Guide to pTEX for developers unfamiliar with Japanese

Japanese TEX Development Community*

version p3.8.2, November 26, 2019

pTEX and its variants, upTEX, ε-pTEX and ε-upTEX, are all TEX engines with native Japanese support. Its output is always a DVI file, which can be processed by several DVI drivers with Japanese support including *dvips* and *dvipdfmx*. Formats based on LATEX is called pLATEX when running on pTEX/ε-pTEX, and called upLATEX when running on upTEX/ε-upTEX.

Purpose of this document

This document is written for developers of TEX/LATEX, who aim to support pTEX/pLATEX and its variants upTEX/upLATEX. Knowledge of the followings are assumed:

- Basic knowledge of Western TEX (Knuthian TEX, ε-TEX and pdfTEX),
- ... and its programming conventions.

Any knowledge of Japanese (characters, encodings, typesetting conventions etc.) is not assumed; some explanations are provided in this document when needed. We hope that this document helps authors of packages or classes to proceed with supporting pTEX family smoothly.

> Note: This English edition (ptex-guide-en.pdf) is *not* meant to be a complete translation of Japanese edition (ptex-manual.pdf). For example, this document does not cover the following aspects of pTEX:
>
> - Typesetting conventions of Japanese characters
> - Details of vertical writing
>
> For beginners of writing Japanese texts, please refer to the Japanese edition.

*`https://texjp.org`, e-mail: `issue(at)texjp.org`

Contents

Part I
Brief introduction

1 pTEX and its variants

There is no advantage to choose pTEX/upTEX over ε-pTEX/ε-upTEX, so we focus mainly on ε-pTEX/ε-upTEX.

2 Compatibility with Western TEX

pTEX/upTEX are almost upward compatible with Knuthian TEX, however, they do not pass the TRIP test. In pTEX/upTEX, input handling is different from Knuthian TEX; if a pair of two or more 8-bit codes matches Japanese character code, it is regarded as one Japanese character. There is no difference in handling 8-bit TFM font.

ε-pTEX/ε-upTEX are almost upward compatible with ε-TEX, however, input handling is similar to pTEX/upTEX. It does not pass the e-TRIP test. That said, please note that "raw ε-TEX" is unavailable anymore in TEX Live and derived distributions; they provide a command `etex` only as "DVI mode of pdfTEX." We should note that ε-pTEX/ε-upTEX are *not* upward compatible with DVI mode of pdfTEX, which will be discussed later in section 6.7.

3 LATEX on pTEX/upTEX — pLATEX/upLATEX

Formats based on LATEX is called pLATEX when running on pTEX/ε-pTEX, and called upLATEX when running on upTEX/ε-upTEX. In recent versions (around 2011) of TEX Live and its derivatives, the default engines of pLATEX and upLATEX are ε-pTEX and ε-upTEX. That is, the command `platex` starts ε-pTEX (not pTEX) with preloaded format `platex.fmt`.

In the kernel level (`platex.ltx` and `uplatex.ltx`), pLATEX and upLATEX adds some additional commands related to the followings:

- Selection of Japanese fonts
- Crop marks (called "tombow") for printings
- Adjustment for mixing horizontal and vertical texts

For authors, pLATEX/upLATEX are almost upward compatible with original LATEX, except for the followings:

- Order of float objects; in pLATEX/upLATEX, ⟨*bottom float*⟩ is placed above ⟨*footnote*⟩. That is, the complete order is ⟨*top float*⟩ → ⟨*body text*⟩ → ⟨*bottom float*⟩ → ⟨*footnote*⟩.

For developers, additional care may be needed, for changes in the kernel macros and/or absence of pdfTEX features.

4 Eminent characteristics of pTEX/upTEX

The most important characteristics of pTEX/upTEX can be summarized as follows:

- Japanese characters are interpreted and handled completely apart from Western characters.
- Texts can be aligned vertically, called “tate-gumi” (縦組). The horizontal alignment of texts is called “yoko-gumi” (横組), and both “tate-gumi” and “yoko-gumi” can be mixed even within a single document.

Part II
Details

5 Output format — DVI

The output of pTEX family is always a DVI file. This is in contrast to the mainstream of pdfTEX in the Western TEX world.

