Introduction to i18n

Tomohiro KUBOTA <debianattmaildotplaladotordotjp(retiredDD)>

29 Dezember 2009

Abstract

This document describes basic concepts for i18n (internationalization), how to write an internationalized software, and how to modify and internationalize a software. Handling of characters is discussed in detail. There are a few case-studies in which the author internationalized softwares such as TWM.

Copyright Notice

Contents

Chapter 1

About This Document

1.1 Scope

This document describes the basic ideas of I18N; it's written for programmers and package maintainers of Debian GNU/Linux and other UNIX-like platforms. The aim of this document is to offer an introduction to the basic concepts, character codes, and points where care should be taken when one writes an I18N-ed software or an I18N patch for an existing software. There are many know-hows and case-studies on internationalization of softwares. This document also tries to introduce the current state and existing problems for each language and country.

Minimum requirements - for example, that characters should be displayed with fonts of the proper charset (users of the software must be able to at least guess what is written), that characters must be inputed from keyboard, and that softwares must not destroy characters - are stressed in the document. I am trying to describe a HOWTO to satisfy these requirements.

This document is strongly related to programming languages such as C and standardized I18N methods such as using locales and `gettext`.

1.2 New Versions of This Document

The current version of this document is available at DDP (Debian Documentation Project) (`http://www.debian.org/doc/ddp`) page.

Note that the author rewrote this document in November 2000.

Since then, Debian had several releases and its packages support I18N better with their supports of UTF-8. This document does not cover these new developments but is kept here since this helps understandings of fundamental I18N issues.

1.3 Feedback and Contributions

This document needs contributions, especially for a chapter on each languages (‘Characters in Each Country’ on page 33) and a chapter on instances of I18N (‘Examples of I18N’ on page 89). These chapters consist of contributions.

Otherwise, this will be a document only on Japanization, because the original author Tomohiro KUBOTA (<kubota@debian.org>, retired DD and this is not a working e-mail address any more) speaks Japanese and live in Japan.

‘Spanish language / used in Spain, most of America and Equatorial Guinea’ on page 42 is written by Eusebio C Rufian-Zilbermann <eusebio@acm.org>.

Discussions are held at debian-devel@lists.debian.org and debian-i18n@lists.debian.org mailing list. Please contact debian-doc@lists.debian.org if you wish to update this document.

Chapter 2

Introduction

2.1 General Concepts

Debian includes many pieces of software. Though many of them have the ability to process, input, and output text data, some of these programs assume text is written in English (ASCII). For people who use non-English languages, these programs are barely usable. And more, though many softwares can handle not only ASCII but also ISO-8859-1, some of them cannot handle multibyte characters for CJK (Chinese, Japanese, and Korean) languages, nor combined characters for Thai.

So far, people who use non-English languages have given up using their native languages and have accepted computers as they were. However, we should now forget such a wrong idea. It is absurd that a person who wants to use a computer has to learn English in advance.

I18N is needed in the following places.

- Displaying characters for the users' native languages.
- Inputing characters for the users' native languages.
- Handling files written in popular encodings [1] that are used for the users' native languages.
- Using characters from the users' native languages for file names and other items.
- Printing out characters from the users' native languages.
- Displaying messages by the program in the users' native languages.
- Formatting input and output of numbers, dates, money, etc., in a way that obeys customs of the users' native cultures.

[1] There are a few terms related to character code, such as character set, character code, charset, encoding, codeset, and so on. These words are explained later.

- Classifying and sorting characters, in a way that obey customs of the users' native cultures.
- Using typesetting and hyphenation rules appropriate for the users' native languages.

This document puts emphasis on the first three items. This is because these three items are the basis for the other items. An another reason is that you cannot use softwares lacking the first three items at all, while you can use softwares lacking the other items, albeit inconveniently. This document will also mention translation of messages (item 6) which is often called as 'I18N'. Note that the author regards the terminology of 'I18N' for calling translation and `gettext`ization as completely wrong. The reason may be well explained by the fact that the author did not include translation and `gettext`ization in the important first three items.

Imagine a word processor which can display error and help messages in your native language while cannot process your native language. You will easily understand that the word processor is not usable. On the other hand, a word processor which can process your native language, but only displays error and help messages in English, is usable, though it is not convenient. Before we think of developing convenient softwares, we have to think of developing usable softwares.

The following terminology is widely used.

- I18N (internationalization) means modification of a software or related technologies so that a software can potentially handle multiple languages, customs, and so on in the world.
- L10N (localization) means implementation of a specific language for an already internationalized software.

However, this terminology is valid only for one specific model out of a few models which we should consider for I18N. Now I will introduce a few models other than this I18N-L10N model.

a. L10N (localization) model This model is to support two languages or character codes, English (ASCII) and another specific one. Examples of softwares which is developed using this model are: Nemacs (Nihongo Emacs, an ancestor of MULE, MULtilingual Emacs) text editor which can input and output Japanese text files, and Hanterm X terminal emulator which can display and input Korean characters via a few Korean encodings. Since each programmer has his or her own mother tongue, there are numerous L10N patches and L10N programs written to satisfy his or her own need.

b. I18N (internationalization) model This model is to support many languages but only two of them, English (ASCII) and another one, at the same time. One have to specify the 'another' language, usually by `LANG` environmental variable. The above I18N-L10N model can be regarded as a part of this I18N model. `gettext`ization is categorized into I18N model.

c. M17N (multilingualization) model This model is to support many languages at the same time. For example, Mule (MULtilingual Enhancement to GNU Emacs) can handle a text file which contains multiple languages - for example, a paper on differences between Korean and Chinese whose main text is written in Finnish. GNU Emacs 20 and XEmacs now include Mule. Note that the M17N model can only be applied in character-related instances. For example, it is nonsense to display a message like 'file not found' in many languages at the same time. Unicode and UTF-8 are technologies which can be used for this model. [2]

Generally speaking, the M17N model is the best and the second-best is the I18N model. The L10N model is the worst and you should not use it except for a few fields where the I18N and M17N models are very difficult, like DTP and X terminal emulator. In other words, it is better for text-processing softwares to handle many languages at the same time, than handle two (English and another language).

Now let me classify approaches for support of non-English languages from another viewpoint.

A. Implementation *without* knowledge of each language This approach is done by utilizing standardized methods supplied by the kernel or libraries. The most important one is **locale** technology which includes **locale category**, conversion between **multibyte** and **wide characters** (`wchar_t`), and so on. Another important technology is `gettext`. The advantages of this approach are (1) that when the kernel or libraries are upgraded, the software will automatically support new additional languages, (2) that programmers need not know each language, and (3) that a user can switch the behavior of softwares with common method, like LANG variable. The disadvantage is that there are categories or fields where a standardized method is not available. For example, there are no standardized methods for text typesetting rules such as line-breaking and hyphenation.

B. Implementation using knowledge of each language This approach is to directly implement information about each language based on the knowledge of programmers and contributors. L10N almost always uses this approach. The advantage of this approach is that a detailed and strict implementation is possible beyond the field where standardized methods are available, such as auto-detection of encodings of text files to be read. Language-specific problems can be perfectly solved; of course, it depends on the skill of the programmer). The disadvantages are (1) that the number of supported languages is restricted by the skill or the interest of the programmers or the contributors, (2) that labor which should be united and concentrated to upgrade the kernel or libraries is dispersed into many softwares, that is, re-inventing of the wheel, and (3) a user has to learn how to configure each software, such as `LESSCHARSET` variable, `.emacs` file, and other methods. This approach can cause problems: for example, GNU roff (before version 1.16) assumes `0xad` as a hyphen character, which is valid only for ISO-8859-1. However, a majestic M17N software such as Mule can be built using this approach.

[2] I recommend not to implement Unicode and UTF-8 directly. Instead, use locale technology and your software will support not only UTF-8 but also many encodings in the world. If you implement UTF-8 directly, your software can handle UTF-8 only. Such a software is not convenient.

Using this classification, let me consider the L10N, I18N, and M17N models from the programmer's point of view.

The L10N model can be realized only using his or her own knowledge on his or her language (i.e. approach B). Since the motivation of L10N is usually to satisfy the programmer's own need, extendability for the third languages is often ignored. Though L10N-ed softwares are primarily useful for people who speaks the same language to the programmer, it is sometimes useful for other people whose coding system is similar to the programmer's. For example, a software which doesn't recognize EUC-JP but doesn't break EUC-JP, will not break EUC-KR also.

The main part of the I18N model is, in the case of a C program, achieved using standardized locale technology and `gettext`. An locale approach is classified into I18N because functions related to locale change their behavior by the current locales for six categories which are set by `setlocale()`. Namely, approach A is emphasized for I18N. For field where standardized methods are not available, however, approach B cannot be avoided. Even in such a case, the developers should be careful so that a support for new languages can be easily added later even by other developers.

The M17N model can be achieved using international encodings such as ISO 2022 and Unicode. Though you can hard-code these encodings for your software (i.e. approach B), I recommend to use standardized locale technology. However, using international encodings is not sufficient to achieve the M17N model. You will have to prepare a mechanism to switch **input methods**. You will also want to prepare an encoding-guessing mechanism for input files, such as `jless` and `emacs` have. Mule is the best software which achieved M17N (though it does not use locale technology).

2.2 Organization

Let's preview the contents of each chapter in this document.

As I wrote, this document will put stress on correct handling of characters and character codes for users' native languages. To achieve this purpose, I will start the real contents of this document by discussing basic important concepts on characters in 'Important Concepts for Character Coding Systems' on page 9. Since this chapter includes many terminologies, all of you will need to this chapter. The next chapter, 'Coded Character Sets And Encodings in the World' on page 15, introduces many national and international standards of *coded character sets* and *encodings*. I think almost of you can do without reading this chapter, since *LOCALE* technology will enable us to develop international softwares without knowledges on these character sets and encodings. However, knowing about these standards will help you to understand the merit and necessity of LOCALE technology.

The following chapter of 'Characters in Each Country' on page 33 describes the detailed informations for each language. These informations will help people who develop high-quality text processing softwares such as DTP and Web Browsers.

Chapter of 'LOCALE technology' on page 49 describes the most important concept for I18N. Not only concepts but also many important C functions are introduced in this chapter.

A few following chapters of 'Output to Display' on page 59, 'Input from Keyboard' on page 65, 'Internal Processing and File I/O' on page 71, and 'the Internet' on page 79 are important and frequent applications of LOCALE technology. You can get solutions for typical problems on I18N in these chapters.

You may need to develop software using some special libraries or other languages than C/C++. Chapters of 'Libraries and Components' on page 83 and 'Softwares Written in Other than C/C++' on page 87 are written for such purposes.

Next chapter of 'Examples of I18N' on page 89 is a collection of case studies. Both of generic and special technologies will be discussed. You can also contribute writing a section for this chapter.

You may want to study more; The last chapter of 'References' on page 113 is supplied for this purpose. Some of references listed in the chapter are very important.

Chapter 3

Important Concepts for Character Coding Systems

Character coding system is one of the fundamental elements of the software and information processing. Without proper handling of character codes, your software is far from realization of internationalization. Thus the author begins this document with the story on character codes.

In this chapter, basic concepts such as *coded character set* and *encoding* are introduced. These terms will be needed to read this document and other documents on internationalization and character codes including Unicode.

3.1 Basic Terminology

At first I begin this chapter by defining a few very important word.

As many people point out, there is a confusion on terminology, since words are used in various different ways. The author does not want to add a new terminology to a confusing ocean of various terminologies. Otherwise, terminology of RFC 2130 (http://www.faqs.org/rfcs/rfc2130.html) will be adopted in this document, besides one exception of a word 'character set'.

Character Character is an individual unit of which sentence and text consist. Character is an abstract notion.

Glyph Glyph is a specific instance of character. *Character* and *glyph* is a pair of words. Sometimes a character has multiple glyphs (for example, '$' may have one or two vertical bar. Arabic characters have four glyphs for each character. Some of CJK ideograms have many glyphs). Sometimes two or more characters construct one glyph (for example, ligature of 'fi'). For almost cases, text data, which intend to contain not visual information but abstract idea, don't have to have information on glyphs, since difference between

glyphs does not affect the meaning of the text. However, distinction between different glyphs for a single CJK ideogram may be sometimes important for proper noun such as names of persons and places. However, there are no standardized method for plain text to have informations on glyphs so far. This makes plain texts cannot be used for some special fields such as citizen registration system, serious DTP such as newspaper system, and so on.

Encoding Encoding is a rule where characters and texts are expressed in combinations of bits or bytes in order to treat characters in computers. Words of *character coding system*, *character code*, *charset*, and so on are used to express the same meaning. Basically, *encoding* takes care of *characters*, not *glyphs*. There are many official and de-facto standards of encodings such as ASCII, ISO 8859-{1,2,...,15}, ISO 2022-{JP, JP-1, JP-2, KR, CN, CN-EXT, INT-1, INT-2}, EUC-{JP, KR, CN, TW}, Johab, UHC, Shift-JIS, Big5, TIS 620, VISCII, VSCII, so-called 'CodePages', UTF-7, UTF-8, UTF-16LE, UTF-16BE, KOI8-R, and so on so on. To construct an encoding, we have to consider the following concepts. (Encoding = one or more CCS + one CES).

Character Set Character set is a set of characters. This determines a range of characters where the encoding can handle. In contrast to *coded character set*, this is often called as *non-coded character set*.

Coded Character Set (CCS) Coded character set (CCS) is a word defined in RFC 2050 (http://www.faqs.org/rfcs/rfc2050.html) and means a character set where all characters have unique numbers by some method. There are many national and international standards for CCS. Many national standards for CCS adopt the way of coding so that they obey some of international standards such as ISO 646 or ISO 2022. ASCII, BS 4730, JISX 0201 Roman, and so on are examples of ISO-646 variants. All ISO-646 variants, ISO 8859-*, JISX 0208, JISX 0212, KSX 1001, GB 2312, CNS 11643, CCCII, TIS 620, TCVN 5712, and so on are examples of ISO 2022-compliant CCS. VISCII and Big5 are examples of non-ISO 2022-compliant CCS. UCS-2 and UCS-4 (ISO 10646) are also examples of CCS.

Character Encoding Scheme (CES) Character Encoding Scheme is also a word defined in RFC 2050 (http://www.faqs.org/rfcs/rfc2050.html) to call methods to construct an encoding using one or more CCS. This is important when two or more CCS are used to construct an encoding. ISO 2022 is a method to construct an encoding from one or more ISO 2022-compliant CCS. ISO 2022 is very complex system and subsets of ISO 2022 are usually used such as EUC-JP (ASCII and JISX 0208), ISO-2022-KR (ASCII and KSX 1001), and so on. CES is not important for encodings with only one 8bit CCS. UTF series (UTF-8, UTF-16LE, UTF-16BE, and so on) can be regarded as CES whose CCS is Unicode or ISO 10646.

Some other words are usually used related to character codes.

Character code is a widely-used word to mean *encoding*. This is an primitive and crude word to call the way a computer handles characters with assigning numbers. For example, *character code* can call *encoding* and can call *coded character set*. Thus this word can be used only in the case when both of them can be regard in the same category. This word should be avoided in serious discussions. This document will not use this word hereafter.

Codeset is a word to call *encoding* or *character encoding scheme*. [1]

charset is also a well-used word. This word is used very widely, for example, in MIME (like `Content-Type: text/plain, charset=iso8859-1`), in XLFD (X Logical Font Description) font name (CharSetResigtry and CharSetEncoding fields), and so on. Note that *charset* in MIME is *encoding*, while *charset* in XLFD font name is *coded character set*. This is very confusing. In this document, *charset* and *character set* are used in XLFD meaning, since I think *character set* should mean a set of characters, not encoding.

Ken Lunde's "CJKV Information Processing" uses a word **encoding method**. He says that ISO-2022, EUC, Big5, and Shift-JIS are examples of *encoding methods*. It seems that his *encoding method* is *CES* in this document. However, we should notice that Big5 and Shift-JIS are encodings while ISO-2022 and EUC are not. [2]

Character Encoding Model, Unicode Technical Report #17 (`http://www.unicode.org/unicode/reports/tr17/`) (hereafter, *"the Report"*) suggests five-level model.

- ACR: abstract character repertoire
- CCS: Coded Character Set
- CEF: Character Encoding Form
- CES: Character Encoding Scheme
- TES: Transfer Encoding Syntax

TES is also suggested in RFC 2130 (`http://www.faqs.org/rfcs/rfc2130.html`). Some examples of TES are: *base64*, *uuencode*, *BinHex*, *quoted-printable*, *gzip*, and so on. TES means a transform of encoded data which may (or may not) include textual data. Thus, TES is not a part of character encoding. However, TES is important in the Internet data exchange.

When using a computer, we rarely have a chance to face with **ACR**. Though it is true that CJK people have their national standard of ACR (for example, standard for ideograms which can be used for personal names) and some of us may need to handle these ACR with computers (for example, citizen registration system), this is too heavy theme for this document. This is because there are no standardized or encouraged methods to handle these ACR. You may have to build the whole system for such purposes. Good luck!

CCS in *"the Report"* is same as what I wrote in this document. It has concrete examples: ASCII, ISO 8859-{1,2,...,15}, JISX 0201, JISX 0208, JISX 0212, KSX 1001, KSX 1002, GB 2312, Big5, CNS 11643, TIS 620, VISCII, TCVN 5712, UCS2, UCS4, and so on. Some of them are national standards, some are international standards, and others are de-facto standards.

[1] This document used a word *codeset* before Novermber 2000 to call *encoding*. I changed terminology since I could not find a word *codeset* in documents written in English (I adopted this word from a book in Japanese). *encoding* seems more popular.

[2] During I18N programming, we will frequently meet with EUC-JP or EUC-KR, while we well rarely meet with EUC. I think it is not appropriate to stress EUC, a class of encodings, over EUC-JP, EUC-KR, and so on, concrete encodings. It is just like regarding ISO 8859 as a concrete encoding, though ISO 8859 is a class of encodings of ISO 8859-{1,2,...,15}.

CEF and **CES** in *"the Report"* correspond to **CES** in this document. This document will not distinguish these two, since I think there are no inconvenience. An encoding with a significant CEF doesn't have a significant CES (in *"the Report"* meaning), and vice versa. Then why should we have to distinguish these two? The only exception is UTF-16 series. In UTF-16 series, UTF-16 is a CEF and UTF-16BE is a CES. This is the only case where we need distinction between CEF and CES.

Now, **CES** is a concrete concept with concrete examples: ASCII, ISO 8859-{1,2,...,15}, EUC-JP, EUC-KR, ISO 2022-JP, ISO 2022-JP-1, ISO 2022-JP-2, ISO 2022-CN, ISO 2022-CN-EXT, ISO 2022-KR, ISO 2022, VISCII, UTF-7, UTF-8, UTF-16LE, UTF-16BE, and so on. Now they are encodings themselves.

The most important concept in this section is distinction between *coded character set* and *encoding*. *Coded character set* is a component of *encoding*. Text data are described in *encoding*, not *coded character set*.

3.2 Stateless and Stateful

To construct an encoding with two or more CCS, CES has to supply a method to avoid collision between these CCS. There are two ways to do that. One is to make all characters in the all CCS have unique code points. The other is to allow characters from different CCS to have the same code point and to have a code such as escape sequence to switch **SHIFT STATE**, that is, to select one character set.

An encoding with shift states is called **STATEFUL** and one without shift states is called **STATELESS**.

Examples of stateful encodings are: ISO 2022-JP, ISO 2022-KR, ISO 2022-INT-1, ISO 2022-INT-2, and so on.

For example, in ISO 2022-JP, two bytes of `0x24 0x2c` may mean a Japanese Hiragana character 'GA' or two ASCII character of '$' and ',' according to the shift state.

3.3 Multibyte encodings

Encodings are classified into multibyte ones and the others, according to the relationship between number of characters and number of bytes in the encoding.

In non-multibyte encoding, one character is always expressed by one byte. On the other hand, one character may expressed in one or more bytes in multibyte encoding. Note that the number is not fixed even in a single encoding.

Examples of multibyte encodings are: EUC-JP, EUC-KR, ISO 2022-JP, Shift-JIS, Big5, UHC, UTF-8, and so on. Note that all of UTF-* are multibyte.

Examples of non-multibyte encodings are: ISO 8859-1, ISO 8859-2, TIS 620, VISCII, and so on.

Note that even in non-multibyte encoding, number of characters and number of bytes may differ if the encoding is stateful.

Ken Lunde's "CJKV Information Processing" [3] classifies encoding methods into the following three categories:

- modal
- non-modal
- fixed-length

Modal corresponds to *stateful* in this document. Other two are *stateless*, where *non-modal* is *multibyte* and *fixed-length* is *non-multibyte*. However, I think *stateful - stateless* and *multibyte - non-multibyte* are independent concept. [4]

3.4 Number of Bytes, Number of Characters, and Number of Columns

One ASCII character is always expressed by one byte and occupies one column on console or X terminal emulators (fixed font for X). One must not make such an assumption for I18N programming and have to clearly distinguish number of bytes, characters, and columns.

Speaking of relationship between characters and bytes, in multibyte encodings, two or more bytes may be needed to express one character. In stateful encodings, escape sequences are not related to any characters.

Number of columns is not defined in any standards. However, it is usual that CJK ideograms, Japanese Hiragana and Katakana, and Korean Hangul occupy two columns in console or X terminal emulators. Note that 'Full-width forms' in UCS-2 and UCS-4 coded character set will occupy two columns and 'Half-width forms' will occupy one column. Combining characters used for Thai and so on can be regarded as zero-column characters. Though there are no standards, you can use `wcwidth()` and `wcswidth()` for this purpose. See 'Number of Columns' on page 61 for detail.

[3] ISBN 1-56592-224-7, O'Reilly, 1999

[4] though there are no existing encodings which is stateful and non-multibyte.

Chapter 4

Coded Character Sets And Encodings in the World

Here major coded character sets and encodings are introduced. Note that you don't have to know the detail of these character codes if you use LOCALE and `wchar_t` technology.

However, these knowledge will help you to understand why number of bytes, characters, and columns should be counted separately, why `strchr()` and so on should not be used, why you should use LOCALE and `wchar_t` technology instead of hard-code processing of existing character codes, and so on so on.

These varieties of character sets and encodings will tell you about struggles of people in the world to handle their own languages by computers. Especially, CJK people could not help working out various technologies to use plenty of characters within ASCII-based computer systems.

If you are planning to develop a text-processing software beyond the fields which the LOCALE technology covers, you will have to understand the following descriptions very well. These fields include automatic detection of encodings used for the input file (Most of Japanese-capable text viewers such as `jless` and `lv` have this mechanism) and so on.

4.1 ASCII and ISO 646

ASCII is a CCS and also an encoding at the same time. ASCII is 7bit and contains 94 printable characters which are encoded in the region of `0x21-0x7e`.

ISO 646 is the international standard of ASCII. Following 12 characters of

- 0x23 (number),
- 0x24 (dollar),
- 0x40 (at),

- 0x5b (left square bracket),
- 0x5c (backslash),
- 0x5d (right square bracket),
- 0x5e (caret),
- 0x60 (backquote),
- 0x7b (left curly brace),
- 0x7c (vertical line),
- 0x7d (right curly brace), and
- 0x7e (tilde)

are called **IRV** (International Reference Version) and other 82 (94 - 12 = 82) characters are called **BCT** (Basic Code Table). Characters at IRV can be different between countries. Here is a few examples of versions of ISO 646.

