OpenWindows 4

Introduction

The OpenWindows product has changed significantly in Solaris 2.x. The strategic direction for the OpenWindows product was to build it on standards. That is, OpenWindows provides developers with Application Program Interfaces (APIs) that are widely used, making the product source compatible and, if possible, binary compatible with other MIT-based X Window System products. Of course, binary compatibility requires use of similar hardware and system software.

The OpenWindows product is based on the following components:

- OPEN LOOK Window Manager
  The familiar Inter Client Communication Conventions Manual (ICCCM) compliant window manager.
- OLIT/DevGuide
  OpenWindows is focusing on the MIT Xt-based OLIT toolkit. This will be the premier toolkit promoted by SunSoft, Inc. DevGuide, the graphical interface builder, has been enhanced to provide full support for the OLIT user interface.
- X11 Server
  OpenWindows Version 4 will be based on MIT X11 Release 5 server; it will adhere to all of MIT's standard APIs with SunSoft value-added components.
- Display PostScript
  OpenWindows Version 4 will support the Display PostScript® System (DPS) as an extension to the server.

In addition, all the desktop integration has been preserved from earlier releases to standardize communication protocols. Figure 4-1 illustrates the OpenWindows components.

New features of the OpenWindows server include:

- standardized graphics hardware support through the device-dependent X (DDX) layer
- support for multiple hardware color maps (MHC) and multiple plane groups (MPG)

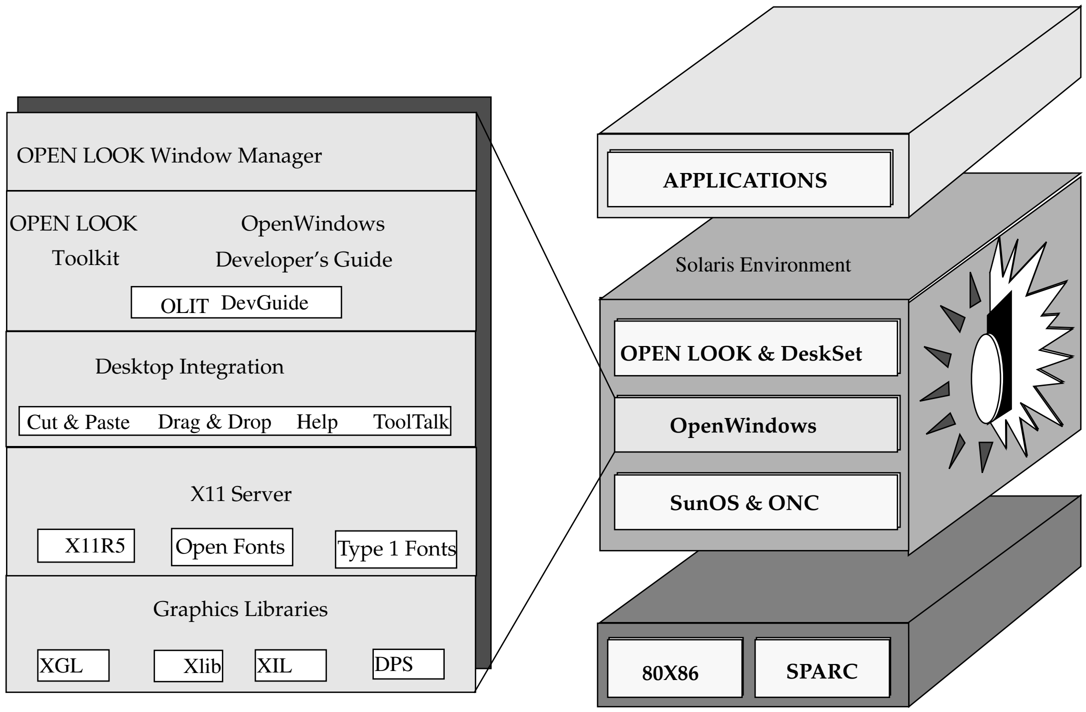

Figure 4-1 OpenWindows Components

- Direct Graphics Access (DGA)
- support for OpenFonts, Type 1 fonts, and PostScript® fonts
- support for dynamically-loadable extensions
- standardized input device interface using X Input Extensions
- enabling technology
- new server options
- internationalization
- device-independent color support
- Display PostScript System (DPS) extension to provide *de facto* standard PostScript imaging

Server performance will be comparable to MIT's Sun Sample Server. There will be no support for SunView™ or NeWS®/tNt with this release.

New OLIT features include:

- new widgets (`NumericField`, `TextLine`, `FileChooser`, `FontChooser`)
- bug fixes
- internationalization support
- new input focus mode based on supercaret
- support for ANSI C and C++
- unified Desktop look and feel (use of OLGX)
- improved programmer's interface for ScrollingList

OPEN LOOK Intrinsics Toolkit

The OPEN LOOK Intrinsics Toolkit (OLIT) is based on MIT's X Toolkit Intrinsics (Xt) and was jointly developed by AT&T and Sun Microsystems, Inc. Beginning with OpenWindows Version 3.0, SunSoft, Inc. became responsible for the development and enhancement of the OLIT toolkit.

OpenWindows Version 3.1, bundled with Solaris 2.x, is the last fully featured release of the three toolkits currently supported by OpenWindows. The three toolkits are OLIT, XView™, and tNt.

Note – For the next major OpenWindows release, all efforts will go towards the development and enhancement of one toolkit, OLIT. At the same time, XView and tNt will begin to be phased out. Please contact your SunSoft marketing representative for more information on the dates and times for this phase-out.

This section of the porting guide is aimed at:

- Developers who are evaluating the OLIT toolkit and would like to know what new features have been added to the latest release of OLIT—in particular, developers who are thinking of moving from other toolkits (XView, tNt, Motif) to OLIT.
- OLIT developers who would like to enhance their existing product with new features of OLIT.

This section discusses some of the new OLIT features introduced in the latest releases of OpenWindows. Each feature is listed and then described in detail, with some sample programs to give you an understanding of how to utilize the feature. This section describes new features in OLIT 3.1 and 3.2.

Bug Fixes

Although bug fixes are not features, they are one of the most critical points in evaluating the toolkit. Since the introduction of 3D OLIT (OLIT 2.5) in early 1991, OLIT has gone through several revisions, and heavy testing, and a large number of bugs have been fixed. The toolkit has matured rapidly to become stable with state of the art features of GUI toolkits. In addition to bug fixing, a significant amount of work has been done to improve OLIT performance and fix memory leaks in the widgets.

ANSI C Function Prototypes

With Solaris 2.x, ANSI C has become the default compiler. One of the things ANSI C requires is function prototypes. All OLIT header files are now ANSI C-compliant and have function prototypes. The header files have been updated to work with C++. Therefore, you are strongly encouraged to use either the ANSI C or C++ compiler to be compatible with the Solaris 2.x environment.

Internationalization

Internationalization (I18N) in the context of Graphical User Interface (GUI) software, refers primarily to adding the flexibility needed to operate in situations where English and the associated subset of the Roman script are not the only way you can interact textually with the interface. That is, the toolkit is flexible about entering and rendering text in different scripts and writing systems.

Localization (L10N) refers to providing support for a particular *locale*. This entails storing information about the locale’s script, then writing a system and cultural environment so the internationalized software can “operate” in that locale. Locales for which information is stored are called *supported* locales. Each locale has supporting information provided as a Feature Package by SunSoft.

An application developed and compiled with international OLIT will be able to operate in any of the supported languages and process data according to the rules of that language. The first version of international OLIT supports Japanese, Korean, Chinese, and traditional Chinese, with other Asian locales to follow. European locale handling is also supported, if supported by the operating system. Adding new locales to an international OLIT application consists primarily of providing a new set of messages, labels, and other strings which can be non-intrusively stored in application default files in parallel directory hierarchies, one for each locale. OLIT automatically picks up these defaults from the appropriate hierarchy, depending on the locale. The only actual changes to code can be minimized to the following actions:

- Removing hard-coded values for string resources in the program; delegating these to defaults files.
- Changing the default text format from its single-byte default.

- Specifying the style of interaction with the input method (for Asian locales).

The *input method* is the method by which text is entered into the system. OLIT uses the portable Xlib API to access the X Input Method. Although OLIT can use the default input method implementation provided by SunSoft for each supported locale, OLIT operates equally well with input method implementations from OEMs or independent software vendors (ISVs).

System Requirements

OLIT supports Level 4 internationalization. The first release of internationalized OLIT, Asian OpenWindows, enables developers to localize to several Asian languages. To run this release of international OLIT, you need Solaris 2.x and the Feature Package for the locale in which you intend to run it. For example, you need the Japanese Feature Package (JFP) for Japan and the Korean Feature Package (KFP) for Korea. These packages consist of extensions to Solaris and incorporate numerous facilities for handling local linguistic and cultural conventions.

Note – Level 4 is a Sun Microsystems internal definition of a level of internationalization. For more information, see the chapter on "Internationalization."

Overview of Internationalizing Applications

To internationalize your application, you must address the following issues:

- **Locale Setting**
  Locale setting is the method by which the language or cultural environment is set.
- **Character Encoding**
  Character encoding is the method by which a language's character set is represented. Conventional applications use 7-bit ASCII encoding to represent each character. However, some languages have larger character sets that require more than the 128 character range permitted by 7-bit encoding.
- **Font Set Handling**
  Font handling is simple for the languages conventional OLIT supports because they only use one character set. However, some languages use multiple character sets and therefore require multiple fonts or a *font set*.
- **Localized Text Handling**
  The international developer should be able to use error messages, menu text, and button text in the native language and have those strings retrieved in the language specified by the locale.

- **Input Method:**
  The input method is the method by which end users enter text in a particular language. To enter data into a conventional application, the end user types in the information to be processed. Some languages, however, use multiple alphabets which require several keystrokes to create one character.

- **Standards**
  Software internationalization is supported by many standards organizations, including IEEE (POSIX), ANSI, X/Open, and the MIT X Consortium. To make applications portable across a wide variety of hardware platforms, it is important to use a toolkit, such as international OLIT, that follows these standards as much possible.

Locale Setting

The X Toolkit Intrinsics, on which OLIT is layered, provides an application resource, called `xnlLanguage` (class `XnlLanguage`), which announces the end user's locale to the toolkit and the operating system.

Resource	Type	Default	Access
XtNxnlLanguage	XtNstring	NULL	I

OLIT supports all locales that are supported by SunOS on any given system. As mentioned earlier, locale support is added by installing the locale's Feature Package. Sun currently supports most European locales (such as `de`, `fr`, and `it`) and four Asian locales: `ko`, `ja`, `zh`, and `zh_TW`. There are four ways to establish the locale:

1. Specify `–xnlLanguage` on the command line of an OLIT application. For example, to set the locale for an application called `myapplication` to Korean, you type:

```
% myapplication –nxlLanguage ko
```

2. Specify `*xnlLanguage: <language>` in an X11 Resource Manager database file. For example, to set the locale to traditional Chinese, add the following line to your `.Xdefaults` file:

```
*xnlLanguage: zh_TW
```

3. Set the OpenWindows locale category `BasicLocale` through the Workspace Prop Sheet and optionally the four categories `DisplayLang`, `Numeric`, `InputLang`, and `Time`.

4. Set the `LANG` environment variable in the shell that you are starting the application from. For example, to set the locale to Japanese with `csh`, type:

```
% setenv LANG ja
```

To set the locale to Chinese, type:

```
% setenv LANG zh
```

These four ways of establishing locale are listed in order of precedence. Establishing the OS locale is the responsibility of OLIT and Xt. You should not use the OS function `setlocale()` in your OLIT applications because OLIT already calls this function internally and depends on that state to configure its internals.

Character Representation and Text Formats

International OLIT supports three character representation types, or *text formats:*

- Single byte, which represents each character with one byte.
- Multibyte, which represents characters with a variable number of bytes. Each locale has a unique definition of variable width multibyte characters.
- Wide character, which represents each character with a fixed number of bytes (4 in SVR4).

You can create single-byte, multibyte, or wide character OLIT objects, or a combination of these in a single application. If you are writing an application intended for single-byte locales only, you may want to use single-byte text format to avoid the performance overhead incurred by processing multibyte text. However, if you use single-byte format, you lose the flexibility of handling locales that contain multibyte characters, such as Japanese, Chinese, and Taiwanese.

International OLIT supports the single-byte text format primarily for backward compatibility.

The multibyte text format supports the use of ASCII. While programming with multibyte text can be difficult because each character can potentially be a different size, on the positive side, the multibyte format uses memory efficiently. This format is, therefore, useful for storing display-only text which needs to be manipulated infrequently.

Wide character text format, on the other hand, is easier to program since all characters are represented with the same number of bytes. However, the wide character format consumes more storage because it represents all characters with a fixed number of bytes. If all characters are ASCII, many of the bytes will be superfluous.

When deciding which text format is appropriate for a user interface, you should take the following considerations into account:

- Conversion between formats reduces performance.
- OLIT honors the requested text format which can be bound to each widget.
- Processing multibyte data is inherently more time-consuming than processing single-byte or wide character data. Objects that perform intensive data manipulation (for example, a text-editing object such as OLIT's `TextEdit` or `TextField` widgets) will perform better if created as wide character objects (or as single-byte objects as an optimization, if it is certain that only 8-bit data will be handled by the object).

You should design your application so that conversion between formats is minimized. For example, you may want to decide on an object-by-object basis if textual data will be processed intensively and use wide character format if it is.

Note – The widget makes assumptions based on the text format. For example, if you specify the text format of a widget to be single byte and supply a wide character label, an error results.

OLIT widget resources for presentation text are of type `OlStr`. For more information on this type, see "Setting the Default Text Format for an Entire Application".

To provide support for multiple text formats, international OLIT introduces the `XtNtextFormat` resource. This resource allows you to inform widgets what text format to expect for resources for presentation text. Note that this is an interface for programmers only; there is no equivalent interface for end users to specify widget text formats. This allows you to decide the text format of objects and freeze them; so you can then use this information to decide whether you need to convert the text format when you transfer data from one object to another, thereby improving the performance of your applications. Also note that the text format of an object is persistent for the lifetime of the object; it is established at object creation time and cannot be changed.

There are several ways to set the text format:
- Do nothing. If you do nothing, the text format defaults to single byte.
- Set the default text format for the entire application with `OlSetDefaultTextFormat().`
- Set the text format for an individual widget by changing the widget's `XtNtextFormat` resource.

Setting the Default Text Format for an Entire Application

OLIT maintains a default text format that widgets inherit when they are created without a specified text format by one of the `XtCreateWidget()` family of functions.

For compatibility with previous OLIT releases, the default text format is single byte unless you change it. To change it, use the `OlSetDefaultTextFormat()` function, which is defined as follows:

```
void OlSetDefaultTextFormat(OlStrRep format);
```

where `format` specifies the character representation of the text.

The `format` argument can have the following values:

`format` Value	Meaning
OL_SB_STR_REP	Single-byte character representation
OL_WC_STR_REP	Wide character representation
OL_MB_STR_REP	Multibyte character representation

Your application should call `OlToolkitInitialize()`, then call `OlSetDefaultTextFormat()`, and then create the widget hierarchy. Objects subsequently created will have the text format specified in the most recent call to `OlSetDefaultTextFormat()`, unless overridden by explicit arguments. Note that if the application is single byte only, it does not need to call `OlSetDefaultTextFormat()`.

You can create a widget hierarchy consisting of a combination of multibyte, wide character and single-byte objects. To do this, either change the default text format between widget creation calls or specify an object's text format explicitly when creating it.

Setting the Text Format for an Individual Widget

OLIT provides a resource, `XtNtextFormat`, that allows you to set the text format for individual widgets.

Resource	Type	Default	Access
XtNtextFormat	XtROlStrRep	OL_SB_STR_REP	GI

The `XtNtextFormat` resource can have any of the following values:

Value	Meaning
OL_SB_STR_REP	Single-byte character representation (for backward compatibility)
OL_WC_STR_REP	Wide character representation
OL_MB_STR_REP	Multibyte character representation

You can only specify the XtNtextFormat resource once for each widget at creation time. You cannot set it with XtSetValues(). The XtNtextFormat resource automatically propagates to the component widgets of a compound widget. For example, if you set it for a TextField widget, it will automatically be set for the TextEdit widget component of the TextField widget.

Font Set Handling

To represent data that consists of multiple character sets, Xlib provides the notion of a *Font Set*. An XFontSet is an X11 Release 5 data structure that supports this notion. From the user's perspective, an XFontSet represents a list of X11 Logical Font Description (XLFD) fonts that allows the application to fully represent the characters used in a particular locale. For full details on Font Sets, refer to the X11 Release 5 documentation.

