The Unicode HOWTO

Table of Contents

Table of Contents

The Unicode HOWTO

Bruno Haible, <haible@clisp.cons.org>

v0.12, 19 October 1999

This document describes how to change your Linux system so it uses UTF−8 as text encoding. − This is work in progress. Any tips, patches, pointers, URLs are very welcome.

Next Previous Contents Next Previous Contents

1. Introduction

1.1 Why Unicode?

People in different countries use different characters to represent the words of their native languages. Nowadays most applications, including email systems and web browsers, are 8−bit clean, i.e. they can operate on and display text correctly provided that it is represented in an 8−bit character set, like ISO−8859−1.

There are far more than 256 characters in the world − think of cyrillic, hebrew, arabic, chinese, japanese, korean and thai −, and new characters are being invented now and then. The problems that come up for users are:

- It is impossible to store text with characters from different character sets in the same document. For example, I can cite russian papers in a German or French publication if I use TeX, xdvi and PostScript, but I cannot do it in plain text.
- As long as every document has its own character set, and recognition of the character set is not automatic, manual user intervention is inevitable. For example, in order to view the homepage of the XTeamLinux distribution http://www.xteamlinux.com.cn/ I have to tell Netscape that the web page is coded in GB2312.
- New symbols like the Euro are being invented. ISO has issued a new standard ISO−8859−15, which is mostly like ISO−8859−1 except that it removes some rarely used characters (the old currency sign)

> and replaced it with the Euro sign. If users adopt this standard, they have documents in different character sets on their disk, and they start having to think about it daily. But computers should make things simpler, not more complicated.

The solution of this problem is the adoption of a world−wide usable character set. This character set is Unicode http://www.unicode.org/. For more info about Unicode, do `` `man 7 unicode' `` (manpage contained in the ldpman−1.20 package).

1.2 Unicode encodings

This reduces the user's problem of dealing with character sets to a technical problem: How to transport Unicode characters using the 8−bit bytes? 8−bit units are the smallest addressing units of most computers and also the unit used by TCP/IP network connections. The use of 1 byte to represent 1 character is, however, an accident of history, caused by the fact that computer development started in Europe and the U.S. where 96 characters were found to be sufficient for a long time.

There are basically four ways to encode Unicode characters in bytes:

UTF−8

128 characters are encoded using 1 byte (the ASCII characters). 1920 characters are encoded using 2 bytes (Roman, Greek, Cyrillic, Coptic, Armenian, Hebrew, Arabic characters). 63488 characters are encoded using 3 bytes (Chinese and Japanese among others). The other 2147418112 characters (not assigned yet) can be encoded using 4, 5 or 6 characters. For more info about UTF−8, do `` `man 7 utf−8' `` (manpage contained in the ldpman−1.20 package).

UCS−2

Every character is represented as two bytes. This encoding can only represent the first 65536 Unicode characters.

UTF−16

This is an extension of UCS−2 which can represent 1114112 Unicode characters. The first 65536 Unicode characters are represented as two bytes, the other ones as four bytes.

UCS−4

Every character is represented as four bytes.

The space requirements for encoding a text, compared to encodings currently in use (8 bit per character for European languages, more for Chinese/Japanese/Korean), is as follows. This has an influence on disk storage space and network download speed.

UTF−8

No change for US ASCII, just a few percent more for ISO−8859−1, 50% more for Chinese/Japanese/Korean, 100% more for Greek and Cyrillic.

UCS−2 and UTF−16

No change for Chinese/Japanese/Korean. 100% more for US ASCII and ISO−8859−1, Greek and Cyrillic.

UCS−4

100% more for Chinese/Japanese/Korean. 300% more for US ASCII and ISO−8859−1, Greek and Cyrillic.

Given the penalty for US and European documents caused by UCS−2, UTF−16, and UCS−4, it seems unlikely that these encodings have a potential for wide−scale use. The Microsoft Win32 API supports the UCS−2 encoding since 1995 (at least), yet this encoding has not been widely adopted for documents − SJIS remains prevalent in Japan.

UTF−8 on the other hand has the potential for wide−scale use, since it doesn't penalize US and European users, and since many text processing programs don't need to be changed for UTF−8 support.

In the following, we will describe how to change your Linux system so it uses UTF−8 as text encoding.

Footnotes for C/C++ developers

The Microsoft Win32 approach makes it easy for developers to produce Unicode versions of their programs: You "#define UNICODE" at the top of your program and then change many occurrences of `` `char' `` to `` `TCHAR' ``, until your program compiles without warnings. The problem with it is that you end up with two versions of your program: one which understands UCS−2 text but no 8−bit encodings, and one which understands only old 8−bit encodings.

Moreover, there is an endianness issue with UCS−2 and UCS−4. The IANA character set registry says about ISO−10646−UCS−2: "this needs to specify network byte order: the standard does not specify". Network byte order is big endian. Whereas Microsoft, in its C/C++ development tools, recommends to use machine−dependent endianness (i.e. little endian on ix86 processors) and either a byte−order mark at the beginning of the document, or some statistical heuristics(!).

The UTF−8 approach on the other hand keeps `` `char*' `` as the standard C string type. As a result, your program will handle US ASCII text, independently of any environment variables, and will handle both ISO−8859−1 and UTF−8 encoded text provided the LANG environment variable is set accordingly.

1.3 Related resources

Markus Kuhn's very up−to−date resource list:

- http://www.cl.cam.ac.uk/~mgk25/unicode.html
- http://www.cl.cam.ac.uk/~mgk25/ucs−fonts.html

Roman Czyborra's overview of Unicode, UTF−8 and UTF−8 aware programs:
http://czyborra.com/utf/#UTF−8

Some example UTF−8 files:

- The files `quickbrown.txt`, `utf−8−test.txt`, `utf−8−demo.txt` in the `examples` directory of Markus Kuhn's ucs−fonts package http://www.cl.cam.ac.uk/~mgk25/download/ucs−fonts.tar.gz
- ftp://ftp.cs.su.oz.au/gary/x−utf8.html
- The file `iso10646` in the Kosta Kostis' trans−1.1.1 package ftp://ftp.nid.ru/pub/os/unix/misc/trans111.tar.gz
- ftp://ftp.dante.de/pub/tex/info/lwc/apc/utf8.html
- http://www.cogsci.ed.ac.uk/~richard/unicode−sample.html

Next Previous ContentsNextPreviousContents

2. Display setup

We assume you have already adapted your Linux console and X11 configuration to your keyboard and locale. This is explained in the Danish/International HOWTO, and in the other national HOWTOs: Finnish, French, German, Italian, Polish, Slovenian, Spanish, Cyrillic, Hebrew, Chinese, Thai, Esperanto. But please do not follow the advice given in the Thai HOWTO, to pretend you were using ISO−8859−1 characters (U0000..U00FF) when what you are typing are actually Thai characters (U0E01..U0E5B). Doing so will only cause problems when you switch to Unicode.

