Internationalization 7

Introduction

To compete in the global market, applications must display and print in the appropriate native language of its target markets. For example, an application running on a Solaris system has a potential worldwide market that is twice the size of the United States market, providing that it can be easily customized for each language and culture. *Internationalization* is the process of designing software that can be easily ported to a specific language or region. The main goal of internationalization is to produce one executable image that can run everywhere. An internationalized application uses a database which contains locale-specific data that can be configured for a particular environment by an end user.

Including internationalization as part of the base design will allow the software to be easily customized for each locale. This can result in significant savings of time and money over the life span of the application.

Internationalization vs. Localization

Internationalization, commonly known as I18N, involves preparing software portable between languages and regions. *Localization*, occasionally called L10N, is the process of adapting software for a specific language or region through the establishment of on-line information. The information for a language or region is called a *locale*. Internationalized software has interfaces that use the locale information to modify program behavior at runtime. Although you may have an internationalized application, you still need to localize your error messages to the native language for those countries where you plan to distribute the software.

Internationalization Levels

SunSoft defines four levels of internationalization:

- **Level 1: Text and Codesets** – Software is considered Level 1 compliant if it is *eight-bit clean*—that is, code that uses all eight bits to represent a character set. An application that uses the high bit is considered *eight-bit dirty*. The ISO 8859-1 code set, known as ISO Latin-1, requires all eight bits to represent the extended ASCII codeset.

- **Level 2: Formats and Collation** – Level 2 compliant software uses format and collation methods that are locale sensitive. These formatting methods involve date, time, currency, numbers, and units. Sorting methods and paper size are also encompassed in Level 2 compliance.

- **Level 3: User Message and Text Presentation** – In Level 3 compliant software, all user-visible text is translated at runtime. This includes help text, error messages, property sheets, buttons, and icons. Text is placed in message files that correspond to a specified locale.

- **Level 4: Asian Language Support** – At Level 4, software provides support for Asian languages.Under Solaris 2.x this is done using the Extended UNIX Code (EUC), a method of handling the multibyte character standard used to represent Asian character sets.

Issues to Consider

When writing international software, application developers should keep the following issues in mind:

- Input processing and output should be independent of codeset.
- Formats for date, time, numbers, currency, etc., should be defined separately and be selectable by locale.
- Sorting should be based on collation tables independent of codeset.
- All strings should be in separate message files for easy translation.
- Space for any display text should be flexible because strings vary in height and width based upon different written languages.
- Documentation should be in standard formats with accepted vocabulary lists.
- Conversion tables for PCs, terminals and mainframes should be selectable by locale.

Text and Codesets Compliance (Level 1)

Call setlocale()

Solaris 2.x provides support for the POSIX/ANSI C function called `setlocale()`, which initializes language and cultural conventions. Most applications should set the locale category `LC_CTYPE` except for those not concerned with character interpretation, such as block I/O to disk or network. To control dynamic handling of different codesets in an application, add these lines to your code:

```
#include <locale.h>
main() {
  (void) setlocale(LC_CTYPE, "");
}
```

Among other things, this ensures that European accented characters, such as ö and è, are correctly identified with an `isalpha()` library call. Note that the empty string argument indicates that the application should set its codeset according to the environment variable `LC_ALL`, `LC_CTYPE`, or `LANG` in that order of precedence. If none of these environment variables are set, the default locale is `C`, which results in old-style UNIX behavior.

Internally, this call changes values in the `_ctype` array of the C library. This in turn affects the behavior of various `ctype` library routines. The `LC_TYPE` locale category may also affect other functions, including wide-character handling.

To set all the locale categories at the same time, use the `LC_ALL` argument to `setlocale()` instead of `LC_CTYPE`. In practice, most applications should set the `LC_ALL` category once and for all. Applications that fail to call `setlocale()` would fail to get international features.

