Chapter 1

Internationalization

1.1 Introduction

This chapter explains basic concepts of software internationalization and interfaces under IRIX. After reading this chapter, you should be able to write internationalized applications and should understand issues relevant to the software internationalization and localization processes.

1.1.1 Disclaimer

This copy of what will eventually be the Internationalization chapter of the IRIX System Programming Guide is a DRAFT. It may be incomplete or inaccurate in some respects.

1.1.2 Definitions

Except for some general terms presented here, definitions are given in the text as needed.

Locale

The word *locale* is used in two ways. They are almost interchangeable.

When used to refer to part of the state of a program, *locale* means a global state variable maintained by the C library. This variable is described in detail in Section 1.3, “Locale.”

'Locale' is also used to mean an environment. It refers to the set of local customs, including natural language, culture, character set and encoding, formatting and sorting rules, and more. This environment is a *locale*, and it is selected by the variable *locale*.

Internationalization (I18n)

Note: "Internationalization" contains 20 letters: an I followed by 18 letters followed by an 'n'. It is commonly abbreviated "i18n." "I18n" is pronounced "internationalization," *not* "eye-eighteen-enn."

A program written for a specific locale may be difficult to run in a different environment. Porting such a program to operate in each desired environment would be tedious and costly.

The goal of the developer, then, should be to write programs which make no assumptions about language, local customs or coded character set. Such programs are said to be *internationalized*. Internationalized applications can run in a user's native environment following native conventions with native messages, without recompiling or relinking. Internationalization thus requires only a single copy of the software for a world of different users: it results in software that is *locale-independent*.

Localization (L10n)

Localization is the act of providing a locale-independent application with the environment and data it needs in order to operate in a particular locale. For example, adding German system messages to IRIX is a part of localizing IRIX for the German locale.

Note: Localization is often abbreviated "l10n."

Nationalized software

Nationalized programs run in one language and are governed by one set of customs; in other words, in a nationalized program the locale is bound to the application. The application might not use ASCII or English, but so long as it is a single-language program it is nationalized, not internationalized. Most older UNIX programs can be thought of as being nationalized for the United

States. With the exception of this section, this document does not address nationalized software.

Consider two applications, *hello* and *bonjour*. *hello* always produces

```
Hello, world.
```

and *bonjour* always produces

```
Bon jour, tout le monde.
```

Neither *hello* nor *bonjour* are internationalized; they are both nationalized.

For both terminal and graphics programs, there are no special requirements for writing or porting nationalized applications. Terminal-based programs will work on suitable terminals, including internationalized terminal emulators. “Suitable” means that the terminal supports any necessary fonts and understands the encoding of the application output. Graphics programs simply do as they have always done. Applications using existing interfaces to operate in non-English or non-ASCII environments should continue to compile and run.

Multilingual software

*Multilingual*ism refers to a program being to deal with several different cultures at the same time. Examples are described in Section 1.3.3, “Multilingual support.”

1.1.3 Areas of Concern in Internationalizing Software

Few developers will have to pay attention to more than a few items described in this guide. Most will need to catalog their strings. Some will need to use library routines for character sorting or locale-dependent date, time, or number formatting. A few whose applications use the eighth bit of 8-bit characters inappropriately will need to stop doing that. The few applications which do arithmetic to manipulate characters will need to be cleaned up. Some Graphical User Interface (GUI) designers will have to spend just a little more time thinking. For the large majority of developers, there isn’t that much to do.

Here, we try to address everything which a developer might worry about. Feel free to skip past issues which are not germane to your applications.

1.2 Standards

IRIX 5.0 internationalization includes features of these standards:

- ANSI C and POSIX (ISO 9945-1): locale
- X/OPEN Portability Guide (XPG/3): message catalogs, interpretation of locale strings, and so on
- AT&T UNIX™ System V Release 4: MNLS message catalogs
- X11R5: Input methods, text rendering, resource files
- Motif 1.2 [[what about it?]]

1.3 Locale

An internationalized system is capable of presenting and receiving data understandably in a number of different formats, cultures, languages and codesets. An application must announce how it wants the system to behave. IRIX uses *locale* for that announcement.

Locale is a state variable, global per process, which is central to most internationalization interfaces. Locale governs the behavior of certain library routines as they are called.

1.3.1 Setting the current locale

Applications begin in the *C* locale, which is the system default and is typically American English. Applications should call *setlocale()* as soon as possible to get the process into the desired locale. The syntax for *setlocale()* is:

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

The call almost always looks like this:

```
if (setlocale(LC_ALL, "") == NULL)
    exit_with_error();
```

Details of the two parameters are given below.

Category

Applications need not perform every aspect of their work in the same locale. Although this approach is not recommended, an application could (for example) perform most of its activities in the English locale but use French sorting rules. You can use locale categories to do this kind of locale-mixing. (Mixing locale categories is not the same as multilingual support—see Section 1.3.3 for information about multilingual support.)

The *category* argument is a symbolic constant which tells *setlocale()* which items in a locale to set. Table 1-1 lists the available category choices:

Category	Affects
LC_ALL	Attempts to set all categories below
LC_COLLATE	Regular expressions, *strcoll()* and *strxfrm()*
LC_CTYPE	Regular expressions, ctype routines (such as *islower()*)
LC_MESSAGES[a]	gettxt(), *pfmt()*, and *nl_langinfo()*
LC_MONETARY	localeconv()
LC_NUMERIC	decimal-point character for formatted I/O and nonmonetary formatting information returned by *localeconv()*
LC_TIME	ascftime(), *cftime()*, *getdate()*, *strftime()*

Table 1-1 Locale Categories

a. LC_MESSAGES is supported by SVR4 but isn't required or mentioned by XPG3

Categories correspond to databases which contain relevant information for each defined locale. Location of these databases is described in Section 1.3.1, "Location of locale-specific data."

Locale

setlocale() attempts to set the locale of the specified category to the value of the *locale* parameter passed in. You should almost always pass the empty string as the *locale* parameter.

Nonempty strings in calls to *setlocale()*

- NULL **string**: specifying a locale value of NULL causes *setlocale()* to return the name of the current environment.
- **"C"**: specifying a locale value of the single-character string "C" requests the system default locale.
- **Nonempty strings**: specify a particular locale to be used. Since this overrides any user preferences, this should only be done with good reason.

The empty string

An empty string passed as the *locale* parameter is special. It specifies that the locale should be chosen based on environment variables. This is the way a user specifies a preferred locale, and that preference should almost always be honored. The variables are checked hierarchically, depending on category, as shown in Table 1-2; for instance, if the category is LC_COLLATE, an empty-string locale parameter indicates that the locale should be chosen based on the value of the environment variable LC_COLLATE or, if that value is undefined, the value of the environment variable LANG.

Category	First Environment Variable	Second Environment Variable
LC_COLLATE	LC_COLLATE	LANG
LC_CTYPE	LC_CTYPE	LANG
LC_MESSAGES	LC_MESSAGES	LANG
LC_MONETARY	LC_MONETARY	LANG
LC_NUMERIC	LC_NUMERIC	LANG
LC_TIME	LC_TIME	LANG

Table 1-2 Category Environment Variables

Specifying category LC_ALL attempts to set each category to the given locale, using the appropriate environment variables.

On success, setlocale returns the value of the category. If setlocale couldn't set category to the value requested, it returns NULL and does not change locale.

If no non-null environment variable is available, setlocale returns the current value.

Location of locale-specific data

Except for XPG3 message catalogs, locale-specific data, (that is, the "compiled" files with the LC_COLLATE, LC_MONETARY, and other information) are located in */usr/lib/locale/*locale/category, where *locale* and *category* are names of the locale and category, respectively. For example, the database for LC_COLLATE category of the French locale *fr* would be in */usr/lib/locale/fr/LC_COLLATE*.

There will probably be multiple locales symbolically linked to each other, usually in cases where a very specific locale name points to the more general case. For example, */usr/lib/locale/En_US.ascii* might point to */usr/lib/locale/C*.

1.3.2 Locale naming conventions

A locale string is of the form:

[language[_territory[.encoding]][@modifier]

Where:

- *language* is the two letter ISO 639 abbreviation for the language name
- *territory* is the two-uppercase-letter ISO 3166 abbreviation for the territory name.
- *encoding* is the name of the character encoding (mapping between numbers and characters). For western languages, this is typically the codeset, such as 8859-1 or ASCII. For Asian languages, where an encoding may encode multiple codesets, the encodings themselves have names, such as UJIS or EUC.
- *modifiers* are not actually part of the locale name definition; they give more specific information about the desired localized behavior of an application. For example, under X11R5, a user can select an input method with modifiers. (To use the Xsi wnn Japanese input method, for instance, one adds @*im*=_*XWNMO* to the locale string.) There are no standards for this part of a locale string.

Language data is implementation specific; databases for language *en* (English) might contain English cultural data in England and American cultural data in the United States. If other than the default settings are required, the territory field may be used. For example, the above cases could be more rigidly bound by setting LANG to *en_EN* or *en_US*. Full rigor might lead to *en_EN.88591* for England (the locale encoding specification for ISO 8859/1 is "88591") and *en_US.ascii* for the USA.

ANSI C has defined a special locale value of *C*. The *C* locale is guaranteed to work on all compliant systems and provides the user with the default locale. This default will typically be American English and ASCII, but need not be. POSIX has also defined a special locale value of *POSIX*, which is identical to the *C* locale.

The length of the locale string may not exceed NL_LANGMAX characters. (See *nl_langinfo.h*). However, XPG/3 recommends that this string (not counting modifiers) not exceed 14 characters.