2.1 Linux console

I'm not talking much about the Linux console here, because on those machines on which I don't have xdm running, I use it only to type my login name, my password, and "xinit".

Anyway, the kbd−0.99 package ftp://sunsite.unc.edu/pub/Linux/system/keyboards/kbd−0.99.tar.gz and a heavily extended version, the console−tools−0.2.2 package ftp://sunsite.unc.edu/pub/Linux/system/keyboards/console−tools−0.2.2.tar.gz contains in the kbd−0.99/src/

(or console−tools−0.2.2/screenfonttools/) directory two programs: `unicode_start' and `unicode_stop'. When you call `unicode_start', the console's screen output is interpreted as UTF−8. Also, the keyboard is put into Unicode mode (see "man kbd_mode"). In this mode, Unicode characters typed as Alt−x1 ... Alt−xn (where x1,...,xn are digits on the numeric keypad) will be emitted in UTF−8. If your keyboard or, more precisely, your normal keymap has non−ASCII letter keys (like the German Umlaute) which you would like to be CapsLockable, you need to apply the kernel patch linux−2.2.9−keyboard.diff or linux−2.3.12−keyboard.diff.

You will want to use display characters from different scripts on the same screen. For this, you need a Unicode console font. The ftp://sunsite.unc.edu/pub/Linux/system/keyboards/console−data−1999.08.29.tar.gz package contains a font (LatArCyrHeb−{08,14,16,19}.psf) which covers Latin, Cyrillic, Hebrew, Arabic scripts. It covers ISO 8859 parts 1,2,3,4,5,6,8,9,10 all at once. To install it, copy it to /usr/lib/kbd/consolefonts/ and execute "/usr/bin/setfont /usr/lib/kbd/consolefonts/LatArCyrHeb−14.psf".

If you want cut&paste to work with UTF−8 consoles, you need the patch linux−2.3.12−console.diff from Edmund Thomas Grimley Evans and Stanislav Voronyi.

2.2 X11 Foreign fonts

Don't hesitate to install Cyrillic, Chinese, Japanese etc. fonts. Even if they are not Unicode fonts, they will help in displaying Unicode documents: at least Netscape Communicator 4 and Java will make use of foreign fonts when available.

The following programs are useful when installing fonts:

- "mkfontdir directory" prepares a font directory for use by the X server, needs to be executed after installing fonts in a directory.
- "xset fp+ directory" adds a directory to the X server's current font path. To add a directory permanently, add a "FontPath" line to your /etc/XF86Config file, in section "Files".
- "xset fp rehash" needs to be executed after calling mkfontdir on a directory that is already contained in the X server's current font path.
- "xfontsel" allows you to browse the installed fonts by selecting various font properties.
- "xlsfonts −fn fontpattern" lists all fonts matching a font pattern. Also displays various font properties. In particular, "xlsfonts −ll −fn font" lists the font properties CHARSET_REGISTRY and CHARSET_ENCODING, which together determine the font's encoding.
- "xfd −fn font" displays a font page by page.

The following fonts are freely available (not a complete list):

- The ones contained in XFree86, sometimes packaged in separate packages. For example, SuSE has only normal 75dpi fonts in the base `xf86' package. The other fonts are in the packages `xfnt100', `xfntbig', `xfntcyr', `xfntscl'.
- The Emacs international fonts, ftp://ftp.gnu.org/pub/gnu/intlfonts/intlfonts−1.1.tar.gz As already mentioned, they are useful even if you prefer XEmacs to GNU Emacs or don't use any Emacs at all.

2.3 X11 Unicode fonts

Application wishing to display text belonging to different scripts (like Cyrillic and Greek) at the same time, can do so by using different X fonts for the various pieces of text. This is what Netscape Communicator and Java do. However, this approach is more complicated, because instead of working with `Font' and `XFontStruct', the programmer has to deal with `XFontSet', and also because not all fonts in the font set need to have the same dimensions.

- Markus Kuhn has assembled fixed−width 75dpi fonts with Unicode encoding covering Latin, Greek, Cyrillic, Armenian, Georgian, Hebrew, Symbol scripts. They cover ISO 8859 parts 1,2,3,4,5,7,8,9,10,13,14,15 all at once. This font is required for running xterm in utf−8 mode. http://www.cl.cam.ac.uk/~mgk25/download/ucs−fonts.tar.gz
- Roman Czyborra has assembled an 8x16 / 16x16 75dpi font with Unicode encoding covering a huge part of Unicode. Download unifont.hex.gz and hex2bdf from http://czyborra.com/unifont/. It is not fixed−width: 8 pixels wide for European characters, 16 pixels wide for Chinese characters. Installation instructions:

```
$ gunzip unifont.hex.gz
$ hex2bdf < unifont.hex > unifont.bdf
$ bdftopcf −o unifont.pcf unifont.bdf
$ gzip −9 unifont.pcf
# cp unifont.pcf.gz /usr/X11R6/lib/X11/fonts/misc
# cd /usr/X11R6/lib/X11/fonts/misc
# mkfontdir
# xset fp rehash
```

- Primoz Peterlin has assembled an ETL family fonts covering Latin, Greek, Cyrillic, Armenian, Georgian, Hebrew scripts. ftp://ftp.x.org/contrib/fonts/etl−unicode.tar.gz Use the "bdftopcf" program in order to install it.

2.4 Unicode xterm

xterm is part of X11R6 and XFree86, but is maintained separately by Tom Dickey. http://www.clark.net/pub/dickey/xterm/xterm.html Newer versions (patch level 109 and above) contain support for converting keystrokes to UTF−8 before sending them to the application running in the xterm, and for displaying Unicode characters that the application outputs as UTF−8 byte sequence.