- UK version (BS 4730)
- US version (ASCII): 0x23 is pound currency mark, and so on.
- Japanese version (JISX 0201 Roman): 0x5c is yen currency mark, and so on.
- Italian version (UNI 0204-70): 0x7b is 'a' with grave accent, and so on.
- French version (NF Z 62-010): 0x7b is 'e' with acute accent, and so on.

As far as I know, all encodings (besides EBCDIC) in the world are compatible with ISO 646.

Characters in 0x00 - 0x1f, 0x20, and 0x7f are control characters.

Nowadays usage of encodings incompatible with ASCII is not encouraged and thus ISO 646-* (other than US version) should not be used. One of the reason is that when a string is converted into Unicode, the converter doesn't know whether IRVs are converted into characters with same shapes or characters with same codes. Another reason is that source codes are written in ASCII. Source code must be readable anywhere.

4.2 ISO 8859

ISO 8859 is both a series of CCS and a series of encodings. It is an expansion of ASCII using all 8 bits. Additional 96 printable characters encoded in 0xa0 - 0xff are available besides 94 ASCII printable characters.

There are 10 variants of ISO 8859 (in 1997).

ISO-8859-1 Latin alphabet No.1 (1987) characters for western European languages

ISO-8859-2 Latin alphabet No.2 (1987) characters for central European languages

ISO-8859-3 Latin alphabet No.3 (1988)

ISO-8859-4 Latin alphabet No.4 (1988) characters for northern European languages

ISO-8859-5 Latin/Cyrillic alphabet (1988)

ISO-8859-6 Latin/Arabic alphabet (1987)

ISO-8859-7 Latin/Greek alphabet (1987)

ISO-8859-8 Latin/Hebrew alphabet (1988)

ISO-8859-9 Latin alphabet No.5 (1989) same as ISO-8859-1 except for Turkish instead of Icelandic

ISO-8859-10 Latin alphabet No.6 (1993) Adds Inuit (Greenlandic) and Sami (Lappish) letters to ISO-8859-4

ISO-8859-11 Latin/Thai alphabet (2001) same as TIS-620 Thai national standard

ISO-8859-13 Latin alphabet No.7 (1998)

ISO-8859-14 Latin alphabet No.8 (Celtic) (1998)

ISO-8859-15 Latin alphabet No.9 (1999)

ISO-8859-16 Latin alphabet No.10 (2001)

A detailed explanation is found at http://park.kiev.ua/mutliling/ml-docs/iso-8859.html.

4.3 ISO 2022

Using ASCII and ISO 646, we can use 94 characters at most. Using ISO 8859, the number includes to 190 (= 94 + 96). However, we may want to use much more characters. Or, we may want to use some, not one, of these character sets. One of the answer is ISO 2022.

ISO 2022 is an international standard of CES. ISO 2022 determines a few requirement for CCS to be a member of ISO 2022-based encodings. It also defines a very extensive (and complex) rules to combine these CCS into one encoding. Many encodings such as EUC-*, ISO 2022-*, compound text, [1] and so on can be regarded as subsets of ISO 2022. ISO 2022 is so complex that you may be not able to understand this. It is OK; What is important here is the concept of ISO 2022 of building an encoding by switching various (ISO 2022-compliant) coded character sets.

[1] Compound text is a standard for text exchange between X clients.

The sixth edition of ECMA-35 is fully identical with ISO 2022:1994 and you can find the official document at http://www.ecma.ch/ecma1/stand/ECMA-035.HTM.

ISO 2022 has two versions of 7bit and 8bit. At first 8bit version is explained. 7bit version is a subset of 8bit version.

The 8bit code space is divided into four regions,

- 0x00 - 0x1f: C0 (Control Characters 0),
- 0x20 - 0x7f: GL (Graphic Characters Left),
- 0x80 - 0x9f: C1 (Control Characters 1), and
- 0xa0 - 0xff: GR (Graphic Characters Right).

GL and GR is the spaces where (printable) character sets are mapped.

Next, all character sets, for example, ASCII, ISO 646-UK, and JIS X 0208, are classified into following four categories,

- (1) character set with 1-byte 94-character,
- (2) character set with 1-byte 96-character,
- (3) character set with multibyte 94-character, and
- (4) character set with multibyte 96-character.

Characters in character sets with 94-character are mapped into 0x21 - 0x7e. Characters in 96-character set are mapped into 0x20 - 0x7f.

For example, ASCII, ISO 646-UK, and JISX 0201 Katakana are classified into (1), JISX 0208 Japanese Kanji, KSX 1001 Korean, GB 2312-80 Chinese are classified into (3), and ISO 8859-* are classified to (2).

The mechanism to map these character sets into GL and GR is a bit complex. There are four buffers, G0, G1, G2, and G3. A character set is **designated** into one of these buffers and then a buffer is **invoked** into GL or GR.

Control sequences to 'designate' a character set into a buffer are determined as below.

- A sequence to designate a character set with 1-byte 94-character
  - into G0 set is: ESC 0x28 F,
  - into G1 set is: ESC 0x29 F,
  - into G2 set is: ESC 0x2a F, and
  - into G3 set is: ESC 0x2b F.

- A sequence to designate a character set with 1-byte 96-character
  - into G1 set is: ESC 0x2d F,
  - into G2 set is: ESC 0x2e F, and
  - into G3 set is: ESC 0x2f F.
- A sequence to designate a character set with multibyte 94-character
  - into G0 set is: ESC 0x24 0x28 F (exception: 'ESC 0x24 F' for F = 0x40, 0x41, 0x42.),
  - into G1 set is: ESC 0x24 0x29 F,
  - into G2 set is: ESC 0x24 0x2a F, and
  - into G3 set is: ESC 0x24 0x2b F.
- A sequence to designate a character set with multibyte 96-character
  - into G1 set is: ESC 0x24 0x2d F,
  - into G2 set is: ESC 0x24 0x2e F, and
  - into G3 set is: ESC 0x24 0x2f F.

where 'F' is determined for each character set:

- character set with 1-byte 94-character
  - F=0x40 for ISO 646 IRV: 1983
  - F=0x41 for BS 4730 (UK)
  - F=0x42 for ANSI X3.4-1968 (ASCII)
  - F=0x43 for NATS Primary Set for Finland and Sweden
  - F=0x49 for JIS X 0201 Katakana
  - F=0x4a for JIS X 0201 Roman (Latin)
  - and more
- character set with 1-byte 96-character
  - F=0x41 for ISO 8859-1 Latin-1
  - F=0x42 for ISO 8859-2 Latin-2
  - F=0x43 for ISO 8859-3 Latin-3
  - F=0x44 for ISO 8859-4 Latin-4
  - F=0x46 for ISO 8859-7 Latin/Greek
  - F=0x47 for ISO 8859-6 Latin/Arabic
  - F=0x48 for ISO 8859-8 Latin/Hebrew
  - F=0x4c for ISO 8859-5 Latin/Cyrillic

  - and more

- character set with multibyte 94-character

  - F=0x40 for JISX 0208-1978 Japanese
  - F=0x41 for GB 2312-80 Chinese
  - F=0x42 for JISX 0208-1983 Japanese
  - F=0x43 for KSC 5601 Korean
  - F=0x44 for JISX 0212-1990 Japanese
  - F=0x45 for CCITT Extended GB (ISO-IR-165)
  - F=0x46 for CNS 11643-1992 Set 1 (Taiwan)
  - F=0x48 for CNS 11643-1992 Set 2 (Taiwan)
  - F=0x49 for CNS 11643-1992 Set 3 (Taiwan)
  - F=0x4a for CNS 11643-1992 Set 4 (Taiwan)
  - F=0x4b for CNS 11643-1992 Set 5 (Taiwan)
  - F=0x4c for CNS 11643-1992 Set 6 (Taiwan)
  - F=0x4d for CNS 11643-1992 Set 7 (Taiwan)
  - and more

The complete list of these coded character set is found at International Register of Coded Character Sets (http://www.itscj.ipsj.or.jp/ISO-IR/).

Control codes to 'invoke' one of G{0123} into GL or GR is determined as below.

- A control code to invoke G0 into GL is: (L)SO ((Locking) Shift Out)
- A control code to invoke G1 into GL is: (L)SO ((Locking) Shift In)
- A control code to invoke G2 into GL is: LS2 (Locking Shift 2)
- A control code to invoke G3 into GL is: LS3 (Locking Shift 3)
- A control code to invoke one character in G2 into GL is: SS2 (Single Shift 2)
- A control code to invoke one character in G3 into GL is: SS3 (Single Shift 3)
- A control code to invoke G1 into GR is: LS1R (Locking Shift 1 Right)
- A control code to invoke G2 into GR is: LS2R (Locking Shift 2 Right)
- A control code to invoke G3 into GR is: LS3R (Locking Shift 3 Right)

[2]

Note that a code in a character set invoked into GR is or-ed with 0x80.

ISO 2022 also determines **announcer** code. For example, 'ESC 0x20 0x41' means 'Only G0 buffer is used. G0 is already invoked into GL'. This simplify the coding system. Even this announcer can be omitted if people who exchange data agree.

7bit version of ISO 2022 is a subset of 8bit version. It does not use C1 and GR.

Explanation on C0 and C1 is omitted here.

4.3.1 EUC (Extended Unix Code)

EUC is a CES which is a subset of 8bit version of ISO 2022 except for the usage of SS2 and SS3 code. Though these codes are used to invoke G2 and G3 into GL in ISO 2022, they are invoked into GR in EUC. **EUC-JP**, **EUC-KR**, **EUC-CN**, and **EUC-TW** are widely used encodings which use EUC as CES.

EUC is stateless.

EUC can contain 4 CCS by using G0, G1, G2, and G3. Though there is no requirement that ASCII is designated to G0, I don't know any EUC codeset in which ASCII is not designated to G0.

For EUC with G0-ASCII, all codes other than ASCII are encoded in 0x80 - 0xff and this is upward compatible to ASCII.

Expressions for characters in G0, G1, G2, and G3 character sets are described below in binary:

- G0: 0???????
- G1: 1??????? [1??????? [...]]
- G2: SS2 1??????? [1??????? [...]]
- G3: SS3 1??????? [1??????? [...]]

where SS2 is 0x8e and SS3 is 0x8f.

4.3.2 ISO 2022-compliant Character Sets

There are many national and international standards of coded character sets (CCS). Some of them are ISO 2022-compliant and can be used in ISO 2022 encoding.

ISO 2022-compliant CCS are classified into one of them:

- 94 characters

[2] WHAT IS THE VALUE OF THESE CONTROL CODES?

- 96 characters
- 94x94x94x... characters

The most famous 94 character set is US-ASCII. Also, all ISO 646 variants are ISO 2022-compliant 94 character sets.

All ISO 8859-* character sets are ISO 2022-compliant 96 character sets.

There are many 94x94 character sets. All of them are related to CJK ideograms.

JISX 0208 (aka JIS C 6226) National standard of Japan. 1978 version contains 6802 characters including Kanji (ideogram), Hiragana, Katakana, Latin, Greek, Cyrillic, numeric, and other symbols. The current (1997) version contains 7102 characters.

JISX 0212 National standard of Japan. 6067 characters (almost of them are Kanji). This character set is intended to be used in addition to JISX 0208.

JISX 0213 Japanese national standard. Released in 2000. This includes JISX 0208 characters and additional thousands of characters. Thus, this is intended to be an extension and a replacement of JISX 0208. This has two 94x94 character sets, one of them incluCdes JISX 0208 plus about 2000 characters and the another includes about 2400 characters. Exactly speaking, JISX 0213 is not a simple superset of JISX 0208 because a few tens of Kanji variants which is unified and share the same code points in JISX 0208 are dis-unified and have separate code points in JISX 0213. Share many characters with JISX 0212.

KSX 1001 (aka KSC 5601) National standard of South Korea. 8224 characters including 2350 Hangul, Hanja (ideogram), Hiragana, Katakana, Latin, Greek, Cyrillic, and other symbils. Hanja are ordered in reading and Hanja with multiple readings are coded multiple times.

KSX 1002 National standard of South Korea. 7659 characters including Hangul and Hanja. Intended to be used in addition to KSX 1001.

KPS 9566 National standard of North Korea. Similar to KSX 1001.

GB 2312 National standard of China. 7445 characters including 6763 Hanzi (ideogram), Latin, Greek, Cyrillic, Hiragana, Katakana, and other symbols.

GB 7589 (aka GB2) National standard of China. 7237 Hanzi. Intended to be used in addition to GB 2312.

GB 7590 (aka GB4) National standard of China. 7039 Hanzi. Intended to be used in addition to GB 2312 and GB 7589.

GB 12345 (aka GB/T 12345, GB1 or GBF) National standard of China. 7583 characters. Traditional characters version which correspond to GB 2312 simplified characters.

GB 13131 (aka GB3) National standard of China. Traditional characters version which correspond to GB 7589 simplified characters.

GB 13132 (aka GB5) National standard of China. Traditional characters version which correspond to GB 7590 simplified characters.

CNS 11643 National standard of Taiwan. Has 7 plains. Plain 1 and 2 includes all characters included in Big5. Plain 1 includes 6085 characters including Hanzi (ideogram), Latin, Greek, and other symbols. Plain 2 includes 7650. Number of character for plain 3 is 6184, plain 4 is 7298, plain 5 is 8603, plain 6 is 6388, and plain 7 is 6539.

There is a 94x94x94 character set. This is **CCCII**. This is national standard of Taiwan. Now 73400 characters are included. (The number is increasing.)

Non-ISO 2022-compliant character sets are introduced later in 'Other Character Sets and Encodings' on page 30.

4.3.3 ISO 2022-compliant Encodings

There are many ISO 2022-compliant encodings which are subsets of ISO 2022.

Compound Text This is used for X clients to communicate each other, for example, copy-paste.

EUC-JP An EUC encoding with ASCII, JISX 0208, JISX 0201 Kana, and JISX 0212 coded character sets. There are many systems which does not support JISX 0201 Kana and JISX 0212. Widely used in Japan for POSIX systems.

EUC-KR An EUC encoding with ASCII and KSX 1001.

CN-GB (aka EUC-CN) An EUC encoding with ASCII and GB 2312. The most popular encoding in R. P. China. This encoding is sometimes referred as simply 'GB'.

EUC-TW An extended EUC encoding with ASCII, CNS 11643 plain 1, and other (2-7) plains of CNS 11643.

ISO 2022-JP Described in. RFC 1468 (http://www.faqs.org/rfcs/rfc1468.html).

***** Not written yet *****

ISO 2022-JP-1 (upward compatible to ISO 2022-JP) Described in RFC 2237 (http://www.faqs.org/rfcs/rfc2237.html).

***** Not written yet *****

ISO 2022-JP-2 (upward compatible to ISO 2022-JP-1) Described in RFC 1554 (http://www.faqs.org/rfcs/rfc1554.html).

***** Not written yet *****

ISO 2022-KR aka Wansung. Described in RFC 1557 (http://www.faqs.org/rfcs/rfc1557.html).

***** Not written yet *****

ISO 2022-CN Described in RFC RFC 1922 (http://www.faqs.org/rfcs/rfc1922.html).

***** Not written yet *****

Non-ISO 2022-compliant encodings are introduced later in 'Other Character Sets and Encodings' on page 30.

4.4 ISO 10646 and Unicode

ISO 10646 and Unicode are an another standard so that we can develop international softwares easily. The special features of this new standard are:

- A united single CCS which intends to include all characters in the world. (ISO 2022 consists of multiple CCS.)
- The character set intends to cover all conventional (or *legacy*) CCS in the world. [3]
- Compatibility with ASCII and ISO 8859-1 is considered.
- Chinese, Japanese, and Korean ideograms are united. This comes from a limitation of Unicode. This is not a merit.

ISO 10646 is an official international standard. Unicode is developed by Unicode Consortium (http://www.unicode.org). These two are almost identical. Indeed, these two are exactly identical at code points which are available in both two standards. Unicode is sometimes updated and the newest version is 3.0.1.

4.4.1 UCS as a Coded Character Set

ISO 10646 defines two CCS (coded character sets), **UCS-2** and **UCS-4**. UCS-2 is a subset of UCS-4.

UCS-4 is a 31bit CCS. These 31 bits are divided into 7, 8, 8, and 8 bits and each of them has special term.

- The top 7 bits are called **Group**.
- Next 8 bits are called **Plane**.
- Next 8 bits are **Row**.
- The smallest 8 bits are **Cell**.

[3] This is obviously not true for CNS 11643 because CNS 11643 contains 48711 characters while Unicode 3.0.1 contains 49194 characters, only 483 excess than CNS 11643.

The first plane (Group = 0, Plane = 0) is called **BMP** (Basic Multilingual Plane) and UCS-2 is same to BMP. Thus, UCS-2 is a 16bit CCS.

Code points in UCS are often expressed as **u+????**, where `????` is hexadecimal expression of the code point.

Characters in range of u+0021 - u+007e are same to ASCII and characters in range of u+0xa0 - u+0xff are same to ISO 8859-1. Thus it is very easy to convert between ASCII or ISO 8859-1 and UCS.

Unicode (version 3.0.1) uses a 20bit subset of UCS-4 as a CCS. [4]

The unique feature of these CCS compared with other CCS is *open repertoire*. They are developing even after they are released. Characters will be added in future. However, already coded characters will not changed. Unicode version 3.0.1 includes 49194 distinct coded characters.

4.4.2 UTF as Character Encoding Schemes

A few CES are used to construct encodings which use UCS as a CCS. They are **UTF-7**, **UTF-8**, **UTF-16**, **UTF-16LE**, and **UTF-16BE**. UTF means Unicode (or UCS) Transformation Format. Since these CES always take UCS as the only CCS, they are also names for encodings. [5]

UTF-8

UTF-8 is an encoding whose CCS is UCS-4. UTF-8 is designed to be upward-compatible to ASCII. UTF-8 is multibyte and number of bytes needed to express one character is from 1 to 6.

Conversion from UCS-4 to UTF-8 is performed using a simple conversion rule.

```
UCS-4 (binary)                      UTF-8 (binary)
00000000 00000000 00000000 0???????  0???????
00000000 00000000 00000??? ????????  110????? 10??????
00000000 00000000 ???????? ????????  1110???? 10?????? 10??????
00000000 000????? ???????? ????????  11110??? 10?????? 10?????? 10??????
000000?? ???????? ???????? ????????  111110?? 10?????? 10?????? 10?????? 10??
0??????? ???????? ???????? ????????  1111110? 10?????? 10?????? 10?????? 10??
```

Note the shortest one will be used though longer representation can express smaller UCS values.

UTF-8 seems to be one of the major candidates for standard codesets in the future. For example, Linux console and xterm supports UTF-8. Debian package of `locales` (version 2.1.97-1) contains `ko_KR.UTF-8` locale. I think the number of UTF-8 locale will increase.

[4] Exactly speaking, u+000000 - u+10ffff.

[5] Compare UTF and EUC. There are a few variants of EUC whose CCS are different (EUC-JP, EUC-KR, and so on). This is why we cannot call EUC as an encoding. In other words, calling of 'EUC' cannot specify an encoding. On the other hands, 'UTF-8' is the name for a specific concrete encoding.

UTF-16

UTF-16 is an encoding whose CCS is 20bit Unicode.

Characters in BMP are expressed using 16bit value of code point in Unicode CCS. There are two ways to express 16bit value in 8bit stream. Some of you may heard a word *endian*. *Big endian* means an arrangement of octets which are part of a datum with many bits from most significant octet to least significant one. *Little endian* is opposite. For example, 16bit value of `0x1234` is expressed as `0x12 0x34` in big endian and `0x34 0x12` in little endian.

UTF-16 supports both endians. Thus, Unicode character of `u+1234` can be expressed either in `0x12 0x34` or `0x34 0x12`. Instead, the UTF-16 texts have to have a **BOM (Byte Order Mark)** at first of them. The Unicode character `u+feff` zero width no-break space is called BOM when it is used to indicate the byte order or endian of texts. The mechanism is easy: in big endian, `u+feff` will be `0xfe 0xff` while it will be `0xff 0xfe` in little endian. Thus you can understand the endian of the text by reading the first two bytes. [6]

Characters not included in BMP are expressed using **surrogate pair**. Code points of `u+d800` - `u+dfff` are reserved for this purpose. At first, 20 bits of Unicode code point are divided into two sets of 10 bits. The significant 10 bits are mapped to 10bit space of `u+d800` - `u+dbff`. The smaller 10 bits are mapped to 10bit space of `u+dc00` - `u+dfff`. Thus UTF-16 can express 20bit Unicode characters.

UTF-16BE and UTF-16LE

UTF-16BE and UTF-16LE are variants of UTF-16 which are limited to big and little endians, respectively.

UTF-7

UTF-7 is designed so that Unicode can be communicated using 7bit communication path.

***** Not written yet *****

UCS-2 and UCS-4 as encodings

Though I introduced UCS-2 and UCS-4 are CCS, they can be encodings.

In UCS-2 encoding, Each UCS-2 character is expressed in two bytes. In UCS-4 encoding, Each UCS-4 character is expressed in four bytes.

[6] I heard that BOM is mere a suggestion by a vendor. Read Markus Kuhn's UTF-8 and Unicode FAQ for Unix/Linux (http://www.cl.cam.ac.uk/~mgk25/unicode.html) for detail.

4.4.3 Problems on Unicode

All standards are not free from politics and compromise. Though a concept of united single CCS for all characters in the world is very nice, Unicode had to consider compatibility with preceding international and local standards. And more, unlike the ideal concept, Unicode people considered efficiency too much. IMHO, surrogate pair is a mess caused by lack of 16bit code space. I will introduce a few problems on Unicode.

Han Unification

This is the point on which Unicode is criticized most strongly among many Japanese people.

A region of 0x4e00 - 0x9fff in UCS-2 is used for Eastern-Asian ideographs (Japanese Kanji, Chinese Hanzi, and Korean Hanja). There are similar characters in these four character sets. (There are two sets of Chinese characters, simplified Chinese used in P. R. China and traditional Chinese used in Taiwan). To reduce the number of these ideograms to be encoded (the region for these characters can contain only 20992 characters while only Taiwan CNS 11643 standard contains 48711 characters), these similar characters are assumed to be the same. This is Han Unification.

However these characters are not exactly the same. If fonts for these characters are made from Chinese one, Japanese people will regard them wrong characters, though they may be able to read. Unicode people think these united characters are the same character with different glyphs.

An example of Han Unification is available at U+9AA8 (http://www.unicode.org/cgi-bin/GetUnihanData.pl?codepoint=9AA8). This is a Kanji character for 'bone'. U+8FCE (http://www.unicode.org/cgi-bin/GetUnihanData.pl?codepoint=8FCE) is an another example of a Kanji character for 'welcome'. The part from left side to bottom side is 'run' radical. 'Run' radical is used for many Kanjis and all of them have the same problem. U+76F4 (http://www.unicode.org/cgi-bin/GetUnihanData.pl?codepoint=76F4) is an another example of a Kanji character for 'straight'. I, a native Japanese speaker, cannot recognize Chiense version at all.

Unicode font vendors will hesitate to choose fonts for these characters, simplified Chinese character, traditional Chinese one, Japanese one, or Korean one. One method is to supply four fonts of simplified Chinese version, traditional Chinese version, Japanese version, and Korean version. Commercial OS vendor can release localized version of their OS — for example, Japanese version of MS Windows can include Japanese version of Unicode font (this is what they are exactly doing). However, how should XFree86 or Debian do? I don't know... [7] [8]

[7] XFree86 4.0 includes Japanese and Korean versions of ISO 10646-1 fonts.

