Hello World
or
Καλημέρα κόσμε
or
こんにちは世界

Rob Pike
Ken Thompson

rob,ken@plan9.bell-labs.com

ABSTRACT

Plan 9 from Bell Labs has recently been converted from ASCII to an ASCII-compatible variant of the Unicode Standard, a 16-bit (now 21-bit) character set. In this paper we explain the reasons for the change, describe the character set and representation we chose, and present the programming models and software changes that support the new text format. Although we stopped short of full internationalization—for example, system error messages are in Unixese, not Japanese—we believe Plan 9 is the first system to treat the representation of all major languages on a uniform, equal footing throughout all its software.

Introduction

The world is multilingual but most computer systems are based on English and ASCII. The first release of Plan 9 [Pike90], a new distributed operating system from Bell Laboratories, seemed a good occasion to correct this chauvinism. It is easier to make such deep changes when building new systems than by refitting old ones.

The ANSI C standard [ANSIC] contains some guidance on the matter of 'wide' and 'multi-byte' characters but falls far short of solving the myriad associated problems. We could find no literature on how to convert a *system* to larger character sets, although some individual *programs* had been converted. This paper reports what we discovered as we explored the problem of representing multilingual text at all levels of an operating system, from the file system and kernel through the applications and up to the window system and display.

Plan 9 has not been 'internationalized': its manuals are in English, its error messages are in English, and it can display text that goes from left to right only. But before we can address these other problems, we need to handle, uniformly and comfortably, the textual representation of all the major written languages. That subproblem is richer than we had anticipated.

Originally appeared, in a slightly different form, in *Proc. of the Winter 1993 USENIX Conf.*, pp. 43–50, San Diego. It has been revised to reflect the move to 21-bit Unicode.

Standards

Our first step was to select a standard. At the time (January 1992), there were only two viable options: ISO 10646 [ISO10646] and Unicode [Unicode]. The documents describing both proposals were still in the draft stage.

The draft of ISO 10646 was not very attractive to us. It defined a sparse set of 32-bit characters, which would be hard to implement and have punitive storage requirements. Also, the draft attempted to mollify national interests by allocating 16-bit subspaces to national committees to partition individually. The suggested mode of use was to ‘‘flip’’ between separate national standards to implement the international standard. This did not strike us as a sound basis for a character set. As well, transmitting 32-bit values in a byte stream, such as in pipes, would be expensive and hard to implement. Since the standard does not define a byte order for such transmission, the byte stream would also have to carry state to enable the values to be recovered.

The Unicode Standard is a proposal by a consortium of mostly American computer companies formed to protest the technical failings of ISO 10646. It defines a uniform 16-bit code based on the principle of unification: two characters are the same if they look the same even though they are from different languages. This principle, called Han unification, allows the large Japanese, Chinese, and Korean character sets to be packed comfortably into a 16-bit representation.

We chose the Unicode Standard for its technical merits and because its code space was better defined. Moreover, the Unicode Consortium was derailing the ISO 10646 standard. (Now, in 1995, ISO 10646 is a standard with one 16-bit group defined, which is almost exactly the Unicode Standard. As most people expected, the two standards bodies reached a détente and ISO 10646 and Unicode represent the same character set.)

The Unicode Standard defines an adequate character set but an unreasonable representation. It states that all characters are 16 bits wide and are communicated and stored in 16-bit units. It also reserves a pair of characters (hexadecimal FFFE and FEFF) to detect byte order in transmitted text, requiring state in the byte stream. (The Unicode Consortium was thinking of files, not pipes.) To adopt this encoding, we would have had to convert all text going into and out of Plan 9 between ASCII and Unicode, which cannot be done. Within a single program, in command of all its input and output, it is possible to define characters as 16-bit quantities; in the context of a networked system with hundreds of applications on diverse machines by different manufacturers, it is impossible.

We needed a way to adapt the Unicode Standard to the tools-and-pipes model of text processing embodied by the Unix system. To do that, we needed an ASCII-compatible textual representation of Unicode characters for transmission and storage. In the draft ISO standard there was an informative (non-required) Annex called UTF that provided a byte stream encoding of the 32-bit ISO code. The encoding uses multibyte sequences composed from the 190 printable characters of Latin-1 to represent character values larger than 159.