To specify a font set in a resource file, use a comma-separated list of fonts. For example, you can enter the following in your .Xdefaults file for a Japanese fontset:

```
*font: -misc-fixed-medium-r-normal--20-200-75-75-c-100-iso8859-1, \
 -sun-gothic-medium-r-normal--14-120-75-75-c-120-jisx0208.1983-0, \
 -sun-gothic-medium-r-normal--14-120-75-75-c-60-jisx0201.1976-0
```

When you internationalize your application, you should specify fonts in a resource file. If you specify them within your application, it will be impossible for others to localize it. OLIT handles XFontStruct and XFontSet in a unified way, as is described in the next section.

OlFont

OLIT unifies support for both the font and font set notions by introducing the OlFont type for objects that display text. OlFont is an opaque pointer type whose interpretation depends on the setting of the XtNtextFormat resource of the associated object. If you create an object as a multibyte or wide character, the value of the OlFont field will be a valid XFontSet identifier. If you create an object as single byte, the OlFont field will be a valid pointer to an XFontStruct.

If the text format is OL_SB_STR_REP and a font set has been specified (using a comma-separated list), the first font in the list will be used to construct an Xfont structure.

Figure 4-2 illustrates OlFont overloading.

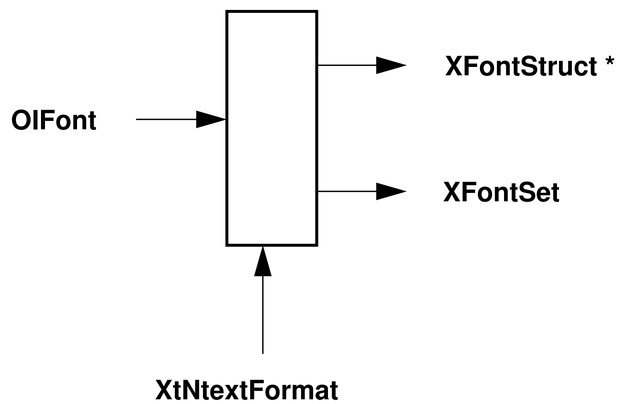

Figure 4-2 OlFont Overloading

OlStr

OLIT has introduced the data type `OlStr` to represent strings. The `OlStr` data type behaves similarly to the `OlFont` data type in the way `XtNtextFormat` is used to overload it:

- If the text format is `OL_SB_STR_REP`, then `OlStr` is interpreted as a string of single-byte characters.
- If the text format is `OL_MB_STR_REP`, then `OlStr` is interpreted as a string of multibyte characters.
- If the text format is `OL_WC_STR_REP`, then `OlStr` is interpreted as a string of wide characters.

Values for `OlStr` resources in resource files should always be specified in single-byte or multibyte format and never in wide character format. OLIT converts the fields to wide character format internally if the object's text format is `OL_WC_STR_REP`.

Figure 4-3 illustrates `OlStr` overloading.

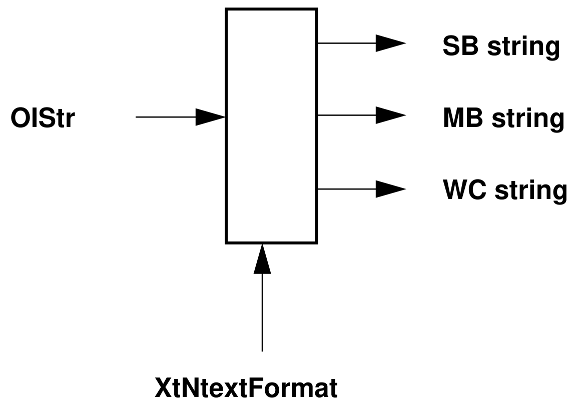

Figure 4-3 OlStr Overloading

Getting the Default Font or Font Set

The OLIT 3.1 widget set provides an `XtNolDefaultFont` resource which specifies the default font or font set for an application. This resource is discussed in the "Common Resources" section of the *OLIT 3.1 Reference Manual* and the on-line man pages.

If no font is specified for a widget's `XtNfont` resource, then `XtNolDefaultFont` determines the widget's font. In international OLIT, the default value of `XtNolDefaultFont` is determined as follows:

- In the C `locale`, the default value of `XtNolDefaultFont` is lucida sans serif with `Resolution_X` and `Resolution_Y` set to 75.
- In other locales, the default value of `XtNolDefaultFont` is a font set required to display all the characters in the code set of the locale.

OLIT provides a convenience routine, `OlGetDefaultFont()`, which enables you to get the font or font set that will be used if you do not set the `XtNfont` resource for a widget. The syntax of `OlGetDefaultFont()` is:

```
OlFont OlGetDefaultFont(widget w);
```

where `widget` is a widget in the application for which you want to get the default font.

Note that the returned value will not be the same as the value for `XtNolDefaultFont` in the following cases:

- If the XtNdefaultFont resource is set. OlGetDefaultFont() returns the XtNdefaultFont value instead of XtNolDefaultFont.
- If XtNolDefaultFont specifies a font which is not available, OlGetDefaultFont() returns a null font name.

Localized Text Handling

OLIT provides resources that enable you to set various text strings (messages, menu labels, button labels, etc.) in an application. For information on the text string resources for a particular widget, see the reference section for the widget in the *OLIT 3.1 Reference Manual*. When you create an internationalized application, you should remove all these resources from your application and keep them in locale-specific resource files.

Using Internationalized Help

You can register multibyte help text for a widget with OLIT's OlRegisterHelp() function. To register single-byte help, use the same mechanism. Single-byte is a subset of multibyte text; therefore, it works correctly and without overhead because help is display-only. The syntax for OlRegisterHelp() is as follows:

```
void OlRegisterHelp(
    OlDefine id_type,
    XtPointer id,
    String tag,
    OlDefine source_type,
    XtPointer source);
```

To use OlRegisterHelp() with international OLIT, specify one of the following values for the id_type argument:

id_type Value	**Meaning**
OL_WIDGET_HELP	Specifies multibyte format help text for an individual widget (for backward compatibility)
OL_FLAT_HELP	Specifies multibyte format help text for a flat widget (for backward compatibility)

If you specify OL_DISK_SOURCE for the source_type argument, you must specify a single-byte or multibyte filename for the source argument.

Help searches the directories specified by XFILESEARCHPATH for the specified filename. For a description of how XFILESEARCHPATH is used, see the description of the function XtResolvePathname in the *Xt Reference Manual*. If you set the filename from within your program as a full path, help ignores XFILESEARCHPATH.

For a description of the other arguments of OlRegisterHelp(), see the section entitled “Help Functions” in the *OLIT 3.1 Reference Manual* or the on-line man pages.

Locale Announcement

To announce locale, OLIT registers a private language procedure with the Intrinsics during OlToolkitInitialize(). For information on this interface, see XtLanguageProc in the *Xt Reference Manual*.

Do not register an application-specific language procedure. If you do, it will interfere with OLIT’s locale-announcement mechanism and you will have to take responsibility for it in your application. If your application requires processing before OLIT’s language procedure is called, you can provide your own procedure and call OLIT’s procedure from it. To get the language procedure registered by OLIT, use the call

```
    olit_proc = XtSetLanguageProc(NULL, NULL, NULL);
```

New Multibyte and Wide Character Text Buffer Utility Functions

International OLIT provides multibyte and wide character equivalents to the single-byte OLIT 3.1 text buffer functions. A list of these functions and their syntax is provided in Appendix C.

Input Method

The input method (IM) is the algorithm by which users enter the text of a language. The input method for each language may be different, depending on the linguistic structure and conventions of that language. OLIT interfaces with Xlib objects that form the top layer of this functionality. Internally, Xlib may communicate with another process, commonly known as the IM server, to do the actual work.

International OLIT follows the X Window System Version 11, Input Method Specification, Draft 3.0 [XIM90]. This specification was derived as a result of discussions among X Consortium members on standardizing the input handling of characters in various languages by X clients.

In English, end users enter the desired text by typing in a sequence of letters to create a word. However, for languages based on ideographic characters, input is more complicated. For example, there are two phonetic alphabets in Japanese—Hiragana and Katakana—and the traditional ideogrammatic alphabet, Kanji. In any piece of writing, all three alphabets may be used. Japanese words can also be spelled out phonetically in English; this is called Romaji.

To handle languages which do not have a one-to-one key to character mapping, input methods provide features such as the following:

- A control key sequence, which selects the *input mode.*
- A pre-edit region, which displays characters as the end user enters them but before the end user *commits* them.
- A lookup choice region, which displays ideographic characters and allows the user to choose one.
- A status region, which provides information such as whether conversion is activated and the state or mode of the input method.

The use of these features varies with the input method. For more information, see the documentation for the input method you are using. OpenWindows provides input methods for all locales it supports.

Figure 4-4 shows the input method screen regions.

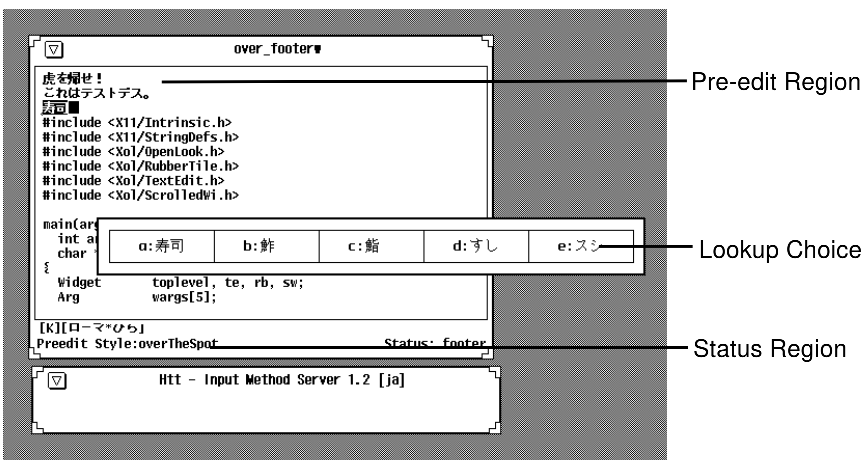

Figure 4-4 Input Method Screen Regions for "ja" Locale

Setting the Input Method Pre-edit and Status Styles

There are two aspects of the input method that an OLIT application can control:

- The *pre-edit style*, which specifies where and how pre-edit data is presented.
- The *status style*, which specifies where status feedback is presented.

OLIT provides two new resources that specify the pre-edit and status styles: `XtNimPreeditStyle` and `XtNimStatusStyle`.

XtNimPreeditStyle

The `TextEdit` resource, `XtNimPreeditStyle` (`XtCImPreeditStyle` class) selects the pre-edit style.

Resource	Type	Default	Access
`XtNimPreeditStyle`	`XtROlImPreeditStyle`	`OL_NO_PREEDIT`	`GI`

If the specified style is not supported by the input method, the ability to pre-edit is lost. The currently supported pre-edit styles are listed in the following table.

Table 4-1 Supported Pre-edit Styles

XtNimPreeditStyle Value	Meaning
`OL_ON_THE_SPOT/"onTheSpot"`	IM directs the application to display the pre-edit data.
`OL_OVER_THE_SPOT/"overTheSpot"`	IM displays pre-edit data in its own window over the application window.
`OL_ROOT_WINDOW/"rootWindow"`	IM displays pre-edit data outside the application in a window that is visually separate from the application.
`OL_NO_PREEDIT/"none"`	IM does not display pre-edit data.

Figure 4-5 shows an example of the `onTheSpot` pre-edit style. Note that the pre-edit data is shown in reverse video. When the user commits the data, it is sent to the client and displayed in normal video.

Figure 4-5 `onTheSpot` *Pre-edit Style for "ko" Locale*

Figure 4-6 shows an example of the `overTheSpot` pre-edit style. The difference between this style and `onTheSpot` is that `overTheSpot` obscures data in the application during pre-edit, while `onTheSpot` inserts data into the application without obscuring data.

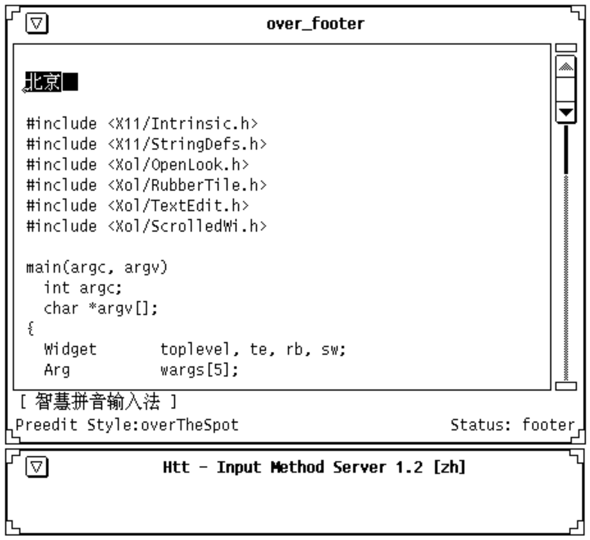

Figure 4-6 `overTheSpot` *Pre-edit Style for "zh" Locale*

Figure 4-7 shows an example of the `rootWindow` pre-edit style.

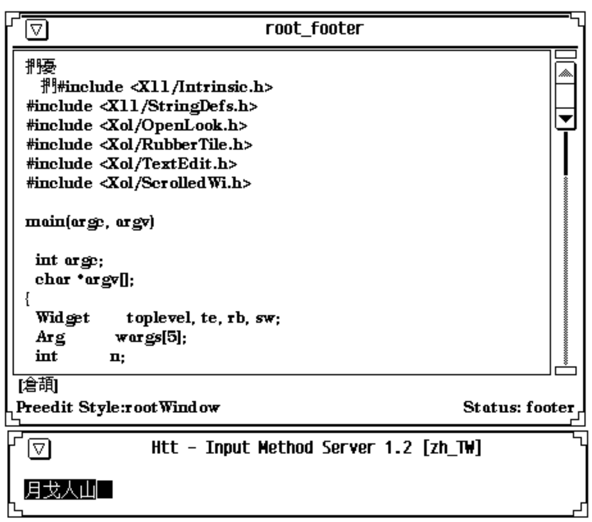

Figure 4-7 `rootWindow` *Preedit Style for "zh_TW" Locale*

See the X11 Release 5 documentation for a full description of each of the pre-edit styles.

XtNimStatusStyle

The application resource, `XtNimStatusStyle` (`XtCImStatusStyle` class), determines the style of IM status feedback.

Resource	Type	Default	Access
`XtNimStatusStyle`	`XtROlImStatusStyle`	`OL_IM_STATUS_NONE`	`GI`

The supported styles are:

XtNimStatusStyle Value	Meaning
`OL_IM_STATUS_FOOTER/"footer"`	IM generates status feedback in the footer of the shell window.
`OL_IM_STATUS_ROOT/"rootWindow"`	IM generates status feedback in a separate window visually separate from the application.
`OL_IM_STATUS_NONE/"none"`	IM doesn't generate status feedback.

When `XtNimStatusStyle` is set to `OL_IM_STATUS_FOOTER`, a number of other resources are available to set characteristics of the application footer. See "Application Resources for the IM Footer" below.

Figure 4-8 shows an example of the `footer` status style.

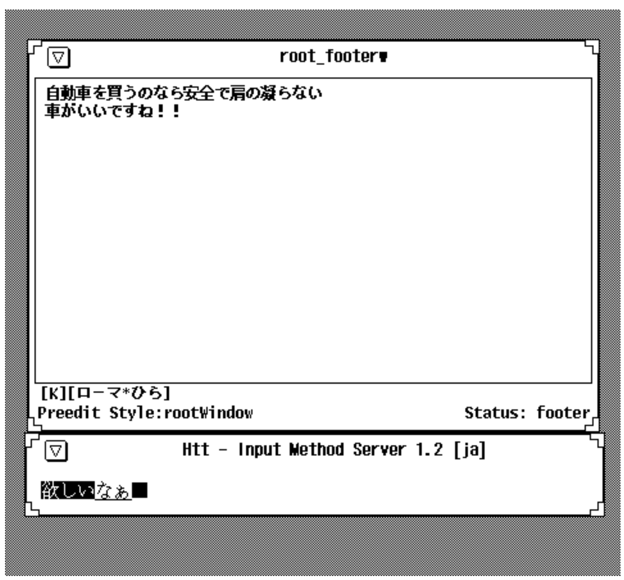

Figure 4-8 `footer` *Status Style for "ja" Locale*

Figure 4-9 shows an example of the rootWindow IM status style.

