if_horizontal:NTF
. [140](#), [140](#), [7811](#), [7816](#), [7817](#)
 - \box_if_horizontal_p:N [140](#), [140](#), [7811](#)
 - \box_if_vertical:NTF
. [140](#), [140](#), [7811](#), [7820](#), [7821](#)
 - \box_if_vertical_p:N . [140](#), [140](#), [7811](#)
 - \box_log:N [141](#), [141](#), [7846](#)
 - \box_log:Nnn [141](#), [141](#), [7846](#)
 - \box_move_down:nn . [139](#), [751](#), [7800](#),
[17421](#), [17429](#), [17467](#), [17474](#), [17550](#)
 - \box_move_left:nn [139](#), [7800](#)

- \box_move_right:nn ... [139](#), [139](#), [7800](#)
- \box_move_up:nn ... [139](#), [139](#), [7800](#), [8443](#), [8731](#), [17438](#), [17447](#), [17481](#), [17494](#), [19782](#)
- \box_new:N ... [138](#), [138](#), [7745](#), [7835](#), [7836](#), [7837](#), [7838](#), [7839](#), [7954](#), [8019](#), [17126](#), [19790](#)
- \box_resize:Nnn [202](#), [202](#), [17241](#), [17681](#)
- \box_resize_to_ht:Nn [202](#), [202](#), [17281](#)
- \box_resize_to_ht_plus_dp:Nn ... [202](#), [202](#), [17281](#)
- \box_resize_to_wd:Nn [202](#), [202](#), [17281](#)
- \box_resize_to_wd_and_ht:Nnn ... [203](#), [203](#), [17281](#)
- \box_rotate:Nn [203](#), [203](#), [17127](#), [17543](#)
- \box_scale:Nnn [203](#), [203](#), [17347](#), [17703](#)
- \box_set_dp:Nn ... [140](#), [140](#), [751](#), [7787](#), [8405](#), [8423](#), [8735](#), [17168](#), [17381](#), [17384](#), [17424](#), [17432](#), [17470](#), [17475](#), [17555](#), [19787](#), [20155](#), [20572](#)
- \box_set_eq:NN ... [138](#), [138](#), [7754](#), [7765](#), [8136](#), [8425](#), [8739](#), [17452](#), [17499](#)
- \box_set_eq_clear:NN ... [138](#), [138](#), [7771](#)
- \box_set_ht:Nn ... [140](#), [140](#), [7787](#), [8404](#), [8422](#), [8733](#), [17167](#), [17380](#), [17385](#), [17441](#), [17450](#), [17484](#), [17497](#), [17553](#), [19786](#), [20154](#), [20571](#)
- \box_set_to_last:N ... [140](#), [140](#), [7829](#)
- \box_set_wd:Nn ... [140](#), [140](#), [7787](#), [8406](#), [8424](#), [8737](#), [17169](#), [17397](#), [17556](#), [19559](#), [19785](#), [20153](#), [20570](#)
- \box_show:N ... [141](#), [141](#), [7840](#)
- \box_show:Nnn ... [141](#), [141](#), [7840](#)
- \box_trim:Nnnnn ... [203](#), [203](#), [17409](#)
- \box_use:N ... [139](#), [139](#), [7796](#), [8440](#), [8443](#), [8521](#), [8658](#), [8728](#), [8731](#), [17156](#), [17171](#), [17392](#), [17401](#), [17414](#), [17422](#), [17430](#), [17439](#), [17448](#), [17460](#), [17468](#), [17474](#), [17482](#), [17495](#), [17551](#), [17558](#), [19552](#), [19577](#), [19591](#), [19782](#), [19788](#), [19830](#), [19844](#), [19862](#), [20156](#), [20216](#), [20233](#), [20252](#), [20573](#)
- \box_use_clear:N ... [139](#), [139](#), [7796](#)
- \box_viewport:Nnnnn ... [204](#), [204](#), [17455](#)
- \box_wd:N [140](#), [140](#), [7781](#), [7792](#), [8151](#), [8152](#), [8216](#), [8220](#), [8226](#), [8231](#), [8373](#), [8406](#), [8424](#), [8441](#), [8729](#), [8738](#), [12062](#), [17143](#), [17263](#), [17356](#), [17461](#), [17570](#), [17575](#), [17740](#), [17747](#), [19548](#), [19554](#), [19825](#), [19832](#), [20183](#), [20185](#), [20220](#)
- \c_empty_box ... [140](#), [141](#), [7754](#), [7756](#), [7835](#), [8518](#)
- \g_tmpa_box ... [141](#), [7836](#)
- \l_tmpa_box ... [141](#), [7836](#)
- \g_tmpb_box ... [141](#), [7836](#)
- \l_tmpb_box ... [141](#), [7836](#)
- \c_void_box ... [138](#)
- box internal commands:
 - \l__box_angle_fp ... [17115](#), [17132](#), [17133](#), [17134](#), [17164](#)
 - \l__box_bottom_dim ... [17118](#), [17142](#), [17199](#), [17203](#), [17208](#), [17214](#), [17219](#), [17223](#), [17232](#), [17234](#), [17252](#), [17262](#), [17271](#), [17307](#), [17355](#), [17361](#)
 - \l__box_bottom_new_dim ... [17122](#), [17168](#), [17200](#), [17211](#), [17222](#), [17233](#), [17270](#), [17360](#), [17381](#), [17385](#)
 - \l__box_cos_fp ... [17116](#), [17134](#), [17147](#), [17152](#), [17179](#), [17191](#)
 - \l__box_internal_box [17126](#), [17156](#), [17157](#), [17163](#), [17167](#), [17168](#), [17169](#), [17171](#), [17371](#), [17380](#), [17381](#), [17384](#), [17385](#), [17392](#), [17397](#), [17401](#), [17411](#), [17419](#), [17422](#), [17424](#), [17427](#), [17430](#), [17432](#), [17434](#), [17436](#), [17439](#), [17441](#), [17442](#), [17445](#), [17447](#), [17448](#), [17450](#), [17452](#), [17457](#), [17465](#), [17468](#), [17470](#), [17473](#), [17474](#), [17475](#), [17479](#), [17482](#), [17484](#), [17492](#), [17495](#), [17497](#), [17499](#)
 - \l__box_left_dim ... [17118](#), [17144](#), [17199](#), [17201](#), [17210](#), [17214](#), [17219](#), [17225](#), [17230](#), [17234](#), [17264](#), [17357](#)
 - \l__box_left_new_dim [17122](#), [17159](#), [17170](#), [17202](#), [17213](#), [17224](#), [17235](#)
 - __box_resize:N ... [17241](#), [17293](#), [17310](#), [17324](#), [17342](#)
 - __box_resize:NNN ... [17241](#)
 - __box_resize_common:N ... [17274](#), [17364](#), [17369](#)
 - __box_resize_set_corners:N ... [17241](#), [17286](#), [17303](#), [17320](#), [17334](#)
 - \l__box_right_dim .. [17118](#), [17143](#), [17197](#), [17203](#), [17208](#), [17212](#), [17221](#), [17223](#), [17232](#), [17236](#), [17248](#), [17263](#), [17269](#), [17322](#), [17336](#), [17356](#), [17363](#)
 - \l__box_right_new_dim ... [17122](#), [17170](#), [17204](#), [17215](#), [17226](#), [17237](#), [17268](#), [17362](#), [17389](#), [17391](#), [17397](#)
 - __box_rotate:N ... [17127](#)
 - __box_rotate_quadrant_four: ... [17127](#), [17228](#)
 - __box_rotate_quadrant_one: ... [17127](#), [17195](#)
 - __box_rotate_quadrant_three: ... [17127](#), [17217](#)
 - __box_rotate_quadrant_two: ... [17127](#), [17206](#)

- _box_rotate_x:nnN
 17127, 17173, 17201, 17203, 17212,
 17214, 17223, 17225, 17234, 17236
 - _box_rotate_y:nnN
 17127, 17184, 17197, 17199, 17208,
 17210, 17219, 17221, 17230, 17232
 - \l_box_scale_x_fp . 17239, 17247,
 17269, 17292, 17309, 17321, 17323,
 17335, 17352, 17363, 17375, 17387
 - \l_box_scale_y_fp
 17239, 17249, 17271, 17273, 17287,
 17292, 17304, 17309, 17323, 17337,
 17353, 17359, 17361, 17376, 17378
 - _box_show:NNnn 7844, 7854, 7861
 - \l_box_sin_fp
 .. 17116, 17133, 17145, 17180, 17190
 - \l_box_top_dim 17118, 17141, 17197,
 17201, 17210, 17212, 17221, 17225,
 17230, 17236, 17252, 17261, 17273,
 17290, 17307, 17340, 17354, 17359
 - \l_box_top_new_dim
 17122, 17167, 17198, 17209, 17220,
 17231, 17272, 17358, 17380, 17384
 - \boxdir 931
 - \boxmaxdepth 307
 - bp 197
 - \brokenpenalty 308
- C**
- \c 18976
 - \catcode 4, 5, 6, 7, 10, 212, 213, 214, 215,
 216, 217, 218, 219, 220, 225, 226,
 227, 228, 229, 230, 231, 232, 233, 309
 - catcode commands:
 - _c_catcode_active_tl
 51, 320, 320, 3201, 3261
 - _c_catcode_letter_token
 51, 320, 3183, 3251
 - _c_catcode_other_space_tl
 178, 11005, 11016, 11018, 11020, 11046
 - _c_catcode_other_token
 51, 320, 3183, 3256
 - \catcodetable 864
 - cc 197
 - ceil 193
 - \char 310, 3366
 - char commands:
 - _l_char_active_seq
 51, 173, 179, 3013, 10588
 - _char_generate:nn 48, 48,
 61, 3039, 9511, 11005, 18347, 18799,
 18800, 18812, 18813, 18933, 18934
 - _char_gset_active_eq:NN 47, 3019
 - _char_gset_active_eq:nN 47, 3019
 - _char_set_active_eq:NN
 47, 47, 3019, 10591
 - _char_set_active_eq:nN .. 47, 47, 3019
 - _char_set_catcode:nn 49,
 49, 242, 243, 244, 245, 246, 247,
 248, 249, 250, 2904, 2914, 2916,
 2918, 2920, 2922, 2924, 2926, 2928,
 2930, 2932, 2934, 2936, 2938, 2940,
 2942, 2944, 2946, 2948, 2950, 2952,
 2954, 2956, 2958, 2960, 2962, 2964,
 2966, 2968, 2970, 2972, 2974, 2976
 - _char_set_catcode_active:N
 48, 2913, 3020, 3088, 3202, 8941
 - _char_set_catcode_active:n
 49, 2945, 3144, 9643, 9644
 - _char_set_catcode_alignment:N ...
 48, 2913, 3190
 - _char_set_catcode_alignment:n ...
 49, 260, 2945, 3123
 - _char_set_catcode_comment:N 48, 2913
 - _char_set_catcode_comment:n 49, 2945
 - _char_set_catcode_end_line:N 48, 2913
 - _char_set_catcode_end_line:n 49, 2945
 - _char_set_catcode_escape:N . 48, 2913
 - _char_set_catcode_escape:n . 49, 2945
 - _char_set_catcode_group_begin:N .
 48, 2913
 - _char_set_catcode_group_begin:n .
 49, 2945, 3116
 - _char_set_catcode_group_end:N ...
 48, 2913
 - _char_set_catcode_group_end:n ...
 49, 2945, 3118
 - _char_set_catcode_ignore:N . 48, 2913
 - _char_set_catcode_ignore:n
 49, 257, 258, 2945
 - _char_set_catcode_invalid:N 48, 2913
 - _char_set_catcode_invalid:n 49, 2945
 - _char_set_catcode_letter:N
 48, 48, 2913, 12041,
 12593, 12594, 12794, 12845, 12861,
 12903, 12904, 12905, 12906, 12961,
 12975, 12976, 13003, 13490, 13491
 - _char_set_catcode_letter:n
 49, 49, 261, 263, 2945, 3140
 - _char_set_catcode_math_subscript:N
 48, 2913, 3194
 - _char_set_catcode_math_subscript:n
 49, 2945, 3135
 - _char_set_catcode_math_superscript:N
 48, 2913
 - _char_set_catcode_math_superscript:n
 49, 262, 2945, 3133

- \char_set_catcode_math_toggle:N [48](#), [2913](#), [3188](#)
- \char_set_catcode_math_toggle:n [49](#), [2945](#), [3121](#)
- \char_set_catcode_other:N [48](#), [2913](#), [12902](#)
- \char_set_catcode_other:n [49](#), [259](#), [264](#), [2945](#), [3091](#), [3142](#)
- \char_set_catcode_parameter:N [48](#), [2913](#)
- \char_set_catcode_parameter:n [49](#), [2945](#), [3131](#)
- \char_set_catcode_space:N [48](#), [2913](#)
- \char_set_catcode_space:n [49](#), [265](#), [2945](#), [3138](#)
- \char_set_lccode:nn [49](#), [49](#), [2977](#), [3026](#), [3148](#), [3149](#), [5108](#), [5125](#), [8938](#), [8939](#), [8940](#)
- \char_set_mathcode:nn [50](#), [50](#), [2977](#)
- \char_set_sfcode:nn [50](#), [50](#), [2977](#)
- \char_set_uccode:nn [50](#), [50](#), [2977](#)
- \char_show_value_catcode:n [49](#), [49](#), [2904](#)
- \char_show_value_lccode:n [50](#), [50](#), [2977](#)
- \char_show_value_mathcode:n [50](#), [50](#), [2977](#)
- \char_show_value_sfcode:n [51](#), [51](#), [2977](#)
- \char_show_value_uccode:n [50](#), [50](#), [2977](#)
- \l_char_special_seq [51](#), [3013](#)
- \char_value_catcode:n [49](#), [49](#), [242](#), [243](#), [244](#), [245](#), [246](#), [247](#), [248](#), [249](#), [250](#), [2904](#), [5120](#), [5150](#), [18349](#)
- \char_value_lccode:n [50](#), [50](#), [2977](#)
- \char_value_mathcode:n [50](#), [50](#), [2977](#)
- \char_value_sfcode:n [51](#), [51](#), [2977](#)
- \char_value_uccode:n [50](#), [50](#), [2977](#)
- char internal commands:
 - __char_generate:nn [61](#), [61](#), [3039](#), [18761](#), [18763](#), [18805](#), [18807](#), [18818](#), [18820](#)
 - __char_generate_aux:nn [3039](#)
 - __char_generate_aux:nnw [3039](#)
 - __char_generate_aux:w [3041](#), [3052](#)
 - __char_generate_invalid_catcode: [3039](#)
 - \c_char_max_int [3039](#)
 - __char_tmp:n [3146](#), [3158](#), [3161](#), [3163](#), [3166](#)
 - __char_tmp:nN [3021](#), [3032](#), [3033](#)
 - \l__char_tmp_tl [3039](#)
- \chardef [222](#), [235](#), [311](#)
- chk internal commands:
 - __chk_if_exist_cs:N [21](#), [21](#), [1569](#), [1574](#), [1579](#), [1584](#), [1774](#), [2309](#)
 - __chk_if_exist_var:N [22](#), [22](#), [1762](#), [2531](#), [2536](#), [2541](#), [2546](#), [2551](#), [2556](#), [2561](#), [2566](#), [5006](#), [5028](#), [5029](#), [5034](#), [5035](#), [5042](#), [5043](#), [5044](#), [5049](#), [5050](#), [5051](#)
 - __chk_if_free_cs:N [21](#), [21](#), [317](#), [482](#), [1739](#), [1788](#), [1836](#), [3184](#), [3186](#), [3196](#), [3729](#), [3748](#), [4472](#), [4724](#), [4817](#), [4883](#), [4889](#), [4894](#), [6415](#), [7480](#), [7748](#)
 - __chk_if_free_msg:nn [8826](#), [8849](#)
 - __chk_log:n [22](#), [22](#), [22](#), [1694](#), [1756](#), [2456](#), [8843](#), [9963](#)
 - __chk_resume_log: [22](#), [22](#), [22](#), [269](#), [269](#), [1694](#), [8027](#)
 - __chk_suspend_log: [22](#), [22](#), [22](#), [269](#), [269](#), [1694](#), [8020](#)
- choice commands:
 - .choice: [163](#), [10096](#)
- choices commands:
 - .choices:nn [163](#), [10098](#)
- \cite [19151](#)
- \cleaders [312](#)
- \clearmarks [865](#)
- clist commands:
 - \clist_clear:N [123](#), [123](#), [6941](#), [6958](#), [7154](#), [10219](#), [10237](#)
 - clist_clear:N [123](#)
 - \clist_clear_new:N [123](#), [123](#), [6945](#)
 - \clist_concat:NNN [124](#), [124](#), [6985](#), [7038](#), [7051](#)
 - \clist_const:Nn [123](#), [123](#), [6938](#)
 - \clist_count:N [128](#), [128](#), [130](#), [7333](#), [7362](#), [7394](#), [17518](#)
 - \clist_count:n [128](#), [7333](#), [7425](#), [17509](#)
 - \clist_gclear:N [123](#), [6941](#), [6960](#)
 - \clist_gclear_new:N [123](#), [6945](#)
 - \clist_gconcat:NNN [124](#), [6985](#), [7040](#), [7053](#)
 - \clist_get:NN [129](#), [129](#), [7063](#)
 - \clist_get:NNTF [129](#), [129](#), [7100](#), [7109](#), [7110](#)
 - \clist_gpop:NN [129](#), [129](#), [7074](#)
 - \clist_gpop:NNTF [130](#), [130](#), [7100](#), [7128](#), [7129](#)
 - \clist_gpush:Nn [130](#), [7131](#)
 - \clist_gput_left:Nn [124](#), [7037](#), [7139](#), [7140](#), [7141](#), [7142](#), [7143](#), [7144](#), [7145](#), [7146](#)
 - \clist_gput_right:Nn [125](#), [7050](#)
 - \clist_gremove_all:Nn [125](#), [7164](#)
 - \clist_gremove_duplicates:N [125](#), [7148](#)
 - \clist_greverse:N [125](#), [7195](#)
 - .clist_gset:N [163](#), [10108](#)
 - \clist_gset:Nn [124](#), [427](#), [7031](#)

- \clist_gset_eq:NN ... [124](#), [6949](#), [7151](#)
- \clist_gset_from_seq:NN ... [124](#), [6957](#)
- \clist_gsort:Nn ... [126](#), [7213](#), [16841](#)
- \clist_if_empty:NTF
 . [126](#), [126](#), [6994](#), [7181](#), [7213](#), [7258](#),
 [7288](#), [7307](#), [7455](#), [9984](#), [16849](#), [17517](#)
- \clist_if_empty:nTF
 [126](#), [126](#), [7217](#), [7461](#)
- \clist_if_empty_p:N .. [126](#), [126](#), [7213](#)
- \clist_if_empty_p:n .. [126](#), [126](#), [7217](#)
- \clist_if_exist:NTF [124](#),
 [124](#), [7000](#), [7360](#), [7455](#), [10704](#), [10731](#)
- \clist_if_exist_p:N .. [124](#), [124](#), [7000](#)
- \clist_if_in:NnTF [126](#),
 [126](#), [7157](#), [7231](#), [7247](#), [7248](#), [7249](#), [7250](#)
- \clist_if_in:nnTF
 [126](#), [7231](#), [7253](#), [7254](#), [11527](#)
- \clist_item:Nn
 [130](#), [130](#), [441](#), [753](#), [7391](#), [17518](#)
- \clist_item:nn .. [130](#), [753](#), [7422](#), [17513](#)
- \clist_log:N [204](#), [204](#), [17503](#)
- \clist_log:n [204](#), [204](#), [17503](#)
- \clist_map_break:
 [127](#), [127](#), [7262](#), [7267](#),
 [7276](#), [7280](#), [7295](#), [7313](#), [7329](#), [10382](#)
- \clist_map_break:n
 [128](#), [128](#), [7329](#), [10325](#), [16852](#)
- \clist_map_function:NN
 [42](#), [123](#), [127](#),
 [127](#), [127](#), [6442](#), [6452](#), [7256](#), [7338](#), [7456](#)
- \clist_map_function:Nn [438](#)
- \clist_map_function:nN
 [127](#), [439](#), [6447](#),
 [6457](#), [7272](#), [7463](#), [10423](#), [19666](#), [20023](#)
- \clist_map_function:nn [20425](#)
- \clist_map_inline:Nn
 [127](#), [127](#), [127](#), [437](#), [732](#), [7155](#), [7286](#),
 [10320](#), [10373](#), [10733](#), [10748](#), [16852](#)
- \clist_map_inline:nn [127](#),
 [7286](#), [9948](#), [10015](#), [19232](#), [19324](#), [19333](#)
- \clist_map_variable:NNn [127](#), [127](#), [7305](#)
- \clist_map_variable:nnN ... [127](#), [7305](#)
- \clist_new:N [123](#), [123](#), [123](#), [427](#), [6936](#),
 [7147](#), [7466](#), [7467](#), [7468](#), [7469](#), [9771](#)
- \clist_pop:NN [129](#), [129](#), [7074](#)
- \clist_pop:NNTF
 [130](#), [130](#), [7100](#), [7125](#), [7126](#)
- \clist_push:Nn [130](#), [130](#), [7131](#)
- \clist_put_left:Nn
 [124](#), [124](#), [7037](#), [7131](#), [7132](#),
 [7133](#), [7134](#), [7135](#), [7136](#), [7137](#), [7138](#)
- \clist_put_right:Nn
 [125](#), [125](#), [7050](#), [7158](#), [10410](#)
- \clist_rand_item:N .. [204](#), [204](#), [17508](#)
- \clist_rand_item:n .. [204](#), [204](#), [17508](#)
- \clist_remove_all:Nn .. [125](#), [125](#), [7164](#)
- \clist_remove_duplicates:N
 [125](#), [125](#), [7148](#)
- \clist_reverse:N [125](#), [125](#), [7195](#)
- \clist_reverse:n
 . [125](#), [125](#), [434](#), [434](#), [7196](#), [7198](#), [7201](#)
- .clist_set:N [163](#), [10108](#)
- \clist_set:Nn
 .. [124](#), [124](#), [427](#), [7031](#), [7038](#), [7040](#),
 [7051](#), [7053](#), [7237](#), [7301](#), [7318](#), [9983](#)
- \clist_set_eq:NN
 ... [124](#), [124](#), [6949](#), [7149](#), [9990](#), [10305](#)
- \clist_set_from_seq:NN [124](#), [124](#), [6957](#)
- \clist_show:N
 [130](#), [130](#), [204](#), [753](#), [7452](#), [17504](#)
- \clist_show:n [130](#), [130](#), [204](#), [7452](#), [17506](#)
- \clist_sort:Nn .. [126](#), [126](#), [7213](#), [16841](#)
- \clist_use:Nn [129](#), [129](#), [7358](#)
- \clist_use:Nnnn .. [128](#), [128](#), [425](#), [7358](#)
- \clist_wrap_item:n [732](#), [732](#), [732](#)
- \c_empty_clist
 ... [131](#), [6933](#), [7065](#), [7080](#), [7102](#), [7118](#)
- \l_foo_clist [200](#)
- \g_tmpa_clist [131](#), [7466](#)
- \l_tmpa_clist [131](#), [7466](#)
- \g_tmpb_clist [131](#), [7466](#)
- \l_tmpb_clist [131](#), [7466](#)
- clist internal commands:
- __clist_concat:NNNN [6985](#)
- __clist_count:n [7333](#)
- __clist_count:w [7333](#)
- __clist_get:wN [7063](#), [7105](#)
- __clist_if_empty:n:w [7217](#)
- __clist_if_empty:n:wNw [7217](#)
- __clist_if_in_return:nn [7231](#)
- \l__clist_internal_clist
 ... [6934](#), [7043](#), [7044](#), [7056](#), [7057](#),
 [7237](#), [7238](#), [7301](#), [7302](#), [7318](#), [7319](#)
- \l__clist_internal_remove_clist ..
 [7147](#), [7154](#), [7157](#), [7158](#), [7160](#)
- __clist_item:nnnN [7391](#), [7424](#)
- __clist_item:n:nw [7422](#)
- __clist_item_n_end:n [7422](#)
- __clist_item_N_loop:nw [7391](#)
- __clist_item_n_loop:nw [7422](#)
- __clist_item_n_strip:w [7422](#)
- __clist_map_function:Nw
 [437](#), [7256](#), [7292](#)
- __clist_map_function_n:Nn [437](#), [7272](#)
- __clist_map_unbrace:Nw ... [437](#), [7272](#)
- __clist_map_variable:Nnw [7305](#)
- __clist_pop:NNN [7074](#)
- __clist_pop:wN [7074](#)

- __clist_pop:wwNNN ... [431](#), [7074](#), [7121](#)
- __clist_pop_TF:NNN ... [7100](#)
- __clist_put_left:NNNn ... [7037](#)
- __clist_put_right:NNNn ... [7050](#)
- __clist_rand_item:nn ... [17508](#)
- __clist_remove_all: ... [7164](#)
- __clist_remove_all:NNn ... [7164](#)
- __clist_remove_all:w [433](#), [434](#), [7164](#)
- __clist_remove_duplicates:NN . [7148](#)
- __clist_reverse:wwNww ... [435](#), [435](#), [435](#), [435](#), [7201](#)
- __clist_reverse_end:ww ... [435](#), [7201](#)
- __clist_set_from_seq:NNNN ... [6957](#)
- __clist_set_from_seq:w ... [6957](#)
- __clist_tmp:w ... [434](#), [434](#), [434](#), [6935](#), [7170](#), [7191](#), [7242](#), [7244](#)
- __clist_trim_spaces:n ... [6939](#), [7010](#), [7032](#), [7034](#)
- __clist_trim_spaces:nn ... [430](#), [7010](#)
- __clist_trim_spaces_generic:nn . [429](#), [7004](#)
- __clist_trim_spaces_generic:nw . [437](#), [7004](#), [7012](#), [7022](#), [7027](#), [7274](#), [7282](#)
- __clist_use:nwn ... [7358](#)
- __clist_use:nwwwnwn ... [439](#), [7358](#)
- __clist_use:wn ... [7358](#)
- __clist_wrap_item:n ... [6957](#), [16855](#)
- \closein ... [313](#)
- \closeout ... [314](#)
- \clubpenalties ... [612](#)
- \clubpenalty ... [315](#)
- cm ... [197](#)
- code commands:
 - .code:n ... [163](#), [10106](#)
- coffin commands:
 - \coffin_attach:NnnNnnnn ... [147](#), [147](#), [478](#), [8400](#)
 - \coffin_attach_mark:NnnNnnnn ... [8400](#), [8588](#), [8609](#), [8625](#)
 - \coffin_clear:N ... [146](#), [146](#), [8008](#)
 - \coffin_display_handles:Nn ... [148](#), [148](#), [8631](#)
 - \coffin_dp:N ... [148](#), [148](#), [8147](#), [8748](#), [17679](#), [17705](#)
 - \coffin_ht:N ... [148](#), [148](#), [8147](#), [8747](#), [17679](#), [17705](#)
 - \coffin_if_exist:NTF ... [146](#), [146](#), [7985](#), [7996](#), [7997](#), [8001](#)
 - \coffin_if_exist_p:N . [146](#), [146](#), [7985](#)
 - \coffin_join:NnnNnnnn [148](#), [148](#), [8363](#)
 - \coffin_log_structure:N ... [205](#), [205](#), [17751](#)
 - \coffin_mark_handle:Nnnn ... [149](#), [149](#), [8576](#)
 - \coffin_new:N ... [146](#), [146](#), [464](#), [8017](#), [8141](#), [8143](#), [8144](#), [8145](#), [8146](#), [8524](#), [8525](#), [8526](#)
 - \coffin_resize:Nnn ... [204](#), [204](#), [17672](#)
 - \coffin_rotate:Nn ... [204](#), [204](#), [17530](#)
 - \coffin_scale:Nnn ... [204](#), [204](#), [17699](#)
 - \coffin_set_eq:NN ... [146](#), [146](#), [8132](#), [8397](#), [8416](#), [8445](#), [8652](#)
 - \coffin_set_horizontal_pole:Nnn . [147](#), [147](#), [8184](#)
 - \coffin_set_vertical_pole:Nnn ... [147](#), [147](#), [8184](#)
 - \coffin_show_structure:N ... [149](#), [149](#), [205](#), [760](#), [8741](#), [17752](#)
 - \coffin_typeset:Nnnnnn [148](#), [148](#), [8516](#)
 - \coffin_wd:N ... [148](#), [148](#), [8147](#), [8749](#), [17675](#), [17709](#)
 - \c_empty_coffin ... [149](#), [8141](#), [8519](#)
 - \l_tmpa_coffin ... [149](#), [8145](#)
 - \l_tmpb_coffin ... [149](#), [8145](#)
- coffin internal commands:
 - __coffin_align:NnnNnnnnN ... [8365](#), [8402](#), [8420](#), [8428](#), [8519](#)
 - \l__coffin_aligned_coffin ... [8141](#), [8366](#), [8367](#), [8371](#), [8377](#), [8379](#), [8380](#), [8396](#), [8397](#), [8403](#), [8404](#), [8405](#), [8406](#), [8407](#), [8409](#), [8411](#), [8415](#), [8416](#), [8421](#), [8422](#), [8423](#), [8424](#), [8425](#), [8459](#), [8474](#), [8520](#), [8521](#), [8726](#), [8733](#), [8735](#), [8737](#), [8739](#)
 - \l__coffin_aligned_internal_coffin ... [8141](#), [8438](#), [8445](#)
 - \l__coffin_bottom_corner_dim ... [17526](#), [17550](#), [17554](#), [17624](#), [17635](#), [17636](#), [17656](#), [17664](#)
 - \l__coffin_bounding_prop ... [17524](#), [17539](#), [17567](#), [17569](#), [17572](#), [17574](#), [17580](#), [17643](#)
 - \l__coffin_bounding_shift_dim ... [17525](#), [17548](#), [17642](#), [17648](#), [17649](#)
 - __coffin_calculate_intersection:Nnn ... [8256](#), [8430](#), [8433](#), [8719](#)
 - __coffin_calculate_intersection:nnnnnnnn ... [8256](#), [8665](#)
 - __coffin_calculate_intersection_aux:nnnnnN ... [8256](#)
 - \c__coffin_corners_prop ... [7957](#), [8024](#), [8166](#)
 - \l__coffin_cos_fp ... [754](#), [756](#), [17522](#), [17533](#), [17607](#), [17616](#)
 - __coffin_display_attach:Nnnnn [8631](#)
 - \l__coffin_display_coffin ... [8524](#), [8652](#), [8658](#), [8728](#), [8729](#), [8734](#), [8736](#), [8738](#), [8739](#)

\l__coffin_display_coord_coffin .
 [8524](#),
 [8590](#), [8610](#), [8626](#), [8673](#), [8693](#), [8712](#)
 \l__coffin_display_font_tl
 [8569](#), [8598](#), [8681](#)
 __coffin_display_handles_-
 aux:nnnn [8631](#)
 __coffin_display_handles_-
 aux:nnnnnn [8631](#)
 \l__coffin_display_handles_prop .
 [8527](#), [8601](#), [8605](#), [8684](#), [8688](#)
 \l__coffin_display_offset_dim ...
 [8564](#), [8627](#), [8628](#), [8713](#), [8714](#)
 \l__coffin_display_pole_coffin ..
 [8524](#), [8578](#), [8589](#), [8633](#), [8671](#)
 \l__coffin_display_poles_prop ...
 [8568](#),
 [8643](#), [8648](#), [8651](#), [8653](#), [8655](#), [8662](#)
 \l__coffin_display_x_dim
 [8566](#), [8668](#), [8723](#)
 \l__coffin_display_y_dim
 [8566](#), [8669](#), [8725](#)
 \l__coffin_error_bool [7976](#), [8260](#),
 [8264](#), [8278](#), [8293](#), [8324](#), [8664](#), [8666](#)
 __coffin_find_bounding_shift: ..
 [17542](#), [17640](#)
 __coffin_find_bounding_shift_-
 aux:nn [17640](#)
 __coffin_find_corner_maxima:N ..
 [17541](#), [17620](#)
 __coffin_find_corner_maxima_-
 aux:nn [17620](#)
 __coffin_get_pole:NnN
 [8153](#), [8258](#), [8259](#), [8483](#),
 [8484](#), [8487](#), [8488](#), [8645](#), [8646](#), [8649](#)
 __coffin_gset_eq_structure:NN [8170](#)
 __coffin_if_exist:NTF
 [7999](#), [8010](#), [8032](#), [8049](#),
 [8082](#), [8099](#), [8134](#), [8186](#), [8197](#), [8743](#)
 \l__coffin_internal_box ... [7954](#),
 [8065](#), [8071](#), [8076](#), [8115](#), [8121](#), [8126](#),
 [17544](#), [17553](#), [17555](#), [17556](#), [17558](#)
 \l__coffin_internal_dim
 [7954](#), [8372](#), [8374](#), [8375](#),
 [17571](#), [17573](#), [17575](#), [17704](#), [17707](#)
 \l__coffin_internal_tl
 [7954](#), [7963](#), [7964](#),
 [7965](#), [7966](#), [7967](#), [7968](#), [7969](#), [7970](#),
 [7971](#), [7972](#), [7973](#), [8457](#), [8458](#), [8460](#),
 [8602](#), [8603](#), [8606](#), [8607](#), [8615](#), [8620](#),
 [8685](#), [8686](#), [8689](#), [8690](#), [8699](#), [8704](#)
 \l__coffin_left_corner_dim
 [17526](#), [17548](#), [17557](#),
 [17625](#), [17631](#), [17632](#), [17655](#), [17663](#)
 __coffin_mark_handle_aux:nnnnNnn
 [8576](#)
 __coffin_offset_corner:Nnnnn . [8466](#)
 __coffin_offset_corners:Nnn ...
 [8385](#), [8386](#), [8392](#), [8393](#), [8466](#)
 __coffin_offset_pole:Nnnnnnn . [8447](#)
 __coffin_offset_poles:Nnn [8383](#),
 [8384](#), [8389](#), [8390](#), [8412](#), [8413](#), [8447](#)
 \l__coffin_offset_x_dim [7977](#), [8369](#),
 [8370](#), [8373](#), [8381](#), [8383](#), [8385](#), [8391](#),
 [8394](#), [8414](#), [8434](#), [8442](#), [8722](#), [8730](#)
 \l__coffin_offset_y_dim
 [7977](#), [8384](#), [8386](#), [8391](#),
 [8394](#), [8414](#), [8436](#), [8443](#), [8724](#), [8731](#)
 \l__coffin_pole_a_tl
 [7979](#), [8258](#), [8263](#), [8483](#),
 [8486](#), [8487](#), [8490](#), [8645](#), [8647](#), [8650](#)
 \l__coffin_pole_b_tl
 ... [7979](#), [8259](#), [8263](#), [8484](#), [8486](#),
 [8488](#), [8490](#), [8646](#), [8647](#), [8649](#), [8650](#)
 \c__coffin_poles_prop [7962](#), [8026](#), [8168](#)
 __coffin_reset_structure:N
 [8013](#), [8041](#),
 [8062](#), [8089](#), [8112](#), [8163](#), [8377](#), [8407](#)
 __coffin_resize_common:Nnn
 [17682](#), [17685](#), [17710](#)
 \l__coffin_right_corner_dim
 .. [17526](#), [17557](#), [17623](#), [17633](#), [17634](#)
 __coffin_rotate_bounding:nnn ...
 [17540](#), [17577](#)
 __coffin_rotate_corner:Nnnn ...
 [17535](#), [17577](#)
 __coffin_rotate_pole:Nnnnnn ...
 [17537](#), [17589](#)
 __coffin_rotate_vector:nnNN ...
 .. [17579](#), [17585](#), [17591](#), [17592](#), [17601](#)
 __coffin_scale_corner:Nnnn
 [17688](#), [17721](#)
 __coffin_scale_pole:Nnnnnn
 [17690](#), [17721](#)
 __coffin_scale_vector:nnNN
 [17714](#), [17723](#), [17729](#)
 \l__coffin_scale_x_fp [17668](#), [17674](#),
 [17691](#), [17701](#), [17703](#), [17709](#), [17717](#)
 \l__coffin_scale_y_fp ... [17668](#),
 [17676](#), [17702](#), [17703](#), [17707](#), [17719](#)
 \l__coffin_scaled_total_height_-
 dim [17670](#), [17706](#), [17711](#)
 \l__coffin_scaled_width_dim
 [17670](#), [17708](#), [17711](#)
 __coffin_set_bounding:N [17538](#), [17565](#)
 __coffin_set_eq_structure:NN ...
 [8137](#), [8170](#)

- `__coffin_set_pole:Nnn`
 8066, 8116, 8184, 8459,
 8496, 8500, 8508, 8512, 17594, 17730
- `__coffin_shift_corner:Nnnn`
 17560, 17651
- `__coffin_shift_pole:Nnnnnn`
 17562, 17651
- `\l__coffin_sin_fp`
 . 754, 756, 17522, 17532, 17608, 17615
- `\l__coffin_slope_x_fp`
 7974, 8318, 8323, 8331, 8337
- `\l__coffin_slope_y_fp`
 7974, 8320, 8323, 8332, 8337
- `\l__coffin_top_corner_dim`
 .. 17526, 17554, 17622, 17637, 17638
- `__coffin_update_B:nnnnnnnnN` . 8481
- `__coffin_update_corners:N`
 8043, 8064, 8091, 8114, 8211
- `__coffin_update_poles:N` .. 8042,
 8063, 8090, 8113, 8222, 8380, 8411
- `__coffin_update_T:nnnnnnnnN` . 8481
- `__coffin_update_vertical_-
 poles:NNN` 8396, 8415, 8481
- `\l__coffin_x_dim` . 7981, 8267, 8276,
 8296, 8299, 8306, 8315, 8326, 8341,
 8431, 8435, 8454, 8462, 8668, 8720,
 17579, 17581, 17585, 17587, 17591,
 17596, 17723, 17725, 17729, 17732
- `\l__coffin_x_prime_dim` 7981,
 8431, 8435, 8720, 8723, 17593, 17597
- `__coffin_x_shift_corner:Nnnn` ...
 17694, 17736
- `__coffin_x_shift_pole:Nnnnnn` ...
 17696, 17736
- `\l__coffin_y_dim` 7981, 8268,
 8281, 8284, 8291, 8308, 8313, 8342,
 8432, 8437, 8455, 8462, 8669, 8721,
 17579, 17581, 17585, 17587, 17591,
 17596, 17723, 17725, 17729, 17732
- `\l__coffin_y_prime_dim` 7981,
 8432, 8437, 8721, 8725, 17593, 17598
- `\color` 8584, 8596, 8639, 8679
- color commands:
 - `\color_ensure_current:`
 150, 150, 461, 8037, 8084, 8793
 - `\color_group_begin:` 150,
 150, 150, 8036, 8058, 8084, 8107, 8787
 - `\color_group_end:` 150,
 150, 150, 8039, 8060, 8087, 8110, 8787
- `\columnwidth` 8056, 8105
- `\copy` 316
- `\copyfont` 946
- `cos` 194
- `cosd` 194
- `cot` 194
- `cotd` 194
- `\count` 164, 166, 167, 168,
 172, 173, 175, 176, 179, 181, 182,
 183, 184, 188, 189, 191, 192, 317, 3374
- `\countdef` 318
- `\cr` 319
- `\crampeddisplaystyle` 866
- `\crampedscriptscriptstyle` 867
- `\crampedscriptstyle` 868
- `\crampedtextstyle` 869
- `\crrcr` 320
- `\cs` 13385
- cs commands:
 - `\cs:w` 16, 16, 16, 17, 1306,
 1326, 1328, 1376, 1638, 1666, 1914,
 1974, 2052, 2091, 2105, 2107, 2111,
 2112, 2113, 2148, 2154, 2174, 2176,
 2181, 2188, 2189, 2252, 2256, 2286,
 2491, 2738, 2740, 3730, 3819, 4473,
 4683, 4725, 4795, 4818, 7749, 10360,
 10379, 10389, 10403, 11326, 11351,
 12022, 12138, 12324, 12356, 12678,
 12785, 12817, 14829, 15098, 15614
 - `\cs_end:` 16, 16, 16, 295,
 1306, 1326, 1328, 1332, 1376, 1632,
 1638, 1660, 1666, 1846, 1914, 1974,
 2052, 2091, 2105, 2107, 2111, 2112,
 2113, 2148, 2154, 2174, 2176, 2181,
 2188, 2189, 2252, 2256, 2286, 2491,
 2735, 2741, 2743, 2745, 2747, 2749,
 2751, 2753, 2755, 2757, 2759, 2761,
 3730, 3819, 4473, 4683, 4725, 4795,
 4818, 7749, 10068, 10360, 10380,
 10390, 10403, 11326, 11359, 11513,
 12022, 12141, 12328, 12360, 12678,
 12785, 12820, 14829, 15098, 15773
 - `\cs_generate_from_arg_count:NNnn`
 14, 14, 1895, 1932
 - `\cs_generate_variant:Nn` 10, 24, 24,
 25, 25, 25, 26, 290, 291, 291, 291,
2307, 2500, 2509, 2510, 2511, 2512,
 2525, 2526, 2579, 2580, 2581, 2582,
 2592, 2695, 2696, 2701, 2702, 2831,
 2832, 2862, 2863, 2864, 2865, 2878,
 2879, 2880, 2881, 3035, 3036, 3037,
 3038, 3733, 3754, 3770, 3771, 3776,
 3777, 3779, 3780, 3782, 3783, 3796,
 3797, 3798, 3799, 3808, 3809, 3810,
 3811, 3815, 3816, 3818, 4417, 4476,
 4482, 4485, 4486, 4491, 4492, 4500,
 4501, 4503, 4504, 4506, 4507, 4511,
 4512, 4516, 4517, 4682, 4712, 4728,
 4734, 4737, 4738, 4743, 4744, 4752,

4753, 4755, 4756, 4758, 4759, 4763,
 4764, 4768, 4769, 4794, 4802, 4803,
 4805, 4821, 4827, 4831, 4832, 4837,
 4838, 4846, 4847, 4849, 4850, 4852,
 4853, 4857, 4858, 4862, 4863, 4867,
 4869, 4886, 4897, 4898, 4903, 4904,
 4909, 4910, 4923, 4924, 4941, 4942,
 4943, 4944, 4945, 4946, 4963, 4964,
 4965, 4966, 4967, 4968, 4969, 4970,
 4987, 4988, 4989, 4990, 4991, 4992,
 4993, 4994, 5097, 5098, 5099, 5100,
 5168, 5169, 5170, 5171, 5233, 5234,
 5239, 5240, 5243, 5244, 5245, 5246,
 5247, 5248, 5249, 5250, 5259, 5260,
 5261, 5262, 5272, 5273, 5274, 5275,
 5295, 5296, 5297, 5298, 5317, 5318,
 5319, 5328, 5329, 5330, 5373, 5374,
 5375, 5376, 5393, 5405, 5421, 5427,
 5436, 5448, 5449, 5472, 5473, 5571,
 5582, 5583, 5605, 5615, 5632, 5633,
 5634, 5635, 5752, 5758, 5778, 5793,
 5794, 5802, 5857, 5858, 5859, 5860,
 5861, 5862, 5863, 5864, 5875, 5876,
 5877, 5878, 5901, 5902, 5903, 5904,
 5964, 6022, 6099, 6118, 6156, 6171,
 6188, 6189, 6190, 6203, 6267, 6418,
 6421, 6424, 6427, 6430, 6459, 6460,
 6461, 6462, 6463, 6464, 6500, 6501,
 6506, 6507, 6529, 6530, 6531, 6532,
 6537, 6538, 6539, 6540, 6557, 6558,
 6583, 6584, 6601, 6602, 6611, 6612,
 6613, 6614, 6634, 6635, 6636, 6637,
 6638, 6639, 6668, 6681, 6682, 6697,
 6723, 6724, 6729, 6730, 6731, 6732,
 6733, 6734, 6743, 6744, 6745, 6746,
 6747, 6748, 6749, 6750, 6751, 6752,
 6753, 6754, 6778, 6798, 6827, 6838,
 6839, 6849, 6872, 6886, 6925, 6940,
 6981, 6982, 6983, 6984, 6998, 6999,
 7035, 7036, 7046, 7047, 7048, 7049,
 7059, 7060, 7061, 7062, 7073, 7098,
 7099, 7109, 7110, 7111, 7125, 7126,
 7127, 7128, 7129, 7130, 7162, 7163,
 7193, 7194, 7199, 7200, 7247, 7248,
 7249, 7250, 7251, 7252, 7253, 7254,
 7255, 7271, 7304, 7328, 7341, 7381,
 7390, 7414, 7421, 7465, 7483, 7486,
 7489, 7492, 7495, 7531, 7532, 7533,
 7534, 7541, 7542, 7561, 7562, 7563,
 7564, 7577, 7598, 7599, 7600, 7601,
 7602, 7603, 7617, 7619, 7621, 7623,
 7640, 7641, 7651, 7652, 7653, 7654,
 7677, 7678, 7679, 7680, 7681, 7682,
 7683, 7684, 7694, 7695, 7696, 7697,
 7698, 7699, 7714, 7715, 7730, 7741,
 7752, 7757, 7758, 7763, 7764, 7769,
 7770, 7775, 7776, 7784, 7785, 7786,
 7793, 7794, 7795, 7798, 7799, 7815,
 7816, 7817, 7818, 7819, 7820, 7821,
 7822, 7825, 7826, 7827, 7828, 7833,
 7834, 7842, 7845, 7848, 7860, 7880,
 7881, 7886, 7887, 7892, 7893, 7905,
 7906, 7916, 7917, 7922, 7923, 7931,
 7932, 7937, 7938, 7947, 7948, 7995,
 7996, 7997, 7998, 8016, 8029, 8046,
 8079, 8096, 8131, 8140, 8208, 8209,
 8210, 8399, 8427, 8523, 8630, 8716,
 8758, 9237, 9562, 9792, 9893, 9909,
 9966, 10079, 10212, 10213, 10216,
 10224, 10231, 10234, 10242, 10251,
 10260, 10287, 10426, 10697, 10714,
 10778, 10781, 10791, 10792, 10793,
 10824, 10842, 10917, 10924, 10941,
 10959, 10965, 10968, 10986, 11644,
 11647, 13183, 13381, 13426, 16242,
 16291, 16359, 16392, 16396, 16443,
 16641, 16648, 16649, 16650, 16653,
 16654, 16657, 16658, 16663, 16664,
 16671, 16672, 16673, 16674, 16682,
 16819, 16826, 16833, 16840, 16845,
 16846, 17113, 17258, 17297, 17314,
 17328, 17346, 17368, 17408, 17454,
 17501, 17507, 17520, 17564, 17684,
 17713, 17753, 17823, 17827, 17907,
 17908, 17909, 17910, 17952, 17963,
 17978, 17981, 18000, 18023, 18024,
 18047, 18053, 18135, 18136, 18160,
 18161, 18175, 18210, 19164, 19173,
 19595, 19884, 20273, 20304, 20488
 \cs_gset:Nn 14, 14, 1909, 1969
 \cs_gset:Npn 10, 12, 12, 1361,
 1794, 1808, 1810, 4031, 5397, 6802,
 7291, 7721, 8865, 8867, 8905, 12044
 \cs_gset:Npx
 12, 1361, 1795, 1808, 1811, 4038, 6807
 \cs_gset_eq:NN
 15, 15, 15, 1826, 1843, 1851, 2506,
 2508, 2517, 2518, 2519, 2520, 2542,
 2547, 2557, 3033, 4884, 4915, 4916,
 4917, 4918, 5036, 6416, 6812, 6817,
 7481, 7718, 7726, 10839, 10956,
 19241, 19242, 19243, 19244, 19249,
 19250, 19251, 19252, 19263, 19264,
 19265, 19266, 19270, 19271, 19272,
 19273, 19279, 19280, 19281, 19282
 \cs_gset_nopar:Nn . . 14, 14, 1909, 1969
 \cs_gset_nopar:Npn
 12, 12, 1361, 1792, 1800, 1804

- \cs_gset_nopar:Npx
 . [12](#), [1361](#), [1793](#), [1800](#), [1805](#), [4890](#),
 [4895](#), [4936](#), [4938](#), [4940](#), [4956](#), [4958](#),
 [4960](#), [4962](#), [4980](#), [4982](#), [4984](#), [4986](#)
- \cs_gset_protected:Nn
 [14](#), [14](#), [1909](#), [1969](#)
- \cs_gset_protected:Npn
 [12](#), [12](#), [1361](#), [1798](#), [1820](#),
 [1822](#), [8836](#), [13430](#), [16748](#), [16756](#), [16769](#)
- \cs_gset_protected:Npx
 [12](#), [1361](#), [1799](#), [1820](#), [1823](#)
- \cs_gset_protected_nopar:Nn
 [14](#), [14](#), [1909](#), [1969](#)
- \cs_gset_protected_nopar:Npn ...
 [12](#), [12](#), [1361](#), [1796](#), [1814](#), [1816](#)
- \cs_gset_protected_nopar:Npx ...
 [12](#), [1361](#), [1797](#), [1814](#), [1817](#)
- \cs_if_eq:NNTF [20](#),
 [20](#), [2001](#), [2008](#), [2009](#), [2012](#), [2013](#),
 [2016](#), [2017](#), [9078](#), [10035](#), [11817](#),
 [11827](#), [11853](#), [11855](#), [11857](#), [12049](#)
- \cs_if_eq_p:NN [20](#), [20](#), [2001](#)
- \cs_if_exist:N
 [20](#), [2597](#), [2599](#), [3784](#), [3786](#),
 [4493](#), [4495](#), [4745](#), [4747](#), [4839](#), [4841](#),
 [4925](#), [4926](#), [6508](#), [6510](#), [7000](#), [7002](#),
 [7642](#), [7644](#), [7777](#), [7779](#), [13239](#), [13240](#)
- \cs_if_exist:NTF
 [20](#), [20](#), [160](#), [278](#), [323](#), [489](#),
 [1618](#), [1675](#), [1677](#), [1679](#), [1681](#), [1683](#),
 [1685](#), [1687](#), [1689](#), [1766](#), [1776](#), [2020](#),
 [3093](#), [3094](#), [3096](#), [3341](#), [3358](#), [3756](#),
 [3757](#), [3759](#), [3760](#), [4413](#), [4452](#), [4453](#),
 [5821](#), [6222](#), [6230](#), [6276](#), [6279](#), [6292](#),
 [6384](#), [7987](#), [7989](#), [8801](#), [8807](#), [8823](#),
 [9806](#), [9916](#), [9962](#), [10303](#), [10345](#),
 [10357](#), [10370](#), [10376](#), [10387](#), [10401](#),
 [10444](#), [10452](#), [10564](#), [10644](#), [10663](#),
 [10766](#), [10770](#), [10855](#), [10887](#), [10905](#),
 [10909](#), [12042](#), [12187](#), [13230](#), [16479](#),
 [16490](#), [16746](#), [16764](#), [16767](#), [17830](#),
 [17881](#), [18311](#), [18340](#), [18366](#), [18377](#),
 [18384](#), [18497](#), [18723](#), [18747](#), [19093](#),
 [19239](#), [19247](#), [19255](#), [19277](#), [19443](#),
 [19450](#), [19456](#), [19462](#), [19480](#), [19489](#)
- \cs_if_exist_p:N
 [20](#), [20](#), [21](#), [1618](#), [19259](#), [19287](#)
- \cs_if_exist_use:N [16](#), [16](#), [1674](#), [10359](#)
- cs_if_exist_use:N [268](#)
- \cs_if_exist_use:NTF
 [16](#), [16](#), [1674](#), [1676](#),
 [1678](#), [1684](#), [1686](#), [10001](#), [11525](#),
 [12148](#), [18303](#), [18336](#), [19059](#), [19061](#)
- \cs_if_free:NTF [20](#), [20](#), [34](#),
 [489](#), [1646](#), [1741](#), [1751](#), [2448](#), [2487](#), [9109](#)
- \cs_if_free_p:N
 [19](#), [20](#), [20](#), [21](#), [22](#), [34](#), [1646](#)
- \cs_log:N [159](#), [201](#), [201](#), [742](#), [742](#), [17107](#)
- \cs_meaning:N
 .. [15](#), [15](#), [256](#), [1315](#), [1329](#), [1337](#), [2026](#)
- \cs_new:Nn [12](#), [12](#), [35](#), [1909](#), [1969](#)
- \cs_new:Npn [10](#), [11](#), [11](#), [14](#),
 [34](#), [34](#), [35](#), [737](#), [1784](#), [1808](#), [1812](#),
 [1883](#), [1885](#), [1893](#), [1940](#), [2006](#), [2007](#),
 [2008](#), [2009](#), [2010](#), [2011](#), [2012](#), [2013](#),
 [2014](#), [2015](#), [2016](#), [2017](#), [2031](#), [2040](#),
 [2041](#), [2045](#), [2046](#), [2047](#), [2048](#), [2049](#),
 [2050](#), [2051](#), [2053](#), [2055](#), [2067](#), [2073](#),
 [2079](#), [2090](#), [2092](#), [2099](#), [2100](#), [2102](#),
 [2104](#), [2106](#), [2108](#), [2115](#), [2117](#), [2122](#),
 [2127](#), [2133](#), [2139](#), [2145](#), [2151](#), [2157](#),
 [2164](#), [2171](#), [2178](#), [2185](#), [2193](#), [2194](#),
 [2195](#), [2196](#), [2197](#), [2198](#), [2199](#), [2200](#),
 [2201](#), [2208](#), [2209](#), [2210](#), [2211](#), [2212](#),
 [2221](#), [2222](#), [2227](#), [2229](#), [2234](#), [2244](#),
 [2246](#), [2248](#), [2249](#), [2251](#), [2253](#), [2259](#),
 [2265](#), [2267](#), [2274](#), [2276](#), [2277](#), [2278](#),
 [2279](#), [2281](#), [2283](#), [2285](#), [2286](#), [2287](#),
 [2289](#), [2294](#), [2303](#), [2304](#), [2374](#), [2390](#),
 [2395](#), [2404](#), [2417](#), [2432](#), [2489](#), [2590](#),
 [2609](#), [2619](#), [2621](#), [2623](#), [2625](#), [2629](#),
 [2639](#), [2641](#), [2646](#), [2648](#), [2657](#), [2658](#),
 [2659](#), [2660](#), [2661](#), [2662](#), [2663](#), [2665](#),
 [2668](#), [2674](#), [2680](#), [2684](#), [2685](#), [2691](#),
 [2693](#), [2697](#), [2699](#), [2703](#), [2711](#), [2716](#),
 [2724](#), [2730](#), [2737](#), [2739](#), [2741](#), [2742](#),
 [2744](#), [2746](#), [2748](#), [2750](#), [2752](#), [2754](#),
 [2756](#), [2758](#), [2760](#), [2762](#), [2767](#), [2768](#),
 [2769](#), [2770](#), [2771](#), [2772](#), [2773](#), [2774](#),
 [2775](#), [2776](#), [2785](#), [2787](#), [2801](#), [2807](#),
 [2815](#), [2822](#), [2829](#), [2833](#), [2839](#), [2871](#),
 [2877](#), [2899](#), [2909](#), [2982](#), [2991](#), [3000](#),
 [3009](#), [3039](#), [3045](#), [3052](#), [3098](#), [3106](#),
 [3168](#), [3288](#), [3320](#), [3393](#), [3405](#), [3406](#),
 [3414](#), [3423](#), [3432](#), [3445](#), [3446](#), [3447](#),
 [3448](#), [3498](#), [3506](#), [3508](#), [3510](#), [3520](#),
 [3530](#), [3616](#), [3619](#), [3628](#), [3637](#), [3659](#),
 [3662](#), [3668](#), [3686](#), [3694](#), [3702](#), [3715](#),
 [3717](#), [3724](#), [3819](#), [3826](#), [3840](#), [3845](#),
 [3851](#), [3862](#), [3867](#), [3874](#), [3876](#), [3878](#),
 [3880](#), [3882](#), [3884](#), [3886](#), [3896](#), [3901](#),
 [3906](#), [3911](#), [3916](#), [3918](#), [3941](#), [3949](#),
 [3957](#), [3963](#), [3969](#), [3977](#), [3985](#), [3991](#),
 [3997](#), [4004](#), [4018](#), [4053](#), [4067](#), [4073](#),
 [4105](#), [4137](#), [4139](#), [4141](#), [4147](#), [4153](#),
 [4165](#), [4173](#), [4185](#), [4193](#), [4226](#), [4259](#),
 [4261](#), [4263](#), [4265](#), [4267](#), [4272](#), [4277](#),

4282, 4287, 4288, 4289, 4290, 4291,
4292, 4293, 4294, 4295, 4296, 4297,
4298, 4299, 4300, 4301, 4302, 4303,
4312, 4313, 4322, 4328, 4330, 4339,
4346, 4352, 4354, 4356, 4372, 4383,
4406, 4518, 4523, 4525, 4533, 4541,
4549, 4551, 4563, 4569, 4584, 4586,
4588, 4590, 4592, 4594, 4599, 4604,
4609, 4614, 4616, 4623, 4631, 4639,
4645, 4651, 4659, 4667, 4673, 4679,
4683, 4684, 4691, 4699, 4701, 4703,
4788, 4791, 4795, 4797, 4800, 4864,
5095, 5276, 5331, 5332, 5333, 5334,
5344, 5345, 5350, 5355, 5360, 5365,
5367, 5377, 5380, 5386, 5388, 5422,
5424, 5426, 5428, 5437, 5442, 5447,
5450, 5457, 5464, 5466, 5476, 5488,
5496, 5502, 5508, 5512, 5519, 5530,
5539, 5541, 5548, 5554, 5556, 5558,
5572, 5574, 5576, 5584, 5589, 5594,
5606, 5607, 5608, 5616, 5662, 5671,
5702, 5723, 5730, 5738, 5744, 5751,
5820, 5834, 5842, 5879, 5884, 5889,
5894, 5899, 5905, 5911, 5916, 5921,
5926, 5931, 5933, 5940, 5951, 5960,
5963, 5965, 5973, 5975, 5982, 6003,
6013, 6015, 6020, 6021, 6023, 6031,
6033, 6041, 6047, 6053, 6072, 6074,
6083, 6089, 6095, 6097, 6100, 6110,
6117, 6119, 6127, 6132, 6137, 6148,
6155, 6157, 6163, 6165, 6170, 6172,
6178, 6179, 6184, 6185, 6186, 6187,
6191, 6196, 6201, 6204, 6205, 6213,
6218, 6232, 6246, 6404, 6492, 6498,
6528, 6541, 6596, 6632, 6666, 6718,
6755, 6757, 6765, 6771, 6779, 6781,
6783, 6792, 6840, 6848, 6850, 6873,
6874, 6875, 6882, 6884, 6973, 6980,
7004, 7009, 7010, 7017, 7097, 7190,
7192, 7201, 7208, 7211, 7224, 7230,
7256, 7265, 7272, 7278, 7285, 7329,
7331, 7333, 7351, 7358, 7382, 7383,
7386, 7388, 7391, 7399, 7415, 7422,
7430, 7432, 7446, 7451, 7474, 7518,
7565, 7571, 7662, 7668, 7700, 7706,
7731, 7733, 8904, 8990, 8991, 8992,
8993, 8994, 8995, 9082, 9107, 9238,
9499, 9508, 9513, 9522, 9527, 9532,
9537, 9607, 9608, 9612, 9617, 9747,
9752, 9754, 9874, 10399, 10417,
10422, 10424, 10427, 10435, 10440,
10625, 10991, 11029, 11200, 11201,
11202, 11203, 11204, 11205, 11206,
11207, 11208, 11209, 11229, 11231,
11233, 11239, 11245, 11254, 11256,
11268, 11269, 11271, 11281, 11291,
11301, 11311, 11321, 11324, 11334,
11338, 11343, 11345, 11347, 11349,
11361, 11363, 11365, 11367, 11391,
11393, 11395, 11396, 11397, 11399,
11401, 11403, 11405, 11415, 11425,
11433, 11448, 11449, 11460, 11474,
11481, 11483, 11490, 11509, 11631,
11632, 11633, 11634, 11635, 11636,
11637, 11642, 11645, 11689, 11691,
11700, 11701, 11710, 11723, 11736,
11743, 11757, 11773, 11785, 11795,
11804, 11815, 11825, 11851, 11862,
11871, 11882, 11887, 11907, 11909,
11920, 11925, 11938, 11961, 11962,
11972, 11978, 11995, 11996, 12020,
12029, 12034, 12057, 12086, 12089,
12093, 12095, 12101, 12108, 12115,
12123, 12146, 12157, 12177, 12185,
12200, 12215, 12226, 12236, 12246,
12251, 12260, 12277, 12290, 12295,
12301, 12303, 12310, 12340, 12368,
12384, 12395, 12400, 12418, 12436,
12447, 12462, 12467, 12477, 12487,
12497, 12513, 12560, 12567, 12584,
12595, 12607, 12623, 12635, 12646,
12665, 12673, 12682, 12691, 12693,
12695, 12697, 12699, 12701, 12703,
12705, 12707, 12709, 12711, 12713,
12721, 12723, 12726, 12728, 12730,
12732, 12749, 12751, 12761, 12764,
12770, 12780, 12781, 12788, 12795,
12826, 12841, 12842, 12846, 12862,
12877, 12879, 12888, 12909, 12933,
12935, 12949, 12962, 12977, 12989,
13004, 13018, 13035, 13040, 13045,
13050, 13060, 13065, 13076, 13091,
13102, 13115, 13134, 13152, 13154,
13170, 13181, 13184, 13194, 13205,
13208, 13246, 13265, 13270, 13297,
13298, 13306, 13318, 13324, 13330,
13338, 13346, 13352, 13358, 13366,
13374, 13391, 13410, 13440, 13451,
13475, 13477, 13479, 13481, 13492,
13499, 13501, 13502, 13527, 13533,
13540, 13541, 13549, 13560, 13562,
13569, 13571, 13579, 13586, 13609,
13611, 13621, 13641, 13650, 13664,
13672, 13680, 13687, 13694, 13702,
13712, 13726, 13737, 13738, 13744,
13761, 13768, 13770, 13777, 13782,
13799, 13800, 13801, 13819, 13825,
13835, 13847, 13854, 13868, 13876,

