Distinguishing 8-bit characters and Japanese characters in (u)pTEX

Hironori Kitagawa

Abstract

pTEX (an extension of TEX for Japanese typesetting) uses a legacy encoding as the internal Japanese encoding, while accepting UTF-8 input. This means that pTEX does code conversion in input and output. Also, pTEX (and its Unicode extension upTEX) distinguishes 8-bit character tokens and Japanese character tokens, while this distinction disappears when tokens are processed with `\string` and `\meaning`, or printed to a file or the terminal.

These facts cause several unnatural behaviors with (u)pTEX. For example, pTEX garbles "ſ" (long s) to "顛" on some occasions. This paper explains these unnatural behaviors, and discusses an experiment in improvement by the author.

1 Introduction

Since TEX Live 2018, UTF-8 has been the new default input encoding in LATEX [8]. However, with pLATEX, which is a modified version of LATEX for the pTEX engine, the source

```
%#!platex
\documentclass{minimal}
\begin{document}ſ\end{document} % long s
```

gives an inconsistent error message [4] (edited to fit *TUGboat*'s narrow columns):

```
! Package inputenc Error: Unicode character
    顛 (U+C4CF) not set up for use with LaTeX.
```

Here "顛", "ſ" and `U+C4CF` are all different characters.

The purpose of this paper is to investigate the background of this message and propose patches to resolve this issue. This paper is based on a cancelled talk [6] in TEXConf 2019.[1]

In this paper, the following are assumed:

- All inputs and outputs are encoded in UTF-8.
- pTEX uses EUC-JP as the internal Japanese encoding (see Section 2.1).
- Sources are typeset in plain pTEX (`ptex`), unless stated otherwise by `%#!`.
- The notation <u>`<AB>`</u> describes a byte `0xab`, or a character token whose code is `0xab`.

2 Overview of pTEX

pTEX is an engine extension of TEX82 for Japanese typesetting. It can typeset Japanese documents of professional quality [9], including Japanese line breaking rules and vertical typesetting.

pTEX and pLATEX were originally developed by the ASCII Corporation[2] [1]. However, pTEX and pLATEX in TEX Live, which are our concern, are community editions. These are currently maintained by the Japanese TEX Development Community.[3] For more detail, please see the English guide for pTEX [3].

pTEX itself does not have ε-TEX features, but there is ε-pTEX [7], which merges pTEX, ε-TEX and additional primitives. Anything discussed about pTEX in this paper (besides this paragraph) also applies to ε-pTEX, so I simply write "pTEX" instead of "pTEX and ε-pTEX". Note that the pLATEX format in TEX Live is produced by ε-pTEX, because recent versions of LATEX require ε-TEX features.

2.1 Input code conversion by `ptexenc`

Although pTEX in TEX Live accepts UTF-8 inputs, the internal Japanese character set is limited to JIS X 0208 (JIS level 1 and 2 kanjis), which is a legacy character set before Unicode. pTEX uses Shift_JIS (Windows) or EUC-JP (other) as the internal encoding of JIS X 0208.

In pTEX and related programs, the `ptexenc` library [12] converts an input line to the internal encoding. pTEX's input processor actually reads the converted result by `ptexenc`. A valid UTF-8 sequence which does not represent a JIS X 0208 character — such as <u>`<C5><BF>`</u> ("ſ") or <u>`<C3><9F>`</u> ("ß") — is converted to `^^`-notation, such as `^^ab`.

On the other hand, an invalid UTF-8 sequence is converted into <u>`<A2><AF>`</u> (an undefined code point in EUC-JP) sometimes, in TEX Live 2019 or prior. In TEX Live 2020, the sequence is always converted into `^^`-notation.

2.2 Japanese character tokens

pTEX divides character tokens into two groups: ordinary 8-bit character tokens and Japanese character tokens. The former are not different from tokens in 8-bit engines, say, TEX82 and pdfTEX. A `^^`-notation sequence is always treated as an 8-bit character.

A Japanese character token is represented by its character code. In other words, although there is a `\kcatcode` primitive, which is the counterpart of `\catcode`, its information is *not* stored in tokens. Hence, changing `\kcatcode` by users is not recommended.