To get an UTF−8 xterm running, you need to:

- Fetch http://www.clark.net/pub/dickey/xterm/xterm.tar.gz,
- Configure it by calling "./configure −−enable−wide−chars ...", then compile and install it.
- Have a Unicode fixed−width font installed. Markus Kuhn's ucs−fonts.tar.gz (see above) is made for this.
- Start "xterm −u8 −fn fixed". The option "−u8" turns on Unicode and UTF−8 handling. The "fixed" font is Markus Kuhn's Unicode font.
- Take a look at the sample files contained in Markus Kuhn's ucs−fonts package:

```
$ cd .../ucs-fonts
$ cat quickbrown.txt
$ cat utf-8-demo.txt
```

  You should be seeing (among others) greek and russian characters.
- To make xterm come up with UTF−8 handling each time it is started, add the line XTerm*utf8: 1 to your $HOME/.Xdefaults (for yourself only). I don't recommend changing the system−wide /usr/X11R6/lib/X11/app−defaults/XTerm, because then your changes will be erased next time you upgrade to a new XFree86 version.
- If you also changed the font name, you need a line of the form *VT100*font: your−font in your $HOME/.Xdefaults. For "fixed" it is not needed, because "fixed" is the default anyway.

2.5 Miscellaneous

A small program which tests whether a Linux console or xterm is in UTF−8 mode can be found in the ftp://sunsite.unc.edu/pub/Linux/system/keyboards/x−lt−1.18.tar.gz package by Ricardas Cepas, files testUTF−8.c and testUTF8.c.

NextPreviousContentsNextPreviousContents

3. Locale setup

3.1 Files & the kernel

You can now already use any Unicode characters in file names. No kernel or file utilities need modifications. This is because file names in the kernel can be anything not containing a null byte, and '/' is used to delimit subdirectories. When encoded using UTF−8, non−ASCII characters will never be encoded using null bytes or slashes. All that happens is that file and directory names occupy more bytes than they contain characters. For example, a filename consisting of five greek characters will appear to the kernel as a 10−byte filename. The kernel does not know (and does not need to know) that these bytes are displayed as greek.

This is the general theory, as long as your files stay inside Linux. On filesystems which are used from other operating systems, you have mount options to control conversion of filenames to/from UTF−8:

- The "vfat" filesystems has a mount option "utf8". See file:/usr/src/linux/Documentation/filesystems/vfat.txt. When you give an "iocharset" mount option different from the default (which is "iso8859−1"), the results with and without "utf8" are not consistent. Therefore I don't recommend the "iocharset" mount option.
- The "msdos", "umsdos" filesystems have the same mount option, but it appears to have no effect.

- The "iso9660" filesystem has a mount option "utf8". See file:/usr/src/linux/Documentation/filesystems/isofs.txt.
- Since Linux 2.2.x kernels, the "ntfs" filesystem has a mount option "utf8". See file:/usr/src/linux/Documentation/filesystems/ntfs.txt.

The other filesystems (nfs, smbfs, ncpfs, hpfs, etc.) don't convert filenames; therefore they support Unicode file names in UTF−8 encoding only if the other operating system supports them. Recall that to enable a mount option for all future remount, you add to the fourth column of the corresponding /etc/fstab line.

3.2 Ttys & the kernel

Ttys are some kind of bidirectional pipes between two program, allowing fancy features like echoing or command−line editing. When in an xterm, you execute the "cat" command without arguments, you can enter and edit any number of lines, and they will be echoed back line by line. The kernel's editing actions are not correct, especially the Backspace (erase) key and the tab bey are not treated correctly.

To fix this, you need to:

- apply the kernel patch linux−2.0.35−tty.diff or linux−2.2.9−tty.diff or linux−2.3.12−tty.diff and recompile your kernel,
- if you are using glibc2, apply the patch glibc211−tty.diff and recompile your libc (or if you are not so adventurous, it is sufficient to patch an already installed include file: glibc−tty.diff),
- apply the patch stty.diff to GNU sh−utils−1.16b, and rebuild the "stty" program, then test it using "stty −a" and "stty iutf8".
- add the command "stty iutf8" to the "unicode_start" script, and add the command "stty −iutf8" to the "unicode_stop script.
- apply the patch xterm.diff to xterm−109, and rebuild "xterm", then test it by starting "xterm −u8"/"xterm +u8" and running "stty −a" and interactive "cat" inside it.

To make this fix persistent across rlogin and telnet, you also need to:

- Define new values for the TERM environment variable, "linux−utf8" as an alias to "linux", and "xterm−utf8" as an alias to "xterm". If your system has the ncurses library and the /usr/lib/terminfo (or /usr/share/terminfo) database, do this by running

  ```
  $ tic linux-utf8.terminfo
  $ tic xterm-utf8.terminfo
  ```

  as non−root (this will create the terminfo entries in your $HOME/.terminfo directory). Here are linux−utf8.terminfo and xterm−utf8.terminfo. I don't recommend running this as root, because it will create the terminfo entries in /usr/lib/terminfo where they might be erased next time you upgrade your system. If your system has an /etc/termcap file, you should also edit that file: copy the linux and xterm entries and give them the new names "linux−utf8" and "xterm−utf8". For an example, see termcap.diff.
- Each time you call "unicode_start" or "unicode_stop" from the console, also execute "export TERM=linux−utf8" or "export TERM=linux", respectively.
- Apply the patch xterm2.diff to xterm−109, rebuild "xterm", and remove any "XTerm*termName"

line from /usr/X11R6/lib/X11/app−defaults/XTerm and $HOME/.Xdefaults. Now xterm sets the TERM variable to "xterm−utf8" instead of "xterm" when running in UTF−8 mode.
- Apply the patches netkit.diff, netkitb.diff and telnet.diff and rebuild "rlogind" and "telnetd". Now rlogin and telnet put the tty into UTF−8 editing mode whenever the TERM environment variable is "linux−utf8" or "xterm−utf8".

3.3 General data conversion

You will need a program to convert your locally (probably ISO−8859−1) encoded texts to UTF−8. (The alternative would be to keep using texts in different encodings on the same machine; this is not fun in the long run.) One such program is `iconv', which comes with glibc−2.1. Simply use

```
$ iconv --from-code=ISO-8859-1 --to-code=UTF-8 < old_file > new_file
```

Here are two handy shell scripts, called "i2u" i2u.sh (for ISO to UTF conversion) and "u2i" u2i.sh (for UTF to ISO conversion). Adapt according to your current 8−bit character set.

