UTF-8 and Unicode FAQ for Unix/Linux

by Markus Kuhn

This text is a very comprehensive one-stop information resource on how you can use Unicode/UTF-8 on POSIX systems (Linux, Unix). You will find here both introductory information for every user, as well as detailed references for the experienced developer.

Unicode has started to replace ASCII, ISO 8859 and EUC at all levels. It enables users to handle not only practically any script and language used on this planet, it also supports a comprehensive set of mathematical and technical symbols to simplify scientific information exchange.

With the UTF-8 encoding, Unicode can be used in a convenient and backwards compatible way in environments that were designed entirely around ASCII, like Unix. UTF-8 is the way in which Unicode is used under Unix, Linux, and similar systems. It is now time to make sure that you are well familiar with it and that your software supportsUTF-8 smoothly.

Contents

- What are UCS and ISO 10646?
- What are combining characters?
- What are UCS implementation levels?
- Has UCS been adopted as a national standard?
- What is Unicode?
- So what is the difference between Unicode and ISO 10646?
- What is UTF-8?
- Who invented UTF-8?
- Where do I find nice UTF-8 example files?
- What different encodings are there?
- What programming languages support Unicode?
- How should Unicode be used under Linux?
- How do I have to modify my software?
- C support for Unicode and UTF-8
- How should the UTF-8 mode be activated?
- How do I get a UTF-8 version of xterm?
- How much of Unicode does xterm support?
- Where do I find ISO 10646-1 X11 fonts?
- What are the issues related to UTF-8 terminal emulators?
- What UTF-8 enabled applications are available?**[UPDATED]**
- What patches to improve UTF-8 support are available?
- Are there free libraries for dealing with Unicode available?
- What is the status of Unicode support for various X widget libraries?

- What packages with UTF-8 support are currently under development?
- How does UTF-8 support work under Solaris?
- Can I use UTF-8 on the Web?
- How are PostScript glyph names related to UCS codes?
- Are there any well-defined UCS subsets?
- What issues are there to consider when converting encodings
- Is X11 ready for Unicode?
- What are useful Perl one-liners for working with UTF-8?
- How can I enter Unicode characters?
- Are there any good mailing lists on these issues?
- Further references

What are UCS and ISO 10646?

The international standard **ISO 10646** defines the **Universal Character Set (UCS)**. UCS is a superset of all other character set standards. It guarantees *round-trip compatibility* to other character sets. This means simply that no information is lost if you convert any text string to UCS and then back to its original encoding.

UCS contains the characters required to represent practically all known languages. This includes not only the Latin, Greek, Cyrillic, Hebrew, Arabic, Armenian, and Georgian scripts, but also Chinese, Japanese and Korean Han ideographs as well as scripts such as Hiragana, Katakana, Hangul, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, Lao, Khmer, Bopomofo, Tibetan, Runic, Ethiopic, Canadian Syllabics, Cherokee, Mongolian, Ogham, Myanmar, Sinhala, Thaana, Yi, and others. For scripts not yet covered, research on how to best encode them for computer usage is still going on and they will be added eventually. This includes not only historic scripts such as Cuneiform, Hieroglyphs and various Indo-European notations, but even some selected artistic scripts such as Tolkien's Tengwar and Cirth. UCS also covers a large number of graphical, typographical, mathematical and scientific symbols, including those provided by TeX, PostScript, APL, the International Phonetic Alphabet(IPA), MS-DOS, MS-Windows, Macintosh, OCR fonts, as well as many word processing and publishing systems. The standard continues to be maintained and updated. Ever more exotic and specialized symbols and characters will be added for many years to come.

ISO 10646 originally defined a 31-bit character set. The subsets of2^{16} characters where the elements differ (in a 32-bitinteger

representation) only in the 16 least-significant bits are called the *planes* of UCS.

The most commonly used characters, including all those found in major older encoding standards, have been placed into the first plane(0x0000 to 0xFFFD), which is called the **Basic Multilingual Plane (BMP)** or Plane 0. The characters that were later added outside the 16-bit BMP are mostly for specialist applications such as historic scripts and scientific notation. Current plans are that there will never be characters assigned outside the21-bit code space from 0x000000 to 0x10FFFF, which covers a bit over one million potential future characters. The ISO 10646-1 standard was first published in 1993 and defines the architecture of the character set and the content of the BMP. A second part ISO 10646-2 was added in2001 and defines characters encoded outside the BMP. In the 2003edition, the two parts were combined into a single ISO 10646 standard. New characters are still being added on a continuous basis, but the existing characters will not be changed any more and are stable.

UCS assigns to each character not only a code number but also an official name. A hexadecimal number that represents a UCS or Unicode value is commonly preceded by "U+" as in U+0041 for the character "Latin capital letter A". The UCS characters U+0000 to U+007F are identical to those in US-ASCII (ISO 646 IRV) and the range U+0000 toU+00FF is identical to ISO 8859-1 (Latin-1). The range U+E000 toU+F8FF and also larger ranges outside the BMP are reserved for private use. UCS also defines several methods for encoding a string of characters as a sequence of bytes, such as UTF-8 and UTF-16.

The full reference for the UCS standard is

International Standard ISO/IEC 10646, Information technology—Universal Multiple-Octet Coded Character Set (UCS) . Third edition, International Organization for Standardization, Geneva, 2003.

The standard can be ordered online from ISO as a set of PDF files on CD-ROM for 112 CHF.

In September 2006, ISO released a free online PDF copy of ISO 10646:2003 on its Freely Available Standards web page. The ZIP file is 82 MB long.

What are combining characters?

Some code points in UCS have been assigned to **combining characters**. These are similar to the non-spacing accent keys on a typewriter. A combining character is not a full character by itself. It is an accent or other diacritical mark that is added to the previous character. This way, it is possible to place any accent on any character. The most important accented characters, like those used in the orthographies of common languages, have codes of their own in UCS to ensure backwards compatibility with older character sets. They are known as **precomposed characters**. Precomposed characters are available in UCS for backwards compatibility with older encodings that have no combining characters, such as ISO 8859. The combining-character mechanism allows one to add accents and other diacritical marks to any character. This is especially important for scientific notations such as mathematical formulae and the International Phonetic Alphabet, where any possible combination of a base character and one or several diacritical marks could be needed.

Combining characters follow the character which they modify. For example, the German umlaut character Ä (“Latin capital letter A with diaeresis”) can either be represented by the precomposed UCS codeU+00C4, or alternatively by the combination of a normal “Latin capital letter A” followed by a “combining diaeresis”: U+0041 U+0308. Several combining characters can be applied when it is necessary to stack multiple accents or add combining marks both above and below the base character. The Thai script, for example, needs up to two combining characters on a single base character.

What are UCS implementation levels?

Not all systems can be expected to support all the advanced mechanisms of UCS, such as combining characters. Therefore, ISO 10646specifies the following three implementation levels:

Level 1

> Combining characters and Hangul Jamo characters are not supported.
>
> [Hangul Jamo are an alternative representation of precomposed modern Hangul syllables as a sequence of consonants and vowels. They are required to fully support the Korean script including Middle Korean.]

Level 2

> Like level 1, however in some scripts, a fixed list of combining characters is now allowed (e.g., for Hebrew, Arabic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugo, Kannada, Malayalam, Thai and Lao). These scripts cannot be represented adequately in UCS without support for at least certain combining characters.

Level 3

All UCS characters are supported, such that, for example, mathematicians can place a tilde or an arrow (or both) on any character.

Has UCS been adopted as a national standard?

Yes, a number of countries have published national adoptions of ISO10646, sometimes after adding additional annexes with cross-references to older national standards, implementation guidelines, and specifications of various national implementation subsets:

- China: GB 13000.1-93
- Japan: JISX 0221-1:2001
- Korea: KS X 1005-1:1995 (includes ISO 10646-1:1993 amendments 1-7)
- Vietnam: TCVN6909:2001
  (This "16-bit Coded Vietnamese Character Set" is a small UCS subset and to be implemented for data interchange with and within government agencies as of 2002-07-01.)
- Iran: ISIRI6219:2002, Information Technology — Persian Information Interchange and Display Mechanism, using Unicode. (This is not aversion or subset of ISO 10646, but a separate document that provides additional national guidance and clarification on handling the Persian language and the Arabic script in Unicode.)

What is Unicode?

In the late 1980s, there have been two independent attempts to create a single unified character set. One was the ISO 10646 project of the International Organization for Standardization (ISO), the other was the Unicode Project organized by a consortium of (initially mostly US) manufacturers of multi-lingual software. Fortunately, the participants of both projects realized in around 1991 that two different unified character sets is not exactly what the world needs. They joined their efforts and worked together on creating a single code table. Both projects still exist and publish their respective standards independently, however the Unicode Consortium and ISO/IEC JTC1/SC2 have agreed to keep the code tables of the Unicode and ISO 10646 standards compatible and they closely coordinate any further extensions. Unicode 1.1 corresponded to ISO10646-1:1993, Unicode 3.0 corresponded to ISO 10646-1:2000, Unicode3.2 added ISO 10646-2:2001, and Unicode 4.0 corresponds to ISO10646:2003, and Unicode 5.0 corresponds to ISO 10646:2003 plus its amendments 1–3. All Unicode versions since 2.0 are compatible, only new characters will be added, no existing characters will be removed or renamed in the future.

The Unicode Standard can be ordered like any normal book, for instance via amazon.com for around 60 USD:

The Unicode Consortium: The Unicode Standard 5.0,
Addison-Wesley, 2006,
ISBN 0-321-48091-0.

If you work frequently with text processing and character sets, you definitely should get a copy. Unicode 5.0 is also available online.

So what is the difference between Unicode and ISO 10646?

The Unicode Standard published by the Unicode Consortium corresponds to ISO10646 at implementation level 3. All characters are at the same positions and have the same names in both standards.

The Unicode Standard defines in addition much more semantics associated with some of the characters and is in general a better reference for implementers of high-quality typographic publishing systems. Unicode specifies algorithms for rendering presentation forms of some scripts (say Arabic), handling of bi-directional texts that mix for instance Latin and Hebrew, algorithms for sorting and string comparison, and much more.

The ISO 10646 standard on the other hand is not much more than a simple character set table, comparable to the old ISO 8859 standards. It specifies some terminology related to the standard, defines some encoding alternatives, and it contains specifications of how to use UCS in connection with other established ISO standards such as ISO6429 and ISO 2022. There are other closely related ISO standards, for instance ISO14651 on sorting UCS strings. A nice feature of the ISO 10646-1standard is that it provides CJK example glyphs in five different style variants, while the Unicode standard shows the CJK ideographs only in a Chinese variant.

What is UTF-8?

UCS and Unicode are first of all just code tables that assign integer numbers to characters. There exist several alternatives for how a sequence of such characters or their respective integer

values can be represented as a sequence of bytes. The two most obvious encodings store Unicode text as sequences of either 2 or 4 bytes sequences. The official terms for these encodings are UCS-2 and UCS-4, respectively. Unless otherwise specified, the most significant byte comes first in these (Bigendian convention). An ASCII or Latin-1 file can be transformed into a UCS-2 file by simply inserting a 0x00 byte in front of every ASCII byte. If we want to have a UCS-4 file, we have to insert three 0x00 bytes instead before every ASCII byte.

Using UCS-2 (or UCS-4) under Unix would lead to very severe problems. Strings with these encodings can contain as parts of many wide characters bytes like "\0" or "/" which have a special meaning in filenames and other C library function parameters. In addition, the majority of UNIX tools expects ASCII files and cannot read 16-bit words as characters without major modifications. For these reasons,**UCS-2** is not a suitable external encoding of **Unicode** in filenames, text files, environment variables, etc.

The **UTF-8** encoding defined in ISO 10646-1:2000 Annex D and also described in RFC 3629 as well as section 3.9 of the Unicode 4.0 standard does not have these problems. It is clearly the way to go for using **Unicode** under Unix-style operating systems.

UTF-8 has the following properties:

- UCS characters U+0000 to U+007F (ASCII) are encoded simply as bytes 0x00 to 0x7F (ASCII compatibility). This means that files and strings which contain only 7-bit ASCII characters have the same encoding under both ASCII and UTF-8.
- All UCS characters >U+007F are encoded as a sequence of several bytes, each of which has the most significant bit

set. Therefore, no ASCII byte (0x00-0x7F) can appear as part of any other character.