[1] TEXConf 2019 (the annual meeting of Japanese TEX users, `texconf2019.tumblr.com`) was canceled due to a typhoon.

[2] Currently ASCII DWANGO in DWANGO Co. Ltd.

[3] `texjp.org/`. Several GitHub repositories: `github.com/texjporg/tex-jp-build` ((u)pTEX), `github.com/texjporg/platex` (pLATEX).

2.3 An example input

Now we look at an example. Our input line is

`a<C3><9F><E6><BC><A2><C5><BF><C2><A7>` (aß漢ſ§)

First, `ptexenc` converts this line into

`a^^c3^^9f<B4><C1>^^c5^^bf<A1><F8>`

which is fed to pTEX's input processor. The final character "§" is included in JIS X 0208.

From the result above, pTEX produces tokens

a$_{11}$ <C3>$_{12}$ <9F>$_{12}$ 漢 <C5>$_{12}$ <BF>$_{12}$ §

where 漢 and § are Japanese character tokens. From this example, we can see that we cannot write "§" directly to output this character in a Latin font (use commands or `^^c2^^a7`).

3 Stringization in pTEX

3.1 Overview

Names of multiletter control sequences, which include control sequences with single Japanese character name, such as `\あ`, are stringized, that is to say, they are stored into the string pool. Similarly, some primitives, such as `\string`, `\jobname`, `\meaning` and `\the` (almost always the case), first stringize their intermediate results into the string pool, and then retokenize these intermediate results.

Stringization of pTEX has two crucial points.

- The origin of a byte is lost in stringization. A byte sequence, for example <C5><BF>, in the string pool may be the result of stringization of a Japanese character "頗", or that of two 8-bit characters <C5> and <BF>.
- In retokenization, a byte sequence which represents a Japanese character in the internal encoding is *always* converted to a Japanese character token. For example, <C5><BF> is always converted to a Japanese token 頗.

These points cause unnatural behavior, namely bytes from 8-bit characters becoming garbled to Japanese character tokens. We look into several examples.

3.2 Control sequence name

Let's begin with the following source:

```
\font\Z=ec-lmr10 \Z % T1 encoding
\expandafter\def\csname uſ\endcsname{AA}
\expandafter\def\csname u頗\endcsname{BB}
\def\ZZ#1{#1 (\string#1) }
\expandafter\ZZ\csname u^^c5^^bf\endcsname% (1)
\expandafter\ZZ\csname uſ\endcsname       % (2)
\expandafter\ZZ\csname u頗\endcsname      % (3)
```

With pTEX, (1)–(3) produces the same result

BB (\u 頗)

This is because all of

```
\csname u^^c5^^bf\endcsname
\csname uſ\endcsname  % ſ:  <C5><BF> in UTF-8
\csname u頗\endcsname % 頗: <C5><BF> in EUC-JP
```

have the same name u<C5><BF> in pTEX, hence they are treated as the same control sequence. Applying `\string` to them, we get the same token list

\\$_{12}$ u$_{12}$ 頗

This explains the error message in the introduction. "頗 (U+C4CF)" in the message is generated from

```
\expandafter\string
\csname u8:\string<C5>\string<BF>\endcsname
```

The inputenc package expects that applying `\string` to the above control sequence produces

\\$_{12}$ u$_{12}$ 8$_{12}$:$_{12}$ <C5>$_{12}$ <BF>$_{12}$

but the result in pLATEX is

\\$_{12}$ u$_{12}$ 8$_{12}$:$_{12}$ 頗

3.3 `\meaning`

The result of

```
\font\Z=ec-lmr10 \Z % T1 encoding
\def\fuga{^^c5^^bf頗ſ}\meaning\fuga
```

differs between plain TEX and plain pTEX:

plain TEX `macro:->`Å£éąŻÅ£

plain pTEX `macro:->頗頗頗`

Now we look at what happened with pTEX. The definition of `\fuga` is represented by the token list

<C5>$_{12}$ <BF>$_{12}$ 頗 <C5>$_{12}$ <BF>$_{12}$

This gives the following string as the intermediate result of `\meaning`.