13914, 13923, 13942, 13944, 13946,
 13955, 13966, 13978, 13993, 14006,
 14019, 14027, 14044, 14061, 14068,
 14076, 14086, 14087, 14096, 14097,
 14106, 14116, 14130, 14140, 14151,
 14159, 14161, 14172, 14178, 14213,
 14234, 14236, 14238, 14240, 14247,
 14256, 14261, 14268, 14275, 14295,
 14300, 14317, 14332, 14333, 14338,
 14346, 14347, 14370, 14383, 14390,
 14398, 14399, 14400, 14401, 14402,
 14403, 14411, 14417, 14419, 14421,
 14443, 14448, 14458, 14468, 14478,
 14491, 14502, 14507, 14514, 14523,
 14525, 14534, 14543, 14557, 14559,
 14561, 14574, 14584, 14589, 14598,
 14606, 14613, 14619, 14628, 14630,
 14642, 14647, 14655, 14660, 14670,
 14676, 14682, 14689, 14696, 14698,
 14703, 14705, 14710, 14711, 14725,
 14735, 14747, 14752, 14759, 14778,
 14792, 14806, 14826, 14839, 14841,
 14846, 14859, 14864, 14872, 14877,
 14887, 14899, 14928, 14929, 14930,
 14932, 14934, 14936, 14950, 14956,
 14965, 14984, 14990, 15000, 15019,
 15027, 15066, 15068, 15077, 15079,
 15093, 15095, 15104, 15106, 15122,
 15138, 15154, 15170, 15186, 15202,
 15207, 15232, 15252, 15281, 15297,
 15307, 15318, 15339, 15354, 15359,
 15364, 15366, 15380, 15382, 15397,
 15405, 15413, 15437, 15452, 15467,
 15482, 15497, 15512, 15527, 15542,
 15550, 15564, 15566, 15572, 15584,
 15592, 15599, 15775, 15783, 15794,
 15795, 15797, 15798, 15809, 15816,
 15818, 15824, 15835, 15845, 15852,
 15859, 15874, 15913, 15926, 15957,
 15962, 15967, 15981, 16000, 16002,
 16019, 16034, 16046, 16053, 16058,
 16060, 16069, 16082, 16085, 16106,
 16119, 16134, 16152, 16167, 16177,
 16186, 16199, 16215, 16232, 16237,
 16238, 16239, 16240, 16243, 16248,
 16275, 16287, 16289, 16292, 16297,
 16323, 16348, 16357, 16358, 16360,
 16365, 16375, 16390, 16393, 16395,
 16397, 16402, 16407, 16414, 16428,
 16433, 16435, 16445, 16447, 16449,
 16451, 16483, 16492, 16497, 16503,
 16509, 16526, 16531, 16541, 16551,
 16553, 16566, 16579, 16593, 16602,
 16607, 16616, 16618, 16625, 16860,
 16862, 16964, 16977, 16982, 16988,
 16989, 16995, 17001, 17007, 17013,
 17019, 17020, 17022, 17029, 17035,
 17508, 17510, 17515, 17782, 17784,
 17832, 17838, 17857, 17862, 17867,
 17875, 17890, 17899, 17901, 17903,
 17905, 17911, 17920, 17935, 17954,
 17962, 17964, 17970, 17982, 17983,
 17984, 17993, 18002, 18004, 18010,
 18016, 18048, 18055, 18091, 18104,
 18110, 18115, 18127, 18128, 18129,
 18176, 18177, 18178, 18179, 18180,
 18181, 18182, 18190, 18197, 18208,
 18211, 18217, 18229, 18234, 18242,
 18256, 18261, 18272, 18283, 18289,
 18294, 18301, 18313, 18317, 18332,
 18342, 18357, 18358, 18360, 18362,
 18364, 18375, 18401, 18412, 18421,
 18435, 18441, 18444, 18459, 18465,
 18467, 18477, 18483, 18499, 18509,
 18516, 18538, 18548, 18561, 18569,
 18575, 18595, 18604, 18610, 18632,
 18645, 18654, 18660, 18678, 18684,
 18689, 18719, 18721, 18977, 18985,
 18992, 19003, 19015, 19020, 19028,
 19042, 19055, 19057, 19067, 19073,
 19087, 19091, 19100, 19112, 19118,
 19167, 19172, 19315, 19316, 19320,
 19321, 19671, 19808, 20028, 20429
 \cs_new:Npx 11, 289, 289, 1784, 1808,
 1813, 2328, 2471, 7342, 7352, 11023,
 11964, 12071, 13581, 15611, 16463
 \cs_new_eq:NN 15,
 15, 15, 270, 273, 317, 317, 1523,
 1826, 2030, 2039, 2061, 2299, 2300,
 2496, 2497, 2499, 2513, 2514, 2515,
 2516, 2517, 2518, 2519, 2520, 2895,
 3182, 3189, 3191, 3192, 3193, 3195,
 3198, 3199, 3441, 3442, 3443, 3649,
 3650, 3651, 3652, 3653, 3761, 3762,
 3765, 3817, 3924, 4052, 4437, 4466,
 4467, 4468, 4622, 4681, 4711, 4793,
 4796, 4799, 4804, 4866, 4868, 4911,
 4912, 4913, 4914, 4915, 4916, 4917,
 4918, 5227, 5228, 5379, 5777, 5791,
 5792, 5939, 6243, 6265, 6266, 6411,
 6431, 6432, 6433, 6434, 6435, 6436,
 6437, 6438, 6887, 6888, 6889, 6890,
 6891, 6892, 6893, 6894, 6895, 6896,
 6897, 6898, 6899, 6900, 6901, 6902,
 6903, 6904, 6905, 6906, 6907, 6908,
 6909, 6910, 6911, 6912, 6933, 6936,
 6937, 6941, 6942, 6943, 6944, 6945,
 6946, 6947, 6948, 6949, 6950, 6951,

- 6952, 6953, 6954, 6955, 6956, 7131,
 7132, 7133, 7134, 7135, 7136, 7137,
 7138, 7139, 7140, 7141, 7142, 7143,
 7144, 7145, 7146, 7496, 7497, 7498,
 7499, 7500, 7501, 7502, 7503, 7781,
 7782, 7783, 7796, 7797, 7808, 7809,
 7810, 7894, 7895, 7903, 7904, 7944,
 7945, 7946, 8147, 8148, 8149, 8150,
 8151, 8152, 8787, 10753, 10777,
 10852, 10882, 10916, 10962, 10992,
 11077, 11196, 11330, 11519, 11520,
 11521, 11522, 11756, 11784, 11792,
 11793, 11794, 11802, 11803, 12559,
 16442, 16444, 16488, 16640, 16651,
 16652, 17102, 17103, 18068, 18069,
 18072, 18073, 18076, 18077, 18354,
 18496, 18567, 18568, 19234, 19235,
 19236, 19237, 19291, 19292, 19293,
 19294, 19295, 19296, 19297, 19298,
 19302, 19303, 19304, 19305, 19306,
 19307, 19308, 19309, 19594, 20272
 \cs_new_nopar:Nn . . . 13, 13, 1909, 1969
 \cs_new_nopar:Npn 11,
 11, 270, 272, 1784, 1800, 1806, 2029
 \cs_new_nopar:Npx 11, 1784, 1800, 1807
 \cs_new_protected:Nn 13, 13, 1909, 1969
 \cs_new_protected:Npn 11,
 11, 1784, 1820, 1824, 1826, 1827,
 1828, 1829, 1830, 1831, 1832, 1833,
 1834, 1839, 1840, 1841, 1842, 1844,
 1895, 1905, 1907, 1917, 1922, 2018,
 2023, 2025, 2027, 2062, 2192, 2202,
 2203, 2204, 2205, 2206, 2207, 2213,
 2214, 2215, 2216, 2217, 2218, 2219,
 2220, 2239, 2280, 2302, 2307, 2334,
 2340, 2345, 2354, 2444, 2481, 2499,
 2501, 2503, 2505, 2507, 2521, 2523,
 2583, 2588, 2794, 2886, 2904, 2911,
 2913, 2915, 2917, 2919, 2921, 2923,
 2925, 2927, 2929, 2931, 2933, 2935,
 2937, 2939, 2941, 2943, 2945, 2947,
 2949, 2951, 2953, 2955, 2957, 2959,
 2961, 2963, 2965, 2967, 2969, 2971,
 2973, 2975, 2977, 2984, 2986, 2993,
 2995, 3002, 3004, 3011, 3023, 3182,
 3449, 3451, 3453, 3459, 3476, 3478,
 3480, 3494, 3496, 3539, 3727, 3734,
 3768, 3769, 3772, 3774, 3778, 3781,
 3788, 3790, 3792, 3794, 3800, 3802,
 3804, 3806, 3812, 3814, 3820, 4027,
 4034, 4046, 4408, 4418, 4470, 4477,
 4483, 4484, 4487, 4489, 4497, 4499,
 4502, 4505, 4508, 4510, 4513, 4515,
 4713, 4722, 4729, 4735, 4736, 4739,
 4741, 4749, 4751, 4754, 4757, 4760,
 4762, 4765, 4767, 4806, 4815, 4822,
 4828, 4830, 4833, 4835, 4843, 4845,
 4848, 4851, 4854, 4856, 4859, 4861,
 4870, 4881, 4887, 4892, 4899, 4901,
 4905, 4907, 4919, 4921, 4929, 4931,
 4933, 4935, 4937, 4939, 4947, 4949,
 4951, 4953, 4955, 4957, 4959, 4961,
 4971, 4973, 4975, 4977, 4979, 4981,
 4983, 4985, 5057, 5059, 5061, 5063,
 5085, 5103, 5115, 5117, 5131, 5160,
 5162, 5164, 5166, 5172, 5194, 5200,
 5229, 5231, 5235, 5237, 5314, 5315,
 5316, 5394, 5403, 5406, 5412, 5414,
 5468, 5470, 5578, 5580, 5753, 5759,
 5765, 5767, 6241, 6242, 6413, 6419,
 6422, 6425, 6428, 6439, 6444, 6449,
 6454, 6465, 6467, 6469, 6502, 6504,
 6512, 6520, 6533, 6535, 6543, 6545,
 6547, 6559, 6561, 6563, 6585, 6587,
 6589, 6640, 6648, 6658, 6669, 6671,
 6673, 6675, 6683, 6692, 6698, 6700,
 6702, 6799, 6804, 6809, 6815, 6821,
 6828, 6919, 6935, 6938, 6957, 6959,
 6961, 6985, 6987, 6989, 7031, 7033,
 7037, 7039, 7041, 7050, 7052, 7054,
 7063, 7071, 7074, 7076, 7078, 7086,
 7116, 7148, 7150, 7152, 7164, 7166,
 7168, 7195, 7197, 7240, 7286, 7299,
 7305, 7316, 7321, 7452, 7458, 7478,
 7484, 7487, 7490, 7493, 7508, 7510,
 7519, 7525, 7535, 7543, 7552, 7604,
 7605, 7606, 7625, 7627, 7629, 7716,
 7735, 7746, 7753, 7755, 7759, 7761,
 7765, 7767, 7771, 7773, 7787, 7789,
 7791, 7800, 7802, 7804, 7806, 7829,
 7831, 7840, 7843, 7846, 7849, 7861,
 7876, 7877, 7879, 7882, 7884, 7888,
 7890, 7896, 7898, 7899, 7901, 7907,
 7908, 7909, 7911, 7913, 7915, 7918,
 7920, 7924, 7929, 7933, 7935, 7939,
 7949, 7999, 8008, 8017, 8030, 8047,
 8080, 8095, 8097, 8130, 8132, 8153,
 8163, 8170, 8177, 8184, 8195, 8206,
 8211, 8222, 8256, 8271, 8349, 8363,
 8400, 8418, 8428, 8447, 8452, 8466,
 8471, 8481, 8492, 8504, 8516, 8576,
 8623, 8631, 8660, 8709, 8717, 8741,
 8788, 8794, 8798, 8826, 8847, 8852,
 8854, 8861, 8863, 8870, 8911, 8923,
 8928, 8945, 8975, 8981, 9005, 9007,
 9076, 9117, 9137, 9138, 9151, 9156,
 9182, 9191, 9193, 9195, 9212, 9239,
 9241, 9243, 9245, 9252, 9543, 9545,

9557, 9563, 9565, 9582, 9584, 9586,
 9628, 9645, 9651, 9656, 9669, 9675,
 9680, 9694, 9697, 9707, 9717, 9739,
 9748, 9750, 9784, 9786, 9793, 9798,
 9803, 9816, 9821, 9845, 9857, 9878,
 9894, 9910, 9912, 9914, 9930, 9940,
 9942, 9944, 9960, 9967, 9981, 9994,
 9999, 10003, 10011, 10013, 10023,
 10050, 10059, 10068, 10069, 10080,
 10082, 10084, 10086, 10088, 10090,
 10092, 10094, 10096, 10098, 10100,
 10102, 10104, 10106, 10108, 10110,
 10112, 10114, 10116, 10118, 10120,
 10122, 10124, 10126, 10128, 10130,
 10132, 10134, 10136, 10138, 10140,
 10142, 10144, 10146, 10148, 10150,
 10152, 10154, 10156, 10158, 10160,
 10162, 10164, 10166, 10168, 10170,
 10172, 10174, 10176, 10178, 10180,
 10182, 10184, 10186, 10188, 10190,
 10192, 10194, 10196, 10198, 10200,
 10202, 10204, 10206, 10214, 10217,
 10225, 10232, 10235, 10243, 10252,
 10261, 10266, 10271, 10288, 10301,
 10315, 10341, 10355, 10365, 10408,
 10457, 10472, 10585, 10630, 10632,
 10640, 10677, 10682, 10691, 10698,
 10715, 10717, 10722, 10727, 10777,
 10779, 10782, 10794, 10805, 10808,
 10825, 10831, 10843, 10845, 10869,
 10871, 10916, 10919, 10922, 10925,
 10942, 10948, 10960, 10963, 10966,
 10969, 10975, 10981, 10988, 10990,
 11034, 11035, 11079, 11089, 11096,
 11106, 11122, 11135, 11142, 11157,
 11163, 11211, 11507, 11523, 11540,
 11542, 11544, 11546, 11574, 11576,
 11578, 11580, 11600, 11602, 11604,
 11606, 11608, 11610, 11612, 11614,
 11616, 13161, 13168, 13427, 16012,
 16639, 16642, 16644, 16646, 16655,
 16656, 16659, 16661, 16665, 16666,
 16667, 16668, 16669, 16675, 16680,
 16705, 16724, 16732, 16744, 16781,
 16788, 16813, 16820, 16827, 16834,
 16841, 16843, 16847, 16870, 16879,
 16899, 16909, 16919, 16921, 16923,
 16924, 16931, 16934, 16944, 16954,
 17045, 17052, 17054, 17069, 17107,
 17109, 17111, 17127, 17139, 17173,
 17184, 17195, 17206, 17217, 17228,
 17241, 17259, 17266, 17276, 17281,
 17298, 17315, 17329, 17347, 17369,
 17406, 17409, 17455, 17503, 17505,
 17530, 17565, 17577, 17583, 17589,
 17601, 17620, 17629, 17640, 17646,
 17651, 17659, 17672, 17685, 17699,
 17714, 17721, 17727, 17736, 17743,
 17751, 17755, 17760, 17768, 17773,
 17786, 17788, 17790, 17796, 17803,
 17813, 17816, 17819, 17821, 17825,
 17828, 17887, 17948, 17950, 17979,
 18025, 18027, 18029, 18035, 18037,
 18039, 18045, 18070, 18074, 18078,
 18131, 18133, 18137, 18153, 18156,
 18158, 18162, 18352, 18353, 19162,
 19165, 19178, 19195, 19200, 19207,
 19209, 19211, 19317, 19318, 19496,
 19498, 19500, 19502, 19514, 19519,
 19541, 19556, 19582, 19596, 19598,
 19600, 19602, 19604, 19609, 19614,
 19624, 19633, 19635, 19638, 19640,
 19642, 19644, 19649, 19654, 19659,
 19661, 19673, 19678, 19680, 19682,
 19684, 19686, 19688, 19690, 19692,
 19703, 19708, 19717, 19726, 19731,
 19735, 19737, 19739, 19749, 19754,
 19759, 19764, 19774, 19793, 19804,
 19806, 19816, 19834, 19853, 19875,
 19880, 19882, 19885, 19892, 19898,
 19900, 19902, 19907, 19912, 19921,
 19931, 19933, 19936, 19938, 19954,
 19959, 19979, 20002, 20005, 20018,
 20030, 20035, 20037, 20039, 20041,
 20043, 20045, 20047, 20049, 20054,
 20064, 20077, 20090, 20095, 20097,
 20099, 20108, 20120, 20132, 20144,
 20163, 20165, 20167, 20169, 20171,
 20223, 20236, 20265, 20270, 20274,
 20279, 20281, 20289, 20299, 20307,
 20312, 20317, 20328, 20338, 20348,
 20350, 20352, 20354, 20385, 20387,
 20392, 20394, 20396, 20399, 20420,
 20431, 20444, 20446, 20448, 20450,
 20452, 20454, 20456, 20458, 20460,
 20471, 20473, 20475, 20477, 20486,
 20489, 20491, 20493, 20495, 20506,
 20537, 20539, 20541, 20543, 20558
 \cs_new_protected:Npx 11,
 289, 289, 294, 1784, 1820, 1825,
 1911, 1971, 2317, 2321, 2326, 2463,
 2467, 2468, 3029, 3558, 5800, 9016,
 9018, 9020, 9022, 9024, 9033, 9035,
 9037, 9261, 9263, 9265, 9267, 9269,
 9278, 9280, 9282, 11151, 19441,
 19448, 19454, 19460, 19478, 19487
 \cs_new_protected_nopar:Nn
 13, 13, 1909, 1969

`\cs_new_protected_nopar:Npn` 11, 11, [1784](#), [1801](#), [1814](#), [1818](#)
`\cs_new_protected_nopar:Npx` [11](#), [1784](#), [1814](#), [1819](#)
`\cs_set:Nn` . [13](#), [13](#), [275](#), [275](#), [1909](#), [1969](#)
`\cs_set:Npn` [10](#), [11](#), [11](#), [34](#), [34](#), [35](#), [272](#),
[275](#), [444](#), [1347](#), [1376](#), [1382](#), [1383](#),
[1384](#), [1385](#), [1386](#), [1387](#), [1388](#), [1389](#),
[1390](#), [1391](#), [1392](#), [1393](#), [1394](#), [1395](#),
[1396](#), [1397](#), [1398](#), [1399](#), [1400](#), [1401](#),
[1402](#), [1403](#), [1404](#), [1405](#), [1406](#), [1407](#),
[1408](#), [1409](#), [1410](#), [1411](#), [1413](#), [1567](#),
[1572](#), [1577](#), [1582](#), [1589](#), [1595](#), [1596](#),
[1608](#), [1612](#), [1614](#), [1616](#), [1674](#), [1676](#),
[1678](#), [1680](#), [1682](#), [1684](#), [1686](#), [1688](#),
[1737](#), [1784](#), [1800](#), [1808](#), [1808](#), [1909](#),
[1969](#), [3089](#), [3550](#), [3556](#), [3655](#), [3670](#),
[3678](#), [5138](#), [5203](#), [5323](#), [5474](#), [5823](#),
[7170](#), [7242](#), [7512](#), [8856](#), [8858](#), [9733](#),
[11549](#), [11557](#), [11566](#), [11583](#), [11591](#),
[11619](#), [12900](#), [16804](#), [17798](#), [19327](#)
`\cs_set:Npx` [11](#), [279](#), [512](#),
[1347](#), [1598](#), [1808](#), [1809](#), [3463](#), [3468](#),
[3485](#), [3486](#), [5205](#), [9720](#), [9726](#), [9976](#),
[11039](#), [11040](#), [11041](#), [11042](#), [11043](#)
`\cs_set_eq:NN` [15](#), [15](#), [15](#), [273](#),
[281](#), [1521](#), [1826](#), [2321](#), [2344](#), [2467](#),
[2502](#), [2504](#), [2513](#), [2514](#), [2515](#), [2516](#),
[2532](#), [2537](#), [2552](#), [3032](#), [3226](#), [3457](#),
[3461](#), [3482](#), [3484](#), [3561](#), [4911](#), [4912](#),
[4913](#), [4914](#), [5030](#), [6591](#), [6592](#), [6594](#),
[6704](#), [6705](#), [6716](#), [9971](#), [9986](#), [10018](#),
[10029](#), [10039](#), [11045](#), [11046](#), [11047](#),
[11055](#), [11508](#), [17047](#), [17048](#), [17053](#)
`\cs_set_nopar:Nn` [13](#), [13](#), [1909](#), [1969](#)
`\cs_set_nopar:Npn`
. [10](#), [11](#), [11](#), [52](#), [272](#), [272](#),
[272](#), [1347](#), [1375](#), [1800](#), [1802](#), [9932](#), [9997](#)
`\cs_set_nopar:Npx`
. [11](#), [1347](#), [1379](#), [1800](#), [1803](#), [2064](#),
[2241](#), [4930](#), [4932](#), [4934](#), [4948](#), [4950](#),
[4952](#), [4954](#), [4972](#), [4974](#), [4976](#), [4978](#)
`\cs_set_protected:Nn` [13](#), [13](#), [1909](#), [1969](#)
`\cs_set_protected:Npn` [10](#),
[11](#), [11](#), [254](#), [272](#), [1347](#), [1363](#), [1365](#),
[1367](#), [1369](#), [1371](#), [1373](#), [1377](#), [1415](#),
[1417](#), [1419](#), [1421](#), [1423](#), [1428](#), [1430](#),
[1432](#), [1434](#), [1436](#), [1441](#), [1453](#), [1468](#),
[1485](#), [1502](#), [1508](#), [1514](#), [1520](#), [1522](#),
[1524](#), [1536](#), [1550](#), [1690](#), [1692](#), [1695](#),
[1696](#), [1697](#), [1701](#), [1702](#), [1706](#), [1708](#),
[1711](#), [1713](#), [1715](#), [1716](#), [1717](#), [1720](#),
[1733](#), [1735](#), [1739](#), [1749](#), [1760](#), [1764](#),
[1774](#), [1782](#), [1786](#), [1820](#), [1820](#), [1853](#),
[1874](#), [2529](#), [2534](#), [2539](#), [2544](#), [2549](#),
[2554](#), [2559](#), [2564](#), [3021](#), [3146](#), [3316](#),
[3334](#), [4780](#), [4997](#), [5004](#), [5026](#), [5032](#),
[5040](#), [5047](#), [5773](#), [5796](#), [6286](#), [6297](#),
[6313](#), [6320](#), [6324](#), [6334](#), [6337](#), [6351](#),
[6366](#), [6369](#), [6378](#), [6386](#), [6620](#), [8085](#),
[8108](#), [8803](#), [8809](#), [8959](#), [9002](#), [9248](#),
[9250](#), [9497](#), [9964](#), [10987](#), [10989](#),
[11081](#), [11083](#), [11976](#), [12565](#), [12633](#),
[12644](#), [12907](#), [17049](#), [18749](#), [18753](#),
[18756](#), [18783](#), [18794](#), [18925](#), [18928](#),
[18950](#), [19176](#), [19336](#), [19528](#), [19533](#)
`\cs_set_protected:Npx`
. [11](#), [240](#), [1347](#), [1704](#), [1820](#), [1821](#), [9124](#)
`\cs_set_protected_nopar:Nn`
. [14](#), [14](#), [1909](#), [1969](#)
`\cs_set_protected_nopar:Npn`
. [12](#), [12](#), [272](#), [1347](#), [1814](#), [1814](#)
`\cs_set_protected_nopar:Npx`
. [12](#), [1347](#), [1814](#), [1815](#)
`\cs_show:N` [16](#), [16](#), [20](#), [159](#), [201](#),
[278](#), [742](#), [742](#), [742](#), [2025](#), [17108](#), [17110](#)
`\cs_to_str:N`
. [4](#), [17](#), [17](#), [96](#), [102](#), [265](#), [266](#), [266](#),
[407](#), [1589](#), [1602](#), [6249](#), [6250](#), [6251](#),
[6252](#), [6253](#), [6254](#), [6255](#), [6256](#), [6257](#),
[6258](#), [6259](#), [6260](#), [10992](#), [13164](#), [19158](#)
`\cs_undefine:N`
. [15](#), [15](#), [513](#), [1842](#), [9286](#), [9287](#), [9288](#)
cs internal commands:
`__cs_count_signature:N` . [22](#), [22](#), [1883](#)
`__cs_count_signature:nnN` [1883](#)
`__cs_generate_from_signature:n` .
. [1927](#), [1940](#)
`__cs_generate_from_signature:NNn`
. [1913](#), [1917](#)
`__cs_generate_from_signature:nnNNn`
. [1919](#), [1922](#)
`__cs_generate_internal_variant:n`
. [2451](#), [2463](#)
`__cs_generate_internal_variant:wwnNwnn`
. [2465](#), [2481](#)
`__cs_generate_internal_variant:wwnw`
. [2463](#)
`__cs_generate_internal_variant_-loop:n` [2463](#)
`__cs_generate_variant:N` . [2310](#), [2317](#)
`__cs_generate_variant:nnNN`
. [2313](#), [2345](#)
`__cs_generate_variant:Nnnw`
. [2352](#), [2354](#)
`__cs_generate_variant:ww` [2317](#)
`__cs_generate_variant:wwNN`
. [291](#), [292](#), [292](#), [292](#), [2361](#), [2444](#)

- __cs_generate_variant:wwNw .. [2317](#)
 - __cs_generate_variant_loop:nNwN
 - [291](#), [292](#), [2362](#), [2374](#)
 - __cs_generate_variant_loop_-
 - end:nwwwNNnn [291](#), [292](#), [292](#), [2364](#), [2374](#)
 - __cs_generate_variant_loop_-
 - invalid:NNwNNnn [291](#), [2374](#)
 - __cs_generate_variant_loop_-
 - long:wNNnn [292](#), [2367](#), [2374](#)
 - __cs_generate_variant_loop_-
 - same:w [291](#), [2374](#)
 - __cs_generate_variant_same:N ...
 - [291](#), [2393](#), [2432](#)
 - __cs_get_function_name:N [22](#), [22](#), [1614](#)
 - __cs_get_function_signature:N ..
 - [22](#), [22](#), [1614](#)
 - __cs_parm_from_arg_count:nnTF ..
 - [1443](#), [1853](#), [1897](#)
 - __cs_parm_from_arg_count_-
 - test:nnTF [1853](#)
 - __cs_split_function:NN [22](#),
 - [22](#), [266](#), [1425](#), [1438](#), [1529](#), [1530](#), [1598](#), [1615](#), [1617](#), [1884](#), [1919](#), [2311](#)
 - __cs_split_function_auxi:w .. [1598](#)
 - __cs_split_function_auxii:w . [1598](#)
 - __cs_tmp:w ... [22](#), [289](#), [293](#), [1784](#),
 - [1800](#), [1802](#), [1803](#), [1804](#), [1805](#), [1806](#), [1807](#), [1808](#), [1809](#), [1810](#), [1811](#), [1812](#), [1813](#), [1814](#), [1815](#), [1816](#), [1817](#), [1818](#), [1819](#), [1820](#), [1821](#), [1822](#), [1823](#), [1824](#), [1825](#), [1909](#), [1945](#), [1946](#), [1947](#), [1948](#), [1949](#), [1950](#), [1951](#), [1952](#), [1953](#), [1954](#), [1955](#), [1956](#), [1957](#), [1958](#), [1959](#), [1960](#), [1961](#), [1962](#), [1963](#), [1964](#), [1965](#), [1966](#), [1967](#), [1968](#), [1969](#), [1977](#), [1978](#), [1979](#), [1980](#), [1981](#), [1982](#), [1983](#), [1984](#), [1985](#), [1986](#), [1987](#), [1988](#), [1989](#), [1990](#), [1991](#), [1992](#), [1993](#), [1994](#), [1995](#), [1996](#), [1997](#), [1998](#), [1999](#), [2000](#), [2321](#), [2344](#), [2452](#), [2467](#), [4780](#), [4790](#), [4997](#), [5010](#), [5012](#)
 - __cs_to_str:N [265](#), [1589](#)
 - __cs_to_str:w [265](#), [265](#), [1589](#)
 - csc [194](#)
 - cscd [194](#)
 - \csname [14](#), [21](#), [39](#), [43](#), [49](#),
 - [68](#), [90](#), [92](#), [98](#), [123](#), [146](#), [150](#), [222](#), [321](#)
 - \currentgrouplevel [613](#)
 - \currentgrouptype [614](#)
 - \currentifbranch [615](#)
 - \currentiflevel [616](#)
 - \currentifttype [617](#)
- D**
- \day [322](#)
 - dd [197](#)
 - \deadcycles [323](#)
 - \def [74](#),
 - [75](#), [76](#), [105](#), [122](#), [124](#), [125](#), [143](#), [144](#), [147](#), [163](#), [178](#), [207](#), [211](#), [236](#), [275](#), [324](#)
 - default commands:
 - .default:n [164](#), [10116](#)
 - \defaultthyphenchar [325](#)
 - \defaultskewchar [326](#)
 - deg [196](#)
 - \delcode [327](#)
 - \delimiter [328](#)
 - \delimiterfactor [329](#)
 - \delimitershortfall [330](#)
 - \detokenize [68](#), [222](#), [618](#)
 - \DH [18960](#)
 - \dh [18960](#)
 - dim commands:
 - \dim_abs:n [76](#), [76](#), [4518](#)
 - \dim_add:Nn [76](#), [76](#), [4508](#)
 - \dim_case:nn [79](#), [79](#), [4594](#)
 - \dim_case:nnTF [79](#), [79](#), [4594](#), [4599](#), [4604](#)
 - \dim_compare:nNnTF
 - [77](#), [77](#), [79](#), [79](#), [79](#), [80](#), [4553](#), [4618](#), [4653](#), [4661](#), [4670](#), [4676](#), [8274](#), [8277](#), [8280](#), [8289](#), [8292](#), [8295](#), [8304](#), [8311](#), [8369](#), [8374](#), [8381](#), [8494](#), [8506](#), [17417](#), [17434](#), [17463](#), [17477](#), [17487](#)
 - \dim_compare:nTF [74](#), [78](#), [78](#), [80](#), [80](#),
 - [80](#), [80](#), [84](#), [4558](#), [4625](#), [4633](#), [4642](#), [4648](#)
 - \dim_compare_p:n [78](#), [78](#), [4558](#)
 - \dim_compare_p:nNn [77](#), [77](#), [4553](#)
 - \dim_const:Nn [75](#), [75](#), [4477](#), [4715](#), [4716](#)
 - \dim_do_until:nn [80](#), [80](#), [4623](#)
 - \dim_do_until:nNnn [79](#), [79](#), [4651](#)
 - \dim_do_while:nn [80](#), [80](#), [4623](#)
 - \dim_do_while:nNnn [79](#), [79](#), [4651](#)
 - \dim_eval:n
 - [77](#), [78](#), [80](#), [80](#), [80](#), [88](#), [4597](#), [4602](#), [4607](#), [4612](#), [4679](#), [4714](#), [8070](#), [8120](#), [8190](#), [8201](#), [8214](#), [8216](#), [8218](#), [8220](#), [8226](#), [8231](#), [8237](#), [8244](#), [8251](#), [8477](#), [8478](#), [8747](#), [8748](#), [8749](#), [17470](#), [17494](#), [17568](#), [17570](#), [17575](#), [17655](#), [17656](#), [17663](#), [17664](#), [17740](#), [17747](#)
 - \dim_gadd:Nn [76](#), [4508](#)
 - .dim_gset:N [164](#), [10124](#)
 - \dim_gset:Nn [76](#), [4480](#), [4497](#)
 - \dim_gset_eq:NN [76](#), [4502](#)
 - \dim_gsub:Nn [76](#), [4508](#)
 - \dim_gzero:N [75](#), [4483](#), [4490](#)
 - \dim_gzero_new:N [75](#), [4487](#)

\dim_if_exist:NTF 75, 75, 4488, 4490, 4493
 \dim_if_exist_p:N 75, 75, 4493
 \dim_log:N 211, 211, 18068
 \dim_log:n 211, 211, 18068
 \dim_max:nn . 76, 76, 4518, 17634, 17638
 \dim_min:nn
 . . . 76, 76, 4518, 17632, 17636, 17649
 \dim_new:N 75, 75, 75,
4469, 4479, 4488, 4490, 4717, 4718,
 4719, 4720, 7955, 7977, 7978, 7981,
 7982, 7983, 7984, 8564, 8566, 8567,
 17118, 17119, 17120, 17121, 17122,
 17123, 17124, 17125, 17525, 17526,
 17527, 17528, 17529, 17670, 17671
 \dim_ratio:nn 77, 77, 77, 360, 4549, 4708
 .dim_set:N 164, 10124
 \dim_set:Nn 76,
 76, 4497, 8053, 8102, 8276, 8281,
 8291, 8296, 8306, 8313, 8326, 8352,
 8372, 8431, 8432, 8434, 8436, 8454,
 8455, 8565, 8668, 8669, 8720, 8721,
 8722, 8724, 17141, 17142, 17143,
 17175, 17186, 17261, 17262, 17263,
 17278, 17354, 17355, 17356, 17358,
 17360, 17362, 17571, 17603, 17611,
 17622, 17623, 17624, 17625, 17631,
 17633, 17635, 17637, 17642, 17648,
 17704, 17706, 17708, 17716, 17718
 \dim_set_eq:NN
 . 76, 76, 4502, 8055, 8056, 8104, 8105
 \dim_show:N 82, 82, 4711
 \dim_show:n . . 82, 82, 769, 4713, 18071
 \dim_sub:Nn 76, 76, 4508
 \dim_to_decimal:n . . . 81, 81, 4684,
 4700, 4705, 20182, 20183, 20184,
 20185, 20186, 20188, 20310, 20315,
 20321, 20322, 20323, 20324, 20333,
 20334, 20335, 20426, 20445, 20553
 \dim_to_decimal_in_bp:n 81, 81, 81,
4699, 19547, 19548, 19549, 19607,
 19612, 19618, 19619, 19620, 19628,
 19629, 19668, 19672, 19676, 19770,
 19824, 19825, 19826, 19905, 19910,
 19916, 19917, 19925, 19926, 19927,
 20025, 20029, 20033, 20139, 20430
 \dim_to_decimal_in_sp:n
 81, 81, 81, 4701
 \dim_to_decimal_in_unit:nn
 81, 81, 81, 4703
 \dim_to_fp:n 82, 82, 82, 585, 602, 719,
4711, 8319, 8321, 8331, 8332, 8333,
 8334, 8356, 8357, 8358, 8359, 16407,
 17179, 17180, 17190, 17191, 17248,
 17251, 17252, 17279, 17289, 17290,
 17306, 17307, 17322, 17336, 17339,
 17340, 17607, 17608, 17615, 17616,
 17675, 17678, 17679, 17717, 17719
 \dim_until_do:nn 80, 80, 4623
 \dim_until_do:nNnn 79, 79, 4651
 \dim_use:N 80, 80, 80, 80,
 4521, 4527, 4528, 4529, 4535, 4536,
 4537, 4561, 4582, 4680, 4681, 4687,
 8462, 17573, 17575, 17581, 17587,
 17596, 17597, 17598, 17725, 17732
 \dim_while_do:nn 80, 80, 4623
 \dim_while_do:nNnn 80, 80, 4651
 \dim_zero:N 75, 75, 4483,
 4488, 8267, 8268, 17144, 17264, 17357
 dim_zero:N 75
 \dim_zero_new:N 75, 75, 4487
 \c_max_dim 82, 85, 4715, 4809,
 17622, 17623, 17624, 17625, 17642
 \g_tmpa_dim 82, 4717
 \l_tmpa_dim 82, 4717
 \g_tmpb_dim 82, 4717
 \l_tmpb_dim 82, 4717
 \c_zero_dim 82, 4483, 4715,
 4808, 7898, 7912, 8274, 8277, 8280,
 8289, 8292, 8295, 8304, 8311, 8369,
 8374, 8381, 17400, 17421, 17432,
 17438, 17450, 17463, 17467, 17475,
 17477, 17481, 17487, 17497, 19559
 dim internal commands:
 __dim_abs:N 4518
 __dim_case:nnTF 4594
 __dim_case:nw 4594
 __dim_case_end:nw 4594
 __dim_compare:w 4558
 __dim_compare:wNN 356, 4558
 __dim_compare_!:w 4558
 __dim_compare_<:w 4558
 __dim_compare_=:w 4558
 __dim_compare_>:w 4558
 __dim_compare_end:w 4566, 4592
 __dim_eval:w . . . 88, 88, 360, 585,
4466, 4498, 4509, 4514, 4521, 4527,
 4528, 4529, 4535, 4536, 4537, 4552,
 4555, 4561, 4582, 4587, 4680, 4687,
 4702, 7788, 7790, 7792, 7801, 7803,
 7805, 7807, 7883, 7897, 7910, 7926,
 7950, 12099, 12112, 12669, 17413,
 17415, 17459, 17461, 17547, 19778
 __dim_eval_end: 88, 88, 88, 88, 4466,
 4498, 4509, 4514, 4521, 4531, 4539,
 4552, 4555, 4680, 4687, 4702, 7788,
 7790, 7792, 7801, 7803, 7805, 7807,

- 7883, 7897, 7910, 7926, 7950, 17413,
 17415, 17459, 17461, 17549, 19778
 _dim_maxmin:wwN 4518
 _dim_ratio:n 4549
 _dim_to_decimal:w 4684
 \dimen 331, 3373
 \dimendef 332
 \dimexpr 619
 \directlua 16, 23, 53, 59, 61, 870
 \disablecjktoken 1153
 \discretionary 333
 \displayindent 334
 \displaylimits 335
 \displaystyle 336
 \displaywidowpenalties 620
 \displaywidowpenalty 337
 \displaywidth 338
 \divide 339
 \DJ 18961
 \dj 18961
 \doublehyphenemerits 340
 \dp 341
 \draftmode 947
 driver internal commands:
 _driver_absolute_lengths:n ...
 19808, 19821
 _driver_box_use_clip:N
 . 221, 221, 17407, 19541, 19816, 20171
 _driver_box_use_rotate:Nn
 . 221, 221, 17162, 19556, 19834, 20223
 _driver_box_use_scale:Nnn 222,
 222, 17373, 19582, 19849, 19853, 20236
 \g_driver_clip_path_int
 20171, 20358,
 20361, 20374, 20403, 20406, 20414
 _driver_color_ensure_current: .
 222, 222, 8795, 8801, 8804,
 19478, 19514, 19523, 19875, 20265
 _driver_color_reset:
 .. 19478, 19514, 19523, 19875, 20265
 \l_driver_color_stack_int
 19477, 19483, 19492
 \l_driver_cos_fp 19556
 \l_driver_current_color_tl
 19467, 19484, 19504, 19516,
 19530, 19865, 19877, 20255, 20267
 _driver_draw_add_to_path:n ...
 20307, 20353
 _driver_draw_begin:
 222, 222, 19596, 19885, 20274
 _driver_draw_cap_but:
 .. 225, 225, 225, 19661, 20018, 20420
 _driver_draw_cap_rectangle: ...
 225, 19661, 20018, 20420
 _driver_draw_cap_round:
 225, 19661, 20018, 20420
 _driver_draw_clip:
 224, 224, 19638, 19936, 20352
 \g_driver_draw_clip_bool
 19936, 20352
 _driver_draw_closepath:
 223, 223, 19638, 19936, 20352
 _driver_draw_closestroke:
 223, 223, 19638, 19936, 20352
 _driver_draw_color_cmyk:nnnn ..
 225, 225, 19692, 20049, 20460
 _driver_draw_color_cmyk_-
 aux:nnnn 19692
 _driver_draw_color_cmyk_-
 aux:NNnnnn
 20460, 20472, 20474, 20476
 _driver_draw_color_cmyk_-
 fill:nnnn 225, 19692, 20049, 20460
 _driver_draw_color_cmyk_-
 stroke:nnnn 225, 19692, 20049, 20460
 _driver_draw_color_gray:n
 .. 225, 225, 225, 19692, 20049, 20460
 _driver_draw_color_gray_aux:n .
 19692
 _driver_draw_color_gray_-
 aux:NNn . 20477, 20490, 20492, 20494
 _driver_draw_color_gray_-
 aux:nNN 20481, 20486, 20488
 _driver_draw_color_gray_fill:n
 225, 19692, 20049, 20460
 _driver_draw_color_gray_-
 stroke:n . 225, 19692, 20049, 20460
 _driver_draw_color_reset: .. 20049
 _driver_draw_color_rgb:nnn ...
 225, 19692, 20049, 20460
 _driver_draw_color_rgb_aux:nnn
 19692
 _driver_draw_color_rgb_-
 auxi:NNnnn
 20495, 20538, 20540, 20542
 _driver_draw_color_rgb_-
 auxii:nnn 20464
 _driver_draw_color_rgb_-
 auxii:nnnNN ... 20487, 20499, 20506
 _driver_draw_color_rgb_-
 fill:nnn . 225, 19692, 20049, 20460
 _driver_draw_color_rgb_-
 stroke:nnn 225, 19692, 20049, 20460
 _driver_draw_curveto:nnnnnn ...
 223, 223, 19604, 19902, 20307
 _driver_draw_dash:n
 19661, 20018, 20420

- _driver_draw_dash:nn [224](#), [224](#), [19661](#), [20018](#), [20420](#)
 - _driver_draw_dash_aux:nn ... [20420](#)
 - _driver_draw_discardpath: [224](#), [224](#), [19638](#), [19936](#), [20352](#)
 - _driver_draw_end: [222](#), [222](#), [19596](#), [19885](#), [20274](#)
 - \g_driver_draw_eor_bool . [19633](#), [19647](#), [19652](#), [19657](#), [19931](#), [19947](#), [19964](#), [19972](#), [19984](#), [19995](#), [20011](#)
 - _driver_draw_evenodd_rule: ... [224](#), [19633](#), [19931](#), [20348](#)
 - _driver_draw_fill: [224](#), [224](#), [19638](#), [19936](#), [20352](#)
 - _driver_draw_fillstroke: [224](#), [19638](#), [19936](#), [20352](#)
 - _driver_draw_hbox:nnnnnn [226](#)
 - _driver_draw_hbox:Nnnnnnn [226](#), [19774](#), [20144](#), [20558](#)
 - _driver_draw_join_bevel: [225](#), [19661](#), [20018](#), [20420](#)
 - _driver_draw_join_miter: [225](#), [19661](#), [20018](#), [20420](#)
 - _driver_draw_join_round: [225](#), [19661](#), [20018](#), [20420](#)
 - _driver_draw_lineto:nn [223](#), [223](#), [19604](#), [19902](#), [20307](#)
 - _driver_draw_linewidth:n [224](#), [224](#), [19661](#), [20018](#), [20420](#)
 - _driver_draw_literal:n [19594](#), [19601](#), [19603](#), [19606](#), [19611](#), [19616](#), [19626](#), [19639](#), [19641](#), [19643](#), [19646](#), [19651](#), [19656](#), [19660](#), [19663](#), [19675](#), [19679](#), [19681](#), [19683](#), [19685](#), [19687](#), [19689](#), [19691](#), [19705](#), [19710](#), [19719](#), [19733](#), [19736](#), [19738](#), [19751](#), [19756](#), [19761](#), [19766](#), [19882](#), [19899](#), [19901](#), [19904](#), [19909](#), [19914](#), [19923](#), [19937](#), [19940](#), [19941](#), [19942](#), [19945](#), [19951](#), [19961](#), [19962](#), [19967](#), [19970](#), [19976](#), [19981](#), [19982](#), [19987](#), [19988](#), [19989](#), [19990](#), [19993](#), [19999](#), [20009](#), [20015](#), [20020](#), [20032](#), [20036](#), [20038](#), [20040](#), [20042](#), [20044](#), [20046](#), [20048](#), [20051](#), [20052](#), [20056](#), [20066](#), [20079](#), [20092](#), [20096](#), [20098](#), [20101](#), [20110](#), [20122](#), [20134](#), [20272](#), [20293](#), [20301](#), [20359](#), [20378](#), [20404](#)
 - _driver_draw_miterlimit:n [225](#), [225](#), [19661](#), [20018](#), [20420](#)
 - _driver_draw_move:nn [223](#)
 - _driver_draw_moveto:nn [223](#), [19604](#), [19902](#), [20307](#)
 - _driver_draw_nonzero_rule: ... [224](#), [224](#), [19633](#), [19931](#), [20348](#)
 - _driver_draw_path:n [20352](#)
 - \g_driver_draw_path_int [20352](#)
 - \g_driver_draw_path_tl .. [20307](#), [20363](#), [20379](#), [20381](#), [20408](#), [20417](#)
 - _driver_draw_rectangle:nnnn [223](#), [223](#), [19604](#), [19902](#), [20307](#)
 - _driver_draw_scope:n [20277](#), [20281](#), [20349](#), [20351](#), [20371](#), [20411](#), [20433](#), [20445](#), [20447](#), [20449](#), [20451](#), [20453](#), [20455](#), [20457](#), [20459](#), [20508](#), [20545](#)
 - _driver_draw_scope_begin: [222](#), [222](#), [19597](#), [19600](#), [19898](#), [20276](#), [20281](#)
 - _driver_draw_scope_end: .. [222](#), [222](#), [19599](#), [19600](#), [19898](#), [20280](#), [20281](#)
 - \g_driver_draw_scope_int [20281](#)
 - \l_driver_draw_scope_int [20281](#)
 - _driver_draw_stroke: [223](#), [223](#), [224](#), [19638](#), [19936](#), [20352](#)
 - _driver_draw_transformcm:nnnnnn [226](#), [226](#), [19764](#), [19780](#), [20132](#), [20148](#), [20543](#), [20561](#)
 - _driver_literal:n [19441](#), [19496](#), [19499](#), [19501](#), [19503](#), [19544](#), [19594](#), [19793](#), [19819](#), [19837](#), [19856](#), [20163](#), [20166](#), [20168](#), [20170](#), [20174](#), [20176](#), [20193](#), [20272](#), [20562](#), [20575](#)
 - _driver_matrix:n [19460](#), [19502](#), [19564](#), [19585](#)
 - \g_driver_path_int ... [20367](#), [20384](#)
 - _driver_scope_begin: [821](#), [19448](#), [19498](#), [19543](#), [19558](#), [19584](#), [19779](#), [19804](#), [19818](#), [19836](#), [19855](#), [20146](#), [20165](#), [20560](#)
 - _driver_scope_begin:n .. [20169](#), [20195](#), [20203](#), [20207](#), [20225](#), [20238](#)
 - _driver_scope_end: [19448](#), [19498](#), [19553](#), [19578](#), [19592](#), [19783](#), [19804](#), [19831](#), [19845](#), [19863](#), [20159](#), [20165](#), [20217](#), [20218](#), [20219](#), [20234](#), [20253](#), [20576](#)
 - \l_driver_sin_fp [19556](#)
 - \l_driver_tmp_box [19774](#)
 - \dtou [1120](#)
 - \dump [342](#)
 - \dviextension [871](#)
 - \dvifedback [872](#)
 - \dvivariable [873](#)
- E**
- \edef [4](#), [106](#), [131](#), [209](#), [343](#)
 - \efcode [779](#)

- `\else` 15, 22, 44, 46,
91, 94, 95, 99, 100, 161, 165, 180, 344
- else commands:
 - `\else:` 21,
35, 40, 40, 73, 73, 73, 73, 73, 88,
144, 144, 145, 179, 179, 260, 267,
289, 379, 379, 391, 391, 559, 623,
1292, 1334, 1490, 1622, 1625, 1634,
1640, 1650, 1653, 1662, 1668, 1714,
1848, 1869, 1878, 1889, 1942, 1943,
2004, 2085, 2322, 2378, 2379, 2380,
2436, 2439, 2575, 2605, 2634, 2653,
2778, 2780, 2782, 2784, 2811, 2850,
2858, 3056, 3060, 3063, 3071, 3077,
3113, 3208, 3213, 3218, 3223, 3230,
3237, 3242, 3247, 3252, 3257, 3262,
3267, 3272, 3277, 3294, 3301, 3307,
3310, 3345, 3348, 3410, 3419, 3427,
3436, 3502, 3516, 3525, 3535, 3545,
3669, 3690, 3706, 3709, 3764, 3865,
3892, 3929, 3937, 4223, 4256, 4307,
4455, 4524, 4545, 4556, 4566, 4593,
4776, 5255, 5268, 5281, 5291, 5307,
5340, 5599, 5628, 5651, 5667, 5675,
5685, 5698, 5714, 5846, 5855, 5868,
5873, 6007, 6060, 6063, 6066, 6078,
6092, 6301, 6307, 6317, 6343, 6355,
6360, 6374, 6607, 6644, 6653, 7067,
7082, 7104, 7120, 7672, 7812, 7814,
7824, 9659, 9683, 9701, 9712, 9722,
9732, 9743, 10859, 10862, 11249,
11259, 11260, 11261, 11275, 11285,
11295, 11355, 11412, 11421, 11429,
11444, 11455, 11467, 11470, 11515,
11714, 11727, 11747, 11775, 11776,
11811, 11834, 11835, 11877, 11895,
11930, 11934, 11982, 11999, 12005,
12009, 12013, 12134, 12163, 12172,
12205, 12209, 12221, 12231, 12241,
12272, 12285, 12320, 12330, 12349,
12362, 12375, 12379, 12390, 12413,
12430, 12442, 12456, 12472, 12480,
12482, 12492, 12503, 12519, 12534,
12540, 12545, 12552, 12576, 12602,
12626, 12736, 12739, 12803, 12807,
12815, 12834, 12853, 12868, 12918,
12943, 12952, 12967, 12983, 12995,
13012, 13028, 13070, 13081, 13088,
13121, 13123, 13128, 13143, 13250,
13261, 13282, 13285, 13288, 13291,
13302, 13311, 13314, 13444, 13457,
13460, 13467, 13485, 13499, 13516,
13592, 13595, 13603, 13615, 13626,
13632, 13645, 13658, 13698, 13732,
13752, 13789, 13807, 13810, 13815,
13829, 13864, 13882, 13885, 13888,
13891, 13950, 14023, 14091, 14092,
14101, 14136, 14219, 14223, 14227,
14289, 14538, 14567, 14571, 14730,
14739, 14787, 14798, 14814, 14822,
14881, 14960, 14971, 14976, 15010,
15023, 15039, 15043, 15046, 15217,
15224, 15247, 15271, 15286, 15290,
15312, 15343, 15346, 15371, 15374,
15410, 15418, 15423, 15429, 15432,
15442, 15445, 15463, 15478, 15493,
15508, 15523, 15538, 15559, 15604,
15830, 15868, 15869, 15878, 15922,
15974, 15988, 15989, 15990, 16092,
16114, 16129, 16147, 16195, 16211,
16280, 16372, 16379, 16383, 16418,
16423, 16516, 16537, 16559, 16737,
16738, 18696, 18703, 18715, 19191
- `em` 197
- `\emergencystretch` 345
- `\enablecjktoken` 1154
- `\end` 118, 254, 346, 13382, 19847
- `\endcsname` 14, 21, 39, 43, 49,
68, 90, 92, 98, 123, 146, 150, 222, 347
- `\endgroup` 13, 36,
38, 42, 48, 74, 117, 135, 154, 203, 348
- `\endinput` 136, 349
- `\endL` 621
- `\endlinechar` 221, 234, 350
- `\endR` 622
- `\enquote` 13383
- `\ensuremath` 19151
- `\eqno` 351
- `\errhelp` 108, 127, 352
- `\errmessage` 116, 128, 353
- `\ERROR` 3160
- `\errorcontextlines` 354
- `\errorstopmode` 355
- `\escapechar` 356
- etex commands:
 - `\etex_beginL:D` 609
 - `\etex_beginR:D` 610
 - `\etex_botmarks:D` 611
 - `\etex_clubpenalties:D` 612
 - `\etex_currentgrouplevel:D` 613
 - `\etex_currentgroupstype:D` 614
 - `\etex_currentifbranch:D` 615
 - `\etex_currentiflevel:D` 616
 - `\etex_currentiftypetype:D` 617
 - `\etex_detokenize:D` 380,
618, 1317, 5279, 5426, 5838, 9723, 9741
 - `\etex_dimexpr:D` 619, 4467
 - `\etex_displaywidowpenalties:D` ... 620

<code>\etex_endL:D</code>	621	<code>\etex_TeXTeXtstate:D</code>	665
<code>\etex_endR:D</code>	622	<code>\etex_topmarks:D</code>	666
<code>\etex_eTeXrevision:D</code>	623	<code>\etex_tracingassigns:D</code>	667
<code>\etex_eTeXversion:D</code>	624	<code>\etex_tracinggroups:D</code>	668
<code>\etex_everyeof:D</code>		<code>\etex_tracingifs:D</code>	669
.....	625, 5074, 18142, 18167	<code>\etex_tracingnesting:D</code>	670
<code>\etex_firstmarks:D</code>	626	<code>\etex_tracingscantokens:D</code>	671
<code>\etex_fontchardp:D</code>	627	<code>\etex_unexpanded:D</code> 672, 1273, 1312,	
<code>\etex_fontcharht:D</code>	628	2285, 2288, 2291, 2296, 3127, 3853,	
<code>\etex_fontcharic:D</code>	629	5560, 5586, 5610, 18093, 18184, 18979	
<code>\etex_fontcharwd:D</code>	630	<code>\etex_unless:D</code>	673, 1297
<code>\etex_glueexpr:D</code> . 631, 4750, 4761,		<code>\etex_widowpenalties:D</code>	674
4766, 4785, 4792, 4798, 4801, 16412		<code>\eTeXrevision</code>	623
<code>\etex_glueshrink:D</code>	632, 18060	<code>\eTeXversion</code>	624
<code>\etex_glueshrinkorder:D</code>	633	<code>\etoksapp</code>	874
<code>\etex_gluestretch:D</code>	634, 18059	<code>\etokspre</code>	875
<code>\etex_gluestretchorder:D</code>	635	<code>\euc</code>	1121
<code>\etex_gluetomu:D</code>	636	<code>\everycr</code>	357
<code>\etex_ifcsname:D</code>	637, 1307	<code>\everydisplay</code>	358
<code>\etex_ifdefined:D</code> . 638, 795, 1164,		<code>\everyeof</code>	625
1173, 1176, 1182, 1236, 1237, 1244,		<code>\everyhbox</code>	359
1256, 1263, 1276, 1284, 1306, 1342		<code>\everyjob</code>	66, 67, 360
<code>\etex_iffontchar:D</code>	639	<code>\everymath</code>	361
<code>\etex_interactionmode:D</code>		<code>\everypar</code>	362
.....	640, 7853, 7856, 7857	<code>\everyvbox</code>	363
<code>\etex_interlinepenalties:D</code>	641	<code>ex</code>	197
<code>\etex_lastlinefit:D</code>	642	<code>\exhyphenpenalty</code>	364
<code>\etex_lastnodetype:D</code>	643	<code>exp</code>	192
<code>\etex_marks:D</code>	644	exp commands:	
<code>\etex_middle:D</code>	645	<code>\exp:w</code>	31, 31, 31,
<code>\etex_muexpr:D</code>		32, 32, 32, 32, 32, 260, 265, 281,	
.....	646, 4844, 4855, 4860, 4865	281, 281, 282, 288, 304, 304, 387,	
<code>\etex_mutogluue:D</code>	647	387, 393, 401, 498, 575, 575, 575,	
<code>\etex_numexpr:D</code>	648, 3650, 6275	575, 577, 578, 578, 582, 599, 604,	
<code>\etex_pagediscards:D</code>	649	605, 773, 1313, 1412, 1414, 2058,	
<code>\etex_parshapedimen:D</code>	650	2070, 2076, 2116, 2120, 2125, 2131,	
<code>\etex_parshapeindent:D</code>	651	2137, 2143, 2155, 2160, 2162, 2169,	
<code>\etex_parshapelength:D</code>	652	2225, 2232, 2237, 2245, 2247, 2250,	
<code>\etex_predisplaydirection:D</code> ...	653	2257, 2263, 2272, 2288, 2292, 2297,	
<code>\etex_protected:D</code> 654, 1349, 1351,		2299, 2732, 3041, 3047, 3544, 3898,	
1353, 1354, 1355, 1356, 1357, 1358,		3903, 3908, 3913, 4565, 4596, 4601,	
1359, 1360, 1368, 1370, 1372, 1374		4606, 4611, 5347, 5352, 5357, 5362,	
<code>\etex_readline:D</code>	655, 10876	5562, 5720, 5881, 5886, 5891, 5896,	
<code>\etex_savinghyphcodes:D</code>	656	5913, 5918, 5923, 5928, 5990, 5999,	
<code>\etex_savingvdiscards:D</code>	657	6051, 9501, 11298, 11492, 11690,	
<code>\etex_scantokens:D</code>		11808, 11809, 11810, 11811, 11922,	
.....	658, 5091, 5141, 5156	11940, 11981, 12025, 12032, 12037,	
<code>\etex_showgroups:D</code>	659	12046, 12061, 12069, 12119, 12132,	
<code>\etex_showifs:D</code>	660	12133, 12142, 12152, 12170, 12171,	
<code>\etex_showtokens:D</code>		12191, 12204, 12208, 12230, 12258,	
.....	394, 501, 661, 1280, 9601	12271, 12284, 12308, 12319, 12329,	
<code>\etex_splitbotmarks:D</code>	662	12348, 12361, 12374, 12377, 12389,	
<code>\etex_splitdiscards:D</code>	663	12412, 12441, 12455, 12471, 12491,	
<code>\etex_splitfirstmarks:D</code>	664	12502, 12508, 12518, 12563, 12573,	

12588, 12599, 12614, 12629, 12636,
 12670, 12679, 12688, 12753, 12755,
 12775, 12778, 12785, 12786, 12837,
 12852, 12873, 12879, 12886, 12915,
 12940, 12982, 12994, 13009, 13025,
 13100, 13112, 13140, 13142, 13146,
 13148, 13158, 13178, 13244, 13258,
 13268, 13377, 13378, 13379, 13510,
 13513, 13521, 13545, 13553, 14511,
 15038, 15042, 15061, 15219, 15389,
 15580, 15972, 15973, 15977, 16246,
 16250, 16295, 16299, 16344, 16363,
 16367, 16400, 16404, 16470, 16562,
 16590, 16597, 16617, 16633, 18088,
 18095, 18186, 18223, 18981, 19009
 \exp_after:wN
 . 30, 30, 31, 32, 32, 260, 279, 282,
 346, 392, 450, 501, 552, 554, 554,
 554, 575, 575, 575, 575, 575, 575,
 575, 575, 575, 577, 577, 578, 578,
 636, 637, 694, 694, 694, 1310, 1326,
 1328, 1333, 1335, 1412, 1414, 1458,
 1471, 1489, 1491, 1541, 1546, 1553,
 1593, 1597, 1600, 1601, 1633, 1635,
 1638, 1661, 1663, 1666, 1847, 1849,
 1857, 1877, 1879, 1914, 1974, 2035,
 2045, 2052, 2054, 2057, 2058, 2065,
 2069, 2070, 2075, 2076, 2081, 2086,
 2088, 2091, 2099, 2101, 2103, 2105,
 2107, 2110, 2111, 2112, 2116, 2119,
 2124, 2129, 2130, 2131, 2135, 2136,
 2137, 2141, 2142, 2143, 2147, 2148,
 2149, 2153, 2154, 2155, 2159, 2160,
 2161, 2162, 2166, 2167, 2168, 2169,
 2173, 2174, 2175, 2180, 2181, 2182,
 2183, 2187, 2188, 2189, 2190, 2224,
 2225, 2228, 2231, 2232, 2236, 2237,
 2245, 2247, 2248, 2250, 2252, 2255,
 2256, 2261, 2262, 2266, 2269, 2270,
 2271, 2275, 2282, 2284, 2285, 2286,
 2288, 2291, 2296, 2311, 2312, 2313,
 2314, 2319, 2323, 2350, 2357, 2377,
 2491, 2640, 2643, 2647, 2733, 2804,
 2810, 2812, 2836, 3041, 3042, 3047,
 3048, 3049, 3101, 3109, 3127, 3128,
 3171, 3172, 3173, 3284, 3305, 3352,
 3388, 3417, 3418, 3420, 3426, 3429,
 3501, 3503, 3513, 3514, 3515, 3517,
 3523, 3524, 3526, 3533, 3534, 3536,
 3542, 3543, 3546, 3623, 3632, 3641,
 3664, 3669, 3672, 3673, 3680, 3681,
 3697, 3698, 3719, 3720, 3837, 3842,
 3847, 3870, 3872, 3999, 4000, 4001,
 4195, 4223, 4228, 4256, 4269, 4279,
 4306, 4308, 4309, 4317, 4334, 4378,
 4520, 4524, 4527, 4528, 4535, 4536,
 4560, 4565, 4578, 4581, 4686, 4784,
 5000, 5089, 5090, 5139, 5140, 5154,
 5155, 5214, 5215, 5216, 5221, 5265,
 5278, 5279, 5337, 5338, 5426, 5533,
 5560, 5591, 5596, 5597, 5598, 5600,
 5613, 5623, 5642, 5664, 5674, 5677,
 5694, 5695, 5705, 5710, 5711, 5726,
 5727, 5728, 5838, 5942, 5977, 5978,
 5989, 5990, 5998, 6006, 6008, 6014,
 6017, 6035, 6036, 6037, 6049, 6050,
 6077, 6079, 6085, 6091, 6104, 6124,
 6135, 6151, 6159, 6167, 6174, 6181,
 6193, 6300, 6302, 6308, 6389, 6503,
 6505, 6517, 6525, 6624, 6662, 6674,
 6685, 6686, 6687, 6708, 6709, 6756,
 6785, 6786, 6787, 6857, 6858, 6861,
 7068, 7083, 7105, 7121, 7178, 7186,
 7191, 7365, 7366, 7369, 7370, 7515,
 7709, 8968, 9502, 9503, 9504, 9505,
 9517, 9601, 9602, 9637, 9638, 9647,
 9649, 9666, 9671, 9673, 9690, 9699,
 9703, 9704, 9711, 9713, 9714, 9723,
 9741, 9744, 9753, 9819, 9861, 10007,
 10008, 10280, 10360, 10361, 10380,
 10390, 10403, 10404, 10437, 10557,
 10570, 10806, 10920, 11248, 11250,
 11251, 11262, 11263, 11264, 11274,
 11276, 11284, 11286, 11294, 11296,
 11303, 11304, 11305, 11306, 11307,
 11308, 11313, 11314, 11315, 11316,
 11317, 11318, 11319, 11369, 11371,
 11398, 11402, 11437, 11441, 11456,
 11463, 11492, 11563, 11571, 11588,
 11597, 11643, 11690, 11759, 11760,
 11761, 11823, 11842, 11848, 11874,
 11875, 11878, 11889, 11893, 11900,
 11901, 11912, 11913, 11922, 11929,
 11931, 11932, 11940, 11966, 11981,
 11999, 12000, 12003, 12004, 12006,
 12007, 12011, 12012, 12014, 12015,
 12024, 12025, 12032, 12037, 12046,
 12059, 12060, 12065, 12066, 12068,
 12075, 12076, 12078, 12097, 12098,
 12102, 12105, 12110, 12111, 12113,
 12117, 12118, 12120, 12129, 12130,
 12131, 12132, 12135, 12136, 12137,
 12140, 12152, 12169, 12170, 12180,
 12181, 12191, 12203, 12207, 12220,
 12222, 12230, 12240, 12242, 12248,
 12253, 12255, 12257, 12263, 12264,
 12268, 12270, 12282, 12283, 12305,
 12307, 12313, 12316, 12318, 12322,