- The first byte of a multibyte sequence that represents a non-ASCII character is always in the range 0xC0 to 0xFD and it indicates how many bytes follow for this character. All further bytes in a multibyte sequence are in the range 0x80 to 0xBF. This allows easy resynchronization and makes the encoding stateless and robust against missing bytes.
- All possible 2^{31} UCS codes can be encoded.
- UTF-8 encoded characters may theoretically be up to six bytes long, however 16-bit BMP characters are only up to three bytes long.
- The sorting order of Bigendian UCS-4 byte strings is preserved.
- The bytes 0xFE and 0xFF are never used in the UTF-8 encoding.

The following byte sequences are used to represent a character. The sequence to be used depends on the Unicode number of the character:

U-00000000 – U-0000007F:	0*xxxxxxx*
U-00000080 – U-000007FF:	110*xxxxx* 10*xxxxxx*
U-00000800 – U-0000FFFF:	1110*xxxx* 10*xxxxxx* 10*xxxxxx*
U-00010000 – U-001FFFFF:	11110*xxx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx*
U-00200000 – U-	111110*xx* 10*xxxxxx* 10*xxxxxx*

03FFFFFF:	10*xxxxxx*10*xxxxxx*
U-04000000 – U-7FFFFFFF:	1111110*x* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx*10*xxxxxx* 10*xxxxxx*

The *xxx* bit positions are filled with the bits of the character code number in binary representation. The rightmost *x*bit is the least-significant bit. Only the shortest possible multibyte sequence which can represent the code number of the character can be used. Note that in multibyte sequences, the number of leading 1 bits in the first byte is identical to the number of bytes in the entire sequence.

Examples: The Unicode character `U+00A9 = 10101001` (copyright sign) is encoded in UTF-8 as

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

and character `U+2260 = 0010 0010 0110 0000` (not equal to) is encoded as:

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

The official name and spelling of this encoding is UTF-8, where UTF stands for **U**CS **T**ransformation **F**ormat. Please do not write UTF-8 in any documentation text in other ways (such as utf8or UTF_8), unless of course you refer to a variable name and not the encoding itself.

An important note for developers of UTF-8 decoding routines: For security reasons, a UTF-8 decoder must not accept UTF-8 sequences that are longer than necessary to encode a character. For example, the character U+000A (line feed) must be accepted from a UTF-8 stream **only** in the form 0x0A, but not in any of the following five possible overlong forms:

```
0xC0 0x8A
0xE0 0x80 0x8A
0xF0 0x80 0x80 0x8A
0xF8 0x80 0x80 0x80 0x8A
0xFC 0x80 0x80 0x80 0x80 0x8A
```

Any overlong UTF-8 sequence could be abused to bypass UTF-8substring tests that look only for the shortest possible encoding. All overlong UTF-8 sequences start with one of the following byte patterns:

1100000*x* (10*xxxxxx*)
11100000 100*xxxxx* (10*xxxxxx*)
11110000 1000*xxxx* (10*xxxxxx* 10*xxxxxx*)
11111000 10000*xxx* (10*xxxxxx* 10*xxxxxx*10*xxxxxx*)
11111100 100000*xx* (10*xxxxxx* 10*xxxxxx*10*xxxxxx* 10*xxxxxx*)

Also note that the code positions U+D800 to U+DFFF (UTF-16surrogates) as well as U+FFFE and U+FFFF must not occur in normalUTF-8 or UCS-4 data. UTF-8 decoders should treat them like malformed or overlong sequences for safety reasons.

Markus Kuhn's UTF-8 decoder stress test file contains a systematic collection of malformed and overlong UTF-8 sequences and will help you to verify the robustness of your decoder.

Who invented UTF-8?

The encoding known today as UTF-8 was invented by Ken Thompson. It was born during the evening hours of 1992-09-02 in a New Jersey diner, where he designed it in the presence of Rob Pike on a placemat(see Rob Pike's UTF-8 history). It replaced an earlier attempt to design a FSS/UTF (file system safe UCS transformation format) that was circulated in an X/Open working

document in August 1992 by Gary Miller (IBM), Greger Leijonhufvud and John Entenmann (SMI) as a replacement for the division-heavy UTF-1encoding from the first edition of ISO 10646-1. By the end of the first week of September 1992, Pike and Thompson had turned AT&T Bell Lab's Plan 9into the world's first operating system to use UTF-8. They reported about their experience at the USENIX Winter 1993 Technical Conference, San Diego, January 25-29, 1993,Proceedings, pp. 43-50. FSS/UTF was briefly also referred to as UTF-2and later renamed into UTF-8, and pushed through the standards process by the X/Open Joint Internationalization Group XOJIG.

Where do I find nice UTF-8 example files?

A few interesting UTF-8 example files for tests and demonstrations are:

- UTF-8Sampler web page by the Kermit project
- Markus Kuhn's example plain-text files, including among others the classic demo, decoder test, TeX repertoire, WGL4 repertoire, euro test pages, and Robert Brady's IPA lyrics.
- Unicode Transcriptions
- Generator for Indic Unicode test files

What different encodings are there?

Both the UCS and Unicode standards are first of all large tables that assign to every character an integer number. If you use the term "UCS", "ISO 10646", or "Unicode", this just refers to a mapping between characters and integers. This does not yet

specify how to store these integers as a sequence of bytes in memory.

ISO 10646-1 defines the UCS-2 and UCS-4 encodings. These are sequences of 2 bytes and 4 bytes per character, respectively. ISO10646 was from the beginning designed as a 31-bit character set (with possible code positions ranging from U-00000000 to U-7FFFFFFF),however it took until 2001 for the first characters to be assigned beyond the Basic Multilingual Plane (BMP), that is beyond the first2^{16} character positions (see ISO 10646-2 and Unicode 3.1).UCS-4 can represent all UCS and Unicode characters, UCS-2 can represent only those from the BMP (U+0000 to U+FFFF).

"Unicode" originally implied that the encoding was UCS-2 and it initially didn't make any provisions for characters outside the BMP(U+0000 to U+FFFF). When it became clear that more than 64k characters would be needed for certain special applications (historic alphabets and ideographs, mathematical and musical typesetting, etc.), Unicode was turned into a sort of 21-bit character set with possible code points in the range U-00000000 to U-0010FFFF. The 2×1024 surrogate characters (U+D800 to U+DFFF) were introduced into the BMP to allow1024×1024 non-BMP characters to be represented as a sequence of two 16-bit surrogate characters. This way UTF-16 was born, which represents the extended "21-bit" Unicode in a way backwards compatible withUCS-2. The term UTF-32 was introduced in Unicode to describe a 4-byte encoding of the extended"21-bit" Unicode. UTF-32 is the exact same thing as UCS-4, except that by definition UTF-32 is never used to represent characters aboveU-0010FFFF, while UCS-4 can cover all 2^{31} code positions up to U-7FFFFFFF. The ISO 10646 working group has agreed to modify their standard to exclude code positions beyond U-

0010FFFF, in order to turn the new UCS-4 and UTF-32 into practically the same thing.

In addition to all that, UTF-8 was introduced to provide an ASCII backwards compatible multi-byte encoding. The definitions of UTF-8 in UCS and Unicode differed originally slightly, because in UCS, up to 6-byte long UTF-8 sequences were possible to represent characters up to U-7FFFFFFF, while in Unicode only up to4-byte long UTF-8 sequences are defined to represent characters up toU-0010FFFF. (The difference was in essence the same as between UCS-4and UTF-32.)

No endianess is implied by the encoding names UCS-2, UCS-4, UTF-16,and UTF-32, though ISO 10646-1 says that Bigendian should be preferred unless otherwise agreed. It has become customary to append the letters “BE” (Bigendian, high-byte first) and “LE” (Littleendian, low-byte first) to the encoding names in order to explicitly specify a byte order.

In order to allow the automatic detection of the byte order, it has become customary on some platforms (notably Win32) to start every Unicode file with the character U+FEFF (ZERO WIDTH NO-BREAK SPACE),also known as the Byte-Order Mark (BOM). Its byte-swapped equivalent U+FFFE is not a valid Unicode character, therefore it helps to unambiguously distinguish the Bigendian and Littleendian variants ofUTF-16 and UTF-32.

A full featured character encoding converter will have to provide the following 13 encoding variants of Unicode and UCS:

UCS-2, UCS-2BE, UCS-2LE, UCS-4, UCS-4LE, UCS-4BE, UTF-8, UTF-16,UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, UTF-32LE

Where no byte order is explicitly specified, use the byte order of the CPU on which the conversion takes place and in an input

stream swap the byte order whenever U+FFFE is encountered. The difference between outputting UCS-4 versus UTF-32 and UTF-16 versus UCS-2 lies in handling out-of-range characters. The fallback mechanism for non-representable characters has to be activated in UTF-32 (for characters > U-0010FFFF) or UCS-2 (for characters > U+FFFF) even whereUCS-4 or UTF-16 respectively would offer a representation.

Really just of historic interest are UTF-1, UTF-7, SCSU and a dozen other less widely publicised UCS encoding proposals with various properties, none of which ever enjoyed any significant use. Their uses should be avoided.

A good encoding converter will also offer options for adding or removing the BOM:

- Unconditionally prefix the output text with U+FEFF.
- Prefix the output text with U+FEFF unless it is already there.
- Remove the first character if it is U+FEFF.

It has also been suggested to use the UTF-8 encoded BOM (0xEF 0xBB0xBF) as a signature to mark the beginning of a UTF-8 file. This practice should definitely **not** be used on POSIX systems for several reasons:

- On POSIX systems, the locale and not magic file type codes define the encoding of plain text files. Mixing the two concepts would add a lot of complexity and break existing functionality.
- Adding a UTF-8 signature at the start of a file would interfere with many established conventions such as the kernel looking for "#!"at the beginning of a plaintext executable to locate the appropriate interpreter.

- Handling BOMs properly would add undesirable complexity even to simple programs like `cat` or `grep` that mix contents of several files into one.

In addition to the encoding alternatives, Unicode also specifies various Normalization Forms, which provide reasonable subsets of Unicode, especially to remove encoding ambiguities caused by the presence of precomposed and compatibility characters:

- **Normalization Form D (NFD):** Split up (decompose) precomposed characters into combining sequences where possible ,e.g. use U+0041 U+0308 (LATIN CAPITAL LETTER A, COMBINING DIAERESIS)instead of U+00C4 (LATIN CAPITAL LETTER A WITH DIAERESIS). Also avoid deprecated characters, e.g. use U+0041 U+030A (LATIN CAPITAL LETTER A, COMBINING RING ABOVE) instead of U+212B (ANGSTROM SIGN).
- **Normalization Form C (NFC):** Use precomposed characters instead of combining sequences where possible, e.g. use U+00C4 (“Latin capital letter A with diaeresis”) instead of U+0041 U+0308 (“Latin capital letter A”, “combining diaeresis”). Also avoid deprecated characters, e.g. use U+00C5 (LATIN CAPITAL LETTER A WITH RING ABOVE) instead of U+212B (ANGSTROM SIGN).
  NFC is the preferred form for Linux and WWW.
- **Normalization Form KD (NFKD):** Like NFD, but avoid in addition the use of compatibility characters, e.g. use “fi” instead ofU+FB01 (LATIN SMALL LIGATURE FI).
- **Normalization Form KC (NFKC):** Like NFC, but avoid in addition the use of compatibility characters, e.g. use “fi” instead ofU+FB01 (LATIN SMALL LIGATURE FI).

A full-featured character encoding converter should also offer conversion between normalization forms. Care should be used with mapping to NFKD or NFKC, as semantic information might be lost (for instance U+00B2 (SUPERSCRIPT TWO) maps to 2) and extra mark-up information might have to be added to preserve it (e.g.,`<SUP>2</SUP>` in HTML).

What programming languages support Unicode?

More recent programming languages that were developed after around1993 already have special data types for Unicode/ISO 10646-1characters. This is the case with Ada95, Java, TCL, Perl, Python, C# and others.

