The task of the application developer is to write code which accesses locale databases and thus can be localized. The development effort and any following localization efforts are orthogonal; localization is not typically the chore of the developer.

4.2 Internationalization

Internationalization means creating the condition where an application can be localized. Localized applications interact with a user using that user's language and customs. A goal of internationalization is to provide applications which can be localized without recompiling, relinking, or special reformatting.

Writing a new internationalized application is no harder than writing a new single-language application. Internationalizing an existing single-language application requires a little porting. Existing single-language applications can run without porting, but they will remain single-language applications.

4.3 Locale

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

When used as a term referring to a piece of program state, `locale` means a global state variable maintained by the C library. This variable is central to most supported internationalization interfaces.

Locale is also used in 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 `locale` variable.

The `locale` variable is set via `setlocale(3)`, which should be called as early as possible after invocation of `main()`. The `category` parameter should almost always be `LC_ALL` and the `locale` parameter should almost always be the empty string, in order to honor the locale specified by the user's environment.

```
XtSetLanguageProc( NULL, (XtLanguageProc) NULL,
    NULL );
```

This calls `setlocale()`, `XSupportsLocale()` and `XSetLocaleModifiers()`.

11. Addition of international strings to catalog.
12. Addition of button labels, dialog messages etc. into app-defaults.

3.2.2 Output

13. Use FontSets. Allow user specification of entire sets. Use Xmb or Xwc text routines which understand encodings. Use resources or catalogs for strings and icons.
14. Allow room for localization of GUI objects, provide dynamic layout control, or establish useful localization rules.

3.2.3 Inter-Client Communications

15. When reading and writing properties, convert text strings to and from `TextProperty` data structures. For window names and icon names, use `X[Get/Store]WMName()` instead of `X[Fetch/Store]Name()` and `X[Get/Store]WMIconName()` instead of `X[Get/Store]IconName()`.

3.2.4 Input

16. Connect to, use, and trust an Input Method. (For western languages, IMs are lightweight and unobtrusive.) Decide upon and use an Input Context. The easy way to do this is to use a toolkit text object.
17. Don't make presuppositions about keystroke meaning other than what the IM gives you. Don't assign special meaning to keys which may be needed in a foreign language. Don't hardcode keycodes.

3.2.5 Compiling

18. Match `-DX_LOCALE` and `-DX_WCHAR` with Xlib.

4.0 Glossary

4.1 Localization

Localizing software is the process of supplying databases of information which the software will use (at run time) in order to behave in a manner suitable for a given set of language and customs. Much localization data comes from system locale databases, but applications may have particular data of their own to be localized.

4. Fix any sorting or character "kind" problems. Remove any use of the eighth bit of ASCII characters, or other bitfield invasions. Remove text manipulation done by arithmetic, `strcmp()`, `strlen()`, etc. Use `strcoll()`, `strxfm()`, and the contents of `ctype.h`. Remember that numerical (encoding) order and alphabetic order are independent.

5. Find uses of locale-selectable formatting. Force these to use selectable formats. Use `localeconv()` and `strftime()`. Worry about:

   a. Calendar format, day-of-week strings
   b. Date format
   c. Time of day format
   d. Decimal number format
   e. Negative number format
   f. Number rounding
   g. Monetary format
   h. Currency symbols and symbol position
   i. Page dimensions
   j. "yes" and "no" strings
   k. Cardinal number shape

6. Remove all hardcoded strings. Create and use message catalogs. Provide useful defaults in case there is no catalog for a given locale. Don't use absolute catalog path names. Even individual hardcoded characters should be avoided. Replace calls to `perror()` with `pfmt()`.

7. Regular expression comparison/matching - use `wsregexp.h`

8. Yes/No confirmation - use `sgi_nl.h`

3.2 Graphics programs

Because the GL has no inherent internationalization features, we will approach this problem from an X perspective. Internationalized GL programs will have to use X input. Text output must be done by X, or by routines designed to perform locale-specific string parsing.

Most of the items listed below are often provided by toolkits or toolkit objects.

3.2.1 General

9. Follow the recipe for traditional programs, above.

10. Make sure your locale is supported:

```
if ( !XSupportsLocale() ) {
    /* Warning : Locale not supported by Xlib. */
}
XSetLocaleModifiers( "" );
```

For Xt/Motif applications call:

2.8.0.1 Label size and layout

Labels are used on buttons, menu items, feedback boxes, and more. Words used in labels are different sizes in different languages. The application should either:

1. leave some "room" for sizing localized GUI objects, and supply constraint criteria to the localizer

-or-

2. calculate layout dynamically based on the object size after the strings or graphics have been determined at run time.