The UTF encoding has several good properties. By far the most important is that a byte in the ASCII range 0-127 represents itself in UTF. Thus UTF is backward compatible with ASCII.

UTF has other advantages. It is a byte encoding and is therefore byte-order independent. ASCII control characters appear in the byte stream only as themselves, never as an element of a sequence encoding another character, so newline bytes separate lines of UTF text. Finally, ANSI C’s `strcmp` function applied to UTF strings preserves the ordering of Unicode characters.

To encode and decode UTF is expensive (involving multiplication, division, and modulo operations) but workable. UTF’s major disadvantage is that the encoding is not self-synchronizing. It is in general impossible to find the character boundaries in a UTF string without reading from the beginning of the string, although in practice control characters such as newlines, tabs, and blanks provide synchronization points.

In August 1992, X-Open circulated a proposal for another UTF-like byte encoding of Unicode characters. Their major concern was that an embedded character in a file name (in particular a slash) could be part of an escape sequence in UTF and therefore confuse a traditional file system. Their proposal would allow all 7-bit ASCII characters to represent themselves *and only themselves* in text. Multibyte sequences would contain only characters with the high bit set. We proposed a modification to the new UTF that would address our synchronization problem. Our proposal, which was originally known informally as UTF-2 and FSS-UTF, is now referred to as UTF-8 and has been approved by ISO to become Annex P to ISO 10646.

The model for text in Plan 9 is chosen from these three standards*: the Unicode character set encoded as a byte stream by UTF-8, from (soon to be) Annex P of ISO 10646. Although this mixture may seem like a precarious position for us to adopt, it is not as bad as it sounds. ISO 10646 and the Unicode Standard have converged, other systems such as Linux have adopted the same character set and encoding, and the general feeling seems to be that Unicode and UTF-8 will be accepted as the way to exchange text between systems. The prognosis for wide acceptance is good.

There are a couple of aspects of the Unicode Standard we have not faced. One is the issue of right-to-left text such as Hebrew or Arabic. Since that is an issue of display, not representation, we believe we can defer that problem for the moment without affecting our ability to solve it later. Another issue is diacriticals and ‘combining characters’, which cause overstriking of multiple Unicode characters. Although necessary for some scripts, such as Thai, Arabic, and Hebrew, such characters confuse the issues for Latin languages because they generate multiple representations for accented characters. ISO 10646 describes three levels of implementation; in Plan 9 we decided not to address the issue. Again, this can be labeled as a display issue and its finer points are still being debated, so we felt comfortable deferring. Mañana.

Although we converted Plan 9 in the altruistic interests of serving foreign languages, we have found the large character set attractive for other reasons. The Unicode Standard includes many characters—mathematical symbols, scientific notation, more general punctuation, and more—that we now use daily in our work. We no longer test our imaginations to find ways to include non-ASCII symbols in our text; why type :-) when you can use the character ☺? Most compelling is the ability to absorb documents and data that contain non-ASCII characters; our browser for the Oxford English Dictionary lets us see the dictionary as it really is, with pronunciation in the IPA font, foreign phrases properly rendered, and so on, *in plain text.*

As of Unicode 4.0, characters are now 21 bits wide and the longest UTF-8 encoding of a character requires 4 bytes. We are adapting the system to match.

In the rest of this paper, except when stated otherwise, the term ‘UTF’ refers to the UTF-8 encoding of Unicode characters as adopted by Plan 9.

C Compiler

The first program to be converted to UTF was the C Compiler. There are two levels of conversion. On the syntactic level, input to the C compiler is UTF; on the semantic level, the C language needs to define how compiled programs manipulate the UTF set.

The syntactic part is simple. The ANSI C language standard defines the source character set to be ASCII. Since UTF is backward compatible with ASCII, the compiler needs little change. The only places where a larger character set is allowed are in

* “That’s the nice thing about standards—there’s so many to choose from.” - Andy Tannenbaum (no, the other one)

character constants, strings, and comments. Since 7-bit ASCII characters can represent only themselves in UTF, the compiler does not have to be careful while looking for the termination of a string or comment.

The Plan 9 compiler extends ANSI C to treat any Unicode character with a value outside of the ASCII range as an alphabetic. To a Greek programmer or an English mathematician, α is a sensible and now valid variable name.