ISO C 90 specifies mechanisms to handle multi-byte encoding and wide characters. These facilities were improved with Amendment 1 to ISO C90 in 1994 and even further improvements were made in the ISO C 99 standard. These facilities were designed originally with various East-Asian encodings in mind. They are on one side slightly more sophisticated than what would be necessary to handle UCS (handling of "shift sequences"), but also lack support for more advanced aspects of UCS (combining characters, etc.). UTF-8 is an example of what the ISO C standard calls multi-byte encoding. The type *wchar_t*, which in modern environments is usually a signed 32-bit integer, can be used to hold Unicode characters. (Since *wchar_t* has ended up being a 16-bit type on some platforms and a 32-bit type on others, additional types *char16_t* and *char32_t* have been proposed in ISO TR 19769 for future revisions of the C language, to give application programmers more control over the representation of such wide strings.)

Unfortunately, *wchar_t* was already widely used for various Asian 16-bit encodings throughout the 1990s. Therefore, the ISO C 99 standard was bound by backwards compatibility. It could not be changed to require *wchar_t* to be used with UCS, like Java and Ada95 managed to do. However, the C compiler can at least signal to an application that *wchar_t* is guaranteed to hold UCS values in all locales. To do so, it defines the macro `__STDC_ISO_10646__` to be an integer constant of the form *yyyymm*L. The year and month refer to the version of ISO/IEC 10646 and its amendments that have been implemented. For example, `__STDC_ISO_10646__ == 200009L` if the implementation covers ISO/IEC 10646-1:2000.

How should Unicode be used under Linux?

Before UTF-8 emerged, Linux users all over the world had to use various different language-specific extensions of ASCII. Most popular were ISO 8859-1 and ISO 8859-2 in Europe, ISO 8859-7 in Greece, KOI-8/ ISO 8859-5 / CP1251 in Russia, EUC and Shift-JIS in Japan, BIG5 in Taiwan, etc. This made the exchange of files difficult and application software had to worry about various small differences between these encodings. Support for these encodings was usually incomplete, untested, and unsatisfactory, because the application developers rarely used all these encodings themselves.

Because of these difficulties, major Linux distributors and application developers are now phasing out these older legacy encodings in favour of UTF-8. UTF-8 support has improved dramatically over the last few years and many people now use UTF-8 on a daily basis in

- text files (source code, HTML files, email messages, etc.)

- file names
- standard input and standard output, pipes
- environment variables
- cut and paste selection buffers
- telnet, modem, and serial port connections to terminal emulators

and in any other places where byte sequences used to be interpreted in ASCII.

In UTF-8 mode, terminal emulators such as xterm or the Linux console driver transform every keystroke into the corresponding UTF-8sequence and send it to the stdin of the foreground process. Similarly, any output of a process on stdout is sent to the terminal emulator, where it is processed with a UTF-8 decoder and then displayed using a 16-bit font.

Full Unicode functionality with all bells and whistles (e.g. high-quality typesetting of the Arabic and Indic scripts) can only be expected from sophisticated multi-lingual word-processing packages. What Linux supports today on a broad base is far simpler and mainly aimed at replacing the old 8- and 16-bit character sets. Linux terminal emulators and command line tools usually only support a Level1 implementation of ISO 10646-1 (no combining characters), and only scripts such as Latin, Greek, Cyrillic, Armenian, Georgian, CJK, and many scientific symbols are supported that need no further processing support. At this level, UCS support is very comparable to ISO 8859support and the only significant difference is that we have now thousands of different characters available, that characters can be represented by multibyte sequences, and that ideographic Chinese/Japanese/Korean characters require two terminal character positions (double-width).

Level 2 support in the form of combining characters for selected scripts (in particular Thai)and Hangul Jamo is in parts also available (i.e., some fonts, terminal emulators and editors support it via simple over stringing), but precomposed characters should be preferred over combining character sequences where available. More formally, the preferred way of encoding text in Unicode under Linux should be *Normalization FormC* as defined in Unicode Technical Report #15.

One influential non-POSIX PC operating system vendor (whom we shall leave unnamed here) suggested that all Unicode files should start with the character ZERO WIDTH NOBREAK SPACE (U+FEFF), which is in this role also referred to as the "signature" or "byte-order mark (BOM)", in order to identify the encoding and byte-order used in a file. Linux/Unix does **not** use any BOMs and signatures. They would break far too many existing ASCII syntax conventions (such as scripts starting with `#!`). On POSIX systems, the selected locale identifies already the encoding expected in all input and output files of a process. It has also been suggested to call UTF-8files without a signature "UTF-8N" files, but this non-standard term is usually not used in the POSIX world.

Before you switch to UTF-8 under Linux, update your installation to a recent distribution with up-to-date UTF-8 support. This is particular the case if you use an installation older than SuSE 9.1 or Red Hat 8.0. Before these, UTF-8 support was not yet mature enough to be recommendable for daily use.

RedHat Linux 8.0 (September 2002) was the first distribution to take the leap of switching to UTF-8 as the default encoding for most locales. The only exceptions were Chinese/Japanese/Korean locales, for which there were at the time still too many specialized

tools available that did not yet support UTF-8. This first mass deployment of UTF-8 under Linux caused most remaining issues to be ironed out rather quickly during 2003. SuSE Linux then switched its default locales to UTF-8 as well, as of version 9.1(May 2004). It was followed by Ubuntu Linux, the first Debian-derivative that switched to UTF-8 as the system-wide default encoding. With the migration of the three most popular Linux distributions, UTF-8 related bugs have now been fixed in practically all well-maintained Linux tools. Other distributions can be expected to follow soon.

How do I have to modify my software?

If you are a developer, there are several approaches to add UTF-8support. We can split them into two categories, which I will call soft and hard conversion. In soft conversion, data is kept in its UTF-8form everywhere and only very few software changes are necessary. In hard conversion, any UTF-8 data that the program reads will be converted into wide-character arrays and will be handled as such everywhere inside the application. Strings will only be converted back to UTF-8 at output time. Internally, a character remains a fixed-size memory object.

We can also distinguish hard-wired and locale-dependent approaches for supporting UTF-8, depending on how much the string processing relies on the standard library. C offers a number of string processing functions designed to handle arbitrary locale-specific multibyte encodings. An application programmer who relies entirely on these can remain unaware of the actual details of the UTF-8 encoding. Chances are then that by merely changing the locale setting, several other multi-byte encodings (such as EUC) will automatically be supported as well. The other way a programmer can go is to hardcode knowledge aboutUTF-8 into

the application. This may lead in some situations to significant performance improvements. It may be the best approach for applications that will only be used with ASCII and UTF-8.

Even where support for every multi-byte encoding supported by libc is desired, it may well be worth to add extra code optimized forUTF-8. Thanks to UTF-8's self-synchronizing features, it can be processed very efficiently. The locale-dependent libc string functions can be two orders of magnitude slower than equivalent hardwired UTF-8procedures. A bad teaching example was GNU grep 2.5.1, which relied entirely on locale-dependent libc functions such as `mbrlen()` for its generic multi-byte encoding support. This made it about 100× slower in multibyte mode than in single-byte mode! Other applications with hardwired support for UTF-8regular expressions (e.g., Perl 5.8) do not suffer this dramatic slowdown.

Most applications can do very fine with just soft conversion. This is what makes the introduction of UTF-8 on Unix feasible at all. To name two trivial examples, programs such as `cat` and `echo` do not have to be modified at all. They can remain completely ignorant as to whether their input and output is ISO 8859-2or UTF-8, because they handle just byte streams without processing them. They only recognize ASCII characters and control codes such as`'\n'` which do not change in any way under UTF-8.Therefore the UTF-8 encoding and decoding is done for these applications completely in the terminal emulator.

A small modification will be necessary for any program that determines the number of characters in a string by counting the bytes. With UTF-8, as with other multi-byte encodings, where the length of a text string is of concern, programmers have to distinguish clearly between

1. the number of bytes,
2. the number of characters,
3. the display width (e.g., the number of cursor position cells in aVT100 terminal emulator)

of a string.

C's `strlen(s)` function always counts the *number of bytes*. This is the number relevant, for example, for memory management (determination of string buffer sizes). Where the output of strlen is used for such purposes, no change will be necessary.

The *number of characters* can be counted in C in a portable way using `mbstowcs(NULL,s,0)`. This works for UTF-8 like for any other supported encoding, as long as the appropriate locale has been selected. A hard-wired technique to count the number of characters in a UTF-8 string is to count all bytes except those in the range 0x80 – 0xBF, because these are just continuation bytes and not characters of their own. However, the need to count characters arises surprisingly rarely in applications.

In applications written for ASCII or ISO 8859, a far more common use of `strlen` is to predict the *number of columns* that the cursor of the terminal will advance if a string is printed. With UTF-8, neither a byte nor a character count will predict the display width, because ideographic characters (Chinese, Japanese, Korean) will occupy two column positions, whereas control and combining characters occupy none. To determine the width of a string on the terminal screen, it is necessary to decode the UTF-8sequence and then use the `wcwidth` function to test the display width of each character, or `wcswidth` to measure the entire string.

For instance, the `ls` program had to be modified, because without knowing the column widths of filenames, it cannot format the table layout in which it presents directories to the user. Similarly, all programs that assume somehow that the output is presented in a fixed-width font and format it accordingly have to learn how to count columns in UTF-8 text. Editor functions such as deleting a single character have to be slightly modified to delete all bytes that might belong to one character. Affected were for instance editors (`vi`, `emacs`, `readline`, etc.) as well as programs that use the `ncurses` library.

Any Unix-style kernel can do fine with soft conversion and needs only very minor modifications to fully support UTF-8. Most kernel functions that handle strings (e.g. file names, environment variables, etc.) are not affected at all by the encoding. Modifications were necessary in Linux the following places:

- The console display and keyboard driver (another VT100 emulator)have to encode and decode UTF-8 and should support at least some subset of the Unicode character set. This had already been available in Linux as early as kernel 1.2 (send ESC %G to the console to activate UTF-8 mode).
- External file system drivers such as VFAT and WinNT have to convert file name character encodings. UTF-8 is one of the available conversion options, and the `mount` command has to tell the kernel driver that user processes shall see UTF-8 file names. Since VFAT and WinNT use already Unicode anyway, UTF-8 is the only available encoding that guarantees a lossless conversion here.
- The tty driver of any POSIX system supports a “cooked” mode, in which some primitive line editing functionality is available. In order to allow the character-erase function

(which is activated when you press backspace) to work properly with UTF-8, someone needs to tell it not count continuation bytes in the range 0x80-0xBF as characters, but to delete them as part of a UTF-8 multi-byte sequence. Since the kernel is ignorant of the libc locale mechanics, another mechanism is needed to tell the tty driver about UTF-8 being used. Linux kernel versions 2.6 or newer support a bit IUTF8 in the c_iflag member variable of struct termios. If it is set, the “cooked” mode line editor will treat UTF-8 multi-byte sequences correctly. This mode can be set from the command shell with “stty iutf8”. Xterm and friends should set this bit automatically when called in a UTF-8 locale.

C support for Unicode and UTF-8

Starting with GNU glibc 2.2, the type `wchar_t` is officially 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 ISO C multi-byte conversion functions (`mbsrtowcs()`, `wcsrtombs()`, etc.)are fully implemented in glibc 2.2 or higher and can be used to convert between `wchar_t` and any locale-dependent multibyte encoding, including UTF-8, ISO 8859-1, etc.

