Linux Unicode programming

How to incorporate and utilize Unicode for foreign language support

Thomas Burger

August 01, 2001

A multi-byte character representation system for computers, Unicode provides for the encoding and exchanging of all of the text of the world's languages. This article explains the importance of international language support and the concepts of designing and incorporating Unicode support in Linux applications.

Unicode is not just a programming tool, but also a political and economic tool. Applications that do not incorporate world language support can often be used only by individuals who read and write a language supported by ASCII. This puts computer technology based on ASCII out of reach of most of the world's people. Unicode allows programs to utilize any of the world's character sets and therefore support any language.

Unicode allows programmers to provide software that ordinary people can use in their native language. The prerequisite of learning a foreign language is removed and the social and monetary benefits of computer technology are more easily realized. It is easy to imagine how little computer use would be seen in America if the user had to learn Urdu to use an Internet browser. The Web would never have happened.

Linux has a large degree of commitment to Unicode. Support for Unicode is embedded into both the kernel and the code development libraries. It is, for the most part, automatically incorporated into the code using a few simple commands from the program.

The basis of all modern character sets is the American Standard Code for Information Interchange (ASCII), published in 1968 as ANSIX3.4. The notable exception to this is IBM's EBCDIC (Extended Binary Coded Decimal Information Code) that was defined before ASCII. ASCII is a coded character set (CCS), in other words, a mapping from integer numbers to character representations. The ASCII CCS allows the representation of 256 characters with an eight-bit (a base of 2, 0, or 1 value) field or byte ($2^8 = 256$). This is a highly limited CCS that does not allow the representation of all of the characters of the many different languages (like Chinese and Japanese), scientific symbols, or even ancient scripts (runes and hieroglyphics) and music. It would be useful, but entirely impractical to change the size of a byte to allow a larger set of characters to be coded. All computers are based on the eight-bit byte. The solution is a character encoding scheme (CES)
that can represent numbers larger than 256 using a multi-byte sequence of either fixed or variable length. These values are then mapped through the CCS to the characters they represent.

**Unicode definition**

Unicode is usually used as a generic term referring to a two-byte character-encoding scheme. The Unicode CCS 3.1 is officially known as the ISO 10646-1 Universal Multiple Octet Coded Character Set (UCS). Unicode 3.1 adds 44,946 new encoded characters. With the 49,194 already existing characters in Unicode 3.0, the total is now 94,140.

The Unicode CCS utilizes a four-dimensional coding space of 128 three-dimensional groups. Each group has 256 two-dimensional planes. Each plane consists of 256 one-dimensional rows and each row has 256 cells. A cell codes a character at this coding space or the cell is declared unused. This coding concept is called UCS-4; four octets of bits are used to represent each character specifying the group, plane, row and cell.

The first plane (plane 00 of the group 00) is the Basic Multilingual Plane (BMP). The BMP defines characters in general use in alphabetic, syllabic and ideographic scripts as well as various symbols and digits. Subsequent planes are used for additional characters or other coded entities not yet invented. This full range is needed to cope with all of the world's languages; specifically, some East Asian languages that have almost 64,000 characters.

The BMP is used as a two-octet coded character set identified as the UCS-2 form of ISO 10646. ISO 10646 USC-2 is commonly referred to as (and is identical to) Unicode. This BMP, like all UCS planes, contains 256 rows each of 256 cells, and a character is coded at a cell by just the row and cell octets in the BMP. This allows 16-bit coding characters to be used for writing most commercially important languages. USC-2 requires no code page switching, code extensions or code states. USC-2 is a simple method to incorporate Unicode into software, but it is limited in only supporting the Unicode BMP.

To represent a character coding system (CCS) of more than \(2^8 = 256\) characters with eight-bit bytes, a character-encoding scheme (CES) is required.

**Unicode transformations**

In UNIX the most-used CES is UTF-8. It allows for full support of the entire Unicode, all pages and planes, and will still read standard ASCII correctly. The alternatives to UTF-8 are: UCS-4, UTF-16, UTF-7,5, UTF-7, SCSU, HTML, and JAVA.

Unicode Transformation Formats (UTFs) are CESs that support the use of Unicode by mapping a value in a multi-byte code. This article will examine the UTF-8 CCS, the most popular format.