Making Eight-Bit Clean Code

ISO Latin-1 is an extended ASCII codeset that uses all eight bits of a byte to represent a character. This codeset includes characters with accents and special characters used in western Europe. Software can no longer use the high-order bit if it is to remain eight-bit clean. Old code that encoded the high-order bit needs to be changed. Practices of masking the high order bit or assuming characters are only seven bits long is dirty, as in this example:

```
#define MYBIT 0x80 char c; /* bad practice */
c |= MYBIT;                /* Uses high order bit  */
c = *(string+i) & 0x7F;    /* assumes 7 bit length */
```

Looking for constants in your code like “0x80”, “0x7f”, “0200”, “0177”, “127”, and “128” can often highlight potential problems in the code. Code that assumes a particular character range also needs fixing. These two lines show eight-bit dirty and eight-bit clean code:

```
if ( c >= 'a' && c <= 'z')/* bad practice */
if (islower(c) )          /* recommended way */
```

Use the codeset independent routines found in `<ctype.h>` such as `isalpha()`, `islower()` and `isprint()`. Code that assumes characters fall in the 0-127 range needs to be modified, such as in this example:

```
#include <limits.h>
static int hashtable [127];      /* bad practice */
static int hashtable [UCHAR_MAX];/* recommended way */
```

`UCHAR_MAX` is defined in `<limits.h>` on all ANSI C conforming systems.

Programming Issues

Watch out for the following when modifying your code to be eight-bit clean:

- Many compilers default to using `signed` for all fundamental data types. This can sometimes cause major problems in both application and library code.
- Code that casts `char` to other lengths may be dirty. Because the `char` data type is signed in Solaris 2.x, when a `char` variable holds an international character, the top bit is set, and sign extension takes place during assignment. A negative integer may cause a problem later on.
- Do not pass raw characters to functions that require `short`, `int`, or `long` arguments. This is bad practice because of the sign extension problem.
- Watch for use of unadorned `char`. Unfortunately, they have probably been used extensively throughout most code, making it a non-trivial task to change all `char` data to `unsigned char`, especially as this might garner some lint or compiler warnings, such as the code in this example:

```
char CHAR;
unsigned char U_CHAR;
char ch = 0xa0;
int i;
isascii (CHAR);    /* isascii assumes an integer value which may be negative */
isascii (U_CHAR);  /* recommended: Negative integer is not propagated */
i = ch;            /* i is now negative because ch is a 'signed character */
```

Although all of this may sound like a lot of work, in many cases existing code executes correctly in 8-bit mode. Watch out for coding habits that assume ASCII is the only form of character encoding available.

- Avoid managing the keyboard. If programs use system and window services to read characters, they only need to be eight-bit clean. However, if programs read from `/dev/kbd` or performs keystroke mapping, they are managing the keyboard. If that is the case, support these input methods for all keyboard layouts:
  - entering any character using a single keystroke. Note that German keyboards switch Y with Z, while French keyboards switch Q with W and A with Z.
  - using a dead-key accent followed by a given keystroke to produce any valid ISO 8859-1 (ISO Latin-1) character.
  - pressing the Compose Key followed by two additional keystrokes to produce any valid ISO Latin-1 character.

  The most obvious way to test software that manages the keyboard is to type all the Compose sequences to produce all the special ISO Latin-1 characters on each of the different country kit keyboards. If at all possible, use OpenWindows to manage the keyboard for you. This saves you time, and gives your software a uniform user interface.

Formats and Collation (Level 2)

Many different formats are employed throughout the world to represent date, time, currency, numbers, and units. These formats should not be hard-wired into your code. Instead, programs should call `setlocale()`, then the various locale-specific format routines, leaving format design to localization work for each country or language.

For string collation, sort orders may vary for different languages. Programs should use the `strcoll()` or `strxfrm()` library routines to perform string comparisons, which use a locale-specific collation order. In Solaris 2.x, the `LANG` environment variable determines the format of date, time, currency, and numbers, plus collation order. It also determines the language of error messages and help text.