On the semantic level, ANSI C allows, but does not tie down, the notion of a *wide character* and admits string and character constants of this type. We chose the wide character type to be `unsigned short` (now `unsigned long`). In the libraries, the word `Rune` is now defined by a `typedef` to be equivalent to `unsigned long` and is used to signify a Unicode character.

There are surprises; for example:

`L'x'`	is 120
`'x'`	is 120
`L'ÿ'`	is 255
`'ÿ'`	is −1, stdio `EOF` (if `char` is signed)
`L'α'`	is 945
`'α'`	is illegal

In the string constants,

```
"こんにちは世界"
L"こんにちは世界",
```

the former is an array of `char`s with 22 elements and a null byte, while the latter is an array of `unsigned long`s (`Rune`s) with 8 elements and a null `Rune`.

The Plan 9 library provides an output conversion function, `print` (analogous to `printf`), with formats `%c`, `%C`, `%s`, and `%S`. Since `print` produces text, its output is always UTF. The character conversion `%c` (lower case) masks its argument to 8 bits before converting to UTF. Thus `L'ÿ'` and `'ÿ'` printed under `%c` will be identical, but `L'α'` will print as the Unicode character with decimal value 177. The character conversion `%C` (upper case) masks its argument to 16 bits before converting to UTF. Thus `L'ÿ'` and `L'α'` will print correctly under `%C`, but `'ÿ'` will not. The conversion `%s` (lower case) expects a pointer to `char` and copies UTF sequences up to a null byte. The conversion `%S` (upper case) expects a pointer to `Rune` and performs sequential `%C` conversions until a null `Rune` is encountered.

Another problem in format conversion is the definition of `%10s`: does the number refer to bytes or characters? We decided that such formats were most often used to align output columns and so made the number count characters. Some programs, however, use the count to place blank-padded strings in fixed-sized arrays. These programs must be found and corrected.

Here is a complete example:

```
#include <u.h>

char c[] = "こんにちは世界";
Rune s[] = L"こんにちは世界";

main(void)
{
        print("%d, %d\n", sizeof(c), sizeof(s));
        print("%s\n", c);
        print("%S\n", s);
}
```

This program prints `23, 18` and then two identical lines of UTF text. In practice, `%S` and `L"..."` are rare in programs; one reason is that most formatted I/O is done in unconverted UTF.

Ramifications

All programs in Plan 9 now read and write text as UTF, not ASCII. This change breaks two deep-rooted symmetries implicit in most C programs:

1. A character is no longer a `char`.
2. The internal representation (Rune) of a character now differs from its external representation (UTF).

In the sections that follow, we show how these issues were faced in the layers of system software from the operating system up to the applications. The effects are wide-reaching and often surprising.

Operating system

Since UTF is the only format for text in Plan 9, the interface to the operating system had to be converted to UTF. Text strings cross the interface in several places: command arguments, file names, user names (people can log in using their native name), error messages, and miscellaneous minor places such as commands to the I/O system. Little change was required: null-terminated UTF strings are equivalent to null-terminated ASCII strings for most purposes of the operating system. The library routines described in the next section made that change straightforward.

The window system, once called `8.5`, is now rightfully called 8½.

Libraries

A header file included by all programs (see [Pike92]) declares the `Rune` type to hold 21-bit character values:

```
typedef unsigned long Rune;
```

Also defined are several constants relevant to UTF:

```
enum
{
    UTFmax      = 4,    /* maximum bytes per rune */
    Runesync    = 0x80, /* cannot be in a UTF sequence (<) */
    Runeself    = 0x80, /* rune==UTF sequence (<) */
    Runeerror   = 0xFFFD,       /* decoding error in UTF */
    Runemax     = 0x10FFFF,     /* largest 21-bit rune */
    Runemask    = 0x1FFFFF,     /* bits used by runes (see grep) */
};
```

(With the original UTF, `Runesync` was hexadecimal 21 and `Runeself` was A0.) `UTFmax` bytes are sufficient to hold the UTF encoding of any Unicode character. Characters of value less than `Runesync` only appear in a UTF string as themselves, never as part of a sequence encoding another character. Characters of value less than `Runeself` encode into single bytes of the same value. Finally, when the library detects errors in UTF input—byte sequences that are not valid UTF sequences—it converts the first byte of the error sequence to the character `Runeerror`. There is little a rune-oriented program can do when given bad data except exit, which is unreasonable, or carry on. Originally the conversion routines, described below, returned errors when given invalid UTF, but we found ourselves repeatedly checking for errors and ignoring them. We therefore decided to convert a bad sequence to a valid rune and continue processing. (The ANSI C routines, on the other hand, return errors.)