**UTF-8**

The UTF-8 transformation format is becoming a dominant method for exchanging international text information because it can support all of the world's languages and is compatible ASCII. UTF-8 uses variable-width encoding. The characters numbered 0 to 0x7f (127) encode to themselves as a single byte, and larger character values are encoded into 2 to 6 bytes.
Table 1. UTF-8 coding

<table>
<thead>
<tr>
<th>0x00000000 - 0x0000007F:</th>
<th>0xxxxxx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 - 0x0000007F:</td>
<td>0xxxxxx</td>
</tr>
<tr>
<td>0x00000080 - 0x000007FF:</td>
<td>110xxxxx 10xxxxxx</td>
</tr>
<tr>
<td>0x00001000 - 0x0001FFFF:</td>
<td>1110xxxx 10xxxxxx 10xxxxxx</td>
</tr>
<tr>
<td>0x00200000 - 0x03FFFFFF:</td>
<td>11110xxx 10xxxxxx 10xxxxxx 10xxxxxx</td>
</tr>
<tr>
<td>0x04000000 - 0x7FFFFFFF:</td>
<td>1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx</td>
</tr>
</tbody>
</table>

The 10xxxxxx byte is a continuation byte with the xxxxxx bit positions filled with the bits of the character code number in binary representation. The shortest possible multi-byte sequence that can represent the code is used.

UTF-8 coding examples

The Unicode character copyright sign character 0xA9 = 1010 1001 is encoded in UTF-8 as:

```
11000010 10101001 = 0xC2 0xA9
```

and the "not equal" symbol character 0x2260 = 0010 0010 0110 0000 is encoded as:

```
11100010 10001001 10100000 = 0xE2 0x89 0xA0
```

The original values can be seen by taking out the continuation byte values:

```
```

The first byte defines the number of octets to follow, or if it is 7F or less, it is the value of an ASCII equivalent. Starting each octet with 10xxxxxx makes certain that a byte is not mistaken for an ASCII value.

UTF support

Before you start using UTF-8 under Linux make sure the distribution has glibc 2.2 and XFree86 4.0 or newer versions. Earlier versions lack UTF-8 locale support and ISO10646-1 X11 fonts.

Before UTF-8, Linux users used various different language-specific extensions of ASCII like ISO 8859-1 or ISO 8859-2 in Europe, ISO 8859-7 in Greece and KOI-8 / ISO 8859-5/CP1251 in Russia (Cyrillic). This made data exchange problematic and required application software to be programmed for differences between these encodings. Support was incomplete and exchanges untested. Major Linux distributors and application developers are working to have Unicode, primarily in the UTF-8 form, made standard in Linux.

In order to identify a file as Unicode, Microsoft suggested all Unicode files should start with the character ZERO WIDTH NOBREAK SPACE (U+FEFF). This acts as a "signature" or "byte-order mark (BOM)" to identify the encoding and byte-order used in a file. However, Linux/UNIX does not
use BOMs because this would break existing ASCII-file syntax conventions. On POSIX systems, the selected locale identifies the encoding expected in all input and output files of a process.

There are two approaches for adding UTF-8 support to a Linux application. First, data is stored in UTF-8 form everywhere, which results in only a very few software changes (passive). Alternatively, UTF-8 data that has been read is converted into wide-character arrays using standard C library functions (converted). Strings are converted back to UTF-8 when output as with the function wcsrtombs():

Listing 1. wcsrtombs()

```c
#include <wchar.h>
size_t wcsrtombs (char *dest, const wchar_t **src, size_t len, mbstate_t *ps);
```

The method chosen depends upon the nature of the application. Most applications can operate passively. This is why the use of UTF-8 in UNIX is popular. Programs such as `cat` and `echo` need no modification. A byte stream is simply a byte stream and no processing is done on it. ASCII characters and control codes do not change under UTF-8.