For example, you can write

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

int main()
{
  if (!setlocale(LC_CTYPE, "")) {
    fprintf(stderr, "Can't set the specified locale! "
```

```
            "Check LANG, LC_CTYPE, LC_ALL.\n");
    return 1;
  }
  printf("%ls\n", L"Schöne Grüße");
  return 0;
}
```

Call this program with the locale setting `LANG=de_DE` and the output will be in ISO 8859-1. Call it with `LANG=de_DE.UTF-8` and the output will be in UTF-8. The `%ls` format specifier in `printf` calls `wcsrtombs` in order to convert the wide character argument string into the locale-dependent multi-byte encoding.

Many of C's string functions are locale-independent and they just look at zero-terminated byte sequences:

```
strcpy strncpy strcat strncat strcmp strncmp strdup strchr strrchr
strcspn strspn strpbrk strstr strtok
```

Some of these (e.g. strcpy) can equally be used for single-byte(ISO 8859-1) and multi-byte (UTF-8) encoded character sets, as they need no notion of how many byte long a character is, while others(e.g., strchr) depend on one character being encoded in a single char value and are of less use for UTF-8 (strchr still works fine if you just search for an ASCII character in a UTF-8 string).

Other C functions are locale dependent and work in UTF-8 locales just as well:

```
strcoll strxfrm
```

How should the UTF-8 mode be activated?

If your application is soft converted and does not use the standard locale-dependent C multibyte routines (`mbsrtowcs()`,`wcsrtombs()`, etc.) to convert everything into

`wchar_t` for processing, then it might have to find out in some way, whether it is supposed to assume that the text data it handles is in some 8-bit encoding (like ISO 8859-1, where 1 byte = 1character) or UTF-8. Once everyone uses only UTF-8, you can just make it the default, but until then both the classical 8-bit sets and UTF-8may still have to be supported.

The first wave of applications with UTF-8 support used a whole lot of different command line switches to activate their respective UTF-8modes, for instance the famous `xterm -u8`. That turned out to be a very bad idea. Having to remember a special command line option or other configuration mechanism for *every* application is very tedious, which is why command line options are **not** the proper way of activating a UTF-8 mode.

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 behaviour, including the character encoding, the date/time notation, alphabetic sorting rules, the measurement system and common office paper size, etc. The names of locales usually consist of ISO639-1 language and ISO3166-1 country codes, sometimes with additional encoding names or other qualifiers.

You can get a list of all locales installed on your system (usually in `/usr/lib/locale/`) with the command `locale-a`. Set the environment variable `LANG` to the name of your preferred locale. When a C program executes the `setlocale(LC_CTYPE, "")` function, the library will test the environment variables `LC_ALL`, `LC_CTYPE`, and `LANG` in that order, and the first one of these that has a value will determine which locale data is loaded for the `LC_CTYPE` category (which controls the multibyte conversion functions). The locale data is split up into separate categories. For

example, LC_CTYPE defines the character encoding and LC_COLLATE defines the string sorting order. The LANG environment variable is used to set the default locale for all categories, but the LC_* variables can be used to override individual categories. Do not worry too much about the country identifiers in the locales. Locales such as en_GB (English in Great Britain) and en_AU (English in Australia) differ usually only in the LC_MONETARY category (name of currency, rules for printing monetary amounts),which practically no Linux application ever uses. LC_CTYPE=en_GB and LC_CTYPE=en_AU have exactly the same effect.

You can query the name of the character encoding in your current locale with the command locale charmap. This should sayUTF-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 multibyte functions to do all the conversion between the external character encoding and the wchar_t encoding that you use internally, then the Clibrary will take care of using the right encoding according to LC_CTYPE for you and your program does not even have to know explicitly what the current multibyte encoding is.

However, if you prefer not to do everything using the libc multi-byte functions (e.g., because you think this would require too many changes in your software or is not efficient enough), then your application has to find out for itself when to activate the UTF-8mode. To do this, on any X/Open compliant systems, where <langinfo.h> is available, you can use a line such as

```
utf8_mode = (strcmp(nl_langinfo(CODESET), "UTF-8") == 0);
```

in order to detect whether the current locale uses the UTF-8encoding. You have of course to add a `setlocale(LC_CTYPE,"")` at the beginning of your application to set the locale according to the environment variables first. The standard function call `nl_langinfo(CODESET)` is also what `localecharmap` calls to find the name of the encoding specified by the current locale for you. It is available on pretty much every modern Unix now. FreeBSD added `nl_langinfo(CODESET)` support with version 4.6 (2002-06). If you need an autoconf test for the availability of `nl_langinfo(CODESET)`, here is the one Bruno Haible suggested:

```
======================== m4/codeset.m4 ================================
#serial AM1

dnl From Bruno Haible.

AC_DEFUN([AM_LANGINFO_CODESET],
[
  AC_CACHE_CHECK([for nl_langinfo and CODESET], am_cv_langinfo_codeset,
    [AC_TRY_LINK([#include <langinfo.h>],
      [char* cs = nl_langinfo(CODESET);],
      am_cv_langinfo_codeset=yes,
      am_cv_langinfo_codeset=no)
    ])
  if test $am_cv_langinfo_codeset = yes; then
    AC_DEFINE(HAVE_LANGINFO_CODESET, 1,
      [Define if you have <langinfo.h> and nl_langinfo(CODESET).])
  fi
])
=======================================================================
```

[You could also try to query the locale environment variables yourself without using `setlocale()`. In the sequence `LC_ALL`, `LC_CTYPE`, `LANG`, look for the first of these environment variables that has a value. Make the UTF-8 mode the default (still overridable by command line switches) when this value contains the substring `UTF-8`,as this indicates reasonably reliably that the

C library has been asked to use a UTF-8 locale. An example code fragment that does this is

```
char *s;
int utf8_mode = 0;

if (((s = getenv("LC_ALL"))   && *s) ||
    ((s = getenv("LC_CTYPE")) && *s) ||
    ((s = getenv("LANG"))     && *s)) {
  if (strstr(s, "UTF-8"))
    utf8_mode = 1;
}
```

This relies of course on all UTF-8 locales having the name of the encoding in their name, which is not always the case, therefore the `nl_langinfo()` query is clearly the better method. If you are really concerned that calling `nl_langinfo()` might not be portable enough, there is also Markus Kuhn's portable public domain `nl_langinfo(CODESET)` emulator for systems that do not have the real thing (and another one from Bruno Haible), and you can use the `norm_charmap()` function to standardize the output of the `nl_langinfo(CODESET)` on different platforms.]

How do I get a UTF-8 version of xterm?

The xterm version that comes with XFree864.0 or higher (maintained by Thomas Dickey) includes UTF-8 support. To activate it, start xterm in aUTF-8 locale and use a font with `iso10646-1` encoding, for instance with

```
LC_CTYPE=en_GB.UTF-8 xterm \
  -fn '-Misc-Fixed-Medium-R-SemiCondensed--13-120-75-75-C-60-ISO10646-1'
```

and then cat some example file, such as UTF-8-demo.txtin the newly started xterm and enjoy what you see.

If you are not using XFree86 4.0 or newer, then you can alternatively download the latest xterm development version separately and compile it yourself with "`./configure --enable-wide-chars ; make`" or alternatively with "`xmkmf; make Makefiles; make; make install; makeinstall.man`".

If you do not have UTF-8 locale support available, use command line option `-u8` when you invoke xterm to switch input and output to UTF-8.

How much of Unicode does xterm support?

Xterm in XFree86 4.0.1 only supported Level 1 (no combining characters) of ISO 10646-1 with fixed character width and left-to-right writing direction. In other words, the terminal semantics were basically the same as for ISO 8859-1, except that it can now decode UTF-8 and can access 16-bit characters.

With XFree86 4.0.3, two important functions were added:

- automatic switching to a double-width font for CJK ideographs
- simple overstriking combining characters

If the selected normal font is *X* × *Y* pixels large, then xterm will attempt to load in addition a*2X* × *Y* pixels large font (same XLFD, except for a doubled value of the `AVERAGE_WIDTH` property). It will use this font to represent all Unicode characters that have been assigned the *East Asian Wide (W)* or *East Asian Full Width(F)* property in Unicode Technical Report #11.

The following fonts coming with XFree86 4.x are suitable for display of Japanese and Korean Unicode text with terminal emulators and editors:

```
6x13    -Misc-Fixed-Medium-R-SemiCondensed--13-120-75-75-C-60-ISO10646-1
6x13B   -Misc-Fixed-Bold-R-SemiCondensed--13-120-75-75-C-60-ISO10646-1
6x13O   -Misc-Fixed-Medium-O-SemiCondensed--13-120-75-75-C-60-ISO10646-1
12x13ja -Misc-Fixed-Medium-R-Normal-ja-13-120-75-75-C-120-ISO10646-1

9x18    -Misc-Fixed-Medium-R-Normal--18-120-100-100-C-90-ISO10646-1
9x18B   -Misc-Fixed-Bold-R-Normal--18-120-100-100-C-90-ISO10646-1
18x18ja -Misc-Fixed-Medium-R-Normal-ja-18-120-100-100-C-180-ISO10646-1
18x18ko -Misc-Fixed-Medium-R-Normal-ko-18-120-100-100-C-180-ISO10646-1
```

Some simple support for non-spacing or enclosing combining characters (i.e., those with general category code Mn or Me in the Unicode database) is now also available, which is implemented by just overstriking (logical OR-ing) a base-character glyph with up to two combining-character glyphs. This produces acceptable results for accents below the base line and accents on top of small characters. It also works well, for example, for Thai and Korean Hangul Conjoining Jamo fonts that were specifically designed for use with overstriking. However, the results might not be fully satisfactory for combining accents on top of tall characters in some fonts, especially with the fonts of the “fixed” family. Therefore precomposed characters will continue to be preferable where available.

The fonts below that come with XFree86 4.x are suitable for display of Latin etc. combining characters (extra head-space). Other fonts will only look nice with combining accents on small x-high characters.

```
6x12    -Misc-Fixed-Medium-R-Semicondensed--12-110-75-75-C-60-ISO10646-1
9x18    -Misc-Fixed-Medium-R-Normal--18-120-100-100-C-90-ISO10646-1
9x18B   -Misc-Fixed-Bold-R-Normal--18-120-100-100-C-90-ISO10646-1
```

The following fonts coming with XFree86 4.x are suitable for display of Thai combining characters:

```
6x13    -Misc-Fixed-Medium-R-SemiCondensed--13-120-75-75-C-60-ISO10646-1
9x15    -Misc-Fixed-Medium-R-Normal--15-140-75-75-C-90-ISO10646-1
9x15B   -Misc-Fixed-Bold-R-Normal--15-140-75-75-C-90-ISO10646-1
10x20   -Misc-Fixed-Medium-R-Normal--20-200-75-75-C-100-ISO10646-1
9x18    -Misc-Fixed-Medium-R-Normal--18-120-100-100-C-90-ISO10646-1
```

The fonts18x18ko,18x18Bko,16x16Bko, and16x16koare suitable for displaying Hangul Jamo (using the same simple overstriking character mechanism used for Thai).

A note for programmers of text mode applications:

With support for CJK ideographs and combining characters, the output of xterm behaves a little bit more like with a proportional font, because a Latin/Greek/Cyrillic/etc. character requires one column position, a CJK ideograph two, and a combining character zero.

The Open Group's Single UNIX Specification specifies the two C functions wcwidth() and wcswidth() that allow an application to test how many column positions a character will occupy:

```
#include <wchar.h>
int wcwidth(wchar_t wc);
int wcswidth(const wchar_t *pwcs, size_t n);
```

Markus Kuhn's free wcwidth() implementation can be used by applications on platforms where the Clibrary does not yet provide a suitable function.

Xterm will for the foreseeable future probably not support the following functionality, which you might expect from a more sophisticated full Unicode rendering engine:

- bidirectional output of Hebrew and Arabic characters

- substitution of Arabic presentation forms
- substitution of Indic / Syriac ligatures
- arbitrary stacks of combining characters

Hebrew and Arabic users will therefore have to use application programs that reverse and left-pad Hebrew and Arabic strings before sending them to the terminal. In other words, the bidirectional processing has to be done by the application and not by xterm. The situation for Hebrew and Arabic improves over ISO 8859 at least in the form of the availability of precomposed glyphs and presentation forms. It is far from clear at the moment, whether bidirectional support should really go into xterm and how precisely this should work. BothISO6429 = ECMA-48 and the Unicode bidi algorithm provide alternative starting points. See also ECMA Technical Report TR/53.

If you plan to support bidirectional text output in your application, have a look at either Dov Grobgeld's FriBidi or Mark Leisher's Pretty Good Bidi Algorithm, two free implementations of the Unicode bidi algorithm.

Xterm currently does not support the Arabic, Syriac, or Indic text formatting algorithms, although Robert Brady has published some experimental patches towards bidi support. It is still unclear whether it is feasible or preferable to do this in a VT100 emulator at all. Applications can apply the Arabic and Hangul formatting algorithms themselves easily, because xterm allows them to output the necessary presentation forms. For Hangul, Unicode contains the presentation forms needed for modern(post-1933) Korean orthography. For Indic scripts, the X font mechanism at the moment does not even support the encoding of the necessary ligature variants, so there is little xterm could offer anyway. Applications requiring Indic or Syriac output should better use a

proper Unicode X11 rendering library such as Pango instead of a VT100 emulator like xterm.

Where do I find ISO 10646-1 X11 fonts?

Quite a number of Unicode fonts have become available for X11 over the past few months, and the list is growing quickly:

- Markus Kuhn together with a number of other volunteers has extended the old `-misc-fixed-*-iso8859-1` fonts that come with X11 towards a repertoire that covers all European characters(Latin, Greek, Cyrillic, intl. phonetic alphabet, mathematical and technical symbols, in some fonts even Armenian, Georgian, Katakana, Thai, and more). For more information see the Unicode fonts and tools for X11 page. These fonts are now also distributed with XFree86 4.0.1 or higher.
- Markus has also prepared ISO10646-1 versions of all the Adobe and B&H BDF fonts in the X11R6.4distribution. These fonts already contained the full PostScript font repertoire (around 30 additional characters, mostly those used also by CP1252 MS-Windows, e.g. smart quotes, dashes, etc.), which were however not available under the ISO 8859-1 encoding. They are no wall accessible in the ISO 10646-1 version, along with many additional precomposed characters covering ISO 8859-1,2,3,4,9,10,13,14,15. These fonts are now also distributed with XFree86 4.1 or higher.
- XFree86 4.0 comes with an integrated TrueType font engine that can make available any Apple/Microsoft font to your X application in the ISO 10646-1 encoding.

- Some future XFree86 release might also remove most old BDF fonts from the distribution and replace them with ISO 10646-1 encoded versions. The X server will be extended with an automatic encoding converter that creates other font encodings such as ISO 8859-* from the ISO 10646-1 font file on-the-fly when such a font is requested by old 8-bit software. Modern software should preferably use the ISO10646-1 font encoding directly.
- ClearlyU(cu12) is a 12 point, 100 dpi proportional ISO 10646-1 BDF font for X11 with over 3700 characters by Mark Leisher (example images).
- The Electronic Font Open Laboratory in Japan is also working on a family of Unicode bitmap fonts.
- Dmitry Yu. Bolkhovityanov created a Unicode VGA font in BDF for use by text mode IBM PC emulators etc.
- Roman Czyborra's GNU Unicode font project works on collecting a complete and free8×16/16×16 pixel Unicode font. It currently covers over 34000characters.
- etl-unicode is an ISO 10646-1 BDF font prepared by Primoz Peterlin.
- Primoz Peterlin has also started the freefont project, which extends to better UCS coverage some of the 35 core PostScript outline fonts that URW++ donated to the ghostscript project, with the help of pfaedit.
- George Williams has created a Type1Unicode font family, which is also available in BDF. He also developed the PfaEdit PostScript and bitmap font editor.
- EversonMono is a shareware monospaced font with over 3000 European glyphs, also available from the DKUUG server.

- Birger Langkjer has prepared a Unicode VGA Console Font for Linux.
- Alan Wood has a list of Microsoft fonts that support various Unicode ranges.
- CODE2000is a Unicode font by James Kass.

Unicode X11 font names end with `-ISO10646-1`. This is now the officially registered value for the X Logical Font Descriptor (XLFD) fields `CHARSET_REGISTRY` and `CHARSET_ENCODING` for all Unicode and ISO 10646-1 16-bit fonts. The`*-ISO10646-1` fonts contain some unspecified subset of the entire Unicode character set, and users have to make sure that whatever font they select covers the subset of characters needed by them.

The `*-ISO10646-1` fonts usually also specify a `DEFAULT_CHAR` value that points to a special non-Unicode glyph for representing any character that is not available in the font(usually a dashed box, the size of an H, located at 0x00). This ensures that users at least see clearly that there is an unsupported character. The smaller fixed-width fonts such as 6x13 etc. for xterm will never be able to cover all of Unicode, because many scripts such as Kanji can only be represented in considerably larger pixel sizes than those widely used by European users. Typical Unicode fonts for European usage will contain only subsets of between 1000 and 3000characters, such as the CEN MES-3repertoire.

You might notice that in the *-ISO10646-1 fonts the shapes of the ASCII quotation marks has slightly changed to bring them in line with the standards and practice on other platforms.

What are the issues related to UTF-8 terminal emulators?

VT100 terminal emulators accept ISO2022 (=ECMA-35)ESC sequences in order to switch between different character sets.

UTF-8 is in the sense of ISO 2022 an “other coding system” (see section 15.4 of ECMA 35). UTF-8 is outside the ISO 2022SS2/SS3/G0/G1/G2/G3 world, so if you switch from ISO 2022 to UTF-8,all SS2/SS3/G0/G1/G2/G3 states become meaningless until you leaveUTF-8 and switch back to ISO 2022. UTF-8 is a stateless encoding, i.e. a self-terminating short byte sequence determines completely which character is meant, independent of any switching state. G0 and G1 in ISO 10646-1 are those of ISO 8859-1, and G2/G3 do not exist in ISO10646, because every character has a fixed position and no switching takes place. With UTF-8, it is not possible that your terminal remains switched to strange graphics-character mode after you accidentally dumped a binary file to it. This makes a terminal in UTF-8 mode much more robust than with ISO 2022 and it is therefore useful to have a way of locking a terminal into UTF-8 mode such that it cannot accidentally go back to the ISO 2022 world.

The ISO 2022 standard specifies a range of ESC % sequences for leaving the ISO 2022 world (designation of other coding system, DOCS),and a number of such sequences have been registered for UTF-8 in section 2.8 of the ISO 2375 International Register of Coded Character Sets:

- `ESC %G` activates UTF-8 with an unspecified implementation level from ISO 2022 in a way that allows to go back to ISO 2022 again.

- `ESC %@` goes back from UTF-8 to ISO 2022 in caseUTF-8 had been entered via `ESC %G`.
- `ESC %/G` switches to UTF-8 Level 1 with no return.
- `ESC %/H` switches to UTF-8 Level 2 with no return.
- `ESC %/I` switches to UTF-8 Level 3 with no return.

While a terminal emulator is in UTF-8 mode, any ISO 2022 escape sequences such as for switching G2/G3 etc. are ignored. The only ISO2022 sequence on which a terminal emulator might act in UTF-8 mode is `ESC %@` for returning from UTF-8 back to the ISO 2022scheme.

UTF-8 still allows you to use C1 control characters such as CSI, even though UTF-8 also uses bytes in the range 0x80-0x9F. It is important to understand that a terminal emulator in UTF-8 mode must apply the UTF-8 decoder to the incoming byte stream **before** interpreting any control characters. C1characters are UTF-8 decoded just like any other character aboveU+007F.

Many text-mode applications available today expect to speak to the terminal using a legacy encoding or to use ISO 2022 sequences for switching terminal fonts. In order to use such applications within aUTF-8 terminal emulator, it is possible to use a conversion layer that will translate between ISO 2022 and UTF-8 on the fly. Examples for such utilities are Juliusz Chroboczek's luit and pluto. If all you need is ISO 8859 support in a UTF-8 terminal, you can also use screen(version 4.0 or newer) by Michael Schröder and Jürgen Weigert. As implementation of ISO 2022 is a complex and error-prone task, better avoid implementing ISO 2022 yourself. Implement only UTF-8 and point users who need ISO 2022 at luit (or screen).

What UTF-8 enabled applications are available?

Warning: As of mid-2003, this section is becoming increasingly incomplete. UTF-8 support is now a pretty standard feature for most well-maintained packages. This list will soon have to be converted into a list of the most popular programs that still have problems withUTF-8.

Terminal emulation and communication

- xterm as shipped with XFree86 4.0 or higher works correctly in UTF-8 locales if you use an *-iso10646-1 font. Just try it with for example `LC_CTYPE=en_GB.UTF-8 xterm -fn'-Misc-Fixed-Medium-R-Normal--18-120-100-100-C-90-ISO10646-1'`.
- C-Kermit has supported UTF-8 as the transfer, terminal, and file character set since version 7.0.
- mlterm is a multi-lingual terminal emulator that supports UTF-8 among many other encodings, combining characters, XIM.
- Edmund Grimley Evans extended the BOGL Linux framebuffer graphics library with UCS font support and built a simpleUTF-8 console terminal emulator called `bterm` with it.
- Uterm purports to be a UTF-8 terminal emulator for the Linux framebuffer console.
- Pluto,Juliusz Chroboczek's paranormal Unicode converter, can guess which encoding is being used in a terminal session, and converts it on-the-fly to UTF-8. (Wonderful

for reading IRC channels with mixed ISO 8859 and UTF-8 messages!)

Editing and word processing

- Vim (the popular clone of the classic vi editor) supports UTF-8 with wide characters and up to two combining characters starting from version 6.0.
- Emacs has quite good basic UTF-8 support starting from version 21.3. Emacs 23 changed the internal encoding to UTF-8.
- Yudit is Gaspar Sinai's free X11 Unicode editor.
- Mined 2000 by Thomas Wolff is a very nice UTF-8 capable text editor, ahead of the competition with features such as not only support of double-width and combining characters, but also bidirectional scripts, keyboard mappings for a wide range of scripts, script-dependent highlighting, etc.
- JOE is a popular WordStar-like editor that supports UTF-8 as of version 3.0.
- Cooledit offersUTF-8 and UCS support starting with version 3.15.0.
- QEmacs is a small editor for use on UTF-8 terminals.
- less is a popular plain-text file viewer that had UTF-8 support since version348. (Version 358 had a bug related to the handling of UTF-8 characters and backspace underlining/boldification as used by nroff/man, for which a patch is available, version 381 still has problems with UTF-8 characters in the search-mode input line.)

- GNU bash and readline provide single-line editors and they introduced support for multi-byte character encodings, such as UTF-8, with versions bash 2.05b and readline 4.3.
- gucharmap and UMap are tools to select and paste any Unicode character into your application.
- LaTeX has supportedUTF-8 in its base package since March 2004(still experimental). You can simply write `\usepackage[utf8]{inputenc}` and then encode at least some of TeX's standard character repertoire in UTF-8 in your LaTeX sources. (Before that, UTF-8 was already available in the form of Dominique Unruh's package, which covered far more characters and was rather resource hungry.) XeTeX is a reengineered version of TeX that reads and understands (UTF-8 encoded) Unicode text.
- Abiword.

Programming

- Perl offers useable Unicode andUTF-8 support starting with version 5.8.1. Strings are now tagged in memory as either byte strings or character strings, and the latter are stored internally as UTF-8 but appear to the programmer just as sequences of UCS characters. There is now also comprehensive support for encoding conversion and normalization included. Read "man perluni intro" for details.
- Python got Unicode support added in version 1.6.
- Tcl/Tk started using Unicode as its base character set with version 8.1. ISO10646-1 fonts are supported in Tk from version 8.3.3 or newer.

- CLISP can work with all multi-byte encodings (including UTF-8) and with the functions `char-width` and `string-width` there is an API comparable to `wcwidth()` and `wcswidth()` available.

Mail and Internet

- The Mutt email client has worked since version 1.3.24 in UTF-8 locales. When compiled and linked with ncursesw (ncurses built with wide-character support), Mutt 1.3.x works decently in UTF-8locales under UTF-8 terminal emulators such as xterm.
- Exmh is a GUI frontend for the MH or nmh mail system and partially supports Unicode starting with version 2.1.1 if Tcl/Tk 8.3.3 or newer is used. To enable displaying UTF-8 email, make sure you have the *-iso10646-1 fonts installed and add to .Xdefaults the line "exmh.mime UCharsets: utf-8". Much of the Exmh-internal MIME charset-set mechanics however still dates from the days before Tcl8.1, therefore ignores Tcl/Tk's more recent Unicode support, and could now be simplified and improved significantly. In particular, writing or replying to UTF-8 mail is still broken.
- Most modern web browsers such as Mozilla Firefox have pretty decentUTF-8 support today.
- The popular Pine email client lacks UTF-8 support and is no longer maintained. Switch to its successor Alpine, a complete reimplementation by the same authors, which has excellentUTF-8 support.

Printing

- Cedilla is Juliusz Chroboczek's best-effort Unicode to PostScript text printer.
- Markus Kuhn's hpp is a very simple plain text formatter for HP PCL printers that supports the repertoire of characters covered by the standard PCL fixed-width fonts in all the character encodings for which your C library has a locale mapping. Markus Kuhn's utf2ps is a nearly quick-and-dirty proof-of-concept UTF-8 formatter for PostScript, that was only written to demonstrate which character repertoire can easily be printed using only the standard PostScript fonts and was never intended to be actually used.
- Some post-2004 HP printers have UTF-8PCL firmware support(more).The relevant PCL5 commands appear to be "ESC&t1008P" (encoding method: UTF-8) and "ESC(18N" (Unicode code page). Recent PCL printers from other manufacturers (e.g., Kyocera) also advertiseUTF-8 support (for SAP compatibility).
- The Common UNIX Printing System comes with a texttops tool that converts plaintext UTF-8 to PostScript.
- txtbdf2psby Serge Winitzki is a Perl script to print UTF-8 plaintext to PostScript using BDF pixel fonts.

Misc

- The PostgreSQL DBMS had support for UTF-8 since version 7.1, both as the frontend encoding, and as the backend storage encoding. Data conversion between frontend and backend encodings is performed automatically.
- FIGlet is a tool to output banner text in large letters using mono spaced characters as block graphics elements and added UTF-8 support in version 2.2.

- Charlint is a character normalization tool for the W3C character model.
- The first available UTF-8 tools for Unix came out of the Plan 9 project, Bell Lab's Unix successor and the world's first operating system usingUTF-8. Plan 9's Sam editor and 9termterminal emulator have also been ported to Unix. Wily started out as a Unix implementation of the Plan 9 Acme editor and is a mouse-oriented, text-based working environment for programmers. More recently the Plan 9 from User Space (akaplan9port) package has emerged, a port of many Plan 9 programs from their native Plan 9 environment to Unix-like operating systems.
- The Gnumeric spreadsheet is fully Unicode based from version 1.1.
- The Heirloom Toolchest is a collection of standard Unix utilities derived from original Unix material released as open source by Caldera with support for multibyte character sets, especially UTF-8.
- convmv is a tool to convert the filenames in entire directory trees from a legacy encoding to UTF-8.

What patches to improve UTF-8 support are available?

Many of these already have been included in the respective main distribution.

- The Advanced Utility Development subgroup of the OpenI18N(formerly Li18nux) project have prepared various internationalization patches for tools such as cut, fold, glibc, join, sed, uniq, xterm, etc. that might improve UTF-8 support.

- A collection of UTF-8 patches for various tools as well as a UTF-8support status list is in Bruno Haible's Unicode-HOWTO.
- Bruno Haible has also prepared various patches for stty, the Linux kernel tty, etc.
- The multilingualization patch (w3m-m17n) for the text-mode web browser w3m allows you to view documents in all the common encodings on a UTF-8 terminal like xterm(also switch option "Use alternate expression with ASCII for entity "to OFF after pressing "o"). Another multilingual version (w3mmee) is available as well (have not tried that yet).

Are there free libraries for dealing with Unicode available?

- Ulrich Drepper's GNU Clibrary glibc has featured since version 2.2 full multi-byte locale support for UTF-8, an ISO ISO 14651 sorting order algorithm, and it can recode into many other encodings. All current Linux distributions come with glibc 2.2 or newer, so you definitely should upgrade now if you are still using an earlier Linux C library.
- The International Components for Unicode (ICU) (formerly IBM Classes for Unicode) have become what is probably the most powerful cross-platform standard library for more advanced Unicode character processing functions.
- X.Net's xIUA is a package designed to retrofit existing code for ICU support by providing locale management so that users do not have to modify internal calling interfaces to pass locale parameters. It uses more familiar APIs, for example to collate you use xiua_strcoll, and is thread safe.

- Mark Leisher's UCData Unicode character property and bidi library as well as his `wchar_t` support test code.
- Bruno Haible's libiconv character-set conversion library provides an iconv() implementation, for use on systems which do not have one, or whose implementation cannot convert from/to Unicode.
  It also contains the libcharset character-encoding query library that allows applications to determine in a highly portable way the character encoding of the current locale, avoiding the portability concerns of using nl_langinfo(CODESET) directly.
- Bruno Haible'slibutf8 provides various functions for handling UTF-8 strings, especially for platforms that do not yet offer proper UTF-8 locales.
- Tom Tromey's libunicode library is part of the Gnome Desktop project, but can be built independently of Gnome. It contains various character class and conversion functions. (CVS)
- FriBidi is Dov Grobgeld's free implementation of the Unicode bidi algorithm.
- Markus Kuhn's free wcwidth()implementation can be used by applications on platforms where the C library does not yet provide an equivalent function to find, how many column positions a character or string will occupy on a UTF-8terminal emulator screen.
- Markus Kuhn's transtab is a transliteration table for applications that have to make a best-effort conversion from Unicode to ASCII or some 8-bit character set. It contains a comprehensive list of substitution strings for Unicode characters, comparable to the fallback notations that people use commonly in email and on typewriters to

represent unavailable characters. The table comes in ISO/IEC TR 14652 format, to allow simple inclusion into POSIX locale definition files.

What is the status of Unicode support for various X widget libraries?

- The Pango – Unicode and Complex Text Processing project added full-featured Unicode support to GTK+.
- Qt supported the use of*-ISO10646-1 fonts since version 2.0.
- A UTF-8 extension for the Fast Light Tool Kit was prepared by Jean-Marc Lienher, based on his Xutf8 Unicode display library.

What packages with UTF-8 support are currently under development?

- Native Unicode support is planned for Emacs 23. If you are interested in contributing/testing, please join the `emacs-devel @gnu.org` mailing list.
- The Linux Console Project works on a complete revision of the VT100 emulator built into the Linux kernel, which will improve the simplistic UTF-8 support already there.

How does UTF-8 support work under Solaris?

Starting with Solaris 2.8, UTF-8 is at least partially supported. To use it, just set one of the UTF-8 locales, for instance by typing