This technique does have the unfortunate property that converting invalid UTF byte strings in and out of runes does not preserve the input, but this circumstance only occurs when non-textual input is given to a textual program. The Unicode Standard defines an error character, value FFFD, to stand for characters from other sets that it does not represent. The Runeerror character is a different concept, related to the encoding rather than the character set.

The Plan 9 C library contains a number of routines for manipulating runes. The first set converts between runes and UTF strings:

```
extern  int     runetochar(char*, Rune*);
extern  int     chartorune(Rune*, char*);
extern  int     runelen(long);
extern  int     fullrune(char*, int);
```

Runetochar translates a single Rune to a UTF sequence and returns the number of bytes produced. Chartorune goes the other way, reporting how many bytes were consumed. Runelen returns the number of bytes in the UTF encoding of a rune. Fullrune examines a UTF string up to a specified number of bytes and reports whether the string begins with a complete UTF encoding. All these routines use the Runeerror character to work around encoding problems.

There is also a set of routines for examining null-terminated UTF strings, based on the model of the ANSI standard str routines, but with utf substituted for str and rune for chr:

```
extern  int     utflen(char*);
extern  char*   utfrune(char*, long);
extern  char*   utfrrune(char*, long);
extern  char*   utfutf(char*, char*);
```

Utflen returns the number of runes in a UTF string; utfrune returns a pointer to the first occurrence of a rune in a UTF string; and utfrrune a pointer to the last. Utfutf searches for the first occurrence of a UTF string in another UTF string. Given the synchronizing property of UTF-8, utfutf is the same as strstr if the arguments point to valid UTF strings.

It is a mistake to use strchr or strrchr unless searching for a 7-bit ASCII character, that is, a character less than Runeself.

We have no routines for manipulating null-terminated arrays of Runes. Although they should probably exist for completeness, we have found no need for them, for the same reason that %S and L"..." are rarely used.

Most Plan 9 programs use a new buffered I/O library, BIO, in place of Standard I/O. BIO contains routines to read and write UTF streams, converting to and from runes. Bgetrune returns, as a Rune within a long, the next character in the UTF input stream; Bputrune takes a rune and writes its UTF representation. Bungetrune puts a rune back into the input stream for rereading.

Plan 9 programs use a simple set of macros to process command line arguments. Converting these macros to UTF automatically updated the argument processing of most programs. In general, argument flag names can no longer be held in bytes and arrays of 256 bytes cannot be used to hold a set of flags.

We have done nothing analogous to ANSI C's locales, partly because we do not feel qualified to define locales and partly because we remain unconvinced of that model for dealing with the problems. That is really more an issue of internationalization than conversion to a larger character set; on the other hand, because we have chosen a single character set that encompasses most languages, some of the need for locales is eliminated. (We have a utility, tcs, that translates between UTF and other character sets.)

There are several reasons why our library does not follow the ANSI design for wide and multi-byte characters. The ANSI model was designed by a committee, untried, almost as an afterthought, whereas we wanted to design as we built. (We made several major changes to the interface as we became familiar with the problems involved.) We disagree with ANSI C's handling of invalid multi-byte sequences. Also, the ANSI C library is incomplete: although it contains some crucial routines for handling wide and multi-byte characters, there are some serious omissions. For example, our software can exploit the fact that UTF preserves ASCII characters in the byte stream. We could remove that assumption by replacing all calls to `strchr` with `utfrune` and so on. (Because of the weaker properties of the original UTF, we have actually done so.) ANSI C cannot: the standard says nothing about the representation, so portable code should *never* call `strchr`, yet there is no ANSI equivalent to `utfrune`. ANSI C simultaneously invalidates `strchr` and offers no replacement.