12327, 12332, 12333, 12343, 12344,
12346, 12347, 12350, 12354, 12359,
12373, 12376, 12388, 12397, 12404,
12405, 12406, 12407, 12409, 12411,
12422, 12423, 12424, 12425, 12427,
12429, 12431, 12432, 12433, 12439,
12440, 12450, 12454, 12455, 12457,
12458, 12459, 12464, 12470, 12481,
12483, 12490, 12491, 12493, 12494,
12501, 12507, 12517, 12569, 12570,
12571, 12572, 12586, 12587, 12589,
12597, 12598, 12612, 12613, 12629,
12636, 12667, 12668, 12669, 12675,
12676, 12677, 12678, 12684, 12685,
12686, 12687, 12754, 12766, 12767,
12773, 12774, 12776, 12777, 12783,
12784, 12801, 12802, 12804, 12805,
12813, 12814, 12816, 12819, 12831,
12832, 12833, 12835, 12836, 12837,
12850, 12851, 12852, 12854, 12855,
12856, 12865, 12866, 12867, 12872,
12874, 12883, 12884, 12885, 12891,
12892, 12894, 12912, 12913, 12914,
12917, 12919, 12920, 12921, 12938,
12939, 12942, 12944, 12945, 12946,
12965, 12966, 12968, 12969, 12970,
12980, 12981, 12982, 12984, 12985,
12986, 12992, 12993, 12994, 12996,
12997, 12998, 13007, 13008, 13011,
13013, 13014, 13015, 13023, 13024,
13027, 13029, 13030, 13031, 13052,
13053, 13054, 13055, 13056, 13057,
13058, 13068, 13069, 13071, 13072,
13073, 13093, 13094, 13095, 13096,
13097, 13098, 13099, 13100, 13105,
13106, 13107, 13108, 13109, 13110,
13111, 13126, 13127, 13129, 13130,
13137, 13138, 13139, 13144, 13145,
13147, 13156, 13157, 13172, 13174,
13175, 13176, 13243, 13257, 13267,
13275, 13276, 13376, 13443, 13445,
13484, 13486, 13508, 13509, 13512,
13519, 13520, 13523, 13524, 13531,
13536, 13543, 13544, 13551, 13552,
13599, 13600, 13601, 13603, 13614,
13630, 13631, 13633, 13634, 13644,
13646, 13652, 13653, 13657, 13660,
13682, 13684, 13697, 13699, 13705,
13707, 13710, 13716, 13718, 13720,
13721, 13722, 13724, 13729, 13731,
13733, 13737, 13740, 13746, 13747,
13751, 13753, 13754, 13755, 13763,
13765, 13766, 13773, 13779, 13786,
13787, 13792, 13793, 13794, 13795,
13813, 13814, 13815, 13821, 13822,
13823, 13828, 13830, 13838, 13840,
13842, 13843, 13845, 13856, 13858,
13860, 13861, 13866, 13917, 13918,
13925, 13926, 13928, 13930, 13932,
13934, 13936, 13938, 13940, 13949,
13951, 13957, 13959, 13961, 13962,
13963, 13969, 13971, 13973, 13974,
13975, 13996, 13997, 14000, 14008,
14010, 14014, 14015, 14016, 14017,
14022, 14024, 14030, 14033, 14036,
14039, 14047, 14050, 14053, 14056,
14063, 14065, 14071, 14079, 14081,
14083, 14100, 14102, 14109, 14111,
14114, 14120, 14122, 14124, 14125,
14126, 14128, 14142, 14143, 14146,
14164, 14166, 14168, 14180, 14183,
14186, 14189, 14192, 14195, 14198,
14201, 14205, 14217, 14221, 14225,
14228, 14243, 14249, 14251, 14253,
14263, 14287, 14290, 14302, 14304,
14308, 14309, 14310, 14312, 14313,
14315, 14319, 14320, 14321, 14322,
14335, 14340, 14342, 14349, 14352,
14355, 14358, 14361, 14364, 14372,
14373, 14385, 14393, 14395, 14405,
14407, 14414, 14423, 14425, 14428,
14431, 14434, 14437, 14450, 14452,
14460, 14462, 14470, 14472, 14481,
14484, 14487, 14494, 14509, 14510,
14527, 14529, 14530, 14587, 14600,
14602, 14608, 14621, 14623, 14625,
14649, 14663, 14665, 14672, 14674,
14714, 14715, 14716, 14718, 14719,
14720, 14722, 14723, 14729, 14731,
14732, 14738, 14740, 14741, 14742,
14743, 14755, 14761, 14763, 14794,
14801, 14808, 14828, 14829, 14831,
14833, 14835, 14848, 14853, 14854,
14855, 14856, 14857, 14861, 14866,
14868, 14874, 14880, 14882, 14883,
14889, 14890, 14891, 14892, 14893,
14894, 14895, 14896, 14901, 14903,
14905, 14907, 14909, 14913, 14915,
14917, 14919, 14921, 14923, 14941,
14945, 14953, 14954, 14959, 14961,
14970, 14973, 14974, 14975, 14977,
14978, 14979, 14987, 14993, 15005,
15008, 15009, 15011, 15012, 15030,
15031, 15035, 15041, 15045, 15047,
15064, 15082, 15088, 15097, 15098,
15099, 15100, 15108, 15124, 15140,
15156, 15172, 15188, 15214, 15218,
15219, 15223, 15225, 15258, 15264,

- 15265, 15267, 15269, 15270, 15272,
15273, 15283, 15284, 15287, 15288,
15289, 15291, 15292, 15293, 15310,
15311, 15313, 15314, 15320, 15322,
15325, 15328, 15331, 15334, 15342,
15345, 15347, 15350, 15357, 15361,
15369, 15370, 15373, 15375, 15377,
15381, 15385, 15390, 15391, 15399,
15400, 15401, 15402, 15419, 15420,
15426, 15427, 15552, 15553, 15554,
15556, 15574, 15575, 15576, 15577,
15578, 15579, 15586, 15595, 15602,
15603, 15613, 15777, 15778, 15779,
15800, 15801, 15811, 15820, 15828,
15829, 15831, 15832, 15837, 15838,
15847, 15854, 15863, 15864, 15877,
15879, 15907, 15908, 15917, 15920,
15945, 15951, 15952, 15971, 15977,
15979, 16004, 16005, 16007, 16021,
16022, 16030, 16040, 16074, 16077,
16087, 16088, 16091, 16093, 16099,
16113, 16115, 16156, 16159, 16179,
16241, 16245, 16250, 16253, 16259,
16260, 16279, 16281, 16282, 16290,
16294, 16299, 16302, 16308, 16334,
16335, 16341, 16342, 16343, 16350,
16358, 16362, 16367, 16370, 16378,
16381, 16382, 16384, 16385, 16395,
16399, 16404, 16409, 16410, 16411,
16417, 16419, 16422, 16424, 16466,
16468, 16469, 16486, 16506, 16512,
16515, 16518, 16521, 16528, 16534,
16538, 16548, 16557, 16562, 16564,
16573, 16585, 16595, 16596, 16604,
16611, 16620, 16621, 16629, 16728,
16866, 16867, 16876, 16895, 16896,
16911, 16912, 16913, 17834, 17835,
17913, 17914, 17973, 17988, 17997,
18003, 18006, 18093, 18113, 18145,
18146, 18184, 18221, 18238, 18239,
18395, 18396, 18417, 18418, 18466,
18487, 18520, 18614, 18664, 18760,
18762, 18768, 18791, 18804, 18806,
18817, 18819, 18941, 18979, 19007,
19024, 19025, 19069, 19070, 19122,
19185, 19190, 19192, 19198, 19216
\exp_arg:N 30
\exp_args:cc
27, 1325, 1494, 1505, 1511, 1517, 2104
\exp_args:Nc 27, 27, 277, 1325, 1329,
1337, 1761, 1783, 1801, 1827, 1832,
1839, 1894, 1906, 1973, 2006, 2007,
2008, 2009, 2024, 2104, 5183, 5398,
5778, 5802, 10463, 11510, 11831, 17793
\exp_args:Ncc 28, 1829, 1833,
1841, 2014, 2015, 2016, 2017, 2104
\exp_args:Nccc 28, 2104
\exp_args:Ncco 28, 2164
\exp_args:Nccx 29, 2208
\exp_args:Ncf 28, 2127
\exp_args:NcNc 28, 2164
\exp_args:NcNo 28, 2164
\exp_args:Ncnx 29, 2208
\exp_args:Nco 28, 284, 2127
\exp_args:Ncx 28, 2193, 9510
\exp_args:Nf 27,
27, 2115, 3899, 3904, 3909, 3914,
4069, 4138, 4140, 4158, 4167, 4178,
4187, 4325, 4342, 4597, 4602, 4607,
4612, 5732, 5733, 5749, 5967, 5969,
6025, 6027, 6043, 6759, 6760, 6776,
7354, 7405, 7419, 7440, 9515, 13420,
17509, 17851, 17864, 17892, 17894,
18112, 18571, 18579, 18634, 18686
\exp_args:Nff 28, 2193
\exp_args:Nfo 28, 2193
\exp_args:NNc
... 28, 28, 256, 1828, 1831, 1840,
1908, 2010, 2011, 2012, 2013, 2028,
2104, 4030, 4037, 9517, 10806, 10920
\exp_args:Nnc 28, 2193
\exp_args:NNf 28,
2127, 4023, 10805, 10919, 15546, 15547
\exp_args:Nnf 28, 2193
\exp_args:Nnnc 28, 2208
\exp_args:NNNo
.... 28, 28, 2099, 5080, 18148, 18171
\exp_args:NNno 28, 2208
\exp_args:Nnno 28, 2208
\exp_args:NNNV 28, 2164
\exp_args:NNNx 29, 2208
\exp_args:NNnx 29, 29, 2208
\exp_args:Nnnx 29, 2208
\exp_args:NNo 24,
24, 24, 28, 2099, 4057, 7509, 11073
\exp_args:Nno 28, 2193, 3615,
4568, 5094, 7309, 11548, 11556,
11565, 11582, 11590, 11618, 18152
\exp_args:NNoo 28, 28, 2208
\exp_args:NNox 29, 2208
\exp_args:Nnox 29, 2208
\exp_args:NNV 28, 2127
\exp_args:NNv 28, 2127
\exp_args:NnV 28, 2193
\exp_args:NNx 28, 28, 2193
\exp_args:Nnx 28, 2193
\exp_args:No 27, 27, 732,
2099, 4057, 4144, 4150, 4790, 5074,

- 5079, 5314, 5315, 5316, 5331, 5332,
5333, 5334, 5387, 5404, 5413, 5469,
5471, 5579, 5581, 5607, 5616, 5751,
5963, 5974, 6021, 6032, 6098, 6117,
6155, 6170, 6495, 7007, 7185, 7196,
7198, 7233, 7238, 7434, 7438, 10103,
10121, 10149, 10171, 10973, 11099,
16855, 17808, 17859, 18142, 19172
- `\exp_args:Noc` 28, [2193](#)
`\exp_args:Nof` 28, [2193](#)
`\exp_args:Noo` 28, 28, [2193](#)
`\exp_args:Nooo` 28, [2208](#)
`\exp_args:Noox` 29, [2208](#)
`\exp_args:Nox` 28, [2193](#)
`\exp_args:NV` 27,
27, [2115](#), 10101, 10119, 10147, 10169
`\exp_args:Nv` 27, 27, [2115](#)
`\exp_args:NVV` 28, [2127](#)
`\exp_args:Nx`
. . . 27, 27, 1855, [2192](#), 3150, 8930,
9547, 9583, 10105, 10123, 10151, 10173
`\exp_args:Nxo` 28, [2193](#)
`\exp_args:Nxx` 28, [2193](#)
`\exp_end:`
. . . 31, 31, 31, 31, 31, 31, 32, 260,
262, 265, 281, 282, 305, 387, 387,
393, 401, 401, 498, 604, 771, 1314,
1495, 1506, 1512, 1518, 2088, 2097,
[2299](#), 2764, 2767, 2768, 2769, 2770,
2771, 2772, 2773, 2774, 2775, 2776,
3050, 3083, 3098, 3101, 3106, 3109,
3168, 3172, 5378, 5540, 5726, 5727,
6020, 9505, 12763, 15979, 16564, 18214
`\exp_end_continue_f:nw` . . . 32, 32, [2299](#)
`\exp_end_continue_f:w`
. 32, 32, 32, 32, 32, 32,
281, 281, 577, 578, 578, 2058, 2116,
2131, 2155, 2225, 2250, 2288, [2299](#),
3544, 4565, 11298, 11492, 11813,
11922, 11940, 11961, 12032, 12037,
12046, 12061, 12119, 12152, 12159,
12191, 12563, 12588, 12629, 12636,
12670, 12772, 12775, 12785, 12786,
13140, 13142, 13146, 13148, 13244,
13258, 13268, 13377, 13378, 13379,
13510, 13521, 13545, 13553, 14511,
15219, 15389, 15580, 16246, 16250,
16295, 16299, 16344, 16363, 16367,
16400, 16404, 16471, 16597, 18088
`\exp_last_two_unbraced:Nnn`
. 29, 29, [2281](#), 8261, 8485, 8489
`\exp_last_unbraced:Nn`
. 29, 29, 29, 29,
[2244](#), 3175, 4156, 4176, 5151, 6968,
7449, 8614, 8619, 8697, 8703, 16854
`\exp_last_unbraced:Nnn`
. 29, [2244](#), 5547, 7260,
7292, 7702, 8459, 18229, 18288, 19015
`\exp_last_unbraced:Nnn`
. 29, 29, [2244](#), 7567, 7657, 17966
`\exp_last_unbraced:NNnn` 29, [2244](#)
`\exp_last_unbraced:Nnnn` 29, [2244](#)
`\exp_not:N` 30, 30, 30, 30,
178, 197, 282, 282, 322, 328, 372,
374, 374, 390, 391, 583, 732, 771,
[1310](#), 1462, 1528, 1531, 1600, 1601,
1602, 1603, 1604, 1608, 1609, 1913,
1914, 1973, 1974, [2045](#), 2081, [2285](#),
2286, 2319, 2320, 2321, 2322, 2323,
2324, 2325, 2326, 2328, 2329, 2330,
2334, 2335, 2340, 2341, 2342, 2361,
2370, 2399, 2400, 2401, 2402, 2452,
2465, 2466, 2467, 2469, 2472, 2475,
2476, 2481, 2482, 2483, 2491, 3030,
3115, 3119, 3153, 3203, 3207, 3212,
3217, 3222, 3229, 3236, 3241, 3246,
3251, 3256, 3261, 3271, 3276, 3281,
3284, 3285, 3286, 3288, 3289, 3300,
3305, 3320, 3321, 3338, 3343, 3344,
3345, 3346, 3347, 3348, 3350, 3352,
3353, 3354, 3355, 3360, 3361, 3382,
3385, 3386, 3388, 3389, 3390, 3393,
3394, 3396, 3397, 3399, 3400, 3402,
3465, 3470, 3488, 3512, 3515, 3522,
3523, 3532, 3533, 3562, 4042, 4305,
4691, 5074, 5136, 5207, 5620, 5623,
5639, 5642, 5673, 5680, 5801, 6213,
6494, 6496, 6832, 7344, 7347, 7355,
7356, 7610, 7633, 7854, 9017, 9019,
9021, 9023, 9028, 9029, 9034, 9036,
9038, 9262, 9264, 9266, 9268, 9273,
9274, 9279, 9281, 9283, 9549, 9721,
9728, 9874, 9875, 9883, 9885, 9899,
9901, 9953, 9954, 10074, 10075,
10478, 11071, 11154, 11966, 11967,
11968, 11972, 11973, 11998, 11999,
12078, 12079, 12080, 12081, 12082,
12086, 12087, 12089, 12091, 12161,
12202, 12206, 12228, 12321, 12353,
12438, 12452, 12469, 12479, 12489,
12526, 12529, 12797, 12799, 12964,
12979, 12991, 13079, 13583, 16465,
16466, 16468, 16469, 16470, 16471,
16472, 16474, 16810, 16817, 16824,
18041, 18143, 18167, 18761, 18763,
18805, 18807, 18818, 18820, 19181,
19182, 19483, 19484, 19485, 19492

`\exp_not:n` 30,
 30, 30, 30, 30, 30, 31, 31, 31, 31,
 98, 98, 99, 100, 101, 114, 118, 119,
 128, 129, 130, 134, 178, 204, 210,
 211, 212, 213, 216, 269, 369, 372,
 372, 377, 377, 377, 377, 386, 417,
 428, 431, 435, 435, 446, 738, 738,
 738, 1310, 1463, 1465, 1533, 1707,
 1709, 1857, 2045, 2241, 2285, 2406,
 2421, 3154, 3357, 3466, 3471, 3485,
 3489, 3561, 3563, 4043, 4412, 4890,
 4920, 4922, 4930, 4932, 4936, 4938,
 4948, 4950, 4952, 4954, 4956, 4958,
 4960, 4962, 4972, 4974, 4976, 4978,
 4980, 4982, 4984, 4986, 5004, 5006,
 5007, 5045, 5052, 5096, 5138, 5145,
 5204, 5208, 5209, 5210, 5465, 5467,
 5748, 5853, 6319, 6516, 6517, 6524,
 6525, 6541, 6573, 6593, 6596, 6599,
 6667, 6720, 6775, 6833, 6873, 6883,
 6977, 6978, 6993, 6995, 7026, 7092,
 7185, 7192, 7212, 7348, 7354, 7382,
 7387, 7418, 7451, 7574, 7611, 7614,
 7615, 7634, 7638, 9011, 9028, 9126,
 9256, 9273, 9560, 9662, 9664, 9686,
 9688, 9721, 9729, 9730, 9754, 9920,
 9956, 9978, 10076, 10221, 10239,
 10412, 10414, 10611, 10622, 10876,
 10967, 10984, 11065, 12073, 13584,
 15613, 16643, 16645, 16647, 16966,
 17983, 17996, 19316, 19319, 19321

`\exp_stop_f:`
 31, 31, 31, 32, 32, 32, 281,
 389, 417, 417, 417, 549, 618, 738,
 2055, 3068, 3074, 3094, 3175, 3666,
 3676, 3684, 3864, 3869, 4576, 6018,
 6058, 6064, 6092, 6528, 9517, 10806,
 10920, 11200, 11409, 11435, 11716,
 11765, 11835, 11980, 12010, 12219,
 12239, 12266, 12280, 12315, 12342,
 12351, 12370, 12386, 12402, 12420,
 12480, 12499, 12515, 13122, 13295,
 13297, 13300, 13308, 13309, 13591,
 13593, 13613, 13655, 13728, 13750,
 13804, 13805, 14145, 14163, 14307,
 14324, 14564, 14565, 14662, 14783,
 14801, 14810, 14967, 15002, 15101,
 15213, 15257, 15262, 15344, 15407,
 15440, 15454, 15469, 15484, 15499,
 15514, 15529, 15601, 15827, 16039,
 16112, 16121, 16136, 16188, 16201,
 16251, 16300, 16354, 16368, 16985,
 18345, 18357, 18572, 18587, 18642

exp internal commands:

`__exp_arg_last_unbraced:nn` .. 2221
`__exp_arg_next:Nnn` 2045, 2052
`__exp_arg_next:nnn`
 .. 2045, 2054, 2057, 2065, 2069, 2075
`__exp_eval_error_msg:w` 2079
`__exp_eval_register:N` 2070,
 2076, 2079, 2120, 2125, 2137, 2143,
 2161, 2162, 2169, 2232, 2237, 2245,
 2247, 2257, 2263, 2272, 2292, 2297
`\l__exp_internal_tl`
 32, 258, 1375, 1379,
 1380, 2045, 2064, 2065, 2241, 2242
`__exp_last_two_unbraced:nnN` . 2281
`\expandafter` 13,
 14, 21, 38, 39, 42, 43, 48, 49, 66, 67,
 90, 92, 98, 123, 146, 154, 169, 185, 365
`\expanded` 876
`\expandglyphsinfont` 948
`\expansionERROR` 2303
`\ExplFileDate`
 . 6, 19420, 19424, 19428, 19432, 19436
`\ExplFileDescription` 6
`\ExplFileName` 6
`\ExplFileVersion`
 . 6, 19420, 19424, 19428, 19432, 19436
`\ExplSyntaxOff` 3, 6,
 6, 6, 6, 7, 207, 232, 233, 233, 240, 254
`\ExplSyntaxOn` 3, 6,
 6, 6, 6, 7, 232, 233, 233, 236, 313, 373

F

false 197
`\fam` 366
`\fi` 17, 35, 41,
 51, 70, 71, 72, 95, 96, 97, 100, 101,
 130, 139, 152, 153, 170, 186, 204, 367

fi commands:

`\fi:` 21,
 35, 40, 40, 73, 73, 73, 88, 144, 144,
 145, 179, 260, 265, 267, 306, 337,
 346, 377, 379, 379, 401, 403, 405,
 557, 582, 596, 596, 622, 682, 1292,
 1336, 1459, 1472, 1492, 1542, 1547,
 1554, 1592, 1597, 1627, 1628, 1636,
 1642, 1655, 1656, 1664, 1670, 1718,
 1758, 1772, 1850, 1870, 1880, 1891,
 1943, 2004, 2036, 2084, 2087, 2094,
 2095, 2330, 2351, 2358, 2367, 2382,
 2383, 2384, 2398, 2418, 2420, 2441,
 2442, 2569, 2577, 2607, 2634, 2653,
 2778, 2780, 2782, 2784, 2786, 2788,
 2805, 2813, 2837, 2852, 2860, 3062,
 3065, 3066, 3067, 3068, 3073, 3074,

3079, 3080, 3081, 3082, 3119, 3177,
 3179, 3208, 3213, 3218, 3223, 3230,
 3237, 3242, 3247, 3252, 3257, 3262,
 3267, 3272, 3277, 3296, 3301, 3312,
 3313, 3360, 3361, 3412, 3421, 3430,
 3438, 3504, 3518, 3527, 3537, 3547,
 3669, 3692, 3709, 3710, 3712, 3758,
 3767, 3822, 3830, 3857, 3865, 3871,
 3894, 3931, 3939, 4224, 4257, 4305,
 4310, 4457, 4524, 4547, 4556, 4575,
 4579, 4593, 4778, 5001, 5220, 5221,
 5257, 5270, 5283, 5293, 5310, 5322,
 5326, 5342, 5534, 5587, 5592, 5601,
 5603, 5630, 5653, 5669, 5678, 5687,
 5693, 5700, 5705, 5716, 5720, 5728,
 5848, 5855, 5868, 5873, 5956, 5960,
 5961, 6010, 6020, 6062, 6068, 6069,
 6080, 6092, 6093, 6114, 6152, 6178,
 6303, 6309, 6321, 6322, 6347, 6359,
 6363, 6364, 6376, 6390, 6569, 6572,
 6609, 6625, 6646, 6656, 6707, 6712,
 7069, 7084, 7107, 7123, 7674, 7710,
 7812, 7814, 7824, 8845, 9667, 9691,
 9705, 9715, 9735, 9736, 9745, 10864,
 10865, 11252, 11261, 11265, 11277,
 11287, 11297, 11357, 11395, 11396,
 11397, 11398, 11399, 11400, 11401,
 11402, 11403, 11404, 11405, 11406,
 11413, 11423, 11431, 11442, 11445,
 11457, 11464, 11469, 11472, 11517,
 11623, 11696, 11697, 11706, 11707,
 11718, 11719, 11720, 11731, 11732,
 11733, 11740, 11751, 11752, 11753,
 11763, 11764, 11768, 11769, 11777,
 11780, 11781, 11789, 11799, 11812,
 11835, 11879, 11898, 11899, 11908,
 11914, 11934, 11935, 11963, 11984,
 12001, 12008, 12016, 12017, 12062,
 12063, 12064, 12127, 12143, 12165,
 12167, 12174, 12182, 12211, 12212,
 12215, 12217, 12218, 12223, 12233,
 12236, 12238, 12243, 12274, 12287,
 12292, 12298, 12301, 12302, 12336,
 12337, 12364, 12365, 12378, 12381,
 12392, 12415, 12434, 12444, 12460,
 12474, 12480, 12484, 12489, 12495,
 12510, 12521, 12539, 12549, 12551,
 12557, 12578, 12604, 12630, 12742,
 12743, 12744, 12806, 12810, 12821,
 12822, 12838, 12857, 12871, 12875,
 12881, 12893, 12922, 12947, 12957,
 12971, 12987, 12999, 13016, 13032,
 13074, 13083, 13089, 13102, 13104,
 13124, 13125, 13131, 13149, 13252,
 13263, 13275, 13276, 13277, 13284,
 13286, 13287, 13293, 13294, 13303,
 13304, 13312, 13313, 13315, 13316,
 13446, 13459, 13469, 13470, 13475,
 13476, 13477, 13478, 13479, 13480,
 13487, 13496, 13501, 13525, 13528,
 13530, 13537, 13566, 13569, 13570,
 13575, 13597, 13598, 13604, 13617,
 13635, 13637, 13647, 13661, 13691,
 13700, 13734, 13756, 13774, 13791,
 13808, 13809, 13811, 13812, 13816,
 13831, 13864, 13893, 13894, 13895,
 13896, 13897, 13910, 13952, 14025,
 14090, 14092, 14093, 14103, 14132,
 14135, 14136, 14147, 14167, 14230,
 14231, 14232, 14244, 14283, 14284,
 14285, 14286, 14292, 14295, 14297,
 14307, 14324, 14540, 14544, 14546,
 14550, 14557, 14558, 14568, 14569,
 14572, 14664, 14733, 14744, 14756,
 14782, 14789, 14800, 14816, 14823,
 14884, 14940, 14950, 14952, 14962,
 14980, 14981, 15013, 15016, 15025,
 15027, 15029, 15048, 15062, 15063,
 15083, 15104, 15105, 15118, 15134,
 15150, 15166, 15182, 15198, 15212,
 15220, 15226, 15237, 15240, 15249,
 15259, 15261, 15267, 15274, 15277,
 15286, 15294, 15315, 15348, 15349,
 15376, 15378, 15392, 15393, 15411,
 15422, 15431, 15434, 15435, 15444,
 15447, 15464, 15479, 15494, 15509,
 15524, 15539, 15542, 15544, 15561,
 15606, 15833, 15869, 15870, 15880,
 15921, 15922, 15946, 15978, 15984,
 15985, 15988, 15990, 15991, 15995,
 16007, 16026, 16031, 16038, 16041,
 16073, 16083, 16084, 16094, 16116,
 16131, 16149, 16157, 16160, 16188,
 16196, 16212, 16250, 16272, 16284,
 16299, 16320, 16353, 16367, 16373,
 16386, 16387, 16425, 16426, 16522,
 16523, 16539, 16563, 16574, 16575,
 16591, 16729, 16741, 16742, 16798,
 16868, 16877, 16888, 16897, 16906,
 16941, 16951, 16961, 17069, 17071,
 17974, 18713, 18714, 18717, 19181,
 19182, 19183, 19193, 19200, 19202

file commands:
 \file_add_path:nN 173,
 173, 179, 10630, 10672, 10784, 10796
 \g_file_current_name_tl 173, 8880,
 10555, 10573, 10708, 10709, 10712
 \file_if_exist:nTF

- .. [173](#), [173](#), [173](#), [173](#), [179](#), [10670](#),
[10684](#), [17757](#), [17762](#), [17770](#), [17775](#)
- \file_if_exist_input:n [205](#), [205](#)
- \file_if_exist_input:nTF
..... [205](#), [205](#), [17755](#), [17760](#), [17768](#)
- \file_input:n
.. [173](#), [173](#), [173](#), [174](#), [205](#), [205](#), [10677](#)
- \file_list: [174](#), [174](#), [10727](#)
- \file_path_include:n
..... [173](#), [173](#), [173](#), [205](#), [10715](#)
- \file_path_remove:n . [174](#), [174](#), [10715](#)
- file internal commands:
- __file_add_path:nN [10630](#)
- __file_add_path_search:nN ... [10630](#)
- __file_if_exist:nTF
..... [179](#), [10677](#), [18139](#), [18164](#)
- __file_input:n
.. [10677](#), [17758](#), [17765](#), [17771](#), [17778](#)
- __file_input_aux:n [10677](#)
- \g__file_internal_ior [179](#), [10634](#),
[10635](#), [10638](#), [10655](#), [10656](#), [10880](#)
- \l__file_internal_name_tl
..... [179](#), [528](#), [10577](#),
[10593](#), [10594](#), [10595](#), [10600](#), [10607](#),
[10608](#), [10610](#), [10611](#), [10617](#), [10622](#),
[10672](#), [10673](#), [10680](#), [10784](#), [10785](#),
[10787](#), [10796](#), [10797](#), [10800](#), [17758](#),
[17765](#), [17771](#), [17778](#), [18147](#), [18170](#)
- \l__file_internal_seq [10582](#), [10648](#),
[10650](#), [10729](#), [10735](#), [10740](#), [10742](#)
- \l__file_internal_tl
.. [10576](#), [10590](#), [10591](#), [10711](#), [10712](#)
- __file_name_sanitiz:nn
... [179](#), [179](#), [10585](#), [10631](#), [10687](#),
[10716](#), [10724](#), [10780](#), [10790](#), [10923](#)
- __file_name_sanitiz_aux:n .. [10585](#)
- __file_path_include:n [10715](#)
- \g__file_record_seq [530](#), [532](#),
[532](#), [10568](#), [10701](#), [10706](#), [10729](#), [10749](#)
- \l__file_saved_search_path_seq ..
..... [10579](#), [10646](#), [10666](#)
- \l__file_search_path_seq
.... [10578](#), [10647](#), [10649](#), [10650](#),
[10653](#), [10665](#), [10719](#), [10720](#), [10725](#)
- \g__file_stack_seq
..... [530](#), [10567](#), [10708](#), [10711](#)
- \finalhyphendemerits [368](#)
- \firstmark [369](#)
- \firstmarks [626](#)
- \firstvalidlanguage [877](#)
- \floatingpenalty [370](#)
- floor [193](#)
- \fmtname [145](#)
- \font [371](#)
- \fontchardp [627](#)
- \fontcharht [628](#)
- \fontcharic [629](#)
- \fontcharwd [630](#)
- \fontdimen [372](#)
- \fontid [878](#)
- \fontname [373](#)
- \forcecjktoken [1155](#)
- \formatname [879](#)
- fp commands:
- \c_e_fp [188](#), [190](#), [16683](#)
- \fp_abs:n
.. [192](#), [197](#), [197](#), [720](#), [16445](#), [17279](#),
[17359](#), [17361](#), [17363](#), [17707](#), [17709](#)
- \fp_add:Nn [183](#), [183](#), [720](#), [16665](#)
- \fp_compare:nNnTF
.... [185](#), [185](#), [186](#), [186](#), [186](#), [186](#),
[8322](#), [13254](#), [13349](#), [13355](#), [13360](#),
[13368](#), [13412](#), [13418](#), [16584](#), [16598](#),
[16627](#), [17145](#), [17147](#), [17152](#), [17378](#),
[17387](#), [17691](#), [19561](#), [19567](#), [19839](#)
- \fp_compare:nTF
.... [185](#), [185](#), [186](#), [186](#), [186](#), [187](#),
[192](#), [13241](#), [13321](#), [13327](#), [13332](#), [13340](#)
- \fp_compare_p:n [185](#), [185](#), [13241](#)
- \fp_compare_p:nNn ... [185](#), [185](#), [13254](#)
- \fp_const:Nn [182](#),
[182](#), [16642](#), [16683](#), [16684](#), [16685](#), [16686](#)
- \fp_do_until:nn [186](#), [186](#), [13318](#)
- \fp_do_until:nNnn ... [186](#), [186](#), [13346](#)
- \fp_do_while:nn [186](#), [186](#), [13318](#)
- \fp_do_while:nNnn ... [186](#), [186](#), [13346](#)
- \fp_eval:n
[183](#), [183](#), [185](#), [191](#), [191](#), [191](#), [192](#),
[192](#), [192](#), [192](#), [192](#), [192](#), [192](#), [192](#),
[192](#), [192](#), [192](#), [192](#), [193](#), [193](#), [193](#),
[193](#), [193](#), [193](#), [194](#), [194](#), [194](#), [194](#),
[194](#), [194](#), [194](#), [194](#), [194](#), [194](#), [194](#),
[194](#), [194](#), [194](#), [194](#), [194](#), [194](#), [194](#),
[194](#), [194](#), [195](#), [195](#), [195](#), [195](#), [195](#),
[195](#), [195](#), [195](#), [195](#), [196](#), [196](#), [196](#),
[197](#), [197](#), [726](#), [16442](#), [19572](#), [19587](#),
[19589](#), [19679](#), [19697](#), [19698](#), [19699](#),
[19700](#), [19712](#), [19713](#), [19721](#), [19722](#),
[19729](#), [19736](#), [19738](#), [19744](#), [19745](#),
[19746](#), [19757](#), [19762](#), [19768](#), [19769](#),
[19841](#), [19858](#), [19859](#), [20036](#), [20058](#),
[20059](#), [20070](#), [20071](#), [20083](#), [20084](#),
[20096](#), [20098](#), [20103](#), [20114](#), [20126](#),
[20137](#), [20138](#), [20230](#), [20245](#), [20246](#),
[20447](#), [20465](#), [20466](#), [20467](#), [20482](#),
[20500](#), [20501](#), [20502](#), [20551](#), [20552](#)
- \fp_flag_off:n [189](#), [189](#), [11507](#)

- \fp_flag_on:n [189](#), [189](#), [11509](#), [11553](#),
[11562](#), [11570](#), [11587](#), [11596](#), [11627](#)
- \fp_format:nn [198](#)
- \fp_function:Nw [13154](#)
- \fp_gadd:Nn [183](#), [16665](#)
- .fp_gset:N [164](#), [10132](#)
- \fp_gset:Nn .. [182](#), [16642](#), [16666](#), [16668](#)
- \fp_gset_eq:NN [183](#), [16651](#), [16656](#)
- \fp_gsub:Nn [183](#), [16665](#)
- \fp_gzero:N [182](#), [16655](#), [16662](#)
- \fp_gzero_new:N [182](#), [16659](#)
- \fp_if_exist:Ntf
.. [184](#), [184](#), [13239](#), [16660](#), [16662](#), [16677](#)
- \fp_if_exist_p:N [184](#), [184](#), [13239](#)
- \fp_if_flag_on:nTF .. [189](#), [189](#), [11511](#)
- \fp_if_flag_on_p:n .. [189](#), [189](#), [11511](#)
- \fp_if_nan:nTF [198](#)
- \fp_log:N [207](#), [207](#), [17819](#)
- \fp_log:n [207](#), [207](#), [17819](#)
- \fp_max:nn [197](#), [197](#), [16447](#)
- \fp_min:nn [197](#), [16447](#)
- \fp_new:N [182](#), [182](#), [182](#),
[7974](#), [7975](#), [16639](#), [16660](#), [16662](#),
[16687](#), [16688](#), [16689](#), [16690](#), [17115](#),
[17116](#), [17117](#), [17239](#), [17240](#), [17522](#),
[17523](#), [17668](#), [17669](#), [19580](#), [19581](#)
- \fp_new_function:Npn [13161](#)
- .fp_set:N [164](#), [10132](#)
- \fp_set:Nn [182](#), [182](#),
[8318](#), [8320](#), [16642](#), [16665](#), [16667](#),
[17132](#), [17133](#), [17134](#), [17247](#), [17249](#),
[17287](#), [17304](#), [17321](#), [17335](#), [17337](#),
[17352](#), [17353](#), [17532](#), [17533](#), [17674](#),
[17676](#), [17701](#), [17702](#), [19560](#), [19563](#)
- \fp_set_eq:NN [183](#),
[183](#), [16651](#), [16655](#), [17292](#), [17309](#), [17323](#)
- \fp_show:N . [189](#), [189](#), [762](#), [16675](#), [17820](#)
- \fp_show:n [189](#), [189](#), [16675](#), [17822](#)
- \fp_step_function:nnnN
..... [187](#), [187](#), [13374](#), [13432](#)
- \fp_step_inline:nnnn [187](#), [187](#), [13427](#)
- \fp_sub:Nn [183](#), [183](#), [16665](#)
- \fp_to_decimal:N [183](#),
[183](#), [184](#), [11500](#), [16289](#), [16391](#), [16442](#)
- \fp_to_decimal:n
[183](#), [183](#), [183](#), [183](#), [184](#), [718](#), [16289](#),
[16394](#), [16444](#), [16446](#), [16448](#), [16450](#)
- \fp_to_dim:N [183](#), [183](#), [16390](#)
- \fp_to_dim:n [183](#), [183](#), [188](#),
[8328](#), [8354](#), [16390](#), [17177](#), [17188](#),
[17279](#), [17605](#), [17613](#), [17717](#), [17719](#)
- \fp_to_int:N [184](#), [184](#), [16395](#)
- \fp_to_int:n [184](#),
[184](#), [723](#), [16395](#), [16599](#), [16600](#), [17854](#)
- \fp_to_int_dispatch:w [718](#)
- \fp_to_scientific:N
..... [184](#), [184](#), [11501](#), [16240](#)
- \fp_to_scientific:n
..... [184](#), [184](#), [184](#), [16240](#)
- \fp_to_tl:N ... [184](#), [184](#), [16358](#), [16678](#)
- \fp_to_tl:n [184](#), [184](#),
[11214](#), [11552](#), [11561](#), [11586](#), [11595](#),
[11624](#), [13404](#), [13415](#), [16358](#), [16681](#)
- \fp_trap:nn [188](#), [189](#), [189](#),
[562](#), [11523](#), [11638](#), [11639](#), [11640](#), [11641](#)
- \fp_until_do:nn [186](#), [186](#), [13318](#)
- \fp_until_do:nnnn [186](#), [186](#), [13346](#)
- \fp_use:N
.. [184](#), [184](#), [16442](#), [19566](#), [19570](#), [19575](#)
- \fp_while_do:nn [187](#), [187](#), [13318](#)
- \fp_while_do:nnnn [186](#), [186](#), [13346](#)
- \fp_zero:N [182](#), [182](#), [16655](#), [16660](#), [19562](#)
- fp_zero:N [182](#)
- \fp_zero_new:N [182](#), [182](#), [16659](#)
- \c_inf_fp [187](#),
[196](#), [11223](#), [12638](#), [13907](#), [13987](#),
[15009](#), [15244](#), [15248](#), [15270](#), [15546](#)
- \c_nan_fp [196](#), [565](#), [587](#), [11223](#), [11563](#),
[11571](#), [11643](#), [11823](#), [11842](#), [11848](#),
[12032](#), [12037](#), [12046](#), [12152](#), [12191](#),
[12629](#), [12639](#), [12894](#), [13191](#), [13408](#),
[15216](#), [15977](#), [16486](#), [16548](#), [16562](#)
- \c_one_fp [187](#), [12642](#), [13107](#),
[13127](#), [13484](#), [15003](#), [15211](#), [15260](#),
[15470](#), [15500](#), [15971](#), [16557](#), [16683](#)
- \c_pi_fp .. [188](#), [196](#), [596](#), [12640](#), [16685](#)
- \g_tmpa_fp [188](#), [16687](#)
- \l_tmpa_fp [188](#), [16687](#)
- \g_tmpb_fp [188](#), [16687](#)
- \l_tmpb_fp [188](#), [16687](#)
- \c_zero_fp [187](#), [627](#), [726](#),
[11223](#), [11270](#), [12643](#), [13119](#), [13130](#),
[13486](#), [13737](#), [13903](#), [15012](#), [15239](#),
[15273](#), [16193](#), [16417](#), [16640](#), [16655](#),
[16656](#), [17145](#), [17147](#), [17152](#), [17378](#),
[17387](#), [17691](#), [19561](#), [19567](#), [19839](#)
- fp internal commands:
 fp&_o:ww [616](#), [623](#), [13489](#)
 _fp*_o:ww [13868](#)
 _fp+_o:ww [625](#),
 [626](#), [626](#), [626](#), [626](#), [626](#), [654](#), [13586](#)
 _fp._o:ww [607](#), [12860](#)
 _fp-_o:ww [625](#), [626](#), [626](#), [13581](#)
 _fp/_o:ww [635](#), [635](#), [674](#), [13978](#)
 _fp^_o:ww [15207](#)
 _fp_acos_o:w [711](#), [713](#), [16134](#)
 _fp_acot_o:Nw . [12692](#), [12694](#), [15957](#)
 _fp_acotii_o:Nww [15960](#), [15965](#), [15981](#)

__fp_acotii_o:ww 706
 __fp_acsc_normal_o:NnwNnw
 713, 16192, 16207, 16215
 __fp_acsc_o:w 16186
 __fp_add:NNNn 16665
 __fp_add_big_i:wNww 628
 __fp_add_big_i_o:wNww
 625, 629, 13657, 13664, 15035
 __fp_add_big_ii:wNww 628
 __fp_add_big_ii_o:wNww 13660, 13664
 __fp_add_inf_o:Nww ... 13601, 13621
 __fp_add_normal_o:Nww
 628, 13600, 13641
 __fp_add_npos_o:NnwNnw
 628, 13644, 13650
 __fp_add_return_ii_o:Nww
 13603, 13609, 13614
 __fp_add_significand_carry_-
 o:wwwNN 630, 13697, 13712
 __fp_add_significand_no_carry_-
 o:wwwNN 630, 13699, 13702
 __fp_add_significand_o:NnnwnnnNw
 629, 629, 13667, 13675, 13680
 __fp_add_significand_pack:NNNNNN
 13680
 __fp_add_significand_test_o:N 13680
 __fp_add_zeros_o:Nww . 13599, 13611
 __fp_and_return:wNw 13489
 __fp_array_count:n
 11474, 11807, 15970, 16556
 __fp_array_count_loop:Nw 11474
 __fp_array_to_clist:n
 .. 11868, 12954, 12955, 16451, 16589
 __fp_array_to_clist_loop:Nw . 16451
 __fp_asec_o:w 16199
 __fp_asin_auxi_o:NnNww
 16164, 16167, 16226
 __fp_asin_auxi_o:nNww . 711, 711, 713
 __fp_asin_isqrt:wn 16167
 __fp_asin_normal_o:NnwNnnnw ...
 16125, 16141, 16152
 __fp_asin_o:w 16119
 __fp_atan_auxi:ww . 708, 16044, 16058
 __fp_atan_auxii:w 16058
 __fp_atan_combine_aux:ww 16085
 __fp_atan_combine_o:NwwwwN ...
 707, 708, 16004, 16021, 16085
 __fp_atan_dispatch_o:NNnNw .. 15957
 __fp_atan_div:wnwnw
 708, 16032, 16034
 __fp_atan_inf_o:NNNw
 706, 706, 15992,
 15993, 15994, 16002, 16137, 16210
 __fp_atan_near:wwn 16034
 __fp_atan_near_aux:wwn 16034
 __fp_atan_normal_o:NNnwNnw
 706, 15996, 16012
 __fp_atan_o:Nw . 12696, 12698, 15957
 __fp_atan_Taylor_break:w 16069
 __fp_atan_Taylor_loop:www
 709, 16064, 16069
 __fp_atan_test_o:NwwNwwN
 712, 16015, 16019, 16174
 __fp_atanii_o:Nww 15960, 15965, 15981
 __fp_basics_pack_high:NNNNNw ...
 ... 630, 646, 13560, 13705, 13856,
 13957, 13969, 14109, 14302, 14761
 __fp_basics_pack_high_carry:w ..
 625, 13560
 __fp_basics_pack_low:NNNNNw ...
 636, 646,
 13560, 13707, 13858, 13959, 13971,
 14111, 14251, 14253, 14304, 14763
 __fp_basics_pack_weird_high:NNNNNNNw
 199, 13571, 13716, 14120
 __fp_basics_pack_weird_low:NNNNw
 199, 13571, 13718, 14122
 \c__fp_big_leading_shift_int ...
 .. 11335, 14181, 14451, 14461, 14471
 \c__fp_big_middle_shift_int
 11335, 14184, 14187, 14190,
 14193, 14196, 14199, 14203, 14453,
 14463, 14473, 14482, 14485, 14488
 \c__fp_big_trailing_shift_int ...
 11335, 14207, 14495
 \c__fp_Bigg_leading_shift_int ...
 11340, 14031, 14048
 \c__fp_Bigg_middle_shift_int ...
 .. 11340, 14034, 14037, 14051, 14054
 \c__fp_Bigg_trailing_shift_int ..
 11340, 14040, 14057
 __fp_case_return:nw
 11396, 11436, 11439, 11444, 11928,
 14968, 15992, 15993, 15994, 16252,
 16301, 16369, 16371, 16372, 16417
 __fp_case_return_i_o:ww . 11403,
 13602, 13616, 13625, 13901, 15984
 __fp_case_return_ii_o:ww
 .. 11403, 13902, 15258, 15276, 15985
 __fp_case_return_o:Nw
 558, 11397, 15003,
 15008, 15011, 15211, 15216, 15239,
 15248, 15470, 15500, 16193, 16195
 __fp_case_return_o:Nww
 11401, 13903, 13904,
 13907, 13908, 15260, 15269, 15272
 __fp_case_return_same_o:w
 . 558, 11399, 14132, 14136, 14788,

- 15015, 15236, 15455, 15463, 15478,
- 15493, 15508, 15515, 15523, 15538,
- 16122, 16130, 16148, 16194, 16211
- __fp_case_use:nw 11395, 13627, 13899,
- 13900, 13905, 13906, 13986, 13989,
- 14134, 14781, 14784, 15242, 15456,
- 15461, 15471, 15476, 15486, 15491,
- 15501, 15506, 15516, 15521, 15531,
- 15536, 16124, 16127, 16137, 16139,
- 16145, 16189, 16191, 16202, 16205,
- 16210, 16255, 16266, 16304, 16314
- __fp_chk:w 547, 547, 548, 548,
- 548, 549, 549, 549, 549, 550, 596,
- 596, 626, 628, 628, 628, 630, 636,
- 636, 639, 639, 11210, 11223, 11224,
- 11225, 11226, 11227, 11230, 11232,
- 11235, 11241, 11245, 11266, 11269,
- 11271, 11281, 11291, 11304, 11323,
- 11407, 11433, 11619, 11624, 11825,
- 11871, 11880, 11882, 12649, 13246,
- 13271, 13272, 13391, 13404, 13408,
- 13448, 13449, 13452, 13463, 13464,
- 13472, 13473, 13481, 13492, 13495,
- 13507, 13533, 13587, 13606, 13607,
- 13609, 13610, 13611, 13619, 13622,
- 13638, 13639, 13641, 13650, 13726,
- 13877, 13911, 13912, 13915, 13994,
- 14130, 14138, 14140, 14317, 14321,
- 14778, 14790, 14792, 15000, 15017,
- 15019, 15208, 15227, 15229, 15230,
- 15233, 15245, 15250, 15253, 15278,
- 15279, 15281, 15297, 15382, 15395,
- 15397, 15401, 15405, 15452, 15465,
- 15467, 15480, 15482, 15495, 15497,
- 15510, 15512, 15525, 15527, 15540,
- 15550, 15982, 15997, 15998, 16002,
- 16013, 16119, 16132, 16134, 16150,
- 16153, 16163, 16186, 16197, 16199,
- 16213, 16215, 16220, 16248, 16273,
- 16276, 16297, 16321, 16324, 16365,
- 16388, 16438, 16439, 16566, 16568
- __fp_compare:wNNNNw 13035
- __fp_compare_aux:wn 13254
- __fp_compare_back:ww 617, 617, 13122, 13267, 13270, 13462
- __fp_compare_nan:w 617, 13270
- __fp_compare_npos:nwnw 616,
- 617, 618, 13281, 13298, 13728, 14564
- __fp_compare_return:w 13241
- __fp_compare_significand:nnnnnnnn 13298
- __fp_cos_o:w 15467
- __fp_cot_o:w 693, 15527
- __fp_cot_zero_o:Nnw 692, 693, 15485, 15527
- __fp_csc_o:w 15482
- __fp_decimate:nNnnnn 558,
- 11349, 11418, 11452, 11884, 13666,
- 13674, 13753, 15051, 15055, 16330
- __fp_decimate_:Nnnnn 11361
- __fp_decimate_auxi:Nnnnn 11365
- __fp_decimate_auxii:Nnnnn ... 11365
- __fp_decimate_auxiii:Nnnnn .. 11365
- __fp_decimate_auxiv:Nnnnn ... 11365
- __fp_decimate_auxix:Nnnnn ... 11365
- __fp_decimate_auxv:Nnnnn 11365
- __fp_decimate_auxvi:Nnnnn ... 11365
- __fp_decimate_auxvii:Nnnnn .. 11365
- __fp_decimate_auxviii:Nnnnn . 11365
- __fp_decimate_auxix:Nnnnn 11365
- __fp_decimate_auxxi:Nnnnn ... 11365
- __fp_decimate_auxxii:Nnnnn .. 11365
- __fp_decimate_auxxiii:Nnnnn . 11365
- __fp_decimate_auxxiv:Nnnnn .. 11365
- __fp_decimate_auxxv:Nnnnn ... 11365
- __fp_decimate_auxxvi:Nnnnn .. 11365
- __fp_decimate_pack:nnnnnnnnnw .
- 556, 11372, 11391
- __fp_decimate_pack:nnnnnnnw 11392, 11393
- __fp_decimate_tiny:Nnnnn 11361
- __fp_div_npos_o:Nww 638, 639, 13983, 13993
- __fp_div_significand_calc:wwnnnnnnnn 642,
- 642, 14010, 14019, 14065, 14861, 14868
- __fp_div_significand_calc_-
- i:wwnnnnnnnn 14019
- __fp_div_significand_calc_-
- ii:wwnnnnnnnn 14019
- __fp_div_significand_i_o:wnnw ..
- 639, 642, 14000, 14006
- __fp_div_significand_ii:wwn 644, 14014, 14015, 14016, 14061
- __fp_div_significand_iii:wwnnnnnn 644, 14017, 14068
- __fp_div_significand_iv:wwnnnnnnnn 644, 14071, 14076
- __fp_div_significand_large_-
- o:wwNNNNwN 646, 14102, 14116
- __fp_div_significand_pack:NNN ..
- 645, 645,
- 645, 676, 676, 676, 676, 676, 676,
- 676, 14063, 14096, 14848, 14866, 14874
- __fp_div_significand_small_-
- o:wwNNNNwN 646, 14100, 14106

- __fp_div_significand_test_o:w .. [645](#), [646](#), [14008](#), [14097](#)
- __fp_div_significand_v:NN [14081](#), [14083](#), [14086](#)
- __fp_div_significand_v:NNw .. [14076](#)
- __fp_div_significand_vi:Nw [645](#), [14076](#)
- \l__fp_division_by_zero_flag_token [11519](#)
- __fp_division_by_zero_o:Nnw ... [562](#), [11583](#), [11631](#), [14785](#), [15546](#), [15547](#)
- __fp_division_by_zero_o:NNww ... [562](#), [11591](#), [11631](#), [13987](#), [13990](#), [15244](#)
- __fp_ep_compare:www ... [14559](#), [16028](#)
- __fp_ep_compare_aux:www ... [14559](#)
- __fp_ep_div:wwwwn [704](#), [14589](#), [14700](#), [15947](#), [16043](#), [16047](#), [16056](#), [16223](#)
- __fp_ep_div_eps_pack:NNNNw .. [14619](#)
- __fp_ep_div_epsilon:wnNNNNn [666](#)
- __fp_ep_div_epsilon:wnNNNNNn [14616](#), [14619](#)
- __fp_ep_div_epsilon:wnNNNNNNn .. [14619](#)
- __fp_ep_div_esti:wwwwn [665](#), [14595](#), [14598](#)
- __fp_ep_div_estii:wwnnwn ... [14598](#)
- __fp_ep_div_estiii:NNNNNwwwn .. [14598](#)
- __fp_ep_inv_to_float:wN [694](#)
- __fp_ep_inv_to_float:wwN [702](#), [14696](#), [14704](#), [15489](#), [15504](#)
- __fp_ep_isqrt:wN [14642](#), [16184](#)
- __fp_ep_isqrt_aux:wN [14642](#)
- __fp_ep_isqrt_auxi:wN [14645](#), [14647](#)
- __fp_ep_isqrt_auxii:wwnnwn .. [14642](#)
- __fp_ep_isqrt_epsilon:wN [668](#), [14679](#), [14682](#)
- __fp_ep_isqrt_epsilon:wwN [14682](#)
- __fp_ep_isqrt_esti:wwnnwn [14657](#), [14660](#)
- __fp_ep_isqrt_estii:wwnnwn .. [14660](#)
- __fp_ep_isqrt_estiii:NNNNNwwwn . [14660](#)
- __fp_ep_mul:wwwwn [14574](#), [15904](#), [15934](#), [16171](#), [16182](#)
- __fp_ep_mul_raw:wwwN [14574](#), [15568](#), [15854](#)
- __fp_ep_to_ep:wwN . [14525](#), [14576](#), [14579](#), [14591](#), [14594](#), [14644](#), [16172](#)
- __fp_ep_to_ep_end:www [14525](#)
- __fp_ep_to_ep_loop:N [701](#), [14525](#), [15855](#)
- __fp_ep_to_ep_zero:ww [14525](#)
- __fp_ep_to_fixed:wwn [14507](#), [15565](#), [16050](#), [16059](#), [16169](#)
- __fp_ep_to_fixed_auxi:www ... [14507](#)
- __fp_ep_to_fixed_auxii:nnnnnnwn [14507](#)
- __fp_ep_to_float:wN [694](#)
- __fp_ep_to_float:wwN [691](#), [702](#), [14696](#), [14708](#), [15459](#), [15474](#), [15953](#)
- __fp_error:nnnn [11552](#), [11560](#), [11569](#), [11586](#), [11594](#), [11622](#), [11645](#), [11818](#), [11820](#), [11841](#), [11846](#), [12953](#), [13403](#), [13414](#)
- __fp_exp_after_?_f:nw ... [584](#), [12020](#)
- __fp_exp_after_array_f:w [11324](#), [12611](#), [13511](#), [13522](#), [13546](#), [13554](#)
- __fp_exp_after_f:nw [584](#), [11271](#), [12648](#), [12790](#)
- __fp_exp_after_mark_f:nw [584](#), [12020](#)
- __fp_exp_after_normal:nnNw [11274](#), [11284](#), [11294](#), [11311](#)
- __fp_exp_after_normal:Nwwww ... [11313](#), [11321](#)
- __fp_exp_after_o:nw [11271](#)
- __fp_exp_after_o:w .. [552](#), [11271](#), [11400](#), [11404](#), [11406](#), [11878](#), [11922](#), [11940](#), [12767](#), [13141](#), [13480](#), [13497](#), [13610](#), [14319](#), [15394](#), [15399](#), [16633](#)
- __fp_exp_after_special:nnNw ... [11276](#), [11286](#), [11296](#), [11301](#)
- __fp_exp_after_stop_f:nw [11324](#)
- __fp_exp_large:w [15095](#)
- __fp_exp_large:wN [15095](#)
- __fp_exp_large_after:wwn [15095](#)
- __fp_exp_large_i:wN [15095](#)
- __fp_exp_large_ii:wN [15095](#)
- __fp_exp_large_iii:wN [15095](#)
- __fp_exp_large_iv:wN [15095](#)
- __fp_exp_large_v:wN .. [15095](#), [15381](#)
- __fp_exp_normal:w [15005](#), [15019](#)
- __fp_exp_o:w [15000](#)
- __fp_exp_overflow: ... [15041](#), [15066](#)
- __fp_exp_pos:NNwnw [15022](#), [15024](#), [15027](#)
- __fp_exp_pos:Nnwnw [15019](#)
- __fp_exp_pos_large:NnnNwn [15056](#), [15095](#)
- __fp_exp_Taylor:Nnnwn [15052](#), [15068](#), [15204](#)
- __fp_exp_Taylor_break:Nww ... [15068](#)
- __fp_exp_Taylor_ii:ww . [15074](#), [15077](#)
- __fp_exp_Taylor_loop:www [15068](#)
- __fp_expand:n [720](#), [11483](#), [16455](#)
- __fp_expand_loop:nwnN [11483](#)
- __fp_exponent:w [11245](#)
- __fp_fixed_add:nnNnnwn [14401](#)
- __fp_fixed_add:Nnnnnwn [14401](#)


```

\__fp_fixed_add:wnn .....
... 655, 655, 14401, 14640, 14942,
14950, 14961, 14979, 16055, 16115
\__fp_fixed_add_after:NNNNNwn . 14401
\__fp_fixed_add_one:wN .....
.. 14333, 14633, 15085, 15094, 16181
\__fp_fixed_add_pack:NNNNNwn . 14401
\__fp_fixed_continue:wn .....
14332, 14577, 14582, 14592, 15108,
15124, 15140, 15156, 15172, 15188,
15357, 15603, 15892, 16173, 16182
\__fp_fixed_div_int:wnN ..... 14370
\__fp_fixed_div_int:wwN .....
..... 14370, 14941, 15084, 16074
\__fp_fixed_div_int_after:Nw ...
..... 657, 14370
\__fp_fixed_div_int_auxi:wnn . 14370
\__fp_fixed_div_int_auxii:wnn ...
..... 657, 14370
\__fp_fixed_div_int_pack:Nw ....
..... 657, 657, 657, 657, 657, 14370
\__fp_fixed_div_myriad:wn .....
..... 14338, 14637
\__fp_fixed_inv_to_float:wN ....
..... 14703, 15024, 15293
\__fp_fixed_mul:nnnnnnnw ..... 14421
\__fp_fixed_mul:wnn .....
..... 655, 656, 700, 702, 14421,
14586, 14617, 14632, 14634, 14638,
14691, 14694, 14707, 14943, 14953,
14993, 15086, 15105, 15205, 15303,
15861, 15915, 16062, 16095, 16097
\__fp_fixed_mul_add:nnnnwnnnn ...
..... 14489, 14491
\__fp_fixed_mul_add:nnnnwnnwN ...
..... 14496, 14502
\__fp_fixed_mul_add:Nwnnnwnnn ...
..... 14454, 14464, 14474, 14478
\__fp_fixed_mul_add:wwn ..... 14448
\__fp_fixed_mul_after:wnn .....
..... 659, 14340, 14346, 14349,
14423, 14450, 14460, 14470, 15320
\__fp_fixed_mul_one_minus_-
mul:wnn ..... 14448
\__fp_fixed_mul_short:wnn 14347,
14615, 14636, 14678, 14680, 16108
\__fp_fixed_mul_sub_back:wwn ...
..... 14448,
14692, 15882, 15884, 15885, 15886,
15887, 15888, 15889, 15890, 15891,
15895, 15897, 15898, 15899, 15900,
15901, 15902, 15903, 15928, 15930,
15931, 15932, 15933, 15936, 15938,
15939, 15940, 15941, 16075, 16083
\__fp_fixed_one_minus_mul:wnn ...
..... 660, 661, 14468
\__fp_fixed_sub:wnn .....
..... 14401, 14684, 14959, 14975,
14987, 15607, 16056, 16113, 16179
\__fp_fixed_to_float:Nw 14710, 14968
\__fp_fixed_to_float:wN .....
..... 655, 710, 14697, 14710,
14988, 14998, 15022, 15289, 16103
\__fp_fixed_to_float_pack:ww ...
..... 14742, 14752
\__fp_fixed_to_float_rad:wN ....
..... 14705, 16103
\__fp_fixed_to_float_round-
up:wnnnnw ..... 14755, 14759
\__fp_fixed_to_float_zero:w ....
..... 14738, 14747
\__fp_fixed_to_loop:N .....
..... 14715, 14725, 14729
\__fp_fixed_to_loop_end:w .....
..... 14731, 14735
\__fp_from_dim:wNNnnnnnn ..... 16407
\__fp_from_dim:wnnnnwnN 16434, 16435
\__fp_from_dim:wnnnnwNw ..... 16407
\__fp_from_dim:wNw ..... 16407
\__fp_from_dim_test:ww .....
..... 719, 12097, 12110, 12667, 16407
\__fp_function_apply:nw .....
.. 614, 614, 615, 13156, 13172, 13194
\__fp_function_args:Nwn .. 614, 13161
\__fp_function_store:wwNwnn ...
..... 615, 13194
\__fp_function_store_end:wnnn ...
..... 615, 13194
\__fp_inf_fp:N ..... 11229, 11607
\__fp_int:wTF ..... 11407, 16568
\__fp_int_normal:nnnn ..... 11407
\__fp_int_p:w ..... 11407
\__fp_int_test:Nw ..... 11407
\__fp_invalid_operation:nnw ....
..... 562, 562, 11549, 11631,
11643, 16257, 16268, 16306, 16316
\l__fp_invalid_operation_flag_-
token ..... 11519
\__fp_invalid_operation_o:nw 562,
11642, 14134, 14781, 15462, 15477,
15492, 15507, 15522, 15537, 16128,
16146, 16162, 16190, 16203, 16219
\__fp_invalid_operation_o:Nww ...
..... 562, 11557, 11631,
13630, 13633, 13905, 13906, 15388
\__fp_invalid_operation_tl_o:nn .
..... 562, 11566, 11631, 11866, 16588

