

## UTF-8 installations of T<sub>E</sub>X

Igor Liferenko

### Abstract

In its design T<sub>E</sub>X has the concepts of “internal encoding” and “external encoding”. This fact allows T<sub>E</sub>X to work with any encoding.

We use Unicode as T<sub>E</sub>X’s external encoding. Then we change the necessary parts of T<sub>E</sub>X to use UTF-8 as the input/output encoding.

The resulting implementation passes the `trip` test.

### 1. Initialization

Note: we use the `web2w` [1] implementation of T<sub>E</sub>X, but the ideas, described in the present article, can be applied to any implementation.

First, we change the data type of characters in text files to `wchar_t` to accommodate Unicode values.

Background: this predefined C type allocates four bytes per character (on most systems). Character constants of this type are written as `L'...'` and string constants as `L"..."`.

(For brevity, in the diffs following, the original code from `web2w`’s `ctex.w` source is preceded with `<` characters, and the new code with `>`. Both are sometimes reformatted for presentation in this article, and for readability we sometimes leave a blank line between the pieces. The actual implementation uses the file `utex.patch` [2].)

```
< @d text_char pascal_char
> @d text_char wchar_t
```

Use values from the basic multilingual plane (BMP) of Unicode.

```
< @d last_text_char 255
> @d last_text_char 65535
```

Then we change the size of the `xord` array [3] to 2<sup>16</sup> bytes.

```
< ASCII_code xord[256];
> ASCII_code xord[65536];
```

Elements in the `xchr` array [3] are overridden using the file `mapping.w` [2].

```
@i mapping.w
```

This file specifies the character(s) required for a particular installation of T<sub>E</sub>X, for example:

```
xchr[0xf1] = L'ë';
```

In `tex.web` the type `char` (called `pascal_char` in this article—to distinguish it from the type `char` in C) is able to contain the same characters as in text files (see §19 in [3]). Therefore, we change `pascal_char` to `wchar_t`.

```
< pascal_char TEX_format_default0[form...],
<      *TEX_format_default = ...

> wchar_t      TEX_format_default0[form...],
>      *TEX_format_default = ...

< strcpy(TEX_format_default+1, "TeXforma...
> wcsncpy(TEX_format_default+1,L"TeXforma...

< pascal_char months0[36+1], *months = ...
> wchar_t      months0[36+1], *months = ...

< strcpy(months+1, "JANFEBMARAPRMYJUNJU...
> wcsncpy(months+1,L"JANFEBMARAPRMYJUNJU...
```

The `name_of_file` buffer must have the type `char` (instead of `wchar_t`), since file names are in UTF-8 (on most systems). The possibility of using a different type in this case is explicitly stated in comment to `name_of_file` (see §26 in [3]).

```
< pascal_char name_of_file0[file_name...],
<      *name_of_file = ...

> char      name_of_file0[file_name...],
>      *name_of_file = ...
```

It remains to set the `LC_CTYPE` locale category, on which depends the behavior of the C library functions used below

```
setlocale(LC_CTYPE, "C.UTF-8");
```

and to add the necessary headers.

```
#include <wchar.h>
#include <locale.h>
```

### 2. Input

For automatic conversion from UTF-8 to Unicode, text files (including the terminal) must be read with the C library function `fgetc` [4].

In `ctex.w` the macro `get` is used for reading text files, as well as font metric and format files.

Text files are read in the functions `a_open_in` and `input_ln`. In `a_open_in` we replace the macro `reset` with its expansion and then in both functions we change `'get(*f)'` to `'f->d=fgetc(f->f)'`.

### 3. Output

Printed strings can be specified directly in UTF-8. And as they are already in ASCII (which is part of UTF-8) we need no special treatment for them, except that `%c` is changed to `%lc`.

```

< wlog("%c",months[k]);
> wlog("%lc",months[k]);

< wterm("%c",xchr[s]);
> wterm("%lc",xchr[s]);

< wlog("%c",xchr[s]);
> wlog("%lc",xchr[s]);

< write(write_file[selector],"%c",xchr[s]);
> write(write_file[selector],"%lc",xchr[s]);

```

#### 4. The file name buffer

The name of the file to be opened, which is stored in the *name\_of\_file* buffer, must be encoded in UTF-8. Therefore, each character passed to *append\_to\_name*, before being added to *name\_of\_file*, must be converted to UTF-8. This is done using the C library function *wctomb* [4].