Finally, ANSI did nothing to integrate wide characters into the I/O system: it gives no method for printing wide characters. We therefore needed to invent some things and decided to invent everything. In the end, some of our entry points do correspond closely to ANSI routines—for example `chartorune` and `runetochar` are similar to `mbtowc` and `wctomb`—but Plan 9's library defines more functionality, enough to write real applications comfortably.

Converting the tools

The source for our tools and applications had already been converted to work with Latin-1, so it was '8-bit safe', but the conversion to the Unicode Standard and UTF is more involved. Some programs needed no change at all: `cat`, for instance, interprets its argument strings, delivered in UTF, as file names that it passes uninterpreted to the `open` system call, and then just copies bytes from its input to its output; it never makes decisions based on the values of the bytes. (Plan 9 `cat` has no options such as `-v` to complicate matters.) Most programs, however, needed modest change.

It is difficult to find automatically the places that need attention, but `grep` helps. Software that uses the libraries conscientiously can be searched for calls to library routines that examine bytes as characters: `strchr`, `strrchr`, `strstr`, etc. Replacing these by calls to `utfrune`, `utfrrune`, and `utfutf` is enough to fix many programs. Few tools actually need to operate on runes internally; more typically they need only to look for the final slash in a file name and similar trivial tasks. Of the 170 C source programs in the top levels of `/sys/src/cmd`, only 23 now contain the word `Rune`.

The programs that *do* store runes internally are mostly those whose *raison d'être* is character manipulation: `sam` (the text editor), `sed`, `sort`, `tr`, `troff`, `8½` (the window system and terminal emulator), and so on. To decide whether to compute using runes or UTF-encoded byte strings requires balancing the cost of converting the data when read and written against the cost of converting relevant text on demand. For programs such as editors that run a long time with a relatively constant dataset, runes are the better choice. There are space considerations too, but they are more complicated: plain ASCII text grows when converted to runes; UTF-encoded Japanese shrinks.

Again, it is hard to automate the conversion of a program from `chars` to `Runes`. It is not enough just to change the type of variables; the assumption that bytes and characters are equivalent can be insidious. For instance, to clear a character array by

```
memset(buf, 0, BUFSIZE)
```

becomes wrong if `buf` is changed from an array of `chars` to an array of `Runes`. Any program that indexes tables based on character values needs rethinking. Consider `tr`, which originally used multiple 256-byte arrays for the mapping. The naïve conversion would yield multiple 1,114,112-rune arrays. Instead Plan 9 `tr` saves space by building in effect a run-encoded version of the map.

`Sort` has related problems. The cooperation of UTF and `strcmp` means that a simple sort—one with no options—can be done on the original UTF strings using `strcmp`. With sorting options enabled, however, `sort` may need to convert its input to runes: for example, option `−tα` requires searching for alphas in the input text to crack the input into fields. The field specifier `+3.2` refers to 2 runes beyond the third field. Some of the other options are hopelessly provincial: consider the case-folding and dictionary order options (Japanese doesn't even have an official dictionary order) or −M which compares by case-insensitive English month name. Handling these options involves the larger issues of internationalization and is beyond the scope of this paper and our expertise. Plan 9 `sort` works sensibly with options that make sense relative to the input. The simple and most important options are, however, usually meaningful. In particular, `sort` sorts UTF into the same order that `look` expects.

Regular expression-matching algorithms need rethinking to be applied to UTF text. Deterministic automata are usually applied to bytes; converting them to operate on variable-sized byte sequences is awkward. On the other hand, converting the input stream to runes adds measurable expense and the state tables expand from size 256 to 1,114,112; it can be expensive just to generate them. For simple string searching, the Boyer-Moore algorithm works with UTF provided the input is guaranteed to be only valid UTF strings; however, it does not work with the old UTF encoding. At a more mundane level, even character classes are harder: the usual bit-vector representation within a non-deterministic automaton is unwieldy with 1,114,112 characters in the alphabet.

We compromised. An existing library for compiling and executing regular expressions was adapted to work on runes, with two entry points for searching in arrays of runes and arrays of chars (the pattern is always UTF text). Character classes are represented internally as runs of runes; the reserved value `FFFF` marks the end of the class. Then *all* utilities that use regular expressions—editors, `grep`, `awk`, etc.—except the shell, whose notation was grandfathered, were converted to use the library. For some programs, there was a concomitant loss of performance, but there was also a strong advantage. To our knowledge, Plan 9 is the only Unix-like system that has a single definition and implementation of regular expressions; patterns are written and interpreted identically by all the programs in the system.