`macro:-><C5><BF><C5><BF><C5><BF>`

Retokenizing this string gives the final result

`macro:->頗頗頗`

which we have already seen.

3.4 A tricky application

The behavior described in Section 3.2 has a tricky application: generating a Japanese character token from its code number, even in an expansion-only context. This can be constructed as follows:

```
%#!eptex
\font\Z=ec-lmr10 \Z  % T1 encoding
\input expl3-generic % for \char_generate:nn
\ExplSyntaxOn
\cs_generate_variant:Nn \cs_to_str:N { c }
```

```
\cs_new:Npn \tkchar #1 {
 \cs_to_str:c {
  \char_generate:nn % upper byte
   { \int_div_truncate:nn { #1 } { 256 } }
   { 12 }
  \char_generate:nn % lower byte
   { \int_mod:nn { #1 } { 256 } } { 12 }
 }
}
\ExplSyntaxOff
\edef\A{\tkchar{`漢}\tkchar{`字}}
\meaning\A % ==> macro:->漢字
```

This `\tkchar` will be unnecessary as of TeX Live 2020, since the `\Uchar` and `\Ucharcat` primitives were added into ε-pTeX at that time.

4 Output to file or terminal

4.1 Output code conversion

As with input, pTeX does a code conversion from the internal Japanese encoding to UTF-8 in outputting to a file or the terminal. This is done in two steps:

- As with TeX82, pTeX uses the *print* procedure for printing a string.[4] In pTeX, a byte is printable if and only if its value is between 32 ("␣") and 126 ("~"), or it is used in the internal Japanese encoding (<A1>–<FE> in EUC-JP).
- pTeX uses the *putc2* function instead of the standard *putc* C function. *putc2* is a variation of *putc* with code conversion, and is defined in `ptexenc`.

Hence pTeX may garble 8-bit characters, such as <C5><BF>, into a Japanese character in output. We look into two examples, one is of `\write` and the other is of `\message`.

4.2 `\write`

With pTeX, the following source

```
\newwrite\OUT
\immediate\openout\OUT=test.dat
\immediate\write\OUT{顛fß}
\immediate\closeout\OUT
```

produces a file `test.dat`, whose contents are

`顛顛<C3>^^9f`

Let's look at what happened.

First, the argument of `\write` is (expanded to) the following token list.

顛 <C5>$_{12}$ <BF>$_{12}$ <C3>$_{12}$ <9F>$_{12}$

[4] In fact, *slow_print* is used for printing a string which might contain unprintable characters. However, *slow_print* calls *print* internally.

Then, pTeX prints this token list. Since <A1>–<FE> are printable and <9F> is not, the *putc2* function receives the following string, one byte per call.

`<C5><BF><C5><BF><C3>^^9f`

Each <C5><BF> is converted to "顛" by *putc2*, while the single <C3> remains unchanged. Hence the final result is "顛顛<C3>^^9f", as shown.

4.3 `\message`

`\message` is similar to `\write`, but differs in that it stringizes its argument. Now consider an input line

`\message{^^fe^^f3:𪚲:}`

Here 𪚲 (<F0><AA><9A><B2> in UTF-8) is a character included in JIS X 0213, but not in JIS X 0208.

The argument of `\message` is (expanded to) the following token list.

<FE>$_{12}$ <F3>$_{12}$:$_{12}$ <F0>$_{12}$ <AA>$_{12}$ <9A>$_{12}$ <B2>$_{12}$:$_{12}$

Then, this token list is stringized to

`<FE><F3>:<F0><AA><9A><B2>:`

This string is "printed" by *print*; since only <9A> is unprintable, *putc2* receives

`<FE><F3>:<F0><AA>^^9a<B2>:`

Now, *putc2* converts <FE><F3> (an undefined code point in EUC-JP) to the null character <00>, and <F0><AA> to "險". Hence the final result is