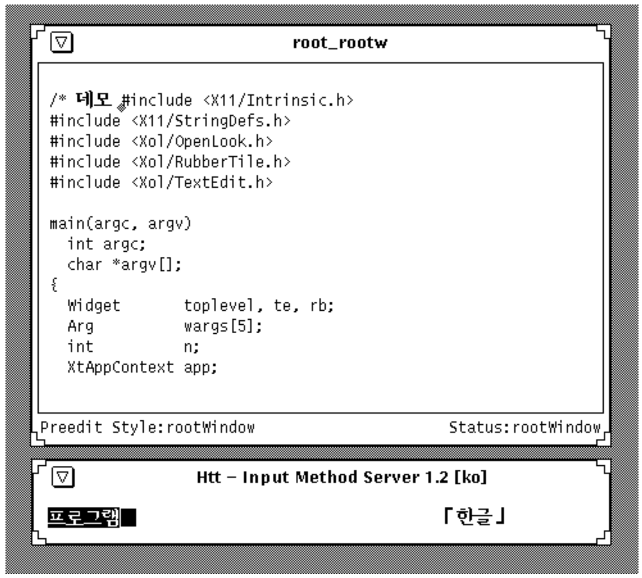

Figure 4-9 rootWindow *Status Style for "ko" Locale*

Application Resources for the IM Footer

Table 4-2 Summary of Footer Resources

Resource	Type	Default	Access
XtNimForeground	Pixel	XtDefaultForeground	SGI
XtNimBackground	Pixel	XtDefaultBackground	SGI
XtNimBackgroundPixmap	Pixmap	NULL	SGI
XtNimFontSet	OlFont	XtDefaultFont	SGI
XtNdefaultImName	String	NULL	SGI

- XtNimFontSet
  The XtNimFontSet resource specifies the font set to be used by the IM to display pre-edit and status feedback.

- XtNdefaultImName
  The XtNdefaultImName resource specifies the string to identify the IM Server. This is equivalent to specifying XMODIFIERS="@im=<string>" where you need to specify <string> in the resource.

OLIT Internationalization Example

To internationalize a simple OLIT application:

- Remove any resources that contain display text from the application code and put them in a resource file.
- Specify the text format for widgets that display text (by specifying a default text format for the application or by specifying the text format for individual widgets).
- Specify a font set for resources that contain `OlFont` values (such as `XtNfont`) in the resource file.

Suppose you want to write an internationalized application that displays a `StaticText` widget with some wide character text in it. To do this, you might write code similar to the following:

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/StaticText.h>

main(argc, argv)
    int   argc;
    char *argv[];
{
    Widget toplevel, msg_widget;
    XtAppContext app;

    /* Initialize the OLIT toolkit */
    OlToolkitInitialize((XtPointer)NULL);

    /* Set the default text format to wide character */
    OlSetDefaultTextFormat(OL_WC_STR_REP);

    toplevel = XtAppInitialize(&app, "Memo",
                    (XrmOptionDescList)NULL,
                    0, &argc, argv, NULL,
                    (ArgList) NULL, 0);

    /* Create a staticText widget. */
    msg_widget = XtVaCreateManagedWidget("msg",
                        staticTextWidgetClass,
                        toplevel,
                        NULL);

  /* Realize the widgets and enter the event loop. */
  XtRealizeWidget(toplevel);
  XtAppMainLoop(app);
}
```

This application does not specify the string that appears in the `StaticText` widget. You must specify this text in the resource file, as follows:

```
*StaticText.string: <text>
```

Notice that this application differs from a conventional OLIT application; it calls `OlSetDefaultTextFormat()` to set the default text format. `OlSetDefaultTextFormat()` sets the default text format for any widgets in the application that display text.

To set the text format for an individual widget, set its `XtNtextFormat` resource. For example, in the previous application, create the `StaticText` widget as follows:

```
msg_widget = XtVaCreateManagedWidget("msg",
                        staticTextWidgetClass,
                        toplevel,
                        XtNtextFormat, OL_WC_STR_REP,
                        NULL);
```

For some locales, you may also need to specify a font set for the application. To do this, add the following line to the resources file:

```
    *StaticText.font:   <font set>
```

Figure 4-10 shows the application with Korean text. Figure 4-11 shows the application with Japanese text.

Figure 4-10 Hello World in Korean

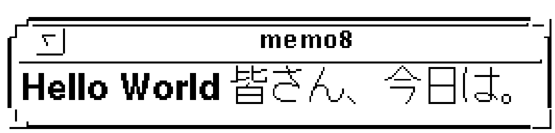

Figure 4-11 Hello World in Japanese

Standards

Internationalized OLIT is based on three standards: Internationalized UNIX, MIT X11 Internationalization Standard, and Internationalized Extension of OPEN LOOK Specification.

For more information on support of internationalization in the toolkit, see the “Internationalization Tutorial” and “Text Buffer Utility Functions for Internationalization in OLIT” sections of Appendix C.

Unified Desktop Look And Feel

With the release of OLIT 3.2, all OLIT objects that require rendering use OLGX . This will make OLIT objects look more like XView objects, such as the DeskSet applications. For most developers, this transition will be automatic. Developers will benefit from a more consistent look for their applications on the the OPEN LOOK desktop. However, if you are subclassing widgets and want to do custom rendering, you will need knowledge of the OLGX API and how to migrate your code from OLG to OLGX or write new rendering code. A white paper addressing this topic will soon be available through public distribution channels. Look for this paper to learn more about using OLGX to develop applications with custom rendering.

ScrollingList Widget (New Features)

Developers have lamented the lack of support from the OLIT toolkit in managing the `ScrollingList` widget. In Solaris 2.x, OLIT has an enhanced API for this widget. Resources and convenience routines have been added to help you manage the `ScrollingList` widget. What follows is an overview of the widget and a sample program demonstrating the associated changes.

The `ScrollingList` widget can now be created in one of the following modes:

`OL_EXCLUSIVE`	Lets you choose one item from a list of items; at any given time one item is current or chosen.
`OL_EXCLUSIVE_NONESET`	Variation of the “exclusive” scrollinglist. Here you can choose one or none of the items on the list.
`OL_NONEXCLUSIVE`	Lets you choose none, one, or multiple items from the list.
`OL_NONE`	None of the preceding three modes. For backward compatibility. This is the *default* mode for the scrollinglist. In this mode, you may *not* use any of the new functions or resources that follow.

Other features include making an item insensitive, double-clicking, item list management now done by widget, and resources and convenience functions.

To avoid confusion, all the new resources and convienence functions use the new data type `OlSlistItemPtr` as the scrollinglist item, as opposed to `OlListToken`.

Note – Do **not** mix the old and new API. The new API is sufficient in the sense that it provides equivalent routines for the old ones. For example, you cannot use the old API method of doing an `XtGetValues()` of the `XtNapplAddItem` resource and then use new convenience functions. Instead, you must use `OlSlistAddItem()` specifying the `OlSlistItemAttrs` structure, as shown in the following section.

New Item Attribute Structure

There now exists a new item data structure, `OlSlistItemAttrs`, which is used in the `OlSlistAddItem()` function to indicate the attributes of interest for a scrollinglist item. Its source code is shown in Code Example 4-1.

Code Example 4-1 OlSlistAddItem Attributes

```
#define OlItemLabelType (1L << 0)
#define OlItemLabel (1L << 1)
#define OlItemImage (1L << 2)
#define OlItemSensitive (1L << 3)
#define OlItemCurrent (1L << 4)
#define OlItemMnemonic (1L << 5)
#define OlItemUserData (1L << 6)
```

The following structure exists for each `scrollinglist` item:

```
typedef struct {
        long    flags; /* indicates which fields in this
                          struct are valid */
        OlDefine  label_type;
        XtPointer item_label;
        XImage    *item_image;
        Boolean   item_sensitive;
        Boolean   item_current;
        unsigned char item_mnemonic;
        XtPointer user_data;
} OlSlistItemAttrs, *OlSlistItemAttrsPtr;
```

The following table describes the attributes of the `OlSlistAddItem` structure:

Attribute	Function
`flags`	Marks which fields in this structure are valid or to be used.
`label_type`	Identifies the type of label to display for the item. It can have one of the following values: `OL_STRING`, `OL_IMAGE`, or `OL_BOTH` (for string and image).
`item_label`	The text label for this item

Attribute	Function
`item_image`	The glyph associated with this item.
`item_sensitive`	Indicates whether item active (receive events) or inactive.
`item_current`	Indicates whether the item is current (chosen).
`item_mnemonic`	A single character that is used as a mnemonic accelerator.
`user_data`	For application use

New Callback Structures for ScrollingLists

The following callbacks are new to OLIT and are used with the `ScrollingList` widget:

Code Example 4-2 OlSlistCallbackStruct

```
typedef struct {
 int reason;
 XEvent *event;
 OlDefine mode;
 OlSlistItemPtr item;
 int item_pos;
 OlSlistItemPtr *cur_items;
 int num_cur_items;
 int *cur_items_pos; /* not implemented */
} OlSlistCallbackStruct;
```

Code Example 4-3 OlSlistUserDeleteCallbackStruct

```
typdef struct {
 int reason;
 XEvent *event;
 OlDefine mode;
 OlSlistItemPtr *user_del_items;
 int num_del_items;
} OlSlistUserDeleteCallbackStruct;
```

The following table describes the parameters of these structures:

Parameter	Definition
`reason`	Indicates the reason for the callback.
`mode`	Indicates what mode the scrollinglist is in. Can be `OL_EXCLUSIVES`, `OL_EXCLUSIVES_NONESET`, or `OL_NONEXCLUSIVES`.
`event`	Points to the XEvent that triggered the callback. Can be NULL.
`item`	Last current item at the time of the event that triggered the callback. This points to temporary storage that the application must copy into its own space as needed.
`item_pos`	Specifies the position of the item in the list.

Parameter	Definition
cur_items	Specifies a list of current items at the time of the event that triggered the callback. When the mode is OL_NONEXCLUSIVE, this gives a list of all the current items.
OL_EXLCUSIVE and OL_EXCLUSIVE_NONESET	This variable Has the same value as the field 'item.' This points to temporary storage that the application must copy into its own space if needed.
num_cur_items	Gives the number of current items in the list.
cur_items_pos	An array of integers, representing one for each current item. This points to temporary storage that the application must copy into its own space if needed.

Changes in Existing Resources for ScrollingList Widget

The following change has been implemented in existing resources to support the ScrollingList widget:

- XtNuserDeleteItems

Type:	XtCallbackList
Default:	NULL
Access:	SGIO
Action:	Defines the callbacks issued when the end user tries to delete items from the list. Currently, the ScrollingList widget handles deletions through a cut operation. The call_data parameter in OL_NONE mode is OlListDelete, as documented in the *OLIT Reference Manual*; for all other modes it is OlSlistUserDelCallbackStruct, and the reason field is set to OL_REASON_USER_DELETE_ITEMS. It is currently up to the application to act on each item by calling OlSlistDeleteItem or OlSlistDeleteItems and providing necessary feedback.

New Resources for ScrollingLists

The following X resources have been added to the OLIT toolkit:

- XtNscrollingListMode

Type:	Oldefine
Default:	OL_NONE
Access:	GI
Action:	Defines the scrollinglist's mode—exclusive or non-exclusive.

Values:

`OL_EXCLUSIVE`	Exclusive mode—one item current at all times.
`OL_EXCLUSIVE_NONESET`	Exclusive mode—one item or none current at all times.
`OL_NONEXCLUSIVE`	Non-exclusive mode—one or many current.
`OL_NONE`	Nothing—for backward compatibility.

- `XtNitemCurrentCallback`

Type:	`XtCallbackList`
Default:	`NULL`
Access:	SGIO
Action:	Defines the callbacks to be invoked when an item is made current by an end user pressing the mouse button to select an item when the mode is anything other than `OL_NONE`, the reason field in `OlSlistCallbackStruct` is set to `OL_REASON_ITEM_CURRENT`. If the mode is `OL_NONE` (default value), then only `XtNuserMakeCurrent` callbacks will be called.

- `XtNmultiClickCallback`

Type:	`XtCallbackList`
Default:	`NULL`
Access:	SGIO
Action:	Defines the callbacks to be invoked when the SELECT mouse button is pressed on an item multiple times. The reason field in `OlSlistCallbackStruct` is set to `OL_REASON_DOUBLE_CLICK`.

The time span for a button click followed by another in milliseconds is determined by the `XtNmultiClickTimeout` resource. This activates `XtNitemCurrentCallback` callbacks.

- `XtNfirstViewableItem`

Type:	`OlSlistItemPtr`
Default:	calculated
Access:	SG
Action:	Specifies which item is the first viewable item in the list. The default is the first item added to the list.

- `XtNlastViewableItem`

Type:	`OlSlistItemPtr`
Default:	calculated
Access:	SG
Action:	Specifies which item is the last viewable item in the list.

- `XtNviewableItems`

Type:	`OlSlistItemPtr *`
Default:	`NULL`
Access:	G
Action:	Gives the viewable list of items in the widget. The number of items in this list is determined by the resource `XtNviewHeight`. If the widget is empty, `XtNviewableItems` it returns `NULL`.

- `XtNitems`

Type:	`OlSlistItemPtr *`
Default:	`NULL`
Access:	G
Action:	Gives the entire list of items in the widget. The number of items in this list is determined by the resource `XtNnumItems`. If the widget is empty `XtNitems` returns NULL.

- `XtNnumItems`

Type:	`int`
Default:	`0` (zero)
Access:	G
Action:	Defines the total number of items in the list at any given time. It is automatically updated by the `scrollinglist` whenever an item is added or deleted.

- `XtNcurrentItems`

Type:	`OlSlistItemPtr *`
Default:	`NULL`
Access:	G
Action:	Points to an array of `OlSlistItemPtr` that are to be set current in the list. `XtGetValues` for this resource returns the items that are currently current and the application must not free these items.

- `XtNnumCurrentItems`

Type:	`int`
Default:	`NULL`
Access:	G
Action:	Specifies the number of items current. The value must be the number of items in `XtNcurrentItems` and must not be negative.

- `XtNitemNotCurrentCallback`

Type:	`XtCallbackList`
Default:	`NULL`
Access:	SGIO
Action:	Defines the list of callbacks to be invoked when an end user presses the SELECT mouse button on a current item. The reason field in `OlSlistCallbackStruct` is set to `OL_REASON_ITEM_NOT_CURRENT`.

New Convenience Functions

The following convenience functions have been added to OLIT:

```
void OlSlistTouchItem(Widget widget, OlSlistItemPtr item)
```

A function that the application should use when it changes the attributes of an item.

```
void OlSlistUpdateView(Widget widget, Boolean ok)
```

A function that the application can call to keep the `ScrollingList` widget from updating the view or to let it again update the view. If `ok` is `TRUE`, then the `ScrollingList` widget can update the view as it changes; if `ok` is `FALSE`, then no update.

```
OlSlistItemPtr OlSlistAddItem(Widget widget, OlSlistItemAttrsPtr item_attr,
OlSlistItemPtr ref_item)
```

A function that adds an item before `ref_item`, given the attributes of the item and returns a pointer to the item. The value of `ref_item` can be zero to append the new item to the bottom of the list.

```
void OlSlistDeleteAllItems(Widget widget)
```

A function that deletes all items from the scrollinglist.

```
void OlSlistDeleteItem(Widget widget, OlSlistItemPtr item)
```

A function that deletes an item from the scrollinglist.

```
void OlSlistDeleteItems(Widget widget, OlSlistItemPtr *items, int num_items)
```

A function that deletes items from the scrollinglist.

```
void OlSlistMakeAllItemsNotCurrent(Widget widget)
```

A function that makes all items not current.

```
void OlSlistMakeItemNotCurrent(Widget widget, OlSlistItemPtr item, Boolean
issue_callback)
```

A function that makes an item not current in the scrollinglist and also removes it from the current list; issues an `XtNitemNotCurrentCallback` if `issue_callback` is `TRUE`.

```
Boolean OlSlistIsValidItem(Widget widget OlSlistItemPtr item)
```

A function that returns `TRUE` if the specified item is a valid item in the list, else it returns `FALSE`.

```
void OlSlistMakeItemCurrent(Widget widget, OlSlistItemPtr item, Boolean
issue_callback)
```

A function that makes an item/row current and adds it to the list of current items and issues a current callback if the boolean value `issue_callback` is `TRUE`.

```
void OlSlistFirstViewableItem(Widget widget, OlSlistItemPtr item)
```

A function that sets the first viewable item in the list.

```
void OlSlistLastViewableItem(Widget widget, OlSlistItemPtr item)
```

A function that sets the last viewable item in the list.

```
OlSlistItemPtr OlSlistGetNextItem(Widget widget, OlSlistItemPtr item)
```

A function that returns the next item, given a valid `item`. If `item` is the last item in the scrollinglist, then it returns `NULL`.

```
OlSlistItemPtr OlSlistGetPrevItem(Widget widget, OlSlistItemPtr item)
```

A function that returns the previous item, given a valid `item`. If `item` is the first item in the scrollinglist then it returns `NULL`.