A handful of programs have the notion of character built into them so strongly as to confuse the issue of what they should do with UTF input. Such programs were treated as individual special cases. For example, `wc` is, by default, unchanged in behavior and output; a new option, `−r`, counts the number of correctly encoded runes—valid UTF sequences—in its input; −b the number of invalid sequences.

It took us several months to convert all the software in the system to the Unicode Standard and the old UTF. When we decided to convert from that to the new UTF, only three things needed to be done. First, we rewrote the library routines to encode and decode the new UTF. This took an evening. Next, we converted all the files containing UTF to the new encoding. We wrote a trivial program to look for non-ASCII bytes in text files and used a Plan 9 program called `tcs` (translate character set) to change encodings. Finally, we recompiled all the system software; the library interface was unchanged, so recompilation was sufficient to effect the transformation. The second two steps were done concurrently and took an afternoon. We concluded that the actual encoding is relatively unimportant to the software; the adoption of large characters and a byte-stream encoding *per se* are much deeper issues.

Graphics and fonts

Plan 9 provides only minimal support for plain text terminals. It is instead designed to be used with all character input and output mediated by a window system such as 8½. The window system and related software are responsible for the display of UTF text as Unicode character images. For plain text, the window system must provide a user-settable *font* that provides a (possibly empty) picture for each Unicode character. Fancier applications that use bold and Italic characters need multiple fonts storing multiple pictures for each Unicode value. All the issues are apparent, though, in just the problem of displaying a single image for each character, that is, the Unicode equivalent of a plain text terminal. With 128 or even 256 characters, a font can be just an array of bitmaps. With 1,114,112 characters, a more sophisticated design is necessary. To store the ideographs for just Japanese as $16\times16\times1$ bit images, the smallest they can reasonably be, takes over a quarter of a megabyte. Make the images a little larger, store more bits per pixel, and hold a copy in every running application, and the memory cost becomes unreasonable.

The structure of the bitmap graphics services is described at length elsewhere [Pike91]. In summary, the memory holding the bitmaps is stored in the same machine that has the display, mouse, and keyboard: the terminal in Plan 9 terminology, the workstation in others'. Access to that memory and associated services is provided by device files served by system software on the terminal. One of those files, `/dev/bitblt`, interprets messages written upon it as requests for actions corresponding to entry points in the graphics library: allocate a bitmap, execute a raster operation, draw a text string, etc. The window system acts as a multiplexer that mediates access to the services and resources of the terminal by simulating in each client window a set of files mirroring those provided by the system. That is, each window has a distinct `/dev/mouse`, `/dev/bitblt`, and so on through which applications drive graphical input and output.

One of the resources managed by 8½ and the terminal is the set of active *subfonts.* Each subfont holds the bitmaps and associated data structures for a sequential set of Unicode characters. Subfonts are stored in files and loaded into the terminal by 8½ or an application. For example, one subfont might hold the images of the first 256 characters of the Unicode space, corresponding to the Latin-1 character set; another might hold the standard phonetic character set, Unicode characters with value 0250 to 02E9. These files are collected in directories corresponding to typefaces: `/lib/font/bit/pelm` contains the Pellucida Monospace character set, with subfonts holding the Latin-1, Greek, Cyrillic and other components of the typeface. A suffix on subfont files encodes (in a subfont-specific way) the size of the images: `/lib/font/bit/pelm/latin1.9` contains the Latin-1 Pellucida Monospace characters with lower case letters 9 pixels high; `/lib/font/bit/jis/jis5400.16` contains 16-pixel high ideographs starting at Unicode value 5400.