```


\c__fp_leading_shift_int
 [11331](#), [14341](#),
 [14350](#), [14424](#), [15321](#), [15802](#), [15839](#)
 __fp_ln_c:NwNn [677](#)
 __fp_ln_c:NwNw ... [678](#), [14925](#), [14956](#)
 __fp_ln_div_after:Nw
 [676](#), [676](#), [14828](#), [14877](#)
 __fp_ln_div_i:w [14850](#), [14859](#)
 __fp_ln_div_ii:wn
 .. [14853](#), [14854](#), [14855](#), [14856](#), [14864](#)
 __fp_ln_div_vi:wn [14857](#), [14872](#)
 __fp_ln_exponent:wn [678](#), [14804](#), [14965](#)
 __fp_ln_exponent_one:ww [14970](#), [14984](#)
 __fp_ln_exponent_small:NNww ...
 [14973](#), [14977](#), [14990](#)
 \c__fp_ln_i_fixed_tl [14769](#)
 \c__fp_ln_ii_fixed_tl [14769](#)
 \c__fp_ln_iii_fixed_tl [14769](#)
 \c__fp_ln_iv_fixed_tl [14769](#)
 \c__fp_ln_ix_fixed_tl [14769](#)
 __fp_ln_npos_o:w
 [671](#), [672](#), [14790](#), [14792](#)
 __fp_ln_o:w [671](#), [687](#), [14778](#)
 __fp_ln_significand:NNNNnnN ...
 [673](#), [14803](#), [14806](#), [15301](#)
 __fp_ln_square_t_after:w
 [14901](#), [14932](#)
 __fp_ln_square_t_pack:NNNNw ...
 .. [14903](#), [14905](#), [14907](#), [14909](#), [14930](#)
 __fp_ln_t_large:NNw
 [676](#), [14882](#), [14889](#), [14899](#)
 __fp_ln_t_small:Nw ... [14880](#), [14887](#)
 __fp_ln_t_small:w [676](#)
 __fp_ln_Taylor:wwNw [677](#), [14933](#), [14934](#)
 __fp_ln_Taylor_break:w [14939](#), [14950](#)
 __fp_ln_Taylor_loop:www
 [14935](#), [14936](#), [14945](#)
 __fp_ln_twice_t_after:w [14913](#), [14929](#)
 __fp_ln_twice_t_pack:Nw . [14915](#),
 [14917](#), [14919](#), [14921](#), [14923](#), [14928](#)
 \c__fp_ln_vi_fixed_tl [14769](#)
 \c__fp_ln_vii_fixed_tl [14769](#)
 \c__fp_ln_viii_fixed_tl [14769](#)
 \c__fp_ln_x_fixed_tl
 [14769](#), [14987](#), [14994](#)
 __fp_ln_x_ii:wnnnn ... [14808](#), [14826](#)
 __fp_ln_x_iii:NNNNNw . [14835](#), [14839](#)
 __fp_ln_x_iii_var:NNNNw
 [14833](#), [14841](#)
 __fp_ln_x_iv:wnnnnnnn
 [675](#), [14831](#), [14846](#)
 \c__fp_max_exponent_int
 [548](#), [548](#), [11228](#), [11236](#),
 [11242](#), [11259](#), [11260](#), [14548](#), [14749](#),
 [15040](#), [15067](#), [15356](#), [16261](#), [16309](#)
 __fp_max_fp:N [11233](#)
 \c__fp_middle_shift_int
 [11331](#), [14353](#),
 [14356](#), [14359](#), [14362](#), [14426](#), [14429](#),
 [14432](#), [14435](#), [15323](#), [15326](#), [15329](#),
 [15332](#), [15805](#), [15812](#), [15842](#), [15848](#)
 __fp_min_fp:N [11233](#)
 __fp_minmax_auxi:ww
 [13456](#), [13468](#), [13475](#)
 __fp_minmax_auxii:ww
 [13458](#), [13466](#), [13475](#)
 __fp_minmax_break_o:w . [13449](#), [13479](#)
 __fp_minmax_loop:Nww
 [621](#), [13443](#), [13445](#), [13451](#)
 __fp_minmax_o:Nw
 [616](#), [12700](#), [12702](#), [13440](#)
 __fp_mul_cases_o:NnNnw
 [638](#), [13870](#), [13876](#), [13980](#)
 __fp_mul_cases_o:nNnnw [13876](#)
 __fp_mul_npos_o:Nw [635](#),
 [636](#), [638](#), [719](#), [719](#), [13873](#), [13914](#), [16437](#)
 __fp_mul_significand_drop:NNNNw
 [636](#), [13923](#)
 __fp_mul_significand_keep:NNNNw
 [13923](#)
 __fp_mul_significand_large_-
 f:NwNNNN [13951](#), [13955](#)
 __fp_mul_significand_o:nnnnNnnnn
 [636](#), [636](#), [13921](#), [13923](#)
 __fp_mul_significand_small_-
 f:NNwwN [13949](#), [13966](#)
 __fp_mul_significand_test_f:NNN
 [637](#), [13925](#), [13946](#)
 __fp_neg_sign:N ... [626](#), [11254](#), [13584](#)
 __fp_new_function:NNnnn [13161](#)
 __fp_not_o:w [616](#), [12583](#), [13481](#)
 \c__fp_one_fixed_tl
 [14330](#), [14941](#), [15099](#),
 [15357](#), [15381](#), [16008](#), [16074](#), [16179](#)
 __fp_overflow:w
 [551](#), [562](#), [564](#), [564](#), [11262](#), [11631](#)
 \l__fp_overflow_flag_token ... [11519](#)
 __fp_pack:NNNNw .. [11331](#), [14342](#),
 [14352](#), [14355](#), [14358](#), [14361](#), [14364](#),
 [14425](#), [14428](#), [14431](#), [14434](#), [14437](#),
 [15322](#), [15325](#), [15328](#), [15331](#), [15334](#)
 __fp_pack_big:NNNNNw ... [11335](#),
 [14183](#), [14186](#), [14189](#), [14192](#), [14195](#),
 [14198](#), [14201](#), [14205](#), [14452](#), [14462](#),
 [14472](#), [14481](#), [14484](#), [14487](#), [14494](#)

- __fp_pack_Bigg:NNNNNNw
 [11340](#), [14033](#),
 [14036](#), [14039](#), [14050](#), [14053](#), [14056](#)
- __fp_pack_eight:wNNNNNNNN
 [633](#), [11347](#),
 [13849](#), [14154](#), [14516](#), [15574](#), [15575](#)
- __fp_pack_twice_four:wNNNNNNNN .
 [11345](#), [11915](#), [11916](#), [13792](#), [13793](#),
 [14517](#), [14518](#), [14519](#), [14551](#), [14552](#),
 [14553](#), [14740](#), [14741](#), [15071](#), [15072](#),
 [15073](#), [15576](#), [15577](#), [15790](#), [16430](#)
- __fp_parse:n .. [585](#), [604](#), [615](#), [620](#),
 [720](#), [725](#), [726](#), [11946](#), [12094](#), [12751](#),
 [12768](#), [13196](#), [13244](#), [13258](#), [13268](#),
 [13379](#), [13422](#), [16246](#), [16295](#), [16363](#),
 [16400](#), [16446](#), [16448](#), [16450](#), [16586](#),
 [16622](#), [16643](#), [16645](#), [16647](#), [16670](#)
- __fp_parse_after:ww [12751](#)
- __fp_parse_apply_binary:NwNwN ..
 [577](#), [578](#), [578](#),
 [578](#), [609](#), [12781](#), [12884](#), [12913](#), [12958](#)
- __fp_parse_apply_compare:NwNNNNwN
 [13106](#), [13115](#)
- __fp_parse_apply_compare_
 aux:NNwN [13126](#), [13129](#), [13134](#)
- __fp_parse_apply_juxtapose:NwNwN
 [609](#), [12935](#)
- __fp_parse_apply_unary:NNNwN ...
 [12560](#), [12569](#), [12675](#), [12684](#)
- __fp_parse_compare:NNNNNNN .. [13035](#)
- __fp_parse_compare_auxi:NNNNNNN
 [13035](#)
- __fp_parse_compare_auxii:NNNNN .
 [13035](#)
- __fp_parse_compare_end:NNNNw . [13035](#)
- __fp_parse_continue [605](#)
- __fp_parse_continue:NwN
 .. [577](#), [578](#), [578](#), [578](#), [578](#), [12770](#),
 [12783](#), [13144](#), [13519](#), [13543](#), [13551](#)
- __fp_parse_continue_compare:NNwNN
 [13137](#), [13152](#)
- __fp_parse_digits:N [11976](#)
- __fp_parse_digits_i:N [11976](#)
- __fp_parse_digits_ii:N [11976](#)
- __fp_parse_digits_iii:N [11976](#)
- __fp_parse_digits_iv:N [11976](#)
- __fp_parse_digits_v:N [11976](#)
- __fp_parse_digits_vi:N
 [11976](#), [12270](#), [12318](#)
- __fp_parse_digits_vii:N
 [590](#), [11976](#), [12257](#), [12307](#)
- __fp_parse_excl_error: [13035](#)
- __fp_parse_expand:w [581](#),
 [582](#), [11961](#), [11963](#), [11985](#), [12025](#),
 [12069](#), [12102](#), [12106](#), [12144](#), [12175](#),
 [12213](#), [12215](#), [12234](#), [12236](#), [12258](#),
 [12275](#), [12288](#), [12308](#), [12338](#), [12366](#),
 [12382](#), [12393](#), [12416](#), [12445](#), [12455](#),
 [12462](#), [12475](#), [12491](#), [12511](#), [12522](#),
 [12579](#), [12605](#), [12614](#), [12680](#), [12689](#),
 [12757](#), [12837](#), [12852](#), [12874](#), [12917](#),
 [12942](#), [12982](#), [12994](#), [13011](#), [13027](#),
 [13100](#), [13113](#), [13159](#), [13179](#), [13515](#)
- __fp_parse_exponent:N
 [594](#), [12068](#), [12249](#), [12398](#), [12465](#), [12467](#)
- __fp_parse_exponent:Nw
 [12273](#), [12286](#),
 [12335](#), [12363](#), [12414](#), [12443](#), [12462](#)
- __fp_parse_exponent_aux:N ... [12467](#)
- __fp_parse_exponent_body:N ...
 [12493](#), [12497](#)
- __fp_parse_exponent_digits:N ...
 [12501](#), [12513](#)
- __fp_parse_exponent_keep:N .. [12524](#)
- __fp_parse_exponent_keep:NTF ...
 [12504](#), [12524](#)
- __fp_parse_exponent_sign:N ...
 [12483](#), [12487](#)
- __fp_parse_function:NNN
 [12673](#), [12692](#), [12694](#),
 [12696](#), [12698](#), [12700](#), [12702](#), [12704](#),
 [12706](#), [12727](#), [12729](#), [12731](#), [12745](#)
- __fp_parse_infix:NN
 .. [584](#), [586](#), [601](#), [607](#), [615](#), [12024](#),
 [12153](#), [12192](#), [12612](#), [12631](#), [12636](#),
 [12648](#), [12670](#), [12790](#), [12795](#), [12850](#)
- __fp_parse_infix_
 [12609](#), [12831](#), [12846](#), [12856](#), [12925](#),
 [12926](#), [12928](#), [12929](#), [12965](#), [12977](#),
 [12980](#), [12989](#), [12992](#), [13004](#), [13015](#)
- __fp_parse_infix_!:N [13035](#)
- __fp_parse_infix_&:Nw [12974](#)
- __fp_parse_infix(:N [12933](#)
- __fp_parse_infix_):N [12844](#)
- __fp_parse_infix*:N [12960](#)
- __fp_parse_infix+:N
 [615](#), [11961](#), [12898](#)
- __fp_parse_infix_,:N [12860](#)
- __fp_parse_infix_-:N [12898](#)
- __fp_parse_infix_/:N [12898](#)
- __fp_parse_infix::N . [13002](#), [13504](#)
- __fp_parse_infix<:N [13035](#)
- __fp_parse_infix=:N [13035](#)
- __fp_parse_infix>:N [13035](#)
- __fp_parse_infix?:N [13002](#)
- __fp_parse_infix^:N [12898](#)
- __fp_parse_infix_after_operand:NwN
 [586](#), [12059](#), [12117](#), [12586](#), [12788](#)

__fp_parse_infix_and:N [12898](#), [12996](#)
 __fp_parse_infix_check:NNN
 [12816](#), [12826](#)
 __fp_parse_infix_comma:w [12860](#)
 __fp_parse_infix_comma_error:w .
 [12860](#)
 __fp_parse_infix_end:N
 [604](#), [607](#), [12758](#), [12762](#), [12842](#)
 __fp_parse_infix_juxtapose:N . . .
 [609](#), [12805](#), [12814](#), [12934](#), [12935](#)
 __fp_parse_infix_mark:NNN
 [12802](#), [12841](#)
 __fp_parse_infix_mul:N [12898](#), [12968](#)
 __fp_parse_infix_or:N . [12898](#), [12984](#)
 __fp_parse_infix_|:Nw [12974](#)
 __fp_parse_large:N [588](#), [12220](#), [12303](#)
 __fp_parse_large_leading:wwNN . .
 [12305](#), [12310](#)
 __fp_parse_large_round:NN
 [593](#), [12346](#), [12418](#)
 __fp_parse_large_round_aux:wNN .
 [12418](#)
 __fp_parse_large_round_test:NN .
 [12418](#)
 __fp_parse_large_trailing:wwNN .
 [12316](#), [12340](#)
 __fp_parse_letters:N
 [586](#), [587](#), [12132](#), [12146](#)
 __fp_parse_lparen_after:NwN . [12592](#)
 __fp_parse_o:n
 [573](#), [12764](#), [13377](#), [13378](#), [16552](#), [16617](#)
 __fp_parse_one [605](#)
 __fp_parse_one:Nw [577](#), [577](#),
 [578](#), [579](#), [579](#), [579](#), [579](#), [580](#), [588](#),
 [601](#), [11961](#), [11996](#), [12197](#), [12559](#), [12776](#)
 __fp_parse_one_digit:NN
 [600](#), [12012](#), [12115](#)
 __fp_parse_one_fp:NN
 [583](#), [12004](#), [12020](#)
 __fp_parse_one_other:NN [12015](#), [12123](#)
 __fp_parse_one_register:NN
 [12007](#), [12057](#)
 __fp_parse_one_register_aux:Nw .
 [12057](#)
 __fp_parse_one_register_-
 auxii:wwwNw [12057](#)
 __fp_parse_one_register_dim:ww .
 [12057](#)
 __fp_parse_one_register_int:www
 [12057](#)
 __fp_parse_one_register_mu:www .
 [12057](#)
 __fp_parse_one_register_wd:Nw [12057](#)
 __fp_parse_one_register_wd:w . [12057](#)
 __fp_parse_operand [605](#)
 __fp_parse_operand:Nw [577](#), [577](#),
 [577](#), [577](#), [579](#), [579](#), [579](#), [580](#), [580](#),
 [604](#), [614](#), [614](#), [11961](#), [12575](#), [12577](#),
 [12601](#), [12603](#), [12680](#), [12689](#), [12756](#),
 [12770](#), [12873](#), [12916](#), [12941](#), [13010](#),
 [13026](#), [13113](#), [13159](#), [13179](#), [13514](#)
 __fp_parse_pack_carry:w . [591](#), [12290](#)
 __fp_parse_pack_leading:NNNNNww
 [12253](#), [12290](#), [12313](#)
 __fp_parse_pack_trailing:NNNNNww
 [12263](#), [12290](#), [12332](#), [12343](#), [12350](#)
 __fp_parse_prefix:NNN . [12135](#), [12177](#)
 __fp_parse_prefix [12595](#)
 __fp_parse_prefix!:Nw [12565](#)
 __fp_parse_prefix(:Nw [12592](#)
 __fp_parse_prefix):Nw [12623](#)
 __fp_parse_prefix+:Nw [12559](#)
 __fp_parse_prefix-:Nw [12565](#)
 __fp_parse_prefix.:Nw [12584](#)
 __fp_parse_prefix_unknown:NNN [12177](#)
 __fp_parse_return_semicolon:w . .
 [11962](#), [11983](#), [12173](#),
 [12380](#), [12391](#), [12473](#), [12505](#), [12520](#)
 __fp_parse_round:Nw [12732](#)
 __fp_parse_round_after:wN
 [594](#), [12395](#), [12400](#), [12450](#)
 __fp_parse_round_loop:N [594](#), [594](#),
 [594](#), [595](#), [12368](#), [12411](#), [12429](#), [12454](#)
 __fp_parse_round_up:N [12368](#)
 __fp_parse_small:N [589](#), [12240](#), [12251](#)
 __fp_parse_small_leading:wwNN . .
 [12255](#), [12260](#), [12322](#)
 __fp_parse_small_round:NN
 [12282](#), [12400](#), [12439](#)
 __fp_parse_small_trailing:wwNN .
 [12268](#), [12277](#), [12354](#)
 __fp_parse_strim_end:w [12226](#)
 __fp_parse_strim_zeros:N
 [588](#), [600](#), [12207](#), [12226](#), [12590](#)
 __fp_parse_trim_end:w [12200](#)
 __fp_parse_trim_zeros:N [12121](#), [12200](#)
 __fp_parse_unary_function:nNN . .
 [12673](#), [12708](#),
 [12710](#), [12712](#), [12714](#), [12722](#), [12724](#)
 __fp_parse_word:Nw [586](#), [12129](#), [12146](#)
 __fp_parse_word_abs:N [12707](#)
 __fp_parse_word_acos:N [12715](#)
 __fp_parse_word_acosd:N [12715](#)
 __fp_parse_word_acot:N [12691](#)
 __fp_parse_word_acotd:N [12691](#)
 __fp_parse_word_acsc:N [12715](#)
 __fp_parse_word_acscd:N [12715](#)
 __fp_parse_word_asec:N [12715](#)

- __fp_parse_word_asecd:N [12715](#)
- __fp_parse_word_asin:N [12715](#)
- __fp_parse_word_asind:N [12715](#)
- __fp_parse_word_atan:N [12691](#)
- __fp_parse_word_atand:N [12691](#)
- __fp_parse_word_bp:N [12644](#)
- __fp_parse_word_cc:N [12644](#)
- __fp_parse_word_ceil:N [12726](#)
- __fp_parse_word_cm:N [12644](#)
- __fp_parse_word_cos:N [12715](#)
- __fp_parse_word_cosd:N [12715](#)
- __fp_parse_word_cot:N [12715](#)
- __fp_parse_word_cotd:N [12715](#)
- __fp_parse_word_csc:N [12715](#)
- __fp_parse_word_cscd:N [12715](#)
- __fp_parse_word_dd:N [12644](#)
- __fp_parse_word_deg:N [12633](#)
- __fp_parse_word_em:N [12663](#)
- __fp_parse_word_ex:N [12663](#)
- __fp_parse_word_exp:N [12707](#)
- __fp_parse_word_false:N [12633](#)
- __fp_parse_word_floor:N [12726](#)
- __fp_parse_word_in:N [12644](#)
- __fp_parse_word_inf:N [12633](#)
- __fp_parse_word_ln:N [12707](#)
- __fp_parse_word_max:N [12691](#)
- __fp_parse_word_min:N [12691](#)
- __fp_parse_word_mm:N [12644](#)
- __fp_parse_word_nan:N [12633](#)
- __fp_parse_word_nc:N [12644](#)
- __fp_parse_word_nd:N [12644](#)
- __fp_parse_word_pc:N [12644](#)
- __fp_parse_word_pi:N [12633](#)
- __fp_parse_word_pt:N [12644](#)
- __fp_parse_word_rand:N [12691](#)
- __fp_parse_word_randint:N ... [12691](#)
- __fp_parse_word_round:N [12732](#)
- __fp_parse_word_sec:N [12715](#)
- __fp_parse_word_secd:N [12715](#)
- __fp_parse_word_sin:N [12715](#)
- __fp_parse_word_sind:N [12715](#)
- __fp_parse_word_sp:N [12644](#)
- __fp_parse_word_sqrt:N [12707](#)
- __fp_parse_word_tan:N [12715](#)
- __fp_parse_word_tand:N [12715](#)
- __fp_parse_word_true:N [12633](#)
- __fp_parse_word_trunc:N [12726](#)
- __fp_parse_zero: [588](#), [12222](#), [12242](#), [12246](#)
- __fp_pow_B:wwN [15304](#), [15339](#)
- __fp_pow_C_neg:w [15342](#), [15359](#)
- __fp_pow_C_overflow:w [15347](#), [15354](#), [15375](#)
- __fp_pow_C_pack:w [15361](#), [15369](#), [15380](#)
- __fp_pow_C_pos:w [15345](#), [15364](#)
- __fp_pow_C_pos_loop:wN [15365](#), [15366](#), [15373](#)
- __fp_pow_exponent:Nwnnnnnw [15310](#), [15313](#), [15318](#)
- __fp_pow_exponent:wnN . [15302](#), [15307](#)
- __fp_pow_neg:www .. [689](#), [15218](#), [15382](#)
- __fp_pow_neg_aux:wNN ... [689](#), [15382](#)
- __fp_pow_neg_case:w .. [15384](#), [15405](#)
- __fp_pow_neg_case_aux:nnnnn . [15405](#)
- __fp_pow_neg_case_aux:NNNNNNNNw [15405](#)
- __fp_pow_normal:ww [684](#), [685](#), [15223](#), [15252](#)
- __fp_pow_npos:Nww [15264](#), [15281](#)
- __fp_pow_npos:ww [686](#)
- __fp_pow_npos_aux:NNnw [15287](#), [15291](#), [15297](#)
- __fp_pow_zero_or_inf:ww [684](#), [15225](#), [15232](#)
- \c__fp_prec_and_int ... [11946](#), [12930](#)
- \c__fp_prec_colon_int [11946](#), [13026](#), [13514](#)
- \c__fp_prec_comma_int ... [11946](#), [12601](#), [12625](#), [12864](#), [12869](#), [12873](#)
- \c__fp_prec_comp_int [11946](#), [13067](#), [13113](#)
- \c__fp_prec_end_int [604](#), [11946](#), [12756](#)
- \c__fp_prec_func_int .. [11946](#), [12680](#)
- \c__fp_prec_funcii_int [11946](#), [12600](#), [12689](#), [13159](#), [13179](#)
- \c__fp_prec_hat_int ... [11946](#), [12925](#)
- \c__fp_prec_hatii_int . [11946](#), [12925](#)
- \c__fp_prec_not_int [599](#), [11946](#), [12582](#), [12583](#)
- \c__fp_prec_or_int [11946](#), [12931](#)
- \c__fp_prec_paren_int [11946](#), [12603](#), [12848](#)
- \c__fp_prec_plus_int [576](#), [11946](#), [12928](#), [12929](#)
- \c__fp_prec_quest_int [11946](#), [13006](#), [13010](#), [13020](#)
- \c__fp_prec_times_int [11946](#), [12926](#), [12927](#), [12937](#), [12941](#)
- \c__fp_rand_eight_int [16492](#), [16511](#), [16514](#)
- \c__fp_rand_four_int [16492](#), [16517](#), [16520](#), [16533](#), [16536](#)
- __fp_rand_myriads:n [722](#), [723](#), [16497](#), [16552](#), [16622](#)
- __fp_rand_myriads_get:w [16497](#)
- __fp_rand_myriads_last: [16497](#)
- __fp_rand_myriads_last:w [16497](#)
- __fp_rand_myriads_loop:nn ... [16497](#)

- _fp_rand_o: [16541](#)
- _fp_rand_o:Nw [12704](#), [16483](#), [16488](#), [16541](#)
- _fp_rand_o:w [16541](#)
- _fp_rand_size_int [763](#), [763](#), [16492](#), [16598](#), [16609](#), [17848](#), [17849](#), [17860](#), [17869](#)
- _fp_rand_uniform: [16492](#), [16507](#), [16529](#), [16596](#), [16612](#), [16630](#)
- _fp_randint_badarg:w [16553](#)
- _fp_randint_e:w [16553](#)
- _fp_randint_e:wnn [16553](#)
- _fp_randint_e:wwNnn [16553](#)
- _fp_randint_e:wwwNnn ... [723](#), [16553](#)
- _fp_randint_narrow_e:nnnn .. [16553](#)
- _fp_randint_o:Nw [12706](#), [16488](#), [16553](#)
- _fp_randint_wide_e:nnnn ... [16553](#)
- _fp_randint_wide_e:wnnn ... [16553](#)
- _fp_reverse_args:Nww [712](#), [713](#), [11206](#), [15945](#), [16030](#), [16142](#), [16208](#)
- _fp_round:NNN [567](#), [567](#), [569](#), [637](#), [653](#), [11689](#), [11759](#), [13709](#), [13720](#), [13961](#), [13973](#), [14113](#), [14124](#), [14308](#)
- _fp_round:Nwn . [11809](#), [11862](#), [16405](#)
- _fp_round:Nww . [11810](#), [11831](#), [11862](#)
- _fp_round:Nwww [11811](#), [11825](#)
- _fp_round_digit:Nw [556](#), [557](#), [557](#), [636](#), [637](#), [653](#), [11371](#), [11391](#), [11773](#), [13723](#), [13865](#), [13964](#), [13976](#), [14127](#), [14313](#)
- _fp_round_name_from_cs:N [11821](#), [11847](#), [11851](#), [11867](#)
- _fp_round_neg:NNN [567](#), [634](#), [634](#), [11784](#), [13827](#), [13842](#), [13860](#)
- _fp_round_no_arg_o:Nw [11808](#), [11815](#)
- _fp_round_normal:NnnwNnn .. [11862](#)
- _fp_round_normal:NnwNnn [11862](#)
- _fp_round_normal:NwNnnw [11862](#)
- _fp_round_normal_end:wwNnn . [11862](#)
- _fp_round_o:Nw [11804](#), [12727](#), [12729](#), [12731](#), [12746](#)
- _fp_round_pack:Nw [11862](#)
- _fp_round_return_one: [567](#), [11689](#), [11695](#), [11705](#), [11713](#), [11717](#), [11726](#), [11730](#), [11739](#), [11746](#), [11750](#), [11788](#), [11798](#)
- _fp_round_s:NNNw [567](#), [594](#), [11757](#), [12404](#), [12422](#)
- _fp_round_special:NwNnn ... [11862](#)
- _fp_round_special_aux:Nw ... [11862](#)
- _fp_round_to_nearest:NNN [570](#), [571](#), [11689](#), [11793](#), [11817](#), [11827](#), [12746](#), [12750](#), [16405](#)
- _fp_round_to_nearest_neg:NNN [11784](#)
- _fp_round_to_nearest_ninf:NNN . [571](#), [11689](#), [11802](#)
- _fp_round_to_nearest_ninf_neg:NNN [11784](#)
- _fp_round_to_nearest_pinf:NNN . [571](#), [11689](#), [11794](#)
- _fp_round_to_nearest_pinf_neg:NNN [11784](#)
- _fp_round_to_nearest_zero:NNN . [571](#), [11689](#)
- _fp_round_to_nearest_zero_neg:NNN [11784](#)
- _fp_round_to_ninf:NNN [11689](#), [11792](#), [11855](#), [12729](#), [12741](#)
- _fp_round_to_ninf_neg:NNN .. [11784](#)
- _fp_round_to_pinf:NNN [11689](#), [11784](#), [11857](#), [12731](#), [12735](#)
- _fp_round_to_pinf_neg:NNN .. [11784](#)
- _fp_round_to_zero:NNN [11689](#), [11853](#), [12727](#), [12738](#)
- _fp_round_to_zero_neg:NNN .. [11784](#)
- _fp_rrot:www [11207](#), [16075](#)
- _fp_sanitize:Nw [628](#), [631](#), [636](#), [639](#), [647](#), [702](#), [710](#), [710](#), [11256](#), [11923](#), [11941](#), [13652](#), [13746](#), [13917](#), [13996](#), [14142](#), [14794](#), [15030](#), [15283](#), [15907](#), [15951](#), [16087](#)
- _fp_sanitize:wN [586](#), [590](#), [11256](#), [12120](#), [12589](#)
- _fp_sanitize_zero:w [11256](#)
- _fp_sec_o:w [15497](#)
- _fp_set_sign_o:w [12582](#), [14317](#)
- _fp_sin_o:w [599](#), [711](#), [15452](#)
- _fp_sin_series_aux_o:NNnwww . [15859](#)
- _fp_sin_series_o:NNwww .. [691](#), [703](#), [15458](#), [15473](#), [15488](#), [15503](#), [15859](#)
- _fp_small_int:wTF ... [11433](#), [11864](#)
- _fp_small_int_normal:NnwTF ... [559](#), [11433](#)
- _fp_small_int_test:NnnwNnw ... [11453](#), [11460](#)
- _fp_small_int_test:NnnwNTF ... [559](#), [11433](#)
- _fp_small_int_true:wTF . [559](#), [11433](#)
- _fp_sqrt_auxi_o:NNNwnnN [14164](#), [14172](#)
- _fp_sqrt_auxii_o:NnnnnnnnN [649](#), [650](#), [651](#), [14174](#), [14178](#), [14258](#), [14270](#)
- _fp_sqrt_auxiii_o:wnnnnnnnn ... [14175](#), [14213](#), [14259](#)
- _fp_sqrt_auxiv_o:NNNNnw [14213](#)
- _fp_sqrt_auxix_o:wwnnw [14247](#)
- _fp_sqrt_auxv_o:NNNNnw [14213](#)
- _fp_sqrt_auxvi_o:NNNNnw [14213](#)

- __fp_sqrt_auxvii_o:NNNNw ... [14213](#)
- __fp_sqrt_auxviii_o:nnnnnnn ...
... [14235](#), [14237](#), [14239](#), [14245](#), [14247](#)
- __fp_sqrt_auxx_o:Nnnnnnnn
..... [14243](#), [14261](#)
- __fp_sqrt_auxxi_o:wNnnN [14261](#)
- __fp_sqrt_auxxii_o:nnnnnnnnw ...
..... [14271](#), [14275](#)
- __fp_sqrt_auxxiii_o:w [14275](#)
- __fp_sqrt_auxxiv_o:wnnnnnnN ...
..... [14287](#), [14290](#), [14298](#), [14300](#)
- __fp_sqrt_Newton_o:wN
..... [648](#), [14149](#), [14160](#), [14161](#)
- __fp_sqrt_npos_auxi_o:wNnnN . [14140](#)
- __fp_sqrt_npos_auxii_o:wNNNNNNNN [14140](#)
- __fp_sqrt_npos_o:w ... [14137](#), [14140](#)
- __fp_sqrt_o:w [14130](#)
- __fp_step:NnnnnN [13374](#)
- __fp_step:wwN [13374](#)
- __fp_sub_back_far_o:NnnwnnnN ..
..... [633](#), [13755](#), [13801](#)
- __fp_sub_back_near_after:wNNNNw [13761](#), [13838](#)
- __fp_sub_back_near_o:nnnnnnnnN .
..... [631](#), [13751](#), [13761](#)
- __fp_sub_back_near_pack:NNNNNNw [13761](#), [13840](#)
- __fp_sub_back_not_far_o:wwwNN .
..... [13815](#), [13835](#)
- __fp_sub_back_quite_far_ii:NN [13819](#)
- __fp_sub_back_quite_far_o:wwNN .
..... [13813](#), [13819](#)
- __fp_sub_back_shift:wnnnn
..... [632](#), [13773](#), [13777](#)
- __fp_sub_back_shift_ii:ww ... [13777](#)
- __fp_sub_back_shift_iii:NNNNNNNNw [13777](#)
- __fp_sub_back_shift_iv:nnnw . [13777](#)
- __fp_sub_back_very_far_ii_-
o:nnNwwNN [13847](#)
- __fp_sub_back_very_far_o:wwwNN
..... [13814](#), [13847](#)
- __fp_sub_eq_o:Nnnnw [13726](#)
- __fp_sub_npos_i_o:Nnnnw
..... [630](#), [13731](#), [13740](#), [13744](#)
- __fp_sub_npos_ii_o:Nnnnw ... [13726](#)
- __fp_sub_npos_o:NnnNw
..... [630](#), [13646](#), [13726](#)
- __fp_tan_o:w [15512](#)
- __fp_tan_series_aux_o:Nnwww . [15913](#)
- __fp_tan_series_o:NNwww
..... [693](#), [693](#), [15519](#), [15534](#), [15913](#)
- __fp_ternary:NwwN . [616](#), [13008](#), [13502](#)
- __fp_ternary_auxi:NwwN .. [616](#), [13502](#)
- __fp_ternary_auxii:NwwN
..... [616](#), [13024](#), [13502](#)
- __fp_ternary_break_point:n .. [13502](#)
- __fp_ternary_loop:Nw [13502](#)
- __fp_ternary_loop_break:w ... [13502](#)
- __fp_ternary_map_break: [13502](#)
- __fp_tmp:w [556](#), [608](#),
[11365](#), [11375](#), [11376](#), [11377](#), [11378](#),
[11379](#), [11380](#), [11381](#), [11382](#), [11383](#),
[11384](#), [11385](#), [11386](#), [11387](#), [11388](#),
[11389](#), [11390](#), [11976](#), [11988](#), [11989](#),
[11990](#), [11991](#), [11992](#), [11993](#), [11994](#),
[12565](#), [12582](#), [12583](#), [12633](#), [12638](#),
[12639](#), [12640](#), [12641](#), [12642](#), [12643](#),
[12644](#), [12652](#), [12653](#), [12654](#), [12655](#),
[12656](#), [12657](#), [12658](#), [12659](#), [12660](#),
[12661](#), [12662](#), [12907](#), [12925](#), [12926](#),
[12927](#), [12928](#), [12929](#), [12930](#), [12931](#)
- __fp_to_decimal_dispatch:w
..... [716](#), [717](#),
[719](#), [13420](#), [16290](#), [16294](#), [16297](#), [16404](#)
- __fp_to_decimal_huge:wnnnn .. [16297](#)
- __fp_to_decimal_large:Nnnw .. [16297](#)
- __fp_to_decimal_normal:wnnnn
..... [16297](#), [16385](#)
- __fp_to_int_dispatch:w
..... [16395](#), [16399](#), [16402](#)
- __fp_to_scientific_dispatch:w ..
... [714](#), [716](#), [717](#), [16241](#), [16245](#), [16248](#)
- __fp_to_scientific_normal:wnnnn
..... [16248](#), [16378](#), [16382](#)
- __fp_to_scientific_normal:wNw [16248](#)
- __fp_to_tl_dispatch:w
.... [717](#), [16358](#), [16362](#), [16365](#), [16472](#)
- __fp_to_tl_normal:nnnn [16365](#)
- \c_fp_trailing_shift_int
..... [11331](#), [14343](#),
[14365](#), [14438](#), [15335](#), [15805](#), [15842](#)
- __fp_trap_division_by_zero_-
set:N [11574](#)
- __fp_trap_division_by_zero_set_-
error: [11574](#)
- __fp_trap_division_by_zero_set_-
flag: [11574](#)
- __fp_trap_division_by_zero_set_-
none: [11574](#)
- __fp_trap_invalid_operation_-
set:N [11540](#)
- __fp_trap_invalid_operation_-
set_error: [11540](#)
- __fp_trap_invalid_operation_-
set_flag: [11540](#)

- _fp_trap_invalid_operation_-
 set_none: [11540](#)
 - _fp_trap_overflow_set:N [11600](#)
 - _fp_trap_overflow_set:NnNn . [11600](#)
 - _fp_trap_overflow_set_error: [11600](#)
 - _fp_trap_overflow_set_flag: . [11600](#)
 - _fp_trap_overflow_set_none: . [11600](#)
 - _fp_trap_underflow_set:N ... [11600](#)
 - _fp_trap_underflow_set_error: .
 [11600](#)
 - _fp_trap_underflow_set_flag: [11600](#)
 - _fp_trap_underflow_set_none: [11600](#)
 - _fp_trig:NNNNwn . [15458](#), [15473](#),
 [15488](#), [15503](#), [15518](#), [15533](#), [15550](#)
 - _fp_trig_inverse_two_pi:
 [700](#), [15611](#), [15780](#)
 - _fp_trig_large:ww ... [15558](#), [15775](#)
 - _fp_trig_large_auxi:wwwww . [15775](#)
 - _fp_trig_large_auxii:ww [15775](#)
 - _fp_trig_large_auxiii:wNNNNNNNN
 [15775](#)
 - _fp_trig_large_auxiv:wN [15775](#)
 - _fp_trig_large_auxix:Nw [15818](#)
 - _fp_trig_large_auxv:www
 [15791](#), [15798](#)
 - _fp_trig_large_auxvi:wnnnnnnnn
 [15798](#)
 - _fp_trig_large_auxvii:w
 [15801](#), [15818](#)
 - _fp_trig_large_auxviii:w ... [15818](#)
 - _fp_trig_large_auxviii:ww
 [15820](#), [15824](#)
 - _fp_trig_large_auxx:wNNNNN . [15818](#)
 - _fp_trig_large_auxxi:w [15818](#)
 - _fp_trig_large_pack:NNNNw ...
 [15798](#), [15847](#)
 - _fp_trig_small:ww
 . [695](#), [701](#), [15560](#), [15564](#), [15570](#), [15857](#)
 - _fp_trigd_large:ww .. [15558](#), [15572](#)
 - _fp_trigd_large_auxi:nnnnwNNNN
 [15572](#)
 - _fp_trigd_large_auxii:wNw .. [15572](#)
 - _fp_trigd_large_auxiii:www . [15572](#)
 - _fp_trigd_small:ww
 [695](#), [15560](#), [15566](#), [15609](#)
 - _fp_trim_zeros:w
 [16232](#), [16288](#), [16341](#), [16350](#)
 - _fp_trim_zeros_dot:w [16232](#)
 - _fp_trim_zeros_end:w [16232](#)
 - _fp_trim_zeros_loop:w [16232](#)
 - _fp_type_from_scan:N
 [583](#), [11326](#), [11964](#), [12022](#)
 - _fp_type_from_scan:w [11964](#)
 - _fp_underflow:w
 [551](#), [562](#), [564](#), [564](#), [11263](#), [11631](#)
 - \l_fp_underflow_flag_token .. [11519](#)
 - _fp_use_i:ww
 [662](#), [711](#), [11208](#), [14554](#), [16161](#)
 - _fp_use_i:www [11208](#)
 - _fp_use_i_until_s:nw
 [701](#), [11203](#), [11250](#), [11488](#),
 [12891](#), [15602](#), [15800](#), [15806](#), [15837](#)
 - _fp_use_ii_until_s:nnw [11203](#), [11248](#)
 - _fp_use_none_stop_f:n
 [11200](#), [14718](#), [14719](#), [14720](#)
 - _fp_use_none_until_s:w
 .. [11203](#), [14166](#), [15391](#), [16156](#), [16159](#)
 - _fp_use_s:n [11201](#)
 - _fp_use_s:nn [11201](#)
 - _fp_zero_fp:N . [11229](#), [11615](#), [11929](#)
 - _fp_l_o:ww [616](#), [13489](#)
 - _fp_ [13492](#), [13499](#)
 - \futurelet [374](#)
- G**
- \gdef [375](#)
 - \GetIdInfo [6](#), [6](#), [6](#)
 - \gleaders [884](#)
 - \global [166](#), [167](#),
 [181](#), [182](#), [183](#), [194](#), [195](#), [196](#), [197](#),
 [198](#), [199](#), [200](#), [201](#), [202](#), [205](#), [272](#), [376](#)
 - \globaldefs [377](#)
 - \glueexpr [631](#)
 - \glueshrink [632](#)
 - \glueshrinkorder [633](#)
 - \gluestretch [634](#)
 - \gluestretchorder [635](#)
 - \gluetomu [636](#)
 - group commands:
 - \group_align_safe_begin/end: .. [306](#)
 - \group_align_safe_begin: [40](#),
 [40](#), [40](#), [299](#), [377](#), [381](#), [2611](#), [2785](#),
 [3473](#), [3491](#), [5202](#), [5514](#), [18192](#), [18987](#)
 - \group_align_safe_end: [40](#), [40](#), [40](#),
 [377](#), [381](#), [2659](#), [2660](#), [2785](#), [3455](#),
 [3465](#), [3470](#), [3488](#), [5225](#), [5540](#), [18213](#)
 - \group_begin: [9](#),
 [9](#), [9](#), [480](#), [1319](#), [2028](#), [2450](#), [3019](#),
 [3025](#), [3086](#), [3159](#), [3183](#), [3201](#), [3225](#),
 [3315](#), [3333](#), [3549](#), [5067](#), [5073](#), [5101](#),
 [5301](#), [5772](#), [5795](#), [5945](#), [6269](#), [6288](#),
 [6618](#), [7863](#), [8787](#), [8937](#), [8958](#), [9001](#),
 [9247](#), [9496](#), [9642](#), [10587](#), [11006](#),
 [11037](#), [12040](#), [12592](#), [12793](#), [12844](#),
 [12860](#), [12898](#), [12960](#), [12974](#), [13002](#),
 [13489](#), [16790](#), [17131](#), [17245](#), [17285](#),
 [17302](#), [17319](#), [17333](#), [17351](#), [18141](#),

- 18166, 18746, 18755, 18777, 18927,
18949, 19175, 20152, 20286, 20569
- \c_group_begin_token 51, 100, 318,
391, 3183, 3207, 5647, 5682, 7889, 7934
- \group_end: 9, 9, 9, 9, 418, 480, 1319,
2028, 2453, 3027, 3034, 3162, 3180,
3200, 3204, 3232, 3332, 3378, 3614,
5069, 5081, 5159, 5305, 5308, 5790,
5815, 5950, 6295, 6400, 6628, 6633,
7874, 8791, 8944, 8969, 9106, 9292,
9512, 9693, 10621, 11010, 11073,
12055, 12622, 12840, 12859, 12897,
12932, 12973, 13001, 13034, 13500,
16809, 17049, 17136, 17255, 17294,
17311, 17325, 17343, 17365, 18148,
18171, 18769, 18776, 18942, 18948,
18973, 19199, 20157, 20294, 20574
- \c_group_end_token
51, 318, 3183, 3212, 7894, 7895, 7942
- \group_insert_after:N 9, 9, 9, 1324,
19485, 19517, 19531, 19878, 20268
- groups commands:
.groups:n 164, 10140
- ## H
- \H 18976
- \halign 378
- \hangafter 379
- \hangindent 380
- \hbadness 381
- \hbox 382
- hbox commands:
\hbox:n 141,
141, 7876, 8581, 8636, 17160, 17398
- \hbox_gset:Nn 142, 7877
- \hbox_gset:Nw 142, 7888
- \hbox_gset_end: 142, 7888
- \hbox_gset_to_wd:Nnn 142, 7882
- \hbox_overlap_left:n . 142, 142, 7899
- \hbox_overlap_right:n
..... 142, 142, 7899,
19552, 19591, 19830, 19862, 20252
- \hbox_set:Nn 142, 142,
142, 7877, 8034, 8142, 8367, 8438,
8726, 17129, 17156, 17157, 17243,
17283, 17300, 17317, 17331, 17349,
17371, 17407, 17411, 17419, 17427,
17436, 17445, 17457, 17465, 17473,
17479, 17492, 17544, 17558, 19776
- \hbox_set:Nw 142, 142, 7888, 8084
- \hbox_set_end: . . 142, 142, 7888, 8088
- \hbox_set_to_wd:Nnn . . 142, 142, 7882
- \hbox_to_wd:nn . 142, 142, 7896, 17389
- \hbox_to_zero:n
..... 142, 142, 7896, 7900, 7902
- \hbox_unpack:N
..... 142, 142, 7903, 8371, 8518
- \hbox_unpack_clear:N . 142, 142, 7903
- hcoffin commands:
\hcoffin_set:Nn
146, 146, 8030, 8578, 8590, 8633, 8673
- \hcoffin_set:Nw 146, 146, 8080
- \hcoffin_set_end: 146, 146, 8080
- \hfil 383
- \hfill 384
- \hfilneg 385
- \hfuzz 386
- \hjcode 880
- \hoffset 387
- \holdinginserts 388
- \hpack 881
- \hrule 389
- \hsize 390
- \hskip 391
- \hss 392
- \ht 393
- hundred commands:
\c_one_hundred 72, 4445
- \hyphenation 394
- \hyphenationbounds 882
- \hyphenationmin 883
- \hyphenchar 395
- \hyphenpenalty 396
- ## I
- \I 195
- \i 198, 18971
- \if 397
- if commands:
\if:w 21, 47, 47, 47,
264, 265, 265, 265, 292, 1292, 1592,
1942, 1943, 2376, 2379, 2380, 2434,
2437, 3425, 4305, 6315, 9719, 9723,
9741, 12202, 12206, 12228, 12321,
12353, 12372, 12438, 12452, 12469,
12489, 12964, 12979, 12991, 15286
- \if_bool:N . 40, 40, 40, 2496, 2528, 8835
- \if_box_empty:N . 145, 145, 7808, 7824
- \if_case:w ... 73, 73, 346, 401, 401,
403, 557, 638, 682, 682, 1858, 3174,
3649, 4196, 4229, 6018, 6092, 11258,
11409, 11435, 11806, 11835, 13078,
13122, 13589, 13728, 13803, 13827,
13879, 14324, 14563, 14783, 14810,
14967, 15002, 15108, 15124, 15140,
15156, 15172, 15188, 15213, 15262,
15384, 15407, 15454, 15469, 15484,

15499, 15514, 15529, 15969, 15986,
 16038, 16121, 16136, 16188, 16201,
 16251, 16300, 16368, 16555, 16581
`\if_catcode:w`
 21, 328, 382, 391, 392, 1292, 3207,
 3212, 3217, 3222, 3229, 3236, 3241,
 3246, 3251, 3256, 3261, 3271, 3300,
 3507, 3512, 5337, 5638, 5680, 5692,
 5709, 11998, 12161, 12479, 12526,
 12797, 12951, 13079, 19181, 19182
`\if_charcode:w` 21, 47, 328, 390, 391,
 405, 1292, 3276, 3509, 5619, 5673, 6175
`\if_cs_exist:N`
 21, 1306, 1623, 1651, 3308, 3434
`\if_cs_exist:w` 21,
 1306, 1332, 1632, 1660, 1846, 11513
`\if_dim:w` 88, 88, 4466, 4543, 4555, 4580
`\if_eof:w` 179, 179, 10852, 10860
`\if_false:` 21, 36, 306,
 337, 377, 381, 389, 421, 1292, 2786,
 3117, 3842, 4565, 5220, 5221, 5322,
 5326, 5587, 5592, 5603, 5693, 5705,
 5720, 5728, 6569, 6572, 6707, 6712
`\if_hbox:N` 144, 144, 7808, 7812
`\if_int_compare:w` 20, 73,
 73, 337, 337, 338, 338, 1322, 2786,
 2788, 3054, 3057, 3058, 3065, 3066,
 3067, 3068, 3073, 3074, 3292, 3416,
 3649, 3688, 3758, 3822, 3875, 3877,
 3879, 3881, 3883, 3885, 3887, 3890,
 4772, 5844, 5852, 5867, 5872, 6005,
 6057, 6058, 6064, 6076, 6092, 6318,
 6342, 6353, 6354, 6361, 6373, 10857,
 11259, 11260, 11353, 11417, 11451,
 11465, 11694, 11704, 11712, 11725,
 11738, 11745, 11766, 11778, 11787,
 11797, 11891, 11896, 11980, 12010,
 12125, 12127, 12160, 12165, 12219,
 12239, 12266, 12280, 12315, 12342,
 12370, 12386, 12402, 12420, 12479,
 12499, 12515, 12528, 12541, 12574,
 12600, 12625, 12798, 12808, 12810,
 12848, 12864, 12869, 12911, 12937,
 13006, 13020, 13067, 13256, 13286,
 13289, 13300, 13303, 13308, 13309,
 13312, 13315, 13461, 13535, 13593,
 13613, 13655, 13750, 13804, 13805,
 13808, 13811, 13880, 13889, 14090,
 14163, 14216, 14220, 14224, 14242,
 14277, 14278, 14279, 14280, 14281,
 14307, 14565, 14568, 14662, 14754,
 14796, 14812, 14938, 14972, 15033,
 15040, 15044, 15081, 15255, 15257,
 15268, 15286, 15309, 15341, 15344,
 15387, 15415, 15416, 15424, 15425,
 15439, 15557, 15601, 15990, 16027,
 16036, 16071, 16155, 16158, 16278,
 16351, 16377, 16380, 16511, 16517,
 16533, 16571, 16726, 16796, 16864,
 16872, 16883, 16886, 16904, 16939,
 16949, 16959, 18691, 18692, 18697
`\if_int_odd:w` .. 73, 73, 704, 3064,
 3072, 3092, 3649, 3755, 3927, 3935,
 4451, 11716, 11763, 11775, 13118,
 13863, 14145, 15440, 15827, 15866,
 15876, 15919, 15943, 16112, 19180
`\if_meaning:w` 21, 379, 379,
 391, 450, 622, 1292, 1455, 1470,
 1488, 1538, 1543, 1552, 1620, 1638,
 1648, 1666, 1876, 1887, 2003, 2034,
 2081, 2082, 2319, 2347, 2356, 2573,
 2634, 2653, 2803, 2809, 2835, 2848,
 2856, 3266, 3305, 3343, 3346, 3408,
 3500, 3541, 3669, 3704, 3709, 3710,
 3857, 4524, 4573, 4999, 5253, 5265,
 5278, 5289, 5304, 5532, 5596, 5664,
 5954, 6112, 6150, 6306, 6388, 6605,
 6623, 6642, 6650, 7065, 7080, 7102,
 7118, 7670, 7708, 9658, 9682, 9699,
 9710, 11247, 11261, 11273, 11283,
 11293, 11427, 11442, 11462, 11464,
 11623, 11693, 11703, 11715, 11728,
 11729, 11748, 11749, 11763, 11764,
 11775, 11776, 11834, 11873, 11908,
 11911, 11927, 11934, 11999, 12002,
 12062, 12063, 12064, 12179, 12292,
 12298, 12527, 12734, 12737, 12740,
 12828, 12890, 13119, 13136, 13248,
 13275, 13276, 13277, 13278, 13279,
 13280, 13442, 13454, 13455, 13483,
 13494, 13504, 13564, 13573, 13590,
 13624, 13629, 13643, 13689, 13696,
 13772, 13784, 13883, 13886, 13897,
 13948, 14021, 14089, 14092, 14099,
 14132, 14133, 14136, 14536, 14547,
 14727, 14737, 14780, 14879, 14958,
 15007, 15021, 15210, 15222, 15235,
 15238, 15241, 15267, 15368, 15372,
 15869, 15922, 15984, 15985, 15987,
 15988, 16007, 16024, 16090, 16188,
 16250, 16299, 16367, 16416, 16421,
 16570, 16737, 16738, 17972, 19183
`\if_mode_horizontal:` . 21, 1302, 2780
`\if_mode_inner:` 21, 1302, 2782
`\if_mode_math:` 21, 1302, 2784
`\if_mode_vertical:` ... 21, 1302, 2778
`\if_predicate:w`
 34, 36, 40, 40, 2496, 2603

- `\if_true:` [21](#), [36](#), [379](#), [379](#), [1292](#)
- `\if_vbox:N` [144](#), [144](#), [7808](#), [7814](#)
- `\ifabsdim` [949](#)
- `\ifabsnum` [950](#)
- `\ifcase` [398](#)
- `\ifcat` [399](#)
- `\ifcsname` [637](#)
- `\ifdbbox` [1122](#)
- `\ifddir` [1123](#)
- `\ifdefined` [158](#), [638](#)
- `\ifdim` [400](#)
- `\ifeof` [401](#)
- `\iffalse` [402](#)
- `\iffontchar` [639](#)
- `\ifhbox` [403](#)
- `\ifhmode` [404](#)
- `\ifincsname` [780](#)
- `\ifinner` [405](#)
- `\ifmdir` [1124](#)
- `\ifmmode` [406](#)
- `\ifnum` ... [45](#), [60](#), [89](#), [95](#), [100](#), [164](#), [179](#), [407](#)
- `\ifodd` [408](#)
- `\ifpdfabsdim` [742](#)
- `\ifpdfabsnum` [743](#)
- `\ifpdfprimitive` [744](#)
- `\ifprimitive` [856](#)
- `\iftbox` [1125](#)
- `\iftdir` [1126](#)
- `\iftrue` [409](#)
- `\ifvbox` [410](#)
- `\ifvmode` [411](#)
- `\ifvoid` [412](#)
- `\ifx` [14](#), [21](#),
[39](#), [43](#), [49](#), [90](#), [92](#), [98](#), [123](#), [145](#), [146](#), [413](#)
- `\ifybox` [1127](#)
- `\ifydir` [1128](#)
- `\ignoreligaturesinfont` [951](#)
- `\ignorespaces` [414](#)
- `\IJ` [18962](#)
- `\ij` [18962](#)
- `\immediate` [415](#)
- `in` [197](#)
- `\indent` [416](#)
- `inf` [196](#)
- inherit commands:
 - `.inherit:n` [164](#), [10142](#)
- `\inhibitglue` [1129](#)
- `\inhibitxspcode` [1130](#)
- `\initcatcodetable` [885](#)
- initial commands:
 - `.initial:n` [165](#), [10144](#)
- `\input` [50](#), [159](#), [160](#), [417](#)
- `\inputlineno` [418](#)
- `\insert` [419](#)
- `\insertht` [952](#)
- `\insertpenalties` [420](#)
- int commands:
 - `\c_eight` [72](#), [2930](#),
[2962](#), [4357](#), [4420](#), [6005](#), [6014](#), [6142](#),
[11465](#), [12312](#), [12344](#), [15033](#), [15415](#),
[15428](#), [15786](#), [15788](#), [15790](#), [15849](#)
 - `\c_eleven` [72](#),
[2936](#), [2968](#), [4420](#), [13764](#), [13839](#), [13880](#)
 - `\c_fifteen` [72](#), [2944](#), [2976](#), [4420](#)
 - `\c_five` [72](#), [2924](#), [2956](#),
[3066](#), [4420](#), [6143](#), [11712](#), [11725](#),
[11738](#), [11745](#), [14078](#), [14673](#), [15040](#)
 - `\c_four`
[72](#), [2922](#), [2954](#), [4420](#), [5120](#), [6143](#),
[11153](#), [11159](#), [11797](#), [13080](#), [13596](#),
[14512](#), [14528](#), [14713](#), [14754](#), [14812](#),
[14992](#), [15588](#), [15594](#), [15597](#), [15866](#),
[15993](#), [16137](#), [16210](#), [16420](#), [16718](#)
 - `\c_fourteen` [72](#), [2942](#), [2974](#), [4420](#)
 - `\int_abs:n` [62](#), [62](#), [3662](#)
 - `\int_add:Nn`
[64](#), [64](#), [3788](#), [11100](#), [11113](#), [11153](#)
 - `\int_case:nn`
[66](#), [66](#), [346](#), [3896](#), [4059](#), [4065](#)
 - `\int_case:nnTF` .. [23](#), [66](#), [66](#), [3896](#),
[3901](#), [3906](#), [6854](#), [7362](#), [18581](#), [18636](#)
 - `\int_compare:nNnTF` . [64](#), [65](#), [65](#), [66](#),
[66](#), [67](#), [67](#), [2687](#), [3736](#), [3742](#), [3888](#),
[3920](#), [3971](#), [3979](#), [3988](#), [3994](#), [4006](#),
[4009](#), [4020](#), [4055](#), [4143](#), [4149](#), [4155](#),
[4175](#), [4329](#), [4348](#), [4350](#), [4392](#), [4695](#),
[5075](#), [5105](#), [5119](#), [5123](#), [5148](#), [5740](#),
[5747](#), [5984](#), [5986](#), [5995](#), [6234](#), [6767](#),
[6774](#), [7401](#), [7403](#), [7408](#), [7417](#), [7437](#),
[10596](#), [10889](#), [10971](#), [11101](#), [13186](#),
[16217](#), [16326](#), [16328](#), [16609](#), [16713](#),
[17512](#), [17840](#), [17847](#), [17848](#), [17849](#),
[17869](#), [17995](#), [18319](#), [18321](#), [18324](#),
[18344](#), [18469](#), [18501](#), [18504](#), [18540](#),
[18543](#), [18550](#), [18563](#), [18680](#), [19077](#)
 - `\int_compare:nTF` [65](#), [65](#),
[67](#), [67](#), [67](#), [67](#), [74](#), [185](#), [356](#), [3835](#),
[3943](#), [3951](#), [3960](#), [3966](#), [10833](#), [10950](#)
 - `\int_compare_p:n` [65](#), [65](#), [3835](#)
 - `\int_compare_p:nNn`
[20](#), [65](#), [65](#), [3888](#), [18527](#),
[18623](#), [18624](#), [18625](#), [18671](#), [19104](#),
[19105](#), [19131](#), [19132](#), [19260](#), [19288](#)
 - `\int_const:Nn` [63](#), [63](#),
[3095](#), [3114](#), [3734](#), [4358](#), [4359](#), [4360](#),
[4361](#), [4362](#), [4363](#), [4364](#), [4365](#), [4366](#),
[4367](#), [4368](#), [4369](#), [4370](#), [4371](#), [4420](#),
[4421](#), [4422](#), [4423](#), [4424](#), [4425](#), [4426](#),

- 4427, 4428, 4429, 4430, 4431, 4432,
 4433, 4434, 4435, 4440, 4442, 4443,
 4444, 4445, 4446, 4447, 4448, 4449,
 10883, 11228, 11331, 11332, 11333,
 11335, 11336, 11337, 11340, 11341,
 11342, 11946, 11947, 11948, 11949,
 11950, 11951, 11952, 11953, 11954,
 11955, 11956, 11957, 11958, 11959,
 11960, 16494, 16495, 16496, 16779,
 19225, 19227, 19229, 19230, 19231
 \int_decr:N 64, 64, 3800
 \int_div_round:nn 62, 62, 3694
 \int_div_truncate:nn
 63, 63, 63, 3694, 4070, 4168, 4188,
 18702, 18708, 18710, 18720, 19228
 \int_do_until:nn 67, 67, 3941
 \int_do_until:nNnn 66, 66, 3969
 \int_do_while:nn 67, 67, 3941
 \int_do_while:nNnn 66, 66, 3969
 \int_eval:n 14, 26, 26, 62, 62, 62, 62,
 62, 63, 64, 65, 65, 66, 73, 74, 160,
 274, 349, 501, 575, 617, 640, 640,
 642, 771, 1884, 1900, 3654, 3899,
 3904, 3909, 3914, 4024, 4052, 4138,
 4140, 4270, 4280, 4315, 4326, 4332,
 4343, 4374, 4419, 4702, 5439, 5444,
 5733, 5741, 5749, 6055, 6102, 6139,
 6760, 6768, 6776, 6842, 7335, 7344,
 7395, 7405, 7419, 7426, 7441, 7855,
 10819, 10936, 11236, 13165, 17074,
 17075, 17852, 17871, 17878, 17956,
 17998, 18117, 18687, 18720, 18722
 \int_from_alph:n 70, 70, 4313
 \int_from_base:nn
 71, 71, 4330, 4353, 4355, 4357
 \int_from_bin:n 70, 70, 4352
 \int_from_hex:n 71, 71, 4352
 \int_from_oct:n 71, 71, 4352
 \int_from_roman:n 71, 71, 4372
 \int_gadd:Nn 64, 3788
 \int_gdecr:N 64, 3800, 4050,
 5401, 6819, 7296, 7725, 13434, 17801
 \int_gincr:N 64, 3800, 4029,
 4036, 5396, 6811, 7290, 7720, 13429,
 17792, 20173, 20302, 20358, 20403
 .int_gset:N 165, 10152
 \int_gset:Nn
 ... 64, 3739, 3745, 3812, 8988, 20287
 \int_gset_eq:NN 63, 3778, 20295
 \int_gsub:Nn 64, 3788
 \int_gzero:N 63, 3768, 3775
 \int_gzero_new:N 63, 3772
 \int_if_even:nTF 66, 3925
 \int_if_even_p:n 66, 3925
 \int_if_exist:NTF
 . 64, 64, 3773, 3775, 3784, 4386, 4390
 \int_if_exist_p:N 64, 64, 3784
 \int_if_odd:nTF .. 66, 66, 3925, 14651
 \int_if_odd_p:n 66, 66, 3925
 \int_incr:N . 64, 64, 3800, 9950, 11119
 \int_log:N 207, 207, 17825
 \int_log:n 207, 207, 17828
 \int_max:nn
 63, 63, 720, 3662, 14512, 15581
 \int_min:nn 63, 63, 3662
 \int_mod:nn
 .. 63, 63, 3694, 4060, 4159, 4179,
 10598, 16605, 17865, 18722, 19226
 \int_new:N 63, 63, 63,
 2790, 3726, 3738, 3744, 3773, 3775,
 4459, 4460, 4461, 4462, 9769, 10993,
 10995, 10996, 10997, 10998, 16694,
 16695, 16696, 16697, 16698, 16699,
 16700, 16701, 16702, 16703, 16704,
 19477, 20222, 20305, 20306, 20384
 \int_rand:nn
 . 207, 207, 722, 767, 17513, 17518,
17830, 17884, 17989, 18051, 19170
 .int_set:N 165, 10152
 \int_set:Nn 64, 64, 3812, 7864, 7865,
 7867, 9954, 10875, 10877, 10977,
 10979, 10994, 11007, 11038, 11044,
 11085, 11098, 11127, 16707, 16709,
 16711, 16734, 16735, 16750, 16758,
 16759, 16771, 16772, 16784, 16802
 \int_set_eq:NN
 . 63, 63, 3778, 7866, 11058, 16751,
 16783, 16785, 16793, 16805, 20283
 \int_show:N 71, 71, 762, 762, 4408, 17826
 \int_show:n 71, 71, 501, 762, 4418, 17829
 \int_step_function:nnnN .. 68, 68,
 342, 620, 3158, 3163, 3166, 3997, 4049
 \int_step_inline:nnnn
 68, 68, 621, 4027, 10762, 10891, 10901
 \int_step_variable:nnnN 68, 68, 4027
 \int_sub:Nn 64, 64, 3788, 11159
 \int_to_Alph:n 69, 69, 70, 4073
 \int_to_alph:n 69, 69, 69, 69, 70, 4073
 \int_to_arabic:n 68, 68, 4052
 \int_to_Base:n 70
 \int_to_base:n 70
 \int_to_Base:nn ... 70, 71, 4137, 4264
 \int_to_base:nn
 ... 70, 70, 71, 4137, 4260, 4262, 4266
 \int_to_bin:n 69, 69, 70, 70, 4259
 \int_to_Hex:n 70, 70, 71, 4259
 \int_to_hex:n 70, 70, 70, 71, 4259
 \int_to_oct:n 70, 70, 71, 4259