In the *append\_to\_name* macro, the variable *k* is used as the index into the *name\_of\_file* buffer where the last byte was stored. Originally, *k* was always increased and provisions were made that characters will not be written beyond the end of buffer (which has the index *file\_name\_size*); *name\_length* was then set to the minimal value between *k* and *file\_name\_size*.

We cannot do the same in our implementation, because we may reach the end of the buffer in the midst of a multibyte character. Instead, if the next multibyte character does not fit into the buffer, we prevent *k* from being increased by negating its value:

```

< @d append_to_name(X) { c=X;incr(k);
<   if (k <= file_name_size)
<     name_of_file[k]=xchr[c]; }

> @d append_to_name(X) { c=X;
>   if (k >= 0) { /* try to append? */
>     char mb[MB_CUR_MAX];
>     int len = wctomb(mb, xchr[c]);
>     if (k+len <= file_name_size)
>       for (int i = 0; i < len; i++)
>         name_of_file[++k] = mb[i];
>     else
>       k = -k; /* freeze k */ } }

```

In *pack\_file\_name* and *pack\_buffered\_name* (the functions that call *append\_to\_name*), we have to “un-freeze” its value if it was “frozen”.

```
if (k < 0) k = -k;
```

In *make\_name\_string*, each (multibyte) character from *name\_of\_file* must be converted from UTF-8 to Unicode, before being converted to T<sub>E</sub>X’s internal

encoding. This is done using the C library function *mbtowc* [4].

```

< append_char(xord[name_of_file[k]]);

> { wchar_t wc;
>   k += mbtowc(&wc, name_of_file+k,
>             MB_CUR_MAX) - 1;
>   append_char(xord[wc]); }

```

In the code checking *format\_default\_length* for consistency, we use the C library function *wcstombs* [4] to count the number of bytes in the UTF-8 representation of *TEX\_format\_default*.

```

< if (format_default_length >
<   file_name_size)

> if (wcstombs(NULL,
>   TEX_format_default+1,0) >
>   file_name_size)

```

In the function *pack\_buffered\_name*, the code that drops excess characters assumes that each character is one byte:

```

if (n+b-a+1+format_ext_length >
    file_name_size)
    b=a+file_name_size-n-1-format_ext_length;

```

But the number of bytes used to represent a character in UTF-8 may be more than one. Therefore, we use an equivalent method to drop excess characters, the one which will work with multibyte characters: After appending the contents of *buffer[a .. b]* to *name\_of\_file*, we roll back in it character by character until the format extension fits in it. We use the C library function *mblen* [4] to determine the start of the next (multibyte) character to be discarded.

```

while (k+wcstombs(NULL,TEX_format_default+
    format_default_length-
    format_ext_length+1,0) >
    file_name_size) {
    k--;
    while (mblen(name_of_file+k+1,MB_CUR_MAX)
        ==-1)
        k--;
}

```

#### References

- [1] Ruckert, Martin. WEB to cweb.  
[https://w3-o.cs.hm.edu/users/ruckert/public\\_html/web2w/](https://w3-o.cs.hm.edu/users/ruckert/public_html/web2w/)
- [2] Source of the present implementation.  
<https://github.com/igor-liferenko/tex>
- [3] Knuth, Donald E. *T<sub>E</sub>X: The Program*, 1986. ISBN 0201134373.

- [4] Single Unix Specification. Introduction to  
ISO C Amendment 1 (Multibyte Support  
Environment).  
[http://unix.org/version2/whatsnew/  
login\\_mse.html](http://unix.org/version2/whatsnew/login_mse.html)

◇ Igor Liferenko  
igor.liferenko (at) gmail dot com