If you don't have glibc−2.1 and iconv installed, you can use GNU recode 3.5 instead. "i2u" i2u_recode.sh is "recode ISO−8859−1..UTF−8", and "u2i" u2i_recode.sh is "recode UTF−8..ISO−8859−1". ftp://ftp.iro.umontreal.ca/pub/recode/recode−3.5.tar.gzftp://ftp.gnu.org/pub/gnu/recode/recode−3.5.tar.gz Notes: You need GNU recode 3.5 or newer. To compile GNU recode 3.5 on platforms without glibc2 (i.e. on all platforms except recent Linux systems), you need to configure it with −−disable−nls, otherwise it won't link.

Or you can also use CLISP instead. Here are "i2u" i2u.lsp and "u2i" u2i.lsp written in Lisp. Note: You need a CLISP version from July 1999 or newer. ftp://clisp.cons.org/pub/lisp/clisp/source/clispsrc.tar.gz.

Other data conversion programs, less powerful than GNU recode, are `trans' ftp://ftp.informatik.uni−erlangen.de/pub/doc/ISO/charsets/trans113.tar.gz, `tcs' from the Plan9 operating system ftp://ftp.informatik.uni−erlangen.de/pub/doc/ISO/charsets/tcs.tar.gz, and `utrans'/`uhtrans'/`hutrans' ftp://ftp.cdrom.com/pub/FreeBSD/distfiles/i18ntools−1.0.tar.gz by G. Adam Stanislav <adam@whizkidtech.net>.

3.4 Locale environment variables

You may have the following environment variables set, containing locale names:

LANGUAGE

override for LC_MESSAGES, used by GNU gettext only

LC_ALL

override for all other LC_* variables

LC_CTYPE, LC_MESSAGES, LC_COLLATE, LC_NUMERIC, LC_MONETARY, LC_TIME

individual variables for: character types and encoding, natural language messages, sorting rules, number formatting, money amount formatting, date and time display

LANG

default value for all LC_* variables

(See `man 7 locale' for a detailed description.)

Each of the LC_* and LANG variables can contain a locale name of the following form:

language[_territory[.codeset]][@modifier]

where language is an ISO 639 language code (lower case), territory is an ISO 3166 country code (upper case), codeset denotes a character set, and modifier stands for other particular attributes (for example indicating a particular language dialect, or a nonstandard orthography).

LANGUAGE can contain several locale names, separated by colons.

In order to tell your system and all applications that you are using UTF−8, you need to add a codeset suffix of UTF−8 to your locale names. For example, if you were using

```
LANGUAGE=de:fr:en
LC_CTYPE=de_DE
```

you would change this to

```
LANGUAGE=de.UTF−8:fr.UTF−8:en.UTF−8
LC_CTYPE=de_DE.UTF−8
```

3.5 Creating the locale support files

If you have glibc−2.1 or glibc−2.1.1 or glibc−2.1.2 installed, first check using "localedef −−help" that the system directory for character maps is /usr/share/i18n/charmaps. Then apply to the file /usr/share/i18n/charmaps/UTF8 the patch glibc21.diff or glibc211.diff or glibc212.diff, respectively. Then create the support files for each UTF−8 locale you intend to use, for example:

```
$ localedef −v −c −i de_DE −f UTF8 /usr/share/locale/de_DE.UTF−8
```

You typically don't need to create locales named "de" or "fr" without country suffix, because these locales are normally only used by the LANGUAGE variable and not by the LC_* variables, and LANGUAGE is only used as an override for LC_MESSAGES.

3.6 Adding support to the C library

The glibc−2.2 will support multibyte locales, in particular the UTF−8 locales created above. But glibc−2.1 or glibc−2.1.1 do not really support it. Therefore the only real effect of the above creation of the /usr/share/locale/de_DE.UTF−8/* files is that `setlocale(LC_ALL,"")' will return "de_DE.UTF−8", according to your environment variables, instead of stripping off the ".UTF−8" suffix.

To add support for the UTF−8 locale, you should build and install the `libutf8_plug.so' library, from libutf8−0.5.2.tar.gz. Then you can set the LD_PRELOAD environment variable to point to the installed library:

```
        $ export LD_PRELOAD=/usr/local/lib/libutf8_plug.so
```

Then, in every program launched with this environment variable set, the functions in libutf8_plug.so will override the original ones in /lib/libc.so.6. For more info about LD_PRELOAD, see "man 8 ld.so".

This entire thing will not be necessary any more once glibc−2.2 comes out.

3.7 Converting the message catalogs

Now let's fill these new locales with content. The following /bin/sh commands can convert your message catalogs to UTF−8 format. They must be run as root, and require the programs `msgfmt' and `msgunfmt' from GNU gettext−0.10.35 to be installed. convert−msgcat.sh

This too will not be necessary any more once glibc−2.2 comes out, because by then, the gettext function will convert the strings appropriately from the translator's character set to the user's character set, using either iconv or librecode.

NextPreviousContentsNextPreviousContents

4. Specific applications

4.1 Networking

rlogin

is fine with the above mentioned patches.

telnet

telnet is not 8−bit clean by default. In order to be able to send Unicode keystrokes to the remote host, you need to set telnet into "outbinary" mode. There are two ways to do this:

```
$ telnet −L <host>
```

and

```
$ telnet
telnet> set outbinary
telnet> open <host>
```

Additionally, use the above mentioned patches.

4.2 Browsers

Netscape

Netscape 4.05 or newer can display HTML documents in UTF−8 encoding. All a document needs is the following line between the <head> and </head> tags:

```
<meta http−equiv="Content−Type" content="text/html; charset=UTF−8">
```

Netscape 4.05 or newer can also display HTML and text files in UCS−2 encoding with byte−order mark.

http://www.netscape.com/computing/download/

lynx

lynx−2.8 has an options screen (key 'O') which permits to set the display character set. When running in an xterm or Linux console in UTF−8 mode, set this to "UNICODE UTF−8".

Now, again, all a document needs is the following line between the <head> and </head> tags:

```
<meta http−equiv="Content−Type" content="text/html; charset=UTF−8">
```

When you are viewing text files in UTF−8 encoding, you also need to pass the command−line option "−assume_local_charset=UTF−8" (affects only file:/... URLs) or "−assume_charset=UTF−8" (affects all URLs). In lynx−2.8.2 you can alternatively, in the options screen (key 'O'), change the assumed document character set to "utf−8".

There is also an option in the options screen, to set the "preferred document character set". But it has no effect, at least with file:/... URLs and with http://... URLs served by apache−1.3.0.

There is a spacing and line−breaking problem, however. (Look at the russian section of x−utf8.html, or at utf−8−demo.txt.)