`\int_to_Roman:n` 70, 70, 71, [4267](#)
`\int_to_roman:n` . . 70, 70, 70, 71, [4267](#)
`\int_to_symbols:nnn`
. 69, [69](#), 69, [4053](#), 4075, 4107
`\int_until_do:nn` 67, 67, [3941](#)
`\int_until_do:nNnn` 67, 67, [3969](#)
`\int_use:N`
. . . 62, [64](#), 64, 64, 569, 574, [3817](#),
4032, 4039, 5397, 5399, 6812, 6818,
7291, 7293, 7719, 7727, 8904, 9371,
9955, 10877, 10973, 11242, 13430,
13433, 16261, 17794, 20175, 20205,
20361, 20367, 20374, 20406, 20414
`\int_while_do:nn` 67, 67, [3941](#)
`\int_while_do:nNnn` 67, 67, [3969](#)
`\int_zero:N`
63, 63, [3768](#), 3773, 9947, 11060, 11147
`int_zero:N` 63
`\int_zero_new:N` 63, 63, [3772](#)
`\c_max_int` 72, [4448](#), 7841, 7847
`\c_nine`
. . 72, 580, 2932, 2964, 3067, [4420](#),
6115, 6153, 11980, 12010, 12219,
12239, 12266, 12280, 12315, 12342,
12370, 12386, 12402, 12420, 12499,
12515, 13890, 15588, 15594, 15597
`\c_one` 72, 352, 567, 567,
567, 567, 567, 570, 570, 594, 689,
2916, 2948, 3065, 3710, 3801, 3803,
4407, [4420](#), 5076, 5133, 5447, 5741,
5747, 6037, 6061, 6144, 6768, 6774,
6848, 7351, 7355, 7405, 7844, 7867,
8956, 8988, 9224, 9230, 9599, 10628,
10764, 10770, 10833, 10875, 10903,
10909, 10950, 11007, 11038, 11086,
11259, 11482, 11690, 11775, 11776,
11779, 11908, 11933, 12229, 12292,
12302, 12371, 12387, 12391, 12428,
13037, 13042, 13047, 13055, 13056,
13057, 13058, 13465, 13561, 13570,
13574, 13690, 13715, 13754, 13808,
13811, 13833, 13837, 13857, 13881,
13897, 13968, 14001, 14090, 14092,
14119, 14153, 14216, 14220, 14224,
14273, 14278, 14307, 14374, 14386,
14537, 14601, 14603, 14652, 14653,
14728, 14764, 14796, 14843, 14938,
15034, 15036, 15081, 15377, 15387,
15408, 15410, 15417, 15430, 15433,
15443, 15446, 15519, 15557, 15605,
15944, 15970, 15990, 16049, 16071,
16155, 16217, 16278, 16283, 16556,
16573, 16577, 16584, 16708, 16717,
16719, 16734, 16735, 16771, 16772,
16784, 16800, 16805, 16867, 16892,
16893, 16894, 16902, 16903, 16937,
16938, 16947, 16948, 16957, 16958,
17852, 17962, 17995, 17998, 19184
`\c_seven` 72,
2928, 2960, [4420](#), 6076, 6086, 6142,
12351, 13785, 13892, 16007, 16025
`\c_six` 72, 2926, 2958, [4420](#), 6142
`\c_sixteen` . . 72, 4355, [4420](#), 10753,
10833, 10857, 10950, 11353, 11418,
11452, 11884, 15055, 15416, 15421,
16328, 16330, 16336, 16377, 16572
`\c_ten` 72, 580, 2934, 2966, 3057, 4196,
4229, [4420](#), 5153, 13889, 14813, 15988
`\c_thirteen` 72, 2940,
2972, 3054, 3068, [4420](#), 15803, 15840
`\c_three` 72, 2920, 2952, [4420](#), 5120,
6143, 11261, 12128, 12168, 12812,
13594, 13884, 15534, 16029, 16714
`\g_tmpa_int` 72, [4459](#)
`\l_tmpa_int` 2, 72, [4459](#)
`\g_tmpb_int` 72, [4459](#)
`\l_tmpb_int` 2, 72, [4459](#)
`\c_twelve` 72, 2938, 2970, [4420](#)
`\c_two` 72, 2918, 2950,
3710, 4353, [4420](#), 6144, 9225, 10603,
11255, 11260, 13165, 13297, 13741,
13872, 13887, 13890, 13892, 14072,
14148, 14151, 14153, 14169, 14398,
14548, 14652, 14653, 14667, 14749,
14946, 15067, 15356, 15387, 15403,
15474, 15504, 15594, 15866, 15876,
15919, 15944, 15990, 15992, 16078,
16100, 16380, 16709, 16714, 16717,
16721, 16727, 19181, 19182, 19183
`\c_zero` 72,
257, 265, 265, 265, 265, 288, 567,
567, 567, 567, 567, 567, 570, 570,
575, 575, 575, 590, 594, 689, 771,
[1341](#), 1595, 1597, 2300, 2786, 2788,
2914, 2946, 3058, 3073, 3292, 3705,
3736, 3768, 3769, 3822, 3830, 4006,
4009, 4143, 4149, 4377, 4407, [4420](#),
4695, 4774, 5105, 5740, 5844, 5854,
5867, 5873, 5984, 6057, 6059, 6092,
6144, 6271, 6319, 6342, 6353, 6361,
6373, 6767, 7401, 7417, 7437, 7853,
7854, 10605, 10763, 10902, 11261,
11417, 11451, 11476, 11694, 11698,
11700, 11704, 11708, 11721, 11734,
11741, 11754, 11766, 11777, 11778,
11787, 11790, 11800, 11891, 11896,
12121, 12162, 12264, 12333, 12380,
12410, 12453, 12458, 12480, 12529,

- 12543, 12590, 12800, 13038, 13043,
 13048, 13054, 13120, 13289, 13530,
 13535, 13806, 14090, 14242, 14277,
 14279, 14280, 14281, 14968, 14972,
 14986, 15044, 15050, 15064, 15256,
 15268, 15286, 15309, 15341, 15424,
 15425, 15439, 15441, 15459, 15489,
 15557, 15855, 15988, 15994, 16028,
 16042, 16047, 16326, 16351, 16584,
 16783, 17512, 17847, 17958, 19288
- int internal commands:
- __int_abs:N [3662](#)
 - __int_case:nnTF [3896](#)
 - __int_case:nw [3896](#)
 - __int_case_end:nw [3896](#)
 - __int_compare:nnN [338](#), [3835](#)
 - __int_compare:NNw ... [338](#), [338](#), [3835](#)
 - __int_compare:Nw . [337](#), [337](#), [338](#), [3835](#)
 - __int_compare:w [337](#), [3835](#)
 - __int_compare_!=:NNw [3835](#)
 - __int_compare_<:NNw [3835](#)
 - __int_compare_<=:NNw [3835](#)
 - __int_compare_=:NNw [3835](#)
 - __int_compare_==:NNw [3835](#)
 - __int_compare_>:NNw [3835](#)
 - __int_compare_>=:NNw [3835](#)
 - __int_compare_end=:NNw .. [338](#), [3835](#)
 - __int_constdef:Nw [3734](#)
 - __int_div_truncate:NwNw [3694](#)
 - __int_eval:w [74](#), [74](#), [332](#),
[333](#), [337](#), [557](#), [567](#), [569](#), [569](#), [569](#),
[582](#), [596](#), [628](#), [636](#), [636](#), [637](#), [640](#),
[644](#), [682](#), [1858](#), [2734](#), [2906](#), [2907](#),
[2910](#), [2979](#), [2980](#), [2983](#), [2988](#), [2989](#),
[2992](#), [2997](#), [2998](#), [3001](#), [3006](#), [3007](#),
[3010](#), [3042](#), [3043](#), [3049](#), [3649](#), [3656](#),
[3660](#), [3665](#), [3673](#), [3674](#), [3681](#), [3682](#),
[3696](#), [3698](#), [3699](#), [3716](#), [3719](#), [3720](#),
[3721](#), [3750](#), [3789](#), [3791](#), [3813](#), [3838](#),
[3872](#), [3890](#), [3927](#), [3935](#), [4000](#), [4001](#),
[4002](#), [4196](#), [4223](#), [4229](#), [4256](#), [5978](#),
[5991](#), [6009](#), [6014](#), [6037](#), [6038](#), [6050](#),
[6086](#), [11255](#), [11353](#), [11356](#), [11476](#),
[11762](#), [11766](#), [11778](#), [11779](#), [11807](#),
[11890](#), [11894](#), [11933](#), [12121](#), [12126](#),
[12164](#), [12254](#), [12265](#), [12314](#), [12345](#),
[12351](#), [12352](#), [12398](#), [12408](#), [12410](#),
[12426](#), [12428](#), [12451](#), [12453](#), [12590](#),
[12809](#), [13082](#), [13259](#), [13654](#), [13662](#),
[13683](#), [13685](#), [13706](#), [13708](#), [13717](#),
[13719](#), [13741](#), [13748](#), [13754](#), [13764](#),
[13766](#), [13839](#), [13841](#), [13857](#), [13859](#),
[13863](#), [13879](#), [13919](#), [13927](#), [13929](#),
[13931](#), [13933](#), [13935](#), [13937](#), [13939](#),
[13958](#), [13960](#), [13970](#), [13972](#), [13998](#),
[14001](#), [14009](#), [14011](#), [14031](#), [14034](#),
[14037](#), [14040](#), [14048](#), [14051](#), [14054](#),
[14057](#), [14064](#), [14066](#), [14072](#), [14080](#),
[14082](#), [14084](#), [14090](#), [14110](#), [14112](#),
[14121](#), [14123](#), [14144](#), [14165](#), [14169](#),
[14181](#), [14184](#), [14187](#), [14190](#), [14193](#),
[14196](#), [14199](#), [14202](#), [14206](#), [14218](#),
[14222](#), [14226](#), [14229](#), [14250](#), [14252](#),
[14254](#), [14264](#), [14303](#), [14305](#), [14314](#),
[14336](#), [14341](#), [14343](#), [14350](#), [14353](#),
[14356](#), [14359](#), [14362](#), [14365](#), [14374](#),
[14386](#), [14394](#), [14396](#), [14406](#), [14408](#),
[14415](#), [14424](#), [14426](#), [14429](#), [14432](#),
[14435](#), [14438](#), [14451](#), [14453](#), [14461](#),
[14463](#), [14471](#), [14473](#), [14482](#), [14485](#),
[14488](#), [14495](#), [14510](#), [14528](#), [14531](#),
[14587](#), [14601](#), [14603](#), [14609](#), [14622](#),
[14624](#), [14626](#), [14650](#), [14666](#), [14673](#),
[14674](#), [14697](#), [14713](#), [14717](#), [14762](#),
[14764](#), [14802](#), [14813](#), [14832](#), [14834](#),
[14836](#), [14849](#), [14862](#), [14867](#), [14869](#),
[14875](#), [14892](#), [14893](#), [14894](#), [14895](#),
[14896](#), [14897](#), [14902](#), [14904](#), [14906](#),
[14908](#), [14910](#), [14914](#), [14916](#), [14918](#),
[14920](#), [14922](#), [14924](#), [14946](#), [14954](#),
[15032](#), [15036](#), [15089](#), [15300](#), [15321](#),
[15323](#), [15326](#), [15329](#), [15332](#), [15335](#),
[15351](#), [15377](#), [15387](#), [15403](#), [15555](#),
[15587](#), [15596](#), [15778](#), [15779](#), [15802](#),
[15812](#), [15821](#), [15839](#), [15848](#), [15855](#),
[15866](#), [15876](#), [15909](#), [15919](#), [15944](#),
[15953](#), [15970](#), [16006](#), [16023](#), [16025](#),
[16037](#), [16038](#), [16078](#), [16089](#), [16100](#),
[16158](#), [16283](#), [16420](#), [16513](#), [16519](#),
[16535](#), [16556](#), [16867](#), [17835](#), [17836](#)
 - __int_eval_end:
. [74](#), [74](#), [74](#), [74](#), [1858](#), [2734](#), [2906](#),
[2907](#), [2910](#), [2979](#), [2980](#), [2983](#), [2988](#),
[2989](#), [2992](#), [2997](#), [2998](#), [3001](#), [3006](#),
[3007](#), [3010](#), [3649](#), [3656](#), [3660](#), [3665](#),
[3700](#), [3716](#), [3722](#), [3750](#), [3789](#), [3791](#),
[3813](#), [3890](#), [3927](#), [3935](#), [4196](#), [4223](#),
[4229](#), [4256](#), [11255](#), [11479](#), [11807](#),
[11904](#), [11908](#), [13082](#), [13259](#), [13576](#),
[13741](#), [13863](#), [13898](#), [14086](#), [14396](#),
[14531](#), [15351](#), [15403](#), [15588](#), [15597](#),
[15866](#), [15876](#), [15919](#), [15944](#), [15970](#),
[16038](#), [16514](#), [16520](#), [16536](#), [16556](#)
 - __int_from_alpha:N [349](#), [4313](#)
 - __int_from_alpha:nnN [349](#), [4313](#)
 - __int_from_base:N [349](#), [4330](#)
 - __int_from_base:nnN [349](#), [4330](#)
 - __int_from_roman:NN [4372](#)

<code>\c__int_from_roman_C_int</code>	4358	<code>__int_to_roman_Q:w</code>	4267
<code>\c__int_from_roman_c_int</code>	4358	<code>__int_to_Roman_v:w</code>	4267
<code>\c__int_from_roman_D_int</code>	4358	<code>__int_to_roman_v:w</code>	4267
<code>\c__int_from_roman_d_int</code>	4358	<code>__int_to_Roman_x:w</code>	4267
<code>__int_from_roman_error:w</code>	4372	<code>__int_to_roman_x:w</code>	4267
<code>\c__int_from_roman_I_int</code>	4358	<code>__int_to_symbols:nnnn</code>	4053
<code>\c__int_from_roman_i_int</code>	4358	<code>__int_value:w</code>	
<code>\c__int_from_roman_L_int</code>	4358 74 , 74 , 74 , 300 , 337 , 356 ,	
<code>\c__int_from_roman_l_int</code>	4358	551 , 557 , 557 , 559 , 569 , 575 , 575 ,	
<code>\c__int_from_roman_M_int</code>	4358	575 , 575 , 575 , 575 , 582 , 585 , 590 ,	
<code>\c__int_from_roman_m_int</code>	4358	590 , 597 , 618 , 626 , 634 , 642 , 703 ,	
<code>\c__int_from_roman_V_int</code>	4358	716 , 1597 , 2644 , 2647 , 2734 , 3042 ,	
<code>\c__int_from_roman_v_int</code>	4358	3043 , 3049 , 3649 , 3656 , 3660 , 3664 ,	
<code>\c__int_from_roman_X_int</code>	4358	3665 , 3672 , 3673 , 3674 , 3680 , 3681 ,	
<code>\c__int_from_roman_x_int</code>	4358	3682 , 3696 , 3698 , 3699 , 3716 , 3719 ,	
<code>__int_maxmin:wwN</code>	3662	3720 , 3721 , 3838 , 3842 , 3872 , 4000 ,	
<code>__int_mod:ww</code>	3694	4001 , 4002 , 4223 , 4256 , 4552 , 5978 ,	
<code>__int_pass_signs:wn</code>		5979 , 5991 , 6009 , 6014 , 6036 , 6037 ,	
. 349 , 4303 , 4317 , 4334		6038 , 6050 , 6086 , 7989 , 8021 , 8022 ,	
<code>__int_pass_signs_end:wn</code>	4303	8023 , 8025 , 8156 , 8165 , 8167 , 8172 ,	
<code>__int_rand:ww</code>	17830	8173 , 8174 , 8175 , 8179 , 8180 , 8181 ,	
<code>__int_rand_narrow:n</code>	17830	8182 , 8207 , 8213 , 8215 , 8217 , 8219 ,	
<code>__int_rand_narrow:nn</code>		8224 , 8229 , 8234 , 8241 , 8248 , 8379 ,	
. 763 , 17851 , 17857 , 17870		8409 , 8410 , 8449 , 8468 , 8474 , 8644 ,	
<code>__int_rand_narrow:nnn</code>	17830	8753 , 11315 , 11316 , 11317 , 11318 ,	
<code>__int_rand_narrow:nnnn</code>	763 , 17830	11319 , 11370 , 11441 , 11464 , 11476 ,	
<code>__int_step:NnnnN</code>	3997	11762 , 11876 , 11890 , 11892 , 11894 ,	
<code>__int_step:NNnnnn</code>	4027	11897 , 11933 , 12067 , 12098 , 12099 ,	
<code>__int_step:wwwN</code>	3997	12112 , 12121 , 12249 , 12254 , 12256 ,	
<code>__int_to_Base:nn</code>	4137	12265 , 12269 , 12306 , 12314 , 12317 ,	
<code>__int_to_base:nn</code>	4137	12323 , 12334 , 12345 , 12351 , 12352 ,	
<code>__int_to_Base:nnN</code>	4137	12355 , 12398 , 12408 , 12410 , 12426 ,	
<code>__int_to_base:nnN</code>	4137	12428 , 12451 , 12465 , 12542 , 12544 ,	
<code>__int_to_Base:nnnN</code>	4137	12590 , 12669 , 13176 , 13274 , 13599 ,	
<code>__int_to_base:nnnN</code>	4137	13600 , 13601 , 13603 , 13654 , 13657 ,	
<code>__int_to_Letter:n</code>	4137	13660 , 13683 , 13685 , 13706 , 13708 ,	
<code>__int_to_letter:n</code>	4137	13717 , 13719 , 13723 , 13741 , 13748 ,	
<code>__int_to_roman:N</code>	4267	13754 , 13764 , 13766 , 13780 , 13788 ,	
<code>__int_to_roman:w</code>		13796 , 13839 , 13841 , 13857 , 13859 ,	
. 73 , 73 , 338 , 348 , 1322 ,		13862 , 13865 , 13919 , 13927 , 13929 ,	
3153 , 3176 , 3649 , 3848 , 4270 , 4280 ,		13931 , 13933 , 13935 , 13937 , 13939 ,	
11356 , 12326 , 12358 , 14829 , 15098		13958 , 13960 , 13964 , 13970 , 13972 ,	
<code>__int_to_Roman_aux:N</code> 4279 , 4282 , 4285		13976 , 13998 , 14001 , 14009 , 14011 ,	
<code>__int_to_Roman_c:w</code>	4267	14014 , 14015 , 14016 , 14017 , 14031 ,	
<code>__int_to_roman_c:w</code>	4267	14034 , 14037 , 14040 , 14048 , 14051 ,	
<code>__int_to_Roman_d:w</code>	4267	14054 , 14057 , 14064 , 14066 , 14072 ,	
<code>__int_to_roman_d:w</code>	4267	14080 , 14082 , 14084 , 14110 , 14112 ,	
<code>__int_to_Roman_i:w</code>	4267	14121 , 14123 , 14127 , 14144 , 14165 ,	
<code>__int_to_roman_i:w</code>	4267	14169 , 14181 , 14184 , 14187 , 14190 ,	
<code>__int_to_Roman_l:w</code>	4267	14193 , 14196 , 14199 , 14202 , 14206 ,	
<code>__int_to_roman_l:w</code>	4267	14218 , 14222 , 14226 , 14229 , 14250 ,	
<code>__int_to_Roman_m:w</code>	4267	14252 , 14254 , 14264 , 14288 , 14291 ,	
<code>__int_to_roman_m:w</code>	4267	14303 , 14305 , 14311 , 14314 , 14323 ,	
<code>__int_to_Roman_Q:w</code>	4267	14336 , 14341 , 14343 , 14350 , 14353 ,	

- 14356, 14359, 14362, 14365, 14374,
- 14386, 14394, 14396, 14406, 14408,
- 14415, 14424, 14426, 14429, 14432,
- 14435, 14438, 14451, 14453, 14461,
- 14463, 14471, 14473, 14482, 14485,
- 14488, 14495, 14510, 14528, 14531,
- 14587, 14601, 14603, 14609, 14622,
- 14624, 14626, 14650, 14666, 14673,
- 14674, 14717, 14719, 14720, 14721,
- 14762, 14764, 14795, 14802, 14809,
- 14830, 14832, 14834, 14836, 14849,
- 14853, 14854, 14855, 14856, 14857,
- 14862, 14867, 14869, 14875, 14892,
- 14893, 14894, 14895, 14896, 14897,
- 14902, 14904, 14906, 14908, 14910,
- 14914, 14916, 14918, 14920, 14922,
- 14924, 14946, 14954, 14970, 14975,
- 14979, 15032, 15036, 15089, 15101,
- 15285, 15321, 15323, 15326, 15329,
- 15332, 15335, 15342, 15345, 15347,
- 15351, 15373, 15375, 15403, 15555,
- 15587, 15596, 15778, 15779, 15780,
- 15802, 15812, 15821, 15839, 15848,
- 15855, 15865, 15909, 15918, 15953,
- 16006, 16023, 16078, 16089, 16100,
- 16283, 16350, 16412, 16420, 16422,
- 16424, 16507, 16513, 16519, 16529,
- 16535, 16596, 16605, 16612, 16630,
- 16867, 17534, 17536, 17559, 17561,
- 17586, 17626, 17653, 17661, 17687,
- 17689, 17693, 17695, 17724, 17738,
- 17745, 17835, 17836, 17989, 17998
- \interactionmode 640
- \interlinepenalties 641
- \interlinepenalty 421
- ior commands:
- \ior_close:N
 174, 174, 175, 175, 10638, 10810, 10831
- \ior_get:NN
 175, 175, 176, 179, 10869, 17787
- \ior_get_str:NN 11196
- \ior_if_eof:N 761
- \ior_if_eof:Ntf 176,
 176, 10635, 10656, 10853, 17799, 17806
- \ior_if_eof_p:N 176, 176, 10853
- \ior_list_streams:
 175, 175, 762, 10843, 17814
- \ior_log_streams: ... 206, 206, 17813
- \ior_map_break: 206, 206, 17782, 17800
- \ior_map_break:n 206, 206, 17782
- \ior_map_inline:Nn .. 205, 205, 17786
- \ior_new:N 174, 174, 10777, 10880
- \ior_open:Nn 174, 174, 10779
- \ior_open:NnTF
 174, 174, 10789, 10791, 10792
- \ior_str_get:NN
 .. 176, 176, 176, 10871, 11196, 17789
- \ior_str_map_inline:Nn 205, 205, 17786
- \c_term_ior .. 179, 10753, 10777, 10839
- ior internal commands:
- \l_ior_internal_tl 17786
- __ior_list_streams:Nn . 10843, 10962
- __ior_map_inline:NNn 17786
- __ior_map_inline:NNNn 17786
- __ior_map_inline_loop:NNN ... 17786
- __ior_new:N 534, 10804, 10818
- __ior_open:Nn 179,
 179, 10634, 10655, 10787, 10800, 10808
- __ior_open_aux:Nn 10779
- __ior_open_aux:NnTF 10789
- __ior_open_stream:Nn 10808
- \l_ior_stream_tl
 10759, 10811, 10819, 10827
- \g_ior_streams_prop
 10760, 10828, 10836, 10844
- \g_ior_streams_seq
 .. 10754, 10811, 10837, 10838, 10886
- iow commands:
- \iow_char:N .. 177, 177, 9379, 9381,
 9382, 10590, 10992, 15207, 17062,
 17065, 17090, 17091, 17098, 17099
- \iow_close:N 174, 175, 175, 10927, 10948
- \iow_indent:n 178, 178, 178,
 542, 543, 543, 9325, 10498, 11023,
 11047, 11055, 11195, 11655, 11667
- \l_iow_line_count_int
 178, 178, 178, 544, 10993, 11058, 11086
- \iow_list_streams:
 175, 175, 762, 10960, 17817
- \iow_log:n .. 22, 176, 176, 269, 501,
 1690, 1701, 8977, 8978, 8979, 9103,
 9564, 9593, 10741, 10742, 10743, 10987
- \iow_log_streams: ... 206, 206, 17816
- \iow_new:N 174, 174, 10916
- \iow_newline: 176, 177,
 177, 177, 177, 180, 485, 501, 540,
 544, 8933, 8949, 8951, 10991, 11056
- \iow_now:Nn
 176, 176, 176, 176, 176, 177,
 177, 10981, 10987, 10988, 10989, 10990
- \iow_open:Nn 174, 174, 10922
- \iow_shipout:Nn
 .. 176, 176, 176, 177, 177, 540, 10966
- \iow_shipout_x:Nn
 176, 177, 177, 177, 540, 10963
- \iow_term:n 176, 176, 1690,
 8947, 8983, 8984, 8985, 9564, 10987

- \iow_wrap:nnnN 156,
157, 157, 176, 177, 178, 178, 178,
178, 178, 178, 278, 351, 500, 501,
501, 543, 544, 8925, 8926, 8978,
8984, 9101, 9547, 9585, 11035, 11195
- \c_log_iow 179, 537, 10882, 10987, 10988
- \c_term_iow 179,
537, 10882, 10916, 10956, 10989, 10990
- iow internal commands:
 - \l__iow_current_indentation_int .
..... 10996,
11114, 11130, 11153, 11159, 11161
 - \l__iow_current_indentation_tl ..
..... 10999,
11059, 11112, 11133, 11154, 11160
 - \l__iow_current_line_int
..... 10996, 11060, 11100,
11101, 11113, 11119, 11127, 11147
 - \l__iow_current_line_tl
.... 10999, 11061, 11111, 11117,
11126, 11132, 11146, 11148, 11166
 - \l__iow_current_word_int
..... 10996, 11098, 11100, 11129
 - \l__iow_current_word_tl
..... 10999, 11091,
11092, 11099, 11112, 11118, 11133
 - __iow_indent:n 11023, 11047
 - __iow_indent_error:n . 11023, 11055
 - \l__iow_line_start_bool
.. 11004, 11062, 11108, 11110, 11149
 - __iow_list_streams:Nn 10960
 - __iow_new:N 10918, 10935
 - \l__iow_newline_tl
..... 543, 544, 11003, 11056,
11057, 11084, 11086, 11126, 11146
 - __iow_open:Nn 10922
 - __iow_open_stream:Nn 10922
 - \l__iow_stream_tl
..... 10898, 10928, 10936, 10944
 - \g__iow_streams_prop
..... 10899, 10945, 10953, 10961
 - \g__iow_streams_seq
..... 10884, 10928, 10954, 10955
 - \l__iow_target_count_int
.. 543, 544, 10995, 11058, 11085, 11101
 - __iow_with:Nnn
..... 180, 180, 484, 501, 540,
8954, 8956, 9597, 9599, 10969, 10983
 - __iow_with_aux:nNnn 10969
 - __iow_wrap_end: 11163
 - __iow_wrap_end:w 11135
 - \c__iow_wrap_end_marker_tl
..... 11006, 11069
 - __iow_wrap_indent: 11151
 - __iow_wrap_indent:w 11135
 - \c__iow_wrap_indent_marker_tl ...
..... 11006, 11025
 - __iow_wrap_loop:w
..... 11066, 11089, 11104, 11140
 - \c__iow_wrap_marker_tl
..... 542, 542, 11006, 11092, 11138
 - __iow_wrap_newline: 11142
 - __iow_wrap_newline:w 11135
 - \c__iow_wrap_newline_marker_tl ..
..... 11006, 11045, 11067
 - __iow_wrap_set:Nn 11035
 - __iow_wrap_set_target:
..... 543, 544, 11079, 11124, 11144
 - __iow_wrap_special:w . 11093, 11135
 - \l__iow_wrap_tl
.... 11002, 11050, 11053, 11065,
11068, 11074, 11125, 11145, 11165
 - __iow_wrap_unindent: 11157
 - __iow_wrap_unindent:w 11135
 - \c__iow_wrap_unindent_marker_tl .
..... 11006, 11027
 - __iow_wrap_word: 11094, 11096
 - __iow_wrap_word_fits: 11096
 - __iow_wrap_word_newline: 11096
- J**
- \J 197
- \j 18972
- \jcharwidowpenalty 1131
- \jfam 1132
- \jfont 1133
- \jis 1134
- \jobname 422
- K**
- \k 18976
- \kanjiskip 1135
- \kansuji 1136
- \kansujichar 1137
- \kcatcode 1138
- \kchar 1156
- \kchardef 1157
- \kern 423
- kernel internal commands:
 - \l__kernel_expl_bool
..... 7, 235, 238, 253, 267
 - __kernel_primitive:NN 234,
275, 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, 392, 393, 394,
395, 396, 397, 398, 399, 400, 401,
402, 403, 404, 405, 406, 407, 408,
409, 410, 411, 412, 413, 414, 415,
416, 417, 418, 419, 420, 421, 422,
423, 424, 425, 426, 427, 428, 429,
430, 431, 432, 433, 434, 435, 436,
437, 438, 439, 440, 441, 442, 443,
444, 445, 446, 447, 448, 449, 450,
451, 452, 453, 454, 455, 456, 457,
458, 459, 460, 461, 462, 463, 464,
465, 466, 467, 468, 469, 470, 471,
472, 473, 474, 475, 476, 477, 478,
479, 480, 481, 482, 483, 484, 485,
486, 487, 488, 489, 490, 491, 492,
493, 494, 495, 496, 497, 498, 499,
500, 501, 502, 503, 504, 505, 506,
507, 508, 509, 510, 511, 512, 513,
514, 515, 516, 517, 518, 519, 520,
521, 522, 523, 524, 525, 526, 527,
528, 529, 530, 531, 532, 533, 534,
535, 536, 537, 538, 539, 540, 541,
542, 543, 544, 545, 546, 547, 548,
549, 550, 551, 552, 553, 554, 555,
556, 557, 558, 559, 560, 561, 562,
563, 564, 565, 566, 567, 568, 569,
570, 571, 572, 573, 574, 575, 576,
577, 578, 579, 580, 581, 582, 583,
584, 585, 586, 587, 588, 589, 590,
591, 592, 593, 594, 595, 596, 597,
598, 599, 600, 601, 602, 603, 604,
605, 606, 607, 608, 609, 610, 611,
612, 613, 614, 615, 616, 617, 618,
619, 620, 621, 622, 623, 624, 625,
626, 627, 628, 629, 630, 631, 632,
633, 634, 635, 636, 637, 638, 639,
640, 641, 642, 643, 644, 645, 646,
647, 648, 649, 650, 651, 652, 653,
654, 655, 656, 657, 658, 659, 660,
661, 662, 663, 664, 665, 666, 667,
668, 669, 670, 671, 672, 673, 674,
675, 676, 677, 678, 679, 680, 681,
682, 683, 684, 685, 686, 687, 688,
689, 690, 691, 692, 693, 694, 695,
696, 697, 698, 699, 700, 701, 702,
703, 704, 705, 706, 707, 708, 709,
710, 711, 712, 713, 714, 715, 716,
717, 718, 719, 720, 721, 722, 723,
724, 725, 726, 727, 728, 729, 730,
731, 732, 733, 734, 735, 736, 737,
738, 739, 740, 741, 742, 743, 744,
745, 746, 747, 748, 749, 750, 751,
752, 753, 754, 755, 756, 757, 758,
759, 760, 761, 762, 763, 764, 765,
766, 767, 768, 769, 770, 771, 772,
773, 774, 775, 776, 777, 778, 779,
780, 781, 782, 783, 784, 785, 786,
787, 788, 793, 805, 806, 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, 905, 906,
907, 908, 909, 910, 911, 912, 913,
914, 915, 916, 917, 918, 919, 920,
921, 922, 923, 924, 925, 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,
970, 971, 972, 973, 974, 975, 976,
977, 978, 979, 980, 981, 982, 983,
984, 985, 986, 987, 988, 989, 990,
991, 992, 993, 994, 995, 996, 997,
998, 999, 1000, 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, 1039,
1040, 1041, 1042, 1043, 1044, 1045,
1046, 1047, 1048, 1049, 1050, 1051,
1052, 1053, 1054, 1055, 1056, 1057,
1058, 1059, 1060, 1061, 1062, 1063,
1064, 1065, 1066, 1067, 1068, 1069,
1070, 1071, 1072, 1073, 1074, 1075,
1076, 1077, 1078, 1079, 1080, 1081,

- 1082, 1083, 1084, 1085, 1086, 1087,
- 1088, 1089, 1090, 1091, 1092, 1093,
- 1094, 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, 1155, 1156, 1157, 1158, 1159
- _kernel_register_log:N
 [201](#), [201](#), [762](#), [17111](#), [18068](#),
 [18069](#), [18072](#), [18073](#), [18076](#), [18077](#)
- _kernel_register_show:N
 [22](#), [22](#), [201](#), [351](#),
 [394](#), [742](#), [2018](#), [4711](#), [4804](#), [4868](#), [17112](#)
- keys commands:
- _l_keys_choice_int
 [163](#), [165](#), [167](#), [167](#), [167](#),
 [167](#), [168](#), [9769](#), [9947](#), [9950](#), [9954](#), [9955](#)
- _l_keys_choice_tl [163](#),
 [165](#), [167](#), [167](#), [167](#), [168](#), [9769](#), [9953](#)
- _keys_define:nn
 [162](#), [162](#), [162](#), [9784](#), [10496](#)
- _keys_if_choice_exist:nnnTF
 [171](#), [171](#), [10449](#)
- _keys_if_choice_exist_p:nnn
 [171](#), [171](#), [10449](#)
- _keys_if_exist:nnnTF
 [171](#), [171](#), [525](#), [10442](#), [10459](#)
- _keys_if_exist_p:nn . [171](#), [171](#), [10442](#)
- _l_keys_key_tl ... [169](#), [169](#), [9772](#),
 [9889](#), [9905](#), [10291](#), [10377](#), [10379](#), [10412](#)
- _keys_log:nn [207](#), [207](#), [17887](#)
- _l_keys_path_tl [169](#), [169](#), [9776](#), [9812](#),
 [9831](#), [9840](#), [9849](#), [9853](#), [9867](#), [9882](#),
 [9884](#), [9886](#), [9898](#), [9900](#), [9902](#), [9917](#),
 [9920](#), [9924](#), [9932](#), [9933](#), [9934](#), [9937](#),
 [9951](#), [9972](#), [9977](#), [9986](#), [9990](#), [9997](#),
 [10001](#), [10005](#), [10012](#), [10019](#), [10030](#),
 [10036](#), [10040](#), [10055](#), [10064](#), [10072](#),
 [10107](#), [10273](#), [10280](#), [10303](#), [10306](#),
 [10345](#), [10349](#), [10357](#), [10359](#), [10360](#),
 [10371](#), [10374](#), [10394](#), [10419](#), [10420](#)
- _keys_set:nn [161](#),
 [165](#), [169](#), [169](#), [169](#), [169](#), [170](#), [10007](#),
 [10012](#), [10204](#), [10228](#), [10248](#), [10257](#)
- _keys_set_filter:nnn [171](#), [171](#), [10232](#)
- _keys_set_filter:nnnN
 [171](#), [171](#), [171](#), [10232](#)
- _keys_set_groups:nnn [171](#), [171](#), [10232](#)
- _keys_set_known:nn .. [170](#), [170](#), [10214](#)
- _keys_set_known:nnN
 [170](#), [170](#), [170](#), [170](#), [520](#), [10214](#)
- _keys_show:nn
 [171](#), [171](#), [764](#), [10457](#), [17888](#)
- _l_keys_value_tl [169](#), [169](#),
 [9782](#), [10055](#), [10348](#), [10351](#), [10353](#),
 [10361](#), [10381](#), [10390](#), [10404](#), [10414](#)
- keys internal commands:
- _keys_bool_set:Nn
 ... [9878](#), [10081](#), [10083](#), [10085](#), [10087](#)
- _keys_bool_set_inverse:Nn
 ... [9894](#), [10089](#), [10091](#), [10093](#), [10095](#)
- _keys_check_groups: . [10307](#), [10315](#)
- _keys_choice_find:n .. [9911](#), [10417](#)
- _keys_choice_make:
 [9881](#), [9897](#), [9910](#), [9941](#), [10097](#)
- _keys_choice_make:N [9910](#)
- _keys_choice_make_aux:N [9910](#)
- _keys_choices_make:nn
 ... [9940](#), [10099](#), [10101](#), [10103](#), [10105](#)
- _keys_choices_make:Nnn [9940](#)
- _keys_cmd_set:nn ... [9882](#), [9884](#),
 [9886](#), [9898](#), [9900](#), [9902](#), [9933](#), [9934](#),
 [9951](#), [9960](#), [10005](#), [10012](#), [10072](#), [10107](#)
- _c_keys_code_root_tl
 .. [9762](#), [9962](#), [9964](#), [10001](#), [10357](#),
 [10360](#), [10377](#), [10379](#), [10387](#), [10389](#),
 [10401](#), [10403](#), [10445](#), [10453](#), [10464](#)
- _c_keys_default_root_tl
 [9762](#), [9972](#), [9977](#), [10345](#), [10349](#)
- _keys_default_set:n [9891](#),
 [9907](#), [9967](#), [10117](#), [10119](#), [10121](#), [10123](#)
- _keys_define:n [9789](#), [9793](#)
- _keys_define:nn [9789](#), [9793](#)
- _keys_define:nnn [9784](#)
- _keys_define_aux:nn [9793](#)
- _keys_define_code:n ... [9807](#), [9857](#)
- _keys_define_code:w [9857](#)
- _keys_execute:
 .. [10284](#), [10311](#), [10333](#), [10337](#), [10355](#)
- _keys_execute:nn [10355](#), [10419](#), [10420](#)
- _keys_execute_unknown: [10355](#)
- _l_keys_filtered_bool [9778](#),
 [10246](#), [10255](#), [10310](#), [10331](#), [10336](#)
- _keys_find_key_module:w [10261](#)
- _l_keys_groups_clist
 . [9771](#), [9983](#), [9984](#), [9991](#), [10305](#), [10320](#)
- _c_keys_groups_root_tl
 [9762](#), [9986](#), [9990](#), [10303](#), [10306](#)
- _keys_groups_set:n ... [9981](#), [10141](#)
- _keys_inherit:n [9994](#), [10143](#)
- _c_keys_inherit_root_tl
 [9762](#), [9997](#), [10371](#), [10374](#)

- __keys_initialise:n [9999](#), [10145](#), [10147](#), [10149](#), [10151](#)
- __keys_meta_make:n ... [10003](#), [10161](#)
- __keys_meta_make:nn .. [10003](#), [10163](#)
- \l_keys_module_tl [9773](#), [9785](#), [9788](#), [9790](#), [9833](#), [9834](#), [9840](#), [10008](#), [10205](#), [10208](#), [10210](#), [10264](#), [10269](#), [10279](#), [10285](#), [10293](#), [10295](#), [10387](#), [10389](#), [10394](#)
- __keys_multichoice_find:n [9913](#), [10417](#)
- __keys_multichoice_make: [9910](#), [9943](#), [10165](#)
- __keys_multichoices_make:nn ... [9940](#), [10167](#), [10169](#), [10171](#), [10173](#)
- \l_keys_no_value_bool [9774](#), [9795](#), [9800](#), [9859](#), [10052](#), [10061](#), [10263](#), [10268](#), [10343](#), [10413](#)
- \l_keys_only_known_bool [9775](#), [10227](#), [10229](#), [10367](#)
- __keys_parent:n [9917](#), [9920](#), [9924](#), [10371](#), [10374](#), [10424](#)
- __keys_parent:w [10424](#)
- __keys_property_find:n .. [9805](#), [9816](#)
- __keys_property_find:w [9816](#)
- __keys_property_search:w [9841](#), [9845](#), [9854](#)
- \l_keys_property_tl [9777](#), [9806](#), [9809](#), [9812](#), [9818](#), [9819](#), [9825](#), [9837](#), [9850](#), [9862](#), [9863](#), [9866](#), [9870](#)
- \c_keys_props_root_tl [9768](#), [9806](#), [9863](#), [9870](#), [10080](#), [10082](#), [10084](#), [10086](#), [10088](#), [10090](#), [10092](#), [10094](#), [10096](#), [10098](#), [10100](#), [10102](#), [10104](#), [10106](#), [10108](#), [10110](#), [10112](#), [10114](#), [10116](#), [10118](#), [10120](#), [10122](#), [10124](#), [10126](#), [10128](#), [10130](#), [10132](#), [10134](#), [10136](#), [10138](#), [10140](#), [10142](#), [10144](#), [10146](#), [10148](#), [10150](#), [10152](#), [10154](#), [10156](#), [10158](#), [10160](#), [10162](#), [10164](#), [10166](#), [10168](#), [10170](#), [10172](#), [10174](#), [10176](#), [10178](#), [10180](#), [10182](#), [10184](#), [10186](#), [10188](#), [10190](#), [10192](#), [10194](#), [10196](#), [10198](#), [10200](#), [10202](#)
- __keys_remove_spaces:n [9788](#), [9818](#), [9951](#), [10208](#), [10277](#), [10419](#), [10435](#), [10445](#), [10453](#), [10462](#), [10464](#), [10468](#)
- __keys_remove_spaces:w [10435](#)
- \l_keys_selective_bool ... [9778](#), [10245](#), [10249](#), [10254](#), [10258](#), [10282](#)
- \l_keys_selective_seq [9780](#), [10247](#), [10256](#), [10318](#)
- __keys_set:n [10209](#), [10261](#)
- __keys_set:nn [10209](#), [10261](#)
- __keys_set:nnn [10204](#)
- __keys_set_aux: [10261](#)
- __keys_set_aux:nnn [10261](#)
- __keys_set_filter:nnnnN [10232](#)
- __keys_set_known:nnnN [10214](#)
- __keys_set_selective: [10261](#)
- __keys_show:N [10457](#)
- __keys_store_unused: [10312](#), [10332](#), [10338](#), [10355](#)
- \l_keys_tmp_bool [9783](#), [10317](#), [10324](#), [10329](#)
- \c_keys_type_root_tl [9762](#), [9917](#), [9920](#), [9932](#)
- __keys_undefine: . [9996](#), [10013](#), [10199](#)
- \l_keys_unused_clist [519](#), [9781](#), [10215](#), [10219](#), [10221](#), [10222](#), [10233](#), [10237](#), [10239](#), [10240](#), [10410](#)
- __keys_validate_cleanup:w ... [10023](#)
- __keys_validate_forbidden: .. [10023](#)
- __keys_validate_required: ... [10023](#)
- \c_keys_validate_root_tl [9762](#), [10030](#), [10036](#), [10040](#), [10359](#)
- __keys_value_or_default:n [10281](#), [10341](#)
- __keys_value_requirement:nn ... [10023](#), [10201](#), [10203](#)
- __keys_variable_set:NnnN [10069](#), [10109](#), [10111](#), [10113](#), [10115](#), [10125](#), [10127](#), [10129](#), [10131](#), [10133](#), [10135](#), [10137](#), [10139](#), [10153](#), [10155](#), [10157](#), [10159](#), [10175](#), [10177](#), [10179](#), [10181](#), [10183](#), [10185](#), [10187](#), [10189](#), [10191](#), [10193](#), [10195](#), [10197](#)
- keyval commands:
 - \keyval_parse:NNn [172](#), [172](#), [172](#), [9628](#), [9789](#), [10209](#)
- keyval internal commands:
 - __keyval_action: [9707](#)
 - __keyval_def:Nn [9709](#), [9725](#), [9750](#)
 - __keyval_def_aux:n [9750](#)
 - __keyval_def_aux:w [9750](#)
 - __keyval_empty_key: [9744](#), [9748](#)
 - \l_keyval_key_tl [9625](#), [9709](#), [9710](#), [9721](#), [9729](#)
 - __keyval_loop:NNw .. [9631](#), [9637](#), [9697](#)
 - __keyval_sanitise_aux:w [9641](#)
 - __keyval_sanitise_comma: [9636](#), [9641](#)
 - __keyval_sanitise_comma_auxi:w [9641](#)
 - __keyval_sanitise_comma_auxii:w [9641](#)
 - __keyval_sanitise_equals: [9635](#), [9641](#)
 - __keyval_sanitise_equals_auxi:w [9641](#)

- `__keyval_sanitise_equals_-auxii:w` 9641
- `\l_keyval_sanitise_tl` 9627, 9634, 9638, 9647, 9649, 9653, 9660, 9662, 9671, 9673, 9677, 9684, 9686, 9695
- `__keyval_split:NNw` 9702, 9707
- `__keyval_split_tidy:w` 9707
- `__keyval_split_value:NNw` 9707
- `\l_keyval_value_tl` . 9625, 9725, 9730
- `\kuten` 1139, 1158
- L**
- `\L` 18963
- `\l` 18963
- `\l3kernel` 19351
- `\l3kernel.charcat` 220
- `\l3kernel.charcat` 220, 19366
- `\l3kernel.strcmp` 220
- `\l3kernel.strcmp` 220, 19356
- `\label` 19151
- `\language` 424
- `\lastallocatedread` 6276, 6277
- `\lastallocatedtoks` 16765
- `\lastbox` 425
- `\lastkern` 426
- `\lastlinefit` 642
- `\lastnamedcs` 886
- `\lastnodetype` 643
- `\lastpenalty` 427
- `\lastsavedboxresourceindex` 953
- `\lastsavedimageresourceindex` 954
- `\lastsavedimageresourcepages` 955
- `\lastskip` 428
- `\lastxpos` 956
- `\lastypos` 957
- `\latelua` 887
- LaTeX3 error commands:
 - `\LaTeX3_error:` 498, 498
- `\lccode` 166, 181, 194, 196, 198, 200, 202, 205, 429
- `\leaders` 430
- `\left` 431
- left commands:
 - `\c_left_brace_str` 110, 6249
- `\leftghost` 932
- `\lefthyphenmin` 432
- `\leftmargin kern` 781
- `\leftskip` 433
- `\leqno` 434
- `\let` 1, 40, 272, 273, 435
- `\latcharcode` 888
- `\letterspacefont` 782
- `\limits` 436
- `\LineBreak` 80, 81, 82, 83, 84, 85, 86, 87, 105, 112, 113, 114, 122, 124
- `\linedir` 938
- `\linepenalty` 437
- `\lineskip` 438
- `\lineskiplimit` 439
- `\linewidth` 8055, 8104
- `\ln` 15310, 15313
- `ln` 193
- `\localbrokenpenalty` 933
- `\localinterlinepenalty` 934
- `\lcalleftbox` 935
- `\lcalrightbox` 936
- `\loccount` 10770, 10909
- `\loctoks` 16737, 16738, 16764
- `\long` 275, 440, 3368, 3372
- `\LongText` 76, 110, 134
- `\looseness` 441
- `\lower` 442
- `\lowercase` 443
- `\lpcode` 783
- lua commands:
 - `\lua_escape:n` 220, 220, 19315
 - `\lua_escape_x:n` . 220, 220, 220, 19315
 - `\lua_now:n` 219, 219, 219, 19315
 - `\lua_now_x:n` 219, 219, 219, 19315
 - `\lua_shipout:n` .. 219, 219, 219, 19315
 - `\lua_shipout_x` 19334
 - `\lua_shipout_x:n` 219, 219, 19315
- `\luaescapestring` 889
- `\luafunction` 890
- luatex commands:
 - `\luatex_alignmark:D` 859, 1183
 - `\luatex_aligntab:D` 860, 1184
 - `\luatex_attribute:D` 861, 1185
 - `\luatex_attributedef:D` 862, 1186
 - `\luatex_begincsname:D` 863
 - `\luatex_bodydir:D` 930, 1218, 1281
 - `\luatex_boxdir:D` 931, 1219
 - `\luatex_catcodetable:D` 864, 1187
 - `\luatex_clearmarks:D` 865, 1188
 - `\luatex_crampeddisplaystyle:D` ... 866, 1189
 - `\luatex_crampedscriptscriptstyle:D` 867, 1190
 - `\luatex_crampedscriptstyle:D` ... 868, 1191
 - `\luatex_crampedtextstyle:D` 869, 1192
 - `\luatex_directlua:D` 870, 1176, 1177, 3093, 3096, 3102, 5825, 10887, 19315
 - `\luatex_dviextension:D` 871
 - `\luatex_dvifedback:D` 872
 - `\luatex_dvivariable:D` 873
 - `\luatex_etoksapp:D` 874

- \luatex_etokspre:D 875
- \luatex_expanded:D ... 876, 1274, 5838
- \luatex_firstvalidlanguage:D .. 877
- \luatex_fontid:D 878, 1193
- \luatex_formatname:D 879, 1194
- \luatex_gleaders:D 884, 1195
- \luatex_hjcode:D 880
- \luatex_hpack:D 881
- \luatex_hyphenationbounds:D ... 882
- \luatex_hyphenationmin:D 883
- \luatex_initcatcodetable:D 885, 1196
- \luatex_lastnamedcs:D 886
- \luatex_latelua:D ... 887, 1197, 19317
- \luatex_leftghost:D 932, 1220
- \luatex_letcharcode:D 888
- \luatex_linedir:D 938
- \luatex_localbrokenpenalty:D ...
..... 933, 1221
- \luatex_localinterlinepenalty:D .
..... 934, 1222
- \luatex_localleftbox:D 935, 1223
- \luatex_localrightbox:D ... 936, 1224
- \luatex_luaescapestring:D
..... 397, 889, 1198, 5836, 19320
- \luatex_luafunction:D 890, 1199
- \luatex_luatexbanner:D 891
- \luatex_luatexdatestamp:D 892
- \luatex_luatexrevision:D 893
- \luatex_luatexversion:D
..... 894, 1237, 1256,
1342, 3756, 4452, 5821, 10889, 19239
- \luatex_mathdir:D 937, 1225
- \luatex_mathdisplayskipmode:D .. 895
- \luatex_matheqnogapstep:D 896
- \luatex_mathnolimitsmode:D 898
- \luatex_mathoption:D 897
- \luatex_mathrulesfam:D 899
- \luatex_mathscriptsmode:D 900
- \luatex_mathstyle:D 901, 1200
- \luatex_mathsurroundskip:D 902
- \luatex_nohrule:D 903
- \luatex_nokerns:D 904, 1201
- \luatex_noligs:D 905, 1202
- \luatex_nospaces:D 906
- \luatex_novrule:D 907
- \luatex_outputbox:D 908, 1203
- \luatex_pagebottomoffset:D 939, 1226
- \luatex_pagedir:D 940, 1227, 1282
- \luatex_pageleftoffset:D .. 909, 1204
- \luatex_pagerightoffset:D . 941, 1229
- \luatex_pagetopoffset:D ... 910, 1205
- \luatex_pardir:D 942, 1231
- \luatex_pdfextension:D
..... 911, 19443, 19444,
19450, 19451, 19456, 19457, 19462,
19463, 19480, 19481, 19489, 19490
- \luatex_pdffeedback:D 912
- \luatex_pdfvariable:D 913
- \luatex_postexhyphenchar:D 914, 1206
- \luatex_posthyphenchar:D .. 915, 1207
- \luatex_preexhyphenchar:D . 916, 1208
- \luatex_prehyphenchar:D ... 917, 1209
- \luatex_rightghost:D 943, 1232
- \luatex_savecatcodetable:D 918, 1210
- \luatex_scantextokens:D ... 919, 1211
- \luatex_setfontid:D 920
- \luatex_shapemode:D 921
- \luatex_suppressifcsnameerror:D .
..... 922, 1212
- \luatex_suppresslongerror:D 923, 1213
- \luatex_suppressmathparerror:D ..
..... 924, 1214
- \luatex_suppressoutererror:D ...
..... 925, 1215
- \luatex_textdir:D 944, 1233
- \luatex_toksapp:D 926
- \luatex_tokspre:D 927
- \luatex_tpack:D 928
- \luatex_vpack:D 929
- \luatexalignmark 1183
- \luatexaligntab 1184
- \luatexattribute 1185
- \luatexattributedef 1186
- \luatexbanner 891
- \luatexbodydir 1218
- \luatexboxdir 1219
- \luatexcatcodetable 1187
- \luatexclearmarks 1188
- \luatexcrampeddisplaystyle 1189
- \luatexcrampedscriptscriptstyle .. 1190
- \luatexcrampedscriptstyle 1191
- \luatexcrampedtextstyle 1192
- \luatexdatestamp 892
- \luatexfontid 1193
- \luatexformatname 1194
- \luatexgladers 1195
- \luatexinitcatcodetable 1196
- \luatexlatelua 1197
- \luatexleftghost 1220
- \luatexlocalbrokenpenalty 1221
- \luatexlocalinterlinepenalty 1222
- \luatexlocalleftbox 1223
- \luatexlocalrightbox 1224
- \luatexluaescapestring 1198
- \luatexluafunction 1199
- \luatexmathdir 1225
- \luatexmathstyle 1200
- \luatexnokerns 1201

<code>\luatexnoligs</code>	1202	<code>\mathoption</code>	897
<code>\luatexoutputbox</code>	1203	<code>\mathord</code>	456
<code>\luatexpagebottomoffset</code>	1226	<code>\mathpunct</code>	457
<code>\luatexpagedir</code>	1227	<code>\mathrel</code>	458
<code>\luatexpageheight</code>	1228	<code>\mathrulesfam</code>	899
<code>\luatexpageleftoffset</code>	1204	<code>\mathscriptsmode</code>	900
<code>\luatexpagerightoffset</code>	1229	<code>\mathstyle</code>	901
<code>\luatexpagetopoffset</code>	1205	<code>\mathsurround</code>	459
<code>\luatexpagewidth</code>	1230	<code>\mathsurroundskip</code>	902
<code>\luatexpardir</code>	1231	<code>max</code>	193
<code>\luatexpostexhyphenchar</code>	1206	<code>max commands:</code>	
<code>\luatexposthyphenchar</code>	1207	<code>\c_max_char_int</code>	72, 4449
<code>\luatexpreehyphenchar</code>	1208	<code>\c_max_register_int</code>	72, 729,
<code>\luatexprehyphenchar</code>	1209	729, 729, 729, 1342, 3649, 9371,	
<code>\luatexrevision</code>	893	16709, 16735, 16772, 16780, 16784	
<code>\luatexrightghost</code>	1232	<code>max internal commands:</code>	
<code>\luatexsavecatcodetable</code>	1210	<code>\c_max_constdef_int</code>	3734
<code>\luatexscantexttokens</code>	1211	<code>\maxdeadcycles</code>	460
<code>\luatexsuppressfontnotfounderror</code> ...	1182, 1217	<code>\maxdepth</code>	461
<code>\luatexsuppressifcsnameerror</code>	1212	<code>\meaning</code>	462
<code>\luatexsuppresslongerror</code>	1213	<code>\medmuskip</code>	463
<code>\luatexsuppressmathparerror</code>	1214	<code>\message</code>	464
<code>\luatexsuppressoutererror</code>	1215	<code>\MessageBreak</code>	122
<code>\luatextextdir</code>	1233	<code>meta commands:</code>	
<code>\luatextracingfonts</code>	1178	<code>.meta:n</code>	165, 10160
<code>\luatexUchar</code>	1216	<code>.meta:nn</code>	165, 10162
<code>\luatexversion</code>	45, 100, 894	<code>\middle</code>	645
M			
<code>\mag</code>	444	<code>min</code>	193
<code>\mark</code>	445	<code>minus commands:</code>	
<code>\marks</code>	644	<code>\c_minus_inf_fp</code>	187,
<code>math commands:</code>		196, 11223, 13908, 13990, 14785, 15547	
<code>\c_math_subscript_token</code>	51, 319, 3183, 3241	<code>\c_minus_zero_fp</code>	187, 11223, 13904, 16195
<code>\c_math_superscript_token</code>	51, 319, 3183, 3236	<code>\mkern</code>	465
<code>\c_math_toggle_token</code>	51, 318, 3183, 3217	<code>mm</code>	197
<code>\mathaccent</code>	446	<code>mode commands:</code>	
<code>\mathbin</code>	447	<code>\mode_if_horizontal:TF</code> ..	39, 39, 2779
<code>\mathchar</code>	448, 3367	<code>\mode_if_horizontal_p:</code> ..	39, 39, 2779
<code>\mathchardef</code>	449	<code>\mode_if_inner:TF</code>	40, 40, 2781
<code>\mathchoice</code>	450	<code>\mode_if_inner_p:</code>	40, 40, 2781
<code>\mathclose</code>	451	<code>\mode_if_math:TF</code>	40, 40, 2783
<code>\mathcode</code>	452	<code>\mode_if_math_p:</code>	40, 2783
<code>\mathdir</code>	937	<code>\mode_if_vertical:TF</code> ...	40, 40, 2777
<code>\mathdisplayskipmode</code>	895	<code>\mode_if_vertical_p:</code> ...	40, 40, 2777
<code>\matheqnogapstep</code>	896	<code>\month</code>	466
<code>\mathinner</code>	453	<code>\moveleft</code>	467
<code>\mathnolimitsmode</code>	898	<code>\moveright</code>	468
<code>\mathop</code>	454	<code>msg commands:</code>	
<code>\mathopen</code>	455	<code>\msg_critical:nn</code>	153, 9051
		<code>\msg_critical:nnn</code>	153, 9051
		<code>\msg_critical:nnnn</code>	153, 9051
		<code>\msg_critical:nnnnn</code>	153, 9051
		<code>\msg_critical:nnnnnn</code> ..	153, 153, 9051