The subfonts do not identify which portion of the Unicode space they cover. Instead, a font file, in plain text, describes how to assemble subfonts into a complete character set. The font file is presented as an argument to the window system to determine how plain text is displayed in text windows and applications. Here is the beginning of the font file `/lib/font/bit/pelm/jis.9.font`, which describes the layout of a font covering that portion of the Unicode Standard for which we have characters of typical display size, using Japanese characters to cover the Han space:

```
18      14
0x0000  0x00FF  latin1.9
0x0100  0x017E  latineur.9
0x0250  0x02E9  ipa.9
0x0386  0x03F5  greek.9
0x0400  0x0475  cyrillic.9
0x2000  0x2044  ../misc/genpunc.9
0x2070  0x208E  supsub.9
0x20A0  0x20AA  currency.9
0x2100  0x2138  ../misc/letterlike.9
0x2190  0x21EA  ../misc/arrows
0x2200  0x227F  ../misc/math1
0x2280  0x22F1  ../misc/math2
0x2300  0x232C  ../misc/tech
0x2500  0x257F  ../misc/chart
0x2600  0x266F  ../misc/ding

0x3000  0x303f  ../jis/jis3000.16
0x30a1  0x30fe  ../jis/katakana.16
0x3041  0x309e  ../jis/hiragana.16
0x4e00  0x4fff  ../jis/jis4e00.16
0x5000  0x51ff  ../jis/jis5000.16
...
```

The first two numbers set the interline spacing of the font (18 pixels) and the distance from the baseline to the top of the line (14 pixels). When characters are displayed, they are placed so as best to fit within those constraints; characters too large to fit will be truncated. The rest of the file associates subfont files with portions of Unicode space. The first four such files are in the Pellucida Monospace typeface and directory; others reside in other directories. The file names are relative to the font file's own location.

There are several advantages to this two-level structure. First, it simultaneously breaks the huge Unicode space into manageable components and provides a unifying architecture for assembling fonts from disjoint pieces. Second, the structure promotes sharing. For example, we have only one set of Japanese characters but dozens of typefaces for the Latin-1 characters, and this structure permits us to store only one copy of the Japanese set but use it with any Roman typeface. Also, customization is easy. English-speaking users who don't need Japanese characters but may want to read an on-line Oxford English Dictionary can assemble a custom font with the Latin-1 (or even just ASCII) characters and the International Phonetic Alphabet (IPA). Moreover, to do so requires just editing a plain text file, not using a special font editing tool. Finally, the structure guides the design of caching protocols to improve performance and memory usage.

To load a complete Unicode character set into each application would consume too much memory and, particularly on slow terminal lines, would take unreasonably long. Instead, Plan 9 assembles a multi-level cache structure for each font. An application opens a font file, reads and parses it, and allocates a data structure. A message written to `/dev/bitblt` allocates an associated structure held in the terminal, in particular, a bitmap to act as a cache for recently used character images. Other messages copy these images to bitmaps such as the screen by loading characters from subfonts into the cache on demand and from there to the destination bitmap. The protocol to draw characters is in terms of cache indices, not Unicode character number or UTF sequences. These details are hidden from the application, which instead sees only a subroutine to draw a string in a bitmap from a given font, functions to discover character size information, and routines to allocate and to free fonts.

As needed, whole subfonts are opened by the graphics library, read, and then downloaded to the terminal. They are held open by the library in an LRU-replacement list. Even when the program closes a subfont, it is retained in the terminal for later use. When the application opens the subfont, it asks the terminal if it already has a copy to avoid reading it from the file server if possible. This level of cache has the property that the bitmaps for, say, all the Japanese characters are stored only once, in the terminal; the applications read only size and width information from the terminal and share the images.

The sizes of the character and subfont caches held by the application are adaptive. A simple algorithm monitors the cache miss rate to enlarge and shrink the caches as required. The size of the character cache is limited to 2048 images maximum, which in practice seems enough even for Japanese text. For plain ASCII-like text it naturally stays around 128 images.

This mechanism sounds complicated but is implemented by only about 500 lines in the library and considerably less in each of the terminal's graphics driver and 8½. It has the advantage that only characters that are being used are loaded into memory. It is also efficient: if the characters being drawn are in the cache the extra overhead is negligible. It works particularly well for alphabetic character sets, but also adapts on demand for ideographic sets. When a user first looks at Japanese text, it takes a few seconds to read all the font data, but thereafter the text is drawn almost as fast as regular text (the images are larger, so draw a little slower). Also, because the bitmaps are remembered by the terminal, if a second application then looks at Japanese text it starts faster than the first.