Also, in lynx−2.8.2, configured with −−enable−prettysrc, the nice colour scheme does not work correctly any more when the display character set has been set to "UNICODE UTF−8". This is fixed by a simple patch lynx282.diff.

The Lynx developers say: "For any serious use of UTF−8 screen output with lynx, compiling with slang lib and −DSLANG_MBCS_HACK is still recommended."

ftp://ftp.gnu.org/pub/gnu/lynx/lynx−2.8.2.tar.gzhttp://lynx.browser.org/http://www.slcc.edu/lynx/ftp://lynx.isc.org/

Test pages

Some test pages for browsers can be found at the pages of Alan Wood http://www.hclrss.demon.co.uk/unicode/#links and James Kass http://home.att.net/~jameskass/.

4.3 Editors

yudit

yudit by Gáspár Sinai http://czyborra.com/yudit/ is a first−class unicode text editor for the X Window System. It supports simultaneous processing of many languages, input methods, conversions for local character standards. It has facilities for entering text in all languages with only an English keyboard, using keyboard configuration maps.

It can be compiled in three versions: Xlib GUI, KDE GUI, or Motif GUI.

Customization is very easy. Typically you will first customize your font. From the font menu I chose "Unicode". Then, since the command "xlsfonts '*−*−iso10646−1'" still showed some ambiguity, I chose a font size of 13 (to match Markus Kuhn's 13−pixel fixed font).

Next, you will customize your input method. The input methods "Straight", "Unicode" and "SGML" are most remarkable. For details about the other built−in input methods, look in /usr/local/share/yudit/data/.

To make a change the default for the next session, edit your $HOME/.yuditrc file.

The general editor functionality is limited to editing, cut&paste and search&replace. No undo.

mined98

mined98 is a small text editor by Michiel Huisjes, Achim Müller and Thomas Wolff. http://www.inf.fu−berlin.de/~wolff/mined.html It lets you edit UTF−8 or 8−bit encoded files, in an UTF−8 or 8−bit xterm. It also has powerful capabilities for entering Unicode characters.

When it is running in xterm or Linux console in UTF−8 mode, you should set the environment variable `utf8_term`, or call mined with the command−line option `-U`.

mined lets you edit both 8−bit encoded and UTF−8 encoded files. By default it uses an autodetection heuristic. If you don't want to rely on heuristics, pass the command−line option `-u` when editing an UTF−8 file, or `+u` when editing an 8−bit encoded file. You can change the interpretation at any time from within the editor: It displays the encoding ("L:h" for 8−bit, "U:h" for UTF−8) in the menu line. Click on the first of these characters to change it.

A few caveats:

- The Linux binary in the distribution is out of date and does not support UTF−8. You have to rebuild a binary from the source. Then install src/mined as /usr/local/bin/mined, and install doc/mined.help as /usr/local/man/cat1/mined.1, so that `ESC h` will find it.
- mined ignores your "stty erase" setting. When your backspace key sends ASCII code 127, and you have set "stty erase ^?", −− these are the ultimately correct settings −− you have to call mined with option `-B` in order to get the backspace key erase a character to the left of the cursor.
- The "Home", "End", "Delete" keys do not work.

vim

vim (as of version 5.4m) has some support for multi−byte locales, but only as far as the X library has the same support, and only for encodings with at most two bytes per character (i.e. ISO−2022 encodings). No UTF−8 support.

emacs

First of all, you should read the section "International Character Set Support" (node "International") in the Emacs manual. In particular, note that you need to start Emacs using the command

```
$ emacs -fn fontset-standard
```

so that it will use a font set comprising a lot of international characters.

In the short term, the emacs−utf package http://www.cs.ust.hk/faculty/otfried/Mule/ by Otfried Cheong provides a "unicode−utf8" encoding to Emacs. After compiling the program "utf2mule" and installing it somewhere in your $PATH, also install unicode.el, muleuni−1.el, unicode−char.el somewhere, and add the lines

```
(setq load-path (cons "/home/user/somewhere/emacs" load-path))
(if (not (string-match "XEmacs" emacs-version))
  (progn
    (require 'unicode)
    (if (eq window-system 'x)
      (progn
        (create-fontset-from-fontset-spec
          "-misc-fixed-medium-r-normal-*-12-*-*-*-*-*-fontset-standard")
        (create-fontset-from-fontset-spec
          "-misc-fixed-medium-r-normal-*-13-*-*-*-*-*-fontset-standard")
        (create-fontset-from-fontset-spec
          "-misc-fixed-medium-r-normal-*-14-*-*-*-*-*-fontset-standard")
        (create-fontset-from-fontset-spec
          "-misc-fixed-medium-r-normal-*-15-*-*-*-*-*-fontset-standard")
        (create-fontset-from-fontset-spec
          "-misc-fixed-medium-r-normal-*-16-*-*-*-*-*-fontset-standard")
        (create-fontset-from-fontset-spec
          "-misc-fixed-medium-r-normal-*-18-*-*-*-*-*-fontset-standard")))))
```

to your $HOME/.emacs file. To activate any of the font sets, use the Mule menu item "Set Font/FontSet" or Shift−down−mouse−1. Currently the font sets with height 15 and 13 have the best Unicode coverage, due to Markus Kuhn's 9x15 and 6x13 fonts. In order to open an UTF−8 encoded file, you will type

```
M-x universal-coding-system-argument unicode-utf8 RET
M-x find-file filename RET
```

or

```
C-x RET c unicode-utf8 RET
C-x C-f filename RET
```

Note that this works with Emacs in windowing mode only, not in terminal mode.

Richard Stallman plans to add integrated UTF−8 support to Emacs in the long term.

xemacs

(This section is written by Gilbert Baumann.)

Here is how to teach XEmacs (20.4 configured with MULE) the UTF−8 encoding. Unfortunately you need its sources to be able to patch it.

First you need these files provided by Tomohiko Morioka:

http://turnbull.sk.tsukuba.ac.jp/Tools/XEmacs/xemacs−21.0−b55−emc−b55−ucs.diff and http://turnbull.sk.tsukuba.ac.jp/Tools/XEmacs/xemacs−ucs−conv−0.1.tar.gz

The .diff is a diff against the C sources. The tar ball is elisp code, which provides lots of code tables to map to and from Unicode. As the name of the diff file suggests it is against XEmacs−21; I needed to help `patch' a bit. The most notable difference to my XEmacs−20.4 sources is that file−coding.[ch] was called mule−coding.[ch].