```
OlStr OlSlistGetItemLabel(Widget widget, OlSlistItemPtr item)
```

A function that returns the item label given the item; returns `NULL` if no label is present.

```
XImage * OlSlistGetItemImage(Widget widget, OlSlistItemPtr item)
```

A function that returns the item image/glyph given the item; returns `NULL` if no image is present.

```
OlDefine OlSlistGetItemType(Widget widget, OlSlistItemPtr item)
```

A function that returns the item type given the item/row. Valid values are:

OL_STRING	if item just has a string.
OL_IMAGE	if item just has a image/glyph.
OL_BOTH	if item has OL_STRING and OL_IMAGE.
OL_NONE	if item is not a valid item in the list.

A function that returns what mode the scrollinglist is in, such as OL_EXCLUSIVE, if the widget

```
OlDefine OlSlistGetMode(Widget widget)
```

is in exclusive mode, or OL_NONEXCLUSIVE for nonexclusive mode.

```
Boolean OlSlistIsItemCurrent(Widget widget, OlSlistItemPtr item)
```

A function that returns TRUE if the item is current, else FALSE.

```
Boolean OlSlistGetItemSensitivity(Widget widget, OlSlistItemPtr item)
```

A function that returns TRUE if the item is sensitive, else FALSE.

```
XtPointer OlSlistGetItemUserData(Widget widget, OlSlistItemPtr item)
```

A function that returns the item's user data (if any).

```
void OlSlistSetItemAttrs(Widget widget, OlSlistItemPtr item, OlSlistItemAttrs*
set_attrs)
```

Set the attributes pointed to in set_attrs on the specified item.

```
void OlSlistGetItemAttrs(Widget widget, OlSlistItemPtr item, OlSlistItemAttrs*
get_attrs);
```

This function returns the 'item' attributes in get_attrs.

```
void OlSlistEditItem(Widget widget, Boolean insert, OlSlistItemPtr ref_item)
```

This function allows programmatic editing of a scrollinglist item. If `insert` is TRUE then `ref_item` identifies the item before where a new item is to be inserted. If `insert` is TRUE then `ref_item` can be null, in which case a new item is added to the end of the list. If `insert` is FALSE then `ref_item` refers to the item that should be edited and must be a valid item in the list.

```
void OlSlistEndEdit(Widget widget)
```

A function that the application should call when the end user has finished editing an item.

The following program uses some of the widgets described in this section:

Code Example 4-4 Sample Program Using ScrollingList Widget

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/RubberTile.h>
#include <Xol/ScrollingL.h>
#include <Xol/ControlAre.h>
#include <Xol/OblongButt.h>
#include <dirent.h>

main(int argc, char **argv)
{
  XtAppContext app;
  Widget      toplevel, rt, sl, ca, delete, findcurrent;
  String      directory;
  void        initList(Widget, String);
  void      current_callback(Widget, XtPointer, XtPointer);

   (void) OlToolkitInitialize((XtPointer)NULL);
  toplevel = XtAppInitialize(&app, "Scrollinglist",
        (XrmOptionDescList)NULL, 0,
        (Cardinal *)&argc, argv,
        (const char **)NULL,
        (ArgList)NULL,
        0);

  /*
   * Get the command line argument, if any
   */
  directory = argv[1];
  rt = XtVaCreateManagedWidget("rt", rubberTileWidgetClass,
                toplevel,
```

Code Example 4-4 Sample Program Using ScrollingList Widget (Continued)

```
        NULL);
  /*
   * Create the ScrollingList widget
   */
  sl = XtVaCreateManagedWidget("sl", scrollingListWidgetClass,
                 rt,
        XtNviewHeight, 6,
        XtNscrollingListMode, OL_EXCLUSIVE,
        NULL);
  XtAddCallback(sl, XtNitemCurrentCallback, current_callback, NULL);

  XtRealizeWidget(toplevel);
  initList(sl, directory);

  XtAppMainLoop(app);
}

void
initList(Widget sl, String directory)
{
  OlSlistItemAttrs     item;
  DIR           *dirp;
  struct dirent *dp;

  if(directory == NULL)
    directory = ".";
  dirp = opendir(directory);
  if(dirp == NULL) {
    perror(directory);
    exit(-1);
  }
  OlSlistUpdateView(sl, False);
  for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp)) {
    item.flags = OlItemLabelType | OlItemLabel | OlItemMnemonic;
    item.label_type = OL_STRING;
    item.item_label = XtNewString(dp->d_name);
    item.item_mnemonic = NULL;
    OlSlistAddItem(sl, &item, 0);
  }
  closedir(dirp);
  OlSlistUpdateView(sl, True);
}

void
current_callback(Widget w, XtPointer client_data, XtPointer call_data)
```

Code Example 4-4 Sample Program Using ScrollingList Widget (Continued)

```
{
  OlSlistCallbackStruct *token = (OlSlistCallbackStruct *)call_data;
  printf("String selected is %s \n", OlSlistGetItemLabel(w, token->item));
}
```

TextLine Widget

OLIT 3.2 introduces the `TextLine` widget, a light-weight one-line text input widget. The TextLine widget obsoletes the `TextField` widget, providing similar functionality with much less memory consumption. The `TextField` widget remains in the toolkit for compatibility reasons, but its use is discouraged.

The `TextLine` widget is a one-line input field for text data. Once the input focus is moved into the widget, keyboard entry is allowed. If the input value exceeds the length of the input field, the ScrollButtons will appear. Hidden text can then be scrolled into view by pressing the ScrollButtons. Pressing the buttons continuously scrolls the text repeatedly.

The following shows a representation of the `TextLine` widget.

Figure 4-12 TextLine Widget

TextLine Widget Visual Components

The `TextLine` widget has the following visual components:

- right-justified bold label followed by a colon, at the left of the text line
- one point (for Monochrome or 2D) or chiseled underline
- ScrollButtons

For more detail on the programmer's interface for the `TextLine` widget, please consult the man page for the `TextLine` widget in Appendix C “OpenWindows Source Code” in this book.

The following is a sample program using the `TextLine` widget:

Code Example 4-5 TextLine Widget Sample Code

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/ControlAre.h>
```

Code Example 4-5 TextLine Widget Sample Code (Continued)

```
#include <Xol/TextLine.h>
main(int argc, char **argv)
{
    XtAppContext app;
    Widget toplevel, control, tl, tl1;
    void postmodify_cb(Widget, XtPointer, XtPointer);
    void commit_cb(Widget, XtPointer, XtPointer);

    (void) OlToolkitInitialize((XtPointer)NULL);
    toplevel = XtAppInitialize(&app, "Scrollinglist",
        (XrmOptionDescList)NULL, 0,
        (Cardinal *)&argc, argv,
        (const char **)NULL,
        (ArgList)NULL,
        0);

    control = XtVaCreateManagedWidget("control",
        controlAreaWidgetClass,
        toplevel,
        NULL);

    tl = XtVaCreateManagedWidget("textline",
        textLineWidgetClass,
        control,
        XtNcaptionLabel, "GetResults: ",
        XtNcharsVisible, 15,
        XtNmaximumChars, 80,
        XtNstring, "Type something here",
        NULL);

    tl1 = XtVaCreateManagedWidget("textline1",
        textLineWidgetClass,
        control,
        XtNcaptionLabel, "PutResults: ",
        XtNcharsVisible, 15,
        XtNmaximumChars, 80,
        XtNstring, "TextLine Widget Test",
        NULL);

    XtAddCallback(tl, XtNpostModifyCallback, postmodify_cb, (XtPointer) tl1);
    XtAddCallback(tl, XtNcommitCallback, commit_cb, (XtPointer) tl1);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(app);
}
```

Code Example 4-5 TextLine Widget Sample Code (Continued)

```
void
postmodify_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
    char buff[80];
    OlTLPostModifyCallbackStruct *postmodify_cdata =
        (OlTLPostModifyCallbackStruct *)call_data;
    Widget tl = (Widget) client_data;

    if(postmodify_cdata->reason == OL_REASON_POST_MODIFICATION)
    {
        sprintf(buff, "OL_REASON_POST_MODIFICATION: %s",
            postmodify_cdata->buffer);
        XtVaSetValues(tl, XtNstring, buff, NULL);
    }
    else if(postmodify_cdata->reason == OL_REASON_PROG_POST_MODIFICATION)
    {
        sprintf(buff, "OL_REASON__PROG_POST_MODIFICATION: %s",
            postmodify_cdata->buffer);
        XtVaSetValues(tl, XtNstring, buff, NULL);
    }
}

void
commit_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
    char buff[80];
    OlTLCommitCallbackStruct *commit_cdata =
        (OlTLCommitCallbackStruct *)call_data;
    Widget tl = (Widget) client_data;

    sprintf(buff, "OL_REASON_COMMIT: %s", commit_cdata->buffer);
    XtVaSetValues(tl, XtNstring, buff, NULL);
}
```

NumericField Widget

OLIT 3.2 introduces a specialized `TextLine` widget called the `NumericField` widget. This widget allows applications to have a one-line GUI object that controls numeric values. The widget provides delta buttons that increment and decrement the value as specified by the application.

Figure 4-13 contains a representation of the `NumericField` widget.

Figure 4-13 NumericField Widget

NumericField Widget Description

The `NumericField` widget is a one-line input field for numeric text. Once the input-focus is moved into the widget, keyboard entry is allowed. If the input value exceeds the length of the input field, ScrollButtons will appear.

Hidden text can then be scrolled into view by pressing the ScrollButtons. Pressing the buttons continuously scrolls the text repeatedly with a user-adjustable delay.

DeltaButtons can be used to increment or decrement the value in an appropriate manner.

Visual Components

The `NumericField` widget contains the following visual components:
- right-justified bold label followed by a colon at the left of the TextLine
- optional one point (for Monochrome or 2D) or chiseled underline
- ScrollButtons
- Increment and Decrement buttons, otherwise known as DeltaButtons

For more detail on the programmer's interface for the `NumericField` widget, please consult the man page for the `NumericField` widget in Appendix C "OpenWindows Source Code" in this book.

The following is a sample source file using the `NumericField` widget:

Code Example 4-6 NumericField Widget Sample Source

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/ControlAre.h>
#include <Xol/NumericFie.h>
main(int argc, char **argv)
{
XtAppContext app;
Widget toplevel, control, tl;
void delta_cb(Widget, XtPointer, XtPointer);
int intVal, minVal, maxVal, delVal;
```

Code Example 4-6 NumericField Widget Sample Source (Continued)

```
(void) OlToolkitInitialize((XtPointer)NULL);
toplevel = XtAppInitialize(&app, "Scrollinglist",
(XrmOptionDescList)NULL, 0,
(Cardinal *)&argc, argv,
(const char **)NULL,
(ArgList)NULL,
0);
control = XtVaCreateManagedWidget("control",
controlAreaWidgetClass,
toplevel,
NULL);
intVal = 0;
minVal = -100;
maxVal = 100;
delVal = 2;
tl = XtVaCreateManagedWidget("numericfield",
numericFieldWidgetClass,
control,
XtNcaptionLabel, "NumericField: ",
XtNvalue, &intVal,
XtNdelta, &delVal,
XtNmaxValue, &maxVal,
XtNminValue, &minVal,
NULL);
XtAddCallback(tl, XtNdeltaCallback, delta_cb, NULL);
XtRealizeWidget(toplevel);
XtAppMainLoop(app);
}
void
delta_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
OlNFDeltaCallbackStruct *nf_cdata = (OlNFDeltaCallbackStruct *)call_data;
if(nf_cdata->reason == OL_REASON_INCREMENT)
printf("user incremented. new value is: %d\n", (int)(*nf_cdata->value));
else
printf("user decremented. new value is: %d\n", (int)(*nf_cdata->value));
}
```

FileChooser Widget

The `FileChooser` widget is a new widget introduced in the OLIT 3.2. This widget provides an easy interface to the UNIX file systems for end users. The benefit of this functionality inside the toolkit is that now all applications can provide a uniform GUI interface to file systems that

OPEN LOOK users are accustomed to. The `FileChooser` widget has a standard API that is easy to incorporate and yet extensible to provide flexibility to developers. This section contains a technical outline of the widget, along with a small sample program.

The following is a representation of the `FileChooser` widget:

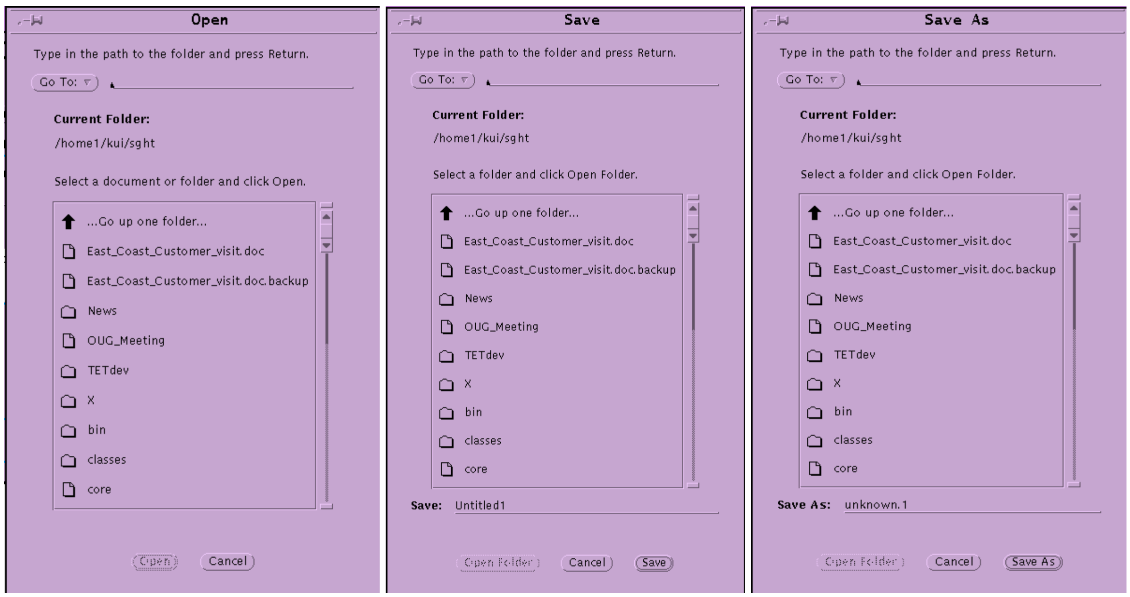

Figure 4-14 FileChooser Widget

For more detail on the programmer's interface for the `FileChooser` widget, please consult the man page for the `FileChooser` widget in Appendix C "OpenWindows Source Code" in this book.