1.3.3 Limitations of the Locale system

Multilingual support

Since there can only be one locale at a time associated with any given process, internationalization does not provide inherent support for multilingual applications such as these:

- Suppose an application creates and maintains windows on four different displays, operated by four different users. The four users may each wish a different locale. However, there is a single controlling process (which has only one locale at a time).
- Or suppose an individual has a sophisticated editing system with a complex user interface. Suppose further that this user's natural language is German. The user may wish to compose a Japanese document, using Japanese input and text manipulation, but may wish the user interface to operate in German. (There is no standard interface for such behavior.)

In writing a multilingual application, the first task is identifying the several locales one wishes to run in and where they apply. There is no standard way of doing this.

Once the desired locales are understood, the application must either:

1. fork and call *setlocale()* differently in each process

or

2. call *setlocale()* repeatedly as the code changes from language to language.

Misusing locales

LANG and the locale variables provide the freedom to configure a locale, but they in no way protect the user from creating a nonsensical combination of settings. For example, you are allowed to set LANG to *fr* (French) and LC_COLLATE to *ja* (Japanese). In such a case, string routines would assume 8859/1-encoded text except for the sorting routines, which might assume EUC-encoded text and Japanese sorting rules. This would likely result in unanticipated behavior.

No filesystem tags for encoding types

The IRIX filesystem does not tag data encoding types within the filesystem. The system has no way of determining how data is encoded. Thus one must assume that data presented to an application in some locale is properly encoded for that locale. In other words, a file is interpreted differently depending on locale; there is no way to ask the file what it thinks its encoding is.

For example, you may have created a file while in a Japanese locale using SJIS. Later, you might try printing it while in a French locale. The results will likely resemble a random collection of Latin-1 characters.

This problem applies to almost all stored strings. Most strings are uninterpreted sequences of nonzero bytes. This includes, for example, filenames. You can, if you want to, name your files using Chinese characters, but the names will look odd to anyone running */bin/ls* in a non-Chinese locale.

1.4 Character sets, codesets, and encodings

One major difference between nationalized and internationalized software is the availability of a wide variety of methods for encoding characters.

Developers of internationalized software no longer have the convenience of always being able to assume ASCII.

A *character set* is a collection of characters. A *codeset* is a character set with an associated numerical encoding for each character. The English alphabet is a character set; ASCII is a codeset.

An *encoding* is another set of characters and associated numbers, but it may consist of more than just one codeset; *EUC* (see the "EUC" section of this chapter), for instance, is an encoding which provides for four codesets in one data stream.

1.4.1 8-bit cleanliness

A program is not *8-bit clean* if, as data passes through it, the program molests the data depending on the status of the high bit of any byte. Otherwise, the program is 8-bit clean.

Writing 8-bit clean code is very easy. It is easier than writing 8-bit dirty code. Since a program must go out of its way to manipulate bytes depending on the high bit, and since changing data without cause is seldom desired, most programs are 8-bit clean. Some are not.

The old *csh* (before this problem was fixed) was a good example of a program which was not 8-bit clean; it used the high bit in input strings to distinguish aliases from unaliased commands. An effect of this misuse was that *csh* stripped the 8th bit from all characters:

```
oldcsh> echo I know an architect named Mañosa
I know an architect named Maqosa
oldcsh>
```

Another example is the specification of Internet messages, which calls for 7-bit data. Thus if *sendmail* fails to strip the 8th bit from a character prior to sending it, it violates a protocol (this protocol problem is being addressed); if it does strip the bit, it could garble a non-ASCII message.

One of the simplest things to do to remove the American bias from a program is to replace the ASCII assumption with the assumption that the Latin-1 codeset will be used. This approach is not true internationalization, but it can make the application usable in most of Western Europe. Latin-1 uses only one byte per character, unlike some other codesets, so 8-bit clean ASCII software should continue to work using the Latin-1 codeset.

Ensuring that code is 8-bit clean is the single most important aspect of internationalizing software.

1.4.2 Multiple-byte character types

Western languages usually require only one byte for each character. Asian languages, however, often require two or even four bytes per character; and some Asian encodings allow a variable number of bytes per character.

The two kinds of character encodings which allow more than one byte per character are wide (*WC* or *wchar*) characters (which are a fixed number of bytes long) and multibyte (*MB*) characters (which are of variable size). The application developer must decide where to use WC and MB characters and strings:

- Multibyte strings are most commonly used for storage and communications. Strings written to or read from file descriptors are MB strings. Multibyte strings are almost the default; string I/O uses MB, MB code works for ASCII and ISO8859, and MB characters use less space than do wide characters. However, manipulating individual characters within a multibyte string is difficult.
- Applications which do heavy string manipulation typically use WC strings in the work area, because manipulating individual WC characters in a string is much simpler than doing the same thing with MB characters. Wide characters are used as necessary to provide programming ease or run-time speed. However, they take up more space than MB characters.

1.4.3 Traditional ASCII or 8859 strings

Traditional strings are merely a special case of multibyte strings, where every character happens to be one byte large and there is only one codeset. All MB code, including converters to/from wchars, works for these traditional ASCII or ISO 8859 strings.

1.4.4 Multibyte characters

A multibyte character is a stream of bytes. The character itself identifies how many bytes are in it. Multibyte characters are referenced as strings; before parsing, a string is indistinguishable from a character. Multibyte characters never have a zero byte, which still denotes End Of String.

Multibyte strings and characters are referred to by *char* *. A single MB character is represented as an array of bytes. An MB string is a null-terminated array of bytes, exactly like a traditional char string.

A multibyte string may contain characters in different codesets. Usually, this is done by incorporating special bytes which switch codesets on a per-character basis. Very little application code should ever care about this.

What to do with multibyte strings?

Multibyte strings are very easy to pass around. They efficiently use space (both data and disk space) since "extra" bytes are used only for characters which require them. Mb strings can be read and written without regard to their contents, as long as the strings remain intact. "Printing" MB strings to a terminal is done with *printf()*, *puts()*, and so on. Many programs (such as *cat*) need never concern themselves with the multibyte nature of MB strings.

Manipulation of individual characters in an MB string can be painful, since finding a particular character or position in a string is non-trivial (see "Handling multibyte characters"). Therefore, it is common to convert to WC strings for this kind of work.

Multibyte strings are used for storage and communications. Strings written to or read from file descriptors are MB strings.

Handling multibyte characters

In general, multibyte characters are handled just like *char* strings. Editing such strings, however, requires some care.

You cannot tell how many bytes are in a particular character until you look at the character. You cannot look at the *n*th character in a string without looking at all the previous n - 1 characters, because you cannot tell where a character starts without knowing where the previous character ends. Given a byte, you don't know its position within a character. Thus we say the string has *state* or is *context-sensitive*. The state (or context) means the knowledge of where we are in a character.

This analysis of characters is locale-dependent, and therefore must be done by routines which understand locale.

How many bytes in a character?

To find out how many bytes make up a single MB character, use *mblen(*3*)*:

```
#include <stdlib.h>
.
.
.
size_t n;
int len;
char *pStr;
.
.
.
len = mblen(pStr, n); /* at most n chars examined */
```

It is the application's responsibility to ensure that *pStr* points to the beginning of a character, not to the middle of a character.

The maximum number of bytes in a multibyte character is MB_LEN_MAX, which is defined in *stdlib.h*. The maximum number of bytes in a character under the current locale is given by the macro MB_CUR_MAX, also defined in *stdlib.h*.

How many bytes in an MB string?

Since *strlen()* simply counts bytes before the first NULL, it will tell you how many bytes are in an MB string.

How many characters in an MB string?

When *mbstowcs()* coverts MB strings to WC strings, it returns the number of characters converted. This is the simplest way to count characters in an MB string.

Note: Many code segments which would need to deal with individual characters within a string would be better suited by wide character strings. Since counting often involves conversion, such segments are often better served by working with a WC string, then converting back.

Getting the length without performing the conversion is straightforward, but not as simple. *mbtowc()* converts one character and returns the number of bytes used, but will return the same information without conversion if a NULL is passed as the address of the WC destination. Thus

```
len = mblen(pStr, n);
```

is equivalent to

```
len = mbtowc((wchar_t *) NULL, pStr, n);
```

Therefore, counting characters in an MB string without converting would look like this:

```
int cLen;
char *tStr = pStr;

numChars = 0;
cLen = mbtowc((wchar_t *) NULL, tStr, MB_CUR_MAX);
while (cLen > 0) {
    tStr += cLen;
    numChars++;
    cLen = mbtowc((wchar_t *) NULL, tStr, MB_CUR_MAX);
    if (cLen == -1)
        numChars = cLen; /* invalid MB character */
}
```

Conversion to constant-size characters

Multibyte characters and strings are convertible to wchars via *mbtowc(*3*)* (individual characters) and *mbstowcs(*3*)* (strings).

1.4.5 Wide characters

A wide character (*WC* or *wchar*) is a data object of type *wchar_t*, which is guaranteed to be able to hold the system's largest numerical code for a character. *wchar_t* is defined in *stdlib.h*. Under IRIX 4.0.x, *sizeof(wchar_t)* is 1.

Under IRIX 5.0, it is 4. All wchars on a system are the same size, independent of locale, encoding, or any other factors.

What to do with wchar strings?

The single advantage of WC strings is that all characters are the same size. Thus a string can be treated as an array, and a program can simply index into the array in order to modify its contents. Most applications' *char* manipulation routines can work with little modification other than a type change to *wchar_t*, with appropriate attention to byte count and *sizeof()*.

So, when applications have significant string editing to perform, they typically keep the strings in WC format for the editing. Whether the WC strings originally were converted from or are later converted to multibyte strings depends on the application.

Wide characters are often large, and are not as space efficient as multibyte strings. Applications which do not need to perform string editing may never use wchars. If an application intends to both maintain and edit large numbers of strings, then the developer needs to make size/complexity trade-off decisions.

Wchar support routines

Analogs to the routines defined in *string.h* and *stdio.h* are supplied in *libw.a* and defined in *widec.h*. This includes routines such as *getwchar()*, *putwchar()*, *putws()*, *wisprint()*, *wscpy()*, *wslen()*, and *wsrchr()*.