`<00>:險^^9a<B2>:`

4.4 Controlling printability

TeX82 and pdfTeX support TCX (TeX Character Translation) files [2], which can be used to specify which characters are printable. In fact, `cp227.tcx` is activated in (pdf)LaTeX and several other formats in TeX Live, to make characters 128–255 and three control characters printable. One can switch to a different TCX file at runtime. For example, only characters 32–126 are printable in

`latex -translate-file=empty.tcx`

However, pTeX was not expected to use TCX files (no TCX files are activated in formats by pTeX in default). inipTeX can make characters printable by a TCX file, and that's all. For example, to make characters 128–255 printable in pTeX, one has to make another format with appropriate option. There is no method to make an arbitrary character, say <A0>, unprintable when using this format.

5 upTEX

5.1 Overview

upTEX [10, 11] is a Unicode extension of pTEX by Takuji Tanaka. upTEX is (almost fully) upward-compatible with pTEX, so it is a very convenient solution for converting existing documents to Unicode with minimal changes.

In upTEX, a Japanese character token is a pair of the character code and `\kcatcode`. Furthermore, `\kcatcode` controls whether a UTF-8 sequence produces a Japanese character token or a sequence of 8-bit tokens. For example, <E9><A1><9B> (顛, U+985B) in an input line is treated as three 8-bit characters when `\kcatcode"985B` is 15, and as a Japanese character otherwise.

5.2 No code conversion

Since upTEX's internal Japanese character code is Unicode (UTF-8 in the string pool), code conversion by `ptexenc` has no effect. Hence the inconsistent error message described in the introduction will not be issued.

5.3 Retokenization and `\kcatcode`

In upTEX, `\kcatcode` is involved in the retokenization process. Specifically, a UTF-8 sequence is converted into a Japanese character token if and only if its `\kcatcode` is not 15. This means that the result of `\meaning` of the same macro depends on `\kcatcode` settings, as in the following example.

```
%#!uptex
\font\Z=ec-lmr10 \Z % T1 encoding
%% default: \kcatcode"3042=17
\def\hoge{^^e3^^81^^82あ}
\kcatcode"3042=15
\meaning\hoge % ==> macro:->ãĄĆãĄĆ
\kcatcode"3042=17
\meaning\hoge % ==> macro:->ああ
```

The definition of `\hoge` is represented by the token list

<E3>$_{12}$ <81>$_{12}$ <82>$_{12}$ あ$_{17}$

Hence the intermediate result of `\meaning\hoge` is

```
macro:-><E3><81><82><E3><81><82>
```

However, because the `\kcatcode` of "あ" is changed, two calls of `\meaning\hoge` give different results.

We will see results of `\string` of multiletter control sequences later.

6 Distinguishing bytes from 8-bit characters and those from Japanese characters

To resolve (u)pTEX's behavior described so far, I have been developing an experimental version[5] of (u)pTEX, where stringization and outputting retain the origin of a byte — an 8-bit character (token) or a Japanese one. I refer to these as "experimental", and (u)pTEX in TEX Live development repository as "trunk".

The implementation approach is to extend the range of a "byte" to 0–511 (Table 1). A value between 0–255 means a byte from an 8-bit character (token), and 256–511 means a "byte" from a Japanese one.

I tested a different approach, namely using <FF> as a prefix to a byte 128–255 which came from an 8-bit character. But this approach caused confusion with <FF>, so I gave up.

6.1 `\write`

For example, consider the source from Section 4.2:

```
\newwrite\OUT
\immediate\openout\OUT=test.dat
\immediate\write\OUT{顛fß}
\immediate\closeout\OUT
```

with the experimental pTEX. When no TCX file is activated, *putc2* receives the string

```
<1C5><1BF>^^c5^^bf^^c3^^9f
```

because a Japanese token 顛 sends <1C5><1BF> to *putc2*, and <80>–<FF> are not printable. Thus the contents of the output `test.dat` are

```
顛^^c5^^bf^^c3^^9f
```

When `cp227.tcx` is activated, they become

```
顛fß
```

because <80>–<FF> are printable in this case.