The following is a sample source file using the `FileChooser` widget:

Code Example 4-7 FileChooser Widget Sample Source

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/ControlAre.h>
#include <Xol/OblongButt.h>
#include <Xol/FileChSh.h>
```

Code Example 4-7   FileChooser Widget Sample Source (Continued)

```
#include <Xol/MenuButton.h>

main(int argc, char **argv)
{
 XtAppContext app;
 Widgettoplevel, ca, menupane, ofilesh, sfilesh, safilesh, dumb;
 void   openfile_cb(Widget, XtPointer, XtPointer);
 void   filechpopup_cb(Widget, XtPointer, XtPointer);

 (void) OlToolkitInitialize((XtPointer)NULL);
 toplevel = XtAppInitialize(&app, "FileChooser",
        (XrmOptionDescList)NULL, 0,
        (Cardinal *)&argc, argv,
        (const char **)NULL,
        (ArgList)NULL,
        0);

 ca = XtVaCreateManagedWidget("controlarea", controlAreaWidgetClass,
        toplevel,
        NULL);

 ofilesh = XtVaCreatePopupShell("openfilechoosershell",
        fileChooserShellWidgetClass,ca,
        XtNoperation,OL_OPEN,
        NULL);

 XtVaGetValues(ofilesh,
        XtNfileChooserWidget,(XtArgVal)&dumb,
        NULL);

 XtAddCallback(dumb, XtNopenDocumentCallback, openfile_cb, NULL);

 sfilesh = XtVaCreatePopupShell("savefilechoosershell",
        fileChooserShellWidgetClass,ca,
        XtNoperation,OL_SAVE,
        NULL);

 safilesh = XtVaCreatePopupShell("saveasfilechoosershell",
        fileChooserShellWidgetClass,ca,
        XtNoperation,OL_SAVE_AS,
        NULL);

 dumb = XtVaCreateManagedWidget("menubutton",
        menuButtonWidgetClass,ca,
        XtNpushpin,(XtArgVal)OL_OUT,
```

Code Example 4-7 FileChooser Widget Sample Source (Continued)

```
        NULL);

 XtVaGetValues(dumb,
        XtNmenuPane,(XtArgVal)&menupane,
        NULL);

 dumb = XtVaCreateManagedWidget("Open...",
        oblongButtonWidgetClass,menupane,
        NULL);

 XtAddCallback(dumb, XtNselect, filechpopup_cb, ofilesh);

 dumb = XtVaCreateManagedWidget("Save...",
     oblongButtonWidgetClass, menupane,
     NULL);

 XtAddCallback(dumb, XtNselect, filechpopup_cb, sfilesh);

 dumb = XtVaCreateManagedWidget("SaveAs...",
     oblongButtonWidgetClass, menupane,
     NULL);

 XtAddCallback(dumb, XtNselect, filechpopup_cb, safilesh);

 XtRealizeWidget(toplevel);
 XtAppMainLoop(app);
}

void
openfile_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
 OlAnyCallbackStruct *any_cd = (OlAnyCallbackStruct *)call_data;

 if (any_cd->reason == OL_REASON_OPEN_DOCUMENT)
 {
 OlFileChDocumentCallbackStruct *cd =
           (OlFileChDocumentCallbackStruct *)call_data;
 }
}

void
filechpopup_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
 XtPopup((Widget)client_data, XtGrabNone);
}
```

FontChooser Widget

The `FontChooser` widget is a new widget introduced in OLIT 3.2. This widget allows you to communicate with an application regarding a font choice. Because the widget is inside the toolkit, all OLIT based applications will be able to popup a font chooser that is OPEN LOOK compliant and presents a consistent and familiar GUI. This section discusses the widget and is followed by a sample program.

Figure 4-15 contains a representation of the `FontChooser` widget:

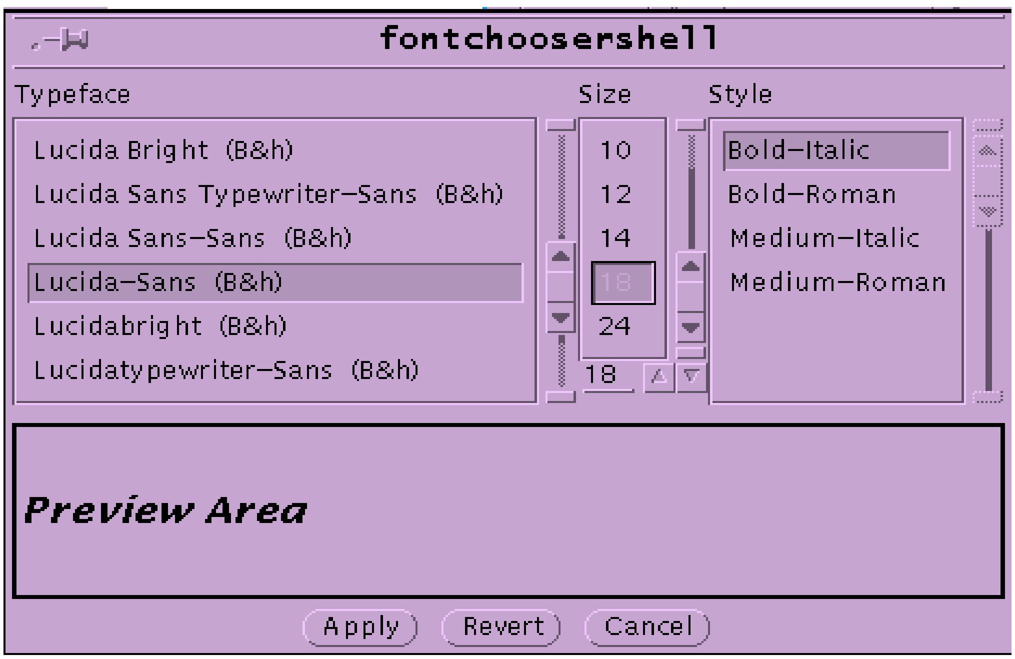

Figure 4-15 FontChooser Widget

For more detail on the programmer's interface for the `FontChooser` widget, please consult the man page for the `FontChooser` widget in Appendix C "OpenWindows Source Code", in this book.

The following is a sample source file using the `FontChooser` widget.

Code Example 4-8 FontChooser Widget Sample Source

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/ControlAre.h>
#include <Xol/OblongButt.h>
#include <Xol/FontChSh.h>

main(int argc, char **argv)
{
 XtAppContext app;
 Widget toplevel, ca, obbutt, tmpwidget, fontchshell;
 void fontchooser_cb(Widget, XtPointer, XtPointer);
 void fontapply_cb(Widget, XtPointer, XtPointer);

 (void) OlToolkitInitialize((XtPointer)NULL);
 toplevel = XtAppInitialize(&app, "FileChooser",
        (XrmOptionDescList)NULL, 0,
        (Cardinal *)&argc, argv,
        (const char **)NULL,
        (ArgList)NULL,
        0);

 ca = XtVaCreateManagedWidget("controlarea", controlAreaWidgetClass,
        toplevel,
        NULL);

 obbutt = XtVaCreateManagedWidget("fontbutton", oblongButtonWidgetClass,
        ca,
        NULL);

 fontchshell = XtVaCreatePopupShell("fontchoosershell",
        fontChooserShellWidgetClass,
        obbutt,
        NULL);

 XtAddCallback(obbutt, XtNselect, fontchooser_cb, (XtPointer)fontchshell);

 XtVaGetValues(fontchshell,
        XtNfontChooserWidget,(XtArgVal)&tmpwidget,
        NULL);

 XtAddCallback(tmpwidget, XtNapplyCallback, fontapply_cb, NULL);
```

Code Example 4-8 FontChooser Widget Sample Source (Continued)

```
  XtRealizeWidget(toplevel);
  XtAppMainLoop(app);
 }

void
fontchooser_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
    XtPopup((Widget)client_data, XtGrabNone);
}

void
fontapply_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
    OlFCApplyCallbackStruct *cdata = (OlFCApplyCallbackStruct *)call_data;
    printf("the font chosen is:\n\t%s\n", cdata->current_font_name);
}
```

SuperCaret Support

Developers who have used OLIT in the past know that when the application written in OLIT gets focus, the widget with focus is colored with either RED or the input focus color defined in the resource file. The input focus is used to tell which widget has the focus in a mouseless model where keyboard navigation is possible. The following section discusses the input focus model implemented in OLIT 3.0, 3.0.1, and 3.1 and then discusses the new input focus model based on supercaret.

Figure 4-16 contains a representation of the `SuperCaret` widget:

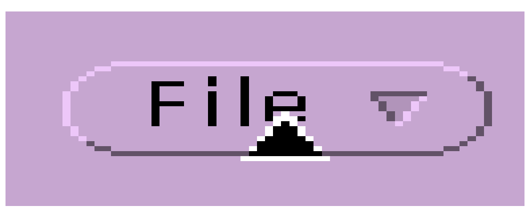

Figure 4-16 SuperCaret Widget

Existing Input Focus Model

The input focus model is available when mouseless mode is on. This is done by:

- setting `XtNmouseless` to `true` on the top-level window of an application.
- setting `*mouseless: true` in resource file.

Note – In OLIT 3.0, `XtNtraversalOn/*traversalOn` did the same thing.

When the input focus is active, the application shows which widget has the focus by using focus color as the foreground for the widget. The application and the end user have the flexibility to change the focus color by setting `XtNinputFocusColor/*inputFocusColor`.

New Focus Mode

With the new input focus model introduced in OLIT 3.2, the input focus feedback will be provided by a *supercaret* in the form of a triangular caret overlaying the GUI component that has keyboard focus. The caret appears to move between objects as OPEN LOOK mouse and mouseless operations cause the focus object to change.

To enable the this feature, the following application resources are available:

Note – Mouseless mode must be on for this to work.

Set `XtNinputFocusFeedback` to:

- `OL_INPUT_FOCUS_COLOR` for the old model with input focus color.
- `OL_SUPERCARET` for the new model with super caret.

The resource, if set by the application, should be done on the top-level window. This resource should be set as an argument to `XtAppInitialize()` or the equivalent of this function.

You can set this resource in your resource file using the line:

```
*inputFocusFeedback: supercaret/inputfocuscolor
```

The default value is for `inputfocuscolor`, although this may change in a future release. If the resource is set to `inputfocuscolor`, the `XtNinputFocusColor` resource is used to determine the input focus color.

`SuperCaret` example:

```
*inputFocusFeedback:  supercaret         <- default
*inputFocusFeedback:  inputfocuscolor    <- old style
```

OpenWindows Server (X11 Release 5 / DPS based)

OpenWindows Version 3.2, as well as earlier releases, contains the X11 Release 4 compliant X11/NeWS server. The next major release, OpenWindows Version 4.x, will include the X11 Release 5 compliant server released by MIT with Display PostScript System (DPS) from Adobe as an extension to the server.

Note – In this section, “OpenWindows Version 4,” “X11 Release 5-compliant server,” and ”Next Generation server” can be used interchangeably and will refer to the redesigned server based on MIT X11 Release 5 and DPS.

This section discusses porting considerations for application and hardware developers. The section focuses on new features of X11 Release 5 from MIT, as well as SunSoft’s value-added features to the server. The performance goals for this server with all the value-added work from SunSoft are similar to the Sun Sampler Server from MIT.

Note – With the OpenWindows Version 4 server, all support of SunView and NeWS will be discontinued.

Changes to OpenWindows Version 4 Server

OpenWindows Version 4 has many new features which are discussed in in this section. These include:

- a porting interface based on DDX (with SunSoft’s value add of loadable DDX, Multiple Plane Groups (MPG) and Multiple Hardware Color maps (MHC) support)
- input extensions for devices (with SunSoft’s value add of allowing dynamically loadable extensions with no server recompile)
- font support for scalable/outline and bitmap fonts (F3 forman, Type 1, Type 3, and Bitmaps)
- new server options
- customization for end users with disabilities
- Display PostScript Extension with support for the DPS toolkit and resources
- internationalization
- device-independent color

The Porting Interface

The OpenWindows Version 4 server uses the DDX interface from MIT as the interface for adding devices to the server. This section assumes that you are is already familiar with the MIT server and its porting interface (DDX). If not, please see the section, “Additional Information”.

SunSoft has added the ability to dynamically load DDX modules at runtime. This will enable third-party developers to develop DDX modules that can be delivered as separate components. SunSoft also provides DDX utility libraries that supply functionality which is common across devices. This functionality will facilitate the task of porting the OpenWindows Version 4 server to new graphics devices.

The DDX interface is quite extensive; the screen structure alone contains approximately 70 function vectors. Fortunately, many of these functions do not need to be specialized for every device. There are general purpose DDX modules that provide functionality that can be used by a device's DDX implementation; these are shown in Figure 4-17.

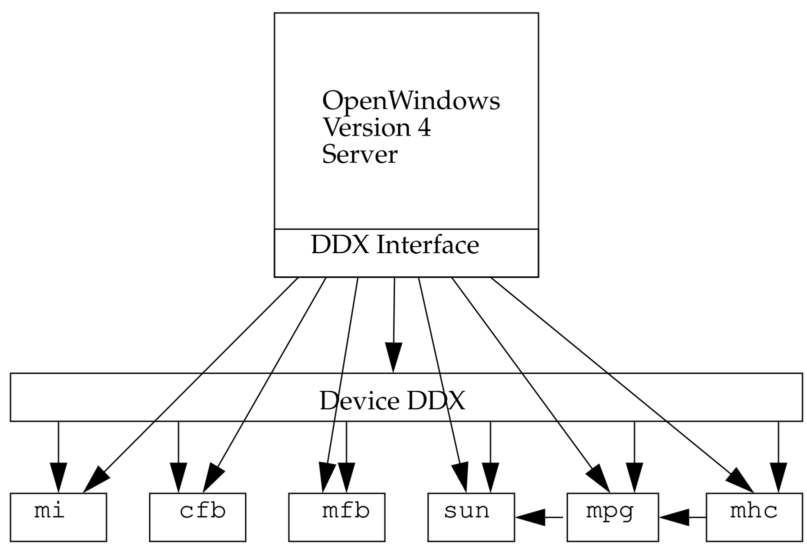

Figure 4-17 DDX module interdependencies

These modules provide different types of functionality. The `mi` module supplies high level, machine independent, functionality; `cfb` and `mfb` supply functionality useful to memory-mapped color and monochrome frame buffers. The `mpg` and `mhc` modules are for devices with multiple plane groups and multiple hardware color maps, and the `sun` module is for frame buffers that have device drivers that respond to specific ioctls.

Note – Although the DDX interface will be exposed to Independent Hardware Vendors (IHVs), it should be clear that it is an uncommitted interface, subject to change in subsequent releases of OpenWindows. There is also a DDX interface for input devices which is not covered in this book.

Loadable DDX

Loadable DDX is a mechanism that allows the server to dynamically load DDX modules at runtime instead of having to re-link the server to add support for new devices. The DDX module is essentially a shared library that is linked into the server at runtime through an explicit call to `dlopen()`. The location of the DDX modules and their mappings between devices is determined by a configuration file.

Dumb Frame Buffer Support

As with the MIT server, OpenWindows Version 4 provides a set of general purpose support routines for dumb frame buffers. This includes `mfb` for monochrome frame buffers and `cfb` for color frame buffers.

Although `cfb` supports depths of 2-, 4-, 8-, 16-, and 32-bits, only the 8-bit and 32-bit depths are currently built and tested.

Sun Device Support

Many frame buffers on the Sun platform have common characteristics. For example, they are accessed through device drivers and these device drivers support a common set of ioctls. The `sun` DDX module provides functionality that takes advantage of these characteristics.

Multiple Plane Group Support

For devices with multiple plane groups there is a DDX module that provides most of the management functions necessary for such devices. The `mpg` support also includes the ability to minimize exposure events between windows that reside in different plane groups. The `mpg` interface is designed so that other layers, such as `cfb` and `mfb`, may be used beneath it to do the actual rendering and window manipulation.

Additional Information

SunSoft will ship an OpenWindows Version 4 Server Device Developer's Kit for IHVs interested in porting their devices to the Solaris platform. The kit contains:

- documentation describing the interfaces
- necessary header files and libraries

- sample implementations (reference ports)
- sufficient server source code for debugging

There are also documents that describe the X11 window server and its porting interface. All of these come on the distribution tape from MIT except the one from Digital Press, which is highly recommended for those new to X11 Server development:

- *Strategies for Porting the X v11 Sample Server*, Susan Angebranndt, Raymond Drewry, Phillip Karlton, Todd Newman, Bon Scheifler, and Keith Packard, MIT X Consortium.
- *Definition of the Porting Layer for the X v11 Sample Server,* Susan Angebranndt, Raymond Drewry, Phillip Karlton, Todd Newman, Bon Scheifler, and Keith Packard, MIT X Consortium.
- *X Server Internals: A Tutorial on the MIT Sample Server at Release 4*, Keith Packard, MIT X Consortium.
- *X Window System, Version 11, Release 5 Release Notes*, MIT X Consortium Staff, MIT Laboratory for Computer Science.
- *The X Window System Server*, Elias Israel and Erik Fortune, Digital Press.

The Loadable DDX Interface

Note – The interfaces described below are subject to change. Please consult the OpenWindows Version 4 Server Device Developer's Kit that accompanies the OpenWindows Version 4 product.

The server interfaces to a loadable DDX routine in the following manner:

1. The module containing the device's DDX module is installed in the configuration directory. This directory is specified to be in the `OW_CONFIG_PATH` which defaults to `/usr/openwin/etc/server`.
2. The device type, device name, device class, the filename, and the name of the initialization function are added to the `ow_config` file (also found in the `OW_CONFIG_PATH`). The frame buffer section looks similar to the following:

# fld	DevName	Class	libName	InitFunc
f	/dev/cgsix0	12	libddxcg6.so.1.0	sunCG6Init
f	/dev/cgthree0	6	libddxcg3.so.1.0	sunCG3Init
f	/dev/cgfour0	8	libddxcg4.so.1.0	sunCG4Init
f	/dev/bwtwo0	2	libddxbw2.so.1.0	sunBW2Init
f	/dev/cgeight0	7	libddxcg8.so.1.0	sunCG8init
f	/dev/fb	0	libddxfb.so.1.0	sunFBInit
f	/dev/gt0	18	libddxgt.so.1.0	sunGTInit

The first field, `# fld`, is the type of device, and it should contain an `f` for all frame buffers or other output devices (other types of modules are described in this file: `i` denotes input handlers and `e` denotes an extension).

The second field, `DevName`, is the filename that corresponds to the device.

The third field, `Class`, is the value returned when the `FBIOGTYPE` ioctl is issued to the device driver. It is only used when the server is started without specifying a device or when the servier is started on `/dev/fb`.

The fourth field, `libName`, is the name of the file that contains the shared library with the DDX module in it. A loadable module may be linked against other loadable modules.