For those unfamilar with the XEmacs−MULE stuff (as I am) a quick guide:

What we call an encoding is called by MULE a `coding−system'. The most important commands are:

```
M−x set−file−coding−system
M−x set−buffer−process−coding−system   [comint buffers]
```

and the variable `file−coding−system−alist', which guides `find−file' to guess the encoding used. After stuff was running, the very first thing I did was this.

This code looks into the special mode line introduced by −*− somewhere in the first 600 bytes of the file about to opened; if now there is a field "Encoding: xyz;" and the xyz encoding ("coding system" in Emacs speak) exists, choose that. So now you could do e.g.

```
;;; −*− Mode: Lisp; Syntax: Common−Lisp; Package: CLEX; Encoding: utf−8; −*−
```

and XEmacs goes into utf−8 mode here.

Atfer everything was running I defined \u03BB (greek lambda) as a macro like:

```
(defmacro \u03BB (x) `(lambda .,x))
```

nedit

xedit

In theory, xedit should be able to edit UTF−8 files if you set the locale accordingly (see above), and add the line "Xedit*international: true" to your $HOME/.Xdefaults file. In practice, it will recognize UTF−8 encoding of non−ASCII characters, but will display them as sequences of "@" characters.

axe

In theory, axe should be able to edit UTF−8 files if you set the locale accordingly (see above), and add the line "Axe*international: true" to your $HOME/.Xdefaults file. In practice, it will simply dump core.

pico

TeX

The teTeX 0.9 (and newer) distribution contains an Unicode adaptation of TeX, called Omega (http://www.gutenberg.eu.org/omega/, ftp://ftp.ens.fr/pub/tex/yannis/omega), but can someone point me to a tutorial for using this system?

Other maybe related links: http://www.dante.de/projekte/nts/NTS−FAQ.html, ftp://ftp.dante.de/pub/tex/language/chinese/CJK/.

4.4 Mailers

MIME: RFC 2279 defines UTF−8 as a MIME charset, which can be transported under the 8bit, quoted−printable and base64 encodings. The older MIME UTF−7 proposal (RFC 2152) is considered to be deprecated and should not be used any further.

Mail clients released after January 1, 1999, should be capable of sending and displaying UTF−8 encoded mails, otherwise they are considered deficient. But these mails have to carry the MIME labels

```
Content−Type: text/plain; charset=UTF−8
Content−Transfer−Encoding: 8bit
```

Simply piping an UTF−8 file into "mail" without caring about the MIME labels will not work.

Mail client implementors should take a look at http://www.imc.org/imc−intl/ and http://www.imc.org/mail−i18n.html.

Now about the individual mail clients (or "mail user agents"):

pine

The situation for an unpatched pine version 4.10 is as follows.

Pine does not do character set conversions. But it allows you to view UTF−8 mails in an UTF−8 text window (Linux console or xterm).

Normally, Pine will warn about different character sets each time you view an UTF−8 encoded mail. To get rid of this warning, choose S (setup), then C (config), then change the value of "character−set" to UTF−8. This option will not do anything, except to reduce the warnings, as Pine has no built−in knowledge of UTF−8.

Also note that Pine's notion of Unicode characters is pretty limited: It will display Latin and Greek characters, but not other kinds of Unicode characters.

A patch by Robert Brady http://www.ents.susu.soton.ac.uk/~robert/pine−utf8−0.1.diff adds UTF−8 support to Pine. With this patch, it decodes and prints headers and bodies properly. The patch depends on the GNOME libunicode http://cvs.gnome.org/lxr/source/libunicode/.

However, alignment remains broken in many places; replying to a mail does not cause the character set to be converted as appropriate; and the editor, pico, cannot deal with multibyte characters.

kmail

kmail (as of KDE 1.0) does not support UTF−8 mails at all.

Netscape Communicator

Netscape Communicator's Messenger can send and display mails in UTF−8 encoding, but it needs a little bit of manual user intervention.

To send an UTF−8 encoded mail: After opening the "Compose" window, but before starting to compose the message, select from the menu "View −> Character Set −> Unicode (UTF−8)". Then compose the message and send it.

When you receive an UTF−8 encoded mail, Netscape unfortunately does not display it in UTF−8 right away, and does not even give a visual clue that the mail was encoded in UTF−8. You have to manually select from the menu "View −> Character Set −> Unicode (UTF−8)".

For displaying UTF−8 mails, Netscape uses different fonts. You can adjust your font settings in the "Edit −> Preferences −> Fonts" dialog; choose the "Unicode" font category.

emacs (rmail, vm)

4.5 Other text−mode applications

less

Get ftp://ftp.gnu.org/pub/gnu/less/less−340.tar.gz and apply the patch less−340−utf−2.diff by Robert Brady <rwb197@ecs.soton.ac.uk>. Then set the environment variable LESSCHARSET:

```
      $ export LESSCHARSET=utf−8
```

If you also have a LESSKEY environment variable set, also make sure that the file it points to does not define LESSCHARSET. If necessary, regenerate this file using the `lesskey' command, or unset the LESSKEY environment variable.

expand, wc

Get the GNU textutils−2.0 and apply the patch textutils−2.0.diff, then configure, add "#define HAVE_MBRTOWC 1", "#define HAVE_FGETWC 1", "#define HAVE_FPUTWC 1" to config.h. In src/Makefile, modify CFLAGS and LDFLAGS so that they include the directories where libutf8 is installed. Then rebuild.

col, colcrt, colrm, column, rev, ul

Get the util−linux−2.9y package, configure it, then define ENABLE_WIDECHAR in defines.h, change the "#if 0" to "#if 1" in lib/widechar.h. In text−utils/Makefile, modify CFLAGS and LDFLAGS so that they include the directories where libutf8 is installed. Then rebuild.

figlet

figlet 2.2 has an option for UTF−8 input: "figlet −C utf8"

kermit

The serial communications program C−Kermit http://www.columbia.edu/kermit/, in versions 7.0beta10 or newer, understands the file and transfer encodings UTF−8 and UCS−2, and understands the terminal encoding UTF−8. Documentation of these features can be found in ftp://kermit.columbia.edu/kermit/test/text/ckermit2.txt.

4.6 Other X11 applications

X11 Xlib kann leider noch kein UTF−8 locale, das müsste auch noch dringend mal gemacht werden.