Remember that guards against unreasonable localization may not be practical.

2.8.0.2 Icons

Icons attempt to be fairly generic representations of their antecedents. Unfortunately, it is very difficult for a person to know what is generic or recognizable in other cultures. Applications authors should allow localization of icons, by acquiring icon handles from message catalogs or X resources.

3.0 A recipe

Following this recipe will get most applications internationalized.

3.1 Traditional programs

"Traditional programs" means programs which run under terminal control, like most unix utilities.

1. Right after `main()` gets invoked, call

   `setlocale( LC_ALL, "" );`

   `setlabel( "UX:`*ls*`" );` /* *ls* is the application name */

   `setcat( "`*uxsgicore*`" );` /* *uxsgicore* is catalog name */

2. Locate code where you will require wchars or wchar strings and modify it. Provide gateways for these code segments for wc/mb conversions. *E.g*. data read from file descriptors is mb; where you have internal buffers requiring wc, use `mbtowc()` or `mbstowcs()`.

3. Remove any presumptions about encodings, atomicity, type, or directions of characters. Byte count does not equal character count, string length, or display length. Use mb routines as necessary. Check your character types and any use of arithmetic or bit operations on chars.

Most programmers will be protected from IMs, ICs and their interfaces by toolkits and text objects. Object authors and those who cannot find an appropriate toolkit will find that using IMs is easier than supporting localizable input themselves.

2.7.4 Drawing text

Rendering mb or wc text is easy with a toolkit object. Without one, it requires R5 Xmb or Xwc extent and rendering routines.

Xmb and Xwc routines take as a parameter a *FontSet*, created via `XCreateFontSet()`, which takes a string naming the fonts to be used. The string might simply enumerate the fonts, or it may employ wild cards. Xlib will examine the locale, determine the character sets required for that locale, and will attempt to find fonts in the string based on the encoding field of the fonts' XLFD names. In the case of wild cards, the first font found with an appropriate encoding is used. FontSet strings should come from resource files or the command line.

Here are some routines to look for. Every Xmb call has an Xwc equivalent.

```
XmbDrawString             XmbDrawImageString
XmbDrawText               XmbTextExtents
XmbTextEscapement         XExtentsOfFontSet
XmbTextPerCharExtents     XContextDependentDrawing
```

2.7.5 Inter-Client Communications

Clients place and examine properties on windows in X. Properties are globally visible text strings, and it is via these properties that different clients communicate with each other. Since there is no way of knowing the locale of the agent who wrote a property, the data type `XTextProperty` has been invented, which contains details regarding format of the property text. Reading and writing properties is done with the help of conversion routines `XmbTextListToTextProperty()`, `XmbTextPropertyToTextList()`, and `Xwc` counterparts.

While communicating with the Window Manager with convinience functions for setting the window name and icon name, use

```
XGetWMName                XStoreWMName
XGetWMIconName            XStoreWMIconName
```

2.8 GUI concerns

The effects of localized strings or icons on a GUI can be significant. It is difficult to predict the sizes of translated button labels. It is difficult to devise icons which will be meaningful in any culture. As discussed below, an internationalized application with a GUI requires more GUI planning and foresight than does a single-locale application.

You need the 3rd edition of the Scheifler book for the internationalization material. Scheifler's Chapter 13 has some (but not a lot of) expository and is the better reference. Flanagan's chapters 4 and 5 are more explanation- and example-oriented. Either book should suffice.

2.7.1 X locale and wchars

Applications must be compiled with compile-time variables `X_LOCALE` and `X_WCHAR` set the same as they were when Xlib was compiled. Most application developers will get these settings via imake config settings which come on the system with Xlib. If you are not using imake, you must verify your settings of `X_LOCALE` and `X_WCHAR`.[2]

Xlib itself provides much locale support. In order to ensure that this is possible, you must call `XSupportsLocale()` early after `setlocale()`. The locale modifiers should be set, in case the application is using the Input Methods by calling `XSetLocaleModifiers()`.

Xt/Motif applications can use `XtSetLanguageProc()` which automatically does setting of locale, checking if Xlib suppports locale and seting of the locale modifiers.