In case you are not familiar with DVI output processing, first we give some general notice on how to get a "correct" output using LATEX in DVI mode.

- The DVI format is, as its name suggests, inherently driver-independent. However, some LATEX packages (graphicx, color, hyperref etc.) embed some `\special` commands into the DVI, which can be interpreted later by some specific DVI driver. Such a DVI is no longer driver-independent, thus those are called driver-dependent packages.
- In almost all major TEX distributions (of course including TEX Live), the default DVI driver is set to `dvips`. When you choose to process the resulting DVI file with a driver other than dvips (e.g. dvipdfmx) after running LATEX, you need to pass a proper driver option (e.g. `[dvipdfmx]`) to all driver-dependent packages.

Now, let's move on to the situation in Japan, which is slightly complicated due to historical reasons but may also apply to other countries:

- There are two major conventions to pass a proper driver option to all driver-dependent packages:

  1. To give a driver option to each driver-dependent package:

     ```
     \documentclass{article}
     \usepackage[dvipdfmx]{graphicx}
     \usepackage[dvipdfmx]{color}
     ```

  2. To have a driver option as global:

     ```
     \documentclass[dvipdfmx]{article}
     \usepackage{graphicx}
     \usepackage{color}
     ```

  The former convention has been used for many years since 1990s when the number of driver-dependent packages was limited. But in recent years (around 2010–), there are much more driver-dependent packages available. Thus we (Japanese TEX experts) advise

a global driver option rather than individual package options for simplicity, but not yet fully widespread.[1]

- Many people still see driver options as "optional"; they do without driver options unless really needed. For example, the convention of having a global driver option does no harm even when no driver-dependent package is used, but some users choose to omit a driver option to avoid a warning:

```
LaTeX Warning: Unused global option(s):
    [dvipdfmx].
```

5.1 Extensions of DVI format in pTEX family

The DVI format output by pTEX family is fully compatible with Knuthian TEX, as long as the following conditions are met:

- No Japanese characters are typeset.
- There is no portion of vertical text alignment.

However, some additional DVI commands, which are defined in the standard [1] but never used in TEX82, can come out.

- `set2` (129), `put2` (134): Appears in both pTEX and upTEX DVI. Used to typeset a Japanese character with 2-byte code.
- `set3` (130), `put3` (135): Appears in only upTEX DVI. Used to typeset a Japanese character with 3-byte code.

When pTEX is going to typeset a Japanese character into DVI, it is encoded in JIS, which is always a 2-byte code. For this purpose, `set2` or `put2` are used. When upTEX is going to output a Japanese character into DVI, it is encoded in UTF-32. If the code is equal to or less than `U+FFFF`, the lower 16-bit is used with `set2` or `put2`. If the code is equal to or greater than `U+10000`, the lower 24-bit is used with `set3` or `put3`.

In addition, pTEX/upTEX defines one additional DVI command.

- `dir` (255): Used to change directions of text alignment.

The DVI format in the preamble is always set to 2, as with TEX82. On the other hand, the DVI ID in the postamble can be special. Normally it is set to 2, as with TEX82; however, when `dir` (255) appears at least once in a single pTEX/upTEX DVI, the `post_post` table of postamble contains ID = 3.

[1] The fact that there had been a mismatch in option names (`[dvipdfm]` vs. `[dvipdfmx]`) between packages may also have been part of it; geometry did not understand `[dvipdfmx]` option until 2018!

5.2 DVI drivers with Japanese support

There are some DVI drivers with Japanese support. The most eminent drivers are *dvips* and *dvipdfmx*. Nowadays most of casual Japanese users are using *dvipdfmx* as a DVI driver. On the other hand, users of *dvips* are unignorable, especially those working in publishing industry. In recent years, most of major driver-dependent packages support both two drivers.

5.2.1 Using *dvipdfmx*

A DVI file which is output by pTEX can be converted directly to a PDF file using dvipdfmx.

5.2.2 Using *dvips*

A DVI file which is output by pTEX can be converted to a PostScript file using dvips.

The resulting PostScript file can then be converted to a PDF file using Ghostscript (ps2pdf) or Adobe Distiller. When using Ghostscript, a proper setup of Japanese font must be done before converting PostScript into PDF. An easy solution for the setup is a script `cjk-gs-integrate` developed by Japanese TEX Development Community.