[8] I heard that Chinese and Korean people don't mind the glyph of these characters. If this is always true, Japanese glyphs should be the default glyphs for these problematic characters for international systems such as Debian.

Cross Mapping Tables

Unicode intents to be a superset of all major encodings in the world, such as ISO-8859-*, EUC-*, KOI8-*, and so on. The aim of this is to keep round-trip compatibility and to enable smooth migration from other encodings to Unicode.

Only providing a superset is not sufficient. Reliable cross mapping tables between Unicode and other encodings are needed. They are provided by Unicode Consortium (http://www.unicode.org/Public/MAPPINGS/).

However, tables for East Asian encodings are not provided. They were provided but now are obsolete (http://www.unicode.org/Public/MAPPINGS/OBSOLETE/EASTASIA/).

You may want to use these mapping tables even though they are obsolete, because there are no other mapping tables available. However, you will find a severe problem for these tables. There are multiple different mapping tables for Japanese encodings which include JIS X 0208 character set. Thus, one same character in JIS X 0208 will be mapped into different Unicode characters according to these mapping tables. For example, Microsoft and Sun use different table, which results in Java on MS Windows sometimes break Japanese characters.

Though we Open Source people should respect interoperativity, we cannot achieve sufficient interoperativity because of this problem. All what we can achieve is interoperativity between Open Source softwares.

GNU libc uses JIS/JIS0208.TXT (http://www.unicode.org/Public/MAPPINGS/OBSOLETE/EASTASIA/JIS/JIS0208.TXT) with a small modification. The modification is that

- original JIS0208.TXT: 0x815F 0x2140 0x005C # REVERSE SOLIDUS
- modified: 0x815F 0x2140 0xFF3C # FULLWIDTH REVERSE SOLIDUS

The reason of this modification is that JIS X 0208 character set is almost always used with combination with ASCII in form of EUC-JP and so on. ASCII 0x5c, not JIS X 0208 0x2140, should be mapped into U+005C. This modified table is found at `/usr/share/i18n/charmaps/EUC-JP.gz` in Debian system. Of course this mapping table is NOT authorized nor reliable.

I hope Unicode Consortium to release an authorized reliable unique mapping table between Unicode and JIS X 0208. You can read the detail of this problem (http://www.debian.or.jp/~kubota/unicode-symbols.html).

Combining Characters

Unicode has a way to synthesize a accented character by combining an accent symbol and a base character. For example, combining 'a' and '~' makes 'a' with tilde. More than two accent symbol can be added to a base character.

Languages such as Thai need combining characters. Combining characters are the only method to express characters in these languages.

However, a few problems arises.

Duplicate Encoding There are multiple ways to express the same character. For example, u with umlaut can be expressed as `u+00fc` and also as `u+0075` + `U+0308`. How can we implement 'grep' and so on?

Open Repertoire Number of expressible characters grows unlimitedly. Non-existing characters can be expressed.

Surrogate Pair

The first version of Unicode had only 16bit code space, though 16bit is obviously insufficient to contain all characters in the world. [9] Thus surrogate pair is introduced in Unicode 2.0, to expand the number of characters, with keeping compatibility with former 16bit Unicode.

However, surrogate pair breaks the principle that all characters are expressed with the same width of bits. This makes Unicode programming more difficult.

Fortunately, Debian and other UNIX-like systems will use UTF-8 (not UTF-16) as a usual encoding for UCS. Thus, we don't need to handle UTF-16 and surrogate pair very often.

ISO 646-* Problem

You will need a codeset converter between your local encodings (for example, ISO 8859-* or ISO 2022-*) and Unicode. For example, Shift-JIS encoding [10] consists from JISX 0201 Roman (Japanese version of ISO 646), not ASCII, which encodes yen currency mark at `0x5c` where backslash is encoded in ASCII.

Then which should your converter convert `0x5c` in Shift-JIS into in Unicode, `u+005c` (backslash) or `u+00a5` (yen currency mark)? You may say yen currency mark is the right solution. However, backslash (and then yen mark) is widely used for escape character. For example, 'new line' is expressed as 'backslash - `n`' in C string literal and Japanese people use 'yen currency mark - `n`'. You may say that program sources must written in ASCII and the wrong point is that you tried to convert program source. However, there are many source codes and so on written in Shift-JIS encoding.

Now Windows comes to support Unicode and the font at `u+005c` for Japanese version of Windows is yen currency mark. As you know, backslash (yen currency mark in Japan) is vitally important for Windows, because it is used to separate directory names. Fortunately,

[9] There are a few projects such as Mojikyo (`http://www.mojikyo.gr.jp/`) (about 90000 characters), TRON project (`http://www.tron.org/index-e.html`) (about 130000 characters), and so on to develop a CCS which contains sufficient characters for professional usage in CJK world.

[10] The standard encoding for Macintosh and MS Windows.

EUC-JP, which is widely used for UNIX in Japan, includes ASCII, not Japanese version of ISO 646. So this is not problem because it is clear `0x5c` is backslash.

Thus all local codesets should not use character sets incompatible to ASCII, such as ISO 646-*.

Problems and Solutions for Unicode and User/Vendor Defined Characters (`http://www.opengroup.or.jp/jvc/cde/ucs-conv-e.html`) discusses on this problem.

4.5 Other Character Sets and Encodings

Besides ISO 2022-compliant coded character sets and encodings described in 'ISO 2022-compliant Character Sets' on page 21 and 'ISO 2022-compliant Encodings' on page 23, there are many popular encodings which cannot be classified into an international standard (i.e., not ISO 2022-compliant nor Unicode). Internationalized softwares should support these encodings (again, you don't need to be aware of encodings if you use LOCALE and `wchar_t` technology). Some organizations are developing systems which go father than limitations of the current international standards, though these systems may be not diffused very much so far.

4.5.1 Big5

Big5 is a de-facto standard encoding for Taiwan (1984) and is upward-compatible with ASCII. It is also a CCS.

In Big5, `0x21` - `0x7e` means ASCII characters. `0xa1` - `0xfe` makes a pair with the following byte (`0x40` - `0x7e` and `0xa1` - `0xfe`) and means an ideogram and so on (13461 characters).

Though Taiwan has ISO 2022-compliant new standard CNS 11643, Big5 seems to be more popular than CNS 11643. (CNS 11643 is a CCS and there are a few ISO 2022-derived encodings which include CNS 11643.)

4.5.2 UHC

UHC is an encoding which is an upward-compatible with **EUC-KR**. Two-byte characters (the first byte: `0x81` - `0xfe`; the second byte: `0x41` - `0x5a`, `0x61` - `0x7a`, and `0x81` - `0xfe`) include KSX 1001 and other Hangul so that UHC can express all 11172 Hangul.

4.5.3 Johab

Johab is an encoding whose character set is identical with **UHC**, i.e., ASCII, KSX 1001, and all other Hangul character. Johab means combination in Korean. In Johab, code point of a Hangul can be calculated from combination of Hangul parts (Jamo).

4.5.4 HZ, aka HZ-GB-2312

HZ is an encoding described in RFC 1842 (http://www.faqs.org/rfcs/rfc1842.html). CCS (Coded character sets) of HZ is ASCII and GB2312. This is 7bit encoding.

Note that HZ is *not* upward-compatible with ASCII, since '~{' means GB2312 mode, '~}' means ASCII mode, and '~~' means ASCII '~'.

4.5.5 GBK

GBK is an encoding which is upward-compatible to CN-GB. GBK covers ASCII, GB2312, other Unicode 1.0 ideograms, and a bit more. The range of two-byte characters in GBK is: `0x81` - `0xfe` for the first byte and `0x40` - `0x7e` and `0x80` - `0xfe` for the second byte. 21886 code points out of 23940 in two-byte region are defined.

GBK is one of popular encodings in R. P. China.

4.5.6 GB18030

GB 18030 is an encoding which is upward-compatible to GBK and CN-GB. It is an recent national standard (released on 17 March 2000) of China. It adds four-byte characters to GBK. Its range is: `0x81` - `0xfe` for the first byte, `0x30` - `0x39` for the second byte, `0x81` - `0xfe` for the third byte, and `0x30` - `0x39` for the forth byte.

It includes all characters of Unicode 3.0's Unihan Extension A. And more, GB 18030 supplies code space for all used and unused code points of Unicode's plane 0 (BMP) and 16 additional planes.

A detailed explanation on GB18030 (ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/pdf/GB18030_Summary.pdf) is available.

4.5.7 GCCS

GCCS is a standard of coded character set by Hong Kong (HKSAR: Hong Kong Special Administrative Region). It includes 3049 characters. It is an abbreviation of Government Common Character Set. It is defined as an *additional character set for Big5*. Characters in GCCS are coded in User-Defined Area (just like Private Use Area for UCS) in Big5.

4.5.8 HKSCS

HKSCS is an expansion and amendment of GCCS. It includes 4702 characters. It means Hong Kong Supplementary Character Set.

In addition to a usage in User-Defined Area in Big5, HKSCS defines a usage in Private Use Area in Unicode.

4.5.9 Shift-JIS

Shift-JIS is one of popular encodings in Japan. Its CCS are JISX 0201 Roman, JISX 0201 Kana, and JISX 0208.

JISX 0201 Roman is Japanese version of ISO 646. It defines yen currency mark for `0x5c`, where ASCII has backslash. `0xa1` - `0xdf` is one-byte character and is JISX 0201 Kana. Two-byte character (the first byte: `0x81` - `0x9f` and `0xe0` - `0xef`; the second byte: `0x40` - `0x7e` and `0x80` - `0xfc`) is JISX 0208.

Japanese version of MS DOS, MS Windows and Macintosh use this encoding, though this encoding is not often used in POSIX systems.

4.5.10 VISCII

Vietnamese language uses 186 characters (Latin alphabets with accents) and other symbols. It is a bit more than the limit of ISO 8859-like encoding.

VISCII is a standard for Vietnamese. It is upward-compatible with ASCII. It is 8bit and stateless, like ISO 8859 series. However, it uses code points of not only `0x21` - `0x7e` and `0xa0` - `0xff` but also `0x02`, `0x05`, `0x06`, `0x14`, `0x19`, `0x1e`, and `0x80` - `0x9f`. This makes VISCII not-ISO 2022-compliant.

Vietnam has a new, ISO 2022-compliant character set **TCVN 5712 VN2** (aka **VSCII**). In TCVN 5712 VN2, accented characters are expressed as a combined character. Note that some of accented characters have their own code points.

4.5.11 TRON

TRON (`http://www.tron.org/index-e.html`) is a project to develop a new operating system, founded as a collaboration of industries and academics in Japan since 1984.

The most diffused version of TRON operating system families is ITRON, a real-time OS for embedded systems. However, our interest is not on ITRON now. TRON determines a TRON encoding.

TRON's encoding is stateful. Each state is assigned to each language. It has already defined about 130000 characters (January 2000).

4.5.12 Mojikyo

Mojikyo (`http://www.mojikyo.gr.jp/`) is a project to develop an environment by which a user can use many characters in the world. Mojikyo project has released an application software for MS Windows to display and input about 90000 characters. You can download the software and TrueType, TeX, and CID fonts, though they are not DFSG-free.

Chapter 5

Characters in Each Country

This chapter describes a specific information for each language. If you are developing a serious DTP software or planning to support detailed I18N, this chapter may help you. Contributions from people speaking each language are welcome. If you are to write a section on your language, please include these points:

1. kinds and number of characters used in the language,
2. explanation on coded character set(s) which is (are) standardized,
3. explanation on encoding(s) which is (are) standardized,
4. usage and popularity for each encoding,
5. de-facto standard, if any, on how many columns characters occupy,
6. writing direction and combined characters,
7. how to layout characters (word wrapping and so on),
8. widely used value for `LANG` environmental variable,
9. the way to input characters from keyboard and whether you want to input yes/no (and so on) in your language or in English,
10. a set of information needed for beautiful displaying, for example, where to break a line, hyphenation, word wrapping, and so on, and
11. other topics.

Writers whose languages are written in different direction from European languages or needs a combined characters (I heard that is used in Thai) are encouraged to explain how to treat such languages.

5.1 Japanese language / used in Japan

This section is the text written by Tomohiro KUBOTA <kubota@debian.org> (no more reachable).

Japanese is the only official language used in Japan. People in Okinawa islands and Ainu ethnic group in Hokkaido region have each language, though they are used among few number of people and they don't have own letters.

Japan is the only region where Japanese language is widely used.

5.1.1 Characters used in Japanese

There are three kinds of characters used in Japan, Hiragana, Katakana, and Kanji. Arabic numerical characters (same as European languages) are widely used in Japanese, though we have Kanji numerical characters. Though Latin alphabets are not a part of Japanese characters, they are widely used for proper nouns for companies and so on.

Hiragana and Katakana are phonogram derived from Kanji. Hiragana and Katakana characters have one-to-one correspondence each other like upper and lower case of Latin alphabets. However, `toupper()` and `tolower()` should not convert Hiragana and Katakana each other. Hiragana contains about 100 characters and of course Katakana does. (FYI: about 50 regular characters, 20 characters with voiced consonant symbol, 5 characters with semi-voiced consonant symbol, and 9 small characters.)

Kanji is ideogram imported from China roughly about 1 - 2 thousands years ago. Nobody knows the whole number of Kanji and almost all of adult Japanese people know several thousands of Kanji characters. Though the origin of Kanji is Chinese character, shapes are changed from original ancient Chinese Kanji. Almost all Kanji have several ways to read, according to the word the Kanji is contained.

5.1.2 Character Sets

JIS (Japan Industrial Standards) is an organization responsible for coded character sets (CCS) and encodings used in Japan. The major coded character sets in Japan are:

- JIS X 0201-1976 Roman characters (Almost same to ASCII but 0x5c is Yen mark instead of backslash and 0x7e is upper bar instead of tilde)
- JIS X 0201-1976 Kana (about 60 KATAKANA characters),
- JIS X 0208-1997 1st and 2nd levels (about 7000 characters including symbols, numeric characters, Latin, Cyrillic and Greek alphabets, Japanese HIRAGANA, KATAKANA, and KANJI),
- JIS X 0212 (about 6000 characters including KANJI, which are not included in JIS X 0208), and

- JIS X 0213:2000 (aka JIS 3rd and 4th levels).

JIS X 0201 Roman is the Japanese version of ISO 646. Though JIS X 0201 is included in SHIFT-JIS encoding (explained later) and widely used for Windows/Macintosh, usage of this is not encouraged in UNIX.

JIS X 0201 Kana defines about 60 KATAKANA characters. This is widely used by old 8bit computers. In deed, SHIFT-JIS encoding was designed to be upward-compatible with 8-bit encoding of JISX 0201 Roman and JISX 0201 Kana. Note this CCS is not included in ISO 2022-JP encoding which is used for e-mail and so on.

JIS X 0212 is not widely used, probably because it cannot be included in SHIFT-JIS, the standard encoding for Japanese version of Windows and Macintosh. And more, this CCS may be obsolete when JIS X 0213 will be popular, since JIS X 0213 has many characters which are included in JIS X 0212. However, the advantage of JIS X 0212 over JIS X 0213 is that all characters in JIS X 0212 are included in the current Unicode (version 3.0.1) while not all characters in JIS X 0213 are.

JIS X 0208 (aka JIS C 6226) is the main standard for Japanese characters. Strictly speaking, it was originally defined in 1978 and revised on 1983, 1990, and 1997. Though 1997 version has 77 more characters than original 1976 version and shape of more than 200 characters are changed, almost softwares don't have to care about the difference between them. However, be careful of that ISO-2022-JP encoding (explained below) contains both JIS X 0208-1978 and JIS X 0208-1983. 1978 version is called 'old JIS' and later is called 'new JIS'. Characters in JIS X 0208 are divided into two levels, 1st and 2nd. Old 8bit computers rarely implemented the 2nd level.

Usage of numeric characters and Latin alphabets in JIS X 0208 is not encouraged because these characters are also included in ASCII and JIS X 0201 Roman, either of which is included in all encodings. When converting into Unicode, these characters are mapped into 'fullwidth forms'.

All of these coded character sets (except for JIS X 0213) are included in Unicode 3.0.1. A part of JIS X 0213 characters are not included in Unicode 3.0.1.

There are a few different tables for conversion between non-letter characters in JIS X 0208 and Unicode. This is a problem because this may deny 'round-trip compatiblilty'. Problems and Solutions for Unicode and User/Vendor Defined Characters (http://www.opengroup.or.jp/jvc/cde/ucs-conv-e.html) discusses this problem in detail.

5.1.3 Encodings

There are three popular encodings widely used in Japan.

- ISO-2022-JP (aka JIS code or JUNET code)
  - stateful
  - subset of 7bit version of ISO-2022, where ASCII, JIS X 0201-1976 Roman, JIS X 0208-1978, and JIS X 0208-1983 are supported.

  - 7bit, which means the most significant bit (MSB) of each byte is always zero.
  - used for e-mail and net-news and preferred for HTML.
  - Determined in RFC 1468.

- EUC-JP (Japanese version of Extended UNIX Code)
  - stateless
  - an implementation of EUC where G0, G1, G2, and G3 are ASCII, JIS X 0208, JIS X 0201 Kana, and JIS X 0212 respectively. There are many implementation which cannot use JIS X 0201 Kana and JIS X 0212.
  - 8bit
  - preferred encoding for UNIX. For example, almost all Japanese message catalogs for gettext is written in EUC-JP.
  - Japanese code is mapped in `0xa0 - 0xff`. This is important for programmer because one doesn't need to care there are fake '\' or '/' (which can be treated in a special way in various context) in the Japanese code.

- SHIFT-JIS (aka Microsoft Kanji Code)
  - stateless
  - NOT subset of ISO-2022
  - 8bit
  - JIS X 0201 Roman, JIS X 0201 Kana, and JIS X 0208 can be expressed, but JIS X 0212 cannot.
  - The standard encoding for Windows/Macintosh. This makes SHIFT-JIS the most popular encoding in Japan. Though MS is thinking about transition to UNICODE, it is suspicious that it can be done successfully.

ISO-2022-JP is a subset of 7bit version of ISO 2022, where only G0 is used and G0 is assumed to be invoked into GL. Character sets included in ISO-2022-JP are:

- ASCII (ESC 0x28 0x42),
- JIS X 0201-1976 Roman (ESC 0x28 0x4a),
- JIS X 0208-1978 (old JIS) (ESC 0x24 0x40), and
- JIS X 0208-1983 (new JIS) (ESC 0x24 0x42).

Note that JIS X 0208-1978 and JIS X 0208-1983 are almost identical and ASCII and JIS X 0201-1976 Roman are also almost identical. A line (stream of bytes between 'newline' control code) must start by ASCII status and to end by ASCII status. See 'ISO 2022' on page 17 for detail.

ISO-2022-JP-2 (RFC 1554) is a subset of 7bit version of ISO 2022 and superset of ISO-2022-JP. Difference between ISO-2022-JP and ISO-2022-JP-2 is that ISO-2022-JP-2 has more coded character sets than ISO-2022-JP. Character sets included in ISO-2022-JP-2 are:

- ASCII (ESC 0x28 0x42)
- JIS X 0201-1976 Roman (ESC 0x28 0x4a),
- JIS X 0208-1978 (old JIS) (ESC 0x24 0x40),
- JIS X 0208-1983 (new JIS) (ESC 0x24 0x42),
- GB2312-80 (simplified Chinese) (ESC 0x24 0x41),
- KS C 5601 (Korean) (ESC 0x24 0x28 0x43),
- JIS X 0212-1990 (ESC 0x24 0x28 0x44),
- ISO 8859-1 (Latin-1) (ESC 0x2e 0x41), and
- ISO 8859-7 (Greek) (ESC 0x2e 0x46).

Though JIS X 0212-1990 may sometimes be used, ISO-2022-JP-2 is rarely used.

ISO-2022-INT-1 is a superset of ISO-2022-JP-2 which has CNS 11643-1986-1 and CNS 11643-1986-2 (traditional Chinese).

EUC-JP is a version of EUC, where G0 is ASCII, G1 is JIS X 0208, G2 is JIS X 0201 Kana, and G3 is JIS X 0212. G2 and G3 are sometimes not implemented. This is the most popular encoding for Linux/Unix. See 'EUC (Extended Unix Code)' on page 21 for detail.

SHIFT-JIS is designed to be a superset of encodings for old 8bit computers which includes JIS X 0201 Roman and JIS X 0201 Kana. `0x20` - `0x7f` is JIS X 0201 Roman and `0xa0` - `0xdf` is JIS X 0201 Kana. `0x80` - `0x9f` and `0xe0` - `0xff` is the first byte of doublebyte characters. The second byte is `0x40` - `0x7e` and `0x80` - `0xfc`. This code space is used for JIS X 0208.

UNICODE is not popular in Japan at all, probably because conversion from these codes into Unicode is a bit difficult. However MS Windows uses Unicode in a limited field, for example, internal code for file names. I guess more and more softwares will come to support Unicode in the future.

You can convert files written in these encodings one another using `nkf` or `kcc` package. Using options `-j`, `-s`, and `-e`, `nkf` convert a file into ISO-2022-JP (aka JIS), SHIFT-JIS (aka MS-KANJI), and EUC-JP, respectively. Note that difference between JIS X 0201 Roman and ASCII is ignored. Though `nkf` can guess the encoding of the input file, you can specify the encoding by command option. This is because there are no algorithm to completely distinguish EUC-JP and SHIFT-JIS, though `nkf` usually guesses correctly. `tcs` can also convert these encodings, though without guessing input encoding. Conversion between these encodings can be done with a simple algorithm since all of them are based on the same character sets. You need a table for code conversion between these encodings and Unicode.

5.1.4 How These Encodings Are Used — Information for Programmers

Since EUC-JP is widely used for UNIX, EUC-JP should be supported. Exceptions are shown below. Of course direct implementation of knowledge on EUC-JP is not encouraged. If you can implement without the knowledge by use of `wchar_t` and so on, you should do so.

- the body of mail and news messages must be written in ISO-2022-JP.
- De-facto standard of ICQ is SHIFT-JIS.
- WWW browser must recognize all encodings.
- Softwares which communicate with Windows/Macintosh should use SHIFT-JIS.
- SHIFT-JIS is widely used for BBS. (BBS is a service like Compuserve).
- File names for Joliet-format CD-ROM used for Windows is written in Unicode.

5.1.5 Columns

In consoles which are able to display Japanese characters (kon, jfbterm, kterm, krxvt, and so on), characters in JIS X 0201 (Roman and Kana) occupy 1 column and characters in JIS X 0208, JIS X 0212, and JIS X 0213 occupy 2 columns.

5.1.6 Writing Direction and Combined Characters

Japanese language can be written in vertical direction. A line goes downward and the row of lies goes from right to left. This direction is the traditional style. For example, most Japanese books, magazines and newspapers except for in the field of natural science (or ones containing many Latin words or equations) are written in vertical direction. Thus a word processor is strongly recommended to support this direction. DTP systems which don't support this direction are almost useless.

Japanese language can also written in the same direction to Latin languages. Japanese books and magazines on science and technology are written in this direction. It is enough for almost usual softwares to support this direction only.

A few Japanese characters have to have different fonts for vertical direction. They are reasonable characters — parentheses and 'long syllable' symbol whose shape is like dash in English or mathematical 'minus' sign. Symbols equivalent to period and comma also have different style for horizontal and vertical direction.