2.7.2 X Resources

X resources can greatly simplify the localization process. X resource files are located by Xlib's parsing of the `LANG` environment variables. Thus an entire app-defaults file should be created for each local. To read in these resources use `XtGetSubresources()`. Generally, it is up to the programmer to employ resources; it is up to the localizer to provide a locale-specific resource file.

A modified version of the standard tool `editres` is available to aid in generation of app-defaults. The tools is available in `ird.esd:/ird/src/editres`.

Resources should be used for fonts, all GUI strings, icons, and anything else which should be localizable.

2.7.3 Keyboard input

Graphics programs should endeavor to use toolkit objects for text I/O. If this is not practical, the application will have to manage input itself. The application should open an *Input Method* (IM), establish an *Input Context* (IC), and use the X11R5 input interfaces. These interfaces are complex; details are found in the books named above as well as in the Irix Internationalization Primer.

IMs take care of things like multistroke key composition, keyboard mappings, and input string translations. IMs provide control keysyms such as `Backspace`, `Delete`, or `Enter` as well as strings containing localized input. These input strings are available in wc or mb format as the application chooses.

2. X11R5 under Irix 4.0.x defines both -DX_LOCALE and -DX_WCHAR. X11R5 under Sherwood defines neither.

2.6.1 Time

`Strftime(3)` formats times and dates according to locale. `Asctime(3)` and `cftime(3)` have some locale support, but calls to them should be examined for possible modification. Consult the man pages for details.

2.6.2 Numbers and money

The new `printf(3)` does some simple formatting of numbers according to locale. Little support is provided for general purpose monetary or numerical formatting, but locale-specific formatting information is available to the application from `localeconv(3)`.

2.6.3 Default cultural information strings

Many applications use common simple strings which differ from locale to locale. Instead of requiring a message catalog for these, such data are available via `sgi_nl.h`. Some such items are "Yes" and "No", days of the week, etc. See `strftime(4)`.

2.6.4 Variably ordered referencing of printf arguments

Printing strings may mean solving word-order issues. For example, suppose you have two localized string variables, one each for a noun and an adjective. You still may not know how to print them. The English "*white house*" should *not* translate in Spanish to "*blanca casa*" but rather "*casa blanca*".

This problem leads to the introduction of *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.

The final solution, then, is to store printf *format strings* in message catalogs as well as translated strings. Localizers can provide format strings which use variables to specify which arguments get printed in which order.

2.7 X11R5 internationalization features

Using the X11R5 interfaces requires more than some simple code clean up and different routines. Developers who cannot use a toolkit object to get to the R5 interfaces should know about the following books in addition to the Irix Internationalization Primer:

- X Window System 3rd Edition, *Scheifler & Gettys*, Digital Press, ISBN 1-55558-088-2
- Programmer's Supplement for Release 5, *Flanagan*, O'Reilly & Associates, ISBN 0-937175-86-2

2.4.2 Printf and pfmt

`Printf(3)` supports variably ordered referencing of its arguments, so that word order can be localized by putting format strings into catalogs. `Printf` also does a small amount of locale-dependent number formatting.

Use `pfmt(3)` for error message generation. It incorporates MNLS message catalog lookup, and produces messages in a standard format.

2.5 Character manipulation

The character manipulation techniques described in Kernighan and Ritchie's C book cannot survive in the presence of internationalized characters and strings.

2.5.1 Collation

Different languages have different collation rules, and of course, different character encodings. `Strcmp(3)` and arithmetic comparisons between bytes no longer work for character or string comparisons. `strcoll(3)` and `strxfm(3)` are supplied for character and string comparisons.

2.5.2 Ctype

Determining the "kind" of a character by arithmetic is also no longer generally possible. For example:

- It is not true that all letters occur between `A` and `Z` inclusive (`á` and `à` are examples).
- Some letters may not have counterparts in upper or lower case. Case shifting cannot be done by arithmetic; indeed case itself may be meaningless. Certainly the relationship between lower and uppercase characters is not arithmetically constant.
- Numerical (encoding) order and alphabetic order are independent.
- Which characters are alphabetic, numeric, printable, &c are not the same in all encodings.

Programmers should use the ctype macros and functions to classify and modify characters for these kinds of purposes. There are also functions for all the macros traditionally supplied in ctype.h.

2.5.3 Character size

Byte count does not equal character count, string length, or display length. Characters are not guaranteed to be 7,8, 16, or 32 bits.

2.6 Cultural items

Some of the simpler or more well-understood cultural differences in computing interfaces have been attacked by giving the programmer simple (although primitive) interfaces to ask for particulars.