Time and Date Formats

Time is represented by both a 12-hour clock and a 24-hour clock—sometimes known as "railroad time" or "military time". The hour and minute separator can be either a colon (`:`) or a period (`.`). Some countries attach letters to the time indicating that this is a time.

Time zone splits occur between and within countries. Although a time zone can be described in terms of how many hours it is ahead of or behind Greenwich Mean Time (GMT), this number is not always an integer. In addition, there is often Daylight Savings Time (DST), which starts and ends on different dates that can vary from country to country.

The following table shows some of the ways to write 11:59 p.m.:

Table 7-1 International Time Formats

Locale	Format
Canadian	23:59
Finnish	23.59
German	23.59 Uhr
Norwegian	Kl 23.59
UK	11:59 PM

The following table shows some of the date formats used around the world with April 12, 1993 as the example:

Table 7-2 International Date Formats

Locale	Convention	Example
Canadian (English)	yyyy–mm–dd	1993-04-12
Canadian (French)	yyyy–mm–dd	1993-04-12
Danish	dd/mm/yy	12/04/93
Finnish	dd.mm.yyyy	12.04.1993
French	dd/mm/yy	12/04/93
German	dd.mm.yy	12.04.93
Italian	dd.mm.yy	12.04.93
Norwegian	dd.mm.yy	12.04.93
Spanish	dd–mm–yy	12-04-93
Swedish	yyyy–mm–dd	1993-04-12
UK-English	dd/mm/yy	12/04/93
US-English	mm–dd–yy	04-12-93

The secret to producing time and date formats valid in many locales is the `strftime()` library routine. First, set the program clock by calling `time()`; then populate a `tm` structure by calling `localtime()`. Pass this structure to `strftime()`, along with a format for date and time, as well as a holding buffer, in the following example:

```
#include <locale.h>
#include <libintl.h>
#include <stdio.h>
#include <time.h>
main()
{
    time_t clock, time();
    struct tm *tm, *localtime();
```

```
    char buf[128];

    setlocale (LC_ALL,"");
    clock = time((time_t *)0);
    tm = localtime(&clock);
    strftime(buf, sizeof(buf), "%C", tm);
    printf("%s \n", buf);
}
```

The recommended formats are `%c` for the local short form of date and time, and `%C` for the local long form. Also, `%x` produces the local date form (numeric), and `%X` yields the local time form.

If you run the preceding program, you will get results similar to this:

```
% setenv LC_TIME de
% a.out
Montag, 16. März 1992, 19:19:19 Uhr PST
% setenv LC_TIME fr
% a.out
lundi, 16 mars 1992, 19:19:20 PST
```

Number and Currency Formats

The United Kingdom and the United States are two of the few places in the world that use a period to indicate the decimal place. Many other countries use a comma. Likewise, while the UK and the US use a comma to separate thousands groups, other countries use a period or a thin space.

This table shows some commonly used numeric formats:

Table 7-3 International Numeric Conventions

Locale	Large Number
Canadian (French)	4 294 967 295,00
Canadian (English)	4 294 967 295,00
Danish	4.294.967.295,00
Finnish	4.294.967.295,00
French	4.294.967.295,00
German	4 294 967 295,00
Italian	4.294.967.295,00
Norwegian	4.294.967.295,00
Spanish	4.294.967.295,00

Table 7-3 International Numeric Conventions

Locale	Large Number
Swedish	4.294.967.295,00
UK-English	4,294,967,295.00
US-English	4,294,967,295.00

Note – Data files containing locale-specific formats will be misinterpreted when transferred to a system in a different locale. For example, a file containing numbers in a French format will not be useful to a UK-specific program.