6 Programming on pTEX family

We focus on programming aspects of pTEX and its variants.

6.1 Number of registers and marks

pTEX and upTEX have exactly the same number (= 256) of registers (count, dimen, skip, muskip, box, and token) as Knuthian TEX. ε-pTEX and ε-upTEX in extended mode have more registers; there are 65536, which is twice as many as 32768 of ε-TEX. Similarly ε-pTEX and ε-upTEX have 65536 mark classes, which is twice as many as 32768 of ε-TEX.

The following code presents an example of detecting the number of regsiters and mark classes available:

```
\ifx\eTeXversion\undefined
  % Knuthian TeX, pTeX, upTeX:
  %   256 registers, 1 mark
\else
  \ifx\omathchar\undefined
    % e-TeX, pdfTeX (in extended mode):
    %   32768 registers, 32768 mark classes
  \else
    % e-pTeX, e-upTeX (in extended mode):
```

```
    %   65536 registers, 65536 mark classes
  \fi
\fi
```

Here a primitive `\omathchar`, which is derived from Ω, is used as a marker of a change file `fam256.ch`.[2]

6.2 Number of math families

In pTEX and upTEX, the number of math fonts is restricted to 16, each of which can contain 256 characters (same as Knuthian TEX). In ε-pTEX and ε-upTEX, a change file `fam256.ch`, which is derived from Ω, extends the upper limit to 256. As a consequence, ε-pTEX and ε-upTEX allows 256 math fonts, each of which can contain 256 characters.[3]

For pLATEX/upLATEX users to use more than 16 math fonts, it is necessary to use macros which exploit Ω-derived primitives such as `\omathchar`. Recent (u)pLATEX (since 2016/11/29) partially supports this, and the maximum number of math alphabets that can be defined by `\DeclareMathAlphabet` is extended to 256 (`\e@mathgroup@top`) without needing any extension package. However, symbol fonts are restricted to 16 as `\DeclareMathSymbol` etc still use the standard `\mathchar` etc. A simple solution to use more symbol fonts as well as math alphabets is to load a package mathfam256[4] though it's still preliminary.

6.3 Additional primitives and keywords

Here we provide only complete lists of additional primitives of pTEX family in alphabetical order. The features of each primitive can be found in Japanese edition.

6.3.1 SyncTEX additions (available in pTEX, upTEX, ε-pTEX, ε-upTEX)

In the standard build of TEX Live, SyncTEX extension is unavailable in Knuthian TEX; however, it is enabled in pTEX family.

- `\synctex`

6.3.2 pTEX additions (available in pTEX, upTEX, ε-pTEX, ε-upTEX)

- `\autospacing`

[2] There is another pTEX-derived engine named pTEX-ng (or Asiatic pTEX) `https://github.com/clerkma/ptex-ng`; it is based on ε-TEX and upTEX, but currently does not adopt `fam256.ch` so it has the same number of registers and mark classes as ε-TEX.

[3] Ω allows 256 math fonts, each of which can contain 65536 characters.

[4] `https://www.ctan.org/pkg/mathfam256`

- ▶ \autoxspacing
- ▶ \disinhibitglue — New primitive since p3.8.2 (TEX Live 2019)
- ▶ \dtou
- ▶ \euc
- ▶ \ifdbox — New primitive since p3.2 (TEX Live 2011)
- ▶ \ifddir — New primitive since p3.2 (TEX Live 2011)
- ▶ \ifmbox — New primitive since p3.7.1 (TEX Live 2017)
- ▶ \ifmdir
- ▶ \iftbox
- ▶ \iftdir
- ▶ \ifybox
- ▶ \ifydir
- ▶ \inhibitglue
- ▶ \inhibitxspcode
- ▶ \jcharwidowpenalty
- ▶ \jfam
- ▶ \jfont
- ▶ \jis
- ▶ \kanjiskip
- ▶ \kansuji
- ▶ \kansujichar
- ▶ \kcatcode
- ▶ \kuten
- ▶ \noautospacing
- ▶ \noautoxspacing
- ▶ \postbreakpenalty
- ▶ \prebreakpenalty
- ▶ \ptexminorversion — New primitive since p3.8.0 (TEX Live 2018)
- ▶ \ptexrevision — New primitive since p3.8.0 (TEX Live 2018)
- ▶ \ptexversion — New primitive since p3.8.0 (TEX Live 2018)