Conversion to MB characters

Wide characters and strings are convertible to MB strings via *wctomb()* (individual characters) and *wcstombs()* (strings).

1.4.6 Reading input data

Input can be thought of in two ways: *user events* and *data*. This section deals with data, which are assumed to come from file descriptors or streams. User events are discussed in Section 1.8, "User input."

It is generally fair to assume that unless otherwise specified, data read by an application is encoded suitable for the current locale. Text strings will typically be in MB format.

Streams can be read in WC format by using routines defined in *widec.h*.

1.5 Cultural items

This section discusses several aspects of a locale which may differ from another locale.

1.5.1 String collation

Different locales can have different rules governing collation of strings, even within identical encodings.

The problem

In English, sorting rules are extremely simple: each character sorts to exactly one unique place. Under ASCII, the characters are even in numeric order. However, neither of those statements are necessarily true for other languages and other codesets. Furthermore:

- Sorting order for a language may be completely unrelated to the (numerical) order of the characters in a given encoding.
- Even with a correctly-sorted list of the characters in a character set, you may not be able to sort words properly.
- Locales using identically-encoded character sets may use very different sorting rules.

Programs using ASCII can do simple arithmetic on characters and directly calculate sorting relationships; such programs frequently rely on truisms such as the fact that

```
'a' < 'b'
```

in ASCII. But internationalized programs cannot rely on ASCII and English sorting rules. Consider some non-English collation rule types:

- *One-to-Two* mappings collate certain characters as if they were two, For example, the German ß collates as if it were “ss”.
- *Many-to-One* mappings collate a string of characters as if they were one. For example, Spanish sorts “ch” as one character, following “c” and preceding “d”. In Spanish, the following list is in correct order: calle, creo, chocolate, decir
- *Don't-Care Character* rules collate certain characters as if they were not present. For example, if “-” were a don't-care character, “co-op” and “coop” would sort identically.
- *First-vowel* rules sort words based on the first vowel of the word, followed by consonants, which may precede or follow the vowel.
- *Primary/Secondary* sorts consider some characters as equals until there is a tie. For example, in French, a, á, à, and â all sort to the same primary location. If two strings (such as “tache” and “tâche”) collate to the same primary order, then the secondary sort distinguishes them.
- Special case sorts exist for some Asian languages. For example, Japanese *Kanji* has no strict sorting rules. *Kanji* strings can be sorted by the strokes which make up the characters, by the *kana* (phonetic) spellings of the characters, or by other agreed-upon rules.

It should be clear that a programmer cannot hope to collate strings by simple arithmetic or by traditional methods.

The solution

Locale-specific collation should be performed with *strcoll()* and *strxfm()*. These are table-driven functions; the tables are supplied as a piece of locale support. The value of LC_COLLATE determines the ordering table used. *strcoll()* has the same interface as *strcmp()*; it can be directly substituted.

1.5.2 Specification of numbers and money

Format of simple numbers differs from locale to locale. Characters used for decimal radix and group separators vary. Grouping rules may also vary. Even though we assume that decimal numbers are universal, there are some eighteen varying aspects of numeric formatting defined by a locale. Many of these are details of monetary formatting.

For example, Germany uses a comma to denote a decimal radix and a period to denote a group separator. English reverses these. India groups digits by two after the first three. Many locales have monetary particulars:

Country	Positive Format	Negative Format
India	Rs 1,02,34,567.89	Rs(1,02,34,567.89)
Italy	L.10.234.567	-L.10.234.567
Japan	¥10,234,567	
Netherlands	F 10.234.567,89	F -10.234.567,89
Norway	kr10.234.567,89	kr10.234.567,89-
Switzerland	SFrs.10,234,567.89	SFrs.10,234,567.89C

Table 1-3 Some numeric formatting differences

Printf()

printf() will monitor LC_NUMERIC and will use the appropriate decimal radix. If none is available, it will try to use ASCII period. No further locale-specific formatting is done by *printf()*, but see Section 1.6.3, "Variably ordered referencing of printf() arguments."

Localeconv()

localeconv() can be called to find out about numeric formatting data, including the decimal radix (inappropriately called *decimal_point*), the grouping separator (inappropriately called *thousands_sep*), the grouping rules, and a great deal of monetary formatting information. Actual use of formatting information is left to the application; there aren't any special print routines which produce formatted numbers according to all of *localeconv()*'s data.

1.5.3 Dates and times

All of these can mean the same date to different people:

```
92.1.4
```

```
4/1/92
```

```
1/4/92
```

All of these can mean the same time to different people:

```
2:30 PM
14:30
14h30
```

Dates and time can be easily formatted by using *strftime()*, *ascftime()* and *cftime()*, which give a host of options for displaying locale-specific dates and times. *asctime()* and *ctime()* are now obsolete; use the new functions instead.

1.5.4 Character classification and ctype

ctype.h is described in *ctype*(3) and defines macros for *isalpha()*, *isupper()*, *islower()*, *isdigit()*, *isxdigit()*, *isalnum()*, *isspace()*, *ispunct()*, *isprint()*, *isgraph()*, *iscntrl()*, and *isascii()*.

The issue

When programmers knew that a character set was ASCII, some convenient assumptions could be made about characters and letters. It was common for programmers to do arithmetic with the ASCII code values in order to perform some simple operations. For example, raising a character to upper case could be done by subtracting the difference between the code for *a* and the code for *A*. Numeric characters could be identified by inspection: if they fell between *0* and *9*, they were numeric, else they weren't. You could tell if a character was (say) printable, a letter, or a symbol by comparing to known encoding values. Macros for such activity have been available in *ctype.h*, but lots of programs did character arithmetic anyway. Since character encoding and linguistic semantics are completely independent, such arithmetic in an internationalized program will certainly lead to unpleasant results.

Furthermore, characters exist outside of ASCII which break some non-arithmetic assumptions. Consider the German character *ß* which is a lowercase alphabetic character (letter) yet has no uppercase. Consider also French (as written in France), where the uppercase of *é* is *E*, not *É*.

Clearly, the programmer has no way of directly computing all the character associations which were available in English under ASCII.

The solution

Strict avoidance of arithmetic on character values should remove any trouble in this area. The macros in *ctype.h* are table-driven and are thereby locale-sensitive.

Functions or macros

Functions exist for all the above macros of *ctype.h*. In order to get a function instead of a macro, simply undefine the name, for example:

```
#undef toascii
char (*xlate)(char);
char a, b;
...
xlate = toascii;
a = (*xlate)(b);
```

1.5.5 Regular expressions

XPG/3 specifies some extensions to traditional regular expressions syntax for internationalized software. Few application developers do their own regular expression parsing and matching, however, so we will not include full details here. Tersely, the extensions provide the ability to specify matches based on:

- **character class** (such as *alpha*, *digit*, *punct*, or *space*)
- **equivalence class** (for instance: *a*, *á*, *à*, *â*, *A*, *Á*, *À*, and *Â* may be equivalent)
- **collating symbols** (such as matching the Spanish "ch" as one element because of its nature as a single collating token would use this)
- generalization of range specifications of the form $[c_1\text{-}c_2]$ to include the above

Programmers who will process expressions themselves will need the full description of internationalized regular expression grammar in Volume 3, Chapter 6 of XPG/3.

1.6 Strings and message catalogs

Message catalogs are compiled databases of strings. While a major role of message catalogs is to provide communications text in locale-specific natural language, the strings can be used for any purpose.The idea is that an application will extract all strings from a catalog, thus allowing localizers to supply catalogs suitable for a given locale.

There are two different and incompatible interfaces to message catalogs in IRIX: *MNLS* and *XPG/3*. MNLS is AT&T's *Multi-National Language Support*. XPG/3 is the *X/Open Portability Guide, Issue 3*, by X/Open, Ltd. Developers working on SVR4 or other AT&T code, or related base system utilities will probably be using MNLS. Developers working on independent projects will likely be using XPG/3. Neither is a solid standard, but XPG/3 is more so than MNLS, and its prognosis looks brighter. Thus applications developers who have to choose between the two interfaces are encouraged to use XPG/3 in order to maximize their portability.

Note: XPG/3 seems to be more popular in Europe. Asia seems to lean towards MNLS.

1.6.1 XPG/3 message catalogs

The XPG/3 message catalog interface requires that a catalog be opened before it is read, and requires that catalog references specify a catalog descriptor.

Since catalog references include a default to be used in case of failure, applications will work normally without a catalog when in the default locale. This means catalog generation is exclusively the task of localizers. But in order to inform the localizer what strings to translate and how they should comprise a catalog, the application developer should provide a catalog for the developer's locale.

Opening and closing catalogs

catopen() locates and opens a message catalog file.

```
#include <nl_types.h>
nl_catd catopen(char *name, int unused);
```

The argument *name* is a string that is used to locate the catalog. Usually, this is a simple name. The name becomes a base for a catalog search depending on environment variables; see "XPG/3 catalog location" for more information. Names that begin with '/' are assumed to be full pathnames to the catalog. Such hard-coded pathnames are strongly discouraged.

When an application is finished using a message catalog, it should close the catalog and free the descriptor using *catclose()*.

```
int catclose(nl_catd);
```

Using a catalog

Catalogs contain *sets* of *numbered messages*. The application developer must know the contents of the catalog in order to specify the set and number of a message to be obtained.

catgets() is used to retrieve strings from a message catalog.

```
#include <nl_types.h>
char *catgets (nl_catd catd, int set_num, int msg_num,
               char *defaultStr);
```

catgets() retrieves message *msg_num* from set *set_num* from the catalog described by *catd*. If for any reason *catgets()* cannot do this, it returns *defaultStr*.