Currency units and presentation order vary greatly around the world. This table shows monetary formats in some countries:

Table 7-4 International Monetary Conventions

Locale	Currency	Example
Canadian (English)	Dollar ($)	$1 234.56
Canadian (French)	Dollar ($)	1 234.56$
Danish	Kroner (kr)	kr.1.234,56
Finnish	Markka (mk)	1.234 mk
French	Franc (F)	F1.234,56
German	Deutsche Mark (DM)	1,234.56DM
Italian	Lira (L)	L1.234,56
Norwegian	Krone (kr)	kr 1.234,56
Spanish	Peseta (Pts)	1.234,56Pts
Swedish	Krona (Kr)	1234.56KR
UK-English	Pound (£)	£1,234.56
US-English	Dollar ($)	$1,234.56

Note – Note that local and international symbols for currency can differ. For example, the designation for the French Franc is “F” in France but this is often written as “FRF” internationally to distinguish it from other Francs, such as the Swiss Franc or the Polynesian Francs.

Use the `localeconv()` function to obtain currency formats. This function reads formatting conventions for the current locale to populate a `lconv` structure, then returns a pointer to the filled-in object.

This gives you a data structure containing information, but not a string. What is needed is a library routine that will accept a decimal (`float`) number and return a string containing the appropriate monetary format. The following example returns a string in the locale's monetary format:

```
#include <locale.h>
#include <libintl.h>
#include <stdio.h>
char *pcurrency(amount)
double amount;
{
    char string [512];
    struct lconv *lconv, *localeconv();

    lconv = localeconv();
    sprintf (string, "%s%s%.2f%s%s\n",
        lconv->p_cs_precedes ? lconv->currency_symbol : "",
        lconv->p_sep_by_space ? " " : "",
        amount,
        lconv->p_sep_by_space ? " " : "",
        lconv->p_cs_precedes ? "" : lconv->currency_symbol );
    return (string);
}
main()
{
  double amount;

  (void)setlocale(LC_ALL, "");
  scanf("%lf", &amount);
  printf("%s\n", pcurrency(amount));
}
```

Replace strcmp() with strcoll()

Alphabetic ordering varies from one language to another. For example, in Spanish *ñ* immediately follows *n*, and digraphs *ch* and *ll* immediately follow *c* and *l*, respectively. In German the ligature ß is collated as if it were *ss*. Swedish has additional unique characters following *z*. Danish and Norwegian have additional characters æ, ø following z.

The traditional library routine for comparing strings, `strcmp()`, remains unchanged. Because it uses ASCII order, `strcmp()` places `'a'` after `'Z'` even in English. This ordering is often unacceptable.

By contrast, the new library routines `strcoll()` and `strxfrm()` can produce any sort order you want. Use `strcoll()` to compare strings and `strxfrm()` to transform strings to ones that collate correctly.

Fortunately, `strcoll()` routine takes the same parameters and returns the same values as `strcmp()`. Unfortunately, `strcoll()` does a lot more work and is consequently slower. To speed up applications that compare strings frequently, use `strxfrm()` to store transformed strings into arrays that collate more efficiently.

This program reads standard input, builds a binary tree in the correct order using `strcoll()` to compare strings, then prints out the binary tree. This code may be used for tasks such as listing files in a subwindow.

Code Example 7-1 String Comparison Routines

```
#include <locale.h>
#include <stdio.h>
#include <string.h>
/*
Compile and execute program. Enter in strings with a carriage return after
each. When done, <Cntrl-D> and the list will be sorted
*/
struct tnode { /* node of binary tree */
char *line;
int count;
struct tnode *left, *right; };
main() /* collate : sort a list of lines using strcoll() */
{
    struct tnode *root, *tree();
    char line[BUFSIZ];
    root = NULL;
    (void) setlocale (LC_ALL, "");
    printf("Please enter strings to be sorted. \n");
    while (fgets(line,BUFSIZ,stdin))
        root = tree(root,line);
    printf("--------Sorted List---------\n");
    treeprint(root);
}
struct tnode*  tree( p, line)  /* install line at or below tree pointer */
struct tnode *p;
char *line;
{
    char *cp, *malloc(), *strcpy();
    int cond;
    if (p==NULL) {
        p = (struct tnode *) malloc (sizeof(struct tnode));
```