In Japan, Arabic numerical characters are widely used, like European languages, though we have Kanji (ideogram) numerical characters. Latin characters can also appear in Japanese texts. If a row of 1 - 3 (or 4) characters of Arabic and Latin appear in Japanese vertical text, these characters can be crowded into one column. If more characters appear (large numbers or long words), the paper is rotated 90 degree in anticlockwise and the characters are written in European way. Sometimes Latin characters which appears in vertical text are written in the same way as Japanese character, i.e., vertical direction. This is not so strong custom. Arabic and Latin characters can always be written in both normal and rotated way in vertical text. [1] DTP system should support all of them.

A version of Japanized TeX (developed by ASCII, a publishing company in Japan) can use vertical direction. This can also treat a page containing both vertical and horizontal texts.

[1] I HAVE TO SHOW EXAMPLE USING GRAPHICS.

5.1.7 Layout of Characters

In Japanese language, words are not separated by space and a line can be broken anywhere, with a few exceptions, unlike European languages. Thus hyphenation is not needed for Japanese.

Characters like open parentheses cannot come to the end of a line. Characters like close parentheses and sorts of sentence-separating marks such as period and comma cannot come to the top of a line. This rule and processing is called 'kinsoku' in Japanese.

In European languages, a break of line is equivalent to a space. In Japanese language, a break of line should be neglected. For example, when rendering an HTML file, line-breaking character in the HTML source should not be converted into whitespace.

5.1.8 LANG variable

Different value of `LANG` used for different encodings.

Following values are used for EUC-JP.

- `LANG=ja_JP.eucJP` (major for Linux and *BSD)
- `LANG=ja_JP.ujis` (used to be major for Linux)
- `LANG=ja_JP` (because EUC-JP is the de-facto standard for UNIX; not recommended)
- `LANG=ja` (because EUC-JP is the de-facto standard for UNIX; not recommended)

`LANG=ja_JP.jis` is used for ISO-2022-JP (aka JIS code or JUNET code).

`LANG=ja_JP.sjis` is used for SHIFT-JIS (aka Microsoft Kanji Code).

Setting LANG is not sufficient for a Japanese user who has just installed Linux to get a minimal Japanese environment. There are several books on establishing Japanese environment on Linux/BSD and magazines on Linux often have feature articles on how to establish Japanese environment. Nowadays many Japanized Linux distributions which are optimized so that many basic software can display and input Japanese are popular. Debian GNU/Linux has `user-ja` (for potato) and `language-env` (for woody and following versions) packages to establish basic Japanese environment.

5.1.9 Input from Keyboard

Since Japanese characters cannot be inputed directly from a keyboard, a software is needed to convert ASCII characters into Japanese. `WNN`, `Canna`, and `SKK` are popular free softwares to input Japanese language. Though `T-Code` is also available, it is difficult to use. Since these adopt server/client model and implement their own protocols, we cannot input Japanese only with `wnn`, `canna`, or `skk` (and their depending packages).

In X Window System environment, `kinput2-*` and `skkinput` packages connects these protocols and XIM, which is the standard input protocol for X. Kinput2 also has an original protocol and `kterm` and so on can be a client of kinput2 protocol. Kinput2 protocol was developed before international standards such as XIM (or Ximp or Xsi) became available.

On console, there are no standard and each software has to support wnn and/or canna protocol. For example, `jvim-canna`, `xemacs21-mule-canna`, and emacs20 with `emacs-dl-canna` or `emacs-dl-wnn`. Thus the ways to operate are different between softwares. `skkfep` provides a general way to input Japanese on console.

Then the way to input Japanese is explained.

Since almost Hiraganas and Katakanas represents a pair of a vowel and a consonant with one character, we can input one Hiragana or one Katakana with two Latin alphabets. A few Hiraganas and Katakanas need one or three alphabets.

Kanji is obtained by converting from Hiragana. There are many Japanese words which are expressed by two or more Kanjis and almost recent converting softwares can convert such words at a time. (Old softwares can convert one Kanji at a time. You must be patient to use this way.) Softwares with good grammar/context analyzer and large dictionary can convert longer phrases or even a whole sentence at a time. However, we usually have to select one Kanji or word from candidates the software shows, because Japanese language has many homophones. For example, 61 Kanjis whose readings are 'KAN' and 6 words whose readings are 'KOUKOU' are registered in dictionary of `canna`. (Today, 2 Oct 1999, I saw a TV advertisement film of Japanese word processor which insists the software can correctly convert an input into 'a cafe which opened today', not 'a cafe which rotated today'. Though Japanese word 'KAITEN' means both 'open (a shop)' and 'rotate', the software knows it is more usual for a cafe to open than to rotate.)

The conversion from Hiragana to Kanji needs a large dictionary which contains the Kanji spelling and readings of Japanese major words and conjugation or inflection. Thus proprietary softwares tend to efficiently convert. They usually have dictionaries larger than few megabytes. Some of these recent proprietary softwares even analyze the topic or meaning of the inputed Hiragana sentence and choose the most appropriate homophone, though they often choose wrong ones.

Nowadays several proprietary conversion softwares such as ATOK, WNN6, and VJE for Linux are sold in Japan.

Since it is complex and hard work for users to input Japanese characters, we don't want to input Y (for YES) or N (for NO) in Japanese. We prefer learning such basic English words to inputing Japanese words by invoking conversion software, inputing Latin alphabetic expression of Japanese, converting it into Hiragana, converting it into Kanji, choosing the correct Kanji, determining the correct Kanji, and ending the conversion software each time we need to input yes or no or similar words.

5.1.10 More Detailed Discussions

Width of Characters

Different from European languages, Japanese characters should written in a fixed width. Exceptions arises when two symbols such as parentheses, periods and commas continue. Kerning should be done for such cases if the software is a word processor. A text editor need not.

Ruby

Ruby is a small (usually 1/2 in length and 1/4 in area or a bit smaller) characters written above (in horizontal direction) or at right side (in vertical direction) of the main text. This is usually used to show a reading of difficult Kanji.

Japanized TeX can use ruby by using an extra macro. Word processors should have Ruby faculty.

Upper And Lower Cases

Japanese character does not have upper and lower case although there two sets of phonograms, Hiragana and Katakana.

Thus `tolower()` and `toupper()` should not convert between Hiragana and Katakana.

Hiragana is used for usual text. Katakana is used mainly for express foreign or imported words, for example, KONPYU-TA for computer, MAIKUROSOFUTO for Microsoft, and AINSYUTAIN for Einstein.

Sorting

Phonograms (Hiragana and Katakana) have sorting order. The order is same to defined in JIS X 0208, with a few exceptions.

Ideograms (Kanji) sorting is difficult. They should be sorted by their reading but almost all kanji have a few readings according to the context. So if you want to sort Japanese text, you will need a dictionary of whole Japanese Kanji words. And more, a few Japanese words written in Kanji have different readings with exactly same series of Kanjis, this can occur especially for names of person. So it is usual that addressbook databases have two 'name' columns, one for Kanji expression and the other for Hiragana.

I know no softwares which can sort Japanese words in perfect way, including free and proprietary softwares.

Ro-ma ji (Alphabetic expression of Japanese)

We have a phonetic alphabetic expression of Japanese, Ro-ma ji. It has almost one-to-one correspondence to Japanese phonogram. It can be used to display Japanese text on Linux console and so on. Since Japanese have many homophones this expression can be crabbed.

There are several variants of Ro-ma ji.

The first distinguishing point is on handling of long syllable. For example, long syllable of 'E' is expressed in:

- 'E' with caret,
- 'E' with upper bar,
- only 'E' in which long syllable mark is ignored,
- 'EE',
- and so on.

The second distinguishing point is some special pairs of vowel and consonant. For example, Hiragana character for combination of 'T' and 'I' is pronounced like 'CHI'.

- TI or CHI, as described above,
- TU or TSU,
- SI or SHI,
- HU or FU,
- WO or O,
- TYA or CHA, and
- N or M.

5.2 Spanish language / used in Spain, most of America and Equatorial Guinea

Section written by Eusebio C Rufian-Zilbermann <eusebio@acm.org>.

Spanish is one of the official languages in Spain, the official language in most of the countries in the American continent and the official language in Equatorial Guinea. It is spoken in many other regions where it is not the official language. Other official languages in Spain are Galician, Catalan and Basque. These other languages each have their own specific issues with regards to Localization. They are not described in this section of the document.

The Spanish Language derives from the variation spoken in the Castille region. The term Castillian is sometimes used to refer to the Spanish language (particularly when an author wants to stress the fact that there are other languages spoken in Spain). Both Castillian and Spanish language refer to the same language, they are not different things.

5.2.1 Characters used in Spanish

Spanish uses a Latin alphabet. The numerical characters used in Spanish are the Arabic numerals.

The character that distinguishes Spanish from other Latin alphabets is the Ñ ('N' with tilde), which exists in uppercase and lowercase versions. Vowels in Spanish may have a mark (the accent) on top of them to indicate intensity intonation. This accent is required for orthography (written correctness) on lowercase vowels but it is optional in uppercase vowels. The letter 'u' may have a dieresis (like the German umlaut), both in uppercase and lowercase forms.

Some punctuation signs are characteristic of the Spanish language. The opening question mark and the opening exclamation sign look like the English question mark and exclamation sign rotated 180 degrees. The English question mark and exclamation sign are referred to as closing question mark and exclamation sign. The small underlined 'a' and 'o' are used mainly for ordinal numbers, similar to the small 'th' in English ordinals.

5.2.2 Character Sets

UNE (Una Norma Española) is the National Standards Organization in Spain. UNE is a member of the ISO and standards that have one-to-one correspondence are usually called by their ISO number, rather than their UNE number.

ISO 8859-1, also known as ISO Latin-1, contains the characters required for Spanish.

5.2.3 Codesets

The codeset mostly used for Spanish is ISO 8859-1. The codepage Windows 1252 a.k.a. Windows Latin-1 is a superset of ISO 8859-1 that adds some characters in the range 128 to 159. Other codesets are Unicode, Macintosh Roman (codepage 1000), MS-DOS Latin-1 (codepage 850) or less frequently MS-DOS Latin US (codepage 437) which contains accented lowercase characters but not uppercase. Some additional Latin codesets are EBCDIC CP500 and CP 1026 (used in IBM mainframes and terminal emulators), Adobe Standard (used as default for Postscript documents), Nextstep Latin, HP Roman 8 (for HPUX and Laserjet resident printer fonts) and the Latin codepage in OS/2. They are all stateless, 8-bit codepages (with the exception of Unicode that is 16-bit).

5.2.4 How These Codesets Are Used — Information for Programmers

In most cases it is safe to use ISO 8859-1 characters. Some exceptions are

- WWW browsers should recognize all codesets.
- Software which communicates with IBM mainframes, Macintosh, MS-DOS, Nextstep, HPUX, OS/2 should handle the corresponding encoding.
- File names for Joliet-format CD-ROM used for Windows is written in Unicode.
- Postscript interpreters should handle the Adobe Standard character set.
- Printer filters or drivers for HP printers should handle the Roman-8 character set if using the internal fonts.

5.2.5 Columns

On console displays, each character occupies one column. Printed text can be equally spaced (one column per character) or proportionally spaced (a character can occupy fractionally more or less than a column, depending on its shape).

Note: Even when using Traditional Sorting, ch and ll occupy two columns. See the comment on Traditional sorting in 'Sorting' on the facing page.

5.2.6 Writing Direction

Spanish is normally written in left to right lines arranged from top to bottom of the page. For artistic purposes it might be written in top to bottom columns arranged left to right within the page. This columnar arrangement would be expected only in graphic and charting programs (e.g., a drawing program, a spreadsheet graph or a page layout program for composing brochures) but regular text editors wouldn't be expected to implement this style.

5.2.7 Layout of Characters

In the Spanish language, words are separated by spaces and a line can be broken at a space, a punctuation sign or a hyphenated word.

There are several sets of paired characters in Spanish. Unlike English, question marks and exclamation signs are also paired. Other paired characters are the same as English (parenthesis, square brackets, and so forth). Opening characters shouldn't appear at the end of a line. Closing characters and punctuation signs such as period and comma shouldn't appear at the beginning of a line.

Words can be broken at a syllabus and hyphenated. Unlike English, syllabi in Spanish end in a vowel more often than in a consonant. Syllabi that end in a consonant letter are typically at the end of a word or followed by a syllabus that starts with another consonant. Anyway, the rules are not completely consistent and a hyphenation dictionary has to be used.

5.2.8 LANG variable

For Bash

```
set meta-flag on        # keep all 8 bits for keyboard input
set output-meta on      # keep all 8 bits for terminal output
set convert-meta off    # don't convert escape sequences
export LC_CTYPE=ISO_8859_1
```

For Tcsh

```
setenv LANG C
setenv LC_CTYPE "iso_8859_1"
```

5.2.9 Input from Keyboard

For the Spanish keyboard to work correctly, you need the command `loadkeys /usr/lib/kbd/keytables/es.map` in the corresponding startup (rc) file.

Most of the Spanish characters are input from the keyboard with a single stroke. A two-key combination is used for accent and dieresis marks above vowels. Traditional typewriter machines used a 'dead key' system with keys that would strike the paper without advancing the carriage to the next character. Typing on a computer keyboard simulates this behavior, typing the accent or dieresis key does not produce any visible output until a vowel is typed afterwards. Usually if the accent or dieresis key is followed by a consonant, the accent key is ignored. Accented or dieresis characters cannot be used for shortcut keys for selecting options.

The words for Yes and No are Sí (the character next to S is 'i' with acute accent) and No. We would commonly use the S and N keys for a Sí/No choice.

Spanish keyboards usually allow for typing not only the Spanish accent signs, but also the accent signs in French and other languages (grave accent, circumflex accent, umlaut on letters other than the u). Other character that is typically available is the cedilla C (that looks like a C with a comma underneath, used for Catalan, Portuguese and French words, for example). There is a Latin-American keyboard layout that does not contain the grave accent and the cedilla C.

5.2.10 More Detailed Discussions

Sorting

Traditional Spanish considered the combinations CH and LL individual single letters. For usage in computers, this required an additional effort for sorting and character counting algorithms. It was decided that the savings in not requiring special algorithms was significant enough and that it would be acceptable to treat them as 2 separate letters. Some software

that already had incorporated the special sorting algorithms now allows for choosing between 'Traditional Spanish Sort' and 'Modern Spanish Sort'.

Accents and dieresis are ignored for sorting purposes. The only exception is the rare case where two words are exactly the same and the accent is the only difference, the word with the unaccented character should be sorted first. E.g., camión (c-a-m-i-o with acute accent-n), camionero, este, éste (e with acute accent-s-t-e).

The ñ (n with tilde) is always sorted after the n and before the l. It cannot be intermixed with the n.

Number format, date and currency symbols

The use of the dot and the comma as a thousands separator and for decimal places is usually the opposite of US English. E.g., 1.000,00 instead of 1,000.00. Some Spanish-speaking countries, notably Mexico, follow the same standards as the US. It is desirable that programs can handle both forms as an independent setting.

The usual date format is DD-MM-YYYY rather than MM-DD-YYYY, but again this depends on the specific country. It is desirable to have the date format as a configurable parameter.

The currency symbol can be prepended or appended to the number and it can be one or several characters long. E.g., 100 PTA for Spanish pesetas or N$ 100 for Mexican pesos. It is desirable that the symbol and position can be individually defined and to allow for currency symbols longer than 1-character.

Varieties of Spanish

Spanish is spoken by a tremendous variety of people. Academics through the different Spanish-speaking countries realized that this could lead to a dismemberment of the language and founded the Academy of the Spanish Language. This academy has branches in most of the Spanish-speaking countries, there is a Royal Academy of the Spanish Language of Spain, an Academy of the Spanish Language of Mexico, et cetera. The members of this Academy study the local evolution of the languages in each country. They meet together to maintain a body of knowledge of what should be considered the Standard Spanish Language and what should be considered local or regional terms and slang terms.

In most cases, software can use terms that are within the Standard set by the Academy. When new terms appear (e.g., when a new product is created that has no previous name in the Spanish language) each region typically starts using a new word. When there is one or two terms that become the de-facto standard, the Academy would incorporate the new term into the Standard. This is a very slow process and there will be temporary usages in different regions within the Spanish-speaking worlds that conflict with each other. Some people speak about Spain-Spanish and American-Spanish but most of the time it doesn't really make sense to make this distinction. First of all, even within America, there are differences between the local varieties that may be greater than the differences with Spain itself. E.g., Spanish as spoken in Mexico, Colombia and Argentina may have between them as much differences as each of them when

compared to how it is spoken in Spain. A computer user in Ecuador may feel more comfortable overall with the terms used in Spain than with the terms used in Mexico (and of course, most comfortable with the terms used in Ecuador itself!). The options are to either produce one Spanish version of a software product that is an acceptable compromise (maybe not perfect) for all Spanish-speaking countries or to produce multiple versions to account for all the regional variations.

A plea to all the people who are localizing software into Spanish: Let's use our efforts judiciously and create one Spanish version and not many. Let's strive for a version that conforms to the Standards and that can be as widely accepted as possible for the areas not covered by the Standards. Wouldn't you rather have a new product translated, instead of two versions of a product where one matches your local variety of the language?

5.3 Languages with Cyrillic script

Section written by Alexander Voropay <a.voropay@globalone.ru>.

First of all, there are a lot of languages with Cyrillic script.

Slavic languages : Russian (ru), Ukrainian (uk), Belarussian (be), Bulgarian (bg), Serbian (sr), and Macedonian (mk).

Another Slavic languages (Polish(pl), Czech(cz), Croatian(hr)) uses Latin script : mainly ISO-8859-2 (Central-European).

During USSR time some non-slavic languages got own alpabets, based on modifyed cyrillic characters. Azerbaijani (az), Turkmen (tk), Kurdish (ku), Uzbek (uz), Kazakh (kk), Kirghiz (ky), Tajik (tg) and Mongolian (mn) Komi (kv) e.t.c.

- http://www.peoples.org.ru/eng_index.html
- http://www-hep.fzu.cz/~piska/
- http://www.srpsko-pismo.org/
- http://www.hr/hrvatska/language/CroLang.html
- http://ftp.fi.muni.cz/pub/localization/charsets/cs-encodings-faq

UNICODE has rich Cyrillic section.

Ufortunately, there are a lot of 8-bit Cyrillic Charsets. There is no one universal 8-bit Cyrillic charset, because, for example, there are about 260 Cyrillic characters in Adobe Glyph List (http://partners.adobe.com/asn/developer/PDFS/TN/5013.Cyrillic_Font_Spec.pdf).

The overview "The Cyrillic Charset Soup (http://czyborra.com/charsets/cyrillic.html)".

The main problem with Russian : there are at least six live Charsets:

- KOI8-R
- Windows-1251
- CP-866
- ISO-8859-5
- MAC-CYRILLIC
- ISO-IR-111

So, Russian computers really live in “Charset mix”, like Japanese : Shift-JIS, ISO2022-JP, EUC-JP. You can get e-mail in any charset, so your Mail Agent should understand all this charsets. Takasiganai.

In POSIX environment you should setup FULL locale name (with .Charset field) :

```
LANG=ru_RU.KOI8-R
LANG=ru_RU.ISO_8859-5
LANG=ru_RU.CP1251
```

e.t.c. for proper sorting, character classification and for readable messages. Any form of abbreviations (“`ru`”, “`ru_RU`” e.t.c.) are sourse of misunderstanding. I hope, Unicode `LANG=ru_RU.UTF-8` will save us in near future…

Chapter 6

LOCALE technology

LOCALE is a basic concept introduced into **ISO C** (ISO/IEC 9899:1990). The standard is expanded in 1995 (ISO 9899:1990 Amendment 1:1995). In LOCALE model, the behaviors of some C functions are dependent on LOCALE environment. LOCALE environment is divided into a few categories and each of these categories can be set independently using `setlocale()`.

POSIX also determines some standards around i18n. Almost of POSIX and ISO C standards are included in **XPG4** (X/Open Portability Guide) standard and all of them are included in XPG5 standard. Note that **XPG5** is included in UNIX specifications version 2. Thus support of XPG5 is mandatory to obtain Unix brand. In other words, all versions of Unix operating systems support XPG5.

The merit of using locale technology over hard-coding of Unicode is:

- The software can be written encoding-independent way. This means that this software can support all encodings which the OS supports, including 7bit, 8bit, multibyte, stateful, and stateless encodings such as ASCII, ISO 8859-*, EUC-*, ISO 2022-*, Big5, VISCII, TIS 620, UTF-*, and so on.
- The software will provides a common unified method to configure locale and encoding. This benefits users. Otherwise, users will have to remember the method to enable UTF-8 mode for each software. Some softwares need `-u8` switch, other need X resource setting, other need `.foobarrc` file, other need a special environmental variable, other use UTF-8 for default. It is nonsense!
- The advancement of the OS means the advancement of the software. Thus, you can use new locale without recompiling your software.

You can read the Unicode support in the Solaris Operating Environment (`http://docs.sun.com/ab2/coll.651.1/SOLUNICOSUPPT`) whitepapaer and understand the merit of this model. Bruno Haible's Unicode HOWTO (`ftp://ftp.ilog.fr/pub/Users/haible/utf8/Unicode-HOWTO.html`) also recommends this model.

6.1 Locale Categories and `setlocale()`

In LOCALE model, the behaviors of some C functions are dependent on LOCALE environment. LOCALE environment is divided into six categories and each of these categories can be set independently using `setlocale()`.

The followings are the six categories:

LC_CTYPE Category related to encodings. Characters which are encoded by LC_CTYPE-dependent encoding is called **multibyte characters**. Note that multibyte character doesn't need to be multibyte.

LC_CTYPE-dependent functions are: character testing functions such as `islower()` and so on, multibyte character functions such as `mblen()` and so on, multibyte string functions such as `mbstowcs()` and so on, and so on.

LC_COLLATE Category related to sorting. `strcoll()` and so on are LC_COLLATE-dependent.

LC_MESSAGES Category related to the language for messages the software outputs. This category is used for `gettext`.

LC_MONETARY Category related to format to show monetary numbers, for example, currency mark, comma or period, columns, and so on. `localeconv()` is the only function which is LC_MONETARY-dependent.

LC_NUMERIC Category related to format to show general numbers, for example, character for decimal point.

Formatted I/O functions such as `printf()`, string conversion functions such as `atof()`, and so on are LC_NUMERIC-dependent.

LC_TIME Category related to format to show time and date, such as name of months and weeks, order of date, month, and year, and so on.

`strftime()` and so on are LC_TIME-dependent.

`setlocale()` is a function to set LOCALE. Usage is char *`setlocale(`int *category*, const char **locale*`);`. Header file of `locale.h` is needed for prototype declaration and definition of macros for category names. For example, `setlocale(LC_TIME, "de_DE");`.

For *category*, the following macros can be used: LC_CTYPE, LC_COLLATE, LC_MONETARY, LC_NUMERIC, LC_TIME, and LC_ALL. For *locale*, specific locale name, `NULL`, or "" can be specified.

Giving `NULL` for *locale* will return the current value of the specified locale category. Otherwise, `setlocale()` returns the newly set locale name, or `NULL` for error.

Given "" for *locale*, `setlocale()` will determine the locale name in the following manner:

- At first, consult `LC_ALL` environmental variable.

- If `LC_ALL` is not available, consult environmental variable same as the name of the locale category. For example, `LC_COLLATE`.
- If none of them are available, consult `LANG` environmental variable.

This is why a user is expected to set `LANG` variable. In other words, all what a user has to do is to set `LANG` variable so that all locale-compliant softwares work well for desired way.

Thus, I recommend strongly to call `setlocale(LC_ALL, "");` at the first of your softwares, if the softwares are to be international.

6.2 Locale Names

We can specify locale names for these six locale categories. Then, which name should we specify?