The fifth field, `InitFunc`, is the symbol name for the entry point into the DDX module's initialization function.

3. From this information, the server, at initialization time, loads the appropriate module and calls its `InitFunc`. It is up to the `InitFunc` to initialize the device and data structures so that the server can run.

The Initialization Function

Each type of device has an initialization function that is called once for each device. The initialization function is responsible for initializing the device and the screen data structure associated with it.

```
xxxInit(index, pScreen, argc, argv)
    int        index;      /* The index of pScreen in the ScreenInfo */
    ScreenPtr   pScreen;   /* The Screen to initialize */
    int         argc;     /* The number of the Server's arguments. */
    char       **argv;    /* The arguments themselves. Don't change!*/
```

Note – A mechanism for setting version numbers is being developed that will allow the server and the DDX modules to detect incompatibilities. This may require another entry point or may be a part of the screen initialization function. A proposal for this system is included at the back of this document.

The Sun Interface

The device initialization procedure mentioned is responsible for initializing the screen structure and all of its function vectors. Since many sun devices have similar characteristics, there is a set of utility functions that make the porting job easier and reduces the size of the server. The Sun interface provides services to do such things as opening the device, checking its type, updating the color maps, and managing the cursor.

Note – The interfaces described in this section are subject to change. Please consult the OpenWindows Version 4 Server Device Developer's Kit that accompanies the OpenWindows Version 4 product.

- sunOpenFrameBuffer

```
int
sunOpenFrameBuffer(expect, pfbType, index, pSunFbs, argc, argv)
    int expect;               /* The expected type of frame buffer */
    struct fbtype *pfbType;   /* Place to store the fb info */
    int index;                /* Screen index */
    fbFd *pSunFbs;            /* Pointer to private fbFd */
    int argc;                 /* Command-line arguments... */
    char **argv;              /* ... */
```

sunOpenFramebuffer() opens the frame buffer, queries the type of frame buffer using the FBIOGATTR and FBIOGTYPE ioctls, verifies the device is what was expected, and returns the file descriptor of the device. If an error occurs, -1 is returned. The structures pointed to by pfbType and pSunFbs are filled in.

- sunScreenAllocate

```
Bool
sunScreenAllocate(ScreenPtr pscreen)
```

SunScreenAllocate() allocates the sun screen private index if necessary and allocates the sunScreenPrivate for the given screen and sets the devPrivates entry to point to it. FALSE is returned if there is an error.

- sunScreenInit

```
Bool
sunScreenInit(ScreenPtr pScreen)
```

sunScreenInit initializes the screen's cursor functions and SaveScreen functions to Sun Specific routines. sunScreenInit returns TRUE upon success.

- `sunCGScreenInit`

```
void
sunCGScreenInit(ScreenPtr pScreen)
```

`sunCGScreenInit` is called if the device is sun Color Frame buffer. It initializes the color map functions (`InstallColormap`, `UninstallColormap`, `ListInstalledColormaps`, `StoreColors`, and `UpdateColormap`) in `pScreen` to use the Sun-specific functions. The underlying device must support the `FBIOPUTCMAP` ioctl for these functions to be used.

- `SunSetPixmapFormat`

```
Bool
sunSetPixmapFormat(PixmapFormatRec request)
```

`SunSetPixmapFormat` is used by each device to advertise what pixmap formats it supports. Since there may be multiple screens supporting different depths it is necessary to have a centralized facility for registering the formats. There can only be one format per depth, so any attempt to define a new format for an existing depth will fail and sunSetPixmapFormat will return `FALSE`.

Currently the following formats are known to be supported by some Sun Devices:

Depth	BitsPerPixel	ScanlinePad
1	1	BITMAP_SCANLINE_PAD
4	8	BITMAP_SCANLINE_PAD
8	8	BITMAP_SCANLINE_PAD
24	32	BITMAP_SCANLINE_PAD
32	32	BITMAP_SCANLINE_PAD

SunSaveScreen

`SunSaveScreen` is the `saveScreen` function for most Sun devices and is installed automatically by `sunScreenInit`. A devices's `ddx` module may need to call this function directly to ensure that video is enabled at start-up and before exiting using the format:

```
sunScreenSave(ScreenPtr pScreen, int on)
```

where the value of `on` has the following meanings:

SCREEN_SAVER_ON	Turn on the screen saver. Blanks the screen.
SCREEN_SAVER_OFF	Turn off the screensave, Enables video.
SCREENSAVER_FORCER	Update the time of the last screen saver mode change.

SunDisableCursor

Sometimes, usually before exiting server when the screen's `closeScreen` function is being called, it is necessary for the `ddx` module to disable the `sunCursor`. Usually this is done to handle the case where the cursor is a hardware cursor.

```
sunDisableCursor(ScreenPtr pScreen)
```

SunGetMonitorRes

The `ddx` module calls this routine to get the monitor's resolution. The monitor's resolution is specified on the command line; this routine fills in what the end user specified. If the end user does not specify a monitor resolution, 90 DPI is the default.

```
void sunGetMonitorRes(int index, int *dpix int *dpiy)
```

The integer `index` is the screen's index, and `dpix` and `dpiy` are filled in by this routine.

Note – Future releases of the OpenWindows server may store the monitor resolution in some sort of configuration file associated with the screen.

SunGetVisualInfo

This routine is called to get the specified visual information for the screen. Since the user may specify the default visual class, the default depth, or grayvisual, the `ddx` module must know how to set up its visuals.

```
void
sunGetVisualInfo(int   index,
                 int   *defClass,
                 int   *devDepth,
                 Bool  *grayVis)
```

The integer `index` is the screen's index, `defClass` is the visual class that should be used as the default, and `defDepth` is the default depth. This routine will assign `-1` to either `defClass` or `defDepth` if there was no specification for these on the command line. `GrayVis` is set to indicate the `-grayvis` command line option which should disable all color visuals, as the device may be connected to a gray-scale monitor.

The MPG Interface

Some devices contain multiple plane groups (MPG) to support overlays and visuals of varying depths. The `mpg` DDX module provides the following features:

- *Window Hierarchy Operations*
  MPG provides the necessary screen functions to move windows, and handle exposure events. Typically when a window is moved, all of its plane groups need to be moved. Likewise, when a window is exposed, all of its plane groups need to be repainted.
- *Minimizing Exposure Events*
  MPG provides specialized window hierarchy functions that minimize exposure event between windows that reside in separate plane groups.
- *Leveraging off of existing DDX interfaces*
  MPG is designed to layer on top of existing rendering and window functionality such as `cfb` or `mfb`.

Architectural Overview

The `mpg` module is basically a data driven design. It is up to the device's DDX module to inform the `mpg` DDX module what plane groups are being used by what windows and how it is using them. From this information, the `mpg` windowing functions take care of moving, preparing and calculating exposure to the plane groups.

The `mpg` layer does not actually do any rendering. Instead, it is designed to layer on top of other DDX modules, like `cfb` and `mfb`, which provide rendering and some basic windowing functions. This way, a frame buffer with a 24-bit plane group and a 1 bit overlay can use `cfb32` and `mfb` to do the rendering and most of the window functions. The `mpg` module manages the switching between modules and provides the rest of the windowing functions.

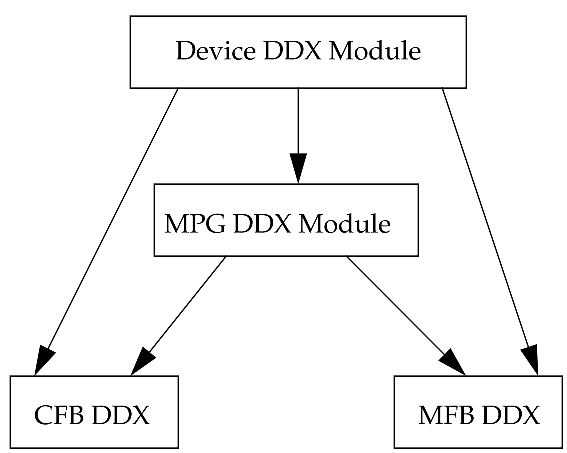

Figure 4-18 MPG DDX Interface Diagram

The `mpg` layer does not explicitly call `cfb` or `mfb`, but can use any device specific screen functions. These are shown as examples of how `mpg` may be configured.

There is also a distinction between window types and plane groups. The example device may have two types of windows—monochrome and color—but may have 3 plane groups: color, overlay and overlay enable. There must be a screen drawable for each plane group and there be certain screen functions for each type of window.

Note – The Functional Interface for MPG can be found in the OpenWindows Version 4 Server Device Developer's Kit that accompanies the OpenWindows Version 4 product.

The Multiple Hardware Color Maps (MHC) Interface

The following terminology is used when discussing multiple hardware color maps:

A *color look-up table* is a hardware look-up table (LUT) that outputs an RGB color triple when a pixel value is input. R, G, and B are the voltage levels to drive the red, green, and blue monitor guns respectively. A LUT is also called a *hardware color map*.

A *software color map* is a software abstraction of the color mapping process that a color LUT provides. Clients can allocate pixels from a software color map by presenting an RGB triple to the server. The server hands back to the client a pixel index where the triple has been stored in the color map. The software color map can be loaded or installed into a hardware color LUT. Software color maps are supported through the X11 protocol as a *color map resource*. A software color map is also called a *color map*.

Note – In the following discussion the word *color map* used alone will refer to a software color map. The acronym *LUT* used alone will refer to a color LUT, or hardware color map.

Color Map Installation–Single LUT Devices

Most devices have only one LUT. Only one color map may be installed in this LUT, where "installed" means the LUT is guaranteed to contain the values of all of the allocated entries in the color map.

In an environment in which there are several clients each having their own color map, only one of these client color maps will be installed at a given time. The windows that are associated with the installed color map will show their correct colors. Windows that are associated with some other color map may show false colors, depending on whether their entries collide with the installed color map. This display of false colors is referred to as *color map flashing* or *technicolor*.

Color Map Loading

The server maximizes the number of valid colors in the hardware color LUT. The server loads all of the allocated entries of the newly installed color map into the LUT. Then, for those that are unallocated, the server looks for corresponding allocated entries in other color maps to load. Slots in the default color map are examined first, followed by slots in the StaticColor color map (only if the OpenWindows option `-favorstatic` is selected). For a given entry, if no allocated entry is found in either of these color maps, the LUT value will not be changed; it will still have the value of the last allocated color map entry that was installed in that LUT.

Color Map Installation–Multiple Look-up Table Devices

The GS and GT frame buffers provided by Sun Microsystems Computer Corporation provide multiple 8-bit indexed look-up tables (LUTs). The purpose of multiple LUTs is to increase the frequency of true colors being displayed and to reduce the occurrence of color map flashing.

The GS has two 8-bit LUTs. The GT has a maximum of eight 8-bit LUTs. These devices also provide separate LUTs for the other supported visual depths. The GS provides a single 4-bit LUT and a single 24-bit LUT. The GT provides a single 24-bit LUT. Because there is only one LUT for the other depths, the installation of color maps for visuals of these depths behaves the same way as for a single LUT device. Therefore, the remainder of this discussion will deal only with 8-bit LUTs.

When there are *n* color maps, each of them may be installed in hardware when there are *n* LUTs. However, when there are fewer than *n* LUTs, only a subset of the color maps can be installed. The decision making process that the server uses to determine which color maps to assign to which LUTs is called the *server color map installation policy*.

Note – Ultimately, the window manager decides which color maps are installed in which LUTs. This is referred to as the *window manager color map installation policy*. This policy is different from the server color map installation policy, which only specifies how the server reacts to various `InstallColormap` and `UninstallColormap` requests. The window manager color map installation policy defines how and when a window manager will send these requests to the server. The window manager policy is outside of the domain of this chapter and will not be mentioned further.

The color map installation policy used by the window server for visuals that provide *n* LUTs is to keep the *n* most recently installed color maps resident in some LUT. “Most recently installed” means that the color maps were installed by the last *n* `InstallColormap` requests sent by the window manager.

For example, if there are two LUTs (L1 and L2) that have two color maps installed (C1 is installed in L1 and C2 is installed in L2), and the server receives an `InstallColormap` request for a third color map (C3), the least recently installed color map (C2) will be *bumped* from L2, and C3 will be installed in L2. If another `InstallColormap` request is received for a fourth color map (C4), C1 will be bumped from L1 (because it is now the least recently installed) and C4 will be installed into L1. Another term used for this bumping is *implicit uninstallation* (this is the term used in the X11 protocol specification).

When a color map is explicitly uninstalled by an `UninstallColormap` or `FreeColormap` request, the server takes advantage of the available LUT and assigns it the most recently installed color map that has been bumped. This is referred to as *unbumping* or *implicit installation*. In the preceding example, if C4 were freed, C1 would be assigned C4's LUT (L1) because C1 is a bumped color map which still wants a LUT if it can get it, and it was requested by the window manager to be installed before the request to install C2.

Note – The Functional Interface for MHC can be found in the OpenWindows Version 4 Server Device Developer's Kit that accompanies the OpenWindows Version 4 product. To see sample source code, see the section "Sample CG3 Source with Sun Interface" in Appendix C titled "OpenWindows Source Code."

Input Extensions for Devices

This section describes the design of how input devices are handled in the OpenWindows Version 4 server. This section discusses how the Consortium's implementation of the Input Extensions has changed and how those changes result in a more flexible implementation, and how to apply the concepts of these concepts to the internals of the server. This section also discusses the interface between the owmux STREAMS module and the server and device libraries.

Client programs use the Input Extension to gain access to non-core devices in the server. The Input Extension is an extension that was adopted as a standard and is a part of the X11 Release 5 distribution. As a part of the distribution there are three documents that are of interest; the X11 Input Extension Library Specification, the X11 Input Extension Protocol Specification, and the X11 Input Extension Porting Document. These documents can be found in the subdirectory `mit/hardcopy/extensions/xinput`. This document is not intended to be a replacement for any of these documents; it intends to present the specifics of how input devices may be dynamically loaded into the server, as well as how this is integrated with the Input Extension.

Current window system servers provide very limited support of input devices. These servers are not dynamic or extensible. The X11/NeWS server is limited to supporting the mouse, the keyboard, SunDials, SunButtons, and the lightpen. It also has the ability to support certain devices that can present themselves as one of the above. The Sun sample server that is a part of the X11 Release 5 distribution is even more limited. It only supports the mouse and keyboard.

With the adoption of the Input Extension as a standard in X11 Release 5, there now exists a protocol specification in X that handles multiple input devices attached to the server and allows multiple clients to receive events from that device. The problem with this extension is that the server must have the code to control each device built into it when the server is created. Once a server is shipped, there is no way to add support for a new device that the server does not know about. The following design for handling input devices in the server solves this problem. It presents a design for dynamically loading input devices, and accessing them through the Input Extension.

Note – Dynamic loading is true for extensions to the server in general.

The client interface for accessing input devices in the server is the Input Extension as defined in X11 Release 5. The design presented here will not change that interface in any way. Any client protocol requests that are used in this document are already defined in the Input Extension.

Overview of Server Configuration and Start-Up

The server reads a configuration file to find out what devices are available and which devices are to be used as the core pointer and core keyboard. Every input device the server wishes to use is dynamically loaded into the server as it is needed. This is done by putting all of the code that is specific to an input device into a shared library. The server may then load this library when it needs to and use the functions in the shared library to control the device.

When the server starts up, it will read the `ow_config` configuration file to find out which devices are to be used as the core keyboard and core pointer. If there are none, the server will exit because these two devices are required by the core X protocol. If the core keyboard and pointer are present, the server loads their shared libraries, initializes them, and starts them. At this point, if no client requests any other devices to be used, via the Input Extension, the server will continue to run just like the X11 Release 5 Sun sample server.

Overview of Client Access to Devices

A client is required by the Input Extension to execute the following steps when gaining access to devices:

- Get a list of the devices that the server supports by issuing a `ListInputDevices` request. The first `ListInputDevices` received by the server causes it to load and initialize all input device found in the `ow_config` file (except the keyboard and mouse which have already been loaded). If the `DeviceControlProc` returns a `!Success`, the server assumes the hardware is not present and unloads the device. All subsequent `ListInputDevices` will cause the server to search the `ow_config` file for any devices that may have been added since the last `ListInputDevices`. If any new devices are found, they are loaded and initialized. During initialization, the device registers all its features with the server. This information is passed back to the client as part of the `ListInputDevices` request. Note that the devices have not been turned on by a `ListInputDevices` request; therefore, the server will not accept input from them.

