Writing Internationalized Code 4

This chapter describes some specific steps that you should take to internationalize applications. The material is divided into four main topics: text and code sets, formatting and collation, user messages, and nonglobal locales.

Linking

Some internationalization components depend on dynamic linking to function correctly. The default when compiling and linking in the Solaris environment is dynamic linking. Take care not to specify static linking.

Text and Code Sets

Call `setlocale()`

The SunOS system supports the POSIX/ANSI C function `setlocale()`, which initializes language and cultural conventions. Most applications should set the locale category `LC_CTYPE` except those not concerned with character interpretation, such as block I/O to disk or network. To control the dynamic handling of different code sets 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 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 is 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`(3) library routines. The `LC_CTYPE` locale category may also affect other functions, including wide-character handling.

In most cases library packages should rely on the programmer to call `setlocale()` inside the application. If not, the call is reentrant in the sense that it affects static data structures. Applications that fail to call `setlocale()` would simply fail to get international features.

To set all the above locale categories at the same time, use the `LC_ALL` argument to `setlocale()` instead of just `LC_CTYPE`. In practice, most applications should set the `LC_ALL` category once and for all.

Make Software 8-bit Clean

Programs shouldn't alter the most significant bit of a `char`. The computer industry used this bit for parity many years ago, but it didn't work out well—data got corrupted because software ignored the parity bit. Now standards committees have decided to define 8-bit code sets, which means you have to clean up your code now. Here are some problems to look for.

Code that explicitly uses the most significant bit for its own purposes is said to be "dirty". There may be valid reasons for altering the most significant bit, but dirty code often involves setting and clearing private flags:

```
#define INVERSE 0x80        /* bad practice */
char c;
c |= INVERSE;
```

Find another way to encode this information. A trick used several times in the operating system was to extend this data type to be `unsigned short` or `unsigned int`, and later set the top bit of the new data type.

Code that assumes characters are only seven bits long is dirty. Here's an example of masking off the most significant bit on the assumption it's just the parity bit:

```
c = *(string+i) & 0x7F;/* bad practice */
```

A useful exercise is to search your code for constants like "0x80", "0x7f", "0200", "0177", "127", and "128". These constants often highlight problematic code immediately, if such bit patterns are used in conjunction with character handling.

Code that assumes a particular character range, such as:

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

must be corrected to:

```
if (islower(c))
```

Use codeset independent routines found in `<ctype.h>` such as `isalpha()`, `isprint()`, and so on. Software should have been using these functions all along, as they were always needed for portability to IBM's EBCDIC codeset. The SunOS system also provides wide-character equivalents such as `iswalpha()` and `iswprint()`.

Fix code that assumes characters fall in the range 0–127 by extending the range of such tables:

```
static int hashtable[127];                /* bad practice */
```

For example, the above declaration would be better coded as follows:

```
#include <limits.h>
static int hashtable[UCHAR_MAX];
```

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

Watch for Sign Extension Problems

One issue that is sometimes invisible to the programmer is the way the C compilers default to using `signed` for all fundamental data types. This can sometimes cause substantial 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 SunOS, when a `char` variable holds an 8-bit character that has the most significant bit set, sign extension takes place during assignment. Needless to say, a negative integer might cause problems later on:

```
int i;
char c = 0xa0;
i = c; /* i is now negative */
```

Do not pass raw characters to functions that require `short`, `int`, or `long` arguments. This is bad practice because of the sign extension problem. For example, the following code is incorrect, as it produces a negative integer index into the C library `__ctype` table. This is because the functions are actually macros that generate stubs of in-line code, which assume the argument is an integer, and propagate the sign bit accordingly.

```
char ch;
isascii(ch);
```

The code above could be written like this:

```
unsigned char ch;
isascii(ch);
```

Watch for the use of unadorned `char`s. Unfortunately they have probably been used extensively throughout most code. It is therefore a nontrivial task to change all `char` data to `unsigned char`, especially as this might garner some `lint` or compiler warnings.

So,

```
char ch;
ch = 0xA0;
```

is better written as:

```
unsigned char ch;
ch = 0xA0;
```

On the other hand,

```
char *cp;
while (isspace(*cp)) {
```

is written as:

```
char *cp;
while (isspace((unsigned char)*cp)) {
```

Although all this may sound like a lot of work, in many cases existing code executes correctly in 8-bit mode *without* any changes to the code. You are primarily looking for lazy coding habits that assume ASCII is the only form of character encoding available. When you fix problems, they are usually easy to test using the Compose key of the Type-4, Type-5, PC-AT101, and PC-AT102 keyboard.