The syntax to build a locale name is determined as follows:

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

where *language* is two lowercase alphabets described in ISO639, such as `en` for English, `eo` for Esperanto, and `zh` for Chinese, *territory* is two uppercase alphabets described in ISO3166, such as `GB` for United Kingdom, `KR` for Republic of Korea (South Korea), `CN` for China. There are no standard for *codeset* and *modifier*. GNU libc uses `ISO-8859-1`, `ISO-8859-13`, `eucJP`, `SJIS`, `UTF8`, and so on for *codeset*, and `euro` for *modifier*.

However, it is depend on the system which locale names are valid. In other words, you have to install *locale database* for locale you want to use. Type `locale -a` to display all supported locale names on the system.

Note that locale names of "`C`" and "`POSIX`" are determined for the names for default behavior. For example, when your software need to parse the output of `date(1)`, you'd better call `setlocale(LC_TIME, "C");` before invocation of `date(1)`.

6.3 Multibyte Characters and Wide Characters

Now we will concentrate on LC_CTYPE, which is the most important category in six locale categories.

Many encodings such as ASCII, ISO 8859-*, KOI8-R, EUC-*, ISO 2022-*, TIS 620, UTF-8, and so on are used widely in the world. It is inefficient and a cause of bugs, even not impossible, for every softwares to implement all these encodings. Fortunately, we can use LOCALE technology to solve this problem. [1]

[1] Usage of UCS-4 is the second best solution for this problem. Sometimes LOCALE technology cannot be used and UCS-4 is the best. I will discuss this solution later.

Multibyte characters is a term to call characters encoded in locale-specific encoding. It is nothing special. It is mere a word to call our daily encodings. In ISO 8859-1 locale, ISO 8859-1 is multibyte character. In EUC-JP locale, EUC-JP is multibyte character. In UTF-8 locale, UTF-8 is multibyte character. In short, multibyte character is defined by `LC_CTYPE` locale category. Multibyte characters is used when your software inputs or outputs text data from/to everywhere out of your software, for example, standard input/output, display, keyboard, file, and so on, as you are doing everyday. [2]

You can handle multibyte characters using ordinal `char` or `unsigned char` types and ordinal character- and string-oriented functions. It is just like you used to do for ASCII and 8bit encodings.

Then why we call it with a special term of *multibyte character*? The answer is, ISO C specifies a set of functions which can handle multibyte characters properly. On the other hand, it is obvious that usual C functions such as `strlen()` cannot handle multibyte characters properly.

Then what is these functions which can handle multibyte characters properly? Please wait a minute. Multibyte character may be stateful or stateless and multibyte or non-multibyte, since it includes all encodings ever used and will be used on the earth. Thus it is not convenient for internal processing. It needs complex algorithm even for, for example, character extraction from a string, addition and division of a string, or counting of number of character in a string. Thus, **wide characters** should be used for internal processing. And, the main part of these C functions which can handle multibyte characters are functions for interconversion between multibyte characters and wide characters. These functions are introduced later. Note that you may be able to do without these functions, since ISO C supplies I/O functions with conversion.

Wide character is defined in ISO C

- that all characters are expressed in fixed width of bits.
- that it is stateless, i.e., it doesn't have shift states.

There are two types for wide characters: `wchar_t` and `wint_t`. `wchar_t` is a type which can contain one wide character. It is just like 'char' type can be used for contain one character. `wint_t` can contain one wide character or `WEOF`, an substitution of `EOF`.

A string of wide characters is achieved by an array of `wchar_t`, just like a string of characters is achieved by an array of `char`.

There are functions for `wchar_t`, substitute for functions for `char`.

- `strcat()`, `strncat()` -> `wcscat()`, `wcsncat()`
- `strcpy()`, `strncpy()` -> `wcscpy()`, `wcsncpy()`
- `strcmp()`, `strncmp()` -> `wcscmp()`, `wcsncmp()`

[2] There are a few exceptions. Compound text should be used for communication between X clients. UTF-8 would be the standard for file names in Linux.

- `strcasecmp()`,`strncasecmp()` -> `wcscasecmp()`,`wcsncasecmp()`
- `strcoll()`,`strxfrm()` -> `wcscoll()`,`wcsxfrm()`
- `strchr()`,`strrchr()` -> `wcschr()`,`wcsrchr()`
- `strstr()`,`strpbrk()` -> `wcsstr()`,`wcspbrk()`
- `strtok()`,`strspn()`,`strcspn()` -> `wcstok()`,`wcsspn()`,`wcscspn()`
- `strtol()`,`strtoul()`,`strtod()` -> `wcstol()`,`wcstoul()`,`wcstod()`
- `strftime()` -> `wcsftime()`
- `strlen()` -> `wcslen()`
- `toupper()`,`tolower()` -> `towupper()`,`towlower()`
- `isalnum()`, `isalpha()`, `isblank()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()` -> `iswalnum()`,`iswalpha()`,`iswblank()`,`iswcntrl()`,`iswdigit()`,`iswgraph()`, `iswlower()`,`iswprint()`,`iswpunct()`,`iswspace()`,`iswupper()`,`iswxdigit()` (`isascii()` doesn't have its wide character version).
- `memset()`,`memcpy()`,`memmove`,`memmove()`,`memchr()` -> `wmemset()`,`wmemcpy()`, `wmemmove`,`wmemmove()`,`wmemchr()`

There are additional functions for `wchar_t`.

- `wcwidth()`,`wcswidth()`
- `wctrans()`,`towctrans()`

You cannot assume anything on the concrete value of `wchar_t`, besides `0x21` - `0x7e` are identical to ASCII. [3] You may feel this limitation is too strong. If you cannot do under this limitation, you can use UCS-4 as the internal encoding. In such a case, you can write your software emulating the locale-sensible behavior using `setlocale()`,`nl_langinfo(CODESET)`, and `iconv()`. Consult the section of '`nl_langinfo()` and `iconv()`' on page 55. Note that it is generally easier to use wide character than implement UCS-4 or UTF-8.

You can write wide character in the source code as `L'a'` and wide string as `L"string"`. Since the encoding for the source code is ASCII, you can only write ASCII characters. If you'd like to use other characters, you should use `gettext`.

There are two ways to use wide characters:

[3] Some of you may know GNU libc uses UCS-4 for the internal expression of `wchar_t`. However, you should not use the knowledge. It may differ in other systems.

- I/O is described using multibyte characters. Inputed data are converted into wide character immediately after reading and data for output are converted from wide character to multibyte character immediately before writing. Conversion can be achieved using functions of `mbstowcs()`, `mbsrtowcs()`, `wcstombs()`, `wcsrtombs()`, `mblen()`, `mbrlen()`, `mbsinit()`, and so on. Please consult the manual pages for these functions.

- Wide characters are directly used for I/O, using wide character functions such as `getwchar()`, `fgetwc()`, `getwc()`, `ungetwc()`, `fgetws`, `putwchar()`, `fputwc()`, `putwc()`, and `fputws()`, formatted I/O functions for wide characters such as `fwscanf()`, `wscanf()`, `swscanf()`, `fwprintf()`, `wprintf()`, `swprintf()`, `vfwprintf()`, `vwprintf()`, and `vswprintf()`, and wide character identifier of `%lc`, `%C`, `%ls`, `%S` for conventional formatted I/O functions. By using this approach, you don't need to handle multibyte characters at all. Please consult the manual pages for these functions.

Though latter functions are also determined in ISO C, these functions have became newly available since GNU libc 2.2. (Of course all UNIX operating systems have all functions described here.)

Note that very simple softwares such as `echo` doesn't have to care about multibyte character. and wide characters. Such software can input and output multibyte character as is. Of course you may modify these softwares using wide characters. It may be a good practice of wide character programming. Examples of a fragment of source codes will be discussed in 'Internal Processing and File I/O' on page 71.

There is an explanation of multibyte and wide characters also in Ken Lunde's "CJKV Information Processing" (p25). However, the explanation is entirely wrong.

6.4 Unicode and LOCALE technology

UTF-8 is considered as the future encoding and many softwares are coming to support UTF-8. Though some of these softwares implement UTF-8 directly, I recommend you to use LOCALE technology to support UTF-8.

How this can be achieved? It is easy! If you are a developer of a software and your software has already written using LOCALE technology, you don't have to do anything!

Using LOCALE technology benefits not only developers but also users. All a user has to do is set locale environment properly. Otherwise, a user has to remember the method to use UTF-8 mode for each software. Some softwares need `-u8` switch, other need X resource setting, other need `.foobarrc` file, other need a special environmental variable, other use UTF-8 for default. It is nonsense!

Solaris has been already developed using this model. Please consult Unicode support in the Solaris Operating Environment (`http://docs.sun.com/ab2/coll.651.1/SOLUNICOSUPPT`) whitepapaer.

However, it is likely that some of upstream developers of softwares of which you are maintaining a Debian package refuses to use `wchar_t` for some reasons, for example, that they are not familiar with LOCALE programming, that they think it is troublesome, that they are not keen on I18N, that it is much easier to modify the software to support UTF-8 than to modify it to use `wchar_t`, that the software must work even under non-internationalized OS such as MS-DOS, and so on. Some developers may think that support of UTF-8 is sufficient for I18N. [4] Even in such cases, you can rewrite such a software so that it checks `LC_*` and `LANG` environmental variables to emulate the behavior of `setlocale(LC_ALL, "");`. You can also rewrite the software to call `setlocale()`, `nl_langinfo()`, and `iconv()` so that the software supports all encodings which the OS supports, as discussed later. Consult the discussion in the Groff mailing list on the support of UTF-8 and locale-specific encodings (`http://ffii.org/archive/mails/groff/2000/Oct/0056.html`), mainly held by Werner LEMBERG, an experienced developer of GNU roff, and Tomohiro KUBOTA, the author of this document.

6.5 `nl_langinfo()` and `iconv()`

Though ISO C defines extensive LOCALE-related functions, you may want more extensive support. You may also want conversion between different encodings. There are C functions which can be used for such purposes.

char *`nl_langinfo`(nl_item *item*) is an XPG5 function to get LOCALE-related informations. You can get the following informations using the following macros for *item* defined in `langinfo.h` header file:

- names for days in week (`DAY_1` (Sunday), `DAY_2`, `DAY_3`, `DAY_4`, `DAY_5`, `DAY_6`, and `DAY_7`)
- abbreviated names for days in week (`ABDAY_1` (Sun), `ABDAY_2`, `ABDAY_3`, `ABDAY_4`, `ABDAY_5`, `ABDAY_6`, and `ABDAY_7`)
- names for months in year (`MON_1` (January), `MON_2`, `MON_3`, `MON_4`, `MON_5`, `MON_6`, `MON_7`, `MON_8`, `MON_9`, `MON_10`, `MON_11`, and `MON_12`)
- abbreviated names for months in year (`ABMON_1` (January), `ABMON_2`, `ABMON_3`, `ABMON_4`, `ABMON_5`, `ABMON_6`, `ABMON_7`, `ABMON_8`, `ABMON_9`, `ABMON_10`, `ABMON_11`, and `ABMON_12`)
- name for AM (`AM_STR`)
- name for PM (`PM_STR`)
- name of era (`ERA`)

[4] In such a case, do they think of abolishing support of 7bit or 8bit non-multibyte encodings? If no, it may be unfair that 8bit language speakers can use both UTF-8 and conventional (local) encodings while speakers of multibyte languages, combining characters, and so on cannot use their popular locale encodings. I think such a software cannot be called "internationalized".

- format of date and time (`D_T_FMT`)
- format of date and time (era-based) (`ERA_D_T_FMT`)
- format of date (`D_FMT`)
- format of date (era-based) (`ERA_D_FMT`)
- format of time (24-hour format) (`T_FMT`)
- format of time (am/pm format) (`T_FMT_AMPM`)
- format of time (era-based) (`ERA_T_FMT`)
- radix (`RADIXCHAR`)
- thousands separator (`THOUSEP`)
- alternative characters for numerics (`ALT_DIGITS`)
- affirmative word (`YESSTR`)
- affirmative response (`YESEXPR`)
- negative word (`NOSTR`)
- negative response (`NOEXPR`)
- encoding (`CODESET`)

For example, you can get names for months and use them for your original output algorithm. `YESEXPR` and `NOEXPR` are convenient for softwares expecting Y/N answer from users.

`iconv_open()`, `iconv()`, and `iconv_close()` are functions to perform conversion between encodings. Please consult manpages for them.

Combining `nl_langinfo()` and `iconv()`, you can easily modify Unicode-enabled software into locale-sensible truly internationalized software.

At first, add a line of `setlocale(LC_ALL, "");` at the first of the software. If it returns non-NULL, enable UTF-8 mode of the software.

```
int conversion = FALSE;
char *locale = setlocale(LC_ALL, "");
   :
   :
(original code to determine UTF-8 mode or not)
   :
   :
if (locale != NULL && utf_mode == FALSE) {
    utf8_mode = TRUE;
    conversion = TRUE;
}
```

Then modify input routine as following:

```
#define INTERNALCODE "UTF-8"
if (conversion == TRUE) {
    char *fromcode = nl_langinfo(CODESET);
    iconv_t conv = iconv_open(INTERNALCODE, fromcode);
    (reading and conversion...)
    iconv_close(conv);
} else {
    (original reading routine)
}
```

Finally modify the output routine as following:

```
if (conversion == TRUE) {
    char *tocode = nl_langinfo(CODESET);
    iconv_t conv = iconv_open(tocode, INTERNALCODE);
    (conversion and writing...)
    iconv_close(conv);
} else {
    (original writing routine)
}
```

Note that whole reading should be done at once since otherwise you may divide multibyte character. You can consult the `iconv_prog.c` file in the distribution of GNU libc for usage of `iconv()`.

Though `nl_langinfo()` is a standard function of XPG5 and GNU libc supports it, it is not very portable. And more, there are no standard for encoding names for `nl_langinfo()` and `iconv_open()`. If this is a problem, you can use Bruno Haible's libiconv (http://www.gnu.org/software/libiconv/). It has `iconv()`, `iconv_open()`, and `iconv_close()`. And more, it has `locale_charset()`, a replacement of `nl_langinfo(CODESET)`.

6.6 Limit of Locale technology

Locale model has a limit. That is, it cannot handle two locales at the same time. Especially, it cannot handle relationship between two locales at all.

For example, EUC-JP, ISO 2022-JP, and Shift-JIS are popular encodings in Japan. EUC-JP is the de-facto standard for UNIX systems, ISO 2022-JP is the standard for Internet, and Shift-JIS is the encoding for Windows and Macintosh. Thus, Japanese people have to handle texts with these encodings. Text viewers such as `jless` and `lv` and editors such as `emacs` can automatically understand the encoding to be read. You cannot write such a software using Locale technology.

Chapter 7

Output to Display

Here 'Output to Display' does not mean translation of messages using `gettext`. I will concern on whether characters are correctly displayed so that we can read it. For example, install `libcanna1g` package and display `/usr/doc/libcanna1g/README.jp.gz` on console or `xterm` (of course after ungzipping). This text file is written in Japanese but even Japanese people can not read such a row of strange characters. Which you would prefer if you were a Japanese speaker, an English message which can be read with a dictionary or such a row of strange characters which is a result of `gettext`ization? [1]

Problems on displaying non-English (non-ASCII) characters are discussed below.

7.1 Console Softwares

In this section, problems on displaying characters on **console** are discussed. [2] Here, console includes a bare **Linux console** including framebuffer and conventional version, special consoles such as **kon2**, **jfbterm**, **chdrv**, and so on constructed by special softwares, and X terminal emulators such as **xterm**, **kterm**, **hanterm**, **xiterm**, **rxvt**, **xvt**, **gnome-terminal**, **wterm**, **aterm**, **eterm**, and so on. Remote environments via telnet and secure shell such as **NCSA telnet** for Macintosh and **Tera Term** for Windows are also regarded as consoles.

The feature of console is that:

- All what a software has to do is to send a correct encoding to standard output. Softwares on console don't need to care about fonts and so on.
- Fonts with fixed sizes are used. The unit of the width of the font is called 'column'. 'Doublewidth' fonts, i.e., fonts whose width is 2 columns, are used for CJK ideograms, Japanese Hiragana and Katakana, Korean Hangul, and related symbols. Combined characters used for Thai and so on can be regarded as 'zero'-column characters.

[1] (Yes, there *are* ways to display Japanese characters correctly – `kon` (in `kon2` package) for console and `kterm` for X, and Japanese people are happy with `gettext`ized Japanese messages.)

[2] This section does not include problems on developing console; This section includes problems on developing softwares which run on console.

7.1.1 Encoding

Softwares running on the console are not responsible for displaying. The console itself is responsible. There are consoles which can display encodings other than ASCII such as

kon in kon2 package EUC-JP, Shift-JIS, and ISO-2022-JP

jfbterm EUC-JP, ISO 2022-JP, and ISO 2022 (including any 94, 96, and 94x94 coded character sets whose fonts are available)

kterm EUC-JP, Shift-JIS, ISO 2022-JP, and ISO 2022 (including ISO8859-{1,2,3,4,5,6,7,8,9}, JISX 0201, JISX 0208, JISX 0212, GB 2312, and KSC 5601)

krxvt in rxvt-ml package EUC-JP

crxvt-gb in rxvt-ml package CN-GB

crxvt-big5 in rxvt-ml package Big5

cxtermb5 in cxterm-big5 package Big5

xcinterm-big5 in xcin package Big5

xcinterm-gb in xcin package CN-GB

xcinterm-gbk in xcin package GBK

xcinterm-big5hkscs in xcin package Big5 with HKSCS

hanterm EUC-KR, Johab, and ISO 2022-KR

xiterm and txiterm in xiterm+thai package TIS 620

xterm UTF-8

However, there are no way for a software on console to know which encoding is available. I think it is a responsibility for a user to properly set LC_CTYPE locale (i.e. LC_ALL, LC_CTYPE, or LANG environmental variable). Provided LC_CTYPE locale is set properly, a software can use it to know which encoding to be supported by the console.

Concerning the translated messages by `gettext`, the software does not need anything. It works well if the user properly set LC_CTYPE and LC_MESSAGES locale.

If you are handling a string in non-ASCII encoding (using multibyte character, UTF-8 directly, and so on), you will have to care about points which you don't have to care about if you are using ASCII.

- 8-bit cleanness. I think everyone understand this.
- Continuity of multibyte characters. In multibyte encodings such as EUC-JP and UTF-8, one character may consist from more than two bytes. These bytes should be outputed continued. Insertion of additional codes between the continuing bytes can break the character. I have seen a software which outputs location control code everytime it outputs one byte. It breaks multibyte character.

7.1.2 Number of Columns

Internationalized console software cannot assume that a character always occupy one column. You can get the number of column of a character of a string using `wcwidth()` and `wcswidth()`. Note that you have to use `wchar_t`-style programming since these functions have a `wchar_t` parameter.

Additional cares have to be taken not to destroy multicolumn characters. For example, imagine your software displayed a double-column character at (row, column) = (1, 1). What will occur when your software then display a single-column character at (row, column) = (1, 2) or at (1, 1) ? The single-column character erases the half of the double-column character? Nobody knows the answer. It depends on the implementation of the console. All what I can tell is that your software should avoid such cases.

If your software inputs a string from keyboard, you will have to take more cares. All of numbers of characters, bytes, and columns differ. For example, in UTF-8 encoding, one character of 'a' with acute accent occupies two bytes and one column. One character of CJK-ideograph occupies three bytes and two columns. For example, if the user types 'Backspace', how many backspace code (0x08) should the software outputs? How many bytes should the software erase from the internal buffer? Don't be nervous; you can use `wchar_t` which assures one character occupy one `wchar_t` everytime and you can use `wcwidth()` to know the number of columns. Note that control codes such as 'backspace' (0x08) and so on are column-oriented everytime. It backs 'one' column even if the character at the position is a doublewidth character.

7.2 X Clients

The way to develop X clients can differ drastically dependent on the toolkits to be used. At first, Xlib-style programming is discussed since Xlib is the fundamental for all other toolkits. Then a few toolkits are discussed.

7.2.1 Xlib programming

X itself is already internationalized. X11R5 has introduced an idea of 'fontset' for internationalized text output. Thus all what X clients have to do is to use the 'fontset'-related functions.

The most important part for internationalization of displaying for X clients is the usage of internationalized **XFontSet**-related functions introduced since X11R5 instead of conventional **XFontStruct**-related functions.

The main feature of XFontSet is that it can handle multiple fonts at the same time. This is related to the distinction between coded character set (CCS) and character encoding scheme (CES) which I wrote at the section of 'Basic Terminology' on page 9. Some encodings in the world use multiple coded character sets at the same time. This is the reason we have to handle

multiple X fonts at the same time. [3]

Another significant feature of XFontSet is that it is locale (LC_CTYPE)-sensible. This means that you have to call `setlocale()` before you use XFontSet-related functions. And more, you have to specify the string you want to draw as a multibyte character or a wide character.

In the conventional `XFontStruct` model, an X client opens a font using `XLoadQueryFont()`, draw a string using `XDrawString()`, and close the font using `XFreeFont()`. On the other hand, in the internationalized `XFontSet` model, an X client opens a font using `XCreateFontSet()`, draw a string using `XmbDrawString()`, and close the font using `XFreeFontSet()`. The following are a concise list of substitution.

- `XFontStruct` -> `XFontSet`
- `XLoadQueryFont()` -> `XCreateFontSet()`
- both of `XDrawString()` and `XDrawString16` -> either of `XmbDrawString()` or `XwcDrawString()`
- both of `XDrawImageString()` and `XDrawImageString16` -> either of `XmbDrawImageString()` or `XwcDrawImageString()`

Note that `XFontStruct` is usually used as a pointer, while `XFontSet` itself is a pointer.

Some people (ISO-8859-1-language speakers) may think that `XFontSet`-related functions are not 8-bit clean. This is wrong. `XFontSet`-related functions work according to `LC_CTYPE` locale. The default LC_CTYPE locale uses ASCII. Thus, if a user doesn't set `LANG`, `LC_CTYPE`, nor `LC_ALL` environmental variable, `XFontSet`-related functions will use ASCII, i.e., not 8-bit clean. The user has to set `LANG`, `LC_CTYPE`, or `LC_ALL` environmental variable properly (for example, `LANG=en_US`).

The upstream developers of X clients sometimes hate to enforce users to set such environmental variables. [4] In such a case, The X clients should have two ways to output text, i.e., `XFontStruct`-related conventional way and `XFontSet`-related internationalized way. If `setlocale()` returns `NULL`, "`C`", or "`POSIX`", use `XFontStruct` way. Otherwise use `XFontSet` way. The author implemented this algorithm to a few window managers such as TWM (version 4.0.1d), Blackbox (0.60.1), IceWM (1.0.0), sawmill (0.28), and so on.

Window managers need more modifications related to inter-clients communication. This topic will be described later.

7.2.2 Athena widgets

Athena widget is already internationalized.

***** Not written yet *****

[3] Though UTF-8 is an encoding with single CCS, the current version of XFree86 (4.0.1) needs multiple fonts to handle UTF-8.

[4] IMHO, all users will have to set LANG properly when UTF-8 will become popular.

7.2.3 Gtk and Gnome

Gtk is already internationalized.

***** Not written yet *****

7.2.4 Qt and KDE

Though internationalized version of Qt was available for a long time, it could not be the official version of Qt. The license of Qt of those days inhibited to distribute internationalized version of Qt. However, Troll Tech at last changed their mind and Qt's license and now the official version of Qt is internationalized.