\msg_critical_text:n [152](#), [152](#), [8990](#), [9054](#)
 \msg_error:nn [154](#), [9062](#)
 \msg_error:nnn [154](#), [9062](#)
 \msg_error:nnnn [154](#), [9062](#)
 \msg_error:nnnnn [154](#), [9062](#)
 \msg_error:nnnnnn [154](#), [154](#), [207](#), [9062](#)
 \msg_error_text:n [152](#), [152](#), [8990](#), [9069](#)
 \msg_expandable_error:nn [208](#), [17890](#)
 \msg_expandable_error:nnn [208](#), [17890](#)
 \msg_expandable_error:nnnn [208](#), [17890](#)
 \msg_expandable_error:nnnnn [208](#), [17890](#)
 \msg_fatal:nn [153](#), [9040](#)
 \msg_fatal:nnn [153](#), [9040](#)
 \msg_fatal:nnnn [153](#), [9040](#)
 \msg_fatal:nnnnn [153](#), [153](#), [9040](#)
 \msg_fatal_text:n [152](#), [152](#), [8990](#), [9043](#)
 \msg_gset:nnn [151](#), [8847](#)
 \msg_gset:nnnn [151](#), [8847](#)
 \msg_if_exist:nnTF [152](#), [152](#), [8821](#), [8828](#), [8838](#), [9119](#)
 \msg_if_exist_p:nn [152](#), [152](#), [8821](#)
 \msg_info:nn [154](#), [9091](#)
 \msg_info:nnn [154](#), [9091](#)
 \msg_info:nnnn [154](#), [9091](#)
 \msg_info:nnnnn [154](#), [154](#), [154](#), [9091](#), [9291](#)
 \msg_info_text:n [153](#), [153](#), [8990](#), [9095](#)
 \msg_interrupt:nnn [156](#), [156](#), [8911](#), [9042](#), [9053](#), [9068](#)
 \msg_line_context: [152](#), [152](#), [483](#), [1737](#), [1756](#), [2459](#), [8843](#), [8904](#), [9411](#), [9963](#)
 \msg_line_number: [152](#), [152](#), [8904](#), [9756](#)
 \msg_log:n [157](#), [157](#), [8975](#), [9093](#)
 \msg_log:nn [154](#), [9099](#)
 \msg_log:nnn [154](#), [9099](#)
 \msg_log:nnnn [154](#), [9099](#)
 \msg_log:nnnnn [154](#), [9099](#)
 \msg_log:nnnnnn [154](#), [154](#), [9099](#)
 \msg_new:nnn [151](#), [8847](#), [9242](#)
 \msg_new:nnnn [151](#), [151](#), [482](#), [482](#), [8847](#), [9240](#)
 \msg_none:nn [155](#), [9105](#)
 \msg_none:nnn [155](#), [9105](#)
 \msg_none:nnnn [155](#), [9105](#)
 \msg_none:nnnnn [155](#), [155](#), [9105](#)
 \msg_redirect_class:nn [155](#), [155](#), [9191](#)
 \msg_redirect_module:nnn [156](#), [156](#), [9191](#)
 \msg_redirect_name:nnn [156](#), [156](#), [9182](#)
 \msg_see_documentation_text:n [153](#), [153](#), [8995](#), [9046](#), [9057](#), [9072](#)
 \msg_set:nnn [151](#), [8847](#), [9246](#)
 \msg_set:nnnn [151](#), [151](#), [8847](#), [9244](#)
 \msg_term:n [157](#), [157](#), [8975](#), [9085](#)
 \msg_warning:nn [154](#), [9083](#)
 \msg_warning:nnn [154](#), [9083](#)
 \msg_warning:nnnn [154](#), [9083](#)
 \msg_warning:nnnnn [154](#), [9083](#)
 \msg_warning:nnnnnn [154](#), [154](#), [9083](#), [9290](#)
 \msg_warning_text:n [152](#), [152](#), [8990](#), [9087](#)
 msg internal commands:
 _msg_class_chk_exist:nTF [9107](#), [9121](#), [9187](#), [9197](#), [9202](#)
 \l_msg_class_loop_seq [492](#), [9116](#), [9206](#), [9214](#), [9224](#), [9225](#), [9228](#), [9230](#)
 _msg_class_new:nn [489](#), [493](#), [9001](#), [9040](#), [9051](#), [9062](#), [9083](#), [9091](#), [9099](#), [9105](#)
 \l_msg_class_tl [490](#), [492](#), [9112](#), [9128](#), [9140](#), [9161](#), [9165](#), [9168](#), [9176](#), [9215](#), [9217](#), [9219](#), [9233](#)
 \c_msg_coding_error_text_tl [160](#), [8762](#), [8773](#), [8872](#), [9296](#), [9304](#), [9330](#), [9348](#), [9357](#), [9364](#), [9378](#), [9387](#), [9396](#), [9403](#), [9425](#), [9432](#), [9439](#), [9447](#), [10495](#), [10516](#), [10522](#), [10529](#)
 \c_msg_continue_text_tl [8872](#), [8916](#)
 \c_msg_critical_text_tl [8872](#), [9059](#)
 \l_msg_current_class_tl [491](#), [9112](#), [9123](#), [9160](#), [9165](#), [9168](#), [9176](#), [9205](#), [9219](#)
 _msg_error:Nnnnnn [9062](#)
 _msg_error_code:nnnnnn [9289](#)
 _msg_expandable_error:n [159](#), [159](#), [499](#), [9496](#), [9515](#)
 _msg_expandable_error:w [498](#), [9496](#)
 _msg_expandable_error_module:nn [17890](#)
 _msg_fatal_code:nnnnnn [9285](#)
 \c_msg_fatal_text_tl [8872](#), [9048](#)
 \c_msg_help_text_tl [8872](#), [8920](#)
 \l_msg_hierarchy_seq [490](#), [490](#), [9115](#), [9143](#), [9153](#), [9158](#)
 \l_msg_internal_tl [501](#), [8818](#), [9589](#), [9590](#), [9593](#), [9602](#)
 _msg_interrupt_more_text:n [484](#), [8923](#)
 _msg_interrupt_text:n [8926](#), [8937](#)


```

\__msg_interrupt_wrap:nn .....
    ..... 8915, 8919, 8923
\__msg_kernel_class_new:nN .....
    ... 494, 9247, 9285, 9289, 9290, 9291
\__msg_kernel_class_new_aux:nN 9247
\__msg_kernel_error:nn ..... 158,
    1720, 8266, 9285, 9734, 9749, 16933
\__msg_kernel_error:nnn 158, 1456,
    1498, 1539, 1544, 1720, 1768, 1778,
    1929, 1936, 2348, 2890, 5176, 7871,
    8004, 9110, 9285, 9577, 9826, 9888,
    9904, 10046, 10063, 10615, 10688,
    10786, 11213, 11535, 17056, 19338
\__msg_kernel_error:nnnn .....
    ..... 158, 1447, 1476, 1558,
    1720, 1743, 1753, 1899, 2409, 8158,
    8830, 8840, 9135, 9285, 9811, 9865,
    9923, 9936, 10054, 10393, 11531, 16926
\__msg_kernel_error:nnnnn .....
    ..... 158, 9285, 17072
\__msg_kernel_error:nnnnnn .....
    ..... 158, 158, 2424, 9285
\__msg_kernel_expandable_-
    error:nn . 159, 2765, 3055, 3059,
    3061, 3069, 3075, 6406, 7475, 9513,
    11031, 12031, 12882, 16485, 17877
\__msg_kernel_expandable_-
    error:nnn .... 159, 2096, 3831,
    4011, 5432, 6868, 7377, 9513, 12036,
    12050, 12052, 12150, 12189, 12195,
    12531, 12536, 12547, 12554, 12617,
    12627, 12829, 12849, 13401, 19329
\__msg_kernel_expandable_-
    error:nnnn ..... 159,
    9513, 13021, 13062, 13517, 17842
\__msg_kernel_expandable_-
    error:nnnnn ..... 159, 9513,
    11646, 13189, 15975, 16546, 16560
\__msg_kernel_expandable_-
    error:nnnnnn ..... 159, 159, 9513
\__msg_kernel_fatal:nn .....
    ..... 158, 9285, 10814, 10931
\__msg_kernel_fatal:nnn ... 158, 9285
\__msg_kernel_fatal:nnnn .. 158, 9285
\__msg_kernel_fatal:nnnnn . 158, 9285
\__msg_kernel_fatal:nnnnnn .....
    ..... 158, 158, 9285
\__msg_kernel_info:nn .... 158, 9290
\__msg_kernel_info:nnn .... 158, 9290
\__msg_kernel_info:nnnn ... 158, 9290
\__msg_kernel_info:nnnnn .. 158, 9290
\__msg_kernel_info:nnnnnn .....
    ..... 158, 158, 9290

\__msg_kernel_new:nnn .....
    ..... 157, 8777, 9239,
    9335, 9337, 9339, 9341, 9343, 9452,
    9454, 9456, 9458, 9460, 9462, 9464,
    9471, 9478, 9485, 10545, 11194,
    11674, 11676, 11678, 11680, 11682,
    11684, 13211, 13213, 13215, 13217,
    13219, 13221, 13223, 13225, 13227,
    13232, 13436, 13438, 16481, 17883
\__msg_kernel_new:nnnn .....
    ..... 157, 157, 8759, 8767,
    8770, 9239, 9293, 9301, 9309, 9316,
    9327, 9345, 9354, 9361, 9368, 9375,
    9384, 9393, 9400, 9410, 9422, 9429,
    9436, 9444, 9755, 10483, 10486,
    10492, 10501, 10507, 10513, 10519,
    10526, 10533, 10539, 11169, 11175,
    11182, 11189, 11496, 11648, 11663,
    17061, 17078, 17085, 17094, 19343
\__msg_kernel_set:nnn .... 157, 9239
\__msg_kernel_set:nnnn 157, 157, 9239
\__msg_kernel_warning:nn .. 158, 9290
\__msg_kernel_warning:nnn . 158, 9290
\__msg_kernel_warning:nnnn 158, 9290
\__msg_kernel_warning:nnnnn 158, 9290
\__msg_kernel_warning:nnnnnn ...
    ..... 158, 158, 9222, 9290
\__msg_log_next: ..... 159, 159,
    159, 159, 742, 9542, 17108, 17110,
    17112, 17504, 17506, 17752, 17814,
    17817, 17820, 17822, 17826, 17829,
    17888, 17949, 17951, 17980, 18046,
    18071, 18075, 18079, 19163, 19166
\g__msg_log_next_bool .....
    500, 501, 9542, 9564, 9579, 9591, 9594
\__msg_log_wrap:n ..... 500
\c__msg_more_text_prefix_tl ....
    ..... 8819, 8858, 8867, 9065
\c__msg_no_info_text_tl .. 8872, 8915
\__msg_no_more_text:nnnn ..... 9062
\c__msg_on_line_text_tl .. 8872, 8907
\__msg_redirect:nnn ..... 9191
\__msg_redirect_loop_chk:nnn ...
    ..... 9191, 9233
\__msg_redirect_loop_list:n .. 9191
\l__msg_redirect_prop .....
    ..... 9114, 9140, 9185, 9188
\c__msg_return_text_tl .....
    ..... 8872, 9299, 9307, 9314
\__msg_show_item:n .... 160, 160,
    160, 500, 501, 6923, 7456, 7463, 9608
\__msg_show_item:nn .....
    ..... 160, 160, 160, 451, 7739, 9608

```


_msg_show_item_unbraced:nn	...	160, 160, 535, 8754, 9608, 10478, 10850
_msg_show_pre:nnnnnn	159, 159, 7460, 8745, 9545, 9571, 10461, 10467, 10847
_msg_show_pre_aux:n	500, 9545
_msg_show_variable:NNNnn	159, 160, 160, 160, 278, 426, 442, 451, 500, 727, 2020, 2585, 4413, 5755, 6921, 7454, 7737, 9565, 16677
_msg_show_wrap:n	159, 160, 160, 160, 160, 394, 500, 500, 501, 501, 501, 501, 501, 501, 525, 2026, 2912, 2985, 2994, 3003, 3012, 5760, 7462, 8750, 9574, 9583, 9584, 10469, 10476, 10849
_msg_show_wrap:Nn	159, 160, 160, 501, 2589, 4419, 4714, 4807, 4871, 9582, 16681
_msg_show_wrap_aux:n	9584
_msg_show_wrap_aux:w	9584
\c_msg_text_prefix_tl	8819, 8823, 8856, 8865, 9045, 9056, 9071, 9088, 9096, 9102, 9518, 9549, 17895
_msg_tmp:w	9497, 9510
\c_msg_trouble_text_tl	8872
_msg_use:nnnnnnn	9011, 9117
_msg_use_code:	490, 490, 9117
_msg_use_hierarchy:nwN	9117
_msg_use_redirect_module:n	...	490, 9117
_msg_use_redirect_name:n	...	9117
\mskip	469
\muexpr	646
multichoice commands:		
.multichoice:	165, 10164
multichoices commands:		
.multichoices:nn	165, 10164
\multiply	470
\muskip	471, 3375
muskip commands:		
\c_max_muskip	87, 4872
\muskip_add:Nn	86, 86, 4854
\muskip_const:Nn	86, 86, 4822, 4872, 4873
\muskip_eval:n	...	87, 87, 87, 4864, 4871
\muskip_gadd:Nn	86, 4854
\muskip_gset:Nn	86, 4825, 4843
\muskip_gset_eq:NN	86, 4848
\muskip_gsub:Nn	86, 4854
\muskip_gzero:N	86, 4828, 4836
\muskip_gzero_new:N	86, 4833
\muskip_if_exist:N	...	86, 86, 4834, 4836, 4839
\muskip_if_exist_p:N	...	86, 86, 4839
\muskip_log:N	211, 211, 18076
\muskip_log:n	211, 211, 18076
\muskip_new:N	...	85, 85, 86, 4814, 4824, 4834, 4836, 4874, 4875, 4876, 4877
\muskip_set:Nn	86, 86, 4843
\muskip_set_eq:NN	86, 86, 4848
\muskip_show:N	87, 87, 4868
\muskip_show:n	...	87, 87, 770, 4870, 18079
\muskip_sub:Nn	86, 86, 4854
\muskip_use:N	...	87, 87, 87, 87, 4865, 4866
\muskip_zero:N	86, 4828, 4834
muskip_zero:N	86
\muskip_zero_new:N	86, 86, 4833
\g_tmpa_muskip	88, 4874
\l_tmpa_muskip	87, 4874
\g_tmpb_muskip	88, 4874
\l_tmpb_muskip	87, 4874
\c_zero_muskip	87, 4829, 4872
\muskipdef	472
\mutoglu	647
N		
nan	196
nc	197
nd	197
\newbox	334
\newcatcodetable	52
\newcount	334
\newdimen	334
\newlinechar	103, 473
\next	58, 58, 59, 74, 106, 131, 140, 144, 147, 155
\NG	18964
\ng	18964
\noalign	474
\noautospace	1140
\noautoxspacing	1141
\noboundary	475
\noexpand	118, 122, 133, 136, 476
\nohrule	903
\noindent	477
\nokerns	904
\noligs	905
\nolimits	478
\nonscript	479
\nonstopmode	480
\normaldeviate	958
\normalend	1263, 1264, 10766, 10905
\normaleveryjob	1265
\normalexpanded	1274
\normalhoffset	1277
\normalinput	1266
\normalitaliccorrection	1276, 1278

- | | | |
|--|---------------|------------------------------------|
| <code>\normallanguage</code> | 1267 | 14324, 14570, 14786, 14811, 14817, |
| <code>\normalleft</code> | 1284, 1285 | 14818, 14819, 14820, 14821, 14969, |
| <code>\normalmathop</code> | 1268 | 15004, 15006, 15014, 15104, 15108, |
| <code>\normalmiddle</code> | 1286 | 15109, 15110, 15111, 15112, 15113, |
| <code>\normalmonth</code> | 1269 | 15114, 15115, 15116, 15117, 15124, |
| <code>\normalouter</code> | 1270 | 15125, 15126, 15127, 15128, 15129, |
| <code>\normalover</code> | 1271 | 15130, 15131, 15132, 15133, 15140, |
| <code>\normalright</code> | 1287 | 15141, 15142, 15143, 15144, 15145, |
| <code>\normalshowtokens</code> | 1280 | 15146, 15147, 15148, 15149, 15156, |
| <code>\normalunexpanded</code> | 1273 | 15157, 15158, 15159, 15160, 15161, |
| <code>\normalvcenter</code> | 1272 | 15162, 15163, 15164, 15165, 15172, |
| <code>\normalvoffset</code> | 1279 | 15173, 15174, 15175, 15176, 15177, |
| <code>\nospaces</code> | 906 | 15178, 15179, 15180, 15181, 15188, |
| notexpanded commands: | | |
| <code>\notexpanded: <token></code> | 59 | 15189, 15190, 15191, 15192, 15193, |
| <code>\novrule</code> | 907 | 15194, 15195, 15196, 15197, 15215, |
| <code>\nulldelimiterspace</code> | 481 | 15263, 15266, 15275, 15386, 15409, |
| <code>\nullfont</code> | 482 | 15456, 15461, 15471, 15476, 15486, |
| <code>\number</code> | 55, 483 | 15491, 15501, 15506, 15516, 15521, |
| <code>\numexpr</code> | 167, 181, 648 | 15531, 15536, 15973, 15993, 15994, |
| | | 16038, 16123, 16126, 16138, 16144, |
| | | 16191, 16193, 16194, 16204, 16210, |
| | | 16253, 16254, 16265, 16302, 16303, |
| | | 16313, 16370, 16371, 16558, 16587 |
- O**
- | | | |
|-------------------------------------|-------------------------------------|--|
| <code>\O</code> | 18965 | |
| <code>\o</code> | 18965 | |
| <code>\OE</code> | 18966 | |
| <code>\oe</code> | 18966 | |
| <code>\omit</code> | 484 | |
| one commands: | | |
| <code>\c_minus_one</code> | 72, 4436, 10882 | |
| <code>\c_one_degree_fp</code> | 188, 196, 12641, 16685 | |
| <code>\openin</code> | 485 | |
| <code>\openout</code> | 486 | |
| <code>\or</code> | 487 | |
| or commands: | | |
| <code>\or:</code> .. | 73, 73, 73, 401, 401, 401, 401, | |
| | 403, 403, 682, 1292, 1860, 1861, | |
| | 1862, 1863, 1864, 1865, 1866, 1867, | |
| | 1868, 3115, 3119, 3122, 3126, 3130, | |
| | 3132, 3134, 3136, 3137, 3139, 3141, | |
| | 3143, 3145, 3649, 4198, 4199, 4200, | |
| | 4201, 4202, 4203, 4204, 4205, 4206, | |
| | 4207, 4208, 4209, 4210, 4211, 4212, | |
| | 4213, 4214, 4215, 4216, 4217, 4218, | |
| | 4219, 4220, 4221, 4222, 4231, 4232, | |
| | 4233, 4234, 4235, 4236, 4237, 4238, | |
| | 4239, 4240, 4241, 4242, 4243, 4244, | |
| | 4245, 4246, 4247, 4248, 4249, 4250, | |
| | 4251, 4252, 4253, 4254, 4255, 6018, | |
| | 6093, 11262, 11263, 11264, 11411, | |
| | 11437, 11438, 11809, 11810, 11835, | |
| | 13085, 13086, 13087, 13123, 13600, | |
| | 13601, 13602, 13730, 13814, 13900, | |
| | 13901, 13902, 13903, 13904, 13905, | |
| | 13906, 13907, 13908, 13985, 13988, | |
- P**
- | | |
|---------------------------------------|--------------------------------------|
| <code>\PackageError</code> | 125, 133 |
| <code>\pagebottomoffset</code> | 939 |
| <code>\pagedepth</code> | 495 |
| <code>\pagedir</code> | 940 |
| <code>\pagediscards</code> | 649 |
| <code>\pagefillllstretch</code> | 496 |
| <code>\pagefillstretch</code> | 497 |
| <code>\pagefilstretch</code> | 498 |
| <code>\pagegoal</code> | 499 |
| <code>\pageheight</code> | 960 |
| <code>\pageleftoffset</code> | 909 |
| <code>\pagerightoffset</code> | 941 |
| <code>\pageshrink</code> | 500 |
| <code>\pagestretch</code> | 501 |
| <code>\pagetopoffset</code> | 910 |
| <code>\pagetotal</code> | 502 |
| <code>\pagewidth</code> | 961 |
| <code>\par</code> | 10, 10, 11, 11, 11, 12, |
| | 12, 12, 13, 13, 13, 14, 14, 14, 175, |

175, 273, 273, 457, 503, 7907, 7908, 7910, 7912, 7914, 7919, 7927, 7941		\pdflastobj	703
		\pdflastxform	704
\paddir	942	\pdflastximage	705
\parfillskip	504	\pdflastximagecolordepth	706
\parindent	505	\pdflastximagepages	707
\parshape	506	\pdflastxpos	756
\parshapedimen	650	\pdflastypos	757
\parshapeindent	651	\pdflinkmargin	708
\parshapelength	652	\pdfliteral	709
\parskip	507	\pdfmapfile	758
\patterns	508	\pdfmapline	759
\pausing	509	\pdfminorversion	710
pc	197	\pdfnames	711
\pdfadjustspacing	745	\pdfnoligatures	760
\pdfannot	675	\pdfnormaldeviate	761
\pdfcatalog	676	\pdfobj	712
\pdfcolorstack	678	\pdfobjcompresslevel	713
\pdfcolorstackinit	679	\pdfoutline	714
\pdfcompresslevel	677	\pdfoutput	715
\pdfcopyfont	746	\pdfpageattr	716
\pdfcreationdate	680	\pdfpagebox	717
\pdfdecimaldigits	681	\pdfpageheight	762
\pdfdest	682	\pdfpageref	718
\pdfdestmargin	683	\pdfpageresources	719
\pdfdraftmode	747	\pdfpagesattr	720
\pdfeachlinedepth	748	\pdfpagewidth	763
\pdfeachlineheight	749	\pdfpkmode	764
\pdfendlink	684	\pdfpkresolution	765
\pdfendthread	685	\pdfprimitive	766
\pdfextension	911	\pdfprotrudechars	767
\pdffeedback	912	\pdfpxdimen	768
\pdffirstlineheight	750	\pdfrandomseed	769
\pdffontattr	686	\pdfrefobj	721
\pdffontexpand	751	\pdfrefxform	722
\pdffontname	687	\pdfrefximage	723
\pdffontobjnum	688	\pdfrestore	724
\pdffontsize	752	\pdfretval	725
\pdfgamma	689	\pdfsave	726
\pdfgentounicode	692	\pdfsavepos	770
\pdfglyphtounicode	693	\pdfsetmatrix	727
\pdfhorigin	694	\pdfsetrandomseed	772
\pdfignoreddimen	753	\pdfshellescape	773
\pdfimageapplygamma	690	\pdfstartlink	728
\pdfimagegamma	691	\pdfstartthread	729
\pdfimagehicolor	695	\pdfstrcmp	40, 771
\pdfimageresolution	696	pdfstrcmp	396
\pdfincludechars	697	\pdfsuppressptexinfo	730
\pdfinclusioncopyfonts	698	pdftex commands:	
\pdfinclusionerrorlevel	699	\pdftex_adjustspacing:D	745, 945
\pdfinfo	700	\pdftex_copyfont:D	746, 946
\pdfinsertht	754	\pdftex_draftmode:D	747, 947
\pdflastannot	701	\pdftex_eachlinedepth:D	748
\pdflastlinedepth	755	\pdftex_eachlineheight:D	749
\pdflastlink	702	\pdftex_efcode:D	779

<code>\pdfTeX_firstlineheight:D</code>	750	<code>\pdfTeX_pdflastximage:D</code>	705, 954
<code>\pdfTeX_fontexpand:D</code>	751, 948	<code>\pdfTeX_pdflastximagecolordepth:D</code>	
<code>\pdfTeX_fontsize:D</code>	752		706
<code>\pdfTeX_ifabsdim:D</code>	742, 949	<code>\pdfTeX_pdflastximagepages:D</code>	707, 955
<code>\pdfTeX_ifabsnum:D</code>	743, 950	<code>\pdfTeX_pdflinkmargin:D</code>	708
<code>\pdfTeX_ifincsname:D</code>	780	<code>\pdfTeX_pdfliteral:D</code>	709, 19445
<code>\pdfTeX_ifprimitive:D</code>	744, 856	<code>\pdfTeX_pdfminorversion:D</code>	710
<code>\pdfTeX_ignoreddimen:D</code>	753	<code>\pdfTeX_pdfnames:D</code>	711
<code>\pdfTeX_ignoreligaturesinfont:D</code>	951	<code>\pdfTeX_pdfobj:D</code>	712
<code>\pdfTeX_insertht:D</code>	754, 952	<code>\pdfTeX_pdfobjcompresslevel:D</code>	713
<code>\pdfTeX_lastlinedepth:D</code>	755	<code>\pdfTeX_pdfoutline:D</code>	714
<code>\pdfTeX_lastxpos:D</code>	756, 956	<code>\pdfTeX_pdfoutput:D</code>	
<code>\pdfTeX_lastypos:D</code>	757, 957		715, 959, 19287, 19288
<code>\pdfTeX_leftmarginkern:D</code>	781	<code>\pdfTeX_pdfpageattr:D</code>	716
<code>\pdfTeX_letterspacefont:D</code>	782	<code>\pdfTeX_pdfpagebox:D</code>	717
<code>\pdfTeX_lpcode:D</code>	783	<code>\pdfTeX_pdfpageref:D</code>	718
<code>\pdfTeX_mapfile:D</code>	758, 1239	<code>\pdfTeX_pdfpageresources:D</code>	719
<code>\pdfTeX_mapline:D</code>	759, 1240	<code>\pdfTeX_pdfpagesattr:D</code>	720
<code>\pdfTeX_noligatures:D</code>	760	<code>\pdfTeX_pdfrefobj:D</code>	721
<code>\pdfTeX_normaldeviate:D</code>	761, 958	<code>\pdfTeX_pdfrefxform:D</code>	722, 965
<code>\pdfTeX_pageheight:D</code>	762, 960, 1228	<code>\pdfTeX_pdfrefximage:D</code>	723, 966
<code>\pdfTeX_pagewidth:D</code>	763, 1230	<code>\pdfTeX_pdfrestore:D</code>	724, 19458
<code>\pdfTeX_pagewith:D</code>	961	<code>\pdfTeX_pdfretval:D</code>	725
<code>\pdfTeX_pdfannot:D</code>	675	<code>\pdfTeX_pdfsave:D</code>	726, 19452
<code>\pdfTeX_pdfcatalog:D</code>	676	<code>\pdfTeX_pdfsetmatrix:D</code>	727, 19464
<code>\pdfTeX_pdfcolorstack:D</code>		<code>\pdfTeX_pdfstartlink:D</code>	728
	678, 19482, 19491	<code>\pdfTeX_pdfstartthread:D</code>	729
<code>\pdfTeX_pdfcolorstackinit:D</code>	679	<code>\pdfTeX_pdfsuppressptexinfo:D</code>	730
<code>\pdfTeX_pdfcompresslevel:D</code>	677	<code>\pdfTeX_pdftexbanner:D</code>	776, 1257
<code>\pdfTeX_pdfcreationdate:D</code>	680	<code>\pdfTeX_pdftexrevision:D</code>	777, 1258
<code>\pdfTeX_pdfdecimaldigits:D</code>	681	<code>\pdfTeX_pdftexversion:D</code>	
<code>\pdfTeX_pdfdest:D</code>	682		253, 778, 1236, 1259, 19247
<code>\pdfTeX_pdfdestmargin:D</code>	683	<code>\pdfTeX_pdfthread:D</code>	731
<code>\pdfTeX_pdfendlink:D</code>	684	<code>\pdfTeX_pdfthreadmargin:D</code>	732
<code>\pdfTeX_pdfendthread:D</code>	685	<code>\pdfTeX_pdftrailer:D</code>	733
<code>\pdfTeX_pdffontattr:D</code>	686	<code>\pdfTeX_pdfuniquefilename:D</code>	734
<code>\pdfTeX_pdffontname:D</code>	687	<code>\pdfTeX_pdfvorigin:D</code>	735
<code>\pdfTeX_pdffontobjnum:D</code>	688	<code>\pdfTeX_pdfxform:D</code>	736, 968
<code>\pdfTeX_pdfgamma:D</code>	689	<code>\pdfTeX_pdfxformattr:D</code>	737
<code>\pdfTeX_pdfgentounicode:D</code>	692	<code>\pdfTeX_pdfxformname:D</code>	738
<code>\pdfTeX_pdfglyphtounicode:D</code>	693	<code>\pdfTeX_pdfxformresources:D</code>	739
<code>\pdfTeX_pdfhorigin:D</code>	694	<code>\pdfTeX_pdfximage:D</code>	740, 969
<code>\pdfTeX_pdfimageapplygamma:D</code>	690	<code>\pdfTeX_pdfximagebbox:D</code>	741
<code>\pdfTeX_pdfimagegamma:D</code>	691	<code>\pdfTeX_pkmode:D</code>	764
<code>\pdfTeX_pdfimagehicolor:D</code>	695	<code>\pdfTeX_pkreolution:D</code>	765
<code>\pdfTeX_pdfimageresolution:D</code>	696	<code>\pdfTeX_primitive:D</code>	766, 857
<code>\pdfTeX_pdfincludechars:D</code>	697	<code>\pdfTeX_protrudechars:D</code>	767, 962
<code>\pdfTeX_pdfinclusioncopyfonts:D</code>	698	<code>\pdfTeX_pxdimen:D</code>	768, 963
<code>\pdfTeX_pdfinclusionerrorlevel:D</code>	699	<code>\pdfTeX_quitvmode:D</code>	784
<code>\pdfTeX_pdfinfo:D</code>	700	<code>\pdfTeX_randomseed:D</code>	769, 964
<code>\pdfTeX_pdflastannot:D</code>	701	<code>\pdfTeX_rightmarginkern:D</code>	785
<code>\pdfTeX_pdflastlink:D</code>	702	<code>\pdfTeX_rpcode:D</code>	786
<code>\pdfTeX_pdflastobj:D</code>	703	<code>\pdfTeX_savepos:D</code>	770, 967
<code>\pdfTeX_pdflastxform:D</code>	704, 953	<code>\pdfTeX_setrandomseed:D</code>	772, 970

- \pdfutex_shellescape:D 773, 858
- \pdfutex_strcmp:D 771, 5820
- \pdfutex_synctex:D 787
- \pdfutex_tagcode:D 788
- \pdfutex_tracingfonts:D
 - 774, 971, 1172, 1174, 1178
- \pdfutex_uniformdeviate:D
 - .. 721, 721, 722, 775, 972, 16479,
 - 16490, 16493, 17830, 17860, 17881
- \pdfutexbanner 776
- \pdfutexrevision 777
- \pdfutexversion 95, 778
- \pdfthread 731
- \pdfthreadmargin 732
- \pdftracingfonts 774, 1173, 1174
- \pdftrailer 733
- \pdfuniformdeviate 775
- \pdfuniqueresname 734
- \pdfvariable 913
- \pdfvorigin 735
- \pdfxform 736
- \pdfxformattr 737
- \pdfxformname 738
- \pdfxformresources 739
- \pdfximage 740
- \pdfximagebbox 741
- peek commands:
 - \peek_after:Nw 40,
 - 55, 55, 56, 3449, 3474, 3492, 3542
 - \peek_catcode:Nw 56, 56, 3566
 - \peek_catcode_ignore_spaces:Nw .
 - 56, 56, 3566
 - \peek_catcode_remove:Nw 56, 56, 3566
 - \peek_catcode_remove_ignore_-
 - spaces:Nw 56, 56, 3566
 - \peek_charcode:Nw 56, 56, 3582
 - \peek_charcode_ignore_spaces:Nw
 - 57, 57, 3582
 - \peek_charcode_remove:Nw 57, 57, 3582
 - \peek_charcode_remove_ignore_-
 - spaces:Nw 57, 57, 3582
 - \peek_gafter:Nw 56, 56, 56, 3449
 - \peek_meaning:Nw 57, 57, 3598
 - \peek_meaning_ignore_spaces:Nw .
 - 57, 57, 3598
 - \peek_meaning_remove:Nw 57, 57, 3598
 - \peek_meaning_remove_ignore_-
 - spaces:Nw 58, 58, 3598
 - \peek_N_type:TF
 - 216, 216, 19175, 19209, 19211
- peek internal commands:
 - __peek_def:nnnn 3549,
 - 3566, 3570, 3574, 3578, 3582, 3586,
 - 3590, 3594, 3598, 3602, 3606, 3610
 - __peek_def:nnnn 3549
 - __peek_execute_branches:
 - 329, 3546, 3561
 - __peek_execute_branches_-
 - catcode: 3506, 3569, 3571, 3577, 3579
 - __peek_execute_branches_-
 - catcode_aux: 3506
 - __peek_execute_branches_-
 - catcode_auxii:N 3506
 - __peek_execute_branches_-
 - catcode_auxiii: 3506
 - __peek_execute_branches_-
 - charcode:
 - 3506, 3585, 3587, 3593, 3595
 - __peek_execute_branches_-
 - meaning: 3498, 3601, 3603, 3609, 3611
 - __peek_execute_branches_N_type:
 - 19175
 - __peek_false:w 795, 795, 3445, 3468,
 - 3486, 3503, 3526, 3536, 19192, 19205
 - __peek_get_prefix_arg_replacement:wN
 - 3615
 - __peek_ignore_spaces_execute_-
 - branches: 3539,
 - 3573, 3581, 3589, 3597, 3605, 3613
 - __peek_N_type:w 19175
 - __peek_N_type_aux:nnw 19175
 - \l_peek_search_tl
 - 326, 328, 3444, 3462, 3483, 3523, 3533
 - \l_peek_search_token
 - 326, 3443, 3461, 3482, 3500
 - __peek_tmp:w 3445, 3457, 19176, 19198
 - __peek_token_generic:NwTF . 795,
 - 3459, 3476, 3478, 19208, 19210, 19212
 - __peek_token_remove_generic:NwTF
 - 3480, 3494, 3496
 - __peek_true:w
 - 795, 795, 3445, 3463, 3484,
 - 3501, 3524, 3534, 19190, 19204, 19205
 - __peek_true_aux:w .. 3445, 3456, 3485
 - __peek_true_remove:w ... 3453, 3484
 - \penalty 510
 - pi 196
 - \pm 13383, 13384
 - \postbreakpenalty 1142
 - \postdisplaypenalty 511
 - \postexhyphenchar 914
 - \posthyphenchar 915
 - \prebreakpenalty 1143
 - \predisplaydirection 653
 - \predisplaypenalty 512
 - \predisplaysize 513
 - \preexhyphenchar 916
 - \prehyphenchar 917

- \pretolerance 514
- \prevdepth 515
- \prevgraf 516
- prg commands:
 - \prg_break_point:Nn 45
 - \prg_do_nothing:
 - . 9, 9, 41, 372, 414, 414, 441, 564,
 - 719, 1529, 1530, 2029, 2039, 2386,
 - 2391, 5062, 5090, 5140, 5155, 6479,
 - 6486, 6726, 6728, 7091, 7431, 7435,
 - 7442, 9700, 11486, 11541, 11575,
 - 11601, 11609, 13094, 16440, 16831,
 - 16838, 16914, 17047, 17048, 18146
 - \prg_new_conditional:Nnn
 - 34, 34, 1428,
 - 2498, 2846, 2854, 2866, 2872, 10853
 - \prg_new_conditional:Npnn ... 34,
 - 34, 34, 319, 329, 1415, 2001, 2498,
 - 2571, 2601, 2777, 2779, 2781, 2783,
 - 3205, 3210, 3215, 3220, 3227, 3233,
 - 3239, 3244, 3249, 3254, 3259, 3264,
 - 3269, 3274, 3281, 3298, 3303, 3338,
 - 3382, 3835, 3888, 3925, 3933, 4553,
 - 4558, 4770, 4782, 5241, 5251, 5263,
 - 5285, 5287, 5335, 5617, 5636, 5655,
 - 5690, 5707, 5718, 5850, 5865, 5870,
 - 6603, 7217, 7646, 7655, 7811, 7813,
 - 7823, 7985, 8821, 10442, 10449,
 - 11407, 11511, 12524, 13241, 13254,
 - 17918, 17927, 17933, 17942, 18081
 - \prg_new_eq_conditional:Nnn
 - 35, 35,
 - 1520, 2498, 2597, 2599, 3784, 3786,
 - 4493, 4495, 4745, 4747, 4839, 4841,
 - 4925, 4926, 5816, 5817, 5818, 5819,
 - 6508, 6510, 6913, 6914, 6915, 6916,
 - 6917, 6918, 7000, 7002, 7213, 7215,
 - 7642, 7644, 7777, 7779, 13239, 13240
 - \prg_new_protected_conditional:Nnn
 - 34, 34, 1428, 2498
 - \prg_new_protected_conditional:Npnn
 - 34, 34, 1415, 2498, 5299,
 - 5320, 6615, 6725, 6727, 6735, 6737,
 - 6739, 6741, 7100, 7112, 7114, 7231,
 - 7235, 7578, 7588, 7685, 10670, 10789
 - \prg_replicate:nn 39, 39, 304,
 - 304, 304, 545, 2730, 9459, 11161,
 - 14512, 15362, 15421, 15428, 15581,
 - 15785, 15786, 15788, 15790, 15803,
 - 15840, 16309, 16336, 16344, 20291
 - \prg_return_false: 35,
 - 36, 36, 36, 110, 337, 337, 390, 418,
 - 435, 1411, 1621, 1626, 1639, 1644,
 - 1652, 1669, 2004, 2498, 2576, 2606,
 - 2778, 2780, 2782, 2784, 2851, 2859,
 - 3208, 3213, 3218, 3223, 3230, 3237,
 - 3242, 3247, 3252, 3257, 3262, 3267,
 - 3272, 3277, 3295, 3301, 3306, 3311,
 - 3344, 3347, 3386, 3411, 3428, 3437,
 - 3833, 3865, 3870, 3893, 3930, 3936,
 - 4556, 4577, 4592, 4593, 4777, 4785,
 - 5256, 5269, 5282, 5292, 5309, 5325,
 - 5341, 5629, 5652, 5668, 5676, 5686,
 - 5699, 5713, 5727, 5847, 5855, 5868,
 - 5873, 6608, 6629, 6652, 7103, 7119,
 - 7220, 7245, 7586, 7596, 7649, 7671,
 - 7692, 7812, 7814, 7824, 7991, 7993,
 - 8824, 10447, 10455, 10674, 10798,
 - 10863, 11412, 11422, 11430, 11516,
 - 12538, 12550, 13249, 13262, 17924,
 - 17930, 17931, 17937, 17946, 18087
 - \prg_return_true/false: 397
 - \prg_return_true: 35,
 - 36, 36, 36, 110, 391, 391, 418, 446,
 - 1411, 1624, 1641, 1649, 1654, 1667,
 - 1672, 2004, 2498, 2574, 2604, 2778,
 - 2780, 2782, 2784, 2849, 2857, 3208,
 - 3213, 3218, 3223, 3230, 3237, 3242,
 - 3247, 3252, 3257, 3262, 3267, 3272,
 - 3277, 3293, 3301, 3309, 3409, 3435,
 - 3865, 3891, 3928, 3938, 4556, 4593,
 - 4775, 4786, 5254, 5267, 5280, 5290,
 - 5306, 5325, 5339, 5627, 5650, 5666,
 - 5684, 5697, 5715, 5726, 5845, 5855,
 - 5868, 5873, 6606, 6633, 6655, 7106,
 - 7122, 7221, 7245, 7584, 7594, 7649,
 - 7673, 7690, 7812, 7814, 7824, 7990,
 - 8824, 10446, 10454, 10675, 10801,
 - 10858, 10861, 10867, 11410, 11428,
 - 11514, 12533, 12556, 13251, 13260,
 - 17922, 17930, 17939, 17945, 17946
 - \prg_set_conditional:Nnn
 - 34, 1428, 2498
 - \prg_set_conditional:Npnn 34, 35,
 - 36, 1415, 1618, 1630, 1646, 1658, 2498
 - \prg_set_eq_conditional:Nnn
 - 35, 1520, 2498
 - \prg_set_protected_conditional:Nnn
 - 34, 1428, 2498
 - \prg_set_protected_conditional:Npnn
 - 34, 1415, 2498
- prg internal commands:
 - __prg_break:
 - . 41, 2039, 2791, 5746, 6762, 7675,
 - 11477, 16458, 16979, 18007, 18013
 - __prg_break:n 41, 41,
 - 41, 2039, 2791, 5748, 6633, 6775, 7574
 - __prg_break_point:

- [41](#), [41](#), [738](#), [768](#), [2039](#),
[2791](#), [5736](#), [6630](#), [6763](#), [7569](#), [7660](#),
[11478](#), [16459](#), [16972](#), [16979](#), [18008](#)
- _prg_break_point:Nn ... [40](#), [40](#),
[40](#), [41](#), [95](#), [95](#), [117](#), [118](#), [127](#), [128](#),
[206](#), [206](#), [278](#), [278](#), [279](#), [342](#), [423](#),
[2030](#), [2791](#), [4050](#), [5384](#), [5401](#), [5410](#),
[6790](#), [6825](#), [6836](#), [7262](#), [7276](#), [7295](#),
[7313](#), [7704](#), [7723](#), [13434](#), [17800](#), [17968](#)
- _prg_case_end:nw
..... [23](#), [23](#), [3924](#), [4622](#), [5345](#), [5939](#)
- _prg_compare_error: [74](#), [74](#), [337](#),
[337](#), [356](#), [3820](#), [3838](#), [3840](#), [4561](#), [4563](#)
- _prg_compare_error:Nw
..... [74](#), [74](#), [337](#), [338](#), [339](#), [3820](#), [3860](#)
- _prg_generate_conditional:nnNnnnnn
..... [1425](#), [1444](#), [1453](#)
- _prg_generate_conditional:nnnnnnw
..... [1453](#)
- _prg_generate_conditional_
count:nnNnn [1428](#)
- _prg_generate_conditional_
count:nnNnnnn [1428](#)
- _prg_generate_conditional_
parm:nnNpnn [1415](#)
- _prg_generate_F_form:wnnnnnnn [1485](#)
- _prg_generate_p_form:wnnnnnnn [1485](#)
- _prg_generate_T_form:wnnnnnnn [1485](#)
- _prg_generate_TF_form:wnnnnnnn [1485](#)
- _prg_map_1:w [41](#)
- _prg_map_2:w [41](#)
- _prg_map_break:Nn
..... [41](#), [41](#), [279](#), [384](#), [439](#), [451](#),
[2030](#), [2791](#), [5423](#), [5425](#), [6780](#), [6782](#),
[7330](#), [7332](#), [7732](#), [7734](#), [17783](#), [17785](#)
- _g_prg_map_int
..... [41](#), [41](#), [342](#), [383](#), [2790](#),
[4029](#), [4032](#), [4036](#), [4039](#), [4050](#), [5396](#),
[5397](#), [5399](#), [5401](#), [6811](#), [6812](#), [6818](#),
[6819](#), [7290](#), [7291](#), [7293](#), [7296](#), [7719](#),
[7720](#), [7725](#), [7727](#), [13429](#), [13430](#),
[13433](#), [13434](#), [17792](#), [17794](#), [17801](#)
- _prg_replicate:N [2730](#)
- _prg_replicate [2730](#)
- _prg_replicate_0:n [2730](#)
- _prg_replicate_1:n [2730](#)
- _prg_replicate_2:n [2730](#)
- _prg_replicate_3:n [2730](#)
- _prg_replicate_4:n [2730](#)
- _prg_replicate_5:n [2730](#)
- _prg_replicate_6:n [2730](#)
- _prg_replicate_7:n [2730](#)
- _prg_replicate_8:n [2730](#)
- _prg_replicate_9:n [2730](#)
- _prg_replicate_first:N [2730](#)
- _prg_replicate_first_:n ... [2730](#)
- _prg_replicate_first_0:n ... [2730](#)
- _prg_replicate_first_1:n ... [2730](#)
- _prg_replicate_first_2:n ... [2730](#)
- _prg_replicate_first_3:n ... [2730](#)
- _prg_replicate_first_4:n ... [2730](#)
- _prg_replicate_first_5:n ... [2730](#)
- _prg_replicate_first_6:n ... [2730](#)
- _prg_replicate_first_7:n ... [2730](#)
- _prg_replicate_first_8:n ... [2730](#)
- _prg_replicate_first_9:n ... [2730](#)
- _prg_set_eq_conditional:NNNn [1520](#)
- _prg_set_eq_conditional:nnNnnNw
..... [1528](#), [1536](#)
- _prg_set_eq_conditional_F_
form:nnn [1536](#)
- _prg_set_eq_conditional_F_
form:wNnnnn [1582](#)
- _prg_set_eq_conditional_
loop:nnnnNw [1536](#)
- _prg_set_eq_conditional_p_
form:nnn [1536](#)
- _prg_set_eq_conditional_p_
form:wNnnnn [1567](#)
- _prg_set_eq_conditional_T_
form:nnn [1536](#)
- _prg_set_eq_conditional_T_
form:wNnnnn [1577](#)
- _prg_set_eq_conditional_TF_
form:nnn [1536](#)
- _prg_set_eq_conditional_TF_
form:wNnnnn [1572](#)
- \primitive [857](#)
- prop commands:
- \c_empty_prop
[137](#), [443](#), [7477](#), [7481](#), [7485](#), [7488](#), [7648](#)
- \prop_clear:N [132](#), [132](#), [7484](#), [7491](#), [8378](#)
- prop_clear:N [132](#)
- \prop_clear_new:N
..... [132](#), [132](#), [7490](#), [8021](#), [8022](#)
- \prop_count:N . [209](#), [209](#), [17954](#), [17989](#)
- \prop_gclear:N [132](#), [7484](#), [7494](#)
- \prop_gclear_new:N [132](#), [7490](#)
- \prop_get:Nn [41](#)
- \prop_get:NnN [42](#), [43](#), [133](#),
[133](#), [134](#), [7535](#), [8601](#), [8605](#), [8684](#), [8688](#)
- \prop_get:NnNTF [133](#),
[134](#), [135](#), [135](#), [7685](#), [7694](#), [7695](#),
[7697](#), [7698](#), [8155](#), [9140](#), [9160](#), [9215](#)
- \prop_gpop:NnN [133](#), [133](#), [7543](#)
- \prop_gpop:NnNTF
..... [133](#), [135](#), [135](#), [7578](#), [7601](#), [7602](#)

- \prop_gput:Nnn
133, 7604, 10774, 10828, 10913, 10945
- \prop_gput_if_new:Nnn 133, 7625
- \prop_gremove:Nn
..... 134, 7519, 10836, 10953
- \prop_gset_eq:NN
132, 7488, 7496, 8023, 8025, 8179, 8181
- \prop_if_empty:NTF 134, 134,
7646, 7652, 7653, 7738, 10848, 17986
- \prop_if_empty_p:N ... 134, 134, 7646
- \prop_if_exist:NTF
.... 134, 134, 7491, 7494, 7642, 7738
- \prop_if_exist_p:N ... 134, 134, 7642
- \prop_if_in:NnTF
134, 134, 7655, 7679, 7680, 7681, 7682
- \prop_if_in_p:Nn 134, 7655
- \prop_item:Nn 134, 134, 209, 7565
- \prop_log:N 210, 210, 17979
- \prop_map_break: ... 136, 136, 766,
7704, 7709, 7723, 7731, 17968, 17973
- \prop_map_break:n 136, 136, 7731
- \prop_map_function:NN
..... 135, 135, 209, 449,
766, 7700, 7739, 8752, 10850, 17959
- \prop_map_inline:Nn ... 135, 135,
7716, 8449, 8468, 8653, 8662, 17534,
17536, 17539, 17559, 17561, 17626,
17643, 17687, 17689, 17693, 17695
- \prop_map_tokens:Nn . 209, 209, 17964
- \prop_new:N 132,
132, 132, 7478, 7491, 7494, 7504,
7505, 7506, 7507, 7957, 7962, 8527,
8568, 9004, 9114, 10760, 10899, 17524
- \prop_pop:NnN 133, 133, 7543
- \prop_pop:NnNTF
.... 133, 135, 135, 7578, 7598, 7599
- \prop_put:Nnn . 133, 133, 137, 290,
443, 7604, 7958, 7959, 7960, 7961,
7964, 7965, 7966, 7968, 7969, 7970,
7971, 7972, 7973, 8207, 8213, 8215,
8217, 8219, 8224, 8229, 8234, 8241,
8248, 8473, 8528, 8530, 8532, 8534,
8536, 8538, 8540, 8542, 8544, 8546,
8548, 8550, 8552, 8554, 8556, 8558,
8560, 8562, 9188, 9204, 9221, 17567,
17569, 17572, 17574, 17580, 17586,
17653, 17661, 17724, 17738, 17745
- \prop_put_if_new:Nnn . 133, 133, 7625
- \prop_rand_key_value:N 210, 210, 17982
- \prop_remove:Nn 134,
134, 7519, 8648, 8651, 8655, 9185, 9200
- \prop_set_eq:NN 132, 132, 7485, 7496,
8165, 8167, 8172, 8174, 8408, 8643
- \prop_show:N 136, 136, 767, 7735, 17980
- \g_tmpa_prop 137, 7504
- \l_tmpa_prop 136, 7504
- \g_tmpb_prop 137, 7504
- \l_tmpb_prop 136, 7504
- prop internal commands:
- __prop_count:nn 17954
- __prop_if_in:N 449, 7655
- __prop_if_in:nwn 449, 7655
- \l__prop_internal_tl .. 137, 447,
447, 7476, 7608, 7614, 7615, 7631, 7638
- __prop_item:Nn:nwn 446
- __prop_item:Nn:nwn 7565
- __prop_map_function:Nwn 7700
- __prop_map_tokens:nwn .. 766, 17964
- __prop_pair:wn ... 137, 137, 137,
443, 443, 443, 444, 444, 449, 450,
450, 7474, 7513, 7516, 7568, 7571,
7610, 7633, 7658, 7662, 7703, 7706,
7719, 7721, 7726, 17967, 17970, 17993
- __prop_put:Nnn 7604
- __prop_put_if_new:Nnn 7625
- __prop_rand:NN 17982
- __prop_rand:nNn 17982, 17983
- __prop_rand_item:Nw 17982
- __prop_split:NnTF
.... 137, 137, 447, 447, 447, 448,
449, 7508, 7521, 7527, 7537, 7545,
7554, 7580, 7590, 7613, 7636, 7687
- __prop_split_aux:NnTF 7508
- __prop_split_aux:w .. 444, 444, 7508
- \protect 542, 12047, 18415, 18442
- \protected 207, 209, 211, 236, 654, 3370, 3372
- \protrudechars 962
- \ProvidesExplClass 6
- \ProvidesExplFile 6, 19418
- \ProvidesExplPackage 6, 6
- pt 197
- ptex commands:
- \ptex_autospacing:D 1118
- \ptex_autoxspacing:D 1119
- \ptex_dtou:D 1120
- \ptex_euc:D 1121
- \ptex_ifdbbox:D 1122
- \ptex_ifddir:D 1123
- \ptex_ifmdir:D 1124
- \ptex_iftbody:D 1125
- \ptex_iftdir:D 1126
- \ptex_ifybox:D 1127
- \ptex_ifydir:D 1128
- \ptex_inhibitglue:D 1129
- \ptex_inhibitxspcode:D 1130
- \ptex_jcharwidowpenalty:D 1131
- \ptex_jfam:D 1132
- \ptex_jfont:D 1133

- `\ptex_jis:D` 1134, 3758, 19260
`\ptex_kanjiskip:D` 1135, 19255
`\ptex_kansuji:D` 1136
`\ptex_kansujichar:D` 1137
`\ptex_kcatcode:D` 1138
`\ptex_kuten:D` 1139
`\ptex_noautospaceing:D` 1140
`\ptex_noautoxspacing:D` 1141
`\ptex_postbreakpenalty:D` 1142
`\ptex_prebreakpenalty:D` 1143
`\ptex_showmode:D` 1144
`\ptex_sjis:D` 1145
`\ptex_tate:D` 1146
`\ptex_tbaselineshift:D` 1147
`\ptex_tfont:D` 1148
`\ptex_xkanjiskip:D` 1149
`\ptex_xspcode:D` 1150
`\ptex_ybaselineshift:D` 1151
`\ptex_yoko:D` 1152
`\pxdimen` 963
- Q**
- quark commands:
- `\q_mark` 23, 23, 43,
 101, 101, 266, 266, 266, 289, 291,
 299, 376, 376, 383, 383, 386, 386,
 386, 386, 399, 399, 406, 429, 429,
 431, 431, 434, 434, 434, 435, 435,
 440, 440, 440, 444, 503, 737, 737,
 737, 738, 740, 1602, 1603, 1609,
 1611, 1612, 2325, 2326, 2328, 2335,
 2341, 2365, 2374, 2388, 2396, 2399,
 2408, 2423, 2445, 2466, 2469, 2482,
 2613, 2614, 2615, 2616, 2619, 2620,
 2621, 2622, 2623, 2624, 2625, 2795,
 3848, 3851, 3917, 4615, 5161, 5163,
 5165, 5167, 5366, 5377, 5453, 5454,
 5457, 5460, 5461, 5467, 5481, 5482,
 5488, 5492, 5494, 5497, 5900, 5932,
 5943, 5960, 6182, 6184, 6862, 6863,
 6877, 6880, 7014, 7023, 7028, 7083,
 7093, 7097, 7121, 7173, 7179, 7192,
 7204, 7205, 7206, 7209, 7210, 7211,
 7220, 7221, 7230, 7275, 7283, 7371,
 7372, 7384, 7385, 7448, 7513, 7515,
 7516, 9145, 9146, 9151, 9154, 9631,
 9638, 9648, 9672, 9694, 9704, 9714,
 11967, 11968, 11973, 16985, 17020,
 17022, 17025, 17029, 17032, 17035,
 17037, 17040, 19187, 19188, 19195
`\q_nil` .. 19, 19, 43, 43, 43, 43, 259,
 299, 299, 299, 307, 309, 309, 309,
 376, 379, 379, 379, 379, 386, 1396,
 1399, 2612, 2614, 2615, 2619, 2620,
 2621, 2622, 2623, 2624, 2795, 2848,
 2869, 2871, 5187, 5265, 5266, 5278,
 5279, 5480, 5484, 5502, 5505, 5508,
 5596, 5597, 9648, 9658, 9672, 9682
`\q_no_value` 42, 43,
 43, 43, 43, 113, 114, 114, 114, 114,
 114, 119, 119, 119, 129, 133, 133,
 133, 173, 307, 309, 419, 420, 431,
 445, 445, 530, 2671, 2795, 2856,
 2875, 2877, 6643, 6651, 6663, 6687,
 7066, 7081, 7539, 7550, 7559, 10642
`\quark_if_nil:n` 309, 309, 309
`\quark_if_nil:NTF` 43, 43, 2846
`\quark_if_nil:nTF`
 .. 43, 43, 308, 2866, 2880, 2881, 5185
`\quark_if_nil_p:N` 43, 43, 2846
`\quark_if_nil_p:n` 43, 43, 2866
`\quark_if_no_value:NTF` 43,
 43, 2676, 2846, 2863, 2864, 8603,
 8607, 8686, 8690, 10673, 10785, 10797
`\quark_if_no_value:nTF` .. 43, 43, 2866
`\quark_if_no_value_p:N` .. 43, 43, 2846
`\quark_if_no_value_p:n` .. 43, 43, 2866
`\quark_if_recursion_tail_stop:N` .
 44, 44, 2801, 4385, 7324, 18952
`\quark_if_recursion_tail_stop:n` .
 . 44, 44, 394, 2815, 7019, 7354, 18785
`quark_if_recursion_tail_stop:n` . 308
`\quark_if_recursion_tail_stop_-`
`do:Nn` 44, 44,
 2801, 4324, 4341, 4388, 6220, 18236,
 18244, 18403, 18423, 19022, 19030
`\quark_if_recursion_tail_stop_-`
`do:nn`
 ... 44, 44, 2815, 17922, 17937, 19075
`\quark_new:N` 42, 42, 2794, 2795, 2796,
 2797, 2798, 2799, 2800, 2882, 2883
`\q_recursion_stop` 19,
 19, 44, 44, 44, 44, 44, 45, 259, 263,
 307, 1398, 1401, 1465, 1533, 2315,
 2799, 4319, 4336, 4379, 4406, 5023,
 6198, 6205, 6210, 6398, 7015, 7312,
 7348, 17919, 17934, 18194, 18197,
 18206, 18217, 18227, 18234, 18240,
 18259, 18261, 18270, 18272, 18279,
 18280, 18283, 18286, 18299, 18397,
 18419, 18441, 18467, 18472, 18475,
 18477, 18480, 18483, 18488, 18509,
 18512, 18514, 18516, 18521, 18604,
 18608, 18610, 18615, 18629, 18654,
 18658, 18660, 18665, 18675, 18924,
 18970, 18989, 18992, 19001, 19003,
 19013, 19020, 19026, 19048, 19052,

19055, 19056, 19071, 19087, 19089,
 19112, 19116, 19118, 19123, 19140
 \q_recursion_tail 44, 44, 44, 44, 44,
 44, 44, 44, 44, 45, 45, 45, 263, 307,
 308, 437, 449, 450, 450, 766, 1465,
 1470, 1533, 1552, 2799, 2803, 2809,
 2818, 2825, 2830, 2835, 2842, 4319,
 4336, 4379, 4999, 5023, 5383, 5400,
 5409, 5735, 6198, 6388, 6398, 7015,
 7261, 7275, 7294, 7312, 7348, 7659,
 7670, 7703, 7708, 9631, 9638, 9699,
 17919, 17934, 17967, 17972, 18194,
 18240, 18274, 18397, 18419, 18452,
 18923, 18969, 18989, 19026, 19071
 \q_stop 19, 19,
 23, 23, 29, 42, 43, 43, 99, 259,
 266, 291, 307, 322, 324, 337, 349,
 349, 376, 385, 399, 403, 435, 435,
 435, 439, 444, 498, 738, 738, 740,
 1397, 1400, 1481, 1486, 1503, 1509,
 1515, 1563, 1567, 1572, 1577, 1582,
 1604, 1609, 1611, 1612, 2329, 2342,
 2369, 2396, 2400, 2404, 2412, 2418,
 2427, 2445, 2472, 2483, 2617, 2625,
 2671, 2674, 2795, 3286, 3289, 3321,
 3355, 3390, 3394, 3400, 3423, 3617,
 3624, 3633, 3642, 3827, 3843, 3845,
 3849, 3862, 3917, 4312, 4318, 4335,
 4566, 4592, 4615, 4786, 4788, 5186,
 5338, 5344, 5366, 5377, 5455, 5457,
 5462, 5464, 5486, 5508, 5587, 5589,
 5606, 5624, 5643, 5665, 5900, 5932,
 5943, 5952, 5958, 5960, 5980, 6039,
 6095, 6107, 6145, 6161, 6168, 6176,
 6178, 6182, 6184, 6308, 6313, 6316,
 6320, 6338, 6351, 6362, 6366, 6369,
 6371, 6372, 6375, 6378, 6663, 6666,
 6674, 6676, 6756, 6757, 6864, 6877,
 6880, 6882, 7068, 7071, 7083, 7086,
 7094, 7097, 7105, 7121, 7179, 7206,
 7209, 7210, 7222, 7230, 7373, 7384,
 7385, 7386, 7412, 7446, 7513, 7516,
 9147, 9590, 9607, 9702, 9717, 9739,
 9753, 9754, 9819, 9821, 9841, 9845,
 9854, 9862, 9875, 10280, 10288,
 10298, 10425, 10427, 10432, 11071,
 11968, 11973, 12082, 12087, 16501,
 16572, 16973, 16982, 16986, 16988,
 17020, 17021, 17022, 17027, 17029,
 17033, 17035, 17043, 17915, 17990,
 18003, 18004, 18006, 18010, 18014,
 18016, 18021, 19176, 19189, 19198
 quark internal commands:
 \s_fp
 . 547, 547, 547, 548, 548, 548, 549,
 549, 549, 550, 580, 581, 584, 596,
 596, 598, 623, 626, 626, 628, 628,
 628, 630, 636, 636, 639, 639, 719,
 11210, 11223, 11224, 11225, 11226,
 11227, 11230, 11232, 11235, 11241,
 11245, 11266, 11269, 11271, 11281,
 11291, 11303, 11323, 11395, 11397,
 11399, 11400, 11401, 11403, 11404,
 11405, 11407, 11433, 11619, 11624,
 11825, 11871, 11880, 11882, 12529,
 12649, 12890, 13246, 13271, 13272,
 13391, 13404, 13408, 13448, 13449,
 13452, 13463, 13464, 13472, 13473,
 13475, 13476, 13477, 13479, 13480,
 13481, 13492, 13495, 13507, 13533,
 13581, 13584, 13587, 13606, 13607,
 13609, 13610, 13611, 13619, 13622,
 13638, 13639, 13641, 13650, 13726,
 13877, 13911, 13912, 13915, 13994,
 14130, 14138, 14140, 14317, 14320,
 14778, 14790, 14792, 15000, 15017,
 15019, 15208, 15227, 15229, 15230,
 15233, 15245, 15250, 15253, 15278,
 15279, 15281, 15297, 15382, 15395,
 15397, 15400, 15405, 15452, 15465,
 15467, 15480, 15482, 15495, 15497,
 15510, 15512, 15525, 15527, 15540,
 15550, 15982, 15997, 15998, 16002,
 16013, 16119, 16132, 16134, 16150,
 16153, 16163, 16186, 16197, 16199,
 16213, 16215, 16220, 16248, 16273,
 16276, 16297, 16321, 16324, 16365,
 16388, 16438, 16439, 16566, 16568
 \s__fp_division 11218
 \s__fp_exact 11218, 11223,
 11224, 11225, 11226, 11227, 13448
 \s__fp_invalid 11218
 \s__fp_mark 560, 580, 580,
 584, 604, 607, 615, 11216, 11487,
 11488, 11490, 11494, 12758, 12799,
 13199, 13200, 13203, 13206, 13207
 \s__fp_overflow 11218, 11232
 \s__fp_stop 11216, 12611,
 12759, 12762, 13201, 13206, 13207,
 13209, 13511, 13522, 13546, 13554
 \s__fp_underflow 11218, 11230
 \s__prop 137, 137, 443, 443,
 443, 443, 444, 444, 450, 450, 766,
 7473, 7474, 7477, 7513, 7516, 7568,
 7571, 7611, 7634, 7658, 7662, 7703,
 7706, 7721, 17967, 17970, 17993, 17998
 __quark_if_nil:w 309, 309, 2866
 __quark_if_no_value:w 2866

- `__quark_if_recursion_tail:w` ... 308, [2815](#), [2842](#)
`__quark_if_recursion_tail_-break:NN` [45](#), [2833](#), [5417](#)
`__quark_if_recursion_tail_-break:nN` [45](#), [45](#), [2833](#), [5390](#), [5746](#), [7267](#), [7280](#)
`\s__seq` [122](#), [412](#), [414](#), [415](#), [415](#), [420](#), [425](#), [731](#), [767](#), [2900](#), [6404](#), [6412](#), [6442](#), [6447](#), [6452](#), [6457](#), [6490](#), [6516](#), [6524](#), [6528](#), [6707](#), [6757](#), [6874](#), [16817](#), [16824](#), [18004](#), [18010](#)
`\s__stop` [46](#), [46](#), [46](#), [738](#), [2898](#), [2899](#), [15078](#), [15093](#), [16235](#), [16239](#), [16986](#), [16988](#)
`\q__tl_act_mark` [387](#), [387](#), [387](#), [2882](#), [5512](#), [5515](#), [5532](#)
`\q__tl_act_stop` [387](#), [2882](#), [5512](#), [5515](#), [5519](#), [5528](#), [5530](#), [5536](#), [5541](#), [5544](#), [5548](#), [5551](#)
`\quitvmode` [784](#)
- R**
- `\r` [18976](#)
`\radical` [517](#)
`\raise` [518](#)
`\rand` [196](#)
`\randint` [196](#)
`\randomseed` [964](#)
`\read` [519](#)
`\readline` [655](#)
`\ref` [19151](#)
`\relax` [14](#), [21](#), [39](#), [43](#), [49](#), [90](#), [92](#), [98](#), [123](#), [146](#), [167](#), [181](#), [212](#), [213](#), [214](#), [215](#), [216](#), [217](#), [218](#), [219](#), [220](#), [221](#), [222](#), [225](#), [226](#), [227](#), [228](#), [229](#), [230](#), [231](#), [232](#), [233](#), [234](#), [235](#), [520](#)
`\relpenalty` [521](#)
`\RequirePackage` [149](#)
reverse commands:
`\reverse_if:N` ... [21](#), [21](#), [337](#), [337](#), [338](#), [405](#), [583](#), [1292](#), [3875](#), [3877](#), [3879](#), [3881](#), [4580](#), [4585](#), [4589](#), [4591](#), [6175](#), [6315](#), [12000](#), [15267](#), [15920](#), [15943](#)
`\right` [522](#)
right commands:
`\c_right_brace_str` [110](#), [6249](#)
`\rightghost` [943](#)
`\righthyphenmin` [523](#)
`\rightmarginkern` [785](#)
`\rightskip` [524](#)
`\romannumeral` [525](#)
`\round` [193](#)
`\rpcode` [786](#)
- `\rule` [8585](#), [8640](#)
- S**
- `\saveboxresource` [968](#)
`\savecatcodetable` [918](#)
`\saveimageresource` [969](#)
`\savepos` [967](#)
`\savingshyphcodes` [656](#)
`\savingsdiscards` [657](#)
scan commands:
`\scan_stop:` [9](#), [9](#), [45](#), [46](#), [58](#), [58](#), [58](#), [59](#), [122](#), [252](#), [266](#), [267](#), [267](#), [267](#), [270](#), [282](#), [282](#), [310](#), [321](#), [322](#), [328](#), [331](#), [338](#), [342](#), [372](#), [372](#), [385](#), [405](#), [405](#), [443](#), [450](#), [498](#), [561](#), [580](#), [583](#), [583](#), [584](#), [585](#), [588](#), [795](#), [1319](#), [1488](#), [1620](#), [1638](#), [1648](#), [1666](#), [2082](#), [2315](#), [2356](#), [2398](#), [2895](#), [3226](#), [3300](#), [3512](#), [3626](#), [3635](#), [3644](#), [4050](#), [4750](#), [4761](#), [4766](#), [4792](#), [4798](#), [4801](#), [4844](#), [4855](#), [4860](#), [4865](#), [5078](#), [5102](#), [6176](#), [6284](#), [6289](#), [6290](#), [6291](#), [6354](#), [6705](#), [7857](#), [7876](#), [8581](#), [8636](#), [9658](#), [9682](#), [9719](#), [9723](#), [9724](#), [9741](#), [9742](#), [10767](#), [10769](#), [10827](#), [10829](#), [10906](#), [10908](#), [10944](#), [10946](#), [11998](#), [12002](#), [12161](#), [12179](#), [12479](#), [12526](#), [12527](#), [12797](#), [12828](#), [13079](#), [13434](#), [16738](#), [18059](#), [18060](#), [18144](#), [18147](#), [18168](#), [19208](#), [19210](#), [19212](#), [19451](#), [19457](#), [19492](#)
scan internal commands:
`\g__scan_marks_tl` ... [2885](#), [2888](#), [2894](#)
`__scan_new:N` . [46](#), [46](#), [2886](#), [2898](#), [2900](#), [7473](#), [11210](#), [11216](#), [11217](#), [11218](#), [11219](#), [11220](#), [11221](#), [11222](#)
`\scantextokens` [919](#)
`\scantokens` [658](#)
`\scriptfont` [526](#)
`\scriptscriptfont` [527](#)
`\scriptscriptstyle` [528](#)
`\scriptspace` [529](#)
`\scriptstyle` [530](#)
`\scrollmode` [531](#)
`sec` [194](#)
`secd` [194](#)
seq commands:
`\c_empty_seq` [121](#), [412](#), [6412](#), [6416](#), [6420](#), [6423](#), [6605](#), [6642](#), [6650](#)
`\seq_clear:N` [112](#), [112](#), [121](#), [6419](#), [6426](#), [6549](#), [9143](#), [9206](#)
`seq_clear:N` [112](#)
`\seq_clear_new:N` [112](#), [112](#), [6425](#)
`\seq_concat:NNN` [113](#), [113](#), [121](#), [121](#), [6502](#), [10649](#)