- Once a client has the list of devices, it may open one of the devices by issuing an `OpenDevice` request. The server tells the device to turn itself on, and when input is pending on that device, the server will read the data and put it into the event stream.
- At this point, the client may issue any of the Input Extension protocol requests to receive events, initiate grabs, and control features of the device. This portion of device handling in the server is exactly as it appears in the X11 Release 5 Input Extension. No changes to the Input Extension protocol have been made.
- When the client is finished with the device, it issues a `CloseDevice` request. If the client issuing the `CloseDevice` request was the last client to have the device open, the server tells the device to turn itself off. If there are other clients that have the device open, the server leaves the device open and continues to read the device until no clients have the device open. Although the server may still read the device, the client that issued the `CloseDevice` request no longer receives events from the device. This is because the event mask for that client has been cleared by the Input Extension as part of the `CloseDevice` processing.

Overview of Server Restart and Shutdown

During server restart and shutdown, all open devices are closed and unloaded. During the close process the device is notified of the shutdown and must free any memory that may have been allocated and close its file descriptor. There is no difference between a server restart and a shutdown.

Overview of the Input Multiplexor Device

The MIT X11 Release 5 server reads the mouse and keyboard file descriptors and then sorts the events from the two devices so that the events are queued in the server in the order they actually happen. The server introduces an input multiplexor that reduces the amount of processing required by the server to handle incoming device events.

The input multiplexor is a M-to-one or lower STREAMS multiplexor. It will multiplex events from multiple devices into a single stream for the server to read. The multiplexor reduces the number of file descriptors you have to read from and simplifies the sorting of events from multiple file descriptors. The multiplexor also removes the need for a read procedure from the device’s shared library.

You do not need the extension devices to use the input multiplexor. When a device is initialized, it must register itself as either having a read procedure or as wanting the input multiplexor to read the device. Only devices that are generating VUID events can use the multiplexor. All others must supply a read procedure to read the device events and format the data.

What follows is a detailed description of the server modifications needed to support dynamic loading of input devices in the OpenWindows Version 4 server and how it interacts with the Input Extension.

Server Configuration and Start-Up

The configuration file `ow_config` is used by the server to read in information about the dynamically loadable extensions available, the different dynamically loadable frame buffer modules, and the dynamically loadable input devices. The server looks for the `ow_config` file in the path as follows:

- `configpathA` comes from the `-cp` command line option
- `configpathB` comes from the `OW_CONFIG_PATH` environment variable
- `configpathC` comes from the `OPENWINHOME` environment variable
- `configpath` = `configpathA:configpathB:configpathC`

The format for the input devices portion of the `ow_config` file is as follows:

Parameter	Meaning
`fld`	`i` (indicates input device)
`DevName`	Name of the device that uniquely identifies the device.
`Class`	`K` indicates the core keyboard device.
	`P` indicates the core pointer device.
	`-` indicates other than core keyboard or pointer device.
`SharedObjectName`	Name of the shared object file.
`CompatFunctionName`	Function that is called to check compatibility between the DDX and the handler.

For example, the core keyboard and mouse would be specified as follows:

# fld	DevName	Class	SharedObjectName	CompatFunctionName
i	`IKBD`	`K`	`ddxSUNWkbd.so.1`	`ddxSUNWkbdHandlerCompatible`
i	`IMOUSE`	`P`	`ddxSUNWmouse.so.1`	`ddxSUNWmouseHandlerCompatible`

When the server is being started or restarted, the routine `InitInput()` gets called. It is the responsibility of this routine to open the muxtiplexor device, initialize the core keyboard and pointer, and register them with the rest of the server. To accomplish this in OpenWindows

Version 4, `InitInput()` calls a new function called `ParseDeviceFile()`. If this function is being called from `InitInput()`, it causes the `ow_config` file to be parsed while the server looks for the core keyboard and pointer. When it encounters one of these devices it calls the `dlopen()` function to open the devices shared object in `$(OPENWINHOME)/lib/X11`. Then the compatibility function specified in the configuration file is retrieved via `dlsym()` and calls this function. If the device handler is compatible with the server, the handler passes back a pointer to its `DeviceControlProc()` to the server. The device is then added and registered with the server via `AddInputDevice()` and `RegisterPointerDevice()` and `RegisterKeyboardDevice()`. After `ParseDeviceFile()` completes, `InitInput()` registers the core pointer and keyboard with the `mieq` layer of the server by calling `mieqInit()`. The second case that `ParseDeviceFile()` needs to handle is discussed in the "Client Access to Device" section.

The DeviceControlProc()

The `DeviceControlProc()` is one of the entries in the `ow_config` file. It is one of the parameters passed to the `AddInputDevice()` function. The purpose of this function is to allow the server to tell the device to perform some sort of action. This allows the server to control the device without having to know how to perform an action on a specific device. The control function can be called with one of four actions.

The first action is `DEVICE_INIT`. During the `DEVICE_INIT` action, the device must initialize any data structures, open up the device, and register all of the features of the device.

The second action that the `DeviceControlProc()` must support is `DEVICE_ON`. This tells the device that it needs to be activated. A device that wants the input multiplexor to read its device events links itself to the multiplexor at this time by using the appropriate ioctl. Devices that have a read procedure to handle device events must at least call `AddEnabledDevice()` and have the device start generating SIGPOLLs when input events are at the STREAM head.

The third action is `DEVICE_OFF`. This tells the device that it will be turned off for a while. A device that is using the input multiplexor must unlink from the multiplexor at this time. A device that has its own read procedure must at least call `RemoveEnabledDevice()`.

The last action the device must support is `DEVICE_CLOSE`. This tells the device that the server is about to exit so the device closes itself and cleans up any states. These four actions allow the server to control a loadable device.

Device Features and Reading the Device

The Input Extension provides a number of routines that allow a device to register features that a client program may use. In OpenWindows Version 4, a device needs to use these registry routines during the `DEVICE_INIT` action of its control function. During `DEVICE_INIT`, the device may use any of the registry functions that are defined in the *X11 Input Extension Porting*

Document. Because of the dynamic loading of input devices in the server, there are a number of new registry functions that will be available. These may be viewed in two categories. The first category contains functions that help define device features to the server. The second category deals with how the data is to be read from the device.

There are several routines that let a device specify whether it supports a feature. OpenWindows Version 4 has six additional routines that a device may use during `DEVICE_INIT` to register its preferences. The routines and their interfaces are briefly described:

- `RegisterModifierCheckProc()`
  This routine registers a function to be called when the validity of a keycode needs to be checked by the device. This can happen when a client wishes to change the Modifier Mapping of a device.
- `RegisterSetDeviceModeProc()`
  This routine registers a function to be called when a client wishes to set the mode of a device. The two modes that a device may have are Absolute and Relative positioning.
- `RegisterSetDeviceValuatorsProc()`
  This routine registers a function to be called when a client wishes to change the valuators of a device.
- `RegisterChangeDeviceControlProc()`
  This routine registers a function to be called when a client wishes to change the control of a device. The only device control that is currently supported is the `DEVICE_RESOLUTION`.
- `RegisterXKeyboardInterest()`
  This routine registers a device as being a possible core keyboard device. This is used in the `ChangeKeyboardDevice` protocol request.
- `RegisterXPointerInterest()`
  This routine registers a device as being a possible core pointer device. It also registers a function to be called when a client wishes to make this device the core pointer. This is used in the `ChangePointerDevice` protocol request.

The second category of registry functions that are provided deal with how the device wishes to be read:

- `RegisterMuxIo()`
  This routine registers a device as one that will be read by the multiplexor.
- `RegisterFdIo()`
  This routine registers a device as one that will read itself.

The general flow of how data is read from devices is as follows. Figure 4-19 can be used to help in following this flow. As presented earlier, there are two ways that a device may be read. The device can chose to let the multiplexor read its events, or it can read the events itself. In the case were the device is reading its own events, the device driver must generate the signal `SIGPOLL` when data is available to be read. This requirement is mainly in place to enforce consistent flow through the server.

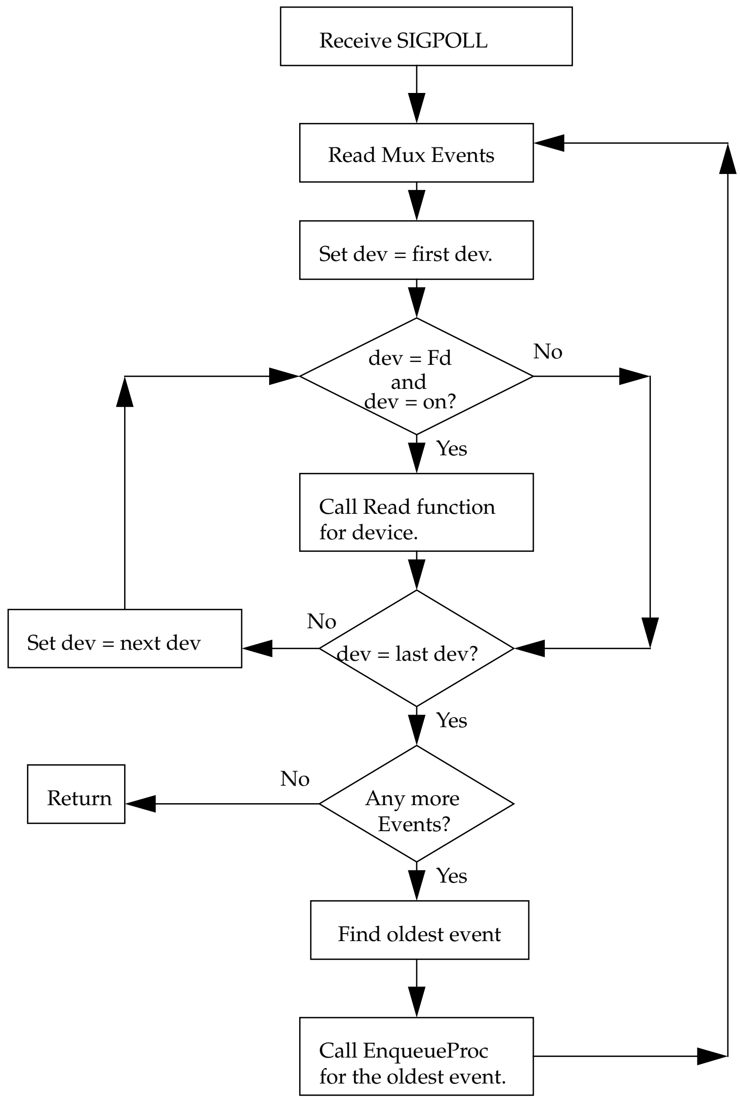

Figure 4-19 Data Flow from Devices

The basic flow is as follows. A `SIGPOLL` is received by the server. When this happens, the server loops through the following steps until no more events are available on any of the input devices.

1. Read all events from the Input Multiplexor.
2. For each device that is on and wishes to read itself, call the `readProc()` function for that device.
3. Check to see if there are any events from all of the sources just read,
   a. If there are no more events, break out of the loop and return.
   b. If there are more events, continue to 4.
4. Find the oldest event.
5. Give the oldest event to the `enqueueProc()` for that device.
6. Loop back up to 1.

The `enqueueProc()` in 5. is present for every device that is being read. The `enqueueProc()` takes an event, does any device specific processing on the event, converts it to an xEvent, and finally places it on the global event queue via `mieqEnqueue()`. Also, it may seem that it is wasteful to attempt to reread devices after every call to `enqueueProc()`, but this is the only way to ensure that events are delivered in the correct order.

Note – This design was taken straight from the X11 Release 5 Sun sample server.

There are two routines that allow a device to specify how the device wishes to be read. Both of the routines take a `DevicePtr`, a file descriptor, and an `enqueueProc()` as arguments. If the device wishes to read itself, it will use `RegisterFdIo()` and supply a `readProc()` as well as the three other arguments. A device that generates `VUID` style events and wishes to be read by the input multiplexor supplies the three arguments to `RegisterMuxIo()`.

That takes care of all of the feature registration portions of the OpenWindows Version 4 Input Extension. This is the most complicated part of handling input devices in the server. It is very important that the interfaces presented are well-defined so dynamic loading of input devices works without any side affects.

Client Access to Devices

At the end of server initialization, the core keyboard and core pointer are the only devices that are initialized and generating events. If a client wishes to use an Input Extension device, the first thing that it needs to do is issue a `ListInputDevices` protocol request. When the server receives this request, it must gather all the features of all the possible devices and pass this information back to the client. At this point the server calls `ParseDeviceFile()`.

The first time `ParseDeviceFile()` is called due to a `ListInputDevices` request, the `ow_config` file is reparsed. Any new devices that have not been initialized will have it's shared library loaded, call `AddInputDevice()` and `RegisterOtherDevice()`, and the control function of the device will be called with `DEVICE_INIT`. This will cause the device to register all of its features with the server, so that the server may send the information back to the client. At this point the device is known by the server, but is not generating any events. This is accomplished through the `OpenDevice` request.

For each additional `ListInputDevices` request `ParseDeviceFile()` will stat the file to see if the user has modified the file since the last `ListInputDevices` request. If the file has not changed, no action is taken. If the file has changed the file is reparsed. New devices are initialized as above. Devices that are no longer in the `ow_config` file are closed.

Each time the `ow_config` file is read due to a `ListInputDevices`, devices that have a class of `K` or `P` and were not the original core pointer and keyboard are treated as extension devices.

When a client wishes to use a device that is in the list of devices returned by the `ListInputDevices` request, it must use the `OpenDevice` request. When the server receives this request, it will tell the device to start generating events by calling the `DeviceControlProc()` with a value of `DEVICE_ON`. If the device is already on, this request adds the client to a list of clients that currently have the device open.

When a client is finished with a device, it uses the `CloseDevice` request to have the server close the device. When the server receives this request, it will clean up all of the references that this client had on this device, such as grabs and event masks. It will also remove the client from the list of clients that currently have the device open. If the client is the only client with the device open, the server will call the `DeviceControlProc()` with a value of `DEVICE_OFF`. This tells the device to stop generating events.

Server Shutdown

When the server is about to exit or restart, it goes through all of the devices that have been initialized and closes them. This is done via a call in the server called `CloseDevice()`. In `CloseDevice()`, the server calls the `DeviceControlProc()` with a value of `DEVICE_CLOSE`. This tells the device to clean up all of its resources because the server is about to exit. In addition to calling the `DeviceControlProc()` in `CloseDevice()`, the server also goes through its data structures and clean up any references to the device. This includes calling `dlclose()` on the shared library handle that is associated with the device. This ensures that if the server is just restarting, the server restarts successfully.

Note – The interfaces for the Device Library Interface Specification will be included in the OpenWindows Version 4 Server Device Developer's Kit that accompanies the OpenWindows Version 4 product.

The owmux STREAMS Module Interface

The owmux STREAMS module is an M-to-one or lower STREAMS multiplexor. The server will uses the owmux to merge events from multiple input devices below it and present a single stream of events for the server to process. The owmux onlys work with devices that generate firm_events as specified by the protocol of a VUID.

The owmux understands the `I_LINK`, `I_UNLINK`, `MUXIOCGETLOWER`, and `MUXIOCSETLOWER` ioctls as defined in `/usr/include/sys/muxio.h` and `/usr/include/sys/stropts.h`. All other ioctls are passed downstream to the device that has been specified by the `MUXIOSETLOWER` ioctl. Table 4-3 describes the ioctls the owmux understands.

Table 4-3 The owmux ioctls

ioctl	Function
`MUXIOCSETLOWER`	Set the device to send all unrecognized ioctls too. The argument is a pointer to an `int` containing the index of the device that is to receive all further unrecognized ioctls.
`MUXIOCGETLOWER`	Get the index of the device that is currently receiving all unrecognized ioctls. The argument is a pointer to an `int`. The index of the device currently receiving unrecognized ioctls is stored in the int pointed to by the argument.
`I_LINK`	Link the stream from the specified device under the multiplexor. The argument is a pointer to an `int`. The device whose file descriptor is pointed to by the argument is linked under the multiplexor. The `I_LINK` ioctl returns an index that is used as the argument of the three other ioctls to refer to the newly linked device.
`I_UNLINK`	Unlink the stream from the specified device under the multiplexor. The argument is a pointer to an `int`. The index of the device to be unlinked is stored in the `int` pointed to by the argument.

The owmux interface (private) in the server is responsible for opening the owmux and linking and unlinking devices in the following situations:

- the server opens the owmux in the `RegisterMuxIo()` function
- the server links a device's stream under the multiplexor in the `LinkDevice()` function

- the server unlinks a device's stream from the multiplexor in the `UnlinkDevice()` function

The only time the owmux interface (public) in the device library references the owmux is when it must send an `ioctl` to the device being controlled by the library.

Note – To see sample source code, see the section "Sample Tablet Device Driver Using the X Input Extension" in Appendix C titled "OpenWindows Source Code."

Font Service and Scalable Fonts

The X11/NeWS server has always offered scalable fonts. A Type 1 font scaler has been incorporated into Solaris 2.x as well. Therefore, you can use F3 format, Type 1, Type 3 (PostScript), and Bitmap fonts in any applications you develop.