Small changes are needed for programs that count characters by counting the bytes. In UTF-8 applications do not count any continuation bytes. The C library `strlen()` function needs to be replaced with the `mbstowcs()` function if a UTF-8 locale has been selected:

Listing 2. mbstowcs() function

```c
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

A common use of `strlen` is to estimate display-width. Chinese and other ideographic characters will occupy two column positions. The `wcwidth()` function is used to test the display-width of each character:

Listing 3. wcwidth() function

```c
#include <wchar.h>
int wcwidth(wchar_t wc);
```

C support for Unicode

Officially, starting with GNU glibc 2.2, the type `wchar_t` is intended to be used only for 32-bit ISO 10646 values, independent of the currently used locale. This is signaled to applications by the definition of the `__STDC_ISO_10646__` macro as required by ISO C99. The `__STDC_ISO_10646__` is defined to indicate that `wchar_t` is Unicode. The exact value is a decimal constant of the form yyyymmL. For example, use:

Listing 4. Indicating that wchar_t is Unicode

```c
#define __STDC_ISO_10646__ Â 200104L
```
to indicate that values of type wchar_t are the coded representations of the characters defined by
ISO/IEC 10646 and all amendments and technical corrigenda as of the specified year and month.

It would be utilized as shown in this example, which uses the macro to determine the method of
writing double quotes in ISO C99 portable code:

**Listing 5. Determining the method of writing double quotes**

```c
#if __STDC_ISO_10646__
  printf("%lc", 0x201c);
#else
  putchar('"');
#endif
```

**The locale**

The proper way to activate UTF-8 is the POSIX locale mechanism. A locale is a configuration
setting that contains information about culture-specific conventions of software behavior. This
includes character encoding, date/time notation, sorting rules and measurement systems. The
names of locales usually consist of ISO 639-1 language, ISO 3166-1 country codes and optional
encoding names and other qualifiers. You can get a list of all locales installed on your system
(usually in `/usr/lib/locale/`) with the command `locale -a`.

If a UTF-8 locale is not preinstalled, you can generate it using the `localedef` command. To
generate and activate a German UTF-8 locale for a specific user, use the following statements:

**Listing 6. Generating a locale for a specific user**

```bash
localedef -v -c -i de_DE -f UTF-8 $HOME/local/locale/de_DE.UTF-8
export LOCPATH=$HOME/local/locale
export LANG=de_DE.UTF-8
```

It is sometimes useful to add a UTF-8 locale for all users. This can be by root using the following
instruction:

**Listing 7. Generating a locale for all users**

```bash
localedef -v -c -i de_DE -f UTF-8 /usr/share/locale/de_DE.UTF-8
```

To make it the default locale for every user add into the `/etc/profile` file the following line:

**Listing 8. Setting the default locale for all users**

```bash
export LANG=de_DE.UTF-8
```

The behavior of functions that deal with multi-byte character code sequences depend on the
LC_CTYPE category of the current locale; it determines locale-dependent multi-byte encoding.
The value LANG=de_DE (German) will cause output to be formatted in ISO 8859-1. The value
LANG=de_DE.UTF-8 will format the output to UTF-8. The locale setting will cause the %ls format specifier in printf to call the wcsrtombs() function in order to convert the wide character argument string into the locale-dependent multi-byte encoding. Country identifiers in the locales such as: LC_CTYPE= en_GB (English in Great Britain) and LC_CTYPE= en_AU (English in Australia) differ only in the LC_MONETARY category for the name of currency and the rules for printing monetary amounts.

Set the environment variable LANG to the name of your preferred locale. When a C program executes the setlocale() function:

Listing 9. setlocale() function

```c
#include <stdio.h>
#include <locale.h>
//char *setlocale(int category, const char *locale);
int main()
{
  if (!setlocale(LC_CTYPE, ""))
  {
    fprintf(stderr, "Locale not specified. Check LANG, LC_CTYPE, LC_ALL. ");
    return 1;
  }
}
```

The library will test the environment variables LC_ALL, LC_CTYPE, and LANG in that order. The first one of these that has a value will determine which locale data is loaded for the LC_CTYPE category. The locale data is split up into separate categories. The LC_CTYPE value defines character encoding and LC_COLLATE defines the sorting order. The LANG environment variable is used to set the default locale for all categories, but LC_* variables can be used to override individual categories.

You can query the name of the character encoding in your current locale with the command `locale charmap`. This should say UTF-8 if you successfully picked a UTF-8 locale in the LC_CTYPE category. The command `locale -m` provides a list with the names of all installed character encodings.

If you use exclusively C library multi-byte functions to do all the conversion between the external character encoding and the wchar_t encoding that you use internally, then the C library will take care of using the right encoding according to LC_CTYPE. The program does not even have to be explicitly coded to the current multi-byte encoding.

If the application is required to be specifically aware of the UTF-8 (or other) conversion method and not use the libc multi-byte functions, the application has to find out whether to activate the UTF-8 mode. X/Open-compliant systems with a <langinfo.h> library header can use the code:

Listing 10. Detecting whether the current locale uses the UTF-8 encoding