- ▶ \scriptbaselineshiftfactor — New primitive since p3.7 (TEX Live 2016)
- ▶ \scriptscriptbaselineshiftfactor — New primitive since p3.7 (TEX Live 2016)
- ▶ \showmode
- ▶ \sjis
- ▶ \tate
- ▶ \tbaselineshift
- ▶ \textbaselineshiftfactor — New primitive since p3.7 (TEX Live 2016)
- ▶ \tfont
- ▶ \xkanjiskip
- ▶ \xspcode
- ▶ \ybaselineshift
- ▶ \yoko
- ▶ H
- ▶ Q
- ▶ zh
- ▶ zw

6.3.3 upTEX additions (available in upTEX, ε-upTEX)

- ▶ \disablecjktoken
- ▶ \enablecjktoken
- ▶ \forcecjktoken
- ▶ \kchar
- ▶ \kchardef
- ▶ \ucs
- ▶ \uptexrevision — New primitive since u1.23 (TEX Live 2018)
- ▶ \uptexversion — New primitive since u1.23 (TEX Live 2018)

6.3.4 ε-pTEX additions (available in ε-pTEX, ε-upTEX)

- ▶ \currentspacingmode — New primitive since 191112 (TEX Live 2020)
- ▶ \currentxspacingmode — New primitive since 191112 (TEX Live 2020)
- ▶ \epTeXinputencoding — New primitive since 160201 (TEX Live 2016)

- ▶ \epTeXversion — New primitive since 180121 (TEX Live 2018)
- ▶ \expanded — New primitive since 180518 (TEX Live 2019)
- ▶ \hfi
- ▶ \ifincsname — New primitive since 190709 (TEX Live 2020)
- ▶ \ifpdfprimitive — New primitive since 150805 (TEX Live 2016)
- ▶ \lastnodechar — New primitive since 141108 (TEX Live 2015)
- ▶ \lastnodesubtype — New primitive since 180226 (TEX Live 2018)
- ▶ \odelcode
- ▶ \odelimiter
- ▶ \omathaccent
- ▶ \omathchar
- ▶ \omathchardef
- ▶ \omathcode
- ▶ \oradical
- ▶ \pagefistretch
- ▶ \pdfcreationdate — New primitive since 130605 (TEX Live 2014)
- ▶ \pdfelapsedtime — New primitive since 161114 (TEX Live 2017)
- ▶ \pdffiledump — New primitive since 140506 (TEX Live 2015)
- ▶ \pdffilemoddate — New primitive since 130605 (TEX Live 2014)
- ▶ \pdffilesize — New primitive since 130605 (TEX Live 2014)
- ▶ \pdflastxpos
- ▶ \pdflastypos
- ▶ \pdfmdfivesum — New primitive since 150702 (TEX Live 2016)
- ▶ \pdfnormaldeviate — New primitive since 161114 (TEX Live 2017)
- ▶ \pdfpageheight
- ▶ \pdfpagewidth
- ▶ \pdfprimitive — New primitive since 150805 (TEX Live 2016)
- ▶ \pdfrandomseed — New primitive since 161114 (TEX Live 2017)
- ▶ \pdfresettimer — New primitive since 161114 (TEX Live 2017)
- ▶ \pdfsavepos

- ▶ `\pdfsetrandomseed` — New primitive since 161114 (TeX Live 2017)
- ▶ `\pdfshellescape` — New primitive since 141108 (TeX Live 2015)
- ▶ `\pdfstrcmp`
- ▶ `\pdfuniformdeviate` — New primitive since 161114 (TeX Live 2017)
- ▶ `\readpapersizespecial` — New primitive since 180901 (TeX Live 2019)
- ▶ `\Uchar` — New primitive since 191112 (TeX Live 2020)
- ▶ `\Ucharcat` — New primitive since 191112 (TeX Live 2020)
- ▶ `\vfi`
- ▶ `fi`

6.3.5 ε-upTeX additions (available in ε-upTeX)

- ▶ `\currentcjktoken` — New primitive since 191112 (TeX Live 2020)

6.4 Omitted primitives and unsupported features

Compared to Knuthian TeX and ε-TeX, some primitives are omitted due to conflict with Japanese handling. One is encTeX extension, such as `\mubyte`. The MLTeX extension, such as `\charsubdef`, is included but not well-tested.