- \seq_count:N 114,
118, 118, 120, 6768, 6840, 6854, 18051
- \seq_elt:w 412, 412
- \seq_elt_end: 412, 412
- \seq_gclear:N 112, 6419, 6429
- \seq_gclear_new:N 112, 6425
- \seq_gconcat:NNN 113, 6502
- \seq_get:NN 119, 119, 6907
- \seq_get:NNTF 119, 119, 6913
- \seq_get_left:NN
113, 113, 6658, 6907, 6908, 6913, 6914
- \seq_get_left:NNTF
..... 115, 115, 6725, 6729, 6730
- \seq_get_right:NN 114, 114, 6683
- \seq_get_right:NNTF
..... 115, 115, 6725, 6732, 6733
- \seq_gpop:NN ... 119, 119, 6907, 10711
- \seq_gpop:NNTF
..... 120, 120, 6913, 10811, 10928
- \seq_gpop_left:NN
114, 114, 6669, 6911, 6912, 6917, 6918
- \seq_gpop_left:NNTF
..... 115, 115, 6735, 6746, 6747
- \seq_gpop_right:NN ... 114, 114, 6698
- \seq_gpop_right:NNTF
..... 115, 115, 6735, 6752, 6753
- \seq_gpush:Nn
.. 24, 120, 6887, 10708, 10838, 10955
- \seq_gput_left:Nn
113, 6512, 6897, 6898, 6899, 6900,
6901, 6902, 6903, 6904, 6905, 6906
- \seq_gput_right:Nn 113, 6533,
10573, 10701, 10706, 10749, 10893
- \seq_gremove_all:Nn 116, 6559
- \seq_gremove_duplicates:N . 116, 6543
- \seq_greverse:N 116, 6585
- \seq_gset_eq:NN 112, 6423, 6431, 6546
- \seq_gset_filter:NNn 210, 18025
- \seq_gset_from_clist:NN ... 112, 6439
- \seq_gset_from_clist:Nn ... 112, 6439
- \seq_gset_map:NNn 210, 18035
- \seq_gset_split:Nnn . 113, 6465, 10756
- \seq_gsort:Nn 116, 6603, 16813
- \seq_if_empty:NTF 116, 116,
6603, 6612, 6613, 6922, 6963, 18050
- \seq_if_empty_p:N 116, 116, 6603
- \seq_if_exist:NTF
113, 113, 6426, 6429, 6508, 6852, 6922
- \seq_if_exist_p:N 113, 113, 6508
- \seq_if_in:NnTF ... 116, 116, 120,
120, 121, 121, 6552, 6615, 6634,
6635, 6636, 6637, 10719, 10837, 10954
- \seq_item:Nn 114,
114, 769, 6755, 9224, 9225, 9230, 18051
- \seq_log:N 210, 210, 18045
- \seq_map_break:
..... 117, 117, 210, 210, 6779,
6789, 6790, 6825, 6836, 10325, 10659
- \seq_map_break:n 118, 118,
423, 6779, 9163, 9177, 16816, 16823
- \seq_map_function:NN
..... 4, 116, 116, 116,
500, 501, 6783, 6845, 6923, 6969, 9228
- \seq_map_inline:Nn
116, 117, 117, 121, 121, 121, 121,
121, 768, 6550, 6821, 9158, 10318,
10588, 10653, 10742, 16816, 16823
- \seq_map_variable:NNn 117, 117, 6828
- \seq_mapthread_function:NNN
..... 210, 210, 18002
- \seq_new:N 4, 112,
112, 112, 3013, 3016, 6413, 6426,
6429, 6542, 6926, 6927, 6928, 6929,
9115, 9116, 9780, 10567, 10568,
10578, 10580, 10583, 10754, 10884
- \seq_pop:NN 119, 119, 6907
- \seq_pop:NNTF 120, 120, 6913
- \seq_pop_left:NN
114, 114, 6669, 6909, 6910, 6915, 6916
- \seq_pop_left:NNTF
..... 115, 115, 6735, 6743, 6744
- \seq_pop_right:NN 114, 114, 6698
- \seq_pop_right:NNTF
..... 115, 115, 6735, 6749, 6750
- \seq_push:Nn 120, 120, 6887, 6894
- \seq_put_left:Nn 113, 113,
6512, 6887, 6888, 6889, 6890, 6891,
6892, 6893, 6894, 6895, 6896, 9153
- \seq_put_right:Nn 113, 113, 120, 121,
121, 6533, 6553, 9214, 10720, 10735
- \seq_rand_item:N 211, 211, 18048
- \seq_remove_all:Nn . 113, 116, 116,
120, 120, 121, 121, 121, 6559, 10725
- \seq_remove_duplicates:N
..... 116, 116, 120, 121, 6543, 10740
- \seq_reverse:N ... 116, 116, 417, 6585
- \seq_set_eq:NN
112, 112, 121, 121, 121, 121, 6420,
6431, 6544, 10646, 10665, 10729, 10886
- \seq_set_filter:NNn
..... 210, 210, 768, 18025
- \seq_set_from_clist:NN 112, 112, 6439
- \seq_set_from_clist:Nn
..... 112, 6439, 10247, 10256
- \seq_set_map:NNn 210, 210, 18035
- \seq_set_split:Nnn
113, 113, 113, 3014, 3017, 6465, 10648
- \seq_show:N . 122, 122, 769, 6919, 18046

- \seq_sort:Nn . . . [116](#), [116](#), [6603](#), [16813](#)
- \seq_use:Nn [119](#), [119](#), [6850](#)
- \seq_use:Nnnn [118](#), [118](#), [6850](#)
- \g_tmpa_seq [122](#), [6926](#)
- \l_tmpa_seq [122](#), [6926](#)
- \g_tmpb_seq [122](#), [6926](#)
- \l_tmpb_seq [122](#), [6926](#)
- seq internal commands:
 - __seq_count:n [6840](#)
 - __seq_get_left:wnw [6658](#)
 - __seq_get_right_loop:nn . . [420](#), [6683](#)
 - __seq_if_in: [6615](#)
 - \l__seq_internal_a_tl
 - [414](#), [6409](#), [6473](#), [6477](#), [6483](#),
 - [6488](#), [6490](#), [6574](#), [6579](#), [6619](#), [6623](#)
 - \l__seq_internal_b_tl
 - [6409](#), [6570](#), [6574](#), [6622](#), [6623](#)
 - __seq_item:n [122](#),
 - [122](#), [122](#), [122](#), [412](#), [412](#), [415](#), [418](#),
 - [418](#), [419](#), [420](#), [421](#), [423](#), [423](#), [423](#),
 - [424](#), [424](#), [425](#), [425](#), [731](#), [732](#), [732](#),
 - [767](#), [768](#), [6404](#), [6516](#), [6524](#), [6534](#),
 - [6536](#), [6541](#), [6591](#), [6592](#), [6594](#), [6599](#),
 - [6620](#), [6663](#), [6666](#), [6676](#), [6704](#), [6705](#),
 - [6716](#), [6802](#), [6807](#), [6813](#), [6817](#), [6861](#),
 - [6876](#), [6879](#), [6882](#), [16817](#), [16824](#), [18041](#)
 - __seq_item:nN [6755](#)
 - __seq_item:nnn [6755](#)
 - __seq_item:wNn [6755](#)
 - __seq_map_function:NNn [6783](#)
 - __seq_mapthread_function:Nnnwnn
 - [18002](#)
 - __seq_mapthread_function:wNN . [18002](#)
 - __seq_mapthread_function:wNw . [18002](#)
 - __seq_pop:NNNN
 - [6640](#), [6670](#), [6672](#), [6699](#), [6701](#)
 - __seq_pop_item_def: [122](#), [122](#), [122](#),
 - [6581](#), [6799](#), [6825](#), [6836](#), [18033](#), [18043](#)
 - __seq_pop_left:NNN . [6669](#), [6736](#), [6738](#)
 - __seq_pop_left:wnwNNN [6669](#)
 - __seq_pop_right:NNN
 - [416](#), [6698](#), [6740](#), [6742](#)
 - __seq_pop_right_loop:nn [6698](#)
 - __seq_pop_TF:NNNN [421](#), [6640](#),
 - [6726](#), [6728](#), [6736](#), [6738](#), [6740](#), [6742](#)
 - __seq_push_item_def: [6799](#)
 - __seq_push_item_def:n
 - [122](#), [122](#), [122](#), [122](#),
 - [6565](#), [6799](#), [6823](#), [6830](#), [18031](#), [18041](#)
 - __seq_put_left_aux:w [415](#), [6512](#)
 - __seq_remove_all_aux:NNn [6559](#)
 - __seq_remove_duplicates:NN . . [6543](#)
 - \l__seq_remove_seq
 - [6542](#), [6549](#), [6552](#), [6553](#), [6555](#)
 - __seq_reverse:NN [6585](#)
 - __seq_reverse_item:nw [417](#), [417](#)
 - __seq_reverse_item:nwn [6585](#)
 - __seq_set_filter:NNNn [18025](#)
 - __seq_set_map:NNNn [18035](#)
 - __seq_set_split:NNnn [6465](#)
 - __seq_set_split_auxi:w [414](#), [414](#), [6465](#)
 - __seq_set_split_auxii:w . . [414](#), [6465](#)
 - __seq_set_split_end: [414](#), [414](#), [6465](#)
 - __seq_tmp:w
 - [6411](#), [6591](#), [6594](#), [6704](#), [6716](#)
 - __seq_use:NNnNn [6850](#)
 - __seq_use:nwnn [6850](#)
 - __seq_use:nwwwwnn [6850](#)
 - __seq_use_setup:w [6850](#)
 - __seq_wrap_item:n
 - [414](#), [768](#), [6442](#), [6447](#), [6452](#),
 - [6457](#), [6474](#), [6499](#), [6541](#), [6577](#), [18031](#)
 - \setbox [532](#)
 - \setfontid [920](#)
 - \setlanguage [533](#)
 - \setrandomseed [970](#)
 - \sfcode [183](#), [534](#)
 - \sffamily [8574](#)
 - \shapemode [921](#)
 - \shellescape [858](#)
 - \shipout [535](#)
 - \ShortText [75](#), [116](#), [133](#)
 - \show [536](#)
 - \showbox [537](#)
 - \showboxbreadth [538](#)
 - \showboxdepth [539](#)
 - \showgroups [659](#)
 - \showifs [660](#)
 - \showlists [540](#)
 - \showmode [1144](#)
 - \showthe [541](#)
 - \showtokens [661](#)
 - sin [194](#)
 - sind [194](#)
 - \sjis [1145](#)
 - \skewchar [542](#)
 - \skip [543](#), [3376](#)
 - skip commands:
 - \c_max_skip [85](#), [4808](#)
 - \skip_add:Nn [83](#), [83](#), [4760](#)
 - \skip_const:Nn [83](#), [83](#), [4729](#), [4808](#), [4809](#)
 - \skip_eval:n
 - [84](#), [84](#), [84](#), [84](#), [84](#), [4773](#), [4791](#), [4807](#)
 - \skip_gadd:Nn [83](#), [4760](#)
 - .skip_gset:N [165](#), [10174](#)
 - \skip_gset:Nn [83](#), [4732](#), [4749](#)
 - \skip_gset_eq:NN [83](#), [4754](#)
 - \skip_gsub:Nn [83](#), [4760](#)

- \skip_gzero:N [83](#), [4735](#), [4742](#)
- \skip_gzero_new:N [83](#), [4739](#)
- \skip_horizontal:N .. [85](#), [85](#), [85](#), [4796](#)
- \skip_horizontal:n
..... [85](#), [85](#), [4796](#), [19554](#), [19832](#), [20220](#)
- \skip_if_eq:nnTF [84](#), [4770](#)
- \skip_if_eq_p:nn [84](#), [84](#), [4770](#)
- \skip_if_exist:NTF
..... [83](#), [83](#), [4740](#), [4742](#), [4745](#)
- \skip_if_exist_p:N [83](#), [83](#), [4745](#)
- \skip_if_finite:nTF [84](#), [84](#), [4780](#), [18057](#)
- \skip_if_finite_p:n [84](#), [84](#), [4780](#)
- \skip_log:N [211](#), [211](#), [18072](#)
- \skip_log:n [211](#), [211](#), [18072](#)
- \skip_new:N . [82](#), [82](#), [83](#), [4721](#), [4731](#),
[4740](#), [4742](#), [4810](#), [4811](#), [4812](#), [4813](#)
- .skip_set:N [165](#), [10174](#)
- \skip_set:Nn [83](#), [83](#), [4749](#)
- \skip_set_eq:NN [83](#), [83](#), [4754](#)
- \skip_show:N [84](#), [84](#), [4804](#)
- \skip_show:n . [84](#), [84](#), [770](#), [4806](#), [18075](#)
- \skip_split_finite_else_action:nnNN
..... [211](#), [211](#), [18055](#)
- \skip_sub:Nn [83](#), [83](#), [4760](#)
- \skip_use:N
..... [84](#), [84](#), [84](#), [84](#), [4785](#), [4792](#), [4793](#)
- \skip_vertical:N [85](#), [85](#), [85](#), [4796](#)
- \skip_vertical:n [85](#), [85](#), [4796](#)
- \skip_zero:N ... [83](#), [83](#), [86](#), [4735](#), [4740](#)
- skip_zero:N [83](#)
- \skip_zero_new:N [83](#), [83](#), [4739](#)
- \g_tmpa_skip [85](#), [4810](#)
- \l_tmpa_skip [85](#), [4810](#)
- \g_tmpb_skip [85](#), [4810](#)
- \l_tmpb_skip [85](#), [4810](#)
- \c_zero_skip
..... [85](#), [4735](#), [4808](#), [18063](#), [18064](#)
- skip internal commands:
- __skip_if_finite:wwNw [4780](#)
- \skipdef [544](#)
- sort commands:
- \sort_ordered: [17102](#)
- \sort_return_same:
..... [200](#), [200](#), [734](#), [16919](#), [17102](#)
- \sort_return_swapped:
..... [200](#), [200](#), [734](#), [16919](#), [17103](#)
- \sort_reversed: [17102](#)
- sort internal commands:
- __sort:nnNnn . [736](#), [736](#), [736](#), [736](#), [737](#)
- \l__sort_A_int
..... [734](#), [16702](#), [16707](#), [16714](#),
[16717](#), [16726](#), [16884](#), [16889](#), [16892](#),
[16912](#), [16927](#), [16946](#), [16948](#), [16949](#)
- \l__sort_B_int
..... [734](#), [734](#), [16702](#), [16889](#), [16893](#),
[16901](#), [16903](#), [16904](#), [16936](#), [16937](#),
[16946](#), [16947](#), [16956](#), [16957](#), [16959](#)
- \l__sort_begin_int [728](#),
[734](#), [734](#), [16700](#), [16881](#), [16949](#), [16959](#)
- \l__sort_block_int
..... [728](#), [728](#), [733](#), [16699](#), [16709](#),
[16714](#), [16718](#), [16721](#), [16726](#), [16727](#),
[16805](#), [16872](#), [16875](#), [16882](#), [16885](#)
- \l__sort_C_int
..... [734](#), [734](#), [16702](#), [16890](#),
[16894](#), [16901](#), [16902](#), [16913](#), [16928](#),
[16936](#), [16938](#), [16939](#), [16956](#), [16958](#)
- __sort_clist:NNn [16841](#)
- __sort_compare:nn . [735](#), [16804](#), [16911](#)
- __sort_compute_range:
..... [727](#), [728](#), [729](#), [16731](#), [16792](#)
- __sort_copy_block: [733](#), [16891](#), [16899](#)
- __sort_disable_toksdef: [16791](#), [17051](#)
- __sort_disabled_toksdef:n ... [17051](#)
- \l__sort_end_int [728](#), [733](#), [734](#), [734](#),
[16700](#), [16873](#), [16881](#), [16882](#), [16883](#),
[16884](#), [16885](#), [16886](#), [16887](#), [16904](#)
- __sort_error: .. [17045](#), [17058](#), [17076](#)
- __sort_i:nnnnNn [738](#)
- \l__sort_length_int
..... [727](#), [728](#), [16694](#), [16802](#), [16872](#)
- __sort_level: [740](#), [16806](#), [16870](#), [17049](#)
- __sort_loop:wNn [737](#), [737](#), [737](#), [737](#), [737](#)
- __sort_main:NNNnNn .. [731](#), [16788](#),
[16815](#), [16822](#), [16829](#), [16836](#), [16851](#)
- \l__sort_max_int [727](#),
[728](#), [728](#), [16694](#), [16711](#), [16785](#), [16796](#)
- \c__sort_max_length_int [16731](#)
- __sort_merge_blocks:
..... [16874](#), [16879](#), [17048](#)
- __sort_merge_blocks_aux:
[733](#), [16895](#), [16909](#), [16942](#), [16952](#), [17047](#)
- __sort_merge_blocks_end:
..... [735](#), [16950](#), [16954](#)
- \l__sort_min_int ... [727](#), [728](#), [730](#),
[732](#), [16694](#), [16708](#), [16716](#), [16734](#),
[16750](#), [16758](#), [16771](#), [16783](#), [16793](#),
[16803](#), [16861](#), [16873](#), [17074](#), [17075](#)
- __sort_quick_cleanup:w [16964](#)
- __sort_quick_end:nnTFNn
..... [739](#), [740](#), [16984](#), [17019](#)
- __sort_quick_only_i:NnnnnNn . [16989](#)
- __sort_quick_only_i_end:nnwnw .
..... [16998](#), [17019](#)
- __sort_quick_only_ii:NnnnnNn . [16989](#)
- __sort_quick_only_ii_end:nnwnw
..... [17004](#), [17019](#)

- __sort_quick_prepare:Nnnn ... [16964](#)
- __sort_quick_prepare_end:NNNnw .
..... [16964](#)
- __sort_quick_single_end:nnnwnw .
..... [16992](#), [17019](#)
- __sort_quick_split:NnNn
.. [737](#), [737](#), [737](#), [738](#), [738](#), [16984](#),
[16989](#), [17024](#), [17031](#), [17037](#), [17039](#)
- __sort_quick_split_end:nnnwnw ..
..... [17010](#), [17016](#), [17019](#)
- __sort_quick_split_i:NnnnnNn ...
..... [737](#), [16989](#)
- __sort_quick_split_ii:NnnnnNn [16989](#)
- __sort_redefine_compute_range: .
..... [16731](#)
- __sort_return_mark:N
..... [16915](#), [16916](#), [16919](#)
- __sort_return_none_error:
..... [734](#), [16917](#), [16919](#)
- __sort_return_same:
..... [16920](#), [16929](#), [16934](#)
- __sort_return_swapped: [16922](#), [16944](#)
- __sort_return_two_error:w ... [16919](#)
- __sort_shrink_range: .. [728](#), [729](#),
[729](#), [16705](#), [16736](#), [16752](#), [16760](#), [16773](#)
- __sort_shrink_range_loop: ... [16705](#)
- __sort_toks:NN [732](#), [16817](#),
[16824](#), [16831](#), [16838](#), [16855](#), [16860](#)
- __sort_toks:NNw [16860](#)
- __sort_too_long_error:NNw
..... [16797](#), [17069](#)
- \l__sort_top_int ... [727](#), [727](#), [730](#),
[732](#), [734](#), [734](#), [16694](#), [16793](#), [16796](#),
[16799](#), [16800](#), [16803](#), [16864](#), [16883](#),
[16886](#), [16887](#), [16890](#), [16939](#), [17075](#)
- \l__sort_true_max_int [727](#),
[728](#), [16694](#), [16708](#), [16721](#), [16735](#),
[16751](#), [16759](#), [16772](#), [16784](#), [17074](#)
- sp [197](#)
- spac commands:
 - \spac_directions_normal_body_dir
..... [1281](#)
 - \spac_directions_normal_page_dir
..... [1282](#)
- \space [55](#)
- \spacefactor [545](#)
- \spaceskip [546](#)
- \span [547](#)
- \special [548](#)
- \splitbotmark [549](#)
- \splitbotmarks [662](#)
- \splitdiscards [663](#)
- \splitfirstmark [550](#)
- \splitfirstmarks [664](#)
- \splitmaxdepth [551](#)
- \splittopskip [552](#)
- sqrt [195](#)
- sr commands:
 - \sr_if_empty_p:N [104](#)
- \SS [18967](#)
- \ss [18967](#)
- str commands:
 - \c_ampersand_str [110](#), [6249](#)
 - \c_atsign_str [110](#), [6249](#)
 - \c_backslash_str [110](#), [6249](#)
 - \c_circumflex_str [110](#), [6249](#)
 - \c_colon_str
... [110](#), [3289](#), [3394](#), [3400](#), [6249](#), [9875](#)
 - \c_dollar_str [110](#), [6249](#)
 - \c_hash_str [110](#),
[6249](#), [6315](#), [20205](#), [20367](#), [20374](#), [20414](#)
 - \c_percent_str . [110](#), [6249](#), [20516](#),
[20517](#), [20518](#), [20529](#), [20530](#), [20531](#)
 - \str_case:nn [104](#), [104](#), [5879](#), [9488](#)
 - \str_case:nn(TF) [339](#)
 - \str_case:nnTF ... [104](#), [104](#), [5879](#),
[5884](#), [5889](#), [5902](#), [5903](#), [10025](#), [18572](#)
 - str_case:nnTF [357](#)
 - \str_case_x:nn [105](#), [5879](#)
 - \str_case_x:nnTF
..... [105](#), [105](#), [5879](#), [5916](#), [5921](#)
 - \str_clear:N [103](#), [103](#), [5772](#)
 - str_clear:N [103](#)
 - \str_clear_new:N [103](#), [103](#), [5772](#)
 - \str_const:Nn [102](#), [102](#),
[5795](#), [6249](#), [6250](#), [6251](#), [6252](#), [6253](#),
[6254](#), [6255](#), [6256](#), [6257](#), [6258](#), [6259](#),
[6260](#), [19219](#), [19223](#), [19245](#), [19253](#),
[19267](#), [19274](#), [19283](#), [19299](#), [19310](#)
 - \str_count:N [105](#), [105](#), [6117](#)
 - \str_count:n . [105](#), [105](#), [105](#), [111](#), [6117](#)
 - \str_count_ignore_spaces:n
..... [105](#), [105](#), [404](#), [6117](#), [11099](#)
 - \str_count_spaces:N [105](#), [6097](#)
 - \str_count_spaces:n
..... [105](#), [105](#), [404](#), [6097](#), [6123](#)
 - \str_fold_case:n [108](#),
[108](#), [109](#), [109](#), [109](#), [109](#), [109](#), [212](#), [6185](#)
 - \str_gclear:N [103](#), [5772](#)
 - \str_gclear_new:N [5772](#)
 - \str_gput_left:Nn [103](#), [5795](#)
 - \str_gput_right:Nn [103](#), [5795](#)
 - \str_gset:Nn [103](#), [5795](#)
 - \str_gset_eq:NN [103](#), [5772](#)
 - \str_head:N [106](#), [106](#), [405](#), [6155](#)
 - \str_head:n ... [106](#), [106](#), [106](#), [374](#),
[390](#), [405](#), [405](#), [5151](#), [5626](#), [5673](#), [6155](#)

- \str_head_ignore_spaces:n [106](#), [106](#), [6155](#)
- \str_if_empty:NTF [104](#), [104](#), [5816](#)
- \str_if_empty_p:N [104](#), [5816](#)
- \str_if_eq:NN [398](#)
- \str_if_eq:nn [132](#), [137](#)
- \str_if_eq:NNTF [104](#), [104](#), [5870](#), [5875](#), [5876](#)
- \str_if_eq:nnTF ... [104](#), [104](#), [104](#), [105](#), [134](#), [209](#), [397](#), [416](#), [5850](#), [5859](#), [5860](#), [5861](#), [5862](#), [5907](#), [6567](#), [8998](#), [9174](#), [9829](#), [9847](#), [10322](#), [10548](#), [12047](#), [18387](#), [18405](#), [18415](#), [18428](#)
- \str_if_eq_p:NN [104](#), [104](#), [5870](#)
- \str_if_eq_p:nn [104](#), [104](#), [5850](#)
- \str_if_eq_x:nn [449](#), [449](#)
- \str_if_eq_x:nnTF [104](#), [104](#), [110](#), [1926](#), [5850](#), [5935](#), [7573](#), [7664](#), [9217](#), [9919](#), [11138](#)
- \str_if_eq_x_p:nn [104](#), [104](#), [5850](#)
- \str_if_exist:NTF [103](#), [103](#), [5816](#)
- \str_if_exist_p:N [103](#), [103](#), [5816](#)
- \str_item:Nn [106](#), [106](#), [5963](#)
- \str_item:nn [106](#), [106](#), [106](#), [400](#), [404](#), [5963](#)
- \str_item_ignore_spaces:nn [106](#), [106](#), [400](#), [5963](#)
- \str_lower_case:n . [108](#), [108](#), [212](#), [6185](#)
- \str_new:N [102](#), [102](#), [103](#), [5772](#), [6261](#), [6262](#), [6263](#), [6264](#)
- \str_put_left:Nn [103](#), [103](#), [5795](#)
- \str_put_right:Nn [103](#), [103](#), [5795](#)
- \str_range:Nnn [107](#), [6021](#)
- \str_range:nnn [107](#), [107](#), [111](#), [404](#), [6021](#)
- \str_range_ignore_spaces:nnn ... [107](#), [6021](#)
- \str_set:Nn [103](#), [103](#), [5795](#)
- \str_set_eq:NN [103](#), [103](#), [5772](#)
- \str_show:N [109](#), [109](#), [6265](#)
- \str_show:n [109](#), [6265](#)
- \str_tail:N [106](#), [106](#), [6170](#)
- \str_tail:n [106](#), [106](#), [106](#), [6170](#)
- \str_tail_ignore_spaces:n [106](#), [106](#), [6170](#)
- \str_upper_case:n . [108](#), [108](#), [212](#), [6185](#)
- \str_use:N [105](#), [105](#), [5772](#)
- \c_tilde_str [110](#), [6249](#)
- \g_tmpa_str [110](#), [6261](#)
- \l_tmpa_str [110](#), [6261](#)
- \g_tmpb_str [110](#), [6261](#)
- \l_tmpb_str [110](#), [6261](#)
- \c_underscore_str [110](#), [6249](#)
- str internal commands:
 - __str_case:nnTF [5879](#)
 - __str_case:nw [5879](#)
 - __str_case_end:nw [5879](#)
 - __str_case_x:nnTF [5879](#)
 - __str_case_x:nw [5879](#)
 - __str_change_case:nn [6185](#)
 - __str_change_case_aux:nn [6185](#)
 - __str_change_case_char:nN ... [6185](#)
 - __str_change_case_char_aux:nN . . [6227](#), [6232](#), [6246](#)
 - __str_change_case_end:nw [6185](#)
 - __str_change_case_end:wn [6204](#), [6221](#)
 - __str_change_case_loop:nw ... [6185](#)
 - __str_change_case_output:nw . [6185](#)
 - __str_change_case_result:n .. [6185](#)
 - __str_change_case_space:n ... [6185](#)
 - __str_collect_delimit_by_q_-stop:w [6049](#), [6072](#)
 - __str_collect_end:nnnnnnnw ... [403](#), [6072](#)
 - __str_collect_end:wn [6072](#)
 - __str_collect_loop:wn [6072](#)
 - __str_collect_loop:wnNNNNNNN . [6072](#)
 - __str_count:n [111](#), [111](#), [404](#), [5979](#), [6036](#), [6117](#)
 - __str_count_aux:n [6117](#)
 - __str_count_loop:NNNNNNNNN .. [6117](#)
 - __str_count_spaces_loop:w ... [6097](#)
 - __str_escape_x:n [5820](#)
 - __str_head:w [405](#), [405](#), [405](#), [6155](#)
 - __str_if_eq_x:nn [110](#), [110](#), [3292](#), [4773](#), [5820](#), [5844](#), [5853](#), [5867](#), [5872](#), [6318](#), [6342](#), [6353](#), [6361](#), [6373](#), [12529](#), [12542](#), [12799](#), [15255](#)
 - __str_if_eq_x_return:nn [110](#), [110](#), [323](#), [3350](#), [5842](#)
 - __str_item:nn [400](#), [400](#), [5963](#)
 - __str_item:w [400](#), [5963](#)
 - __str_lookup_fold:N [6185](#)
 - __str_lookup_lower:N [6185](#)
 - __str_lookup_upper:N [6185](#)
 - __str_range:nnn [111](#), [111](#), [6021](#)
 - __str_range:nnw [6021](#)
 - __str_range:w [6021](#)
 - __str_range_normalize:nn [6044](#), [6045](#), [6053](#)
 - __str_skip_end:NNNNNNNN [401](#), [401](#), [6003](#)
 - __str_skip_end:w [6003](#)
 - __str_skip_exp_end:w [401](#), [403](#), [5990](#), [5999](#), [6003](#), [6051](#)
 - __str_skip_loop:wnNNNNNNNN ... [6003](#)
 - __str_tail_auxi:w [6170](#)
 - __str_tail_auxii:w [406](#), [6170](#)

- `__str_tmp:n` 5773, 5779, 5782, 5796, 5803, 5806
- `__str_to_other:n` 110, 110, 111, 111, 404, 5940, 5970, 6028
- `__str_to_other_end:w` 399, 5940
- `__str_to_other_loop:w` 399, 5940
- `\strcmp` 40
- `\string` 553
- `\suppressfontnotfounderror` 805
- `\suppressifcsnameerror` 922
- `\suppresslongerror` 923
- `\suppressmathparerror` 924
- `\suppressoutererror` 925
- `\synctex` 787
- sys commands:
 - `\c_sys_day_int` 217, 19225
 - `\c_sys_engine_str` 217, 19232
 - `\c_sys_hour_int` 217, 19225
 - `\sys_if_engine luatex:TF` 217, 219, 19232, 19241, 19242, 19322
 - `\sys_if_engine luatex_p:` .. 217, 19232
 - `\sys_if_engine pdftex:TF` 217, 217, 19232, 19249, 19250
 - `\sys_if_engine pdftex_p:` 217, 18780, 19232
 - `\sys_if_engine ptex:TF` 217, 19232, 19270, 19271
 - `\sys_if_engine ptex_p:` ... 217, 19232
 - `\sys_if_engine uptex:TF` 217, 19232, 19263, 19264
 - `\sys_if_engine uptex_p:` 217, 18780, 19232
 - `\sys_if_engine xetex:TF` 217, 19232, 19279, 19280
 - `\sys_if_engine xetex_p:` .. 217, 19232
 - `\sys_if_output_dvi:TF` 218, 218, 19285, 19291, 19292, 19302, 19303
 - `\sys_if_output_dvi_p:` ... 218, 19285
 - `\sys_if_output_pdf:TF` 218, 19285, 19295, 19296, 19306, 19307
 - `\sys_if_output_pdf_p:` ... 218, 19285
 - `\c_sys_jobname_str` .. 173, 217, 19215
 - `\c_sys_minute_int` 217, 19225
 - `\c_sys_month_int` 217, 19225
 - `\c_sys_output_str` 218, 19285
 - `\c_sys_year_int` 217, 19225
- syst commands:
 - `\c_syst_last_allocated_read` 6279, 6280
 - `\c_syst_last_allocated_toks` .. 16765
- T
- `\t` 18976
- `\tabskip` 554
- `\tagcode` 788
- `\tan` 194
- `\tand` 194
- `\tate` 1146
- `\tbaselineshift` 1147
- `\temp` . 163, 169, 174, 177, 178, 185, 190, 193
- TeX and L^AT_EX 2_ε commands:
 - `\@` 6250, 12041
 - `\@@end` 251, 1164, 1165
 - `\@@hyph` 1168
 - `\@@input` 1169
 - `\@@italiccorr` 1170
 - `\@@tracingfonts` 252
 - `\@@underline` 1171
 - `\@addtofilelist` 10705
 - `\@currname` 527, 10564, 10565
 - `\@filelist` 528, 530, 532, 532, 10704, 10731, 10733, 10748
 - `\@firstoftwo` 258
 - `\@ifpackageloaded` 19472, 19509, 19526, 19870, 20260
 - `\@secondoftwo` 258
 - `\@tempa` 143, 145
 - `\@unexpandable@protect` 584, 12042, 12049
 - `\botmark` 324
 - `\box` 139
 - `\char` 60
 - `\chardef` 334
 - `\copy` 139
 - `\count` 60, 729, 729, 729
 - `\cr` 306
 - `\csname` 16
 - `\csstring` 266
 - `\current@color` 19473, 19510, 19871, 20261
 - `\currentgrouplevel` 351, 501, 762
 - `\currentgrouptype` 351, 501, 762
 - `\def` 60
 - `\detokenize` 382
 - `\dimen` 323, 323
 - `\dimendef` 323
 - `\dimexpr` 88
 - `\directlua` 219
 - `\dp` 139, 585, 586
 - `\e@alloc@top` 729, 16751
 - `\edef` 1, 367
 - `\endcsname` 16
 - `\endinput` 153
 - `\endlinechar` 92, 92, 324, 373, 373, 373, 373, 374
 - `\endtemplate` 40, 306
 - `\errhelp` 484, 484
 - `\errmessage` ... 484, 484, 484, 484, 485

- \errorcontextlines . 180, 455, 485, 501
- \escapechar 96, 96, 96, 265, 542
- \everyeof 372, 374
- \expandafter 30, 31
- \expanded 245
- \fi 59
- \firstmark 289, 324
- \frozen@everydisplay 1166
- \frozen@everymath 1167
- \futurelet 306, 326, 328
- \global 234
- \halign 40, 306, 316
- \hskip 85
- \ht 139, 585, 586
- \hyphen 324, 324
- \ifcase 73
- \ifdim 88
- \ifeof 179
- \iffalse 36
- \ifhbox 144
- \ifnum 73
- \ifodd 73, 795
- \iftrue 36
- \ifvbox 144
- \ifvoid 145
- \ifx 21, 230
- \input@path 10644, 10648, 10663
- \italiccorr 324, 324
- \jobname 217
- \l@expl@check@declarations@bool 1763, 2528, 4996, 9408
- \l@expl@log@functions@bool 1700, 1748, 8835
- \lastnamedcs 268
- \lccode 230, 353, 408
- \let 234
- \letcharcode 313
- \loctoks 729, 729, 729
- \long 60, 60
- \lower 751
- \luaescapestring 220
- \m@ne 352, 4437
- \makeatletter 6
- \mathchar 60
- \mathchardef 334
- \meaning 15, 52, 60, 60, 322, 322, 322, 323, 323, 323, 328, 795
- \newif 36
- \newlinechar 92, 92, 180, 269, 373, 373, 373, 373, 485, 501, 540, 540
- \newread 534, 534, 534
- \newtoks 200, 741
- \newwrite 538
- \noexpand 30, 59
- \nullfont 324, 324, 324
- \number 74, 632
- \numexpr 74
- \or 73
- \outer 60, 60, 230, 534, 534, 538, 795, 795
- \par 480
- \pdfmapfile 253
- \pdfmapline 253
- \pdfrandomseed 196
- \pdfsetrandomseed 196
- \pdfstrcmp xii, 227, 227, 228, 230, 230, 244, 800
- \pdfuniformdeviate 196, 721
- \pgfsys@. 222
- \protect 583, 584, 584, 584, 584, 778
- \protected 60, 60
- \protected@edef 544, 11077
- \ProvidesClass 6
- \ProvidesFile 6
- \ProvidesPackage 6
- \randomseed 196
- \read 175
- \readline 176
- \relax 59, 230, 262, 267, 278, 547, 549, 549, 569, 598
- \RequirePackage 6, 230
- \reserveinserts 230, 230
- \robustify 213
- \romannumeral 73
- \scantokens 372, 373, 374
- \set@color 480, 8807, 8810
- \setrandomseed 196
- \sfcode 231
- \show 16, 100, 278
- \showbox 455
- \showthe 278, 352, 360, 364, 367
- \showtokens 101, 159, 500, 501, 501, 501
- \space 324, 324
- \splitbotmark 324
- \splitfirstmark 324
- \strcmp 227, 244
- \string 52
- \synctex 244
- \tenrm 59
- \tex_lowercase:D 316, 316
- \the 59, 64, 80, 84, 87, 281, 281
- \toks xx, 200, 200, 200, 727, 728, 728, 729, 729, 729, 729, 730, 730, 730, 731, 731, 732, 733, 734, 735, 735, 735, 735, 741, 741
- \toksdef 741
- \topmark 60, 324
- \tracingfonts 252
- tracingfonts 252

- `\tracingonline` 455
- `\uccode` 408
- `\Ucharcat` 314, 315, 316
- `\ucharcat@table` 52, 55
- `\unexpanded`
 - 30, 97, 97, 97, 100, 114, 118, 119,
 - 125, 128, 129, 130, 134, 204, 210,
 - 211, 212, 216, 316, 338, 367, 389, 390
- `\unhbox` 142
- `\unhcopy` 142
- `\uniformdeviate` 196, 196, 721
- `\unless` 21
- `\unvbox` 144
- `\unvcopy` 144
- `\valign` 306
- `\vbox` 143
- `\vskip` 85
- `\vsplit` 144
- `\vtop` 143, 461
- `\wd` 140, 585, 586
- `\write` 177, 540
- `\zap@space` 524
- tex commands:
 - `\tex_above:D` 287
 - `\tex_abovedisplayshortskip:D` .. 288
 - `\tex_abovedisplayskip:D` 289
 - `\tex_abovewithdelims:D` 290
 - `\tex_accent:D` 291
 - `\tex_adjdemerits:D` 292
 - `\tex_advance:D`
 - 293, 3789, 3791, 3801, 3803,
 - 4509, 4514, 4761, 4766, 4855, 4860,
 - 16800, 16875, 16882, 16885, 16892,
 - 16893, 16894, 16902, 16903, 16937,
 - 16938, 16947, 16948, 16957, 16958
 - `\tex_afterassignment:D` 294, 3456
 - `\tex_aftergroup:D` 295, 1324
 - `\tex_atop:D` 296
 - `\tex_atopwithdelims:D` 297
 - `\tex_badness:D` 298
 - `\tex_baselineskip:D` 299
 - `\tex_batchmode:D` 300
 - `\tex_begingroup:D` 301, 1320
 - `\tex_belowdisplayshortskip:D` .. 302
 - `\tex_belowdisplayskip:D` 303
 - `\tex_binoppenalty:D` 304
 - `\tex_botmark:D` 305
 - `\tex_box:D` 306, 7772, 7796
 - `\tex_boxmaxdepth:D` 307
 - `\tex_brokenpenalty:D` 308
 - `\tex_catcode:D` 309, 2301,
 - 2305, 2906, 2910, 5102, 6289, 6290
 - `\tex_char:D` 310
 - `\tex_chardef:D`
 - 257, 311, 1314, 1341, 1343,
 - 1587, 1588, 2522, 2524, 2562, 2567,
 - 3379, 3762, 6271, 6274, 10827, 10944
 - `\tex_cleaders:D` 312
 - `\tex_closein:D` 313, 6294, 10835
 - `\tex_closeout:D` 314, 10952
 - `\tex_clubpenalty:D` 315
 - `\tex_copy:D` 316, 7766, 7797
 - `\tex_count:D`
 - .. 317, 6281, 10767, 10769, 10906,
 - 10908, 16734, 16750, 16758, 16759
 - `\tex_countdef:D` 318
 - `\tex_cr:D` 319
 - `\tex_crcrcr:D` 320
 - `\tex_csname:D` 321, 1308
 - `\tex_day:D` 322, 19229
 - `\tex_deadcycles:D` 323
 - `\tex_def:D`
 - .. 324, 791, 792, 793, 1325, 1327,
 - 1329, 1330, 1347, 1349, 1350, 1351,
 - 1353, 1354, 1355, 1357, 1358, 1359
 - `\tex_defaultthyphenchar:D` 325
 - `\tex_defaultskewchar:D` 326
 - `\tex_delcode:D` 327
 - `\tex_delimiter:D` 328
 - `\tex_delimiterfactor:D` 329
 - `\tex_delimitershortfall:D` 330
 - `\tex_dimen:D` 331
 - `\tex_dimendef:D` 332
 - `\tex_discretionary:D` 333
 - `\tex_displayindent:D` 334
 - `\tex_displaylimits:D` 335
 - `\tex_displaystyle:D` 336
 - `\tex_displaywidowpenalty:D` 337
 - `\tex_displaywidth:D` 338
 - `\tex_divide:D` 339, 16727
 - `\tex_doublehyphendemerits:D` ... 340
 - `\tex_dp:D` 341, 7782
 - `\tex_dump:D` 342
 - `\tex_edef:D` 343, 1348, 1352, 1356, 1360
 - `\tex_else:D` 344, 1175, 1295, 1344
 - `\tex_emergencystretch:D` 345
 - `\tex_end:D` . 346, 1165, 1264, 1731, 9049
 - `\tex_endcsname:D` 347, 1309
 - `\tex_endgroup:D` 348, 1162, 1321
 - `\tex_endinput:D` 349, 9060
 - `\tex_endlinechar:D`
 - 251, 252, 266, 350,
 - 5075, 5076, 5077, 5133, 10875, 10877
 - `\tex_eqno:D` 351
 - `\tex_errhelp:D` 352, 8930
 - `\tex_errmessage:D` 353, 1723, 8961

- `\tex_errorcontextlines:D`
..... 354, 7867, 8956, 8988, 9599
- `\tex_errorstopmode:D` 355
- `\tex_escapechar:D`
..... 356, 11007, 11038, 11044
- `\tex_everycr:D` 357
- `\tex_everydisplay:D` 358, 1166
- `\tex_everyhbox:D` 359
- `\tex_everyjob:D` . 360, 1265, 10557,
10559, 10570, 10572, 19216, 19218
- `\tex_everymath:D` 361, 1167
- `\tex_everypar:D` 362
- `\tex_everyvbox:D` 363
- `\tex_exhyphenpenalty:D` 364
- `\tex_expandafter:D` ... 365, 796, 1310
- `\tex_fam:D` 366
- `\tex_fi:D` 367, 797, 1179, 1180, 1181,
1234, 1236, 1237, 1241, 1255, 1260,
1275, 1283, 1288, 1296, 1346, 5054
- `\tex_finalhyphendemerits:D` 368
- `\tex_firstmark:D` 369
- `\tex_floatingpenalty:D` 370
- `\tex_font:D` 371
- `\tex_fontdimen:D` 372
- `\tex_fontname:D` 373
- `\tex_futurelet:D` 374, 3450, 3452
- `\tex_gdef:D` 375, 1361, 1364, 1368, 1372
- `\tex_global:D` 272,
277, 279, 283, 376, 798, 800, 1830,
1837, 2524, 2567, 3185, 3187, 3197,
3452, 3749, 3769, 3781, 3793, 3795,
3805, 3807, 3814, 4484, 4499, 4505,
4510, 4515, 4736, 4751, 4757, 4762,
4767, 4830, 4845, 4851, 4856, 4861,
7768, 7774, 7832, 7879, 7885, 7891,
7915, 7921, 7930, 7936, 10827, 10944
- `\tex_globaldefs:D` 377
- `\tex_halign:D` 378
- `\tex_hangafter:D` 379
- `\tex_hangindent:D` 380
- `\tex_hbadness:D` 381
- `\tex_hbox:D`
..... 382, 7876, 7878, 7883, 7889, 7897, 7898
- `\tex_hfil:D` 383
- `\tex_hfill:D` 384
- `\tex_hfilneg:D` 385
- `\tex_hfuzz:D` 386
- `\tex_hoffset:D` 387, 1277
- `\tex_holdinginserts:D` 388
- `\tex_hrule:D` 389
- `\tex_hsize:D`
..... 390, 8053, 8055, 8056, 8102, 8104, 8105
- `\tex_hskip:D` 391, 4796
- `\tex_hss:D` 392, 7900, 7902, 17393, 17402
- `\tex_ht:D` 393, 7781
- `\tex_hyphen:D` 286, 1168
- `\tex_hyphenation:D` 394
- `\tex_hyphenchar:D` 395
- `\tex_hyphenpenalty:D` 396
- `\tex_if:D` 47, 397, 1298, 1299
- `\tex_ifcase:D` 398, 3653
- `\tex_ifcat:D` 399, 1300
- `\tex_ifdim:D` 400, 4466
- `\tex_ifeof:D` 401, 6299, 10852
- `\tex_iffalse:D` 402, 1293
- `\tex_ifhbox:D` 403, 7808
- `\tex_ifhmode:D` 404, 1303
- `\tex_ifinner:D` 405, 1305
- `\tex_ifmmode:D` 406, 1302
- `\tex_ifnum:D` 407, 1235, 1322
- `\tex_ifodd:D` 408, 1700,
1748, 1763, 2496, 2497, 3652, 4996
- `\tex_iftrue:D` 409, 1292
- `\tex_ifvbox:D` 410, 7809
- `\tex_ifvmode:D` 411, 1304
- `\tex_ifvoid:D` 412, 7810
- `\tex_ifx:D` 413, 1301
- `\tex_ignorespaces:D` 414
- `\tex_immediate:D`
..... 415, 1691, 1693, 10946, 10952, 10984
- `\tex_indent:D` 416
- `\tex_input:D`
..... 417, 1169, 1266, 10710, 18147, 18170
- `\tex_inputlineno:D` ... 418, 1738, 8904
- `\tex_insert:D` 419
- `\tex_insertpenalties:D` 420
- `\tex_interlinepenalty:D` 421
- `\tex_italiccorrection:D`
..... 285, 1170, 1278
- `\tex_jobname:D` 422, 10560, 19219, 19223
- `\tex_kern:D`
..... 423, 8370, 8375, 8441, 8442, 8729,
8730, 17159, 17391, 17400, 17413,
17415, 17459, 17461, 17546, 19778
- `\tex_language:D` 424, 1267
- `\tex_lastbox:D` 425, 7830
- `\tex_lastkern:D` 426
- `\tex_lastpenalty:D` 427
- `\tex_lastskip:D` 428
- `\tex_lccode:D` 429, 2988,
2992, 5946, 5947, 6241, 6354, 18352
- `\tex_leaders:D` 430
- `\tex_left:D` 431, 1285
- `\tex_lefthyphenmin:D` 432
- `\tex_leftskip:D` 433
- `\tex_leqno:D` 434
- `\tex_let:D` . 273, 277, 279, 435, 798,
800, 1165, 1166, 1167, 1168, 1169,

1170, 1171, 1172, 1174, 1178, 1183, 1184, 1185, 1186, 1187, 1188, 1189, 1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1200, 1201, 1202, 1203, 1204, 1205, 1206, 1207, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 1230, 1231, 1232, 1233, 1239, 1240, 1245, 1246, 1247, 1248, 1249, 1250, 1251, 1252, 1253, 1254, 1257, 1258, 1259, 1264, 1265, 1266, 1267, 1268, 1269, 1270, 1271, 1272, 1273, 1274, 1277, 1278, 1279, 1280, 1281, 1282, 1285, 1286, 1287, 1292, 1293, 1294, 1295, 1296, 1297, 1298, 1299, 1300, 1301, 1302, 1303, 1304, 1305, 1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1315, 1316, 1317, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1340, 1347, 1348, 1361, 1362, 1826, 3185, 3187, 3197	<code>\tex_medmuskip:D</code> 463
<code>\tex_limits:D</code> 436	<code>\tex_message:D</code> 464
<code>\tex_linepenalty:D</code> 437	<code>\tex_middle:D</code> 1286
<code>\tex_lineskip:D</code> 438	<code>\tex_mkern:D</code> 465
<code>\tex_lineskiplimit:D</code> 439	<code>\tex_month:D</code> 466, 1269, 19230
<code>\tex_long:D</code> 440, 791, 792, 793, 1325, 1327, 1330, 1349, 1350, 1351, 1352, 1353, 1355, 1357, 1358, 1359, 1360, 1364, 1366, 1372, 1374	<code>\tex_moveleft:D</code> 467, 7801
<code>\tex_looseness:D</code> 441	<code>\tex_moveright:D</code> 468, 7803
<code>\tex_lower:D</code> 442, 7807	<code>\tex_mskip:D</code> 469
<code>\tex_lowercase:D</code> 443, 3027, 3150, 5109, 5126, 5766, 5948, 8942	<code>\tex_multiply:D</code> 470
<code>\tex_mag:D</code> 444	<code>\tex_muskip:D</code> 471
<code>\tex_mark:D</code> 445	<code>\tex_muskipdef:D</code> 472
<code>\tex_mathaccent:D</code> 446	<code>\tex_newlinechar:D</code> 473, 1722, 5077, 5105, 5108, 8954, 9597, 10983
<code>\tex_mathbin:D</code> 447	<code>\tex_noalign:D</code> 474
<code>\tex_mathchar:D</code> 448	<code>\tex_noboundary:D</code> 475
<code>\tex_mathchardef:D</code> 257, 449, 1345, 3765, 3766	<code>\tex_noexpand:D</code> 476, 1311
<code>\tex_mathchoice:D</code> 450	<code>\tex_noindent:D</code> 477
<code>\tex_mathclose:D</code> 451	<code>\tex_nolimits:D</code> 478
<code>\tex_mathcode:D</code> 452, 2979, 2983	<code>\tex_nonscript:D</code> 479
<code>\tex_mathinner:D</code> 453	<code>\tex_nonstopmode:D</code> 480
<code>\tex_mathop:D</code> 454, 1268	<code>\tex_nulldelimiterspace:D</code> 481
<code>\tex_mathopen:D</code> 455	<code>\tex_nullfont:D</code> 482, 3408
<code>\tex_mathord:D</code> 456	<code>\tex_number:D</code> 483, 3649
<code>\tex_mathpunct:D</code> 457	<code>\tex_omit:D</code> 484
<code>\tex_mathrel:D</code> 458	<code>\tex_openin:D</code> 485, 6291, 10829
<code>\tex_mathsurround:D</code> 459	<code>\tex_openout:D</code> 486, 10946
<code>\tex_maxdeadcycles:D</code> 460	<code>\tex_or:D</code> 487, 1294
<code>\tex_maxdepth:D</code> 461	<code>\tex_outer:D</code> 488, 1270
<code>\tex_meaning:D</code> 462, 1315, 1316	<code>\tex_output:D</code> 489
	<code>\tex_outputpenalty:D</code> 490
	<code>\tex_over:D</code> 491, 1271
	<code>\tex_overfullrule:D</code> 492
	<code>\tex_overline:D</code> 493
	<code>\tex_overwithdelims:D</code> 494
	<code>\tex_pagedepth:D</code> 495
	<code>\tex_pagefilllstretch:D</code> 496
	<code>\tex_pagefillstretch:D</code> 497
	<code>\tex_pagefilstretch:D</code> 498
	<code>\tex_pagegoal:D</code> 499
	<code>\tex_pageshrink:D</code> 500
	<code>\tex_pagestretch:D</code> 501
	<code>\tex_pagetotal:D</code> 502
	<code>\tex_par:D</code> 503, 8790
	<code>\tex_parfillskip:D</code> 504
	<code>\tex_parindent:D</code> 505
	<code>\tex_parshape:D</code> 506
	<code>\tex_parskip:D</code> 507
	<code>\tex_patterns:D</code> 508
	<code>\tex_pausing:D</code> 509
	<code>\tex_penalty:D</code> 510
	<code>\tex_postdisplaypenalty:D</code> 511
	<code>\tex_predisplaypenalty:D</code> 512
	<code>\tex_predisplaysize:D</code> 513
	<code>\tex_pretolerance:D</code> 514

<code>\tex_prevdepth:D</code>	515	<code>\tex_the:D</code>	252, 282, 557, 580, 585, 585, 586, 732, 1738, 2021, 2088, 2092, 2910, 2983, 2992, 3001, 3010, 3817, 3819, 4414, 4681, 4683, 4793, 4795, 4866, 7857, 10559, 10572, 12078, 12113, 12542, 16865, 16912, 16913, 16927, 16928, 19218
<code>\tex_prevgraf:D</code>	516	<code>\tex_thickmuskip:D</code>	558
<code>\tex_radical:D</code>	517	<code>\tex_thinmuskip:D</code>	559
<code>\tex_raise:D</code>	518, 7805	<code>\tex_time:D</code>	560, 19226, 19228
<code>\tex_read:D</code>	519, 6305, 10870	<code>\tex_toks:D</code>	561, 732, 16799, 16865, 16901, 16912, 16913, 16927, 16928, 16936, 16946, 16956
<code>\tex_relax:D</code>	520, 550, 1319, 1722, 3651, 4468	<code>\tex_toksdef:D</code>	562, 17059
<code>\tex_relpenalty:D</code>	521	<code>\tex_tolerance:D</code>	563
<code>\tex_right:D</code>	522, 1287	<code>\tex_topmark:D</code>	564
<code>\tex_righthyphenmin:D</code>	523	<code>\tex_topskip:D</code>	565
<code>\tex_rightskip:D</code>	524	<code>\tex_tracingcommands:D</code>	566
<code>\tex_romannumeral:D</code>	265, 265, 265, 265, 265, 288, 525, 1313, 1323, 1591, 2299	<code>\tex_tracinglostchars:D</code>	567
<code>\tex_romannumerl:D</code>	288	<code>\tex_tracingmacros:D</code>	568
<code>\tex_scriptfont:D</code>	526	<code>\tex_tracingonline:D</code>	569, 7866
<code>\tex_scriptscriptfont:D</code>	527	<code>\tex_tracingoutput:D</code>	570
<code>\tex_scriptscriptstyle:D</code>	528	<code>\tex_tracingpages:D</code>	571
<code>\tex_scriptspace:D</code>	529	<code>\tex_tracingparagraphs:D</code>	572
<code>\tex_scriptstyle:D</code>	530	<code>\tex_tracingrestores:D</code>	573
<code>\tex_scrollmode:D</code>	531	<code>\tex_tracingstats:D</code>	574
<code>\tex_setbox:D</code>	532, 7766, 7772, 7830, 7878, 7883, 7889, 7914, 7919, 7926, 7934, 7950	<code>\tex_uccode:D</code>	575, 2997, 3001, 6242, 18353
<code>\tex_setlanguage:D</code>	533	<code>\tex_uchyph:D</code>	576
<code>\tex_sfcode:D</code>	534, 3006, 3010	<code>\tex_undefined:D</code>	272, 279, 324, 324, 324, 800, 1172, 1239, 1240, 1257, 1258, 1259, 1843, 1851, 9973, 9987, 10020, 10041, 11508, 11519, 11520, 11521, 11522, 16737
<code>\tex_shipout:D</code>	535	<code>\tex_underline:D</code>	577, 1171
<code>\tex_show:D</code>	536	<code>\tex_unhbox:D</code>	578, 7904
<code>\tex_showbox:D</code>	537, 7869	<code>\tex_unhcopy:D</code>	579, 7903
<code>\tex_showboxbreadth:D</code>	538, 7864	<code>\tex_unkern:D</code>	580
<code>\tex_showboxdepth:D</code>	539, 7865	<code>\tex_unpenalty:D</code>	581
<code>\tex_showlists:D</code>	540	<code>\tex_unskip:D</code>	582
<code>\tex_showthe:D</code>	541	<code>\tex_unvbox:D</code>	583, 7946
<code>\tex_skewchar:D</code>	542	<code>\tex_unvcopy:D</code>	584, 7945
<code>\tex_skip:D</code>	543	<code>\tex_uppercase:D</code>	585, 5768
<code>\tex_skipdef:D</code>	544	<code>\tex_vadjust:D</code>	586
<code>\tex_space:D</code>	284	<code>\tex_valign:D</code>	587
<code>\tex_spacefactor:D</code>	545	<code>\tex_vbadness:D</code>	588
<code>\tex_spaceskip:D</code>	546	<code>\tex_vbox:D</code>	589, 7907, 7910, 7912, 7914, 7926, 7934
<code>\tex_span:D</code>	547	<code>\tex_vcenter:D</code>	590, 1272
<code>\tex_special:D</code>	548, 19497, 19516, 19520, 19530, 19534, 19795, 19805, 19807, 19877, 19881, 19883, 19887, 19888, 19889, 19890, 19894, 19895, 19896, 20147, 20149, 20150, 20151, 20158, 20164, 20267, 20271	<code>\tex_vfil:D</code>	591
<code>\tex_splitbotmark:D</code>	549	<code>\tex_vfill:D</code>	592
<code>\tex_splitfirstmark:D</code>	550	<code>\tex_vfilneg:D</code>	593
<code>\tex_splitmaxdepth:D</code>	551	<code>\tex_vfuzz:D</code>	594
<code>\tex_splittopskip:D</code>	552	<code>\tex_voffset:D</code>	595, 1279
<code>\tex_string:D</code>	553, 1318		
<code>\tex_tabskip:D</code>	554		
<code>\tex_textfont:D</code>	555		
<code>\tex_textstyle:D</code>	556		

- `\tex_vrule:D` 596, 8581, 8636
- `\tex_vsize:D` 597
- `\tex_vskip:D` 598, 4799
- `\tex_vsplit:D` 599, 7950
- `\tex_vss:D` 600
- `\tex_vtop:D` 601, 7908, 7919
- `\tex_wd:D` 602, 7783
- `\tex_widowpenalty:D` 603
- `\tex_write:D`
 - . 604, 1691, 1693, 10964, 10967, 10984
- `\tex_xdef:D` 605, 1362, 1366, 1370, 1374
- `\tex_xleaders:D` 606
- `\tex_xspaceskip:D` 607
- `\tex_year:D` 608, 19231
- `\textdir` 944
- `\textfont` 555
- `\textstyle` 556
- `\texttt` 13384, 19850
- `\TeXeTstate` 665
- `\tfont` 1148
- `\TH` 18968
- `\th` 18968
- `\the` 67, 212, 213, 214,
 - 215, 216, 217, 218, 219, 220, 221, 557
- `\thickmuskip` 558
- `\thinmuskip` 559
- thousand commands:
 - `\c_one_thousand` 72, 4445
 - `\c_ten_thousand` 72, 4445,
 - 14336, 14367, 14368, 14445, 14505
- `\time` 560
- `\tiny` 8574
- tl commands:
 - `\c_empty_tl`
 - . 101, 345, 4145, 4151, 4884, 4900,
 - 4902, 4927, 5253, 6306, 6933, 9710
 - `\c_space_tl`
 - 101, 4928, 5548, 6213, 7347,
 - 7356, 8908, 10710, 11070, 11084,
 - 11155, 16469, 18229, 18289, 19015,
 - 19503, 19566, 19571, 19574, 20343
 - `\tl_act` 387
 - `\tl_case:Nn` 94, 94, 5345
 - `tl_case:nn` 398
 - `\tl_case:NnTF`
 - . 94, 94, 5345, 5350, 5355, 5374, 5375
 - `\l_tl_case_change_accents_tl` ...
 - 214, 18396, 18974
 - `\l_tl_case_change_exclude_tl` ...
 - 213, 213, 214, 18418, 19148
 - `\l_tl_case_change_math_tl`
 - 213, 213, 18239, 19025, 19143
- `\tl_clear:N` 89, 89,
 - 4899, 4906, 6941, 6942, 9589, 9825,
 - 10279, 10351, 11059, 11065, 11148
- `tl_clear:N` 90
- `\tl_clear_new:N` 90, 90, 4905, 6945, 6946
- `\tl_concat:NNN` 90, 90, 4919, 5040
- `\tl_const:Nn` .. 89, 89, 2794, 3152,
 - 3203, 4887, 4927, 4928, 5056, 6326,
 - 6344, 6356, 6391, 6392, 6393, 6412,
 - 6939, 7477, 8819, 8820, 8872, 8877,
 - 8879, 8881, 8883, 8885, 8890, 8891,
 - 8898, 9762, 9763, 9764, 9765, 9766,
 - 9767, 9768, 11005, 11008, 11014,
 - 11223, 11224, 11225, 11226, 11227,
 - 14330, 14769, 14770, 14771, 14772,
 - 14773, 14774, 14775, 14776, 14777,
 - 16647, 18725, 18726, 18727, 18736,
 - 18737, 18740, 18741, 18742, 18743,
 - 18744, 18750, 18758, 18796, 18809,
 - 18930, 18953, 18954, 18971, 18972
 - `\tl_count:N` 93, 96, 97, 97, 5437, 11086
 - `\tl_count:n` 94, 96, 96, 97, 274, 333,
 - 404, 404, 560, 1445, 1449, 1888,
 - 1933, 5437, 5741, 13165, 13186, 19170
 - `\tl_count_tokens:n` .. 212, 212, 18115
 - `\tl_gclear:N` 89,
 - 4899, 4908, 6943, 6944, 20381, 20417
 - `\tl_gclear_new:N` . 90, 4905, 6947, 6948
 - `\tl_gconcat:NNN` 90, 4919, 5047
 - `\tl_gput_left:Nn` . 90, 4947, 5017, 5018
 - `\tl_gput_right:Nn`
 - 90, 2894, 4971, 5021, 5022, 6536
 - `\tl_gremove_all:Nn` 91, 5235
 - `\tl_gremove_once:Nn` 91, 5229
 - `\tl_greplace_all:Nnn` . 91, 5160, 5238
 - `\tl_greplace_once:Nnn` 91, 5160, 5232
 - `\tl_greverse:N` 97, 5578
 - `.tl_gset:N` 165, 10182
 - `\tl_gset:Nn` 90, 113,
 - 375, 731, 731, 4922, 4929, 5014,
 - 5052, 5060, 5163, 5167, 5471, 5581,
 - 6451, 6456, 6468, 6505, 6522, 6562,
 - 6588, 6672, 6701, 6738, 6742, 6960,
 - 6988, 7034, 7077, 7115, 7167, 7198,
 - 7528, 7557, 7593, 7605, 7628, 10560,
 - 10709, 16645, 16822, 16836, 16844,
 - 18028, 18038, 18134, 18159, 20340
 - `\tl_gset_eq:NN` 90, 4902,
 - 4911, 5032, 5792, 6435, 6436, 6437,
 - 6438, 6953, 6954, 6955, 6956, 7500,
 - 7501, 7502, 7503, 10565, 10712, 16652
 - `\tl_gset_from_file:Nnn` ... 215, 18131
 - `\tl_gset_from_file_x:Nnn` . 215, 18156
 - `\tl_gset_rescan:Nnn` 92, 5057