Code Example 7-1 String Comparison Routines (Continued)

```
        if ((cp = malloc( strlen(line) +1 )) != NULL)
            strcpy (cp, line);
        p->line = cp;
        p->count = 1;
        p->left = p->right = NULL;
    }
    else if ((cond = strcoll (line, p->line )) == 0)
        p->count++;
    else if (cond < 0)
        p->left = tree (p->left, line);
    else  /* cond > 0 */
        p->right = tree(p->right, line);
    return (p);
}
treeprint(p) /* print the tree recursively starting at p */
struct tnode *p;
{
    if (p != NULL) {
        treeprint (p->left);
        while (p->count--)
            printf("%s", p->line);
    treeprint( p->right);
    }
}
```

User Messages and Text Presentation (Level 3)

One of the most critical tasks in software internationalization is providing messages that can be easily translated. Messages such as help text, buttons labels, menus, and error diagnostics are what end users first see. The text representation varies from language to language. Japanese often does not delimit words with a space. Position and type of punctuation symbols can vary between locales. In Spanish, ¿ and ¡ appear at the beginning of sentences, while Finnish words can have a colon (`:`) embedded in them. Words may have a gender associated with them, and a person's title can vary. For these reasons, translated messages can be of different length and height. Internationalized software needs to take into account these problems when displaying text.

The ease of message localization can vary greatly. In a well-designed application, nontechnical people can translate message files into their native languages. In a non-international application, engineers fluent in a language must translate every string inside a program, then recompile the code, creating a separate executable for each locale. There should be no explicit strings in an international application except those passed to `gettext()`.

Two similar but incompatible methods for international messaging in Solaris 2.x are `catgets()` from X/Open standard, and `gettext()` from the POSIX.1b and UniForum proposals. Both routines provide an interface to message catalogs in the form of text databases you can compose, translate, and access. Because the contents of a message catalog are separate from application code, text can be selected by locale at runtime without altering the code.

Solaris 2.x supports the existing SVR4 messaging schemes. This includes both the X/Open XPG3 message catalog scheme using `catgets()`, and the SVR4 private scheme using `gettxt()`. However, both schemes are inflexible when it comes to handling messages identified by mnemonic form. This is the rationale for the messaging interface using `gettext()`, which is based on a proposal made by the UniForum Technical Subcommittee on Internationalization. This extension can be run in conjunction with existing schemes or used as the sole technique for messaging a Solaris 2.x application. Applications that call `gettext()` must include the header file `<libintl.h>` and compile with the `-lintl` linker option.

catgets() vs. gettext()

Although `catget()` is the standard X/Open function, its programming interface is not as convenient as `gettext()`. You must pass `catgets()` two extra numeric parameters. Moreover, `catgets()` requires you to assign a numeric message identifier to each message. Message numbers can easily get out of sync with message strings, making code confusing and difficult to maintain. You should use `gettext()` instead; Solaris 2.x and OpenWindows 3.x use `gettext()` exclusively.

Using gettext()

The routine `gettext()` attempts to retrieve a target string based on the specified message ID argument within the context of a specific domain and the current locale. All you really need to do is go through your program and enclose literal strings inside `gettext()` calls. The following is an example of an error message before and after:

```
printf(" %s: file is too large \n", fname);
printf(gettext(" %s: file is too large \n"), fname);
```

In general, you only need to message a string that an end user will see. Do not create message strings containing commands or filenames. If `gettext()` cannot find an appropriate string in the message catalog, it will return the string it was passed as its first parameter. Code will not fail if message files are missing. A one-line initialization statement must be converted to two, as in the following example.