```
setenv LANG en_US.UTF-8
```

in a C shell.

Now the `dtterm` terminal emulator can be used to input and output UTF-8 text and the `mp` print filter will printUTF-8 files on PostScript printers. The `en_US.UTF-8`locale is at the moment supported by Motif and CDE desktop applications and libraries, but not by OpenWindows, XView, and OPENLOOK DeskSet applications and libraries.

For more information, read Sun's Overview of en_US.UTF-8 Locale Support web page.

Can I use UTF-8 on the Web?

Yes. There are two ways in which a HTTP server can indicate to a client that a document is encoded in UTF-8:

- Make sure that the HTTP header of a document contains the line
- ```
  Content-Type: text/html; charset=utf-8
  ```
  if the file is HTML, or the line
  ```
  Content-Type: text/plain; charset=utf-8
  ```
  if the file is plain text. How this can be achieved depends on your web server. If you use Apache and you have a subdirectory in which all *.html or *.txt files are encoded in UTF-8, then create there a file .htaccess and add to it the two lines
  ```
  AddType text/html;charset=UTF-8 html
  AddType text/plain;charset=UTF-8 txt
  ```
  A webmaster can modify /etc/httpd/mime.types to make the same change for all subdirectories simultaneously.
- If you cannot influence the HTTP headers that the web server prefixes to your documents automatically, then add in a HTML document under HEAD the element
  ```

- `<META http-equiv=Content-Type content="text/html; charset=UTF-8">` which usually has the same effect. This obviously works only for HTML files, not for plain text. It also announces the encoding of the file to the parser only after the parser has already started to read the file, so it is clearly the less elegant approach.

The currently most widely used browsers support UTF-8 well enough to generally recommend UTF-8 for use on web pages. The old Netscape 4browser used an annoyingly large single font for displaying any UTF-8document. Best upgrade to Mozilla, Netscape 6 or some other recent browser (Netscape 4 is generally very buggy and not maintained anymore).

There is also the question of how non-ASCII characters entered into HTML forms are encoded in the subsequent HTTP GET or POST request that transfers the field contents to a CGI script on the server. Unfortunately, both standardization and implementation are still a huge mess here, as discussed in the FORM submission and i18n tutorial by Alan Flavell. We can only hope that a practice of doing all this in UTF-8 will emerge eventually. See also the discussion about Mozilla bug18643.

How are PostScript glyph names related to UCS codes?

See Adobe's Unicode and Glyph Names guide.

Are there any well-defined UCS subsets?

With over 40000 characters, the design of a font that covers every single Unicode character is an enormous project, not just regarding the number of glyphs that need to be created, but also

in terms of the calligraphic expertise required to do an adequate job for each script. As a result, there are hardly any fonts that try to cover “all of Unicode”. While a few projects have attempted to create single complete Unicode fonts, their quality is not comparable with that of many good smaller fonts. For example, the Unicode and ISO 10646 books are still printed using a large collection of different fonts that only together cover the entire repertoire. Any high-quality font can only cover the Unicode subset for which the designer feels competent and confident.

Older, regional character encoding standards defined both an encoding and a repertoire of characters that an individual calligrapher could handle. Unicode lacks the latter, but in the interest of interoperability, it is useful to have defined a hand full of standardized subsets, each a few hundred to a few thousand character large and targeted at particular markets, that font designers could practically aim to cover. A number of different UCS subsets already have been established:

- The Windows Glyph List 4.0 (WGL4) is a set of 650 characters that covers all the 8-bit MS-DOS, Windows, Mac, and ISO code pages that Microsoft had used before. All Windows fonts now cover at least the WGL4 repertoire.WGL4 is a superset of CEN MES-1. (WGL4 testfile).
- Three European UCS subsets MES-1, MES-2, and MES-3 have been defined by the European standards committee CEN/TC304 in CWA 13873:
  - MES-1 is a very small Latin subset with only 335 characters. It contains exactly all characters found in ISO 6937 plus the EURO SIGN. This means MES-1 contains all characters of ISO 8859 parts1,2,3,4,9,10,15. [Note: If your aim is to provide

only the cheapest and simplest reasonable Central European UCS subset, I would implementMES-1 plus the following important 14 additional characters found in Windows code page 1252 but not in MES-1: U+0192, U+02C6, U+02DC,U+2013, U+2014, U+201A, U+201E, U+2020, U+2021, U+2022, U+2026,U+2030, U+2039, U+203A.]

- MES-2 is a Latin/Greek/Cyrillic/Armenian/Georgian subset with 1052characters. It covers every language and every 8-bit code page used in Europe (not just the EU!) and European language countries. It also adds a small collection of mathematical symbols for use in technical documentation. MES-2 is a superset of MES-1. If you are developing only for a European or Western market, MES-2 is the recommended repertoire. [Note: For bizarre committee-politics reasons, the following eight WGL4 characters are missing from MES-2: U+2113,U+212E, U+2215, U+25A1, U+25AA, U+25AB, U+25CF, U+25E6. If you implement MES-2, you should definitely also add those and then you can claim WGL4 conformance in addition.]
- MES-3 is a very comprehensive UCS subset with 2819 characters. It simply includes every UCS collection that seemed of potential use to European users. This is for the more ambitious implementors. MES-3 is a superset of MES-2 and WGL4.

- JIS X 0221-1995 specifies 7 non-overlapping UCS subsets for Japanese users:
  - Basic Japanese (6884 characters): JIS X 0208-1997, JIS X 0201-1997

    - Japanese Non-ideographic Supplement (1913 characters): JIS X0212-1990 non-kanji, plus various other non-kanji
    - Japanese Ideographic Supplement 1 (918 characters): some JIS X0212-1990 kanji
    - Japanese Ideographic Supplement 2 (4883 characters): remaining JISX 0212-1990 kanji
    - Japanese Ideographic Supplement 3 (8745 characters): remaining Chinese characters
    - Full-width Alphanumeric (94 characters): for compatibility
    - Half-width Katakana (63 characters): for compatibility
- The ISO 10646 standard splits up its repertoire into a number of collections that can be used to define and document implemented subsets. Unicode defines similar, but not quite identical, blocks of characters, which correspond to sections in the Unicode standard.
- RFC 1815 is a memo written in 1995 by someone who obviously did not like ISO 10646and was unaware of JIS X 0221-1995. It discusses a UCS subset called"ISO-10646-J-1" consisting of 14 UCS collections, some of which are intersected with JIS X 0208. This is just what a particular font in an old Japanese Windows NT version from 1995 happened to implement. RFC1815 is completely obsolete and irrelevant today and should best be ignored.
- Markus Kuhn has defined in the ucs-fonts.tar.gz README three UCS subsets TARGET1, TARGET2, TARGET3 that are sensible extensions of the corresponding MES subsets and that were the basis for the completion of this xterm font package.