NextPreviousContents Next PreviousContents

---

5. Making your programs Unicode aware

5.1 C/C++

The C `` `char' `` type is 8−bit and will stay 8−bit because it denotes the smallest addressable data unit. Various facilities are available:

For normal text handling

The ISO/ANSI C standard contains, in an amendment which was added in 1995, a "wide character" type `` `wchar_t' ``, a set of functions like those found in `<string.h>` and `<ctype.h>` (declared in `<wchar.h>` and `<wctype.h>`, respectively), and a set of conversion functions between `` `char *' `` and `` `wchar_t *' `` (declared in `<stdlib.h>`).

Good references for this API are

- the GNU libc−2.1 manual, chapters 4 "Character Handling" and 6 "Character Set Handling",
- the manual pages man−mbswcs.tar.gz,
- the Dinkumware C library reference http://www.dinkumware.com/htm_cl/,
- the OpenGroup's Single Unix specification http://www.UNIX−systems.org/online.html.

Advantages of using this API:

- It's a vendor independent standard.
- The functions do the right thing, depending on the user's locale. All a program needs to call is `setlocale(LC_ALL,"");`.

Drawbacks of this API:

- Some of the functions are not multithread−safe, because they keep a hidden internal state between function calls.
- There is no first−class locale datatype. Therefore this API cannot reasonably be used for anything that needs more than one locale or character set at the same time.
- The OS support for this API is not good on most OSes.

Portability notes

A `wchar_t' may or may not be encoded in Unicode; this is platform and sometimes also locale dependent. A multibyte sequence `char *' may or may not be encoded in UTF−8; this is platform and sometimes also locale dependent.

In detail, here is what the Single Unix specification says about the `wchar_t' type: *All wide−character codes in a given process consist of an equal number of bits. This is in contrast to characters, which can consist of a variable number of bytes. The byte or byte sequence that represents a character can also be represented as a wide−character code. Wide−character codes thus provide a uniform size for manipulating text data. A wide−character code having all bits zero is the null wide−character code, and terminates wide−character strings. The wide−character value for each member of the Portable Character Set* (i.e. ASCII) *will equal its value when used as the lone character in an integer character constant. Wide−character codes for other characters are locale− and implementation−dependent. State shift bytes do not have a wide−character code representation.*

One particular consequence is that in portable programs you shouldn't use non−ASCII characters in string literals. That means, even though you know the Unicode double quotation marks have the codes U+201C and U+201D, you shouldn't write a string literal `L"\u201cHello\u201d, he said"` or `"\xe2\x80\x9cHello\xe2\x80\x9d, he said"` in C programs. Instead, use GNU gettext, write it as `gettext("'Hello', he said")`, and create a message database en.UTF−8.po which translates "'Hello', he said" to "\u201cHello\u201d, he said".

Here is a survey of the portability of the ISO/ANSI C facilities on various Unix flavours. GNU glibc−2.2 will support all of it, but for now we have the following picture.

GNU glibc−2.0.x, glibc−2.1.x

<wchar.h> and <wctype.h> exist.
Has wcs/mbs functions, but no fgetwc/fputwc/wprintf.
No UTF−8 locale.
mbrtowc returns EILSEQ for bytes >= 0x80.

Solaris 2.7

<wchar.h> and <wctype.h> exist.
Has wcs/mbs functions, fgetwc/fputwc/wprintf, everything.
Has the following UTF−8 locales: en_US.UTF−8, de.UTF−8, es.UTF−8, fr.UTF−8, it.UTF−8, sv.UTF−8.
mbrtowc returns EILSEQ for bytes >= 0x80.

OSF/1 4.0d

<wchar.h> and <wctype.h> exist.
Has wcs/mbs functions, fgetwc/fputwc/wprintf, everything.
Has an add−on universal.utf8@ucs4 locale, see "man 5 unicode".
mbrtowc does not know about UTF−8.

Irix 6.5

<wchar.h> and <wctype.h> exist.
Has wcs/mbs functions and fgetwc/fputwc, but not wprintf.
Has no multibyte locales.
Has only a dummy definition for mbstate_t.
Doesn't have mbrtowc.

HP−UX 11.00

<wchar.h> exists, <wctype.h> does not.
Has wcs/mbs functions and fgetwc/fputwc, but not wprintf.
Has a C.utf8 locale.
Doesn't have mbstate_t.
Doesn't have mbrtowc.

AIX 4.2

<wchar.h> exists, <wctype.h> does not − instead use <ctype.h> and <wchar.h>.
Has wcs/mbs functions and fgetwc/fputwc, but not wprintf.
Has the following UTF−8 locales: ET_EE.UTF−8, LT_LT.UTF−8, LV_LV.UTF−8, ZH_CN.UTF−8.
Doesn't have mbstate_t.
Doesn't have mbrtowc.

As a consequence, I recommend to use the restartable and multithread−safe wcsr/mbsr functions, forget about those systems which don't have them (Irix, HP−UX, AIX), and use the UTF−8 locale plug−in libutf8_plug.so (see below) on those systems which permit you to compile programs which use these wcsr/mbsr functions (Linux, Solaris, OSF/1).

A similar advice, given by Sun in http://www.sun.com/software/white−papers/wp−unicode/, section

"Internationalized Applications with Unicode", is:

To properly internationalize an application, use the following guidelines:

1. *Avoid direct access with Unicode. This is a task of the platform's internationalization framework.*
2. *Use the POSIX model for multibyte and wide−character interfaces.*
3. *Only call the APIs that the internationalization framework provides for language and cultural−specific operations.*
4. *Remain code−set independent.*

If, for some reason, in some piece of code, you really have to assume that `wchar_t' is Unicode (for example, if you want to do special treatment of some Unicode characters), you should make that piece of code conditional upon the result of `is_locale_utf8()`. Otherwise you will mess up your program's behaviour in different locales or other platforms. The function `is_locale_utf8` is declared in utf8locale.h and defined in utf8locale.c.

The libutf8 library

A portable implementation of the ISO/ANSI C API, which supports 8−bit locales and UTF−8 locales, can be found in libutf8−0.5.2.tar.gz.

Advantages:

- Unicode UTF−8 support now, portably, even on OSes whose multibyte character support does not work or which don't have multibyte/wide character support at all.
- The same binary works in all OS supported 8−bit locales and in UTF−8 locales.
- When an OS vendor adds proper multibyte character support, you can take advantage of it by simply recompiling without −DHAVE_LIBUTF8 compiler option.