Note that the C compiler does not support 8-bit or multi-byte characters in object names—that is, names of routines, variables, and so forth—although it does allow you to initialize 8-bit or multi-byte data in strings.

Employ Standard Code Sets

In certain locales, the OpenWindows 3.3 environment provides support for the ISO 8859-1 standard codeset, also known as ISO Latin-1. Test your software by typing all the Compose sequences in Table B-1 on page 67. If your software can display all characters in ISO Latin-1, it can also display all characters on European keyboards. For Asian locales, use code sets supported in the Asian Feature Sets.

Generating PostScript

Code that generates PostScript must use the `\`*nnn* octal form for characters above hexadecimal 7F, since the PostScript interpreter cannot read characters with the eighth bit set. If you are using TranScript 2.1.1, you can obtain the encoding for ISO Latin-1 by using the name `ISOLatin1Encoding` to reorder the font encoding. Note that `ISOLatin1Encoding` is available in TranScript 2.1.1 but not in previous releases of TranScript®.

PostScript Level Two interpreters have the name `ISOLatin1Encoding` as a defined name. See the *PostScript Language Reference Manual*—2nd edition.

Use `ctype` Library Routines

As mentioned previously, text processing software must avoid hard-coded character ranges. Upper- and lower-case letters, punctuation marks, numeric digits, and spaces should be defined using library routines under `<ctype.h>`, rather than with hard-coded character ranges:

Table 4-1 Library Routines for Codeset Independence

Routine	Character is a...
isalpha(c)	Letter
isupper(c)	Capital letter
islower(c)	Lower case letter
isdigit(c)	Digit from `0–9`
isxdigit(c)	Hexadecimal digit from `0–f`
isalnum(c)	Alphanumeric (letter or digit)
isspace(c)	White space character
ispunct(c)	Punctuation mark
isprint(c)	Printable character
iscntrl(c)	Control character
isascii(c)	7-bit character
isgraph(c)	Visible graphics character

Avoid Managing the Keyboard

Type-5, Type-4, PC-AT101, and PC-AT102 keyboards have provisions for typing all the ISO Latin-1 characters. If programs use system and window services to read characters, they need only be 8-bit clean. But if programs read `/dev/kbd`, or perform keystroke mapping, they are managing the keyboard. If that's the case, make sure to 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 and W with A and Z.

- Using a dead key accent followed by a given keystroke to produce any valid ISO 8859-1 character
- Pressing the Compose key followed by two additional keystrokes to produce any valid ISO character

The most obvious way to test software that manages the keyboard is to type all the Compose sequences in Table 3-1, and to type all keystrokes on several 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

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 routine to perform string comparisons, which use locale-specific collation order.

Note – Locale specific collation requires that the application be dynamically linked.

Time and Date Formats

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, plus a holding buffer:

```
#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);
}
```

Recommended formats are `%c` for the local short form of date and time, or `%C` for the local long form. Also, `%x` produces the local date form (numeric), and `%X` yields the local time form. If you try out the program above, your results will look something like 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
```

Unfortunately many often-used combinations of date and time are missing from the standard. Neither short nor long form of the local date is available, and there is no abbreviation for time without seconds or time zone.

Currency Formats

Use `localeconv`(3) function to obtain currency formats. It reads formatting conventions of the current locale to populate an `lconv` structure, then returns a pointer to the filled-in object.

Unfortunately, this gives you a data structure, but not a string. A library routine is needed that converts a floating-point number to a string containing the appropriate monetary format. Standards committees are working on this. Until they agree on a solution, the following is a highly simplified method to print currency in a locale-independent manner:

```
#include <locale.h>
#include <libintl.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, or `strxfrm()` to transform strings to ones that collate correctly.