6.5 Behavior of Western TeX primitives

Here we provide some notes on behavior of Knuthian TeX and ε-TeX primitives when used within pTeX family.

6.5.1 Primitives with limitations in handling Japanese

Each of the following primitives allows only character codes 0–255; other codes will give an error "! Bad character code."

`\catcode`, `\sfcode`, `\mathcode`, `\delcode`, `\lccode`, `\uccode`.

Each of the following primivies has `\...char` in its name, however, the effective values are restricted to 0–255.

`\endlinechar`, `\newlinechar`, `\escapechar`, `\defaulthyphenchar`, `\defaultskewchar`.

6.5.2 Primitives capable of handling Japanese

The following primitives are extended to support Japanese characters:

- `\char` ⟨*character code*⟩, `\chardef` ⟨*control sequence*⟩=⟨*character code*⟩

  In addition to 0–255, internal codes of Japanese characters are allowed. For putting Japanese characters, a Japanese font is chosen. More information can be found in 6.6.3.

- `\font`, `\fontname`, `\fontdimen`

- `\accent` ⟨*character code*⟩=⟨*character*⟩

- `\if` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩, `\ifcat` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩

  Japanese character token is also allowed. In that case,

  - `\if` tests the internal character code of the Japanese character.
  - `\ifcat` tests the `\kcatcode` of the Japanese character.

TEXbook describes the behavior of `\if` and `\ifcat` as follows;

> If either token is a control sequence, TEX considers it to have character code 256 and category code 16, unless the current equivalent of that control sequence has been `\let` equal to a non-active character token.

However, this includes a lie; in the real implementation of `tex.web`, a control sequence is considered to have a category code 0.

6.6 Case study

Here we provide some code examples which may be useful for package developers.

6.6.1 Detecting pTEX

Since the primitive `\ptexversion` is rather new (added in 2018), the safer solution for detecting pTEX is to test if a primitive `\kanjiskip` is defined.

```
\ifx\kanjiskip\undefined
\else
  % pTeX / upTeX / e-pTeX / e-upTeX
\fi
```

6.6.2 Detecting upTEX

upTEX/ε-upTEX are almost upward compatible with pTEX/ε-pTEX respectively, however, there are two major differences:

1. Improvements in the `\kcatcode` business, mainly for better handling of Latin-1 characters and CJK tokens.
2. Unicode as the default internal kanji encoding, for direct use of its huge character set.

The first difference can be detected by checking if `\...cjktoken` primitive is defined.

```
\ifx\enablecjktoken\undefined
\else
  % upTeX/e-upTeX
\fi
```

The second difference can be detected by checking if the character `0x2121` (fullwidth space in JIS encoding) is stored as `"3000` internally.

```
\ifx\kanjiskip\undefined
\else
  \ifnum\jis"2121="3000
    % upTeX/e-upTeX with internal Unicode
  \else
    % pTeX/e-pTeX
    % or, upTeX/e-upTeX with internal EUC-JP or Shift-JIS
  \fi
\fi
```

Please note that the format-build setting of `-kanji-internal=(sjis|euc)` with upTEX makes it effectively pTEX regarding the character set, which means that only JIS X 0208 character set is supported.

6.6.3 Defining large integer constants

According to [2] (Section 3.3),

> A control sequence that has been defined with a `\chardef` command can also be used as a ⟨*number*⟩. This fact is used in allocation commands such as `\newbox`. Tokens defined with `\mathchardef` can also be used this way.

Here is the list of primitives which can be used for this purpose in pTEX family:

► `\chardef` ⟨*control sequence*⟩=⟨*character code*⟩

Defines a control sequence to be a synonym for `\char` ⟨*character code*⟩.

► `\kchardef` ⟨*control sequence*⟩=⟨*character code*⟩ (for upTEX/ε-upTEX)

Defines a control sequence to be a synonym for `\kchar` ⟨*character code*⟩.

► `\mathchardef` ⟨*control sequence*⟩=⟨*15-bit number*⟩

Defines a control sequence to be a synonym for `\mathchar` ⟨*15-bit number*⟩.