***** Not written yet *****

Chapter 8

Input from Keyboard

it is obvious that a text editor needs ability to input text from keyboard, otherwise the text editor is entirely useless. Similarly, an internationalized text editor needs ability to input characters used for various languages. Other softwares such as shells, libraries such as readline, environments such as consoles and X terminal emulators, script languages such as perl, tcl/tk, python, and ruby, and application softwares such as word processors, draw and paints, file managers such as Midnight Commander, web browsers, mailers, and so on also need ability to input internationalized text. Otherwise these softwares are entirely useless.

There are various languages in the world. Thus, proper input methods vary from languages to languages.

- Some languages such as English doesn't need any special input methods. All characters for the language can be inputted by a single key on a keyboard. Keymap is all which a user has to care.
- Some other languages such as German need a simple extension. For example, u with umlaut can be inputted with two strokes of ':' and 'u'. A way to switch ordinal input mode (key strokes of ':' and 'u' inputs ':' and 'u') and the extension input mode (key strokes of ':' and 'u' bears u with umlaut) has to be supplied. Almost languages in the world can be inputted with this method.
- Other languages such as Chinese and Japanese need a complicated input method, since they use thousands of characters. Since it is very difficult and challenging problem to develop a clever input method, a few companies are developing Japanese input methods. Typical Japanese input methods are shipped with tens of megabytes of conversion dictionary. It is often very troublesome to set up an input method for these languages. [1] You also have to be practiced to use these input methods.

Different technologies are used for these languages. The aim of this chapter is to introduce technologies for them.

[1] This is a field where proprietary systems such as MS Windows and Macintosh are much easier than free systems such as Debian and FreeBSD.

8.1 Non-X Softwares

Ideally, it is a responsibility for console and X terminal emulators to supply an input method. This situation is already achieved for simple languages which don't need complicated input methods. Thus, non-X softwares don't need to care about input methods.

There are a few Debian packages for consoles and X terminal emulators which supply input methods for particular languages.

xiterm in xiterm+thai package Thai characters

hanterm Korean Hangul

cxtermb5 in cxterm-big5 package Big5 traditional Chinese ideograms

cce CN-GB simplified Chinese ideograms

And more, there are a few softwares which supply input methods for existing console environment.

skkfep Japanese (needs SKK as a conversion engine)

uum Japanese (needs Wnn as a conversion engine; not avaliable as a Debian package)

canuum Japanese (needs Canna as a conversion engine; not avaliable as a Debian package)

However, since input methods for complex languages have not been available historically, a few non-X softwares have been developed with input methods.

jvim-canna A text editor which can input Japanese (needs Canna as a conversion engine.)

jed-canna A text editor which can input Japanese (needs Canna as a conversion engine.)

nvi-m17n-canna A text editor which can input Japanese (needs Canna as a conversion engine.)

You have to take care of the differences between number of *characters*, *columns*, and *bytes*. For example, you can find immediately that `bash` cannot handle UTF-8 input properly when you invoke `bash` on UTF-8 Xterm and push BackSpace key. This is because `readline` always erase one column on the screen and one byte in the internal buffer for one stroke of 'BackSpace' key. To solve this problem, **wide character** should be used for internal processing. One stroke of 'BackSpace' should erase `wcwidth()` columns on the screen and one `wchar_t` unit in the internal buffer.

8.2 X Softwares

X11R5 is the first internationalized version of X Window System. However, X11R5 supplied two sample implements of international text input. They are **Xsi** and **Ximp**. Existence of two different protocols was an annoying situation. However, X11R6 determined **XIM**, a new protocol for internationalized text input, as the standard. Internationalized X softwares should support text input using XIM.

They are designed using *server-client* model. The client calls the server when necessary. The server supplies conversion from key stroke to internationalized text.

Kinput and **kinput2** are protocols for Japanese text input, which existed before X11R5. Some softwares such as `kterm` and so on supports kinput2 protocol. `kinput2` is the server software. Since the current version of `kinput2` supports XIM protocol, you don't need to support kinput protocol.

8.2.1 Developing XIM clients

***** Not written yet *****

Development of XIM client is a bit complicated. You can read source code for `rxvt` and `xedit` to study.

Programming for Japanse characters input (`http://www.ainet.or.jp/~inoue/im/index-e.html`) is a good introduction to XIM programming.

8.2.2 Examples of XIM softwares

The following are examples of softwares which can work as XIM clients.

- X Terminal Emulators such as `krxvt`, `kterm`, and so on.
- Text editors such as `xedit`, `gedit`, and so on.
- Web rowser `mozilla`.

The following are examples of softwares which can work as XIM servers.

- `kinput` and `skkinput` for Japanese.

8.2.3 Using XIM softwares

Here I will explain how to use XIM input with Debian system. This will help developers and package maintainers who want to test XIM facility of their softwares. Debian Woody or later systems are assumed.

At first, locale database has to be prepared. Uncomment `ja_JP.EUC-JP EUC-JP`, `ko_KR.EUC-KR EUC-KR`, `zh_CN.GB2312`, and `zh_TW BIG5` lines in `/etc/locale.gen` and invoke `/usr/sbin/locale-gen`. This will prepare locale database under `/usr/share/locale/`. For systems other than Debian Woody or later, please take the valid procedure for these systems to prepare locale database.

Basic Chinese, Japanese, and Korean X fonts are included in `xfonts-base` package for Debian Woody and later.

XIM server must be installed. For **Japanese**, `kinput2` or `skkinput` packages are available. `kinput2` supports Japanese input engines of **Canna** and **FreeWnn** and `skkinput` supports **SKK**. For **Korean**, `ami` is available. For **traditional Chinese** and **simplified Chinese**, `xcin` is available.

Of course you need an XIM client software. `xedit` in `xbase-clients` package is an example of XIM client.

Then, login as a non-root user. Environment variables of `LC_ALL` (or `LANG`) and `XMODIFIERS` must be set as following.

- for **Japanese**/**kinput2**: `LC_ALL=ja_JP.eucJP` and `XMODIFIERS=@im=kinput2`
- for **Korean**/**ami**: `LC_ALL=ko_KR.eucKR` and `XMODIFIERS=@im=Ami`
- for **traditional Chinese**/**xcin**: `LC_ALL=zh_TW.Big5` and `XMODIFIERS=@im=xcin`
- for **simplified Chinese**/**xcin**: `LC_ALL=zh_CN.GB2312` and `XMODIFIERS=@im=xcin-zh_CN.GB2312`

Then invoke the XIM server. Just invoke it with background mode (with &). **kinput2** and **ami** don't open a new window while **xcin** opens a new window and outputs some messages.

Then invoke the XIM client. Focus on an input area of the software. Hit Shift-Space or Control-Space and type something. Did some strange characters appear? This document is too brief to explain how to input valid CJK characters and sentences with these XIM servers. Please consult documents of XIM servers.

8.3 Emacsen

GNU Emacs and **XEmacs** take an entirely different model for international input.

They supply all input methods for various languages. Instead of relying on console or XIM, they use these input methods. These input methods can be selected by `M-x set-input-method` command. The selected input method can be switched on and off by `M-x toggle-input-method` command.

GNU Emacs supplies input methods for British, Catalan, Chinese (array30, 4corner, b5-quick, cns-quick, cns-tsangchi, ctlau, ctlaub, ecdict, etzy, punct, punct-b5, py, py-b5, py-punct,

py-punct-b5, qj, qj-b5, sw, tonepy, ziranma, zozy), Czech, Danish, Devanagari, Esperanto, Ethiopic, Finnish, French, German, Greek, Hebrew, Icelandic, IPA, Irish, Italian, Japanese (egg-wnn, skk), Korean (hangul, hangul3, hanja, hanja3), Lao, Norwegian, Portuguese, Romanian, Scandinavian, Slovak, Spanish, Swedish, Thai, Tibetan, Turkish, Vietnamese, Latin-{1,2,3,4,5}, Cyrillic (beylorussian, jcuken, jis-russian, macedonian, serbian, transit, transit-bulgarian, ul-rainian, yawerty), and so on.

Chapter 9

Internal Processing and File I/O

There are many text-processing softwares, such as `grep`, `groff`, `head`, `sort`, `wc`, `uniq`, `nl`, `expand`, and so on. There are also many script languages which are often used for text processing, such as `sed`, `awk`, `perl`, `python`, `ruby`, and so on. These softwares need to be internationalized.

From a user's point of view, a software can use any internal encodings if I/O is done correctly. It is because a user cannot be aware of which kind of internal code is used in the software.

There are two candidate for internal encoding. One is **wide character** and the another is **UCS-4**. You can also use Mule-type encoding, where a pair of a number to express CCS and a number to express a character consist a unit.

I recommend to use *wide character*, for reasons I alread explained in 'LOCALE technology' on page 49, i.e., wide character can be encoding-independent and can support various encodings in the world including UTF-8, can supply a common united way for users to choose encodings, and so on.

Here a few examples of handling of `wchar_t` are shown.

9.1 Stream I/O of Characters

The following program is a small example of stream I/O of wide characters.

```
#include <stdio.h>
#include <wchar.h>
#include <locale.h>
main()
{
 wint_t c;

 setlocale(LC_ALL, "");
```

```
 while(1) {
 c = getwchar();
 if (c == WEOF) break;
 putwchar(c);
 }
}
```

I think you can easily imagine a corresponding version using `char`. Since this software does not do any character manipulation, you can use ordinal `char` for this software.

There are a few points. At first, never forget to call `setlocale()`. Then, `putwchar()`, `getwchar()`, and `WEOF` are the replacements of `putchar()`, `getchar()`, and `EOF`, respectively. Use `wint_t` instead of `int` for `getwchar()`.

9.2 Character Classification

Here is an example of character clasification using `wchar_t`. At first, this is a non-internationalized version.

```
/*
 *  wc.c
 *
 *  Word Counter
 *
 */

#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
 int n, p=0, d=0, c=0, w=0, l=0;

 while ((n=getchar()) != EOF) {
 c++;
 if (isdigit(n)) d++;
 if (strchr(" \t\n", n)) w++;
 if (n == '\n') l++;
 }

 printf("%d characters, %d digits, %d words, and %d lines\n",
 c, d, w, l);
}
```

Here is the internationalized version.

```
/*
 *  wc-i.c
 *
 *  Word Counter (internationalized version)
 *
 */

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

int main(int argc, char **argv)
{
 int p=0, d=0, c=0, w=0, l=0;
 wint_t n;

 setlocale(LC_ALL, "");

 while ((n=getwchar()) != EOF) {
 c++;
 if (iswdigit(n)) d++;
 if (wcschr(L" \t\n", n)) w++;
 if (n == L'\n') l++;
 }

 printf("%d characters, %d digits, %d words, and %d lines\n",
 c, d, w, l);
}
```

This example shows that `iswdigit()` is used instead of `isdigit()`. And more, `L"string"` and `L'char'` for wide character string and wide character.

9.3 Length of String

The following is a sample program to obtain the length of the inputed string. Note that number of bytes and number of characters are not distinguished.

```
/* length.c
 *
 * a sample program to obtain the length of the inputed string
 * NOT INTERNATIONALIZED
 */

#include <stdio.h>
```

```
#include <string.h>

int main(int argc, char **argv)
{
 int len;

 if (argc < 2) {
 printf("Usage: %s [string]\n", argv[0]);
 return 0;
 }

 printf("Your string is: \"%s\".\n", argv[1]);

 len = strlen(argv[1]);
 printf("Length of your string is: %d bytes.\n", len);
 printf("Length of your string is: %d characters.\n", len);
 printf("Width of your string is: %d columns.\n", len);
 return 0;
}
```

The following is a internationalized version of the program using wide characters.

```
/* length-i.c
 *
 * a sample program to obtain the length of the inputed string
 * INTERNATIONALIZED
 */

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

int main(int argc, char **argv)
{
 int len, n;
 wchar_t *wp;

 /* All softwares using locale should write this line */
 setlocale(LC_ALL, "");

 if (argc < 2) {
 printf("Usage: %s [string]\n", argv[0]);
 return 0;
 }
```

```
 printf("Your string is: \"%s\".\n", argv[1]);

 /* The concept of 'byte' is universal. */
 len = strlen(argv[1]);
 printf("Length of your string is: %d bytes.\n", len);

 /* To obtain number of characters, it is the easiest way */
 /* to convert the string into wide string.  The number of */
 /* characters is equal to the number of wide characters. */
 /* It does not exceed the number of bytes. */
 n = strlen(argv[1]) * sizeof(wchar_t);
 wp = (wchar_t *)malloc(n);
 len = mbstowcs(wp, argv[1], n);
 printf("Length of your string is: %d characters.\n", len);

 printf("Width of your string is: %d columns.\n", wcswidth(wp, len));

 return 0;
}
```

This program can count multibyte characters correctly. Of course the user has to set LANG variable properly.

For example, on UTF-8 xterm...

```
$ export LANG=ko_KR.UTF-8
$ ./length-i (a Hangul character)
Your string is: "(the character)"
Length of your string is: 3 bytes.
Length of your string is: 1 characters.
Width of your string is: 2 columns.
```

9.4 Extraction of Characters

The following program extracts all characters contained in the given string.

```
/* extract.c
 *
 * a sample program to extract each character contained in the string
 * not internationalized
 */

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

```
int main(int argc, char **argv)
{
 char *p;
 int c;

 if (argc < 2) {
 printf("Usage: %s [string]\n", argv[0]);
 return 0;
 }

 printf("Your string is: \"%s\".\n", argv[1]);

 c = 0;
 for (p=argv[1] ; *p ; p++) {
 printf("Character #%d is \"%c\".\n", ++c, *p);
 }
 return 0;
}
```

Using wide characters, the program can be rewritten as following.

```
/* extract-i.c
 *
 * a sample program to extract each character contained in the string
 * INTERNATIONALIZED
 */

#include <stdio.h>
#include <string.h>
#include <locale.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
 wchar_t *wp;
 char p[MB_CUR_MAX+1];
 int c, n, len;

 /* Don't forget. */
 setlocale(LC_ALL, "");

 if (argc < 2) {
 printf("Usage: %s [string]\n", argv[0]);
 return 0;
```

```
    }

    printf("Your string is: \"%s\".\n", argv[1]);

    /* To obtain each character of the string, it is easy to convert */
    /* the string into wide string and re-convert each of the wide */
    /* string into multibyte characters. */
    n = strlen(argv[1]) * sizeof(wchar_t);
    wp = (wchar_t *)malloc(n);
    len = mbstowcs(wp, argv[1], n);
    for (c=0; c<len; c++) {
    /* re-convert from wide character to multibyte character */
    int x;
    x = wctomb(p, wp[c]);
    /* One multibyte character may be two or more bytes. */
    /* Thus "%s" is used instead of "%c". */
    if (x>0) p[x]=0;
    printf("Character #%d is \"%s\" (%d byte(s)) \n", c, p, x);
    }

    return 0;
}
```

Note that this program doesn't work well if the multibyte character is stateful.

Chapter 10

the Internet

The Internet is a world-wide network of computer. Thus the text data exchanged via the Internet must be internationalized.

The concept of internationalization did not exist at the dawn of the Internet, since it was developed in US. Protocols used in the Internet were developed to be upward-compatible with the existing protocols.

One of the key technology of the internationalization of the Internet data exchange is **MIME**.

10.1 Mail/News

Internet mail uses SMTP (RFC 821 (http://www.faqs.org/rfcs/rfc821.html)) and ESMTP (RFC 1869 (http://www.faqs.org/rfcs/rfc1869.html)) protocols. SMTP is 7bit protocol and ESMTP is 8bit.

Original SMTP can only send ASCII characters. Thus non-ASCII characters (ISO 8859-*, Asian characters, and so on) have to be converted into ASCII characters.

MIME (RFC 2045 (http://www.faqs.org/rfcs/rfc2045.html), 2046 (http://www.faqs.org/rfcs/rfc2046.html), 2047 (http://www.faqs.org/rfcs/rfc2047.html), 2048 (http://www.faqs.org/rfcs/rfc2048.html), and 2049 (http://www.faqs.org/rfcs/rfc2049.html)) deals with this problem.

At first RFC 2045 (http://www.faqs.org/rfcs/rfc2045.html) determines three new headers.

- MIME-Version:
- Content-Type:
- Content-Transfer-Encoding:

Now `MIME-Version` is 1.0 and thus all MIME mails have a header like this:

```
MIME-Version: 1.0
```

`Content-Type` describes the type of content. For example, an usual mail with Japanese text has a header like that:

```
Content-Type: text/plain; charset="iso-2022-jp"
```

Available types are described in RFC 2046 (http://www.faqs.org/rfcs/rfc2046.html). `Content-Transfer-Encoding` describes the way to convert the contents. Available values are `BINARY`, `7bit`, `8bit`, `BASE64`, and `QUOTED-PRINTABLE`. Since SMTP cannot handle 8bit data, `8bit` and `BINARY` cannot be used. ESMTP can use them. Base64 and quoted-printable are ways to convert 8bit data into 7bit and 8bit data have to be converted using either of them to sent by SMTP.

RFC 2046 (http://www.faqs.org/rfcs/rfc2046.html) describes media type and sub type for `Content-Type` header. Available types are `text`, `image`, `audio`, `video`, and `application`. Now we are interested in `text` because we are discussing about i18n. Sub types for `text` are `plain`, `enriched`, `html`, and so on. `charset` parameter can also be added to specify encodings. `US-ASCII`, `ISO-8859-1`, `ISO-8859-2`, ..., `ISO-8859-10` are defined by RFC 2046 (http://www.faqs.org/rfcs/rfc2046.html) for `charset`. This list can be added by writing a new RFC.

- RFC 1468 (http://www.faqs.org/rfcs/rfc1468.html) `ISO-2022-JP`
- RFC 1554 (http://www.faqs.org/rfcs/rfc1554.html) `ISO-2022-JP-2`
- RFC 1557 (http://www.faqs.org/rfcs/rfc1557.html) `ISO-2022-KR`
- RFC 1922 (http://www.faqs.org/rfcs/rfc1922.html) `ISO-2022-CN`
- RFC 1922 (http://www.faqs.org/rfcs/rfc1922.html) `ISO-2022-CN-EXT`
- RFC 1842 (http://www.faqs.org/rfcs/rfc1842.html) `HZ-GB-2312`
- RFC 1641 (http://www.faqs.org/rfcs/rfc1641.html) `UNICODE-1-1`
- RFC 1642 (http://www.faqs.org/rfcs/rfc1642.html) `UNICODE-1-1-UTF-7`
- RFC 1815 (http://www.faqs.org/rfcs/rfc1815.html) `ISO-10646-1`

RFC 2045 (http://www.faqs.org/rfcs/rfc2045.html) and and RFC 2046 (http://www.faqs.org/rfcs/rfc2046.html) determine the way to write non-ASCII characters in the main text of mail. On the other hand, RFC 2047 (http://www.faqs.org/rfcs/rfc2045.html) describes 'encoded words' which is the way to write non-ASCII characters in the header. It is like that: `=?`*encoding*`?`*conversion algorithm*`?`*data*`?=`, where *encoding* is selected from the list of `charset` of `Content-Type` header, *algorithm* is `Q` or `q` for quoted-printable or `B` or `b` for base64, and *data* is encoded data whose length is less than 76 bytes. If the *data* is longer than 75 bytes, it must be divided into multiple encoded words. For example,

```
Subject: =?ISO-2022-JP?B?GyRCNEE7eiROJTUlViU4JSclLyVIGyhC?=
```

reads 'a subject written in Kanji' in Japanese (ISO-2022-JP, encoded by base64). Of course human cannot read it.

10.2 WWW

WWW is a system that HTML documents (mainly; and files in other formats) are transferred using HTTP protocol.

HTTP protocol is defined by RFC 2068 (http://www.faqs.org/rfcs/rfc2068.html). HTTP uses headers like mails and `Content-Type` header is used to describe the type of the contents. Though `charset` parameter can be described in the header, it is rarely used.

RFC 1866 (http://www.faqs.org/rfcs/rfc1866.html) describes that the default encoding for HTML is ISO-8859-1. However, many web pages are written in, for example, Japanese and Korean using (of course) encodings different from ISO-8859-1. Sometimes the HTML document describes:

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-2022=jp">
```

which declares that the page is written in ISO-2022-JP. However, there many pages without any declaration of encoding.

Web browsers have to deal with such a circumstance. Of course web browsers have to be able to deal with every encodings in the world which is listed in MIME. However, many web browsers can only deal with ASCII or ISO-8859-1. Such web browsers are useless at all for non-ASCII or non-ISO-8859-1 people.

URL should be written in ASCII character, though non-ASCII characters can be expressed using `%`*nn* sequence where *nn* is hexadecimal value. This is because there are no way to specify encoding. Wester-European people would treat it as ISO-8859-1, while Japanese people would treat it as EUC-JP or SHIFT-JIS.

Chapter 11

Libraries and Components

We sometimes use libraries and components which are not very popular. We may have to pay special attention for internationalization of these libraries and components.

On the other hand, we can use libraries and components for improvement of internationalization. This chapter introduces such a libraries and components.

11.1 Gettext and Translation

GNU Gettext is a tool to internationalize messages a software outputs according to locale status of `LC_MESSAGES`. A `gettext`ized software contains messages written in various languages (according to available translators) and a user can choose them using environmental variables. GNU gettext is a part of Debian system.

Install `gettext` package and read info pages for details.

Don't use non-ASCII characters for '`msgid`'. Be careful because you may tend to use ISO-8859-1 characters. For example, '©' (copyright mark; you may be not able to read the copyright mark NOW in THIS document) is non-ASCII character (0xa9 in ISO-8859-1). Otherwise, translators may feel difficulty to edit catalog files because of conflict between encodings for `msgid` and in `msgstr`.

Be sure the message can be displayed in the assumed environment. In other words, you have to read the chapter of 'Output to Display' in this document and internationalize the output mechanism of your software prior to `gettext`ization. *ENGLISH MESSAGES ARE PREFERRED EVEN FOR NON-ENGLISH-SPEAKING PEOPLE, THAN MEANINGLESS BROKEN MESSAGES.*

The 2nd (3rd, ...) byte of multibyte characters or all bytes of non-ASCII characters in stateful encodings can be 0x5c (same to backslash in ASCII) or 0x22 (same to double quote in ASCII). These characters have to properly escaped because present version of GNU gettext doesn't care the 'charset' subitem of '`Content-Type`' item for '`msgstr`'.

A gettexted message must not used in multiple contexts. This is because a word may have different meaning in different context. For example, a verb means an order or a command if it appears at the top of the sentence in English. However, different languages have different grammar. If a verb is gettexted and it is used both in a usual sentence and in an imperative sentence, one cannot translate it.

If a sentence is gettexted, never divide the sentence. If a sentence is divided in the original source code, connect them so as to single string contains the full sentence. This is because the order of words in a sentence is different among languages. For example, a routine

```
printf("There ");
switch(num_of_files) {
case 0:
        printf("are no files ");
        break;
case 1:
        printf("is 1 file ");
        break;
default:
        printf("are %d files ", num_of_files);
        break;
}
printf("in %s directory.\n", dir_name);
```

has to be written like that:

```
switch(num_of_files) {
case 0:
        printf("There are no files in %s directory", dir_name);
        break;
case 1:
        printf("There is 1 file in %s directory", dir_name);
        break;
default:
        printf("There are %d files in %s directory", num_of_files, dir_name);
        break;
}
```

before it is gettextized.

A software with gettexted messages should not depend on the length of the messages. The messages may get longer in different language.