We considered building a 'font server' to cache character images and associated data for the applications, the window system, and the terminal. We rejected this design because, although isolating many of the problems of font management into a separate program, it didn't simplify the applications. Moreover, in a distributed system such as Plan 9 it is easy to have too many special purpose servers. Making the management of the fonts the concern of only the essential components simplifies the system and makes bootstrapping less intricate.

Input

A completely different problem is how to type Unicode characters as input to the system. We selected an unused key on our ASCII keyboards to serve as a prefix for multi-keystroke sequences that generate Unicode characters. For example, the character ü is generated by the prefix key (typically `ALT` or `Compose`) followed by a double quote and a lower-case `u`. When that character is read by the application, from the file `/dev/cons`, it is of course presented as its UTF encoding. Such sequences generate characters from an arbitrary set that includes all of Latin-1 plus a selection of mathematical and technical characters. An arbitrary Unicode character may be generated by typing the prefix, an upper case X, and four hexadecimal digits that identify the Unicode value.

These simple mechanisms are adequate for most of our day-to-day needs: it's easy to remember to type 'ALT 1 2' for ½ or 'ALT accent letter' for accented Latin letters. For the occasional unusual character, the cut and paste features of 8½ serve well. A program called (perhaps misleadingly) `unicode` takes as argument a hexadecimal value, and prints the UTF representation of that character, which may then be picked up with the mouse and used as input.

These methods are clearly unsatisfactory when working in a non-English language. In the native country of such a language the appropriate keyboard is likely to be at hand. But it's also reasonable—especially now that the system handles Unicode characters—to work in a language foreign to the keyboard.

For alphabetic languages such as Greek or Russian, it is straightforward to construct a program that does phonetic substitution, so that, for example, typing a Latin 'a' yields the Greek 'α'. Within Plan 9, such a program can be inserted transparently between the real keyboard and a program such as the window system, providing a manageable input device for such languages.

For ideographic languages such as Chinese or Japanese the problem is harder. Native users of such languages have adopted methods for dealing with Latin keyboards that involve a hybrid technique based on phonetics to generate a list of possible symbols followed by menu selection to choose the desired one. Such methods can be effective, but their design must be rooted in information about the language unknown to non-native speakers. (`Cxterm`, a Chinese terminal emulator built by and for Chinese programmers, employs such a technique [Pong and Zhang].) Although the technical problem of implementing such a device is easy in Plan 9—it is just an elaboration of the technique for alphabetic languages—our lack of familiarity with such languages has restrained our enthusiasm for building one.

The input problem is technically the least interesting but perhaps emotionally the most important of the problems of converting a system to an international character set. Beyond that remain the deeper problems of internationalization such as multi-lingual error messages and command names, problems we are not qualified to solve. With the ability to treat text of most languages on an equal footing, though, we can begin down that path. Perhaps people in non-English speaking countries will consider adopting Plan 9, solving the input problem locally—perhaps just by plugging in their local terminals—and begin to use a system with at least the capacity to be international.

Acknowledgements

Dennis Ritchie provided consultation and encouragement. Bob Flandrena converted most of the standard tools to UTF. Brian Kernighan suffered cheerfully with several inadequate implementations and converted `troff` to UTF. Rich Drechsler converted his Postscript driver to UTF. John Hobby built the Postscript ☺. We thank them all.

References

[ANSIC] *American National Standard for Information Systems - Programming Language C*, American National Standards Institute, Inc., New York, 1990.

[ISO10646] ISO/IEC DIS 10646-1:1993 *Information technology - Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.*

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, "Plan 9 from Bell Labs", UKUUG Proc. of the Summer 1990 Conf., London, England, 1990.

[Pike91] R. Pike, "8½, The Plan 9 Window System", USENIX Summer Conf. Proc., Nashville, 1991, reprinted in this volume.

[Pike92] R. Pike, "How to Use the Plan 9 C Compiler", this volume.

[Pong and Zhang] Man-Chi Pong and Yongguang Zhang, "cxterm: A Chinese Terminal Emulator for the X Window System", *Software—Practice and Experience,* Vol 22(1), 809-926, October 1992.

[Unicode] *The Unicode Standard, Worldwide Character Encoding, Version 1.0, Volume 1*, The Unicode Consortium, Addison Wesley, New York, 1991.