Fortunately `strcoll()` 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.

```
#include <locale.h>
#include <stdio.h>
#include <string.h>
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, "");
  while (fgets(line, BUFSIZ, stdin))
    root = tree(root, line);
  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));
    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 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

One of the most critical tasks in software internationalization is providing messages that can be translated easily. Messages are what users see first: help text, button labels, menu choices, usage summaries, error diagnostics, and so forth.

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 noninternational application, engineers fluent in a language must translate every string inside a program, then recompile the code. There should be no explicit strings in an international application, except those passed to `gettext()`.

Two similar (but incompatible) methods for international messaging in the SunOS system are: `catgets()` from the X/Open standard, and `gettext()` from the POSIX.1b and UniForum proposals. Both routines provide an interface to message catalogs: text databases that are easy to compose, translate, and access. Because the contents of a message catalog are separate from application code, text can be selected by locale at run-time without altering the code itself.

The SunOS system 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 the existing schemes, or can be used as the sole technique for messaging a SunOS application. Applications that call `gettext`() must include the header file `<libintl.h>` and compile with the `-lintl` linker option.

Localized Text Handling

When creating international applications, developers usually write text strings (error messages, text for buttons and menus, and so forth) in their native language, for later translation into other languages. The SunOS system lets you define any language as the native language, and any other language as the alternate language.

The steps to localize text handling are:

1. Verify that source code uses `textdomain()` and `gettext()`, or else `dgettext()`. These functions accept native language strings as arguments and return equivalent foreign language strings.

2. Extract native language text strings from the `gettext()` functions and store them, with their foreign language equivalents, in a portable message file. This extraction can be done by hand or with the `xgettext` program.

Where Do Messages Reside?

Under teh SunOS system, system messages for libraries and utilities reside in `/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 the SunOS system, applications can associate a directory with a message domain by calling `bindtextdomain()`.

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, users get the untranslated (English or Unix) message, which is a string contained in the `catgets()` or `gettext()` call.

,Using gettext()

It is recommended that programmers use `gettext()` when writing applications for the Solaris environment, even though `gettext()` is not an official standard. The SunOS system and OpenWindows 3.3 both use `gettext()` exclusively. Here is a short program, `demotext`, that uses both the `gettext()` and `catgets()` functions:

```
#include <locale.h>
#include <libintl.h>
#include <stdio.h>
main()/* demotext.c */
{
  setlocale(LC_ALL, "");
  textdomain("demotext");
  printf("%s\n", gettext("Hello world!"));
  printf("%s\n", gettext("Goodbye."));
}
```

The second line sets the message domain with `textdomain()`, specifying the message domain, which is identical to the message file name. The other lines merely surround literal strings in `printf()` statements with calls to `gettext()`, a routine that searches the message database for key strings, returning the corresponding translated string if it can locate one. Make sure to compile with `cc demotext.c -lintl -o demotext`.

Surround Strings with gettext()

Fortunately `gettext()` is much easier to use than `catgets()`. All you really have to do is go through your programs, enclosing literal strings inside `gettext()` calls. Here's an example of an error message, before:

```
printf("%s: Too few arguments\n", argv[0]);
```

and after:

```
printf(gettext("%s: Too few arguments\n"), argv[0]);
```

Some rearrangement is required here, but not much. Library products should use `dgettext()` in place of `gettext()`, since calling sequence cannot be guaranteed, and different domains may be mixed together at random. The library developer chooses the domain name.

```
printf(dgettext("xview", "Cannot find font.\n"));
```

The call is equivalent to combined `textdomain()` and `gettext()` calls.

In general, you only need to message strings that users see. Do not message strings containing system commands or file names, such as `"sort"` or `"/dev/tty"`. Be careful when messaging strings inside `sprintf()`, which is often used to build up path names or command lines. You probably don't need to message strings used only for debugging. Because integers and decimal numbers are not strings, they don't need messaging, either.

Initialized strings require some effort. The one-line initialization statement:

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

must be converted into the two lines:

```
char *greeting;
greeting = gettext("Hello");
```

If strings must be stored in an array, be sure to declare arrays large enough to hold all possible translations. Then call `strcpy()` as follows:

```
char greeting[BUFSIZ];
strncpy(greeting, gettext("Hello"), sizeof(greeting));
```

Use `bindtextdomain()`

Many applications do not require root permission for installation, and thus cannot place their messages in `/usr/lib/locale`. Moreover, most applications need messages in their own directory hierarchy, to simplify export across a network. So most applications should use `bindtextdomain()` to associate a path name with a message domain. This routine is available in the SunOS system. Here's a sample invocation:

```
char buf[BUFSIZ];
strcpy(buf, getenv("OPENWINHOME"));
bindtextdomain("xview", strcat(buf, "/locale"));
textdomain("xview");
printf(gettext("Cannot find font."));
```

If the `LANG` environment had been set to `fr`, in the SunOS system this would obtain a translation from `$OPENWINHOME/locale/fr/LC_MESSAGES/xview.mo`.

Passing a null pointer as the second argument causes `bindtextdomain()` to return the path name associated with the first argument's message domain. Here's how the path name to message files is constructed under the SunOS system.

pathname}/$LANG/LC_MESSAGES/{*domain*}.mo

By default the {*pathname*} is `/usr/lib/locale`, although this can be changed with `bindtextdomain()`. `$LANG` comes from the user's environment, and {*domain*} is supplied by `textdomain()`.

Changing the Text Domain

The following two examples retrieve the same strings but have different effects on the text domain. The first example does not change the current text domain. The second example changes the current text domain to `library_error_strings`, then retrieves the alternate language string of `wrongbutton`.