Without getting into message sets, assuming we want message 1 from message set 1, a typical use is the following:

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

#define SET1      1
#define WRLD_MSG  1

int main(){
    nl_catd msgd;
    char *message;
    setlocale(LC_ALL, "");

    msgd = catopen("hw",0);
    message = catgets(msgd, SET1, WRLD_MSG, "Hello, world\n");
    printf(message);
    catclose(msgd);
```

```
}
```

The above example uses *printf()* instead of *puts()* in order to make a point: the format string of *printf()* came from a catalog. Note that

```
printf(catgets(msgd, set, num, defaultStr));
```

is very different from

```
printf("%s", catgets(msgd, set, num, defaultStr));
```

because strings in catalogs can contain formatting strings of the kind used by *printf()*. For further discussion of issues relating to this important distinction, see Section 1.6.3, "Variably ordered referencing of printf() arguments."

XPG/3 catalog location

XPG/3 message catalogs are located using the environment variable NLSPATH. The default NLSPATH is */nlslib/%L/%N* where *%L* is filled by the LANG environment variable and *%N* is filled in by the *name* argument to *catopen()*. NLSPATH can specify multiple pathnames in ordered precedence, much like the PATH variable. A sample NLSPATH assignment:

```
NLSPATH=/usr/lib/locale/%L/%N:/usr/local/lib/locale/%L/%N:/usr/defaults/%N
```

Full details are in the *catopen(3)* manual page.

Creating message catalogs

Message catalogs are of the format

```
$set n comment
a message-a\n
b message-b\n
c message-c\n
$quote "
d " message-d "
$this is a comment
$this is another comment
```

Each message is identified by a *message number* and a *set*. Sets are often used to separate messages into human-readable chunks, such as error messages, help messages, directives, and so on. Sets can also be used to break down messages by source file.

$set n specifies the beginning of set *n*, where *n* is a set identifier in the range between 1 and NL_SETMAX. All messages following the *$set n* statement belong to set *n* until either *$delset* or another *$set* is reached. Set numbers need not be contiguous, but must be presented in ascending order.

$delset n deletes the set *n* from a message catalog.

$quote c specifies a quote character *c* which can be used to surround message text so that trailing spaces or null (empty) messages are visible in a message source line. By default, there is not quote character and messages are separated by newlines. To continue a message onto a second line, add a backslash to the end of the first line:

```
$set 1
1 Hello, world.
2 here is a long \
string.\n
3 Hello again..
n message-text-n
```

Message #2 in set #1 is "here is a long string.\n".

Compiling catalogs

After creating the message catalog sources, you need to compile them into binary form using *gencat*, which has the following syntax:

```
gencat catfile msgfile [msgfile ...]
```

where *catfile* is the target message catalog and *msgfile* is the message source file. If an old *catfile* exists, *gencat* will attempt to merge new entries in with the old. *gencat* "resolves" set and message number conflicts with new information replacing the old.

The *catfile* then needs to placed in a location where *catopen()* can find it; see the "XPG/3 catalog location" section of this manual.

1.6.2 SVR4 MNLS message catalogs

There are many ways to use strings from MNLS message catalogs. You can get strings directly and then use them, or you can use output routines which search catalogs.

Specifying MNLS catalogs

MNLS message catalogs do not need to be specifically opened. The catalog of choice can be set explicitly once, or it can be specified every time a string is needed.

To specify the default message catalog to be used by subsequent calls to MNLS routines which reference catalogs (such as *gettxt()*, *tfmt()*, or *pfmt()*), use *setcat()*:

```
#include <pfmt.h>
char *setcat(const char *catalog);
```

catalog is a string, limited to 14 characters. No character can be zero or the ASCII codes for / (slash) or *:* (colon). *setcat()* doesn't check to see if the catalog is valid; it just stores the string for future reference. For an example of use, see the "Getting strings from MNLS message catalogs" section. The catalog should be named by the string, which resides in
*/usr/lib/locale/*localename/*LC_MESSAGES*

Getting strings from MNLS message catalogs

MNLS message catalogs do not need to be specifically opened. The catalog of choice can be set explicitly once, or it can be specified in each reference call. Strings are read from a catalog via *gettxt()*:

```
#include <unistd.h>
char *gettxt(const char *msgid, const char *defaultStr);
```

msgid is a string with two fields separated by a colon:

```
msgid:: msgfilename:msgnumber
```

The *msgfilename* is as described in the "Specifying which MNLS catalog" section of this manual. For example, to get message 10 from the *UX* catalog:

```
char *str = gettxt("UX:10", "Hello, world.\n");
```

or

```
setcat("UX");
str = gettxt(":10", "Hello, world.\n");
```

Pfmt()

pfmt() is one of the most important routines dealing with MNLS catalogs, because it is used to produce most system diagnostic messages. *pfmt()* formats like *printf()*, and produces standard error message formats. It generally replaces *perror()*.

Error messages typically are generated with *pfmt()*. For example,

```
pfmt(stderr, MM_ERROR, "UX:64:Permission denied");
```

would produce

```
ERROR: Permission denied.
```

The syntax of *pfmt()* is:

```
#include <pfmt.h>
int pfmt(FILE *stream, long flags, char *format, ... );
```

The *flags* are used to indicate severity, type, or control details to *pfmt()*. The format string includes information specifying which message from which catalog to look for. Flag details are discussed in the “Labels and severity, and flags” section of this chapter; and the format is discussed in the “Pfmt() format strings” section.

Labels and severity, and flags

pfmt() flags are composed of several groups; not more than one from each group should be specified. Multiple flags are specified by ORing them together.

MM_NOSTD, MM_STD—output format control

MM_NOGET, MM_GET—catalog access control

MM_HELT, MM_ERROR, MM_WARNING, MM_INFO—severity

MM_ACTION—specifies an action message

pfmt() prints messages in the form *label:severity:text*. *Severity* is specified in the *flags*. The *text* comes from a message catalog (or a default) as specified in the *format*, and the *label* is specified earlier by the application.

In the example above, we get only

```
ERROR: Permission denied.
```

only if no label has been set. Typically, an application will set the label once early in its life; ensuing error messages will have the label prepended:

```
setlabel("UX:myprog");
...
pfmt(stderr, MM_ERROR, "UX:64:Permission denied");
```

would produce

```
UX:myprog: ERROR: Permission denied.
```

For details, consult the *pfmt(*3*)* and *setlabel(*3*)* manual pages.

Pfmt() format strings

pfmt() format strings are of the form

[[*catalog*:]*msgnum*:]*defaultStr*

The *catalog* field is as in the "Specifying MNLS catalogs" section of this chapter. *msgnum* is the message number in the catalog to use as the format. *defaultStr* is the string to use if the catalog lookup fails for any reason.

An important feature of *pfmt()* is its ability to refer to format arguments in format-specified order as does *printf()*. See Section 1.6.3, "Variably ordered referencing of printf() arguments," for details.

fmtmsg()

fmtmsg() is a comprehensive formatter using the MNLS catalogs and "standard" formats. You may not need everything given in the specification; most applications should get by with *pfmt()*, *gettxt()*, and *printf()*. Consult the *fmtmsg(*3*)* manual page for details.

Putting strings into a catalog

MNLS catalog sources are merely lists of strings in a file, separated by newlines. For an empty string, an empty line is used. Strings are referenced by line number in the original source file.

Adding strings to an existing catalog source should only be done at the end of the file. Otherwise, other applications will no longer know where in the catalog their strings are.

Changing strings already in the source can be done directly.

When a file of strings is ready to be compiled, simply run *mkmsgs* and put the results in */usr/lib/locale/*localename/*LC_MESSAGES*.

1.6.3 Variably ordered referencing of printf() arguments

printf() and its variants can now refer to arguments in specified order. Consider the following scenario: an application has chosen "house" from a list of objects and "white" from a list of colors. The application wishes to print the choice. The code might look like:

```
char *obj, *color;
... /* choices made */
printf("%s %s\n", color, obj);
```

And the *printf()* yields

```
white house
```

If we make sure that *obj* and *color* are localized strings, we are not quite finished. If our locale is Spanish, the *printf()* will yield

```
blanca casa
```

which is wrong; in Spanish, we need

```
casa blanca
```

The solution to this problem is *variably ordered referencing* of *printf()* arguments. The syntax of *printf()* format strings which take arguments has been expanded.

If a format string used to contain %*T* (where *T* represents one of the *printf()* conversion characters), the string can now contain %*D*$*T*. The *T* is the same, but the *D* specifies which argument from the argument list should be used.

This means you can write

```
printf("2nd parameter is %2$s; the 1st is %1$s", p1, p2)
```

and the *second* parameter will get printed *first*, with the first parameter printed second. For example:

```
char *store = "Macy's";
char *obj = "a cup";

printf("At %1$s, I bought %2$s.\n", store, obj);
printf("I bought %2$s at %1$s.\n", store, obj);
```

yields

```
At Macy's, I bought a cup.
I bought a cup at Macy's.
```

even though the parameters to *printf()* are in the same order.

In English, we are able to come up with strings suitable for either word order; in some other language, we might not be so lucky. Nor can we predict which order such languages might prefer. So the developer has no way of knowing how to create *printf()* format strings suitable for all languages.

Developers should therefore use message catalogs for their *printf()* format strings which take linguistic parameters, and allow localizers to localize the format strings as well as text strings. This means that the localizer has much greater ability to create intelligible text. An internationalized version of the above would look like:

```
/* internationalized (XPG/3) version */
char *form = catgets(msgd, set, formNum,
                   "At %1$s, I bought %2$s.\n");
char *store = catgets(msgd, set, storeNum, "Macy's");
char *obj = catgets(msgd, set, objNum, "a cup");

printf(form, store, obj);
```

The unlocalized (default) version will produce

```
At Macy's, I bought a cup.
```

and a localized version might produce

```
Compré una tasa en Macy's.
```

In practice, variably ordered format strings are only found in message catalogs and not in default strings; the default string usually uses the parameters in the order they're given.

1.7 Changes to support internationalization in X11R5

X11R5 internationalization support is provided on the client side. No server changes are necessary, and the protocol is unchanged. Full backward compatibility is preserved and a new internationalized application can run on an old server.

X uses existing internationalization standards for support. This means that the same interfaces are used to set and change locale. Internationalized X applications receive no help from X when attempting multilingual support. There are no X locales or special process state peculiar to X.

1.7.1 Some effects of internationalization decisions

Because of these decisions, some concessions had to be made. Besides some of the ANSI C and POSIX restrictions, the X protocol or Xlib specification has led to the following.

Vertical text

There is no built-in support for vertical text. Applications may draw strings vertically by manually laying out the text.

Character sets

In previous releases of X, there was no provision for any string to use an character set other than Latin 1. X11R5 contains such provisions.

X11R5 defines the *X Portable Character Set*, which is required to exist in all locales supported by Xlib. There is no encoding defined for this set; it is only a character set. The set, similar to printable ASCII, consists of these characters:

abcdefghijklmnoqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789

!"#$%&’()*+,-./:;<=>?@[\]^_‘{|}~

<space> <tab> <newline>

The *Host Portable Character Encoding* is the encoding of the X Portable Character Set on the Xlib host. The encoding is the same in all locales supported by the host.

Strings used or returned by Xlib routines are either in the Host Portable Character Encoding or a locale-specific encoding. The Xlib manual pages specify which encodings are used where. Some string constructs (such as *TextProperty*) contain information regarding their encoding.

Xlib interface change

Full use of X11R5’s internationalization features means calling some new routines supplied in the X11R5 Xlib. While all old Xlib applications will work with the new Xlib, developers will be asked to change their code in places. These are described below.

1.7.2 Resource names

Resource names are compiled into programs. Because of that, their encoding must be known independent of locale. Trying to add a level of indirection here results in a chicken-and-egg story: you’re always left with something compiled. Resource names therefore use the *X Portable Character Set*. The names may be anything; at least they’ll mean something to the application author. (If the names were (say) numbers, they would be meaningless to everybody.)

1.7.3 Getting X Internationalization started

Xlib, like libc, needs to be initialized with *setlocale()*.

Xlib programming

Initialize Xlib's Internationalization state after calling *setlocale()*. Xlib is being initialized, not a server or server-specific object, so a server connection is not necessary:

```
if ( setlocale(LC_ALL, "") == NULL )
    HorribleDisasterSequence();
if ( ! XSupportsLocale() )
    OtherHorribleDisasterSequence();
if ( XSetLocaleModifiers("") == NULL)
    WarningSequence();
```

Just as passing an empty string *locale* to *setlocale()* honors the user's environment, passing an empty string to *XSetLocaleModifiers()* does the same thing.

Toolkit programming

setlocale() should be called as early as possible after *main()*. If locale is set, standard toolkit initialization interfaces will perform required X initialization; no extra work is necessary.

1.7.4 FontSets

X11R5 supports the concept of strings which may contain characters in different codesets. Distinguishing which character is in which codeset is different from locale to locale and from encoding to encoding. Such strings usually cannot be rendered using a single font; enough fonts to render all embedded codesets are typically needed.

A *FontSet* is a collection of fonts suitable for rendering all codesets represented in a locale's encoding. Included in a FontSet is the locale in which it was created.

Example: EUC in Japan

To render strings encoded in EUC in Japan (see Section 1.10.2, “Asian languages”), the application needs fonts encoded in 8859-1, JIS X 208, and JIS X 201. It doesn’t know which characters in a string go with which font, since it doesn’t get into locale-specifics. So, it creates a FontSet which is made from a list of user-specified fonts. One assumes the localizer has provided an appropriate list. Rendering is then done using that FontSet. The locale-aware rendering system will choose the appropriate fonts for the characters being rendered, and will report trouble if the list does not contain suitable fonts.

Specifying a FontSet

A FontSet specification is just a string, enumerating XLFD names of fonts. (See *X Logical Font Description Conventions*, an MIT X Consortium standard.) This string can include wild card characters. For example, a specification of 15-point “fixed” fonts might be:

```
char *fontSetSpecString = "*fixed-medium-r-normal*150*";
```

A particular server might expand this to

```
-jis-fixed-medium-r-normal--16-150-75-75-c-160-jisx0208.1983-0
-sony-fixed-medium-r-normal--16-150-75-75-c-80-iso8859-1
-sony-fixed-medium-r-normal--16-150-75-75-c-80-jisx0201.1976-0
```

Specifying the fontset by simply enumerating the fonts is perfectly acceptable:

```
char *fontSetSpecString =
"-jis-fixed-medium-r-normal*150-75-75*jisx0208.1983-0,\
-sony-fixed-medium-r-normal*150-75-75*iso8859-1,\
-sony-fixed-medium-r-normal*150-75-75*jisx0201.1976-0";
```

A German locale would use this fontset with interest only in the ISO font; a Japanese locale might use all three; a Chinese locale would have trouble with this fontset.

The developer should specify a default FontSet suitable for the default locale. Furthermore, developers should ensure that the application will accept localized FontSet specifications via resources (or message catalogs) or command line options. Localizers are responsible for providing default FontSet specifications suitable for their locales.

Creating a FontSet

Creating FontSets in X is simply a matter of providing a string which names the fonts, as described above.

```
XFontSet fontset;
char *base_name;  /* should get from resource */
char **missingCharsetList;
int missingCharsetCount;
char *defaultStringForMissingCharsets;

base_name = "*fixed-medium-r*150*"; /* use resources! */

fontset = XCreateFontSet(display, base_name,
                        &missingCharsetList,
                        &missingCharsetCount,
                        &defaultStringForMissingCharsets);
```

The locale in effect at create time is bound to the fontset. FontSets are freed with *XFreeFontSet()*.

Using a FontSet

FontSets are used when rendering text with X11R5 *Xmb* or *Xwc* text rendering routines. These routines are described in Section 1.7.5, "Text rendering routines."

1.7.5 Text rendering routines

X11R5 includes text rendering routines which understand multibyte and wide character strings. These routines are analogs to the X11R4 text rendering routines *XDrawText()*, *XDrawString()*, and *XDrawImageString()*. The old routines continue to operate, but do not take fontsets, and do not decode strings.

- *X{mb,wc}DrawString()* renders a string using a fontset. *DrawString* routines render in foreground only, and use the *GC* rasterop.
- *X{mb,wc}DrawImageString()* also renders a string using a fontset. *DrawImageString* routines fill the background rectangle of the entire string with the background, then render the string in the foreground color, ignoring rasterop.

- *X{mb,wc}DrawText()* takes lists of *TextItems*, each of which has (among other things) a string. The strings are rendered using fontsets. *DrawText* routines allow complex spacing and fontset shifts between strings.

Consult the appropriate manual pages for more details on these routines.

1.7.6 New text extents functions

X11R5 provides MB and WC versions of *width* and *extents* interrogation routines. They depend on fontsets for interpreting strings, and use locale-specific data to do their work.

The following structure type demonstrates what is meant by some *extents*:

```
typedef struct {
    XRectangle max_ink_extent;
    XRectangle max_logical_extent;
} XFontSetExtents;
```

The *max_ink_extent* gives the maximum extents of rendering of the drawable characters of a fontset. It considers only the parts of glyphs to be drawn, all relative to a constant origin. The *max_logical_extent* gives the maximum extents of *occupied space* of drawable characters of a fontset. The occupied space of a character is a rectangle specifying minimum spacing to other graphical features; other graphics generated by a client should not intersect this rectangle. *max_logical_extent* is used to compute interline spacing, and minimum space needed for a given number of characters.

Some of the new routines follow. Consult the appropriate manual pages for details.

- *XExtentsOfFontSet()* returns an *XFontSetExtents* structure for a FontSet.
- *X{mb,wc}TextEscapement()* takes a string and returns the distance in pixels (in the draw direction) to the origin of the next character after the string, if the string were drawn. Escapement is always positive, regardless of direction.
- *X{mb,wc}TextExtents()* takes a string and returns information detailing the overall rectangle bounding the string's image and the string's occupied space (for spacing purposes).
- *X{mb,wc}TextPerCharExtents()* takes a string and returns ink- and logical-extents for each character in the string. This can be used for

redrawing portions of lines or for word justification. If the FontSet might include context-dependent drawing, the client cannot assume that it can redraw individual characters and get the same rendering.

- *XContextDependentDrawing()* returns a *Boolean* telling whether a FontSet might include context-dependent drawing.

1.8 User input

No useful physical keyboard could possibly contain a key for every character we would ever wish to type. Certainly, there are commonly used characters which are not present on most USA keyboards. Even keyboards specifically designed for multilingual work cannot cover every character a user would wish to type. So methods have been invented which provide for input of almost any known character on even the most naïve keyboards. These schemes are referred to as *Input Methods* (IMs).

Input Methods vary significantly in design, use, and behavior. But there is a single API which developers use to access them. The object is for the developer to simply ask for an IM and let the system check the locale and choose the appropriate IM.

Some IMs are complex. Some are very simple. The API is designed to be a low-level interface, like Xlib. Usually, only toolkit text object authors will have to deal with the IM interfaces. However, some applications developers will be unable to use toolkit objects, so the concepts are described here.

1.8.1 Don’t Read This Section

This section (Section 1.8) has to do with the translation of physical user events into programmatic character strings or special keyboard data (such as “backspace”). *This kind of work should be done by toolkits*. If you can use a toolkit to manage event processing for you, do so, and blissfully ignore this section. If you are writing a toolkit text object, or are writing a truly extraordinary application, then this section is for you.

1.8.2 Reuse sample code

[[Working sample programs which do the work described in this chapter exist. Point to O'Reilly R4/R5 vol. 1?]]

1.8.3 GL input

The old GL function *qdevice()* has a hardcoded view of a keyboard (see */usr/include/gl/device.h* for details). Some flexibility, particularly for Europe, is available if you queue KEYBD instead of individual keys, but the GL has no general solution to non-ASCII input. There is no supported way to input (say) Chinese to the old GL.

The OpenGL does not contain input code and leaves that to the operating environment, which in IRIX means X.

In short, support for internationalized input means a departure from *qread()*. Under IRIX today, that means using mixed model input, all the better if done by a toolkit.

1.8.4 Keys, keycodes, keysyms

This section provides some background which may help make the following sections easier to understand.

X keyboard support, a micro-tutorial

In general, the X server starts off with an idea of the keyboard layout. When a client connects to the server, the server announces its range of *keycodes* and exports a table of *keysyms*. Each key event the client receives has a single byte *keycode*, which directly represents a physical key and a single byte *state*, which represents active modifier keys, such as Shift or Alt.

Note: The mapping of state bits to modifiers is also done by a table acquired from the server.

Keysyms are well defined, and there has been an attempt to have a keysym for every engraving one might possibly find on any keyboard, anywhere. These are contained in */usr/include/X11/keysymdef.h*. Keysyms represent the engravings on the actual keys, but not their meanings. The server's idea of the keysym table can be changed by clients, and clients may receive *KeyMap* events when this happens. This doesn't typically happen often.

When a client receives a Key event, it asks Xlib to use the keycode to index into its keysym table to find a list of keysyms. (This list is usually very short. Most keys have one or two engravings on them.) Using the state byte, Xlib chooses a keysym from the list, and now knows what was engraved on the key the user pressed.

At this point, the client can choose to act on the keysym itself (it may have been a backspace) or it may ask for a character string represented by the keysym (or both). Generating that string is tricky; it is discussed in Section 1.8.5, "Input methods."

Details on X keyboard support are available in [**Scheifler**]. Details on input methods are available in [**Scheifler**] and [**Flanagan**].

Composed characters

There are two ways to compose typed characters which do not exist on a keyboard, explicit and implicit.

It is common for an application to be modal and switch between the two. For example, Japanese input of *kana* is often done via implicit composition. Users switch from a mode where *Romaji* (Latin) text is input to a mode where strokes are all translated to *kana*.

Furthermore, both styles may operate simultaneously. While an application is supporting implicit composition of certain characters, other characters may be composable via an explicit compose.

The effect on the programmer is simple: just remember that not every keystroke will produce a character, even if the associated keysym would normally imply character text. The event-to-string translation routines (see "LookupString" in this chapter) handle this.

Explicit compose

Explicit composition is requested when the user presses the Compose key and then types a key sequence which corresponds to the desired character. For example, the sequence <Compose, ~, n> might be used to compose the character ñ.

Implicit compose

Implicit composition mimics many existing European typewriters which have "dead" keys: keys which type a character but do not advance the carriage. When a special "dead" key is struck, the system attempts to compose a character using the next character struck. For example, on a keyboard which had a diuresis (¨) and an O, but no Ö, one would simply strike '¨', then 'O' to compose Ö.

Implicit composition support usually comes with some specified way to leave characters uncomposed.

Keyboards

IRIX today supports 12 keyboard layouts: American, Belgian, Danish, English, French, German, Italian, Norwegian, Portuguese, Spanish, Swedish, and Swiss.

All are representable in Latin 1; the American needs only ASCII.

1.8.5 Input methods

Input methods (IMs) concern themselves with translation of key events to text strings. You would use a different input method, for instance, to type on a USA keyboard in Chinese than to type on the same keyboard in English. Nobody would build a keyboard suitable for direct input of some 80,000 distinct Chinese characters.

IMs come in two flavors, *Front-End* and *Back-End*. Both types can use identical application programming interfaces, so we lose no generality by using back-end methods for our examples here.

To use an IM, follow these steps:

- Open the IM
- Query what the IM can do

- Agree upon capabilities to use
- Create input contexts with preferences and window(s) specified
- Set the IC focus
- Process events

Although all applications go through the same set up when establishing input methods, the results can vary widely. In a Japanese locale, one might end up with networked communications with an input method and a *Kanji* translation server, with circuitous paths for Key events. But in (say) a Swiss locale, it is likely that nothing occurs other than a flag or two being set in Xlib. Since operating in non-Asian locales ends up bypassing almost all of the things which might make input methods expensive, Western users are not noticeably penalized for using Asia-ready applications.

1.8.6 Opening an input method

XOpenIM() opens an Input Method appropriate for the locale and modifiers in effect when it is called. The locale is bound to that IM and cannot be changed. (But another IM could be opened in order to effect a change.) Strings returned by *XmbLookupString()* and *XwcLookupString()* will be encoded in the locale which was current when the IM was opened. This is true for any input context used with the input method.

The syntax is:

```
XIM XOpenIM(Display *dpy, XrmDataBase db, char *res_name, char *res_class);
```

where *res_name* is the resource name of the application, *res_class* is the resource class, and *db* is the resource database which the Input Method should use for looking up resources private to itself. Any of these can be NULL.

So opening an IM is easy:

```
XIM im;
im = XOpenIM(dpy, NULL, NULL, NULL);
if (im == NULL)
    Tragedy();
```

XOpenIM() finds the IM appropriate for the current locale. If *XSupportsLocale()* has returned good status (see the “Xlib programming” section of this chapter), and *XOpenIM()* fails, something is amiss with the administration of the system.

Note: The possibility of *XSupportsLocale()* succeeding and *XOpenIM()* failing is difficult to prevent. (The IM may have died since *XSupportsLocale()* was called, for example.) This topic is under some debate in the MIT X consortium.

Most of the complexity associated with IM use comes from configuring an IC to work with the IM. This will be discussed below.

Input methods can be closed with *XCloseIM()*.

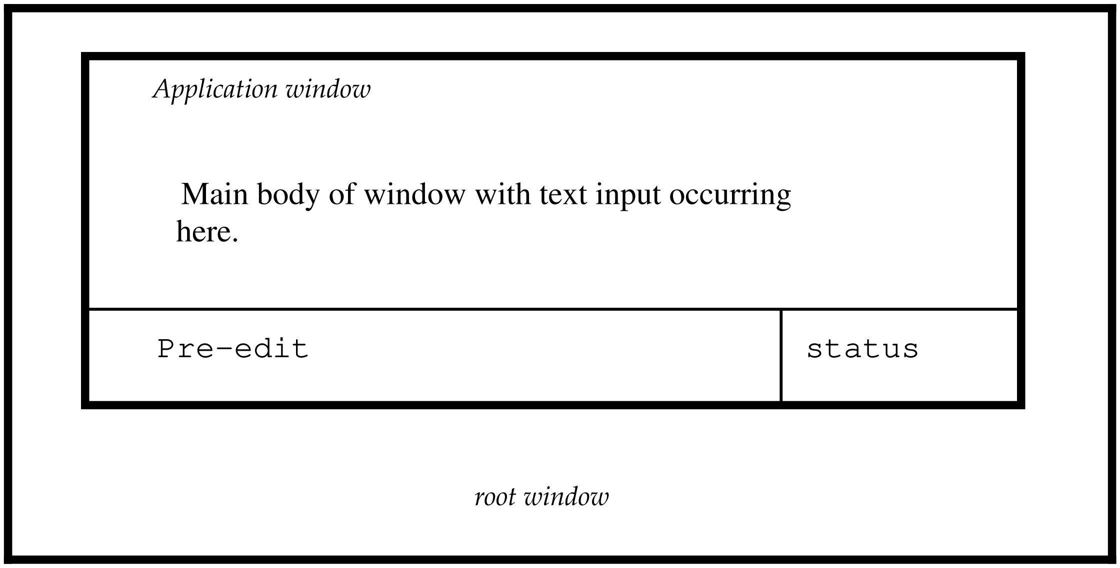
1.8.7 IM styles

Input methods often have *status* information regarding their state. For example, a Japanese IM may wish to announce whether it is in Japanese input mode or *Romaji* input mode. Input methods also have *pre-edit* information regarding their state. For composed characters, partial composition feedback may be available.

The way in which an IM deals with status and pre-edit information is referred to as an IM *style*. This section describes styles and their naming.

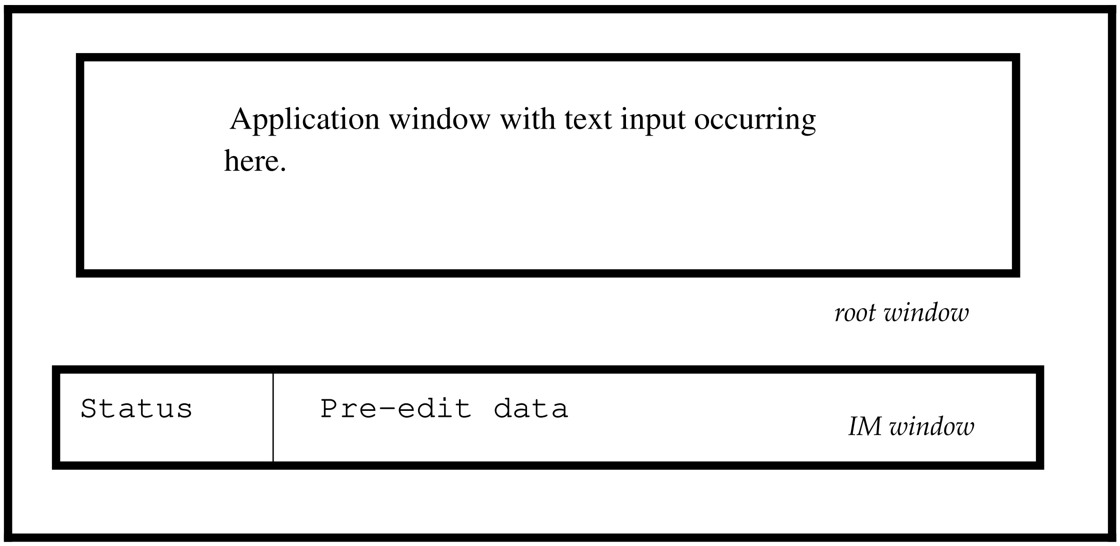
Off the Spot

The *Off-the-Spot* style has a pre-edit area and a status area in the window being used, but these areas are usually in reserved space away from the place of input. A window using off-the-spot input style might look like:


Application window
Main body of window with text input occurring here.
Pre-edit
status
root window

Root Window

The *Root-Window* style has a pre-edit area and a status area in a window owned by the IM as a descendent of the root.



Over the Spot

The *Over-the-Spot* style has the IM creating a (small) window over the point of insertion. The window is owned and managed by the IM as a descendant of the root, but it "looks" like input is happening "in the right place."

On the Spot

On-the-Spot input is by far the most complex for the application developer. The IM delivers all preedit data via callbacks to the application, which must perform in-place editing. This usually means a great deal of string and text rendering support at the input generation level, above and beyond the effort required for completed input.

Done well, this can be the most intuitive for a user.

Naming of styles

In the **Root-Window** style, the application does not manage the preedit data, the preedit area (if any), the status area, or any status data. Everything is left to the Input Method to do in its own window.

In the **Off-the-Spot** style, the application manages the preedit area and status area, but allows the IM to update the data there. (The application provides information regarding foreground & background colors, fonts, and so on.)

In the **Over-the-Spot** style, the application manages the status area as in Off-the-Spot, but specifies the location of the editing so that preedit data can be placed over that spot (by the IM).

In the **On-the-Spot** style, everything is done in place. Since this may mean lots of updating of surrounding data or other display management, everything is left to the application. There is little chance an IM could ever know enough about the application to be able to help it provide user feedback. The IM therefore provides status and edit information via callbacks.

A style describes how an IM presents its pre-edit and status information to the user. An IM will supply information detailing its presentation capabilities. The information comes in the form of flags, ORed together. Flag names and the OR patterns for the styles are:

Root Window	XIMPreeditNothing | XIMStatusNothing
Off the Spot	XIMPreeditArea | XIMStatusArea
Over the Spot	XIMPreeditPosition | XIMStatusArea
On the Spot	XIMPreeditCallbacks | XIMStatusCallbacks

For example, if you wanted a style variable to match an Over-the-Spot IM style, you could write:

```
XIMStyle Over = XIMPreeditPosition | XIMStatusArea;
```

If an IM were to announce *XIMStatusNone*, this would mean that the IM did not supply status information.

Using styles

Input Methods support one or more styles. It is up to the application to find a style which is supported by both the IM and the application. If there are several, the application must choose. If there are none, the application is in trouble.

1.8.8 Input contexts

An input method may be serving multiple clients, or one client with multiple windows, or one client with multiple input styles on one window. The specification of style and client/IM communication is done via *input contexts*.

To set up and use an Input Context:

1. Decide what styles your application can support
2. Query the IM to find out what styles it supports
3. Find a match
4. Determine information which the IC will need in order to work with your application
5. Create the IC
6. Employ the IC

Find an IM style

The IM may support several styles, say for example, Off-the-Spot and Root-Window. The application may be able to do, in preferential order, Over-the-Spot, Off-the-Spot, and Root-Window. The application should determine that the best match in this case is Off-the-Spot.

First, discover what the IM can do, and set up a variable describing what the application can do:

```
XIMStyles *IMcando;
XIMStyle  clientCanDo; /* note type difference */
XIMStyle  styleWeWillUse = NULL;

XGetImValues(im, XNQueryInputStyle, &IMcando, NULL);
```

```
clientCanDo =
/*none*/ XIMPreeditNone | XIMStatusNone |
/*over*/ XIMPreeditPosition | XIMStatusArea |
/*off*/  XIMPreeditArea | XIMStatusArea |
/*root*/ XIMPreeditNothing | XIMStatusNothing;
```

A client should always be able to handle the case of *XIMPreeditNone* | *XIMStatusNone*, which is likely in a Western locale. To the application, this is not very different from a *RootWindow* style, but it will come with less overhead.

Once we know what the application can handle, we look through the IM styles for a match:

```
for(i=0; i<IMcando->count_styles; i++) {
    XIMStyle tmpStyle;
    tmpStyle = IMcando->support_styles[i];

    if ( ((tmpStyle & clientCanDo) == tmpStyle) &&
        prefer(tmpStyle, styleWeWillUse) )
            styleWeWillUse = tmpStyle;
}

if (styleWeWillUse = NULL)
    Tragedy();
XFree(IMcando);

/* styleWeWillUse is set, which is what we were after */
```

The *prefer()* routine simply applies some heuristic to application preference.

Setting up an IC using the style we have selected is described next.

IC values

There are several things an input method may need to know depending on the input context and style chosen by the application. Necessary items are supplied to the input method via the input context. Items which do not affect a style or IM can be omitted.

Full descriptions of every item are supplied in [**Scheifler**]. We include only a brief description here.

- *XNClientWindow* specifies to the IM which client window it can display data in or create child windows in. It is set once and cannot be changed.
- *XNFilterEvents* is an additional event mask for event selection on the client window.
- *XNFocusWindow* specifies the focus window. This specifies which window will get the processed (composed) key events.
- *XNInputStyle* specifies the style for this IC.
- *XNGeometryCallback* specifies a geometry handler which will be called if the client allows an IM to change the geometry of the window.
- *XNResourceClass* and *XNResourceName* specify the resource name and class to use when the IM looks up resources which vary according to IC.
- *XNStatusAttributes* and *XNPreeditAttributes* specify to an IM the attributes to be used for any status and preedit areas. The attributes are nested variable length lists.

Preedit and Status attributes

When an IM is going to provide state, it needs some simple X information with which to do its work. For example, if an IM is going to draw status information in a client window in an off-the-spot style, it needs to know where the area is, what color and font to render text in, and so on. The application gives these data to the IC for use by the IM.

As with the "IC values" section, full details are available in [**Scheifler**].

- *XNArea* specifies a rectangle to be used as a status or preedit area.
- *XNAreaNeeded* specifies the rectangle desired by the attribute writer. If the application reads this value, it is the IM's suggestion. If the application writes it, it is suggesting to the IM.
- *XNForeground* and *XNBackground* specify the background color for rendering.
- *XNBackgroundPixmap* specifies a pixmap to be used for the background of windows the IM creates.
- *XNColormap* specifies the colormap to use.
- *XNCursor* specifies the cursor to use.
- *XNFontSet* specifies the fontset to use when rendering text.

- *XNLineSpacing* specifies the what line spacing is to be used in the preedit window if more than one line is used.
- *XNSpotLocation* specifies where the next insertion point is, for use by *XIMPreeditPosition* styles.
- *XNStdColormap* specifies that the IM should use *XGetRGBColormaps()* with the supplied Atom in order to find out which colormap to use.

Creating an input context

Creating an input context is a simple matter of calling *XCreateIC()* with a variable length list of parameters specifying IC values. A simple example:

```
XVaNestedList arglist;
XIC ic;

arglist = XVaCreateNestedList(0, XNFontSet, fontset,
                              XNForeground,
                              WhitePixel(dpy, screen),
                              XNBackground,
                              BlackPixel(dpy, screen),
                              NULL );

ic = XCreateIC(im, XNInputStyle, styleWeWillUse,
              XNClientWindow, window, XNFocusWindow, window,
              XNStatusAttributes, arglist,
              XNPreeditAttributes, arglist, NULL);
XFree(arglist);

if (ic == NULL)
    Tragedy();
```

Using the IC

A multi-window application may choose to use several input contexts for some reason. For simplicity, we will assume that the application just wants to get to the internationalized input one way in one window.

Using the IC is a matter of (a) making sure we select for events the IC wants, and (b) setting IC focus. If you are setting up a window the first time, you will know the event mask you wish, and you can use it directly. If you are attaching an IC to a previously configured window, you should ask and add in the new event mask.

```
unsigned long imEventMask;

XGetWindowAttributes(dpy, win, &winAtts);
XGetICValues(ic, XNFilterEvents, &imEventMask, NULL);

imEventMask |= winAtts.your_event_mask;
XSelectInput(dpy, window, imEventMask);

XSetICFocus(ic);
```

At this point the window is ready to be used, unless it still needs mapping.

1.8.9 Events under IM control

Processing events under input method control is almost the same as it was under X11R4 and before. There are two essential differences: *XFilterEvent()* and the *LookupString* routines.

XFilterEvent()

Every event received by your application should be fed to the IM via *XFilterEvent()*, which will return a value telling you whether or not to disregard the event. IMs will ask you to disregard the event if they have extracted the data and plan on giving it to you later, possibly in some other form. All events (not just *KeyPress* and *KeyRelease* events) go to *XFilterEvent()*.

Compacting the event processing into a single routine, a typical event loop would look like this:

```
Xevent event;

while (forever) {
    XNextEvent(dpy, &event);

    if (XFilterEvent(&event, None))
        continue;

    DealWithEvent(&event);
}
```

LookupString

When using an input method, calls to *XLookupString()* should be replaced with calls to *XwcLookupString()* or *XmbLookupString()*. The *Xmb* and *Xwc* versions have very similar interfaces. In the examples below, we arbitrarily choose *XmbLookupString()*, but the examples are general to both.

There are two new things to deal with:

1. The string returned may be long.
2. There may be an interesting keysym returned, an interesting set of characters returned, both, or neither.

Dealing with the former is merely a matter of maintaining an arena, as in the example below.

To inform the application about what is interesting about an event, *XmbLookupString()* returns a *status* which is one of the following:

- *XLookupKeysym*, which indicates the keysym should be noticed
- *XLookupChars*, which indicates a string has been typed or composed
- *XLookupBoth*, which means both of the above
- *XLookupNone*, which means neither are ready for processing.
- *XBufferOverflow*, which means the supplied buffer is too small. *XmbLookupString()* should be called again with a big enough buffer.

The example below should help show how these work. Most event processors perform a *switch()* on the event type. Assume we have done that, and we have a *KeyPress* event:

```
case KeyPress:
{
    Keysym keysym;
    Status status;
    int buflength;
    static int bufsize = 16;
    static char *buf = NULL;

    if (buf == NULL) {
        buf = malloc(bufsize);
        if (buf < 0) StopSequence();
    }
```

```
    buflength = XmbLookupString(ic, &event, buf,
                             bufsize, &keysym, &status);

    /* first, check to see if that worked */
    if (status == XBufferOverflow) {
        buf = realloc(buf, (bufsize = buflength));
        buflength = XmbLookupString(ic, &event, buf,
        bufsize, &keysym, &status);
    }

    /* We have a valid status. Check that */
    switch(status) {
    case XLookupKeysym:
    case XLookupBoth:
        DealWithKeysym(keysym);
        if (status == XLookupKeysym)
            break; /* wasn't XLookupBoth */
    case XLookupChars:
        DealWithString(buf, buflength);
    case XLookupNone:
        break;
    } /* end switch(status) */

} /* end case KeyPress segment */
break; /* we are in a switch(event.type) statement */
```

1.9 GUI concerns

It shouldn't be significantly more difficult to internationalize an application with a graphical user interface than an application without such an interface, but there are a few issues which should be addressed.

As with the application itself, the major goal of internationalizing a GUI is to make it localizable.

1.9.1 X resources for strings

Resource lookup mechanisms in Xlib as well as in toolkits monitor locale environment variables when locating resource files. For string constants which will be used within toolkit objects, resources provide a simpler solution than do message catalogs.

Some common objects which should definitely get their labels from resources:

- Labels
- Buttons
- Menu items
- Dialog notices and questions

Any object which employs some sort of text label should be labelled via resources. Since the localizer will want to provide strings for the local version of the application, the app-defaults file for the application should specify every reasonable string resource. Manual pages should identify every localizable string.

1.9.2 Layout

Layout management is especially of interest when you cannot predict how large a button (or other) label might be. The nature of the problem of layout composition and management does not change, but one must construct the layout management without full knowledge of the final appearance.

It's worth noting that localization efforts can be assumed to be "reasonable" in some sense. For example, X resources have always allowed a user to specify an extremely large font for buttons, but applications correctly choose to let such users live with the results. But it's not always that clear what is reasonable and what isn't; we don't always know what will be difficult to translate succinctly in some locale. So while developers need not provide for all combinations of resource specifications, they must take the responsibility to make the application localizable.

There are essentially three different ways of attacking the layout problem.

Dynamic layout

Most toolkits provide *form*, *pane*, *rowcolumn*, *rubberboard*, or other layout objects which will calculate layout depending on the "natural" (localized) size of the objects involved. Most use some hints provided by the developer which can regulate this layout. For example, some Motif widgets which provide these services are *XmForm*, *XmPanedWindow*, and *XmRowColumn*.

Dynamic layout is probably the simplest way to prevent localization difficulties.

Constant layout

Under certain circumstances, an application may insist on having a predefined layout. When this is so, the application must provide objects which are so constructed as to allow localization. A “Quit” button which just barely allows room for the Latin-1 string “Quit” is not likely to suffice when localizers attempt to fit their translations into that small space.

In order to enforce constant layout, the developer incurs the heavy responsibility of making sure the objects are localizable. This means a lot of investigation; the “there, that ought to be enough” approach is chancy at best.

Localized layout

Some toolkits provide for layout control by run-time reading of strings or other data files. Applications which use such toolkits can easily finesse the layout issue by providing capability for localization of the layout as well as the contents of the layout. This provides each localizer maximum freedom in presenting the application to the local users. The application developer is responsible to provide localizers with instructions and mechanism necessary to produce layout data.

1.9.3 Icons

Icons attempt to be fairly generic representations of their antecedents. Unfortunately, it is very difficult for a designer to know what is generic or recognizable in other cultures. Therefore it is important that any pictographic representations used by an application be localizable.

Graphic representations can be stored as strings representing X bitmaps, as names of data files containing pictographs, or in whatever manner the developer thinks best, so long as the developer provides a way for the localizer to produce and deliver localized pictographs.

1.10 Some Popular Encodings

1.10.1 The ISO 8859 family

American English is easily representable in 7-bit ASCII. Most other languages are not. For example, the character é is not in ASCII.

Most Western European languages are representable in 8-bit ISO 8859/1, which is commonly known as Latin 1. Latin 1 is a superset of ASCII including characters used by several Western European languages (such as ö, £, ñ, ç, ¿).

ISO 8859 comes in nine parts, many overlapping, all supersets of ASCII, as shown in Table 1-4:

Character set	Common Name	Languages supported
8859-1	Latin 1	Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish
8859-2	Latin 2	Albanian, Czech, English, German, Hungarian, Polish, Rumanian, Serbocroation, Slovak, Slovene
8859-3	Latin 3	Afrikaans, Catalan, Dutch, English, Esperanto, German, Italian, Maltese, Spanish, Turkish
8859-4	Latin 4	Danish, English, Estonian, Finnish, German, Greenlandic, Lappish, Latvian, Lithuanian, Swedish, Norwegian

Table 1-4 ISO 8859 Character sets

Character set	Common Name	Languages supported
8859-5	Latin/Cyrillic	Bulgarian, Byelorussian, English, Macedonian, Russian, Serbocroation, Ukranian
8859-6	Latin/Arabic	English, Arabic (See ISO 8859-6 spec)
8859-7	Latin/Greek	English, Greek (See ISO 8859-7 spec)
8859-8	Latin/Hebrew	English, Hebrew (See ISO 8859-8 spec)
8859-9	Latin 5	Danish, Dutch, English, Finnish, French, German, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish, Turkish

Table 1-4 (continued) ISO 8859 Character sets

IRIX contains over 500 Latin 1 fonts. At the time of writing, IRIX contains no fonts for use with other 8859-encoded character sets.

1.10.2 Asian languages

Asian languages are commonly pictographic and employ many thousands of characters for their representation. For example, Japanese and Korean can be practically encoded in 16 bits. Daily-use Chinese can also be, but archives and scholars frequently need more; Chinese is often encoded with up to four bytes per character.

Some standards

Various Asian character sets have been developed, some of which are considered standard. Encoding standards for these sets are less standard.

Asian character sets usually require larger-than-byte character types like those described in Section 1.4.2, "Large character types."

Language	Character Set Standards	Support
Japanese	JIS X 0201	*Katakana*
	JIS X 0208	*Kanji*, *Kana*, Latin, Greek, Cyrillic, symbols, others
	JIS X 0212	Supplemental *Kanji*, others
Chinese	mixed bag	
Korean	KSC5601.1987-0	Hangul

Table 1-5 Character Sets for Asian Languages

EUC

EUC is *Extended Unix Code*, an encoding methodology which supports concurrent use of four codesets in one encoding. It employs two magic bytes

```
ss2 = 0x8e
ss3 = 0x8f
```

which are used to identify codesets within a string. The EUC encoding scheme is:

```
Codeset #1 0xxxxxxx
Codeset #2 1xxxxxxx [ 1xxxxxxx ...]
Codeset #3 ss2 1xxxxxxx [ 1xxxxxxx ...]
Codeset #4 ss3 1xxxxxxx [ 1xxxxxxx ...]
```

Codeset #1 is always ASCII. Codesets 2-4 are implementation- or user-defined.

EUC implementations exist (but are not standard) for all ideographic Asian languages.

Japanese

JIS codesets are all 7-bit codes. This allows Japanese to be sent in Internet messages without modification of the messages or the transport. Plain JIS requires imbedded escape sequences in order to switch between one- and two-byte character modes and character sets.

Shift-JIS (SJIS) and *UJIS* (a Japanese EUC encoding) use a string's numeric values to switch between codesets. Both encodings use the eighth bit of bytes.

Most Unix workstations which support Japanese use EUC UJIS for internal code. UJIS codeset #2 is JIS X 208-1990, a 2-byte codeset which includes *Kanji*, *Romaji*, Greek, Cyrillic, symbols, and more. Codeset #3 is JIS X 201-1989, a one-byte codeset; by adding the *ss2* byte, each character becomes 2 bytes. Codeset 4 is not used. Thus the largest characters in UJIS are 2 bytes long.

In Japan, most PCs use SJIS, which shifts the 7-bit JIS around ASCII, becoming compatible with ASCII at the expense of using the eighth bit of bytes.

The Japanese encodings are compatible and convertible. Since JIS is a 7-bit code, SJIS and UJIS are commonly converted to JIS for network transmission.

1.10.3 ISO 10646 and Unicode

ISO and the Unicode Consortium have jointly developed a character set designed to cover almost every character normally used by any language in the world. ISO calls this *ISO IS 10646*. The Unicode Consortium embraces a subset of 10646, called the *Basic Multilingual Plane* (BMP) of 10646, and calls it *Unicode*. The only characters defined in either standard are the characters in the BMP. The characters have 2- and 4-byte representations.

It appears that ISO 10646 will grow significantly in acceptance, but that widespread use is still some years away.

1.11 REFERENCES

Nye, Adrian: *Xlib Programming Manual for Version 11 of the X Window System, Volume One*, third edition (covers Release 4 and Release 5), O'Reilly & Associates, ISBN 1-56592-002-3

Scheifler, Robert and Gettys, Jim: *X Window System 3rd Edition*, Digital Press, ISBN 1-55558-088-2

X/Open Company, Ltd.: *X/Open Portability Guide*, Prentice Hall, ISBN 0-13-685819-8 (Set of 7 Volumes)