The following one-line initialization statement:

```
char *greeting = " Hello";
```

should be replaced by the following two-line initialization statements:

```
/* should be converted to */
char *greeting;
greeting = gettext("Hello");
```

Creating messages that are easily translated takes practice. Remember that word order varies between languages, so complex messages with more than two `%s` parts can be difficult to translate. *Static messaging* involves looking up strings in a message catalog with no reordering of words. *Dynamic messaging* also involves looking up strings in a message catalog, but those strings are reordered and assembled at runtime. International standards provide an ordering extension to `printf()` for dynamic messaging. Static messaging has an advantage in its simplicity. The following example shows a compound message and both a static and dynamic internationalization implementation:

```
/* poor practice */ printf("Unable to %s the %s . \n",
(lock_flg ? "lock" : "find"),
(type_flg ? "page" : record ") );
```

This could produce any of four messages in English or German:

English	German
Unable to lock the page.	Das Programm kann die Seite nicht sperren.
Unable to find the page.	Das Programm kann die Seite nicht finden.
Unable to lock the record.	Das Programm kann den Rekord nicht sperren.
Unable to find the record.	Das Programm kann den Rekord nicht finden.

German syntax requires different word order, so the program's keywords need to be reversed.

Code that produces a dynamic message might look like this:

```
printf(gettext("Unable to %s the %s. \n"),
(lock_flg ? gettext("lock") : gettext("find")),
(type_flg ? gettext("page") : gettext("record")));
```

This example might have the following message catalog:

```
msgid "Unable to %s the %s. \n"
msgstr "Das Programm kann %2$s nicht %1$s. \n"
msgid "lock"
msgstr "sperren"
msgid "find"
msgstr "finden"
msgid "page"
```

```
msgstr "die Seite"
msgid "record"
msgstr "den Rekord"
```

A static implementation, such as the one in the next example, can create four separate messages that are easy to translate.

```
if (lock_flg)
    if (type_flg)
         printf(gettext("Unable to lock page. \n") );
    else
         printf(gettext("Unable to lock record. \n") );
else
    if (type_flg)
        printf(gettext("Unable to find page. \n") );
    else
        printf(gettext("Unable to find record. \n") );
```

This example produces a message table similar to the following message table:

```
msgid "Unable to lock the page. \n"
msgstr "Das Programm kann die Seite nicht sperren. \n"
msgid "Unable to find the page. \n"
msgstr "Das Programm kann die Seite nicht finden. \n"
msgid "Unable to lock the record. \n"
msgstr "Das Programm kann den Rekord nicht sperren. \n"
msgid "Unable to find the record. \n"
msgstr "Das Programm kann den Rekord nicht finden. \n"
```

Creating a Message File

Once you have enclosed all end-user visible strings inside `gettext()` wrappers, you can run the `xgettext` command on your C source files to create a message file. This produces a readable `.po` file (the portable object) for editing by translators. The following example shows the results of `xgettext()` on a program:

```
# cat hello.c
#include<stdio.h>
#include<libintl.h>
main()
{
    printf(gettext("Hello World \n"));
    printf(gettext("Good Bye \n"));
}
# xgettext hello.c
```

```
# cat messages.po
domain "messages"
msgid  "Hello World \n"
msgstr
msgid  "Good Bye \n"
msgstr
```

Next, edit the `messages.po` file and translate the strings into the appropriate language. Upon completion, the file should look something like the next example. Running `msgfmt` on this file produces a binary message object (`.mo`) file, which should be installed under the `LC_MESSAGES` directory.

```
# cat messages.po
domain "messages"
msgid  "Hello World \n"
msgstr "Bonjour, Monde \n"
msgid  "Good Bye \n"
msgstr "Au Revoir \n"
```

Where Messages Reside

Under Solaris 2.x, system messages for libraries and utilities reside in the file `/usr/lib/locale/language/LC_MESSAGES/domain.mo` where `language` is the specific language—`fr` for French, for example—and `domain` is the specific text domain for that application.

Message files should usually be installed in the same directory hierarchy as the application software. Using Solaris 2.x, applications can associate a directory with a message domain by calling `bintextdomain()`.