The Plan9 way

The Plan9 operating system, a variant of Unix, uses UTF−8 as character encoding in all applications. Its wide character type is called `` `Rune``', not `` `wchar_t``'. Parts of its libraries, written by Rob Pike and Howard Trickey, are available at ftp://ftp.cdrom.com/pub/netlib/research/9libs/9libs−1.0.tar.gz. Another similar library, written by Alistair G. Crooks, is ftp://ftp.cdrom.com/pub/NetBSD/packages/distfiles/libutf−2.10.tar.gz. In particular, each of these libraries contains an UTF−8 aware regular expression matcher.

Drawback of this API:

- UTF−8 is compiled in, not optional. Programs compiled in this universe lose support for the 8−bit encodings which are still frequently used in Europe.

For graphical user interface

The Qt−2.0 library http://www.troll.no/ contains a fully−Unicode QString class. You can use the member functions QString::utf8 and QString::fromUtf8 to convert to/from UTF−8 encoded text. The QString::ascii and QString::latin1 member functions should not be used any more.

For advanced text handling

The previously mentioned libraries implement Unicode aware versions of the ASCII concepts. Here are libraries which deal with Unicode concepts, such as titlecase (a third letter case, different from uppercase and lowercase), distinction between punctuation and symbols, canonical decomposition, combining classes, canonical ordering and the like.

ucdata−1.9

The ucdata library by Mark Leisher http://crl.nmsu.edu/~mleisher/ucdata.html deals with character properties, case conversion, decomposition, combining classes.

ICU

IBMs Classes for Unicode http://www.alphaworks.ibm.com/tech/icu/. A very comprehensive internationalization library featuring Unicode strings, resource bundles, number formatters, date/time formatters, message formatters, collation and more. Lots of supported locales. Portable to Unix and Win32, but compiles out of the box only on Linux libc6, not libc5.

libunicode

The GNOME libunicode library http://cvs.gnome.org/lxr/source/libunicode/ by Tom Tromey and others. It covers character set conversion, character properties, decomposition.

For conversion

Two conversion libraries, which support UTF−8 and a large number of 8−bit character sets, are available:

The iconv implementation by Ulrich Drepper, contained in the GNU glibc−2.1.1. ftp://ftp.gnu.org/pub/gnu/glibc/glibc−2.1.1.tar.gz

Advantages:

- iconv is POSIX standardized, programs using iconv to convert from/to UTF−8 will also run under Solaris. However, the names for the character sets differ between platforms. For example, "EUC−JP" under glibc is "eucJP" under HP−UX. (The official IANA name for this character set is "EUC−JP", so it's clearly a HP−UX deficiency.)

- No additional library needed.

librecode by François Pinard ftp://ftp.gnu.org/pub/gnu/recode/recode−3.5.tar.gz.

Advantages:

- Support for transliteration, i.e. conversion of non−ASCII characters to sequences of ASCII characters in order to preserve readability by humans, even when a lossless transformation is impossible.

Drawbacks:

- Non−standard API.

Other approaches

libutf−8

libutf−8 by G. Adam Stanislav <adam@whizkidtech.net> contains a few functions for on−the−fly conversion from/to UTF−8 encoded `FILE*' streams.
http://www.whizkidtech.net/i18n/libutf−8−1.0.tar.gz
Advantages:
Very small.
Drawbacks:
Non−standard API.
UTF−8 is compiled in, not optional. Programs compiled with this library lose support for the 8−bit encodings which are still frequently used in Europe.
Installation is nontrivial: Makefile needs tweaking, not autoconfiguring.

5.2 Java

Java has Unicode support built into the language. The type `char' denotes a Unicode character, and the `java.lang.String' class denotes a string built up from Unicode characters.

Java can display any Unicode characters through its windowing system AWT, provided that 1. you set the Java system property "user.language" appropriately, 2. the /usr/lib/java/lib/font.properties.*language* font set definitions are appropriate, and 3. the fonts specified in that file are installed. For example, in order to display text containing japanese characters, you would install japanese fonts and run "java −Duser.language=ja ...". You can combine font sets: In order to display western european, greek and japanese characters simultaneously, you would create a combination of the files "font.properties" (covers ISO−8859−1), "font.properties.el" (covers ISO−8859−7) and "font.properties.ja" into a single file. ??This is untested??

The interfaces java.io.DataInput and java.io.DataOutput have methods called `readUTF' and `writeUTF' respectively. But note that they don't use UTF−8; they use a modified UTF−8 encoding: the NUL character is encoded as the two−byte sequence 0xC0 0x80 instead of 0x00, and a 0x00 byte is added at the end. Encoded this way, strings can contain NUL characters and nevertheless need not be prefixed with a length field − the C <string.h> functions like strlen() and strcpy() can be used to manipulate them.

5.3 Lisp

The Common Lisp standard specifies two character types: \`base−char' and \`character'. It's up to the implementation to support Unicode or not. The language also specifies a keyword argument \`:external−format' to \`open', as the natural place to specify a character set or encoding.

Among the free Common Lisp implementations, only CLISP http://clisp.cons.org/ supports Unicode. You need a CLISP version from July 1999 or newer. ftp://clisp.cons.org/pub/lisp/clisp/source/clispsrc.tar.gz. The types \`base−char' and \`character' are both equivalent to 16−bit Unicode. The encoding used for file or socket/pipe I/O can be specified through the \`:external−format' argument. The encodings used for tty I/O and the default encoding for file/socket/pipe I/O are locale dependent.

Among the commercial Common Lisp implementations, only Eclipse http://www.elwood.com/eclipse/eclipse.htm supports Unicode. See http://www.elwood.com/eclipse/char.htm. The type \`base−char' is equivalent to ISO−8859−1, and the type \`character' contains all Unicode characters. The encoding used for file I/O can be specified through a combination of the \`:element−type' and \`:external−format' arguments to \`open'. Limitations: Character attribute functions are locale dependent. Source and compiled source files cannot contain Unicode string literals.

The commercial Common Lisp implementation Allegro CL does not support Unicode yet, but Erik Naggum is working on it.

5.4 Ada95

Ada95 was designed for Unicode support and the Ada95 standard library features special ISO 10646−1 data types Wide_Character and Wide_String, as well as numerous associated procedures and functions. The GNU Ada95 compiler (gnat−3.11 or newer) supports UTF−8 as the external encoding of wide characters. This allows you to use UTF−8 in both source code and application I/O. To activate it in the application, use "WCEM=8" in the FORM string when opening a file, and use compiler option "−gnatW8" if the source code is in UTF−8. See the GNAT and Ada95 reference manuals for details.

Next PreviousContents