6.2 The string pool

Since the range of a "byte" is increased to 0–511, the type of the string pool is changed to let each element store a "byte"; concretely, to a 16-bit array. For example, let's reconsider the following source:

```
\font\Z=ec-lmr10 \Z % T1 encoding
\def\fuga{^^c5^^bf顛f}\meaning\fuga
```

With the experimental pTEX, the intermediate result of `\meaning\fuga` is

```
macro:-><C5><BF><1C5><1BF><C5><BF>
```

Hence the result of `\meaning\fuga` is

[5] `github.com/h-kitagawa/tex-jp-build/tree/printkanji_16bit`. GitHub issue: [5]

Table 1: A “byte” in experimental (u)pTEX

“byte” c	**0–255**	**256–511**
origin	an 8-bit character (token)	a Japanese character (token)
printable characters	32–126 (“␣”–“~”)*	all
“safe” printing of c	*print*(c)	*print_char*(c) (not *print*)
putc2(c, ...)	*without* code conversion	with code conversion**
retokenization	an 8-bit character token c	a Japanese character token**

* Web2C’s default; can be extended by a TCX file.
** With adjacent “bytes” which are between 256–511.

```
macro:->Å£ 顛 Å£
```

because only <1C5><1BF> is converted to a Japanese character token 顛.

The change in the type for the string pool increases the size of format files by about the total length of strings, but the amount of increase is not so large. For example, the `platex-dev` format is increased by about 3.5 % (see table below). As of TEX Live 2020, pdfTEX and (u)pTEX use compressed format files, so the amount of increase on disk is smaller.

`platex-dev.fmt` [kB]	trunk	experimental
uncompressed	10412	10774
compressed	2322	2380

I wanted to keep the modification as small and simple as possible; so I left unchanged the structure of the string pool, except for adding a “flag bit”.

6.3 Control sequence names in upTEX

In the experimental pTEX,

```
\csname uf\endcsname
\csname u顛\endcsname
```

are treated as different control sequences. This is because the name of the former is u<C5><BF>, while that of the latter is u<1C5><1BF>. This behavior seems to be natural.

However, the situation is more arguable between the experimental upTEX and the trunk upTEX. For example, let’s compare the results of (1) and (2) in the following source by both versions of upTEX.

```
%#!uptex
\font\Z=ec-lmr10 \Z % T1 encoding
\def\ZZ#1{#1 (\string#1) }
\kcatcode"3042=15
\expandafter\def\csname あ\endcsname{AA}
\kcatcode"3042=17
\expandafter\def\csname あ\endcsname{BB}
\kcatcode"3042=17 \expandafter\ZZ
  \csname あ\endcsname   % (1)
\kcatcode"3042=15 \expandafter\ZZ
  \csname あ\endcsname   % (2)
```

Results are summarized in Table 2. One may feel uneasy about both results.

trunk The results of `\string` for (1) and (2) differ, while they represent the same control sequence (as in Section 5.3).

experimental (1) and (2) represent different control sequences.

6.4 Input buffer(s)

I also introduced an array *buffer2* as a companion array to *buffer*, which contains an input line. *buffer2*[i] plays the role of the “upper byte” of *buffer*[i]. Hence, when (u)pTEX considers a byte sequence *buffer*[i .. j] as a Japanese character, *buffer2*[i .. j] is set to 1. This is needed when scanning a control sequence name in order to distinguish a byte which consists a part of a Japanese character from another byte.

Suppose that the category codes of <C5> and <BF> are both 11 (letter), an input line contains

\\<C5><BF>^^c5^^bf (\顛^^c5^^bf, \顛f) (1)

and pTEX is about to scan this control sequence (1). Since (p)TEX converts ^^-notation in a control sequence name into single characters in *buffer*, the contents of *buffer* become

\\<C5><BF><C5><BF> (\顛顛)

Thus, the control sequence (1) cannot be distinguished from `\顛顛` so far. However, the experimental pTEX can distinguish the control sequence (1) from `\顛顛`, because the contents of *buffer2* differ (see Table 3).