Although the message file need not be named after its application, maintenance is easier when the names are similar. The `catopen()` and `textdomain()` library routines actually open the message file. If a message file is missing, end users get the untranslated message, which is a string contained in the `catgets()` or `gettext()` call.

Asian Support (Level 4)

East Asian alphabets contain more than 255 characters, so they do not fit into an eight-bit address space. As a result, Japanese, Korean, and Chinese require a multibyte codeset for language processing. Characters are a sequence of one or more bytes representing a single graphic symbol, or control code. This should not be confused with the data type `char` in C which is an eight-bit value. A multibyte character takes up more than one byte. A wide character is an integral type large enough to hold any member of an extended character set. In program terms, it is an object of type `wchar_t`, which is an implementation defined integral type defined in Solaris 2.x. A wide-character string is a contiguous sequence of wide characters terminated by and including the null wide character.

Japanese text is composed of three different writing systems mixed together: Kanji *ideographs* derived from Chinese, and two phonetic scripts, Hiragana and Katakana. Although each character in Hiragana has an equivalent in Katakana, Hiragana is more commonly used, and has cursive rather than block letter forms. Kanji characters are used to write root words, Hiragana to change the pronunciation of words, and Katakana mostly to represent foreign words. There are tens of thousands of Kanji characters, but the number commonly used has been steadily declining over time. Now only about 3500 are frequently used. Computer systems must support more than 7000 because that is what the Japanese Industry Standard (JIS) requires. On average 55% of Japanese text is Hiragana, 35% Kanji, and 10% Katakana. Arabic numerals and Roman letters are also present in Japanese text.

Korean is similar to Japanese in that Chinese-based ideographs, called Hanja, are mixed together with a phonetic alphabet, Hangul. Hanja is used mostly to avoid confusion when Hangul would be ambiguous. Hangul characters are formed by combining ten basic vowels and 14 consonants, two to five of which compose one symbol. Hangul characters are often arranged in a square so that the group takes up the same space as a Hanja character. Korean now requires almost 6000 Hanja characters, 94 Hangul elements, and 2350 Hangul symbols.

Chinese normally consists entirely of characters from the ideographic alphabet, called Hanzi. In China (the People's Republic of China), there are almost 7000 commonly used Hanzi characters; however, the emerging standards number Hanzi in the tens of thousands. In China and Taiwan (the Republic of China), current standards require more than 13,000 characters. If a character is not a root character, it usually consists of two or more parts. In two-part characters, one part usually represents meaning, while the other part represents pronunciation. There are several phonetic systems for representing Chinese. In China the most common is Pinyin, which uses Roman characters and is widely employed in the west for place names such as Beijing. Less politically correct than Pinyin is the Wade-Giles system. In Taiwan they often use Zhuyin (or Bopomofo), an extensive phonetic alphabet with unique letter form.

Multibyte Support

The following section discusses different ways in which Solaris 2.x handles multibyte characters.

Extended UNIX Code (EUC)

Solaris 2.x has adopted the Extended UNIX Code (EUC) from AT&T's Multi-National Language Supplement (MNLS). EUC is used primarily for storing data in files. EUC is comprised of four codesets, three of which may be multibyte, and two of which must be announced by 8-bit control codes known as single-shift characters. EUC's primary codeset (codeset 0) is used for ASCII. The three supplementary codesets (codesets 1, 2, and 3) can be assigned to different codesets by the locale administrator. Codeset 0 is single byte, with the most significant bit set to zero. The supplementary codesets can be single- or multi-byte, with the most significant bit set to one. Codesets 2 and 3 have a preceding single-shift character, known as SS2 and SS3 respectively, where SS2 = 0x8E (10001110) and SS3 = 0x8F (10001111). There is no SS1. Differentiating between codesets is done as follows: If the high bit is 0, the codeset is ASCII. If the high bit is 1, the byte is checked for SS2 or SS3 to determine codeset. The length (in bytes) of characters from that codeset is retrieved from the `LC_CTYPE` locale database governing character classification associated with the current locale.