When two or more '%' directive for formatted output functions such as printf() appear in a message, the order of these '%' directives may be changed by translation. In such a case, the translator can specify the order. See section of 'Special Comments preceding Keywords' in info page of gettext for detail.

Now there are projects to translate messages in various softwares. For example, Translation Project (http://www.iro.umontreal.ca/~pinard/po/HTML/).

11.1.1 Gettext-ization of A Software

At first, the software has to have the following lines.

```
int main(int argc, char **argv)
{
        ...
        setlocale (LC_ALL, "");    /* This is not for gettext but
                                      all i18n software should have
                                      this line. */
        bindtextdomain (PACKAGE, LOCALEDIR);
        textdomain (PACKAGE);
        ...
}
```

where *PACKAGE* is the name of the catalog file and *LOCALEDIR* is "/usr/share/locale" for Debian. *PACKAGE* and *LOCALEDIR* should be defined in a header file or Makefile.

It is convenient to prepare the following header file.

```
#include <libintl.h>
#define _(String) gettext((String))
```

and messages in source files should be written as _("message"), instead of "message".

Next, catalog files have to be prepared.

At first, a template for catalog file is prepared using xgettext. At default a template file message.po is prepared. [1]

11.1.2 Translation

Though gettextization of a software is a temporal work, translation is a continuing work because you have to translate new (or modified) messages when (or before) a new version of the software is released.

11.2 Readline Library

***** Not written yet *****

Readline library need to be internationalized.

[1] I HAVE TO WRITE EXPLANATION.

11.3 Ncurses Library

***** Not written yet *****

Ncurses is a free implementation of curses library. Though this library is now maintained by Free Software Foundation, it is not covered by GNU General Public License.

Ncurses library need to be internationalized.

Chapter 12

Softwares Written in Other than C/C++

Though C and C++ was, is, and will be the main language for software development for UNIX-like platforms, other languages, especially scripting languages, are often used.

Generally, languages other than C/C++ have less support for I18N then C/C++. However, nowadays other languages than C/C++ are coming to support Locale and Unicode.

12.1 Fortran

***** Not written yet *****

12.2 Pascal

***** Not written yet *****

12.3 Perl

Perl is one of the most important languages. Indeed, Debian system defines Perl as essential.

Perl 5.6 can handle UTF-8 characters. Declaration of `use utf8;` will enable it. For example, `length()` will return the number of characters, not the number of bytes.

However, it does not work well for me... why?

***** Not written yet *****

12.4 Python

***** Not written yet *****

12.5 Ruby

***** Not written yet *****

12.6 Tcl/Tk

***** Not written yet *****

Tcl/Tk is already internationalized. It is locale-sensible. It automatically uses proper font for various characters. Though it uses UTF-8 as internal encoding, users of Tcl/Tk don't have to aware of it. This is because Tcl/Tk converts encodings.

12.7 Java

Full internationalization is naturally lead from Java's "Write Once, Run Anywhere" principle. To achieve this, Java uses Unicode as internal code for `char` and `String`. It is important that Unicode is *internal* code. Java obeys the current LOCALE and encoding is automatically converted for I/O. Thus, *users* of applications written in Java doesn't need to be aware of Unicode.

Then how about *developers*? They also don't need to be aware of the internal encoding. Character processings such as counting of number of characers in a string work well. And more, you don't have to worry about display/input.

However, you may want to handle specified encodings for, for example, MIME encoding/decoding. For such purposes, I/O can be done by specifying external encoding. Check `InputStreamReader` and `OutputStreamReader` classes. You can also convert between the internal encoding and specified encodings by `String.getBytes(`*encoding*`)` and `String(byte []` *bytes*`,`*encoding*`)`.

12.8 Shell Script

***** Not written yet *****

12.9 Lisp

***** Not written yet *****

Chapter 13

Examples of I18N

Programmers who have internationalized softwares, have written a patch of L10N, and so on are encouraged to contribute to this chapter.

13.1 TWM – usage of XFontSet instead of XFontStruct

The author of this section is Tomohiro KUBOTA (<kubota@debian.org>).

13.1.1 Introduction

TWM is Tabbed (or Tom's) Window Manager, one of the most well-known window managers in the world. It is included in the XFree86 distribution. Since it was not internationalized, I wrote a patch for TWM included in XFree86 version 4.0.1. The patch was adopted in XFree86 version 4.0.1d.

Note: a bug is found for `I18N_FetchName()` and `I18N_GetIconName()` of my patch. The bug is fixed since XFree86 version 4.1.0. This document is also fixed.

The contents of the internationalization are:

- Usage of `XFontSet`-related functions instead of `XFontStruct`, so that font handling will be locale-sensible. This is the main part of the patch.
- Addition of automatic font guessing mechanism (the simplest version). This avoids lack of font caused by ISO8859-1-based font specification in configuration files.
- Usage of `XGetWMName()` and `XmbTextPropertyToTextList()` instead of `XFetchName()`, so that Compound Text can be used for inter-client communication of window title names. This enables TWM to properly receive the internationalized window text names from X clients.

- Usage of `XGetWMIconName()` and `XmbTextPropertyToTextList()` for inter-client communication of window icon names. This enables TWM to properly receive the internationalized window icon names from X clients.
- 8bit-cleanization of the configuration file parser. This enables usage of internationalized texts for menus and so on.

The following will present these items.

13.1.2 Locale Setting - A Routine Work

At first, I added a small part to call `setlocale()` at the beginning of `main()` function.

```
loc = setlocale(LC_ALL, "");
if (!loc || !strcmp(loc, "C") || !strcmp(loc, "POSIX") ||
   !XSupportsLocale()) {
    use_fontset = False;
} else {
    use_fontset = True;
}
```

`loc` is `char *`-type auto (local) variable. `use_fontset` is `Bool`-type global variable, for which I wrote a declaration in `twm.h`.

```
extern Bool use_fontset;
```

I also added inclusion of `X11/Xlocale.h` header file. By including of this header file, locale feature of X11 will be used when compiled in OS without locale features. Otherwise, X11/Xlocale will use locale features of the OS. Thus, you can include `X11/Xlocale.h` regardless of whether the OS support locale.

Checking of NULL, "C", and "POSIX" locales will enable TWM to work 8bit through when the user does not configure locale properly. Under "C" or "POSIX" locale, or without proper configuration of locale, `XFontSet`-related functions will work under 7bit ASCII encoding and these functions will regard all 8bit characters as invalid. In such cases, my patch won't use `XFontSet`-related functions by checking the value of `use_fontset`. Checking of `XSupportLocale()` is needed for cases when the OS support the locale while X doesn't support the locale.

13.1.3 Font Preparation

Almost functions related to `XFontStruct` can be easily substituted by `XFontSet`-related functions.

Fortunately, TWM used a tailored `MyFont` type for font handling. Thus the amount of labor was decreased. The original `MyFont` definition was:

```
typedef struct MyFont
{
    char *name;                 /* name of the font */
    XFontStruct *font;          /* font structure */
    int height;                 /* height of the font */
    int y;                      /* Y coordinate to draw characters */
} MyFont;
```

I added a few lines.

```
typedef struct MyFont
{
    char *name;                 /* name of the font */
    XFontStruct *font;          /* font structure */
    XFontSet fontset;           /* fontset structure */
    int height;                 /* height of the font */
    int y;                      /* Y coordinate to draw characters */
    int ascent;
    int descent;
} MyFont;
```

Then one of the main part of this patch – font preparation. The font preparation is done in the `GetFont()` function in `util.c`. This function is almost entirely rewritten.

```
void
GetFont(font)
MyFont *font;
{
    char *deffontname = "fixed";
    char **missing_charset_list_return;
    int missing_charset_count_return;
    char *def_string_return;
    XFontSetExtents *font_extents;
    XFontStruct **xfonts;
    char **font_names;
    register int i;
    int ascent;
    int descent;
    int fnum;
    char *basename2;

    if (use_fontset) {
        if (font->fontset != NULL){
            XFreeFontSet(dpy, font->fontset);
        }
```

```
    basename2 = (char *)malloc(strlen(font->name) + 3);
    if (basename2) sprintf(basename2, "%s,*", font->name);
    else basename2 = font->name;
    if( (font->fontset = XCreateFontSet(dpy, basename2,
                                        &missing_charset_list_return,
                                        &missing_charset_count_return,
                                        &def_string_return)) == NULL) {
        fprintf (stderr, "%s:  unable to open fontset \"%s\"\n",
                    ProgramName, font->name);
        exit(1);
    }
    if (basename2 != font->name) free(basename2);
    for(i=0; i<missing_charset_count_return; i++){
        printf("%s: warning: font for charset %s is lacking.\n",
               ProgramName, missing_charset_list_return[i]);
    }

    font_extents = XExtentsOfFontSet(font->fontset);
    fnum = XFontsOfFontSet(font->fontset, &xfonts, &font_names);
    for( i = 0, ascent = 0, descent = 0; i<fnum; i++){
        if (ascent < (*xfonts)->ascent) ascent = (*xfonts)->ascent;
        if (descent < (*xfonts)->descent) descent = (*xfonts)->descent;
        xfonts++;
    }
    font->height = font_extents->max_logical_extent.height;
    font->y = ascent;
    font->ascent = ascent;
    font->descent = descent;
    return;
}

if (font->font != NULL)
    XFreeFont(dpy, font->font);

if ((font->font = XLoadQueryFont(dpy, font->name)) == NULL)
{
    if (Scr->DefaultFont.name) {
        deffontname = Scr->DefaultFont.name;
    }
    if ((font->font = XLoadQueryFont(dpy, deffontname)) == NULL)
    {
        fprintf (stderr, "%s:  unable to open fonts \"%s\" or \"%s\"\n",
                 ProgramName, font->name, deffontname);
        exit(1);
    }
```

```
        }
        font->height = font->font->ascent + font->font->descent;
        font->y = font->font->ascent;
        font->ascent = font->font->ascent;
        font->descent = font->font->descent;
    }
```

This function can be divided into two large parts by `if (use_fontset)`. The part inside the `if` is for internationalized version and other part is for conventional version. Conventional version is used when `use_fontset` is false, as you can see. This part is almost the same as the original TWM.

Now let's study the internationalized part of `GetFont()`. It is convenient to compare the internationalized part and conventional part, to study it. The first check and `XFreeFontSet()` is a replacement of `XFreeFont()`. The next several lines is the *automatic font guessing mechanism (the simplest version)*, the second item of the whole patch. It only adds ",*" to the font query string. Then the added string is passed into `XCreateFontSet()`, the key function of font preparation.

13.1.4 Automatic Font Guessing

Let's imagine how this ",*" works. Assume `ja_JP.eucJP` locale, where EUC-JP encoding is used. In EUC-JP encoding, three fonts of

- a font with *charset* (in XLFD meaning) of ISO8859-1 or JISX0201.1976-0,
- a font with *charset* of JISX0208.1983-0 or JISX0208.1990-0, and
- a font with *charset* of JISX0201.1976-0

are used. [1] Again assume that `GetFont` received a string of "-adobe-helvetica-bold-r-normal–*-120-*-*-*-*-*-*" as `font->name`. This string is a very likely specification of font. Actually, I got the example from the default title font for TWM. Now review the behavior of `XLoadQueryFont()`. Since it always gets at most one font, it can succeed or fail. However, since `XCreateFontSet()` may get multiple fonts, it may success only to get a part of the set of requred fonts. The assumed calling of `XCreateFontSet()` with the `font->name` in `ja_JP.eucJP` locale goes into just such a situation. For usual systems, only a font for ISO8859-1 or JISX0201.1976-0 is available. [2] It is easy to solve this situation. Unlike `XLoadQueryFont()`, `XCreateFontSet()` can take a *list of patterns* of fonts with wildcards. `XCreateFontSet()` chooses necessary fonts from the set of fonts which match the patterns. "*" can match all fonts. This works for character sets for which the given `font->name` failed to match any fonts.

There were two solutions I imagined.

[1] Read `/usr/X11R6/lib/X11/locale/ja/XLC_LOCALE` for detail.
[2] In such a case, `XCreateFontSet()` does not fail. Instead, it returns informations on missing fonts.

- Adding ",*" for all font specifications in the configuration file.
- Adding ",*" just before calling `XCreateFontSet()`. (the solution I took.)

The first solution may fail because users can rewrite the configuration file. Though it is likely that a user knows necessary character sets for the encoding (s)he uses, the second way is safer. And more, recent window managers are coming to support *themes* where a set of configuration is packaged and distributed, just as in `http://www.themes.org/`. It is very unlikely that all developers of these themes know this problem and adds ",*" for every font specifications. Thus, window managers which support themes must take the 2nd solution, though TWM does not support themes.

Which font exactly is choosed for wild cards? It depends on the configuration of X Window System. I imagine that the first font in the list generated by `xlsfonts`. You may think the choice of the font should be cleverer. It would be adequate to say that ",*" mechanism is not less cleverer; it has entirely no intelligence. It is not clever at all. Yes, though I didn't implement it to TWM, I also wrote a cleverer guessing mechanism. [3]

13.1.5 Font Preparation (continued)

After calling `XCreateFontSet()`, `GetFont()` builds a few member variables of `MyFont`, i.e., `font->height`, `font->y`, `font->ascent`, and `font->descent`. These parameters are easily get from members of `XFontStruct` structure and are actually often used in TWM. Thus I had to prepare substitutions for `XFontSet` version. These variables also build for `XFontStruct` version so that a united method can be used to get these parameters.

13.1.6 Drawing Text using `MyFont`

To draw a text, `XDrawString()` and `XDrawImageString()` are used for conventional `XFontStruct`. On the other hand, `XmbDrawString()`/`XwcDrawString()` and `XmbDrawImageString()`/`XwcDrawImageString()` are used for internationalized `XFontSet`. The difference between `mb` and `wc` versions are whether the text is given in *multibyte characters* or in *wide characters*. Since TWM does not perform any text processing, I didn't use wide characters and treat strings as they are (in multibyte characters).

TWM has many calls of these functions. Thus I decided to write wrappers which checks `use_fontset` and calls proper version of X function. They are `MyFont_DrawString()` and `MyFont_DrawImageString()`. Thus all calling of `XDrawString()` and `XDrawImageString()` are replaced with the wrappers. Since these two are almost identical, I will explain one of them.

[3] I implemented cleverer mechanism to window managers such as Blackbox, Sawfish, and so on where I think beauty is important than simplicity. The intended algorithm is:

- Choose a font with similar pixel sizes.
- If availavle, choose a font with similar weight and slant.

```
void
MyFont_DrawString(dpy, d, font, gc, x, y, string, len)
    Display *dpy;
    Drawable d;
    MyFont *font;
    GC gc;
    int x,y;
    char *string;
    int len;
{
    if (use_fontset) {
       XmbDrawString(dpy, d, font->fontset, gc, x, y, string, len);
       return;
    }
    XDrawString (dpy, d, gc, x, y, string, len);
}
```

Very simple function! However note that the required paramaters are different in these two functions of conventional version and internationalized version. Font is needed for internationalized version.

Then, is GC not used for specifying a font for internationalized version? Right. This causes to increase the labor. The original version of TWM use a macro of `FBF` to set up the GC. Fortunately, font specification is always performed just before the drawing of the texts. I wrote a function `MyFont_ChangeGC()` for substitution.

```
void
MyFont_ChangeGC(fix_fore, fix_back, fix_font)
    unsigned long fix_fore, fix_back;
    MyFont *fix_font;
{
    Gcv.foreground = fix_fore;
    Gcv.background = fix_back;
    if (use_fontset) {
       XChangeGC(dpy, Scr->NormalGC, GCForeground|GCBackground, &Gcv);
       return;
    }
    Gcv.font = fix_font->font->fid;
    XChangeGC(dpy, Scr->NormalGC, GCFont|GCForeground|GCBackground,&Gcv);
}
```

You may wonder why this is needed. You may think just do as `use_fontset` is false and it will work well. No, because `fix_font->font` is indefinite.

I had to modify one more part related to GC in `gc.c`.

13.1.7 Geting Size of Texts

TWM calls `XTextWidth()` many times. It returns the width in pixels for a text. The internationalized version of the function is `XmbTextExtent()` and `XwcTextExtent()`, where the difference between `mb` version and `wc` version is same as `XmbDrawString()` and so on.

I wrote a wrapper, as I did for other functions.

13.1.8 Getting Window Titles

General discussions have finished. The following discussions are specific to window managers.

Window managers have to get the names for window titles from X clients. `XFetchName()` is the function for this purpose.

Window title names are communicated using **property** mechanism of X. `XA_STRING` and `XA_COMPOUND_TEXT` are types to be used for this purpose. `XA_STRING` means the text data is in ISO8859-1 encoding and `XA_COMPOUND_TEXT` means the data is in compound text. Compound text is a subset of ISO 2022 and can handle international text data.

Now, `XFetchName()` can handle `XA_STRING` type only. Thus we should use `XGetWMName()`. Since handling of compound text needs several lines of source codes, I wrote a wrapper function.

```
/*
 * The following functions are internationalized substitutions
 * for XFetchName and XGetIconName using XGetWMName and
 * XGetWMIconName.
 *
 * Please note that the third arguments have to be freed using free(),
 * not XFree().
 */
Status
I18N_FetchName(dpy, w, winname)
    Display *dpy;
    Window w;
    char ** winname;
{
    int    status;
    XTextProperty text_prop;
    char **list;
    int    num;

    status = XGetWMName(dpy, w, &text_prop);
    if (!status || !text_prop.value || !text_prop.nitems) return 0;
    status = XmbTextPropertyToTextList(dpy, &text_prop, &list, &num);
```

```
    if (status < Success || !num || !*list) return 0;
    XFree(text_prop.value);
    *winname = (char *)strdup(*list);
    XFreeStringList(list);
    return 1;
}
```

13.1.9 Getting Icon Names

Window managers need to get not only window titles but also icon names.

TWM used `XGetWindowProperty()` with `XA_STRING` to get icon names. However, internationalized function `XGetWMIconName()` is available for this purpose and I rewrote using this function. Just like `XGetWMName()`, I wrote a wrapper.

```
Status
I18N_GetIconName(dpy, w, iconname)
    Display *dpy;
    Window w;
    char ** iconname;
{
    int    status;
    XTextProperty text_prop;
    char **list;
    int    num;

    status = XGetWMIconName(dpy, w, &text_prop);
    if (!status || !text_prop.value || !text_prop.nitems) return 0;
    status = XmbTextPropertyToTextList(dpy, &text_prop, &list, &num);
    if (status < Success || !num || !*list) return 0;
    XFree(text_prop.value);
    *iconname = (char *)strdup(*list);
    XFreeStringList(list);
    return 1;
}
```

13.1.10 Configuration File Parser

The parser for configuration file was not 8bit clean. I modified it. It was a very minor change. In `parse.c`, global variables of `buff[]`, `overflowbuff[]`, `stringListSource`, and `currentString` and auto variable of `sl` in `ParseStringList()` are changed from `char` to `unsigned char`.

13.2 8bit-clean-ize of Minicom

The author of this section is Tomohiro KUBOTA (<kubota@debian.org>).

I needed a serial communication software to connect to BBS, though I had a MS-DOS version. I tried several softwares and found Minicom but it could not display Japanese characters in kterm. Thus I decided to modify the source of Minicom. Though it was dirty 'quick hacking', I sent the patch to the upstream developer.

13.2.1 8bit-clean-ize

Minicom is written in C.

At first I explore the source code to find the way for characters to be displayed. I found that it implements a 'ncurses'-like functions.

Since Minicom is used for BBS it seemed to have a conversion table so as to IBM-PC graphics characters (I guess) can be displayed correctly. I made an another pass for characters to go without any modification and added a new command option to activate the pass.

13.2.2 Not to break continuity of multibyte characters

The 'ncurses'-like functions in Minicom outputs location code every time a character is outputted. This breaks continuity of multibyte characters.

13.2.3 Catalog in EUC-JP and SHIFT-JIS

13.3 user-ja – two sets of messages in ASCII and native codeset in the same language

The author of this section is Tomohiro KUBOTA (<kubota@debian.or.jp>).

13.3.1 Introduction

`user-ja` is a Debian-specific software which establishes basic settings for Japanese-speaking beginners. `user-ja` does not automatically establishes the settings. A user who needs Japanese environment has to invoke `user-ja-conf`.

Since `user-ja-conf` is a software to establish Japanese environment, the environment where user-ja runs may be poor Japanese environment. For example, `user-ja-conf` must not assume that Japanese character can be displayed. However, Japanese character should be used in environments where it is possible.

`user-ja` is a simple example which switches two sets of messages, one is written using ASCII characters and the other Japanese characters. Note that both of them are written in Japanese language. This is beyond what gettext can do.

Though `user-ja` is a Japanese-specific software, this problem of ability to display non-ASCII character is common to non-ASCII languages.

13.3.2 Strategy

The following environments can display Japanese characters: `kon` (Kanji Console), `kterm`, and `krxvt` (in `rxvt-ml` package). And more, telnet softwares for Windows and so on may be able to display Japanese characters.

At first, `user-ja-conf` detects the environment. If it can display Japanese characters, go ahead. If not, try to establish a new environment and invoke itself in it. If detection is failed, display Japanese characters and ask the user whether he/she can read it.

13.3.3 Implementation

`user-ja-conf` is a perl script. Here shows a function which check whether Japanese native characters can be displayed or not and try to establish an environment where native characters can be displayed, if not.

```
sub isNC($$)
{
        my ($KANJI, $TTY, $TERM, $DISPLAY, $WHICH);
        $TTY = `/usr/bin/tty`;
        $TERM = $ENV{TERM};
        $DISPLAY = $ENV{DISPLAY};
        $WHICH = '/usr/bin/which';
        $THIS = $_[0];
        $OPT = $_[1];

        if ($TERM eq 'kon' || $TERM eq 'kterm') {
                $KANJI=1;
        } elsif ($DISPLAY ne '' && system("$WHICH kterm >/dev/null")==0) {
                exec("kterm -km euc -e $THIS $OPT");
        } elsif ($DISPLAY ne '' && system("$WHICH krxvt >/dev/null")==0) {
                exec("krxvt -km euc -e $THIS $OPT");
        } else {
                print STDERR &sourceset2displayset(
                  "Japanese sentence in Japanese characters 'Can you read thi
                print STDERR
                  "Japanese sentence in ASCII characters 'Can you read the ab
                $a = <>;
```

```
                if ($a =~ /y|Y/) {
                        $KANJI=1;
                } elsif ($TTY =~ m#/dev/tty[0-9]+#) {
                        print STDERR
                          "Japanese sentence in ASCII characters 'Shall I inv
                        $a = <>;
                        exec("kon -e $THIS $OPT") if ($a !~ /n|N/);
                        $KANJI=0;
                } else {
                        $KANJI=0;
                }
        }
        $KANJI;
}
```

`&sourceset2displayset($)` is a function to convert a string from codeset for source code into codeset for display. This is needed because codeset for program source (in this case, perl script) and dotfiles may be different. [4]

The following function is prepared to display messages in appropriate codeset. Don't care '`Lang::`' package.

```
sub disp ($$) {
        if ($NC) {print STDERR &Lang::sourceset2displayset($_[1]);}
        else {print STDERR $_[0];}
}
```

This is an example how the `disp` function is used.

```
sub disp_finish()
{
        &Sub::disp(<<EOF1,<<EOF2);

[Enter] key WO OSUTO KONO user-ja-conf HA SYUURYOU SHIMASU.
EOF1

Japanese sentence in Japanese characters 'Push [Enter] key to finish.'
EOF2
}
```

[4] There are three popular codesets for Japanese — ISO-2022-JP, EUC-JP, and SHIFT-JIS. EUC-JP should be used for perl source code because all non-ASCII characters in EUC-JP do not have values in 0x21 - 0x7e. However, ISO-2022-JP is the safest codeset to display because EUC-JP and SHIFT-JIS have to be used exclusively. However, ISO-2022-JP is the most difficult codeset to implement and there may be a terminal environment which does not understand ISO-2022-JP (for example, Minicom). On the other hand, dotfiles may be written in any codesets, according to one's favorite and purpose.