buffer2 is also useful in showing contexts in upTEX. For example, let’s look the following input:

```
%#!uptex
\def\J{\kcatcode"3042=17 }
\def\L{\kcatcode"3042=15 }
\J あ\L あ\undefined あ\J あ
```

Table 2: Properties of `\csname あ\endcsname` of TEX source in Section 6.3

	\kcatcode of "あ"	trunk		experimental	
		name	result of `\TEST`	name	result of `\TEST`
(1)	17	<E3><81><82>	BB (\あ)	<1E3><181><182>	BB (\あ)
(2)	15	<E3><81><82>	BB (\ãĄĆ)	<E3><81><82>	AA (\ãĄĆ)

Table 3: Contents of *buffer* and *buffer2* when the experimental pTEX scans control sequences in an input line

	`\顛^^c5^^bf` (`\顛f`)					`\顛顛`				
buffer	\	<C5>	<BF>	<C5>	<BF>	\	<C5>	<BF>	<C5>	<BF>
buffer2	0	1	1	0	0	0	1	1	1	1
name		<1C5><1BF><C5><BF>					<1C5><1BF><1C5><1BF>			

With the experimental upTEX (and no TCX file), we can know that the second "あ" is treated as three 8-bit characters from the error message. I hope this will be useful in debugging.

```
! Undefined control sequence.
l.3 \J あ\L ^^e3^^81^^82\undefined
                                   あ\J あ
```

The third and the final "あ" is not read by upTEX's input processor at the error. So they are printed as if all UTF-8 characters gave Japanese character tokens.

7 Conclusion

The primary factor of the complications discussed in this paper is that (u)pTEX are Japanese extension of an 8-bit engine; this causes the same byte sequence can represent different things, namely a sequence of 8-bit characters (token) or Japanese characters. Although my experiment does not get rid of this factor (only ameliorates it), I hope that it is helpful.

I thank the executive committee of TEXConf 2019, which gave me the opportunity for preparing the original talk, and the people who discussed the topics of this paper with me, especially Hironobu Yamashita, Takuji Tanaka, Takayuki Yato, and Norbert Preining.

References

[1] ASCII Corporation. ASCII Japanese TeX (pTeX) (in Japanese). asciidwango.github.io/ptex/index.html.

[2] K. Berry, O. Weber. Web2c, for version 2019. tug.org/texlive/Contents/live/texmf-dist/doc/web2c/web2c.pdf, Feb. 2019.

[3] Japanese TEX Development Community. Guide to pTEX and friends. ctan.org/pkg/ptex-manual.

[4] JulienPalard. Inconsistent error message. github.com/texjporg/platex/issues/84.

[5] H. Kitagawa. Distinction between a byte sequence and a Japanese character token (in Japanese). github.com/texjporg/tex-jp-build/issues/81.

[6] H. Kitagawa. Distinction of Latin characters and Japanese characters in stringization of pTEX family (in Japanese). osdn.net/projects/eptex/docs/tc19ptex/ja/1/tc19ptex.pdf.

[7] H. Kitagawa. ε-pTEX Wiki (in Japanese). osdn.net/projects/eptex/wiki/FrontPage.

[8] LATEX Project Team. LATEX news, issue 28. www.latex-project.org/news/latex2e-news/ltnews28.pdf, Apr. 2018.

[9] H. Okumura. pTEX and Japanese Typesetting. *The Asian Journal of TEX* 2(1):43–51, 2008. ajt.ktug.org/2008/0201okumura.pdf

[10] T. Tanaka. upTeX, upLaTeX — unicode version of pTeX, pLaTeX. www.t-lab.opal.ne.jp/tex/uptex_en.html.

[11] T. Tanaka. upTEX — Unicode version of pTEX with cjk extensions. *TUGboat* 34(3):285–288, 2013. tug.org/TUGboat/tb34-3/tb108tanaka.pdf

[12] N. Tutimura. UTF-8 対応 (4) — ptetex Wiki (in Japanese). tutimura.ath.cx/ptetex/?UTF-8%C2%D0%B1%FE%284%29.

◇ Hironori Kitagawa
Tokyo, Japan
h_kitagawa2001 (at) yahoo dot co dot jp