Wide Characters

EUC is not convenient for internal processing because its variable-length nature complicates constructing homogeneous character arrays. To assist with internal processing, Solaris 2.x provides a wide-character format, as well as a collection of library functions for operating on wide characters. Additional functions are available to convert between EUC and wide characters. In Solaris 2.x, wide characters are four bytes long. Wide characters provide a standard character size; they are useful for indexing, interprocess communication, memory management, and other tasks that use character counts and known array sizes. Wide characters are also stateless and unambiguous in a given locale; they are ANSI C data type `wchar_t`.

Table 7-5 EUC and Wide Character Representation

Codeset	EUC Representation	Wide-Character Representation
0	0xxxxxxx	00000000 00000000 00000000 0xxxxxxx
1	1xxxxxxx 1yyyyyyy	00110000 00000000 00xxxxxx xyyyyyyy
2	SS2 1xxxxxxx 1yyyyyyy 1zzzzzzz	00010000 000xxxxx xxyyyyyy yzzzzzzz
3	SS3 1xxxxxxx 1yyyyyyy 1zzzzzzz	00100000 000xxxxx xxyyyyyy yzzzzzzz

Multibyte Library Routines

Solaris 2.x provides four routines to convert characters and strings from EUC representations to wide-character representation and back again. It also provides a set of library routines to perform standard operations on wide characters. Use `mbtowc()` to convert EUC representation to a wide character, and `wctomb()` to convert a wide character to EUC representation. For strings (arrays of characters), use `mbstowcs()` to convert EUC to wide character strings and `wcstombs()` to convert wide character to EUC strings. All these are in `libc`.

Solaris 2.x also provides library routines in `libw` to replace or supplement character and string routines in `libc`. Compile multi-byte programs using the `-lw` option to the linker.

A wide-character standard I/O package is available; its routines have the letter `w` in front of `c` (character base routines) or `s` (string based routines), such as `wsprintf()`. All the `is*()` functions are duplicated by `isw*()` functions for wide characters, and all the `str*()` functions by `ws*()` functions for wide character string operations.

Summary of Requirements

The X/Open standard and the Solaris 2.x library routines for internationalization are the standards recommended by the X/Open Consortium. However, Solaris 2.x uses the `gettext()` function instead of the X/Open `catgets()` function internally.

Here is a list of some important do's and don'ts for writing internationalized software:

DO

- call `setlocale()` to initialize language and cultural conventions
- make software 8-bit clean by verifying that it does not alter the most significant bit
- watch for sign extension problems; many compilers treat all characters as signed values
- use standard codesets such as ISO Latin-1 or EUC
- use `ctype` library routines to identify character ranges
- remember that many countries use a comma for the decimal separator
- use `strftime()`; not `ctime()`
- use `localeconv()` if you need to obtain local currency formats
- replace calls to `strcmp()` with calls to `strcoll()`
- surround user message strings with `gettext()` calls and create separate message files for easy translation
- plan for size changes in end-user visible messages after translation
- use explicit functions to center and align text
- allow punctuation characters within words
- provide translation notes, where possible (as in code comments)

DON'T

- split user visible message strings; instead, write out whole messages

- embed graphics in code
- use jargon, abbreviations, or acronyms
- make assumptions about word order in strings
- assume numbers of bytes or characters
- test for alphabetic characters by comparing specific characters, such as "`A`" and "`Z`"
- hard code the decimal separator in parsing or arithmetic calculations
- hard code or limit the size of currency fields
- assume the paper size on which output will be printed
- use single-letter commands to represent the first letter of commands, such as "`p`" for print
- make assumptions of what might be in a particular byte location
- use both `catgets`() and `gettext`()

Bibliography

Developer's Guide to Internationalization, Part Number 801-2857-10, SunSoft, Inc., Mountain View, CA.

Solaris International Developer's Guide, Bill Tuthill, SunSoft Press/Prentice Hall, Englewood Cliffs, NJ (to be published summer 1993).