Markus Kuhn's uniset Perl script allows convenient set arithmetic over UCS subsets for anyone who wants to define a new one or wants to check coverage of an implementation.

What issues are there to consider when converting encodings

The Unicode Consortium maintains a collection of mapping tables between Unicode and various older encoding standards. It is important to understand that the primary purpose of these tables was to demonstrate that Unicode is a superset of the mapped legacy encodings, and to document the motivation and origin behind those Unicode characters that were included into the standard primarily for round-trip compatibility reasons with older character sets. The implementation of good character encoding conversion routines is a significantly more complex task than just blindly applying these example mapping tables! This is because some character sets distinguish characters that others unify.

The Unicode mapping tables alone are to some degree well suited to directly convert text from the older encodings to Unicode. High-end conversion tools nevertheless should provide interactive mechanisms, where characters that are unified in the legacy encoding but distinguished in Unicode can interactively or semi-automatically be disambiguated on a case-by-case basis.

Conversion in the opposite direction from Unicode to a legacy character set requires non-injective (= many-to-one) extensions of these mapping tables. Several Unicode characters have to be mapped to a single code point in many legacy encodings. The Unicode consortium currently does not maintain standard many-to-one tables for this purpose and does not define any standard behavior of coded character set conversion tools.

Here are some examples for the many-to-one mappings that have to be handled when converting from Unicode into something else:

UCS characters	equivalent character	in target code
U+00B5 MICRO SIGN U+03BC GREEK SMALL LETTER MU	0xB5	ISO 8859-1
U+00C5 LATIN CAPITAL LETTER A WITH RING ABOVE U+212B ANGSTROM SIGN	0xC5	ISO 8859-1
U+03B2 GREEK CAPITAL LETTER BETA U+00DF LATIN SMALL LETTER SHARP S	0xE1	CP437
U+03A9 GREEK CAPITAL LETTER OMEGA U+2126 OHM SIGN	0xEA	CP437
U+03B5 GREEK SMALL LETTER EPSILON U+2208 ELEMENT OF	0xEE	CP437
U+005C REVERSE SOLIDUS U+FF3C FULLWIDTH REVERSE SOLIDUS	0x2140	JIS X 0208

A first approximation of such many-to-one tables can be generated from available normalization information, but these then still have to be manually extended and revised. For example, it seems obvious that the character 0xE1 in the original IBM PC character set was meant to be useable as both a Greek small beta (because it is located between the code positions for alpha and gamma) and as a German sharp-s character (because

that code is produced when pressing this letter on a German keyboard). Similarly 0xEE can be either the mathematical element-of sign, as well as a small epsilon. These characters are not Unicode normalization equivalents, because although they look similar in low-resolution video fonts, they are very different characters in high-quality typography. IBM's tables for CP437 reflected one usage in some cases, Microsoft's the other, both equally sensible. A good code converter should aim to be compatible with both, and not just blindly use the Microsoft mapping table alone when converting from Unicode.

The Unicode database does contain in field 5 the Character Decomposition Mapping that can be used to generate some of the above example mappings automatically. As a rule, the output of a Unicode-to-Something converter should not depend on whether the Unicode input has first been converted into Normalization FormC or not. For equivalence information on Chinese, Japanese, and Korean Han/Kanji/Hanja characters, use the Unihan database. In the cases of the IBM PC characters in the above examples, where the normalization tables do not offer adequate mapping, the cross-references to similar looking characters in the Unicode book area valuable source of suggestions for equivalence mappings. In the end, which mappings are used and which not is a matter of taste and observed usage.

The Unicode consortium used to maintain mapping tables to CJK character set standards, but has declared them to be obsolete, because their presence on the Unicode web server led to the development of a number of inadequate and naive EUC converters. In particular, the (now obsolete) CJK Unicode mapping tables had to be slightly modified sometimes to preserve information in combination encodings. For example, the standard mappings provide round-trip compatibility for conversion chains

ASCII to Unicode to ASCII as well as for JIS X 0208to Unicode to JIS X 0208. However, the EUC-JP encoding covers the union of ASCII and JIS X 0208, and the UCS repertoire covered by the ASCII and JIS X 0208 mapping tables overlaps for one character, namelyU+005C REVERSE SOLIDUS. EUC-JP converters therefore have to use a slightly modified JIS X 0208 mapping table, such that the JIS X 0208code 0x2140 (0xA1 0xC0 in EUC-JP) gets mapped to U+FF3C FULLWIDTHREVERSE SOLIDUS. This way, round-trip compatibility from EUC-JP to Unicode to EUC-JP can be guaranteed without any loss of information. Unicode Standard Annex #11: East Asian Width provides further guidance on this issue. Another problem area is compatibility with older conversion tables, as explained in an essay by Tomohiro Kubota.

In addition to just using standard normalization mappings, developers of code converters can also offer transliteration support. Transliteration is the conversion of a Unicode character into a graphically and/or semantically similar character in the target code, even if the two are distinct characters in Unicode after normalization. Examples of transliteration:

UCS characters	equivalent character	in target code
U+0022 QUOTATION MARK U+201C LEFT DOUBLE QUOTATION MARK U+201D RIGHT DOUBLE QUOTATION MARK U+201E DOUBLE LOW-9 QUOTATION MARK U+201F DOUBLE HIGH-REVERSED-9 QUOTATION MARK	0x22	ISO 8859-1

The Unicode Consortium does not provide or maintain any standard transliteration tables at this time. CEN/TC304 has a draft report “European fallback rules” on recommended ASCII fallback characters forMES-2 in the pipeline, but this is not yet mature. Which transliterations are appropriate or not can in some cases depend on language, application field, and most of all personal preference. Available Unicode transliteration tables include, for example, those found in Bruno Haible’s libiconv, the glibc 2.2 locales, and Markus Kuhn’s transtab package.

Is X11 ready for Unicode?

The X11 R7.0 release(2005) is the latest version of the X Consortium’s sample implementation of the X11 Window System standards. The bulk of the current X11 standards and parts of the sample implementation still pre-date widespread interest in Unicode under Unix.

Among the things that have already been fixed are:

- **Keysyms:** Since X11R6.9, a keysym value has been allocated for every Unicode character in Appendix A of the X Window System Protocol specification. Any UCS character in the range U-00000100to U-00FFFFFF can now be represented by a keysym value in the range0x01000100 to 0x01ffffff. This scheme was proposed by Markus Kuhn in1998 and has been supported by a number of applications for many years, starting with xterm. The revised Appendix A now also contains an official UCS cross reference column in its table of pre-Unicode legacy keysyms.
- **UTF-8 locales:** The X11R6.8 sample implementation added support for UTF-8 locales.

- **Fonts:** A number of comprehensive Unicode standard fonts were added in X11R6.8, and they are now supported by some of the classic standard tools, such as xterm.

There remain a number of problems in the X11 standards and some inconveniences in the sample implementation for Unicode users that still need to be fixed in one of the next X11 releases:

- **UTF-8 cut and paste:** The ICCCM standard still does not specify how to transfer UCS strings in selections. Some vendors have added UTF-8 as yet another encoding to the existing COMPOUND_TEXT mechanism (CTEXT). This is not a good solution for at least the following reasons:
  - CTEXT is a rather complicated ISO 2022 mechanism and Unicode offers the opportunity to provide not just another add-on to CTEXT, but to replace the entire monster with something far simpler, more convenient, and equally powerful.
  - Many existing applications can communicate selections via CTEXT, but do not support a newly added UTF-8 option. A user of CTEXT has to decide whether to use the old ISO 2022 encodings or the new UTF-8encoding, but both cannot be offered simultaneously. In other words, adding UTF-8 to CTEXT seriously breaks backwards compatibility with existing CTEXT applications.
  - The current CTEXT specification even explicitly forbids the addition of UTF-8 in section 6: “ISO registered ‘other coding systems’ are not used in Compound Text; extended segments are the only mechanism for non-2022 encodings.”

Juliusz Chroboczek has written an Inter-Client Exchange of Unicode Text draft proposal for an extension of the ICCCM to handle UTF-8 selections with a newUTF8_STRING atom that can be used as a property type and selection target. This clean approach fixes all of the above problems.UTF8_STRING is just as state-less and easy to use as the existing STRING atom (which is reserved exclusively for ISO 8859-1 strings and therefore not usable for UTF-8), and adding a new selection target allows applications to offer selections in both the old CTEXT and the new UTF8_STRING format simultaneously, which maximizes interoperability. The use of UTF8_STRING can be negotiated between the selection holder and requestor, leading to no compatibility issues whatsoever. Markus Kuhn has prepared an ICCCM patch that adds the necessary definition to the standard. Current status: The UTF8_STRING atom has now been officially registered with X.Org, and we hope for an update of the ICCCM in one of the next releases.

- **Application window properties:** In order to assist the window manager in correctly labeling windows, the ICCCM 2.0 specification requires applications to assign properties such as WM_NAME, WM_ICON_NAME and WM_CLIENT_MACHINE to each window. The old ICCCM 2.0 (1993) defines these to be of the polymorphic type TEXT, which means that they can have their text encoding indicated using one of the property types STRING (ISO 8859-1), COMPOUND_TEXT (a ISO 2022subset), or C_STRING (unknown character set). Simply addingUTF8_STRING as a new option for TEXT would break backwards compatibility with old window managers that do not know about this type. Therefore, the

freedesktop.org draft standard developed in the Window Manager Specification Project adds new additional window properties _NET_WM_NAME, _NET_WM_ICON_NAME, etc. that have type UTF8_STRING.

- **Inefficient font data structures:** The Xlib API and X11 protocol data structures used for representing font metric information are extremely inefficient when handling sparsely populated fonts. The most common way of accessing a font in an X client is a call to XLoadQueryFont(), which allocates memory for an XFontStruct and fetches its content from the server. XfontStruct contains an array of XCharStruct entries (12 bytes each). The size of this array is the code position of the last character minus the code position of the first character plus one. Therefore, any"*-iso10646-1" font that contains both U+0020 and U+FFFD will cause an XCharStruct array with 65502 elements to be allocated (even for CharCell fonts), which requires 786 kilobytes of client-side memory and data transmission, even if the font contains only a thousand characters.

  A few workarounds have been used so far:

  - The non-Asian `-misc-fixed-*-iso10646-1` fonts that come with XFree86 4.0 contain no characters above U+31FF. This reduces the memory requirement to 153 kilobytes, which is still bad, but much less so. (There are actually many useful characters above U+31FFpresent in the BDF files, waiting for the day when this problem will be fixed, but they currently all have an encoding of -1 and are therefore ignored by the X server. If you need these

characters, then just install the original fonts without applying the `bdftruncate` script).

- Starting with XFree86 4.0.3, the truncation of a BDF font can also be done by specifying a character code subrange at the end of the XLFD, as described in the XLFD specification, section 3.1.2.12. For example,
  - `-Misc-Fixed-Medium-R-Normal--20-200-75-75-C-100-ISO10646-1[0x1200_0x137f]`

  will load only the Ethiopic part of this BDF font with a correspondingly nicely small XFontStruct. Earlier X server versions will simply ignore the font subset brackets and will give you the full font, so there is no compatibility problem with using that.
- Bruno Haible has written a BIGFONT protocol extension for XFree864.0, which uses a compressed transmission of XCharStruct from server to client and also uses shared memory in Xlib between several clients which have loaded the same font.

These workarounds do not solve the underlying problem that XFontStruct is unsuitable for sparsely populated fonts, but they do provide a significant efficiency improvement without requiring any changes in the API or client source code. One real solution would be to extend or replace XFontStruct with something slightly more flexible that contains a sorted list or hash table of characters as opposed to an array. This redesign of XFontStruct would at the same time also allow the addition of the urgently needed provisions for combining characters and ligatures.