► `\omathchardef` ⟨*control sequence*⟩=⟨*27-bit number*⟩ (for ε-pTEX/ε-upTEX)

Defines a control sequence to be a synonym for `\omathchar` ⟨*27-bit number*⟩.

The first two (`\chardef` and `\kchardef`) are usable only when the integer being defined is in the range of valid character codes, which is not necessarily continuous (see 8.1). The most efficient and convenient way of defining integer constants is as follows:

- 0–255: `\chardef`
- 256–32767: `\mathchardef`
- 32768–134217727: `\omathchardef` (only for ε-pTEX/ε-upTEX)
- (optional) 256–2147483647: `\chardef` (only for upTEX/ε-upTEX)

6.6.4 Creating a Japanese character token with a specified code

Short version:

- With ε-pTEX 191112 or later (TEX Live 2020), you can use expandable primitives `\Uchar` and `\Ucharcat`
- Otherwise, use the "`\kansuji` trick".

■ *The "`\kansuji` trick"*

This is a modified version of the "`\lowercase` trick" available in pTEX family.

Short note on the "`\lowercase` trick": to create a character token with a specified code value between 0–255 with Knuthian TEX, the "`\lowercase` trick" can be used; for example,

```
\begingroup
\lccode`\?=\mycount
\lowercase{\endgroup \def\X{?}}
```

defines \X which expands to a character number \mycount while the \catcode of ? (12) is preserved. However, the trick cannot be applied to Japanese characters, since pTEX family does not support \lccode outside 0–255.

\kansuji is an expandable primitive like \number or \romannumeral, and it converts an integer into its corresponding kanji notation called "kansuji" (漢数字). The important point here is that the number-kanji mapping can be altered by \kansujichar.

Example 1: equivalent to \def\X{あ} (JIS code 0x2422 is "あ"):

```
\begingroup
  \kansujichar1=\jis"2422 \xdef\X{\kansuji1}
\endgroup
```

Example 2: equivalent to \def\日本{Japan}.

```
\begingroup
  \kansujichar5=\jis"467C\relax
  \kansujichar6=\jis"4B5C\relax
  \expandafter\gdef\csname\kansuji56\endcsname{Japan}
\endgroup
```

Since \kansujichar accepts only Japanese character code, the "\kansuji trick" and the "\lowercase trick" should be used complementarily.

■ *\Uchar, \Ucharcat*

The "\kansuji trick" above include an assignment of \kansujichar which is unexpandable. ε-pTEX 191112 or later (TEX Live 2020) provides expandable primitives \Uchar and \Ucharcat, which are derived from XETEX.

► \Uchar ⟨*character code*⟩

Expands to a character token with specified slot ⟨*character code*⟩.

- When an 8-bit number (0–255) is given, it expands to a Latin character token with category code 12, except for a space character (32) which has category code 10.
- When a Japanese character code greater than 255 is given, it expands to a Japanese character token with its current category code; 16–18 for ε-pTEX, 16–19 for ε-upTEX.

► \Ucharcat ⟨*character code*⟩ ⟨*category code*⟩

Expands to a character token with slot ⟨*character code*⟩ and ⟨*category code*⟩ specified.

- With ε-pTEX:
  - Only 8-bit number (0–255) are allowed for ⟨*character code*⟩; that is, only Latin characters can be generated.

  - The values allowed for ⟨*category code*⟩ are 1–4, 6–8, 10–13.
- With ε-upTEX:
  - When ⟨*character code*⟩ is between 0–127, only Latin characters can be generated. Thus, the values allowed for ⟨*category code*⟩ are 1–4, 6–8, 10–13.
  - When ⟨*character code*⟩ is between 128–255, both Latin and Japanese characters can be generated depending on the specified ⟨*category code*⟩; 1–4, 6–8, 10–13: Latin character, 16–19: Japanese character.
  - When ⟨*character code*⟩ is greater than 255, only Japanese characters can be generated. Thus, the values allowed for ⟨*category code*⟩ are 16–19.

6.7 Difference from pdfTEX in DVI mode

As stated above, ε-pTEX/ε-upTEX are *not* upward compatible with DVI mode of pdfTEX.