- `.tl_gset_x:N` [165](#), [10182](#)
- `\tl_gsort:Nn` [98](#), [5512](#), [16827](#)
- `\tl_gtrim_spaces:N` [97](#), [5466](#)
- `\tl_head:N` [98](#), [5584](#)
- `\tl_head:n` [98](#), [98](#), [98](#), [99](#), [389](#), [390](#), [5584](#)
- `\tl_head:w` [99](#), [99](#),
[390](#), [390](#), [391](#), [5584](#), [5624](#), [5643](#), [5665](#)
- `\tl_if_blank:nTF` ... [92](#), [92](#), [98](#), [99](#),
[99](#), [5241](#), [5244](#), [5245](#), [5248](#), [5249](#),
[5611](#), [5775](#), [5798](#), [6330](#), [6340](#), [7355](#),
[7434](#), [9823](#), [10275](#), [10290](#), [10429](#),
[16968](#), [18577](#), [18597](#), [18647](#), [19169](#)
- `\tl_if_blank_p:n` [92](#), [92](#), [5241](#)
- `\tl_if_empty:N` [5818](#), [5819](#), [7213](#), [7215](#)
- `\tl_if_empty:nTF` [93](#), [93](#), [160](#), [5251](#),
[5260](#), [5261](#), [9809](#), [9833](#), [10295](#), [20343](#)
- `\tl_if_empty:nTF` [93](#),
[93](#), [380](#), [381](#), [382](#), [435](#), [1474](#), [1556](#),
[2817](#), [2824](#), [2841](#), [3396](#), [3560](#), [5065](#),
[5174](#), [5263](#), [5274](#), [5275](#), [5276](#), [5324](#),
[5704](#), [5725](#), [6380](#), [6471](#), [6976](#), [7020](#),
[7226](#), [7243](#), [7281](#), [8913](#), [9184](#), [9199](#),
[9319](#), [9323](#), [9466](#), [9467](#), [9474](#), [9481](#),
[9487](#), [9494](#), [9569](#), [9877](#), [9969](#), [11829](#),
[16453](#), [16543](#), [18086](#), [19205](#), [20437](#)
- `\tl_if_empty_p:N` [93](#), [93](#), [5251](#)
- `\tl_if_empty_p:n` ... [93](#), [93](#), [5263](#), [5276](#)
- `\tl_if_eq:NN` [398](#)
- `\tl_if_eq:nn(TF)` [125](#), [125](#)
- `\tl_if_eq:NNTF` [42](#), [93](#), [93](#), [94](#),
[416](#), [5287](#), [5297](#), [5298](#), [5369](#), [6574](#),
[7648](#), [8647](#), [8650](#), [9165](#), [9219](#), [11092](#)
- `\tl_if_eq:nnTF`
..... [93](#), [93](#), [116](#), [116](#), [416](#), [5299](#)
- `\tl_if_eq_p:NN` [93](#), [93](#), [5287](#)
- `\tl_if_exist:N` [5816](#), [5817](#)
- `\tl_if_exist:nTF`
..... [90](#), [90](#), [4906](#), [4908](#), [4925](#), [5430](#), [5755](#)
- `\tl_if_exist_p:N` [90](#), [90](#), [4925](#)
- `\tl_if_head_eq_catcode:nN` [391](#)
- `\tl_if_head_eq_catcode:nNTF`
..... [99](#), [99](#), [5617](#), [18175](#), [18175](#)
- `\tl_if_head_eq_catcode_p:nN`
..... [99](#), [99](#), [5617](#)
- `\tl_if_head_eq_charcode:nN` [390](#)
- `\tl_if_head_eq_charcode:nNTF` ...
..... [99](#), [99](#), [5617](#), [5634](#), [5635](#)
- `\tl_if_head_eq_charcode_p:nN` ...
..... [99](#), [99](#), [5617](#)
- `\tl_if_head_eq_meaning:nN` [391](#)
- `\tl_if_head_eq_meaning:nNTF`
..... [99](#), [99](#), [5617](#)
- `\tl_if_head_eq_meaning_p:nN`
..... [99](#), [99](#), [5617](#)
- `\tl_if_head_is_group:nTF`
..... [100](#), [100](#), [5524](#),
[5646](#), [5681](#), [5707](#), [18202](#), [18266](#), [18997](#)
- `\tl_if_head_is_group_p:n`
..... [100](#), [100](#), [5707](#)
- `\tl_if_head_is_N_type:n` [391](#)
- `\tl_if_head_is_N_type:nTF`
..... [100](#), [100](#), [5521](#), [5621](#), [5640](#), [5657](#),
[5690](#), [18083](#), [18199](#), [18263](#), [18479](#),
[18511](#), [18606](#), [18656](#), [18994](#), [19114](#)
- `\tl_if_head_is_N_type_p:n`
..... [100](#), [100](#), [5690](#)
- `\tl_if_head_is_space:nTF`
..... [100](#), [100](#), [5718](#), [6207](#)
- `\tl_if_head_is_space_p:n`
..... [100](#), [100](#), [5718](#)
- `\tl_if_in:Nn` [436](#)
- `\tl_if_in:NnTF` [93](#), [93](#), [2888](#), [5196](#),
[5314](#), [5314](#), [5315](#), [5317](#), [5318](#), [10608](#)
- `\tl_if_in:nnTF` .. [93](#), [93](#), [381](#), [381](#),
[5116](#), [5180](#), [5182](#), [5314](#), [5315](#), [5316](#),
[5320](#), [5328](#), [5329](#), [8456](#), [10693](#), [19203](#)
- `\tl_if_single:n` [381](#)
- `\tl_if_single:nTF`
..... [93](#), [93](#), [5331](#), [5332](#), [5333](#)
- `\tl_if_single:nTF`
..... [94](#), [94](#), [5332](#), [5333](#), [5334](#), [5335](#), [9588](#)
- `\tl_if_single_p:N` [93](#), [93](#), [5331](#)
- `\tl_if_single_p:n` .. [94](#), [94](#), [5331](#), [5335](#)
- `\tl_if_single_token:nTF`
..... [212](#), [212](#), [18081](#)
- `\tl_if_single_token_p:n`
..... [212](#), [212](#), [18081](#)
- `\tl_item:Nn` [100](#), [5730](#)
- `\tl_item:nn` . [100](#), [100](#), [795](#), [5730](#), [19170](#)
- `\tl_log:N` [215](#), [215](#), [19162](#)
- `\tl_log:n` [215](#), [215](#), [19165](#)
- `\tl_lower_case:n` [212](#), [18176](#)
- `tl_lower_case:n` [108](#)
- `\tl_lower_case:nn` [212](#), [18176](#)
- `\tl_map_break:` [95](#),
[95](#), [5384](#), [5390](#), [5401](#), [5410](#), [5417](#), [5422](#)
- `\tl_map_break:n`
..... [95](#), [95](#), [95](#), [5422](#), [16830](#), [16837](#)
- `\tl_map_function:NN`
..... [94](#), [94](#), [94](#), [94](#), [5380](#), [5445](#), [10600](#)
- `\tl_map_function:nN`
[94](#), [94](#), [94](#), [94](#), [1927](#), [5380](#), [5440](#), [6474](#)
- `\tl_map_inline:Nn`
.. [94](#), [94](#), [95](#), [731](#), [5394](#), [16830](#), [16837](#)
- `\tl_map_inline:nn` [44](#), [94](#), [94](#),
[95](#), [5394](#), [11011](#), [12663](#), [12715](#), [16765](#)
- `\tl_map_variable:NNn` ... [95](#), [95](#), [5406](#)
- `\tl_map_variable:nNn` [95](#), [95](#), [384](#), [5406](#)

- \tl_mixed_case:n [212](#), [18176](#)
- tl_mixed_case:n [108](#), [214](#)
- \tl_mixed_case:nn ... [212](#), [773](#), [18176](#)
- \l_tl_mixed_case_ignore_tl
..... [214](#), [19070](#), [19153](#)
- \l_tl_mixed_change_ignore_tl .. [214](#)
- \tl_new:N [52](#), [89](#), [89](#), [90](#), [369](#),
[2885](#), [3085](#), [3444](#), [4881](#), [4906](#), [4908](#),
[5312](#), [5313](#), [5761](#), [5762](#), [5763](#), [5764](#),
[6409](#), [6410](#), [6934](#), [6936](#), [6937](#), [7476](#),
[7956](#), [7979](#), [7980](#), [8569](#), [8818](#), [9112](#),
[9113](#), [9416](#), [9625](#), [9626](#), [9627](#), [9770](#),
[9772](#), [9773](#), [9776](#), [9777](#), [9781](#), [9782](#),
[10555](#), [10576](#), [10577](#), [10759](#), [10898](#),
[10999](#), [11000](#), [11001](#), [11002](#), [11003](#),
[17812](#), [18974](#), [19143](#), [19148](#), [19153](#),
[19467](#), [19504](#), [19865](#), [20255](#), [20347](#)
- \tl_put_left:Nn [90](#), [90](#), [4947](#), [5015](#), [5016](#)
- \tl_put_right:Nn [90](#), [90](#), [3117](#), [3119](#),
[3122](#), [3124](#), [3130](#), [3132](#), [3134](#), [3136](#),
[3137](#), [3139](#), [3141](#), [3143](#), [3145](#), [4971](#),
[5019](#), [5020](#), [6534](#), [10293](#), [11111](#),
[11117](#), [11125](#), [11145](#), [11154](#), [11165](#)
- \tl_rand_item:N [216](#), [216](#), [19167](#)
- \tl_rand_item:n [216](#), [216](#), [19167](#)
- \tl_remove_all:Nn
..... [91](#), [91](#), [91](#), [91](#), [5235](#), [10607](#)
- \tl_remove_once:Nn [91](#), [91](#), [5229](#)
- \tl_replace_all:Nnn [91](#),
[91](#), [414](#), [433](#), [5160](#), [5236](#), [6483](#), [11084](#)
- \tl_replace_once:Nnn
..... [91](#), [91](#), [3160](#), [5160](#), [5230](#)
- \tl_rescan:nn . [92](#), [92](#), [92](#), [92](#), [92](#), [5057](#)
- \tl_reverse:N [97](#), [97](#), [97](#), [5578](#)
- \tl_reverse:n
. [97](#), [97](#), [97](#), [97](#), [770](#), [5558](#), [5579](#), [5581](#)
- \tl_reverse_items:n [97](#), [97](#), [97](#), [97](#), [5450](#)
- \tl_reverse_tokens:n
..... [212](#), [212](#), [212](#), [18091](#)
- .tl_set:N [165](#), [10182](#)
- \tl_set:Nn [90](#), [90](#), [91](#), [92](#),
[113](#), [165](#), [289](#), [369](#), [375](#), [490](#), [731](#),
[731](#), [3115](#), [3120](#), [3462](#), [3483](#), [4042](#),
[4920](#), [4929](#), [5013](#), [5045](#), [5058](#), [5087](#),
[5146](#), [5161](#), [5165](#), [5302](#), [5303](#), [5416](#),
[5469](#), [5579](#), [6441](#), [6446](#), [6466](#), [6473](#),
[6477](#), [6488](#), [6503](#), [6514](#), [6560](#), [6570](#),
[6579](#), [6586](#), [6619](#), [6622](#), [6643](#), [6651](#),
[6660](#), [6670](#), [6679](#), [6689](#), [6699](#), [6713](#),
[6736](#), [6740](#), [6832](#), [6958](#), [6986](#), [7032](#),
[7066](#), [7072](#), [7075](#), [7081](#), [7088](#), [7113](#),
[7165](#), [7196](#), [7323](#), [7522](#), [7538](#), [7539](#),
[7547](#), [7548](#), [7550](#), [7556](#), [7559](#), [7582](#),
[7583](#), [7592](#), [7604](#), [7608](#), [7626](#), [7631](#),
[7689](#), [7963](#), [7967](#), [8160](#), [8457](#), [8458](#),
[8571](#), [8574](#), [9123](#), [9205](#), [9414](#), [9590](#),
[9634](#), [9653](#), [9660](#), [9677](#), [9684](#), [9695](#),
[9751](#), [9788](#), [9790](#), [9818](#), [9831](#), [9837](#),
[9840](#), [9849](#), [9850](#), [9853](#), [9953](#), [10208](#),
[10210](#), [10221](#), [10222](#), [10239](#), [10240](#),
[10273](#), [10285](#), [10291](#), [10353](#), [10590](#),
[10593](#), [10594](#), [10610](#), [10637](#), [10642](#),
[10658](#), [10819](#), [10936](#), [11050](#), [11056](#),
[11057](#), [11061](#), [11091](#), [11132](#), [11160](#),
[16643](#), [16815](#), [16829](#), [16842](#), [18026](#),
[18036](#), [18132](#), [18155](#), [18157](#), [18169](#),
[18766](#), [18786](#), [18939](#), [18975](#), [19145](#),
[19150](#), [19154](#), [19468](#), [19473](#), [19505](#),
[19510](#), [19866](#), [19871](#), [20256](#), [20261](#)
- \tl_set_eq:NN . [90](#), [90](#), [4900](#), [4911](#),
[5026](#), [5791](#), [6431](#), [6432](#), [6433](#), [6434](#),
[6949](#), [6950](#), [6951](#), [6952](#), [7496](#), [7497](#),
[7498](#), [7499](#), [9168](#), [9176](#), [10347](#), [16651](#)
- \tl_set_from_file:Nnn [215](#), [215](#), [18131](#)
- \tl_set_from_file_x:Nnn
..... [215](#), [215](#), [18156](#)
- \tl_set_rescan:Nnn [92](#), [92](#), [92](#), [92](#), [5057](#)
- .tl_set_x:N [165](#), [10182](#)
- \tl_show:N
[100](#), [100](#), [215](#), [795](#), [5753](#), [6266](#), [19163](#)
- \tl_show:n
[101](#), [101](#), [215](#), [795](#), [5759](#), [6265](#), [19166](#)
- \tl_sort:Nn [98](#), [98](#), [5512](#), [16827](#)
- \tl_sort:nN
... [98](#), [98](#), [736](#), [736](#), [738](#), [5512](#), [16964](#)
- \tl_tail:N [99](#), [5584](#)
- \tl_tail:n [99](#), [99](#), [99](#), [5584](#)
- \tl_to_lowercase:n [49](#), [5765](#)
- \tl_to_str:N [96](#), [96](#), [102](#),
[178](#), [543](#), [5426](#), [5756](#), [5872](#), [6225](#),
[10595](#), [11057](#), [11067](#), [11068](#), [11069](#)
- \tl_to_str:n [92](#), [92](#), [96](#),
[96](#), [96](#), [96](#), [102](#), [102](#), [108](#), [108](#), [109](#),
[109](#), [133](#), [133](#), [160](#), [162](#), [169](#), [169](#),
[178](#), [261](#), [323](#), [323](#), [382](#), [382](#), [382](#),
[394](#), [399](#), [405](#), [448](#), [449](#), [524](#), [1317](#),
[1338](#), [1464](#), [1532](#), [2315](#), [2324](#), [2327](#),
[2335](#), [2341](#), [3285](#), [3289](#), [3321](#), [3322](#),
[3355](#), [3368](#), [3370](#), [3372](#), [3390](#), [3617](#),
[4317](#), [4334](#), [4378](#), [4570](#), [4692](#), [4790](#),
[5079](#), [5177](#), [5266](#), [5338](#), [5426](#), [5597](#),
[5760](#), [5801](#), [5943](#), [5967](#), [5974](#), [6025](#),
[6032](#), [6105](#), [6124](#), [6135](#), [6160](#), [6168](#),
[6176](#), [6182](#), [6194](#), [7509](#), [7567](#), [7568](#),
[7610](#), [7633](#), [7657](#), [7658](#), [8599](#), [8682](#),
[9012](#), [9013](#), [9257](#), [9258](#), [9550](#), [9551](#),
[9552](#), [9553](#), [9583](#), [9610](#), [9614](#), [9615](#),
[9619](#), [9620](#), [9719](#), [10437](#), [10695](#)

- 10736, 10749, 11009, 11968, 11973,
- 12080, 12081, 12087, 12090, 12951,
- 17894, 18366, 18369, 19198, 19203
- \tl_to_uppercase:n 50, [5765](#)
- \tl_trim_spaces:N [97](#), [97](#), [5466](#)
- \tl_trim_spaces:n
..... [97](#), [97](#), [101](#), [5466](#), [6495](#)
- \tl_trim_spaces:n [386](#)
- \tl_upper_case:n [212](#), [212](#), [18176](#)
- tl_upper_case:n [108](#)
- \tl_upper_case:nn ... [212](#), [212](#), [18176](#)
- \tl_use:N [62](#),
[80](#), [84](#), [87](#), [96](#), [96](#), [5428](#), [10019](#), [19096](#)
- \g_tmpa_tl [101](#), [5761](#)
- \l_tmpa_tl [4](#), [91](#), [91](#), [91](#), [101](#), [5763](#)
- \g_tmpb_tl [101](#), [5761](#)
- \l_tmpb_tl [101](#), [5763](#)
- tl internal commands:
- \c_tl_accents_lt_tl [782](#)
- __tl_act:NNnn [387](#), [387](#),
[388](#), [388](#), [771](#), [5512](#), [5563](#), [18096](#), [18119](#)
- __tl_act_count_group:nn [18115](#)
- __tl_act_count_normal:nN [18115](#)
- __tl_act_count_space:n [18115](#)
- __tl_act_end:w [5512](#)
- __tl_act_end:wn [771](#), [5533](#), [5539](#)
- __tl_act_group:nwnNNN [5512](#)
- __tl_act_group_recurse:Nnn
..... [18106](#), [18110](#)
- __tl_act_loop:w [5512](#)
- __tl_act_normal:NwnNNN [5512](#)
- __tl_act_output:n [388](#), [5512](#)
- __tl_act_result:n
[387](#), [5517](#), [5539](#), [5554](#), [5555](#), [5556](#), [5557](#)
- __tl_act_reverse [388](#)
- __tl_act_reverse_output:n
..... [5512](#), [5573](#), [5575](#), [5577](#), [18107](#)
- __tl_act_space:wnNNN [387](#), [5512](#)
- __tl_case:NnTF
..... [5348](#), [5353](#), [5358](#), [5363](#), [5365](#)
- __tl_case:nnTF [5345](#)
- __tl_case:Nw [5345](#)
- __tl_case_end:nw [5345](#)
- __tl_change_case:nnn
.. [18176](#), [18177](#), [18179](#), [18180](#), [18182](#)
- __tl_change_case_aux:nnn [18182](#)
- __tl_change_case_char:nN
..... [18182](#), [19098](#)
- __tl_change_case_char:Nnn ... [18182](#)
- __tl_change_case_char_auxi:nN [18182](#)
- __tl_change_case_char_auxii:nN .
..... [18182](#)
- __tl_change_case_char_UTFviii:nn
..... [18182](#)
- __tl_change_case_char_UTFviii:nnN
..... [18359](#), [18361](#), [18363](#), [18364](#)
- __tl_change_case_char_UTFviii:nnNN
..... [18182](#)
- __tl_change_case_char_UTFviii:nnNNN
..... [18182](#)
- __tl_change_case_char_UTFviii:nnNNNN
..... [18326](#), [18362](#)
- __tl_change_case_cs:N [18182](#)
- __tl_change_case_cs:NN [18182](#)
- __tl_change_case_cs:NNn [18182](#)
- __tl_change_case_cs_accents:NN .
..... [18182](#)
- __tl_change_case_cs_expand:NN [18182](#)
- __tl_change_case_cs_expand:Nnw .
..... [18182](#)
- __tl_change_case_cs_letterlike:Nnn
..... [18182](#), [19046](#)
- __tl_change_case_end:wn [18182](#), [19023](#)
- __tl_change_case_group:nwnn . [18182](#)
- __tl_change_case_if_expandable:NTF
..... [18182](#),
[18485](#), [18518](#), [18612](#), [18662](#), [19120](#)
- __tl_change_case_loop:wn [781](#)
- __tl_change_case_loop:wnn
.. [774](#), [776](#), [18182](#), [19013](#), [19052](#), [19056](#)
- __tl_change_case_lower_az:Nnw [18497](#)
- __tl_change_case_lower_lt:nNnw .
..... [18569](#)
- __tl_change_case_lower_lt:NNw [18569](#)
- __tl_change_case_lower_lt:Nnw [18569](#)
- __tl_change_case_lower_lt:nnw [18569](#)
- __tl_change_case_lower_lt:Nw . [18569](#)
- __tl_change_case_lower_sigma:Nnw
..... [18467](#)
- __tl_change_case_lower_sigma:Nw
..... [18467](#)
- __tl_change_case_lower_sigma:w .
..... [18467](#)
- __tl_change_case_lower_tr:Nnw [18497](#)
- __tl_change_case_lower_tr_-
auxi:Nw [18497](#)
- __tl_change_case_lower_tr_-
auxii:Nw [18497](#)
- __tl_change_case_math:NNNnnn ...
..... [774](#), [18182](#), [19036](#)
- __tl_change_case_math:NwNNnn . [18182](#)
- __tl_change_case_math_group:nwNNnn
..... [18182](#)
- __tl_change_case_math_loop:wNNnn
..... [18182](#)

| | |
|--|---|
| __tl_change_case_math_space:wNNnn | 18182 |
| __tl_change_case_mixed_nl:NNw | 19100 |
| __tl_change_case_mixed_nl:Nnw | 19100 |
| __tl_change_case_mixed_nl:Nw | 19100 |
| __tl_change_case_N_type:Nnnn | 18182 |
| __tl_change_case_N_type:NNNnnn | 18182 |
| __tl_change_case_N_type:Nwnn | 18182 |
| __tl_change_case_output:nwn | 18182, 18471, 18505, 18513, 18529,
18531, 18541, 18552, 18564, 18591,
18600, 18628, 18650, 18681, 19005,
19017, 19081, 19095, 19108, 19136 |
| __tl_change_case_protect:wNN | 18182 |
| __tl_change_case_result:n | 18195, 18208, 18209, 18211, 18990 |
| __tl_change_case_setup:NN | 18950, 18955, 18957 |
| __tl_change_case_space:wnn | 18182 |
| __tl_change_case_upper_az:Nnw | 18497 |
| __tl_change_case_upper_de-alt:Nnw | 18678 |
| __tl_change_case_upper_lt:NNw | 18569 |
| __tl_change_case_upper_lt:Nnw | 18569 |
| __tl_change_case_upper_lt:nnw | 18569 |
| __tl_change_case_upper_lt:Nw | 18569 |
| __tl_change_case_upper_sigma:Nnw | 18467 |
| __tl_change_case_upper_tr:Nnw | 18497 |
| __tl_count:n | 385, 5437 |
| __tl_from_file_do:w | 18131 |
| __tl_head_auxi:nw | 5584 |
| __tl_head_auxii:n | 5584 |
| __tl_if_blank_p:NNw | 5241 |
| __tl_if_empty_return:n | 379, 380,
2868, 2874, 5242, 5276, 18084, 18088 |
| __tl_if_head_eq_meaning_-
normal:nN | 5658, 5662 |
| __tl_if_head_eq_meaning_-
special:nN | 5659, 5671 |
| __tl_if_head_is_N_type:w | 392, 5690 |
| __tl_if_head_is_space:w | 5718 |
| __tl_if_single:nnw | 382, 5337, 5344 |
| __tl_if_single:nTF | 5335 |
| __tl_if_single_p:n | 5335 |
| \l_tl_internal_a_tl | 372, 5082, 5087, 5146, 5299,
18149, 18155, 18169, 18172, 18766,
18768, 18786, 18791, 18939, 18941 |
| \l_tl_internal_b_tl | 5299 |
| __tl_item:nn | 5730 |
| __tl_item_aux:nn | 5730 |
| __tl_lookup_lower:N | 18182 |
| __tl_lookup_title:N | 18182 |
| __tl_lookup_upper:N | 18182 |
| __tl_loop:nn | 18783, 18792, 18823 |
| __tl_map_function:Nn | 383, 5380, 5398 |
| __tl_map_variable:Nnn | 5406 |
| __tl_mixed_case:nn | 18178, 18181, 18977 |
| __tl_mixed_case_aux:nn | 18977 |
| __tl_mixed_case_char:N | 18977 |
| __tl_mixed_case_char:Nn | 19051, 19057 |
| __tl_mixed_case_char:nN | 18977 |
| __tl_mixed_case_group:nwn | 18977 |
| __tl_mixed_case_letterlike:Nw | 18977 |
| __tl_mixed_case_loop:wn | 18977 |
| __tl_mixed_case_N_type:Nnn | 18977 |
| __tl_mixed_case_N_type:NNNnn | 18977 |
| __tl_mixed_case_N_type:Nwn | 18977 |
| __tl_mixed_case_skip:N | 18977 |
| __tl_mixed_case_skip:NN | 18977 |
| __tl_mixed_case_skip_tidy:Nwn | 18977 |
| __tl_mixed_case_space:wn | 18977 |
| __tl_replace:NnNNnn | 375, 376, 5161, 5163, 5165, 5167, 5172 |
| __tl_replace_auxi:NnnNNNnn | 376, 5172 |
| __tl_replace_auxii:nNNNnn | 376, 376, 377, 5172 |
| __tl_replace_next:w | 375,
377, 377, 377, 377, 5165, 5167, 5172 |
| __tl_replace_wrap:w | 375, 377,
377, 377, 377, 377, 5161, 5163, 5172 |
| __tl_rescan:w | 374, 374, 5057, 5138, 5139, 5154 |
| \c_tl_rescan_marker_tl | 374,
5056, 5074, 5095, 5143, 18143, 18154 |
| __tl_reverse_group:nn | 18091 |
| __tl_reverse_group_preserve:nn | 5558 |
| __tl_reverse_items:nwNwn | 5450 |
| __tl_reverse_items:wn | 5450 |
| __tl_reverse_normal:nN | 5558, 18097 |
| __tl_reverse_space:n | 5558, 18099 |
| __tl_set_from_file:NNnn | 18131 |
| __tl_set_from_file_x:NNnn | 18156 |
| __tl_set_rescan:n | 372, 372, 5079, 5101 |
| __tl_set_rescan:NNnn | 5057 |
| __tl_set_rescan:NnTF | 5101 |
| __tl_set_rescan_multi:n | 372, 374, 5057, 5111 |
| __tl_set_rescan_multiple:n | 373 |
| __tl_set_rescan_single:nn | 373, 5101 |
| __tl_set_rescan_single_aux:nn | 5101 |
| __tl_tmp:w | 381,
386, 5323, 5324, 5474, 5511, 18749,
18753, 18756, 18768, 18772, 18773, |

- 18774, 18775, 18791, 18794, 18925,
 18928, 18941, 18944, 18945, 18946
 _tl_trim_spaces:nn 101,
 101, 503, 5467, 5474, 7006, 7448, 9751
 _tl_trim_spaces_auxi:w .. 386, 5474
 _tl_trim_spaces_auxii:w . 386, 5474
 _tl_trim_spaces_auxiii:w 386, 5474
 _tl_trim_spaces_auxiv:w . 386, 5474
 token commands:
 \c_alignment_token 51, 319, 3183, 3222
 \c_parameter_token
 51, 319, 319, 3183, 3226, 3229
 \g_peek_token 56, 56, 3441, 3452
 \l_peek_token ... 55, 56, 328, 329,
 795, 795, 3441, 3450, 3500, 3512,
 3532, 3541, 19181, 19182, 19183, 19186
 \c_space_token
 . 51, 100, 101, 216, 320, 391, 3183,
 3246, 3541, 5648, 5683, 18170, 19183
 \token_get_arg_spec:N .. 58, 58, 3615
 \token_get_prefix_spec:N 59, 59, 3615
 \token_get_replacement_spec:N ...
 58, 58, 3615, 10479
 \token_if_active:NTF ... 53, 53, 3259
 \token_if_active_p:N ... 53, 53, 3259
 \token_if_alignment:NTF
 52, 52, 53, 3220
 \token_if_alignment_p:N . 52, 52, 3220
 \token_if_chardef:NTF .. 54, 54, 3333
 \token_if_chardef_p:N .. 54, 54, 3333
 \token_if_cs:NTF
 54, 54, 3298, 18296, 19044
 \token_if_cs_p:N 54,
 54, 3298, 18526, 18620, 18670, 19128
 \token_if_dim_register:NTF
 54, 54, 3333
 \token_if_dim_register_p:N
 54, 54, 3333
 \token_if_eq_catcode:NNTF
 53, 53, 56, 56, 56, 56, 3269
 \token_if_eq_catcode_p:NN 53, 53, 3269
 \token_if_eq_charcode:NNTF
 ... 53, 53, 56, 57, 57, 57, 3274, 10627
 \token_if_eq_charcode_p:NN
 53, 53, 3274
 \token_if_eq_meaning:NNTF
 . 53, 53, 57, 57, 57, 58, 329, 3264,
 11625, 12609, 13393, 13395, 13400,
 15545, 18246, 18274, 18278, 19032
 \token_if_eq_meaning_p:NN
 53, 53, 3264, 18452
 \token_if_expandable:NTF
 54, 54, 3303, 18446
 \token_if_expandable_p:N 54, 54, 3303
 \token_if_group_begin:NTF 52, 52, 3205
 \token_if_group_begin_p:N 52, 52, 3205
 \token_if_group_end:NTF . 52, 52, 3210
 \token_if_group_end_p:N . 52, 52, 3210
 \token_if_int_register:NTF
 55, 55, 3333
 \token_if_int_register_p:N
 55, 55, 3333
 \token_if_letter:N 321
 \token_if_letter:NTF
 53, 53, 3249, 18491
 \token_if_letter_p:N ... 53, 53, 3249
 \token_if_long_macro:NTF 54, 54, 3333
 \token_if_long_macro_p:N 54, 54, 3333
 \token_if_macro:NTF
 . 54, 54, 3279, 3385, 3621, 3630, 3639
 \token_if_macro_p:N 54, 54, 3279
 \token_if_math_subscript:NTF ...
 53, 53, 3239
 \token_if_math_subscript_p:N ...
 53, 53, 3239
 \token_if_math_superscript:NTF ..
 53, 53, 3233
 \token_if_math_superscript_p:N ..
 53, 53, 3233
 \token_if_math_toggle:NTF 52, 52, 3215
 \token_if_math_toggle_p:N 52, 52, 3215
 \token_if_mathchardef:NTF 54, 54, 3333
 \token_if_mathchardef_p:N 54, 54, 3333
 \token_if_muskip_register:NTF ...
 55, 55, 3333
 \token_if_muskip_register_p:N ...
 55, 55, 3333
 \token_if_other:NTF 53, 53, 3254
 \token_if_other_p:N 53, 53, 3254
 \token_if_parameter:NTF 53, 3225
 \token_if_parameter_p:N . 53, 53, 3225
 \token_if_primitive:NTF . 55, 55, 3379
 \token_if_primitive_p:N . 55, 55, 3379
 \token_if_protected_long_-
 macro:NTF 54, 54, 3333
 \token_if_protected_long_macro_-
 p:N 54, 54, 3333, 18451
 \token_if_protected_macro:NTF ...
 54, 54, 3333
 \token_if_protected_macro_p:N ...
 54, 54, 3333, 18450
 \token_if_skip_register:NTF
 55, 55, 3333
 \token_if_skip_register_p:N
 55, 55, 3333
 \token_if_space:NTF 53, 53, 3244
 \token_if_space_p:N 53, 53, 3244

| | | | |
|-------------------------------------|---|--|-----------------------------------|
| \token_if_toks_register:NTF | 55, 55, 3333 | \toksapp | 926 |
| \token_if_toks_register_p:N | 55, 55, 3333 | \toksdef | 562, 17053 |
| \token_new:Nn | 51, 51, 3182 | \tokspre | 927 |
| \token_to_meaning:N | 52, 52, 321, 324, 1315 ,
1329 , 1744, 1754, 2324, 3182 , 3285,
3354, 3389, 3624, 3633, 3642, 19186 | \tolerance | 563 |
| \token_to_str:N | 4, 17, 52, 52, 52, 102, 160, 178,
265, 323, 323, 338, 393, 580, 581,
582, 1317 , 1329 , 1329, 1448, 1457,
1478, 1499, 1540, 1545, 1560, 1592,
1593, 1603, 1609, 1744, 1754, 1756,
1769, 1779, 1900, 1930, 1937, 2021,
2026, 2349, 2410, 2425, 2458, 2469,
2482, 2586, 2891, 3182 , 3366, 3367,
3372, 3373, 3374, 3375, 3376, 3377,
3856, 4414, 5056, 5695, 5711, 5756,
7872, 8005, 8159, 8746, 9572, 9578,
9663, 9687, 11039, 11040, 11041,
11042, 11043, 11500, 11501, 11967,
11980, 11981, 12010, 12139, 12187,
12219, 12239, 12254, 12266, 12267,
12280, 12281, 12306, 12315, 12317,
12342, 12345, 12370, 12372, 12386,
12402, 12420, 12489, 12499, 12500,
12515, 12516, 12818, 15613, 16678,
17057, 17073, 18336, 18377, 18380,
18388, 18953, 18954, 18971, 18972 | \topmark | 564 |
| token internal commands: | | \topmarks | 666 |
| \c_token_A_int | 3379, 3416 | \topskip | 565 |
| __token_delimit_by_char:w . . | 3315 | \tpack | 928 |
| __token_delimit_by_count:w . . | 3315 | \tracingassigns | 667 |
| __token_delimit_by_dimen:w . . | 3315 | \tracingcommands | 566 |
| __token_delimit_by_macro:w . . | 3315 | \tracingfonts | 971 |
| __token_delimit_by_muskip:w . . | 3315 | \tracinggroups | 668 |
| __token_delimit_by_skip:w . . . | 3315 | \tracingifs | 669 |
| __token_delimit_by_toks:w . . . | 3315 | \tracinglostchars | 567 |
| __token_if_macro_p:w | 3279 | \tracingmacros | 568 |
| __token_if_primitive:NNw | 3379 | \tracingnesting | 670 |
| __token_if_primitive:Nw | 3379 | \tracingonline | 569 |
| __token_if_primitive_loop:N . . | 3379 | \tracingoutput | 570 |
| __token_if_primitive_nullfont:N | 3379 | \tracingpages | 571 |
| __token_if_primitive_space:w . | 3379 | \tracingparagraphs | 572 |
| __token_if_primitive_undefined:N | 3379 | \tracingrestores | 573 |
| __token_tmp:w | 323, 3316,
3325, 3326, 3327, 3328, 3329, 3330,
3331, 3334, 3366, 3367, 3368, 3369,
3371, 3373, 3374, 3375, 3376, 3377 | \tracingscantokens | 671 |
| \toks | 561, 3377 | \tracingstats | 574 |
| | | true | 197 |
| | | trunc | 193 |
| | | two commands: | |
| | | \c_thirty_two | |
| | | 72, 4442 , 12127, 12165, 12810 | |
| | | \c_two_hundred_fifty_five . . | 72, 4443 |
| | | \c_two_hundred_fifty_six . . . | 72, 4443 |
| | | U | |
| | | \u | 18976 |
| | | \uccode | 167, 182, 195, 197, 199, 201, 575 |
| | | \Uchar | 973 |
| | | \Ucharcat | 974 |
| | | \uchyph | 576 |
| | | \ucs | 1159 |
| | | \Udelcode | 975 |
| | | \Udelcodenum | 976 |
| | | \Udelimiter | 977 |
| | | \Udelimiterover | 978 |
| | | \Udelimiterunder | 979 |
| | | \Uhexensible | 980 |
| | | \Umathaccent | 981 |
| | | \Umathaxis | 982 |
| | | \Umathbinbinspacing | 983 |
| | | \Umathbinclosespacing | 984 |
| | | \Umathbininnerspacing | 985 |
| | | \Umathbinopenspacing | 986 |
| | | \Umathbinopspacing | 987 |
| | | \Umathbinordspacing | 988 |
| | | \Umathbinpunctspacing | 989 |
| | | \Umathbinrelspacing | 990 |

| | | | |
|--|-----------|--|------|
| <code>\Umathchar</code> | 991 | <code>\Umathopordspacing</code> | 1045 |
| <code>\Umathcharclass</code> | 992 | <code>\Umathoppunctspacing</code> | 1046 |
| <code>\Umathchardef</code> | 993 | <code>\Umathoprelspacing</code> | 1047 |
| <code>\Umathcharfam</code> | 994 | <code>\Umathordbinspacing</code> | 1048 |
| <code>\Umathcharnum</code> | 995 | <code>\Umathordclosespacing</code> | 1049 |
| <code>\Umathcharnumdef</code> | 996 | <code>\Umathordinnerspacing</code> | 1050 |
| <code>\Umathcharslot</code> | 997 | <code>\Umathordopenspacing</code> | 1051 |
| <code>\Umathclosebinspacing</code> | 998 | <code>\Umathordopspacing</code> | 1052 |
| <code>\Umathcloseclosespacing</code> | 999 | <code>\Umathordordspacing</code> | 1053 |
| <code>\Umathcloseinnerspacing</code> | 1000 | <code>\Umathordpunctspacing</code> | 1054 |
| <code>\Umathcloseopenspacing</code> | 1001 | <code>\Umathordrelspacing</code> | 1055 |
| <code>\Umathcloseopspacing</code> | 1002 | <code>\Umathoverbarkern</code> | 1056 |
| <code>\Umathcloseordspacing</code> | 1003 | <code>\Umathoverbarrule</code> | 1057 |
| <code>\Umathclosepunctspacing</code> | 1004 | <code>\Umathoverbarvgap</code> | 1058 |
| <code>\Umathcloserelspacing</code> | 1005 | <code>\Umathoverdelimiterbgap</code> | 1059 |
| <code>\Umathcode</code> | 158, 1006 | <code>\Umathoverdelimitervgap</code> | 1060 |
| <code>\Umathcodenum</code> | 1007 | <code>\Umathpunctbinspacing</code> | 1061 |
| <code>\Umathconnectoroverlapmin</code> | 1008 | <code>\Umathpunctclosespacing</code> | 1062 |
| <code>\Umathfractiondelsize</code> | 1009 | <code>\Umathpunctinnerspacing</code> | 1063 |
| <code>\Umathfractiondenomdown</code> | 1010 | <code>\Umathpunctopenspacing</code> | 1064 |
| <code>\Umathfractiondenomvgap</code> | 1011 | <code>\Umathpunctopspacing</code> | 1065 |
| <code>\Umathfractionnumup</code> | 1012 | <code>\Umathpunctordspacing</code> | 1066 |
| <code>\Umathfractionnumvgap</code> | 1013 | <code>\Umathpunctpunctspacing</code> | 1067 |
| <code>\Umathfractionrule</code> | 1014 | <code>\Umathpunctrelspacing</code> | 1068 |
| <code>\Umathinnerbinspacing</code> | 1015 | <code>\Umathquad</code> | 1069 |
| <code>\Umathinnerclosespacing</code> | 1016 | <code>\Umathradicaldegreeafter</code> | 1070 |
| <code>\Umathinnerinnerspacing</code> | 1017 | <code>\Umathradicaldegreebefore</code> | 1071 |
| <code>\Umathinneropspacing</code> | 1018 | <code>\Umathradicaldegreeraise</code> | 1072 |
| <code>\Umathinneropspacing</code> | 1019 | <code>\Umathradicalkern</code> | 1073 |
| <code>\Umathinnerordspacing</code> | 1020 | <code>\Umathradicalrule</code> | 1074 |
| <code>\Umathinnerpunctspacing</code> | 1021 | <code>\Umathradicalvgap</code> | 1075 |
| <code>\Umathinnerrelspacing</code> | 1022 | <code>\Umathrelbinspacing</code> | 1076 |
| <code>\Umathlimitabovebgap</code> | 1023 | <code>\Umathrelclosespacing</code> | 1077 |
| <code>\Umathlimitabovekern</code> | 1024 | <code>\Umathrelinnerspacing</code> | 1078 |
| <code>\Umathlimitabovevgap</code> | 1025 | <code>\Umathrelopenspacing</code> | 1079 |
| <code>\Umathlimitbelowbgap</code> | 1026 | <code>\Umathrelopspacing</code> | 1080 |
| <code>\Umathlimitbelowkern</code> | 1027 | <code>\Umathrelordspacing</code> | 1081 |
| <code>\Umathlimitbelowvgap</code> | 1028 | <code>\Umathrelpunctspacing</code> | 1082 |
| <code>\Umathnolimitsubfactor</code> | 1029 | <code>\Umathrelrelspacing</code> | 1083 |
| <code>\Umathnolimitsupfactor</code> | 1030 | <code>\Umathskewedfractionhgap</code> | 1084 |
| <code>\Umathopbinspacing</code> | 1031 | <code>\Umathskewedfractionvgap</code> | 1085 |
| <code>\Umathopclosespacing</code> | 1032 | <code>\Umathspaceafterscript</code> | 1086 |
| <code>\Umathopenbinspacing</code> | 1033 | <code>\Umathstackdenomdown</code> | 1087 |
| <code>\Umathopenclosespacing</code> | 1034 | <code>\Umathstacknumup</code> | 1088 |
| <code>\Umathopeninnerspacing</code> | 1035 | <code>\Umathstackvgap</code> | 1089 |
| <code>\Umathopenopenspacing</code> | 1036 | <code>\Umathsubshiftdown</code> | 1090 |
| <code>\Umathopenopspacing</code> | 1037 | <code>\Umathsubshiftdrop</code> | 1091 |
| <code>\Umathopenordspacing</code> | 1038 | <code>\Umathsubsupshiftdown</code> | 1092 |
| <code>\Umathopenpunctspacing</code> | 1039 | <code>\Umathsubsupvgap</code> | 1093 |
| <code>\Umathopenrelspacing</code> | 1040 | <code>\Umathsubtopmax</code> | 1094 |
| <code>\Umathoperatorsize</code> | 1041 | <code>\Umathsupbottommin</code> | 1095 |
| <code>\Umathopinnerspacing</code> | 1042 | <code>\Umathsupshiftdrop</code> | 1096 |
| <code>\Umathopopenspacing</code> | 1043 | <code>\Umathsupshiftdown</code> | 1097 |
| <code>\Umathopopspacing</code> | 1044 | <code>\Umathsupsubbottommax</code> | 1098 |

| | | | |
|---|------------|---|--------------------|
| <code>\Umathunderbarkern</code> | 1099 | <code>\unvcopy</code> | 584 |
| <code>\Umathunderbarrule</code> | 1100 | <code>\Uoverdelimenter</code> | 1104 |
| <code>\Umathunderbarvgap</code> | 1101 | <code>\uppercase</code> | 585 |
| <code>\Umathunderdelimenterbgap</code> | 1102 | uptex commands: | |
| <code>\Umathunderdelimentervgap</code> | 1103 | <code>\uptex_disablecjktoken:D</code> | |
| undefine commands: | | 1153, 3757, 3760, 19259 | |
| <code>.undefine:</code> | 166, 10198 | <code>\uptex_enablecjktoken:D</code> | 1154 |
| <code>\underline</code> | 577 | <code>\uptex_forcecjktoken:D</code> | 1155 |
| <code>\unexpanded</code> | 672 | <code>\uptex_kchar:D</code> | 1156 |
| <code>\unhbox</code> | 578 | <code>\uptex_kchardef:D</code> | 1157, 3761 |
| <code>\unhcopy</code> | 579 | <code>\uptex_kuten:D</code> | 1158 |
| unicode internal commands: | | <code>\uptex_ucs:D</code> | 1159 |
| <code>\c__unicode_accents_lt_tl</code> | | <code>\Uradical</code> | 1105 |
| 18572, 18723 | | <code>\Uroot</code> | 1106 |
| <code>__unicode_codepoint_to_UTFviii:n</code> | | use commands: | |
| .. 18684, 18767, 18788, 18789, 18940 | | <code>\use:N</code> | 16, 16, |
| <code>__unicode_codepoint_to_UTFviii-</code> | | 16, 16, 262, 1376, 1473, 1555, 1683, | |
| <code>auxi:n</code> | 18684 | 1685, 1687, 1689, 2631, 2650, 3854, | |
| <code>__unicode_codepoint_to_UTFviii-</code> | | 4274, 4284, 4389, 4393, 4395, 4397, | |
| <code>auxii:Nnn</code> | 18684 | 4398, 4402, 4574, 6234, 6238, 9045, | |
| <code>__unicode_codepoint_to_UTFviii-</code> | | 9056, 9071, 9080, 9088, 9096, 9102, | |
| <code>auxiii:n</code> | 18684 | 9128, 9863, 9870, 10071, 11137, | |
| <code>\g__unicode_data_iior</code> | | 17895, 18307, 18344, 18348, 18349 | |
| .. 6271, 6274, 6291, 6294, 6299, 6305 | | <code>\use:n</code> | 17, 17, 18, |
| <code>\c__unicode_dot_above_tl</code> 18628, 18723 | | 18, 59, 89, 213, 258, 374, 455, 498, | |
| <code>\c__unicode_dotless_i_tl</code> | | 498, 560, 581, 738, 738, 738, 738, | |
| 18513, 18529, 18541, 18746 | | 738, 1377, 1382, 1460, 1506, 1526, | |
| <code>\c__unicode_dotted_I_tl</code> 18564, 18746 | | 1606, 1847, 2332, 2338, 2359, 2479, | |
| <code>\c__unicode_final_sigma_tl</code> | | 3139, 3197, 3279, 3318, 3336, 3380, | |
| 18481, 18493, 18723 | | 4410, 4689, 5002, 5062, 5134, 5136, | |
| <code>\c__unicode_I_ogonek_tl</code> 18640, 18746 | | 5418, 5674, 6212, 6302, 6371, 6372, | |
| <code>\c__unicode_i_ogonek_tl</code> 18585, 18746 | | 6375, 7325, 7851, 7869, 9009, 9026, | |
| <code>__unicode_map_inline:n</code> | | 9254, 9271, 9506, 9559, 9872, 10474, | |
| 6286, 6350, 6368, 6383 | | 10619, 10873, 10972, 11063, 11548, | |
| <code>__unicode_map_loop:</code> 6293, 6297, 6310 | | 11556, 11565, 11582, 11590, 11618, | |
| <code>__unicode_parse:w</code> .. 6308, 6313, 6320 | | 11970, 12049, 12084, 12113, 16807, | |
| <code>__unicode_parse_auxi:w</code> | | 17049, 17915, 17975, 19235, 19241, | |
| 6316, 6334, 6351, 6369 | | 19249, 19263, 19270, 19279, 19292, | |
| <code>__unicode_parse_auxii:w</code> | | 19295, 19302, 19307, 19694, 19728, | |
| 6336, 6337, | | 19741, 20422, 20462, 20479, 20497 | |
| 6362, 6366, 6371, 6372, 6375, 6378 | | <code>\use:nn</code> .. 17, 17, 1382, 2061, 3615, | |
| <code>\c__unicode_std_sigma_tl</code> 18492, 18723 | | 4568, 5094, 7309, 12075, 15219, 18152 | |
| <code>__unicode_store:nnnnn</code> | | <code>\use:nnn</code> | 17, 17, 1382, 1897 |
| 6324, 6367, 6381 | | <code>\use:nnnn</code> | 17, 17, 1382 |
| <code>__unicode_tmp:NN</code> ... 6386, 6394, 6396 | | <code>\use_i:nn</code> | 17, |
| <code>\l__unicode_tmp_tl</code> .. 6305, 6306, 6308 | | 17, 17, 256, 260, 300, 301, 444, 559, | |
| <code>\c__unicode_upper_Eszett_tl</code> | | 691, 694, 706, 710, 710, 722, 781, | |
| 18681, 18723 | | 1333, 1386, 1412, 1489, 1633, 1661, | |
| <code>\uniformdeviate</code> | 972 | 1879, 2476, 2627, 2640, 2644, 2661, | |
| <code>\unkern</code> | 580 | 2662, 5598, 6503, 6505, 7515, 11471, | |
| <code>\unless</code> | 673 | 12049, 12692, 12696, 12722, 13828, | |
| <code>\unpenalty</code> | 581 | 14312, 14529, 15045, 15214, 15877, | |
| <code>\unskip</code> | 582 | 16091, 16521, 16905, 16940, 16950, | |
| <code>\unvbox</code> | 583 | | |

- 16960, 18455, 18532, 19243, 19251,
19265, 19272, 19281, 19297, 19304
`\use_i:nnn` 18, 18, 18, 722,
1388, 1615, 3624, 6708, 13786, 16515
`\use_i:nnnn`
18, 18, 18, 1388, 13804, 13811, 14001
`\use_i_delimit_by_q_nil:nw`
19, 19, 1399
`\use_i_delimit_by_q_recursion_-
stop:nw` 19,
19, 1399, 2810, 2826, 17924, 17939,
18248, 18407, 18430, 19034, 19079
`\use_i_delimit_by_q_stop:nw`
19, 19,
405, 405, 1399, 5989, 5998, 6113,
6164, 6167, 7418, 16500, 16572, 17996
`\use_i_ii:nnn`
18, 18, 1388, 2086, 6685, 6785
`\use_ii:nn`
17, 17, 40, 256, 260, 300, 301,
301, 444, 691, 694, 706, 710, 710,
720, 791, 796, 1335, 1386, 1414,
1491, 1635, 1663, 1877, 2030, 2640,
5106, 5600, 7516, 11444, 12694,
12698, 12724, 13830, 15047, 15879,
16093, 16457, 18304, 18454, 18457,
18496, 19062, 19236, 19293, 19308
`\use_ii:nnn` 18, 18, 500, 1388, 1617, 3633
`\use_ii:nnnn` 18, 18, 1388
`\use_iii:nnn` 18,
18, 559, 559, 1388, 2035, 3642, 11456
`\use_iii:nnnn` 18, 18, 1388
`\use_iv:nnnn` 18, 18, 1388
`\use_none:n` 18, 18, 22, 101,
379, 386, 386, 423, 431, 435, 484,
543, 543, 577, 578, 578, 578, 792,
798, 1402, 1506, 1695, 1711, 1715,
1849, 1901, 2812, 2827, 3429, 4144,
4150, 5221, 5242, 5509, 5597, 5613,
5677, 5694, 5704, 5705, 5710, 5725,
5728, 6300, 6407, 6773, 6794, 6968,
7093, 7186, 7212, 8964, 9319, 9323,
9700, 9723, 9741, 10430, 11061,
11377, 11381, 11385, 11389, 11482,
11510, 12619, 12843, 12855, 12866,
12878, 12920, 12945, 13014, 13030,
13072, 13805, 13808, 14716, 15613,
16465, 16505, 16512, 16854, 18018,
18019, 18084, 18592, 19234, 19242,
19250, 19264, 19271, 19280, 19291,
19296, 19303, 19306, 20437, 20439
`\use_none:nn` ... 18, 377, 382, 390,
391, 420, 420, 420, 421, 524, 1402,
1450, 1458, 5204, 5338, 5465, 5624,
5643, 6575, 6694, 6715, 7226, 10438,
11330, 11376, 11380, 11384, 11388
`\use_none:nnn` 18, 391,
1402, 5665, 11375, 11379, 11383, 11387
`\use_none:nnnn`
18, 291, 1402, 2411, 2426, 4012
`\use_none:nnnnn`
18, 259, 564, 1402, 11543, 11577,
11603, 11611, 13406, 16518, 16534
`\use_none:nnnnnn` 18, 1402, 1562
`\use_none:nnnnnnn`
18, 262, 564, 1402, 1480, 11545,
11579, 11605, 11613, 11902, 13844
`\use_none:nnnnnnnn` 18, 1402
`\use_none:nnnnnnnnn` 18, 1402
`\use_none_delimit_by_q_nil:w` ...
19, 19, 1396
`\use_none_delimit_by_q_recursion_-
stop:w` 19, 19, 44, 44, 44,
44, 1396, 1471, 1541, 1546, 1553,
2350, 2357, 2804, 2819, 5000, 6389
`\use_none_delimit_by_q_stop:w` 19,
19, 310, 434, 434, 795, 1396, 3870,
4578, 5987, 5996, 6151, 7173, 7404,
7409, 9146, 11167, 16501, 17914, 19188
use internal commands:
`__use_none_delimit_by_s_stop:w`
..... 46, 46, 46, 2899
`\useboxresource` 965
`\useimageresource` 966
`\Uskewed` 1107
`\Uskewedwithdelims` 1108
`\Ustack` 1109
`\Ustartdisplaymath` 1110
`\Ustartmath` 1111
`\Ustopdisplaymath` 1112
`\Ustopmath` 1113
`\Usubscript` 1114
`\Usuperscript` 1115
utex commands:
`\utex_binbinspacing:D` 983
`\utex_binclosespacing:D` 984
`\utex_bininnerspacing:D` 985
`\utex_binopenspacing:D` 986
`\utex_binopspacing:D` 987
`\utex_binordspacing:D` 988
`\utex_binpunctspacing:D` 989
`\utex_binrelspacing:D` 990
`\utex_char:D` . 407, 973, 1216, 6230,
6238, 6292, 6326, 6328, 6329, 6331,
6345, 6346, 6357, 6358, 6384, 18311,
18340, 18497, 18723, 18725, 18726,
18729, 18730, 18731, 18732, 18733,
18734, 18736, 18737, 18747, 18750

| | | | |
|--|-----------------|--|------|
| <code>\utex_charcat:D</code> | 974, 3094, 3110 | <code>\utex_openopenspacing:D</code> | 1036 |
| <code>\utex_closebinspacing:D</code> | 998 | <code>\utex_openopspacing:D</code> | 1037 |
| <code>\utex_closeclosespacing:D</code> | 999 | <code>\utex_openordspacing:D</code> | 1038 |
| <code>\utex_closeinnerspacing:D</code> | 1000 | <code>\utex_openpunctspacing:D</code> | 1039 |
| <code>\utex_closeopenspacing:D</code> | 1001 | <code>\utex_openrelspacing:D</code> | 1040 |
| <code>\utex_closeopspacing:D</code> | 1002 | <code>\utex_operatorsize:D</code> | 1041 |
| <code>\utex_closeordspacing:D</code> | 1003 | <code>\utex_opinnerspacing:D</code> | 1042 |
| <code>\utex_closepunctspacing:D</code> | 1004 | <code>\utex_opopenspacing:D</code> | 1043 |
| <code>\utex_closerelspacing:D</code> | 1005 | <code>\utex_opopspacing:D</code> | 1044 |
| <code>\utex_connectoroverlapmin:D</code> | 1008 | <code>\utex_opordspacing:D</code> | 1045 |
| <code>\utex_delcode:D</code> | 975, 1245 | <code>\utex_oppunctspacing:D</code> | 1046 |
| <code>\utex_delcodenum:D</code> | 976, 1246 | <code>\utex_oprelspacing:D</code> | 1047 |
| <code>\utex_delimiter:D</code> | 977, 1247 | <code>\utex_ordbinspacing:D</code> | 1048 |
| <code>\utex_delimiterover:D</code> | 978 | <code>\utex_ordclosespacing:D</code> | 1049 |
| <code>\utex_delimiterunder:D</code> | 979 | <code>\utex_ordinnerspacing:D</code> | 1050 |
| <code>\utex_fractiondelsize:D</code> | 1009 | <code>\utex_ordopenspacing:D</code> | 1051 |
| <code>\utex_fractiondenomdown:D</code> | 1010 | <code>\utex_ordopspacing:D</code> | 1052 |
| <code>\utex_fractiondenomvgap:D</code> | 1011 | <code>\utex_ordordspacing:D</code> | 1053 |
| <code>\utex_fractionnumup:D</code> | 1012 | <code>\utex_ordpunctspacing:D</code> | 1054 |
| <code>\utex_fractionnumvgap:D</code> | 1013 | <code>\utex_ordrelspacing:D</code> | 1055 |
| <code>\utex_fractionrule:D</code> | 1014 | <code>\utex_overbarkern:D</code> | 1056 |
| <code>\utex_hextensible:D</code> | 980 | <code>\utex_overbarrule:D</code> | 1057 |
| <code>\utex_innerbinspacing:D</code> | 1015 | <code>\utex_overbarvgap:D</code> | 1058 |
| <code>\utex_innerclosespacing:D</code> | 1016 | <code>\utex_overdelimiter:D</code> | 1104 |
| <code>\utex_innerinnerspacing:D</code> | 1017 | <code>\utex_overdelimiterbgap:D</code> | 1059 |
| <code>\utex_inneropenspacing:D</code> | 1018 | <code>\utex_overdelimitervgap:D</code> | 1060 |
| <code>\utex_inneropspacing:D</code> | 1019 | <code>\utex_punctbinspacing:D</code> | 1061 |
| <code>\utex_innerordspacing:D</code> | 1020 | <code>\utex_punctclosespacing:D</code> | 1062 |
| <code>\utex_innerpunctspacing:D</code> | 1021 | <code>\utex_punctinnerspacing:D</code> | 1063 |
| <code>\utex_innerrelspacing:D</code> | 1022 | <code>\utex_punctopenspacing:D</code> | 1064 |
| <code>\utex_limitabovebgap:D</code> | 1023 | <code>\utex_punctopspacing:D</code> | 1065 |
| <code>\utex_limitabovekern:D</code> | 1024 | <code>\utex_punctordspacing:D</code> | 1066 |
| <code>\utex_limitabovevgap:D</code> | 1025 | <code>\utex_punctpunctspacing:D</code> | 1067 |
| <code>\utex_limitbelowbgap:D</code> | 1026 | <code>\utex_punctrelspacing:D</code> | 1068 |
| <code>\utex_limitbelowkern:D</code> | 1027 | <code>\utex_quad:D</code> | 1069 |
| <code>\utex_limitbelowvgap:D</code> | 1028 | <code>\utex_radical:D</code> | 1105 |
| <code>\utex_mathaccent:D</code> | 981, 1248 | <code>\utex_radicaldegreeafter:D</code> | 1070 |
| <code>\utex_mathaxis:D</code> | 982 | <code>\utex_radicaldegreebefore:D</code> | 1071 |
| <code>\utex_mathchar:D</code> | 991, 1249 | <code>\utex_radicaldegreeraise:D</code> | 1072 |
| <code>\utex_mathcharclass:D</code> | 992 | <code>\utex_radicalkern:D</code> | 1073 |
| <code>\utex_mathchardef:D</code> | 993, 1250 | <code>\utex_radicalrule:D</code> | 1074 |
| <code>\utex_mathcharfam:D</code> | 994 | <code>\utex_radicalvgap:D</code> | 1075 |
| <code>\utex_mathcharnum:D</code> | 995, 1251 | <code>\utex_relbinspacing:D</code> | 1076 |
| <code>\utex_mathcharnumdef:D</code> | 996, 1252 | <code>\utex_relclosespacing:D</code> | 1077 |
| <code>\utex_mathcharslot:D</code> | 997 | <code>\utex_relinnerspacing:D</code> | 1078 |
| <code>\utex_mathcode:D</code> | 1006, 1253 | <code>\utex_reloppspacing:D</code> | 1079 |
| <code>\utex_mathcodenum:D</code> | 1007, 1254 | <code>\utex_relopspacing:D</code> | 1080 |
| <code>\utex_nolimitsubfactor:D</code> | 1029 | <code>\utex_relordspacing:D</code> | 1081 |
| <code>\utex_nolimitsupfactor:D</code> | 1030 | <code>\utex_relpunctspacing:D</code> | 1082 |
| <code>\utex_opbinspacing:D</code> | 1031 | <code>\utex_relrelspacing:D</code> | 1083 |
| <code>\utex_opclosespacing:D</code> | 1032 | <code>\utex_root:D</code> | 1106 |
| <code>\utex_openbinspacing:D</code> | 1033 | <code>\utex_skewed:D</code> | 1107 |
| <code>\utex_openclosespacing:D</code> | 1034 | <code>\utex_skewedfractionhgap:D</code> | 1084 |
| <code>\utex_openinnerspacing:D</code> | 1035 | <code>\utex_skewedfractionvgap:D</code> | 1085 |

WV

v

[illegible]