`Libw.a` supports routines which operate on wide characters and strings. Most `ctype` and `stdio` function is duplicated. For example, the following are defined in `libw.a`: `getwchar()`, `putwchar()`, `putws()`, `wisprint()`, `wscpy()`, `wslen()`, `wsrchr()`.

2.2.3 Mixing and matching large characters

Locale-dependent routines exist to convert strings from mb to wc and back. Applications should establish code domains where each type of character is used and should provide gateway interfaces between them. For reference, see `wctomb(3)`, `wcstombs(3)` `mbtowc(3)`, and `mbstowcs(3)`.

2.3 Reading Input

It is generally fair to assume that unless otherwise specified, data read by an application is encoded suitable for the current locale. Strings will typically be in mb format. To read wide characters directly, use `getwchar(3W)`

Device input via X events is discussed below in section 2.7.3.

The GL `qdevice()` has a hardcoded view of a keyboard. Some flexibility, particularly for Europe, is available if you queue `KEYBD` instead of individual keys, but the GL has no general solution to non-American input. There is actually no way to input (say) Chinese using old GL interfaces.

2.4 Localized strings

Localized text strings are obtained from *message catalogs* and *app-defaults*. The idea is to place all your linguistic strings into a database, and have that database selectable at runtime depending on user locale.

SVR4 supports two different and mostly incompatible interfaces to message catalogs: XPG/3 and MNLS. XPG/3 is slightly more clunky but has an edge in popularity; XPG/3 is probably more portable. For information on XPG/3 refer IRIX Internationalization Primer.

See also section 2.7.2 on app-defaults.

2.4.1 AT&T MNLS message catalogs

The SVR4 message catalog interface uses `gettxt(3)`, `settxt(3)`, `setlabel(3)`, and `pfmt(3)`. Messages have a type, and catalogs need not be explicitly opened and closed. Catalogs are compiled with `mkmsgs(1)`. `exstr(1)` can be used to help extract strings from source code.

1. Under *Irix 4.0.x*, `sizeof(wchar_t)` is 1. Under Sherwood, it is 4.

2. IRIX uses MNLS type catalogs.

Since locale is global per process, there is no inherent support provided for multilingual applications. If you wish to write a multilingual application, you will either have to:

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

-or-

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

2.2 Large character types

Western languages will usually require 1 byte per character and most Asian languages will require 1, 2 or 4 bytes per character, or some mix thereof.

Dealing with characters of unknown size and encoding means using wide (*wc*) characters or multibyte (*mb*) characters. These are described below.

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 kind; string I/O uses mb, mb code works for ASCII and ISO8859, and mb characters use less space than do wide characters.
- Applications which do heavy string manipulation typically use wc strings in the work area. Wide characters are used as necessary to provide programming ease or run-time efficiency.

2.2.1 Multibyte characters

A multibyte *character* is a stream of bytes. The character itself identifies how many bytes are in it. Mb strings cannot be interpreted from random places in the string. Mb strings efficiently use space (both data and disk space) since "extra" bytes are used only for characters which require them. Character manipulation can be painful, since finding a particular character or position in a string is nontrivial.

Since MB characters are streams of bytes, `char*` is used. A traditional ASCII string is a special case of an MB string. `Strlen(3)` continues to return the number of bytes in a string, but not the number of characters; see `mblen(3)`, `mbtowc(3)`, and `mbstowcs(3)`.

2.2.2 Wide characters

A wide character is a data object of type `wchar_t`, which is guaranteed to be able to hold the largest character on the system.[1] Indexing into an array of wide characters is easy and any character in a string can be interpreted on its own.

IRIX Internationalization in a Nutshell

Tom Paquin, Al Chang

1.0 Introduction

1.1 This Nutshell

This guide is an attempt to supply a high-level description of the steps involved in internationalizing software. Readers needing details of concepts or interfaces will find them in the IRIX Internationalization Primer.

1.2 Nationalized software Vs Internationalized software

Porting applications which are written for a specific language or culture is simple: programs should just compile and run. 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. But internationalized software should cater to a multiple of languages without depending on any language specific feature. By the process of localization, internationalized software can be made to run on new languages too.

2.0 The Basics

2.1 Standards

Irix is not inventing any internationalization mechanisms or interfaces. We are introducing support for ANSI, Posix, XPG/3, X11R5, and SVR4 internationalization standards.

2.1.1 Multilingual support (lack of)

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.

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.

The two scenarios above are examples of multilingual applications.