6.8 Recommendation for file encoding

6.9 Input handling

For simplicity, first we introduce of input handling of ε-upTEX.

6.10 Japanese tokens

7 Basic introduction to Japanese typesetting

This section does not aim to explain Japanese typesetting completely; here we provide a minimum requirement for "getting away" with Japanese.

7.1 Automatic insertion of glue and penalties

Sometimes pTEX family automatically inserts glue and penalties between characters.

7.2 Japanese fonts

pTEX family can have 3 different "current" fonts at the same time; a Latin font, a Japanese font for horizontal writing ("yoko-gumi"), and a Japanese font for vertical writing ("tate-gumi"). The first one is the same as in the Knuthian TEX, which is defined in a standard TFM format. The latter two are specific to pTEX family, which are defined in a JFM (Japanese TEX font metric) format.[5]

[5] A JFM is a modified version of the standard TFM. It can be created by (u)pPLtoTF, and decoded by (u)pTFtoPL. Please also refer to the man pages of these programs (`ppltotf.man1.pdf` and `ptftopl.man1.pdf`).

While typesetting, pTEX family automatically switches between these 3 fonts, depending on the character code and the writing direction:

- For typesetting Latin characters, the current Latin font shown by `\the\font` is selected.
- For typesetting Japanese characters, the current Japanese font suitable for the current writing direction is selected. It is shown by `\the\jfont` for horizontal writing and `\the\tfont` for vertical writing.

In Knuthian TEX, the primitive `\nullfont` refers to an "empty font" in which all characters are undefined. However in pTEX family, this is regarded as a Latin font and there is no equivalent to "Japanese `\nullfont`" by design. To elaborate, it is possible *only* when no Japanese font is set globally, i.e. in iniTEX mode. Once a valid Japanese font is selected, there is no way of selecting "Japanese `\nullfont`" to discard all characters.

Moreover, pTEX and friends assume that each Japanese font (except "Japanese `\nullfont`" in iniTEX mode) contains all valid Japanese character code. In other words, all Japanese fonts share the same character set corresponding to the whole valid Japanese character code range.

8 Other strange beasts

8.1 Internal kanji encodings

The ⟨*character code*⟩ is a union of the following two:

- Range of numbers between 0–255, and
- Numbers allowed for internal code of Japanese characters.

The former is the same as Knuthian TEX, but the latter is a problem. In upTEX/ε-upTEX with `-kanji-internal=uptex` (default on), the range is very simple:

$$c \geq 0$$

However in pTEX/ε-pTEX, only legacy encodings (EUC-JP as `euc`, or Shift-JIS as `sjis`) are available for `-kanji-internal`. In this case, the range can be represented as follows:

$$c = 256c_1 + c_2 \ (c_i \in C_i)$$

where

$$\begin{cases} C_1 = C_2 = \{\texttt{"a1}, \ldots, \texttt{"fe}\} & (\texttt{euc}), \\ C_1 = \{\texttt{"81}, \ldots, \texttt{"9f}\} \cup \{\texttt{"e0}, \ldots, \texttt{"fc}\}, C_2 = \{\texttt{"40}, \ldots, \texttt{"7e}\} \cup \{\texttt{"80}, \ldots, \texttt{"fc}\} & (\texttt{sjis}). \end{cases}$$

Therefore, the overall range of ⟨*character code*⟩ is *not* continuous.

To check whether an integer is a valid Japanese character code or not, you can use `\iffontchar` with ε-pTEX 190709 or later (TEX Live 2020). Suppose a count register `\mycount` stores an integer, you can do it as follows:

```
\iffontchar\jfont\mycount
  % \mycount is a valid Japanese character code
\fi
```

Here the primitive `\jfont` is used merely as a representative non-empty[6] Japanese font containing all valid Japanese character code (see 7.2).

[6] This assumption is always safe after one of the standard pTEX formats (e.g. plain pTEX, pLATEX) is loaded.

References

[1] TUG DVI Standards Working Group, *The DVI Driver Standard, Level 0*. https://ctan.org/pkg/dvistd

[2] Victor Eijkhout, *TEX by Topic, A TEXnician's Reference*, Addison-Wesley, 1992. https://www.eijkhout.net/texbytopic/texbytopic.html

Index

F

H

Q

Z