```c
BOOL utf8_mode = FALSE;
if( ! strcmp(nl_langinfo(CODESET), "UTF-8")
  utf8_mode = TRUE;
```
to detect if the current locale uses the UTF-8 encoding. The `setlocale(LC_CTYPE, "")` function must be called to set the locale according to the environment variables first. The `nl_langinfo(CODESET)` function is also what the `locale charmap` command calls to find the name of the encoding specified by the current locale.

Another method that could be used is to query the locale environment variables:

**Listing 11. Querying the locale environment variables**

```c
char *s;
BOOL utf8_mode = FALSE;
if ((s = getenv("LC_ALL")) || (s = getenv("LC_CTYPE")) || (s = getenv("LANG")))
{
    if (strstr(s, "UTF-8"))
        utf8_mode = TRUE;
}
```

This test assumes the UTF-8 locales have the value "UTF-8" in their name, which is not always true, so the `nl_langinfo()` method should be used.

**Summary**

To support the world's languages, a character coding system (CCS) of more than the $2^8 = 256$ characters of ASCII (the extended version using an unsigned byte) with an eight-bit byte character-encoding scheme (CES) is required. Unicode does this as a CCS with a four-dimensional coding space of 128 three-dimensional groups with 94,140 defined character values that is supported with numerous CES methods, the more popular of which, in Linux, is the Unicode transformation format UTF-8.
Related topics

- Visit the [Unicode home page](http://www.unicode.org) of the Unicode Consortium, which is responsible for defining the behavior and relationships between Unicode characters, and providing technical information to implementers.
- The [International Organization for Standardization (ISO)](http://www.iso.org) is a worldwide federation of national standards bodies from 140 countries.
- Read the [IANA Charset Registration Procedures](http://www.iana.org) from IANA (Internet Assigned Numbers Authority).
- Look at a [graphic representation of the Roadmap to the BMP, Plane 0 of the UCS](http://www.unicode.org/standard/standard-dev/roadmap). These tables comprise a real-size map of Plane 0, the BMP (Basic Multilingual Plane) of the UCS (Universal Character Set). Everson Gunn Teoranta is a software and publishing company supporting minority-language communities since 1990, founded by Michael Everson and Marion Gunn.
- Browse the [UTF-8 and Unicode FAQ for UNIX/Linux](http://www.utf8.org), Markus Kuhn's comprehensive one-stop information resource on how you can use Unicode/UTF-8 in POSIX systems (Linux, UNIX).
- Check out [Solution Given by the Universal Character Set](http://www tùtality.com) by Technology Appraisals Ltd, which provides independent high-quality information, education, and training on e-business systems, electronic information delivery, XML, networking, and IT security.
- Consult the Linux Programmer’s Manual’s [UTF-8 - an ASCII compatible multi-byte Unicode encoding](http://www.pathname.com/utf8/).
- Read the [Unicode Standard Annex #15 Unicode Normalization Forms](http://www.unicode.org/standard/standard-dev/unicode-class), a document that describes specifications for four normalized forms of Unicode text. With these forms, equivalent text (canonical or compatibility) will have identical binary representations. When implementations keep strings in a normalized form, they can be assured that equivalent strings have a unique binary representation.
- Read about [setlocale()](http://www.pathname.com/stdlib_3/chap17.html) to change or query current locale, from the MKS Toolkit Documentation. MKS Software is the leading provider of Windows automation tools for system administration and development in a pure Windows or mixed UNIX/Linux and Windows environment.