Here the sentence '[Enter] key WO OSUTO...' is the Latin alphabet expression of Japanese.

Thus almost all messages are duplicated using `disp` function.

13.4 A Quasi-Wrapper to Internationalize Text Output of X Clients

The author of this section is Tomohiro KUBOTA (<kubota@debian.or.jp>).

13.4.1 Introduction

X11 supplies `XFontSet`-related internationalized functions for text output. However, many X clients use conventional `XFontStruct`-related non-internationalized functions and cannot output languages which need multiple fonts (Chinese, Japanese, and Korean).

Now I introduce a wrapper which easily modify non-internationalized X clients to use internationalized X11 functions.

13.4.2 Strategy

Almost `XFontStruct`-related functions can be replaced easily by `XFontSet`-related functions.

- XFontStruct -> XFontSet
- XLoadQueryFont() -> XCreateFontSet()
- XFreeFont() -> XFreeFontSet()
- XDrawString() -> XmbDrawString()
- XDrawImageString() -> XmbImageString()
- XTextExtents() -> XmbTextExtents()
- XTextWidth() -> XmbTextEscapement()

However, there were several problems.

1 The font for `XDrawString` is specified by `GC` parameter, while `XFontSet` parameter is used for `XmbDrawString`.

2 `XFontSet` does not have structure members of `ascent` and `descent`, while `XFontStruct` has them and these members are often referred.

3 Many software specify font name with iso8859-1-specific way. This avoids 'fontset'-related functions work fully and disables non-iso8859-1 languages be displayed.

Though it is possible to solve the first problem, this problem may make the wrapper very complex. Thus, I decided to modify the original source and leave the wrapper simple, instead of writing a complete wrapper. However, if `XGCValues.font` is set and `XCreateGC()` is called, it is needed to avoid `XCreateGC` to fail because of null font specification. Thus I wrote a wrapper of `XCreateGC`.

To solve the second problem, I wrote a wrapper of `XFontSet` which has structure members of `ascent` and `descent`. Thus all wrapper functions are related to this wrapper structure.

To solve the third problem, I wrote a wrapper of `XCreateFontSet()`. This part can be used for many X clients which are already internationalized using fontset-related functions, because these softwares have the same problem. Explanation on this problem and solution will be supplied in other section.

13.4.3 Usage of the wrapper

Replace the following structure and functions. You can use replacement faculty of your text editor.

- XFontStruct -> QuasiXFontStruct
- XLoadQueryFont() -> QuasiXLoadQueryFont()
- XFreeFont() -> QuasiXFreeFont()
- XTextExtents() -> QuasiXTextExtents()
- XTextWidth() -> QuasiXTextWidth()
- XGetWMIconName() -> QuasiXGetWMIconName()
- XGetWMName() -> QuasiXGetWMName()
- XFetchName() -> QuasiXFetchName()
- XGetIconName() -> QuasiXGetIconName()
- XChangeGC() -> QuasiXChangeGC()
- XCreateGC() -> QuasiXCreateGC()

The following two wrapper functions need an additional parameter.

- XDrawString(Display *d, Window w, GC gc, int x, int y, const char *string, int len) -> QuasiXDrawString(Display *d, Window w, *QuasiXFontStruct *q*, GC gc, int x, int y, char *string, int len)
- XDrawImageString(Display *d, Window w, GC gc, int x, int y, const char *string, int len) -> QuasiXDrawImageString(Display *d, Window w, *QuasiXFontStruct *q*, GC gc, int x, int y, char *string, int len)

13.4.4 The Header File of the Wrapper

This is the header file of the wrapper.

```
/* fontset.h */

#ifndef __fontset__h__
#define __fontset__h__

#include <X11/Xlocale.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>

typedef struct {
  Font     fid;  /* dummy */
  XFontSet fs;
  int      ascent, descent;
} QuasiXFontStruct;

#define FONT_ELEMENT_SIZE 50

QuasiXFontStruct *QuasiXLoadQueryFont(Display *d, const char *fontset_name);
void QuasiXFreeFont(Display *d, QuasiXFontStruct *qxfs);
void QuasiXDrawString(Display *d, Window w,
       QuasiXFontStruct *qxfs, GC gc,
       int x, int y, const char* s, int l);
void QuasiXDrawImageString(Display *d, Window w,
    QuasiXFontStruct *qxfs, GC gc,
    int x, int y, char* s, int l);
void QuasiXTextExtents(QuasiXFontStruct *qxfs, char *string, int nchars,
        int *direction_return, int *font_ascent_return,
        int *font_descent_return, XCharStruct *overall_return);
int QuasiXTextWidth(QuasiXFontStruct *qxfs, const char *s, int cnt);

Status QuasiXGetWMIconName(Display *d, Window w,
    XTextProperty *text_prop_return);
Status QuasiXGetWMName(Display *d, Window w,
        XTextProperty *text_prop_return);
Status QuasiXFetchName(Display *d, Window w, char **winname);
Status QuasiXGetIconName(Display *d, Window w,char **iconname);
GC QuasiXCreateGC(Display *d, Drawable dr, unsigned long mask, XGCValues *xgc
int QuasiXChangeGC(Display *d, GC gc, unsigned long mask, XGCValues * xgc);

#else /* !FONTSET */

#define QuasiXFontStruct    XFontStruct
```

```
#define QuasiXLoadQueryFont XLoadQueryFont
#define QuasiXFreeFont      XFreeFont
#define QuasiXTextExtents   XTextExtents
#define QuasiXTextWidth     XTextWidth
#define QuasiXGetWMIconName XGetWMIconName
#define QuasiXGetWMName     XGetWMName
#define QuasiXFetchName     XFetchName
#define QuasiXGetIconName   XGetIconName
#define QuasiXChangeGC      XChangeGC
#define QuasiXCreateGC      XCreateGC
#define QuasiXDrawString(d, w, qxfs, gc, x, y, s, l) \
        XDrawString(d, w, gc, x, y, s, l)
#define QuasiXDrawImageString(d, w, qxfs, gc, x, y, s, l) \
        XDrawImageString(d, w, gc, x, y, s, l)

#endif /* __fontset__h__ */
```

13.4.5 The Source File of the Wrapper

This is the source file of the wrapper.

```
/* fontset.c */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include "fontset.h"


static const char * i_strstr(const char *str, const char *ptn)
{
  const char *s2, *p2;
  for( ; *str; str++) {
    for(s2=str,p2=ptn; ; s2++,p2++) {
      if (!*p2) return str;
      if (toupper(*s2) != toupper(*p2)) break;
    }
  }
  return NULL;
}
```

```
static const char * Font_GetElement(const char *pattern, char *buf, int bufsi
{
  const char *p, *v;
  char *p2;
  va_list va;

  va_start(va, bufsiz);
  buf[bufsiz-1] = 0;
  buf[bufsiz-2] = '*';
  while((v = va_arg(va, char *)) != NULL) {
    p = i_strstr(pattern, v);
    if (p) {
      strncpy(buf, p+1, bufsiz-2);
      p2 = strchr(buf, '-');
      if (p2) *p2=0;
      va_end(va);
      return p;
    }
  }
  va_end(va);
  strncpy(buf, "*", bufsiz);
  return NULL;
}

static const char * Font_GetSize(const char *pattern, int *size)
{
  const char *p;
  const char *p2=NULL;
  int n=0;

  for (p=pattern; 1; p++) {
    if (!*p) {
      if (p2!=NULL && n>1 && n<72) {
 *size = n; return p2+1;
      } else {
 *size = 16; return NULL;
      }
    } else if (*p=='-') {
      if (n>1 && n<72 && p2!=NULL) {
 *size = n;
 return p2+1;
      }
      p2=p; n=0;
    } else if (*p>='0' && *p<='9' && p2!=NULL) {
      n *= 10;
      n += *p-'0';
```

```
    } else {
      p2=NULL; n=0;
    }
  }
}

static XFontSet XCreateFontSetWithGuess(Display *d, const char *pattern, char
{
  XFontSet fs;
  char *pattern2;
  int pixel_size, bufsiz;
  char weight[FONT_ELEMENT_SIZE], slant[FONT_ELEMENT_SIZE];

  /* No problem?  or 'fs' for pattern analysis */
  fs = XCreateFontSet(d, pattern, miss, n_miss, def);
  if (fs && !*n_miss) return fs; /* no need for font guessing */

  /* for non-iso8859-1 language and iso8859-1 specification */
  /* This 'fs' is only for pattern analysis. */
  if (!fs) {
    if (*n_miss) XFreeStringList(*miss);
    setlocale(LC_CTYPE, "C");
    fs = XCreateFontSet(d, pattern, miss, n_miss, def);
    setlocale(LC_CTYPE, "");
  }

  /* make XLFD font name for pattern analysis */
  if (fs) {
    XFontStruct **fontstructs;
    char **fontnames;
    XFontsOfFontSet(fs, &fontstructs, &fontnames);
    pattern = fontnames[0];
  }

  /* read elements of font name */
  Font_GetElement(pattern, weight, FONT_ELEMENT_SIZE,
   "-medium-", "-bold-", "-demibold-", "-regular-", NULL);
  Font_GetElement(pattern, slant, FONT_ELEMENT_SIZE,
   "-r-", "-i-", "-o-", "-ri-", "-ro-", NULL);
  Font_GetSize(pattern, &pixel_size);

  /* modify elements of font name to fit usual font names */
  if (!strcmp(weight, "*")) strncpy(weight, "medium", FONT_ELEMENT_SIZE);
  if (!strcmp(slant,  "*")) strncpy(slant,  "r",      FONT_ELEMENT_SIZE);
  if (pixel_size<3) pixel_size=3; else if (pixel_size>97) pixel_size=97;
```

```
  /* build font pattern for better matching for various charsets */
  bufsiz = strlen(pattern) + FONT_ELEMENT_SIZE*2 + 2*2 + 58;
  pattern2 = (char *)malloc(bufsiz);
  if (pattern2) {
    snprintf(pattern2, bufsiz-1, "%s,"
      "-*-*-%s-%s-*-*-%d-*-*-*-*-*-*-*,"
      "-*-*-*-*-*-*-%d-*-*-*-*-*-*-*,*",
      pattern,
      weight, slant, pixel_size,
      pixel_size);
    pattern = pattern2;
  }
  if (*n_miss) XFreeStringList(*miss);
  if (fs) XFreeFontSet(d, fs);

  /* create fontset */
  fs = XCreateFontSet(d, pattern, miss, n_miss, def);
  if (pattern2) free(pattern2);
  return fs;
}

QuasiXFontStruct *QuasiXLoadQueryFont(Display *d, const char *fontset_name)
{
  char **miss, *def, *pattern;
  int n_miss;
  XFontSet fontset;
  QuasiXFontStruct *wxfs;
  int pixel_size=16, bufsiz;
  char family[FONT_ELEMENT_SIZE], weight[FONT_ELEMENT_SIZE],
       slant[FONT_ELEMENT_SIZE];

  wxfs = (QuasiXFontStruct *)malloc(sizeof(QuasiXFontStruct));
  if (!wxfs) return NULL;

  /* create fontset */
  fontset = XCreateFontSetWithGuess(d, fontset_name, &miss, &n_miss, &def);
  if (!fontset) {free(wxfs); return NULL;}

  if (n_miss) {
    int j;
    fprintf(stderr,
     "QuasiXLoadQueryFont: lacks the font(s) for the following charset(s)\n")
    for (j=0; j<n_miss; j++) {
      fprintf(stderr, "  %s\n", miss[j]);
    }
    XFreeStringList(miss);
```

```
  }
  /* emulating XFontStruct */
  wxfs->fs      = fontset;
  wxfs->ascent  =-XExtentsOfFontSet(fontset)->max_logical_extent.y;
  wxfs->descent = XExtentsOfFontSet(fontset)->max_logical_extent.height
                 +XExtentsOfFontSet(fontset)->max_logical_extent.y;
  return wxfs;
}

void QuasiXFreeFont(Display *d, QuasiXFontStruct *wxfs)
{
  if (!wxfs) return;
  XFreeFontSet(d, wxfs->fs);
  free(wxfs);
}

void QuasiXDrawString(Display *d, Window w, QuasiXFontStruct *qxfs, GC gc,
      int x, int y, const char* s, int l)
{
  XmbDrawString(d, w, qxfs->fs, gc, x, y, s, l);
}

void QuasiXDrawImageString(Display *d, Window w, QuasiXFontStruct *qxfs, GC g
    int x, int y, char* s, int l)
{
  XmbDrawImageString(d, w, qxfs->fs, gc, x, y, s, l);
}

void QuasiXTextExtents(QuasiXFontStruct *qxfs, char *string, int nchars,
        int *direction_return, int *font_ascent_return,
        int *font_descent_return, XCharStruct *overall_return)
{
  XRectangle overall_ink_return;
  XRectangle overall_logical_return;

  XmbTextExtents(qxfs->fs, string, nchars,
  &overall_ink_return, &overall_logical_return );

  *font_ascent_return  = -overall_logical_return.y;
  *font_descent_return =  overall_logical_return.height
                         +overall_logical_return.y;
  *direction_return    = FontLeftToRight; /* unsupported */
  *overall_return; /* unsupported */
}

int QuasiXTextWidth(QuasiXFontStruct *wxfs, const char *s, int cnt)
```

```
{
  return XmbTextEscapement(wxfs->fs, s, cnt);
}

Status QuasiXGetWMName(Display *d, Window w, XTextProperty *text_prop_return)
{
  int status;
  char **list;
  int num;

  status = XGetWMName(d, w, text_prop_return);

  if (!status || !text_prop_return->value || text_prop_return->nitems <= 0) {
    return 0; /* failure */
  }
  if (text_prop_return->encoding == XA_STRING) return 1;
  text_prop_return->nitems = strlen((char *)text_prop_return->value);
  status = XmbTextPropertyToTextList(d, text_prop_return,
      &list, &num);
  if (status >= Success && num > 0 && *list) {
    XFree(text_prop_return->value);
    text_prop_return->value  = (unsigned char*)strdup(*list);
    text_prop_return->nitems = strlen(*list);
    XFreeStringList(list);
    return 1; /* success */
  }
  return 0;
}

Status QuasiXGetWMIconName(Display *d, Window w,
    XTextProperty *text_prop_return)
{
  int status;
  char **list;
  int num;

  status = XGetWMIconName(d, w, text_prop_return);

  if (!status || !text_prop_return->value || text_prop_return->nitems <= 0) {
    return 0;
  }
  if (text_prop_return->encoding == XA_STRING) return 1;
  text_prop_return->nitems = strlen((char *)text_prop_return->value);
  status = XmbTextPropertyToTextList(d, text_prop_return,
      &list, &num);
  if (status >= Success && num > 0 && *list) {
```

```
    XFree(text_prop_return->value);
    text_prop_return->value  = (unsigned char*)strdup(*list);
    text_prop_return->nitems = strlen(*list);
    XFreeStringList(list);
    return 1;
  }
  return 0;
}

Status QuasiXGetIconName(Display *d, Window w, char **iconname)
{
  XTextProperty text_prop;
  char **list;
  int num;

  if (QuasiXGetWMIconName(d, w, &text_prop) != 0) {
    if (text_prop.value && text_prop.nitems) {
      *iconname = (char *)text_prop.value;
      return 1;
    }
  }
  *iconname = NULL;
  return 0;
}

Status QuasiXFetchName(Display *d, Window w, char ** winname)
{
  XTextProperty text_prop;
  char **list;
  int num;

  if (QuasiXGetWMName(d, w, &text_prop) != 0) {
    if (text_prop.value && text_prop.nitems > 0) {
      *winname = (char *)text_prop.value;
      return 1;
    }
  }
  *winname = NULL;
  return 0;
}

GC QuasiXCreateGC(Display *d, Drawable dr, unsigned long mask, XGCValues *xgc
{
  return XCreateGC(d, dr, mask & ~GCFont, xgc);
}
```

```
int QuasiXChangeGC(Display *d, GC gc, unsigned long mask, XGCValues * xgc)
{
  return XChangeGC(d, gc, mask & ~GCFont, xgc);
}
```

Chapter 14

References

General

- Unicode support in the Solaris Operating Environment (http://docs.sun.com/ab2/coll.651.1/SOLUNICOSUPPT) shows what is needed for software developers to support UTF-8.
- The Open Group's summary of ISO C Amendment 1 (http://www.unix-systems.org/version2/whatsnew/login_mse.html) is a detailed explanation on locale and wide character technologies.
- Markus Kuhn's UTF-8 and Unicode FAQ for Unix/Linux (http://www.cl.cam.ac.uk/~mgk25/unicode.html) is a detailed explanation on UTF-8 and Unicode.
- Bruno Haible's Unicode HOWTO (ftp://ftp.ilog.fr/pub/Users/haible/utf8/Unicode-HOWTO.html)
- Tomohiro KUBOTA (original author of this Introduction to I18N), What is MOJIBAKE (http://www8.plala.or.jp/tkubota1/mojibake/) shows what occurs when character handling is improper. Mojibake is a Japanese word which almost all computer users (not only Linux/BSD/Unix but also Windows/Macintosh) know.
- Ken Lunde, "CJKV Information Processing", ISBN 1-56592-224-7, O'Reilly, 1999
- Mikiko NISHIKIMI, Naoto TAKAHASHI, Satoru TOMURA, Ken'ichi HANDA, Seiji KUWARI, Shin'ichi MUKAIGAWA, and Tomoko YOSHIDA, "MARUCHIRINGARU KANKYOU NO JITSUGEN - X Window/Wnn/Mule/WWW BURAUZA DENO TAKOKUGO KANKYO (http://web.kyoto-inet.or.jp/people/tomoko-y/biwa/multi/)" or "Realization of Multilingual Environment - Multilingual Environment in X Window/Wnn/Mule/WWW Browser" (in Japanese), ISBN4-88735-020-1, TOPPAN, 1996
- Yoshihiro KIYOKANE and Youichi SUEHIRO, "KOKUSAIKA PUROGURAMINGU - I18N HANDOBUKKU (http://www.geocities.co.jp/

SiliconValley-PaloAlto/8090/)" or "Internationalization Programming - I18N Handbook" (in Japanese), ISBN4-320-02904-6, KYORITSU, 1998

- Syuuji SADO and Tomoko YOSHIDA, "Linux/FreeBSD NIHONGO KANKYOU NO KOUCHIKU TO KATSUYOU (http://web.kyoto-inet.or.jp/people/tomoko-y/japanese/index.html)" or "Construction and Utilization of Linux/FreeBSD Japanese Environment" (in Japanese), ISBN4-7973-0480-4, SOFTBANK, 1997
- Kouichi YASUOKA and Motoko YASUOKA "MOJI KOODO NO SEKAI (http://www.dendai.ac.jp/press/book_da/ISBN4-501-53060-X.html)" or "The World of Character Codes" (in Japanese), ISBN4-501-53060-X, Tokyo Denki University Press Center, 1999

Characters (general)

- Character Tables (http://www.kudpc.kyoto-u.ac.jp/~yasuoka/CJK.html) Graphic images for various character sets in the world.
- Ken Lunde's CJK info (ftp://ftp.ora.com/pub/examples/nutshell/ujip/doc/cjk.inf) information on CJK (Chinese, Japanese, and Korean) character set standards, written by the writer of "CJKV Information Processing" published by O'Reilly.
- IANA character set registry (http://www.isi.edu/in-notes/iana/assignments/character-sets) Note that both coded character sets (for example, KS_C_5601-1987, MIBenum 36) and encodings (for example, ISO-2022-KR, MIBenum: 37) are registered. How confusing!
- International Register of Coded Character Sets (http://www.itscj.ipsj.or.jp/ISO-IR/) A complete list of registered CCS, with ISO 2022 escape sequences. PDF files for these CCS are also available.

Characters (ISO 8859)

- ISO 8859 Alphabet Soup (http://czyborra.com/charsets/iso8859.html)

Characters (ISO 2022)

- http://www.ecma.ch/ecma1/stand/ECMA-035.HTM

Characters (ISO 10646 and Unicode)

- Unicode Consortium (http://www.unicode.org/)
- Problems and Solutions for Unicode and User/Vendor Defined Characters (http://www.opengroup.or.jp/jvc/cde/ucs-conv-e.html)

Softwares

- Arena-i18n (http://www.wg.omron.co.jp/~shin/Arena-CJK-doc/) Multilingual web browser.
- Mozilla (http://www.mozilla.org/) is also a multilingual web browser.
- Mule (http://www.m17n.org/mule/) Multilingual editor whose function is included in GNU Emacs 20 and XEmacs 20. Mule is the most advanced m17n software in my knowledge.
- JFBTERM (http://www3.justnet.ne.jp/~nmasu/linux/jfbterm/indexn.html) (in Japanese) is a multilingual terminal for Linux framebuffer console. Supported encodings are ISO 2022, EUC-JP, CN-GB, and EUC-KR. Supported CCS are ISO 8859-{1,2,3,4,5,6,7,8,9,10}, JISX 0201, JISX 0208, GB 2312, and KSX 1001.
- UNICON Project (http://www.gnu.org/directory/UNICON.html) intends to implement display/input CJK(Chinese/Japanese/Korean) characters under the Framebuffer under Linux.
- CCE - Chinese Console Environment (http://programmer.lib.sjtu.edu.cn/cce/cce.html) enables CN-GB Chinese to be displayed on Linux and FreeBSD console. It also supplies input methods for Chinese.
- Xterm (http://dickey.his.com/xterm/) is a part of XFree86 distribution. It can display UTF-8 encoding including doublewidth characters and combining characters.
- Rxvt (http://www.rxvt.org/) can display multibyte encodings such as EUC-JP, Shift-JIS, CN-GB, and Big-5.
- libiconv (http://www.gnu.org/software/libiconv/) provides `iconv()` implementation for systems which don't have one. It supports various encodings like ASCII, ISO 8859-*, KOI8-*, EUC-*, ISO 2022-*, Big5, Shift-JIS, TIS 620, UTF-*, UCS-*, CP*, Mac*, and so on. This library also has `locale_charset()`, a replacement of `nl_langinfo(CODESET)`.
- libutf8 - a Unicode/UTF-8 locale plugin (http://clisp.cons.org/~haible/packages-libutf8.html) provides UTF-8 locale support for systems which don't have UTF-8 locales.
- Pango (http://www.pango.org/) is a project to develop a portable high-quality text rendering engine.

Projects and Organizations

- Linux Internationalization Initiative (http://www.li18nux.org/), or Li18nux, focuses on the i18n of a core set of APIs and components of Linux distributions. The results will be proposed to LSB.

- LI18NUX 2000 Globalization Specification (http://www.li18nux.org/li18nux2k/) is the first fruits of Li18nux. focuses on the i18n of a core set of APIs and components of Linux distributions. The results will be proposed to LSB.
- Citrus Project (http://citrus.bsdclub.org/) is a project to implement locale/iconv for BSD series OSes so that these OSes conform to ISO C / SUSV2.
- Translation Project (http://www.iro.umontreal.ca/~pinard/po/HTML/)
- Mojikyo (http://www.mojikyo.org/)
- TRON project (http://www.tron.org/index-e.html)