```
message = dgettext("library_error_strings", "wrongbutton");
```

or

```
textdomain("library_error_strings");
message = gettext("wrongbutton");
```

After writing an application program, create a text domain by extracting `gettext()` strings and placing them in a file with the alternate language equivalent. The following section demonstrates this process.

Create Separate Message Files

Once you have enclosed all 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. Running `msgfmt` on this file produces a binary `.mo` file (the message object), which should be installed under the `LC_MESSAGES` directory. Here's a sample interaction on `demotext.c`:

```
% xgettext -m TRNSLT: demotext.c
% cat messages.po
domain "demotext"
msgid "Hello world!"
msgstr "TRNSLT:Hello world!"
msgid "Goodbye."
msgstr "TRNSLT:Goodbye."
% msgfmt demotext.po
```

```
% su
Password:
# mv demotext.mo /usr/lib/locale/test/LC_MESSAGES
```

In the portable object, the `msgstr` keyword indicates the search string passed to `gettext()`, while the `msgid` keyword indicates the translated string. Use the `-m` flag with `xgettext` to produce test messages by prepending some obvious string.

If `gettext()` can't find an appropriate string in the message catalog, it returns the string you passed as its first parameter. This means code won't fail if message files are missing, since by default they get indexed source strings. However, you should always deliver message files with an application, because translators may not have access to your source code. If you anticipate difficulty translating `msgid`, insert a comment, or hand-edit `msgstr` to make things clear.

See Chapter 1, "Translating Messages" for more information on creating message files.

Text Length and Height May Vary

Be aware that translated messages may be of different length and height than the original messages. Translation into certain languages, such as German, often produces longer messages than in English. Some language translations may yield shorter messages. East Asian language ideographs are usually taller than roman characters.

Window system resource files in OpenWindows 3.3 specify height and width of panel buttons and such. DevGuide 3.3 will employ these facilities. In some cases it's best to use implicit object positioning, letting the window system figure out where to place things. See Chapter 5 for more details.

Avoid Compound Messages

Creating easily translated messages is an art form that involves more than just inserting `gettext()` calls around strings. Remember that word order varies from language to language, so complex messages can be very difficult to translate properly. A common sense guideline is to avoid compound messages with more than two `%s` parts whenever possible.

There are two approaches to messaging: *static* and *dynamic*. Static messaging simply involves looking up strings in a message catalog, with no reordering taking place. Dynamic messaging also involves looking up strings in a message catalog, but those strings are reordered and assembled at run-time. International standards provide an ordering extension to `printf()` for implementing dynamic messaging.

The advantage of static messaging is simplicity. Use it whenever possible. However, avoid splitting strings across two `printf()` statements, which makes messages difficult to translate. You shouldn't need to do this anyway, since you can pass abbreviated strings to `gettext()` to retrieve longer strings in your message catalog. Alternatively, you can use a backslash to hide the line break:

```
printf("This sentence is easy to translate \
because it is included in one printf statement.\n");
```

Translation problems can arise with compound messages, especially when more than one sentence could be produced at run-time. Here is some code that would be very difficult to translate:

```
/* poor practice: multi-part compound message */
printf("%s: Unable to %s %d data %s%s - %s",
    func, (alloc_flg ? "allocate" : "free"),
    count, (file_flg ? "file" : "structure"),
    (count == 1 ? "" : "s"), perror("."));
```

Quite apart from being poor programming practice, this fragment of code would be much clearer to the reader and much easier to translate if it were split into separate print statements inside an `if-else` block that would select the correct message at run time:

```
if (alloc_flg)
    if (file_flg)
        printf("Unable to allocate %d file\n", count);
    else
        printf("Unable to allocate %d structure\n", count);
else
    if (file_flg)
        printf("Unable to free %d file\n", count);
    else
        printf("Unable to free %d structure\n", count);
```

The issue of making the objects plural is not addressed in this example because, in many languages, pluralization involves more than adding "`s`" to the end of a word.

4

Dynamic Messaging

Dynamic messaging is used when the exact content or order of a message is not known until run time. Unless done carefully, dynamic messaging causes translation problems. If the positional dependence of keywords is hard-coded into a program, code needs to be changed before messages can be successfully translated. Obviously, this defeats the purpose of internationalization.

The X/Open standard defines an extension to the `printf()` family that permits changing the order of parameter insertion. The SunOS system supports this extension. For example, the conversion format `%1$s` inserts parameter one as a string, while `%2$s` inserts parameter two. The entire format string is parameter zero.

Here's a small example of how these extensions can be used. This `printf` statement has position-dependent keywords because the verb must precede the object.

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

This could produce any of four messages in English:

```
Unable to lock the page.
Unable to find the page.
Unable to lock the record.
Unable to find the record.
```

Here are those four messages translated into German. Note that the auxiliary verb must follow, not precede, the object.

```
Das Programm kann die Seite nicht sperren.
Das Programm kann die Seite nicht finden.
Das Programm kann den Rekord nicht sperren.
Das Programm kann den Rekord nicht finden.
```

German syntax requires different word order, so the program's keywords must be reversed. Here is that `printf` statement properly written for dynamic messaging:

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

The German message catalog would then appear as follows:

```
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"
```

This example might not work on other vendors' systems because of multiple `gettext()` calls per line.

Please consider carefully the effects of dynamic messaging. You might have to reposition parameters during translation. Often this fact isn't recognized until translation actually begins, by which time it's already too late—the software would have to be laboriously re-released.

Other Languages

The SunOS 5.5 system provides a `gettext`(1) command to retrieve translated messages from a catalog. This command reads the `TEXTDOMAIN` environment variable for domain name, and the `TEXTDOMAINDIR` environment variable for the path name to the message database.

Summary of Requirements

This chapter presented the standard X/Open and SunOS library routines for internationalizing software. Use standard X/Open routines whenever possible, but use the SunOS `gettext()` function instead of the X/Open `catgets()` function, because `gettext()` has an easier programming interface than `catgets()`. If you're writing window-based software, consult Chapter 5 for further information on how to internationalize window objects.

Some messages produced by routines in `libc` are localized. The application must be dynamically linkedto be sure that the localized messages are displayed.

4

Checklists for Internationalization

Do

- Call `setlocale()` to initialize language and cultural conventions.
- Make software 8-bit clean by verifying that it doesn't alter the most significant bit of 8-bit bytes.
- Watch for sign extension problems, since by default Sun C compilers treat all characters as signed values.
- Use standard code sets such as ISO Latin-1 or EUC.
- Use `ctype(3)` library routines to identify character ranges.
- Remember that many countries use a comma for the decimal separator; both `printf()` and `scanf()` have been changed accordingly.
- Use `strftime()`, not `ctime()`.
- Use the `localeconv()` call 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 user 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 message strings—write out whole messages instead.
- Embed graphics in code.
- Use jargon, abbreviations, and acronyms.
- Make assumptions about word order in strings.
- Assume numbers of bytes or characters.
- Test for alphabetic characters by comparing specific characters— "A" and "Z", for instance
- Hard code the decimal separator in parsing or arithmetic calculations.
- Hard code or limit the size of currency fields.
- Assume the size of paper on which application output will be printed.
- Use single-letter commands to represent the first letter of commands (such as "p" for "print").
- Make assumptions about what might be in particular byte locations.

- Reference global external variables used by system libraries, such as:

```
char _cur_locale[][]
unsigned char _ctype[]
unsigned char _numeric[]
char *__time[]
struct _wctype _wcptr
int _lflag
```

- Use both `catgets()` and `gettext()`.

Using X/Open Message Catalogs

Although `catgets()` is the standard X/Open function, its programming interface is not as convenient as `gettext()`. You must pass `catgets()` two extra numeric parameters—one for the catalog descriptor, and another for the message set number. Moreover, `catgets()` requires you to assign a numeric message identifier to each message, which can make code confusing and difficult to maintain. Message numbers can easily get out of sync with their message strings.

Note – Sun recommends that programmers use `gettext()` instead, even though it is not an official standard. The SunOS and OpenWindows 3.3 environments both use `gettext()` exclusively. If source portability is a concern, be advised that `gettext()` source is freely available for integration into your product.