Another approach would be to introduce a new font encoding, which could be called for instance “ISO10646-C” (the C stands for combining, complex, compact, or character-glyph mapped, as you prefer). In this encoding,

the numbers assigned to each glyph are really font-specific glyph numbers and are not equivalent to any UCS character code positions. The information necessary to do a character-to-glyph mapping would have to be stored in to be standardized new properties. This new font encoding would be used by applications together with a few efficient C functions that perform the character-to-glyph code mapping:

- `makeiso10646cglyphmap(XFontStruct *font, iso10646cglyphmap*map)`
  Reads the character-to-glyph mapping table from the font properties into a compact and efficient in-memory representation.
- `freeiso10646cglyphmap(iso10646cglyphmap *map)`
  Frees that in-memory representation.
- `mbtoiso10646c(char *string, iso10646cglyphmap *map, XChar2b*output)`
  `wctoiso10646c(wchar_t *string, iso10646cglyphmap *map,XChar2b *output)`
  These take a Unicode character string and convert it into a `XChar2b` glyph string suitable for output by `XDrawString16` with the ISO10646-C font from which the `iso10646cglyphmap` was extracted.

ISO10646-C fonts would still be limited to having not more than 64kibiglyphs,but these can come from anywhere in UCS, not just from the BMP. This solution also easily provides for glyph substitution, such that we can finally handle the Indic fonts. It solves the huge-XFontStruct problem of ISO10646-1, as XFontStruct grows now proportionally with the number of glyphs, not with the

highest characters. It could also provide for simple overstriking combining characters, but then the glyphs for combining characters would have to be stored with negative width inside an ISO10646-C font. It can even provide support for variable combining accent positions, by having several alternative combining glyphs with accents at different heights for the same combining character, with the ligature substitution tables encoding which combining glyph to use with which base character.

TODO: write specification for ISO10646-C properties, write sample implementations of the mapping routines, and add these to xterm, GTK, and other applications and libraries. Any volunteers?

- **Combining characters:** The X11 specification does not support combining characters in any way. The font information lacks the data necessary to perform high-quality automatic accent placement (as it is found, for example, in all TeX fonts). Various people have experimented with implementing simplest overstriking combining characters using zero-width characters with ink on the left side of the origin, but details of how to do this exactly are unspecified(e.g., are zero-width characters allowed in CharCell and Monospaced fonts?) and this is therefore not yet widely established practice.
- **Ligatures:** The Indic scripts need font file formats that support ligature substitution, which is at the moment just as completely out of the scope of the X11 specification as are combining characters.

Several XFree86 team members have worked on these issues. X.Org, the official successor of the XConsortium and the Open

group as the custodian of the X11 standards and the sample implementation, has taken over the results or is still considering them.

With regard to the font related problems, the solution will probably be to dump the old server-side font mechanisms entirely and use instead XFree86's new Xft. Another related work-in-progress is Standard Type Services (ST)framework that Sun has been working on.

What are useful Perl one-liners for working with UTF-8?

These examples assume that you have Perl 5.8.1 or newer and that you work in a UTF-8 locale (i.e., "locale charmap" outputs "UTF-8").

For Perl 5.8.0, option `-C` is not needed and the examples without `-C` will not work in a UTF-8 locale. You really should no longer use Perl 5.8.0, as its Unicode support had lots of bugs.

Print the euro sign (U+20AC) to stdout:

```
  perl -C -e 'print pack("U",0x20ac)."\n"'
  perl -C -e 'print "\x{20ac}\n"'           # works only from U+0100
upwards
```

Locate malformed UTF-8 sequences:

```
  perl -ne '/^(([\x00-\x7f]|[\xc0-\xdf][\x80-\xbf]|[\xe0-\xef][\x80-
\xbf]{2}|[\xf0-\xf7][\x80-\xbf]{3})*)(.*)$/;print "$ARGV:$.:".($-
[3]+1).":$_" if length($3)'
```

Locate non-ASCII bytes:

```
  perl -ne '/^([\x00-\x7f]*)(.*)$/;print "$ARGV:$.:".($-[2]+1).":$_" if
length($2)'
```

Convert non-ASCII characters into SGML/HTML/XML-style decimal numeric character references (e.g. Ş becomesŞ):

```
perl -C -pe 's/([^\x00-\x7f])/sprintf("&#%d;", ord($1))/ge;'
```

Convert (hexa)decimal numeric character references to UTF-8:

```
perl -C -pe 's/&\#(\d+);/chr($1)/ge;s/&\#x([a-fA-
F\d]+);/chr(hex($1))/ge;'
```

How can I enter Unicode characters?

There are a range of techniques for entering Unicode characters that are not present by default on your keyboard.

Application-independent methods

- Copy-and-paste from a small file that lists your most commonly used Unicode characters in a convenient and for your needs suitably chosen arrangement. This is usually the most convenient and appropriate method for relatively rarely required very special characters, such as more esoteric mathematical operators.
- Extend your keyboard mapping using xmodmap. This is particularly convenient if your keyboard has an AltGr key, which is meant for exactly this purpose (some US keyboards have instead of AltGr just aright Alt key, others lack that key entirely unfortunately, in which case some other key must be assigned the Mode_switch function). Write a file "~/.Xmodmap" with entries such as

```
keycode 113 = Mode_switch Mode_switch
keysym d = d NoSymbol degree        NoSymbol
keysym m = m NoSymbol emdash        mu
keysym n = n NoSymbol endash        NoSymbol
keysym 2 = 2 quotedbl twosuperior   NoSymbol
keysym 3 = 3 sterling threesuperior NoSymbol
```

- `keysym 4 = 4 dollar EuroSign NoSymbol`
- `keysym space = space NoSymbol nobreakspace NoSymbol`
- `keysym minus = minus underscore U2212 NoSymbol`
- `keycode 34 = bracketleft braceleft leftsinglequotemark leftdoublequotemark`
- `keycode 35 = bracketright braceright rightsinglequotemark rightdoublequotemark`
- `keysym KP_Subtract = KP_Subtract NoSymbol U2212 NoSymbol`
- `keysym KP_Multiply = KP_Multiply NoSymbol multiply NoSymbol`
- `keysym KP_Divide = KP_Divide NoSymbol division NoSymbol`

and load it with "xmodmap ~/.Xmodmap" from your X11 startup script into your X server. You will then find that you get with AltGr easily the following new characters out of your keyboard:

AltGr+d	°
AltGr+	NBSP
AltGr+[	‘
AltGr+]	’
AltGr+{	“
AltGr+}	”
AltGr+2	²
AltGr+3	³
AltGr+-	–
AltGr+n	–
AltGr+m	—
AltGr+M	µ
AltGr+keypad-/	÷
AltGr+keypad-*	×

The above example file is meant for a UK keyboard, but easily adapted to other layouts and extended with your own choice of characters. If you use Microsoft Windows,

try Microsoft Keyboard Layout Creator to make similar customizations.

- ISO 14755 defines a hexadecimal input method: Hold down both the Ctrl and Shift key while typing the hexadecimal Unicode number. After releasing Ctrl and Shift, you have entered the corresponding Unicode character.

  This is currently implemented in GTK+ 2, and works in applications such as GNOME Terminal, Mozilla and Firefox.

Application-specific methods

- In VIM, type Ctrl-V u followed by a hexadecimal number. Example: Ctrl-V u 20ac
- In Microsoft Windows, press the Alt key while typing the decimal Unicode number with a leading zero on the numeric keypad. Example: press-Alt 08364 release-Alt
- In Microsoft Word, type a hexadecimal number and then press Alt+X to turn it into the corresponding Unicode character. Example: 20ac Alt-X

Are there any good mailing lists on these issues?

You should certainly be on the `linux-utf8@nl.linux.org`mailing list. That's the place to meet for everyone interested in working towards better UTF-8 support for GNU/Linux or Unix systems and applications. To subscribe, send a message to linux-utf8-request@nl.linux.org with the subject `subscribe`. You can also browse the linux-utf8 archive and subscribe from there via a web interface.

There is also the unicode@unicode.org mailing list, which is the best way of finding out what the authors of the Unicode standard and a lot of other gurus have to say. To subscribe, send to unicode-request@unicode.orga message with the subject line "subscribe" and the text "subscribe *YOUR@EMAIL.ADDRESS* unicode".

The relevant mailing list for discussions about Unicode support in Xlib and the X server is now xorg at xorg.org. In the past, there were also the fonts and i18n atxfree86.org mailing lists, whose archives still contain valuable information.

Further references

- Bruno Haible's Unicode HOWTO.
- The Unicode Standard, Version 5.0, Addison-Wesley, 2006. You definitely should have a copy of the standard if you are doing anything related to fonts and character sets.
- Ken Lunde's CJKV Information Processing, O'Reilly & Associates, 1999. This is clearly the best book available if you are interested in East Asian character sets.
- Unicode Technical Reports
- Mark Davis' Unicode FAQ
- ISO/IEC10646-1:2000
- Frank Tang's Iñtërnâtiônàlizætiøn Secrets
- IBM's Unicode Zone
- Unicode Support in the Solaris 7 Operating Environment
- The USENIX Winter 1993 paper by Rob Pike and Ken Thompson on the introduction of UTF-8 under Plan 9 reports about the experience gained when Plan 9 migrated as the first operating system back in 1992 completely to

UTF-8 (which was at the time still called UTF-2). A must read!

- OpenI18N is a project initiated by several Linux distributors to enhance Unicode support for free operating systems. It published the OpenI18NGlobalization Specification, as well as some patches.
- The Online Single Unix Specification contains definitions of all the ISO C Amendment 1 function, plus extensions such as wcwidth().
- The Open Group's summary of ISOC Amendment 1.
- GNU libc
- The Linux Console Tools
- The Unicode Consortium character database and character set conversion tables are an essential resource for anyone developing Unicode related tools.
- Other conversion tables are available from Microsoft and Keld Simonsen's WG15 archive.
- Michael Everson's Unicode and JTC1/SC2/WG2Archive contains online versions of many of the more recent ISO10646-1 amendments, plus many other goodies. See also his Roadmaps to the Universal Character Set.
- An introduction into The Universal Character Set (UCS).
- Otfried Cheong's essay on Han Unification in Unicode
- The AMS STIX project revised and extended the mathematical characters for Unicode 3.2 and ISO 10646-2. They are now preparing a freely available the STIX Fonts family of fully hintedType1 and TrueType fonts, covering the over 7700 characters needed for scientific publishing in a "Times compatible" design.

- Jukka Korpela's Soft hyphen (SHY) –a hard problem? is an excellent discussion of the controversy surrounding U+00AD.
- James Briggs' Perl, Unicode and I18N FAQ.
- Mark Davis discusses in Forms of Unicode the tradeoffs between UTF-8, UTF-16, and UCS-4 (now also called UTF-32 for political reasons). Doug Ewell wrote A survey of Unicode compression.
- Alan Wood has a good page on Unicode and Multilingual Support in Web Browsers and HTML.
- ISO/JTC1/SC22/WG20 produced various Unicode related standards such as the International String Ordering (ISO 14651) and the Cultural Convention Specification TR (ISO TR 14652) (an extension of the POSIX locale format that covers, for example, transliteration of wide character output).
- ISO/JTC1/SC2/WG2/IRG(Ideographic Rapporteur Group)
- The Letter Database answers queries on languages, character sets and names, as does the Zvon Character Search.
- Vietnamese Unicode FAQs
- China has specified in GB 18030 a new encoding of UCS for use in Chinese government systems that is backwards-compatible with the widely used GB 2312 and GBK encodings for Chinese. It seems though that the first version(released 2000-03) is somewhat buggy and will likely go through a couple more revisions, so use with care. GB 18030 is probably more of a temporary migration path to UCS and will probably not survive for long against UTF-8 or UTF-16, even in Chinese government systems.

- Hong Kong Supplementary Character Set (HKSCS)
- Various people propose UCS alternatives: Rosetta, Bytext.
- Proceedings of the International Unicode Conferences: ICU13, ICU14, ICU15, ICU16, ICU17, ICU18, etc.
- This FAQ has been translated into other languages:
  - Korean: 2001-02

  Be aware that each translation reflects only some past version of this document, which I update several times per month and revise more thoroughly once or twice each year.

I add new material to this document quite frequently, so please come back from time to time. Suggestions for improvement are very welcome. Please help to spread the word in the free software community about the importance of UTF-8.

Special thanks to Ulrich Drepper, Bruno Haible, Robert Brady, Juliusz Chroboczek, Shuhei Amakawa, Jungshik Shi, Robert Rogers, Roman Czyborra, Josef Hinteregger and many others for valuable comments, and to SuSE GmbH, Nürnberg, for their past support.

Markus Kuhn