Bitmap Fonts

Each size-specific bitmap font is provided in a separate file. The format of these bitmap font files can be any of several industry standard formats. All of the bitmap fonts shipped with OpenWindows "Next Generation server" will be in the Portable Compile Format which has a filename extension of `.pcf`. For smaller sizes, some scalable fonts are provided as hand-tuned bitmaps. The hand-tuned bitmaps provide superior results and are used, if available, rather than scaling the desired font from an outline representation.

Note – Some of the binary bitmap file formats cannot be shared between machines of different architectures, as shown in Table 4-4.

Table 4-4 Bitmap File Formats

File Format	File extension	Binary	Architecture-specific
Bitmap Distribution Format	`.bdf`	No	No
Portable Compiled Format	`.pcf`	Yes	No
Server Natural Font format	`.snf`	Yes	Yes
OpenWindows Font Bitmap	`.fb`	Yes	Yes

Outline Fonts

Each outline font is provided in one of four outline font formats:

- OpenFonts™ F3 format scalable outline fonts are stored in files with the `.f3b` extension

- PostScript® Type 3 fonts are stored in files with the `.ps` extension. Because the outlines for these fonts are described by arbitrary PostScript language procedures, these fonts are only visible to DPS applications.
- Adobe Type™ 1 font format files have the `.pfa` or `.pfb` extension.
- Bitstream Speedo™ font format files have the `.spd` extension.

This means a single outline font can be arbitrarily scaled to a variety of sizes, thus replacing a whole family of bitmap files. The "Next Generation" server supplies a core set of 57 F3 fonts, 8 Speedo fonts, and either 8 or 35 PostScript fonts from Adobe.

The outline fonts can be used to generate arbitrarily transformed and rotated characters from DPS. X11 programs use these fonts to get characters that are arbitrarily scaled, but are not rotated. When you use a character from an outline font for the first time, the bitmap must be created; then it can be cached and reused. If initial display-time for previously undisplayed characters is critical, a bitmap font may be preferable.

With X11 Release 5, the X11 server allows for font scaling in a network environment by obtaining font information from a font server running on the network. The font server parses font files of any of a number of supported formats and outputs them in a standardized X Font Service Protocol bitmap format. Although X servers using font service no longer need to parse fonts, most X terminals and workstation-based X servers will still support file-based fonts.

X11 Release 5 also includes font scaling, allowing you to obtain fonts at any point size and resolution without jagged edges. The X server contains algorithms to scale bitmap fonts, Bitstream Speedo fonts, Type 1 fonts, or OpenFonts F3 format fonts.

While the capability of having a font server is nice, it is not necessary. The server itself has the capability to do anything the font server does. OpenWindows Version 3.x contains the X11 "Next Generation" server with MIT's font scaling capabilities, as well as the value-added OpenFont scaler and Type 1 scaler.

The "Next Generation" server will have Type 1 fonts scaled by the Adobe Scaler provided with DPS. Although the OpenWindows Version 3.x server is capable of working with a font server, OpenWindows Version 3.x does not come with a font server. The fonts discussed in the following sections are included in the X11 Next Generation server.

X11 can access the OpenFonts and Type 1 fonts scaling technology through the X Logical Font Description (XLFD) protocol. The XLFD gives numeric values for pixel size, point size, resolution, and average width. Since these variables have an indefinite range of values for a scalable font, these fonts are reported—through `xlsfonts` or `XListFonts()` Xlib requests—as having 0 in these fields. To use them, take the reported name and replace the pixel size or point size field with the desired value. If pointsize and pixel size and Y resolution are all supplied, then the values must be consistent with each other.

Note – For the server to parse these values correctly, the name must be supplied with all the dashes intact. Using wildcards to leave out sections of the name fails, though wildcards are safe to use for individual fields in the name.

Alternatively, the fonts can be accessed by their PostScript language names, with the point size appended to the name (for example, Times-Roman-23). This method works for any scalable font. Note that this is not a portable way of specifying fonts and should be avoided when writing portable code. This font specification is much easier to type and remember than an XLFD format and may be preferable as a command line argument or `.Xdefaults` entry.

Note – This form of font naming will continue to be supported in the "Next Generation" server, but it is recommended that you use XLFD format.

Type 1 Fonts can be installed in the server by typing:

```
xset +fp <dir>
```

where `dir` is the name of the directory containing Type 1 font files; these files have the extension `.pfa` or `.pfb`.

MIT X11 Release 5 compliant servers can work with the font server. To add a font server to the path that the X server searches, type:

```
xset fp
```

When an X server is named, a font server running on a TCP/IP network uses the naming convention

```
tcp/host:port[/directories]
```

where `directories` is an optional parameter that specifies a list of directory names separated by plus signs. For example, a font path containing a TCP server named `ora` might look like this:

```
tcp/ora:7000
tcp/ora:7000/75dpi, /usr/lib/X11/fonts/misc
tcp/ora:7000/100dpi+misc, /usr/lib/X11/fonts/misc
tcp/ora:7000/100dpi+misc, tcp/ora:7001/new-fonts/misc
```

Note – The above example is taken from the *Programmer's Supplement for Release 5*, by David Flanagan (O'Reilly & Associates, Inc., 1991).

The X server port number is a relative offset from a hard-coded preassigned base port number. Conversely, a font server name must specify an absolute port number. The default font server port number is 7000, but this can be changed using a command-line option if it conflicts with other services or font servers.

Selecting Scalable Fonts

The addition of scalable fonts to the X Window System changes the way you address the Xlib function `XListFonts()`. Specifically, you need to ensure backwards compatibility with older applications. In earlier releases of the X Window System, you could insert wildcards for many of the fields in the font name, such as:

```
-*-times-bold-r-*-*-*-120-*
```

When `XListFonts()` comes across a scalable font, it returns the string `0` (the number zero) in the `PIXEL_SIZE` (position seven), `POINT_SIZE` (position eight), and `AVERAGE_WIDTH` (position twelve) fields. With X11 Release 5, the X server and font server must be passed a well-formed font that contains all 14 hyphens specified in the XLFD convention. You may use wildcards in place of any field, but you may not use wildcards in lieu of hyphens. The well-formed version of the previous font name is:

```
-*-times-bold-r-*-*-*-120-75-75-*-*-iso8859-1
```

If `XListFonts()` is passed a font name that is not well-formed, it might not search for scalable fonts.

To have `XListFonts()` search exclusively for scalable fonts, insert the string “`0`” in fields seven, eight, and twelve in the font name.

Deriving Instances of Scalable Fonts

Applications that use a large number of fonts, such as text processors or graphics applications, may want to take advantage of scalable fonts. These kinds of applications need to be modified to recognize and utilize scalable fonts. Note that you must always give `XListFonts()` a well-formed and correctly specified font name.

Code Example 4-9 shows a procedure that determines whether a font name returned by `XListFonts()` represents a scalable font. This procedure is intended to be called each time a font is returned by XListFont`()`.

Note – Code Examples 4-9 and 4-10 are taken from the *Programmer’s Supplement for Release 5*, by David Flanagan (O’Reilly & Associates, Inc., 1991).

Code Example 4-9 Procedure to determine if a font is scalable

```
/*
 * This routine returns True only if the passed name is a
 * well-formed XLFD style font name with a pixel size, point
 * size, and average width (fields 7,8, and 12) of "0".
 */
Bool IsScalableFont(name)
char *name;
{
    int i, field;

    if ((name == NULL) || (name[0] != '-')) return False;

    for(i = field = 0; name[i] != '\0' && field <= 14; i++) {
        if (name[i] == '-') {
          field++;
          if ((field == 7) || (field == 8) || (field == 12))
                if ((name[i+1] != '0') || (name[i+2] != '-'))
                    return False;
        }
    }

    if (field != 14) return False;
    else return True;
}
```

In Code Example 4-10, the procedure takes a scalable font name and a specified point size and loads the returned font at the requested size and at the screen's precise resolution. This procedure is intended to be called with a scalable font name as returned by `XListFonts()`.

Code Example 4-10 Procedure to load a retrieved scalable font

```
/*
 * This routine is passed a scalable font name and a point
 * size. It returns an XFontStruct for the given font scaled
 * to the specified size and the exact resolution of the
 * screen. The font name is assumed to be a well-formed XLFD
 * name, and to have pixel size, point size, and average
 * width fields of "0" and implementation dependent x-
 * resolution and y- resolution fields.  Size is specified
 * in tenths of points. Returns NULL if the name is malformed
 * or no such font exists.
 */
XFontStruct *LoadQueryScalableFont(dpy, screen, name, size)
```

Code Example 4-10 Procedure to load a retrieved scalable font (Continued)

```
Display *dpy;
int screen;
char *name;
int size;
{
    int i,j, field;
    char newname[500]; /* big enough for a long font name */
    int res_x, res_y; /* resolution values for this screen */

    /* catch obvious errors */
    if ((name == NULL) || (name[0] != '-')) return NULL;

    /* calculate our screen resolution in dots per inch. 25.4mm = 1 inch */
    res_x = DisplayWidth(dpy, screen)/(DisplayWidthMM(dpy, screen)/25.4);
    res_y = DisplayHeight(dpy, screen)/(DisplayHeightMM(dpy, screen)/25.4);

    /* copy the font name, changing the scalable fields as we do so */
    for(i = j = field = 0; name[i] != '\0' && field <= 14; i++) {
       newname[j++] = name[i];
       if (name[i] == '-') {
           field++;
        switch(field) {
            case 7:  /* pixel size */
            case 12: /* average width */
           /* change from "-0-" to "-*-" */
               newname[j] = '*';
               j++;
               if (name[i+1] != '\0') i++;
               break;
           case 8:  /* point size */
           /* change from "-0-" to "-<size>-" */
               (void)sprintf(&newname[j], "%d", size);
               while (newname[j] != '\0') j++;
               if (name[i+1] != '\0') i++;
               break;
         case 9:  /* x resolution */
          case 10: /* y resolution */
           /* change from an unspecified resolution to res_x or res_y */
           (void)sprintf(&newname[j], "%d", (field == 9) ? res_x : res_y);
           while(newname[j] != '\0') j++;
           while((name[i+1] != '-') && (name[i+1] != '\0')) i++;
           break;
         }
       }
    }
```

Code Example 4-10 Procedure to load a retrieved scalable font (Continued)

```
    newname[j] = '\0';

    /* if there aren't 14 hyphens, it isn't a well formed name */
    if (field != 14) return NULL;

    return XLoadQueryFont(dpy, newname);
}
```

New Server Options

- `-unsecure`
  With DPS extensions, any application that is allowed to connect to the X Display can also open and write files that the owner of the X Display has permissions for. This is done with PostScript file operators. This leaves a big security hole in the server. Therefore, by default the server will not allow file operators to function. The end user will have to run the server in unsecure mode to get access to the file system.

Note – `-unsecure` is a placeholder for a server option that will be introduced in the new server do implement this functionality. The option may be renamed.

- `-dev`
  This is the frame buffer name of the device to open. There are device options that can be specified with the server as well.

The device options are:

`[left	right	top	bottom]`	orientation relative to previous screen in a two-monitor configuration.
`dpix #`	dots-per-inch in the x direction
`dpiy #`	dots-per-inch in the y direction
`defclass [GrayScale |`	visual to report as default `StaticGray	PseudoColor	StaticColor	DirectColor	TrueColor]`
`defdepth #`	report only `GrayScale` and/or `StaticGray` visuals
`grayvis`	report only `GrayScale` and/or `StaticGray` visuals

Note – The device options have been added to the server to remain compatible with the X11/NeWS server as much as possible.

Other options new to the server are:

`-cp dir1[:dir2...]`	search path for configuration files and dynamically loaded modules
`-flipPixels`	reverse black and white pixel values
`-pit`	set bell pitch
`-dur`	set bell duration
`-mden`	set pointer acceleration denominator
`-keyboarddev`	specify keyboard device
`-pointerdev`	specify pointer device

For more information, consult the man page for `Xsun`.

Enabling Technology

Enabling technology is technology that contains access solutions for people with physical, visual, and hearing disabilities. In the past, workstations have been deficient in providing access technology. The current technology has not enabled handicapped access. Issues involving ease-of-use impact everyone. Access solutions, such as voice recognition and font size control capabilities, are being marketed on other platforms as productivity boosters for the non-handicapped computer user.

Enabling technology increases system access, system control, and overall task productivity for everyone. The traditional name for this has been *adaptive technology*, but the problem with this term is that the word “adaptive” implies that this technology applies only to users with disabilities. The name “enabling technology” has been chosen because enabling someone implies increased productivity and usability. In other words, everyone benefits from enabling technology.

With OpenWindows Version 4, Sun is attempting to solve the problem of accessibility for all users. The following three enabling technology features are accessed via command-line from OpenWindows Version 4:

- T-Lock (also known as “Sticky Keys”)
  Allows you to press modifier keys (Shift, Control) before the keys they modify, rather than requiring that they be pressed simultaneously.

  When T-Lock is enabled, the modifier keys (Shift, Control, Meta, and Alt) will retain state as follows. When a modifier key is pressed and released, it “latches”. A latched modifier key will modify the next non-modifier key as though it had been pressed and held normally. When a modifier key is pressed and released twice in a row, it “locks”. A locked modifier key will modify any non-modifier key until it is unlocked. When the locked modifier key is pressed a third time it unlocks.

- Slow Keys
  Delays the recognition of keypress events so that users with limited coordination can accidentally bump keys without sending keypress events.

  When the Slow Keys option is enabled, keystroke events are filtered as follows: when a key or keys are pressed then released within the time specified as the Keystroke Recognition Delay, both the keypress(s) and the keyrelease(s) are ignored. When multiple keys are pressed and all but one are released prior to the Keystroke Recognition Delay time only the key press held after this time is recognized as a keystroke event.

- Repeat Keys
  Delays the onset of keys repeating so that users with limited coordination have time to release keys before they start sending multiple characters.

  Enabling and disabling the key repeating is already supported by the servers and by the workspace properties panel. However, the key repeating functionality would be modified to allow the following configurations:
  - The Key Repeat Delay specifies the length of time between the first character being transmitted and the onset of key repeating (that is, the transmission of the second character)
  - The Key Repeat Rate specifies the delay between the character transmissions once repeating starts

The man page for `Xsun` contains further information on these options.

Display PostScript System Extension (DPS)

In the last quarter of 1992, SunSoft, Inc. and Adobe Systems Incorporated made a technology agreement to integrate DPS with SunSoft's new X11 Release 5-compliant server. DPS is implemented as an extension to the X11 server. It allows for clients to do PostScript imaging on X window objects. The following are features of DPS:

- Level II compliant DPS extension
- access to Folio Fonts from DPS using `/findfont` operator
- `pswrap` utility to binary encode PostScript code for better performance
- function calls for all PostScript operators for access from C source code
- access to DPS from any X toolkit and Xlib
- support for DPS toolkit (`libdpstk`)
- support for `libpsres`
- memory management including memory reclamation
- composite fonts for Asian Language support
- ability to cache user-defined paths for fast rendering
- support for generating and storing forms
- device-independent color support (CIE)

- improved halftones

For additional information, you can contact the Adobe Developer's Association at (415) 961-4111.

You can also tap into their file server for information: **ps-file-server@adobe.com**

See Chapter 5 on “Graphics” in this book, or the following books for information on the DPS API. These books are available from the MIT X11 Release 5 distribution.

An Overview of the Display PostScript System, Adobe Systems Incorporated, Mountain View, CA.

X Window Systems Programmer’s Supplement to the Client Library Reference Manual, Adobe Systems Incorporated, Mountain View, CA.

pswrap Reference Manual, Adobe Systems Incorporated, Mountain View, CA.

Perspective for Software Developers, Adobe Systems Incorporated, Mountain View, CA.

Color Extensions, Adobe Systems Incorporated, Mountain View, CA.

Client Library Reference Manual, Adobe Systems Incorporated, Mountain View, CA.

Extensions for the Display PostScript System, Adobe Systems Incorporated, Mountain View, CA.

The following book is available as of March, 1993, from Adobe Systems Incorporated:
Display PostScript Programmer’s Guide, Adobe Systems Incorporated, Mountain View, CA.

Internationalization Support in the Server

This section is being written as OpenWindows Version 4 is developed. Since it will be compliant with Release 5 of MIT, we can examine the internationalization support in the server as available from MIT and be fairly certain that it will not be very different. There may be SunSoft value-added features; these will be optional.

Note – The X Input Method server may differ from the MIT distribution example server.

An *internationalized* application is one that runs, without changes to the binary, in any specified locale. The largest new part of Solaris 2.x is the support for writing internationalized programs.

Solaris 2.x has the ANSI-C internationalization features; the C library reads a “localization database” customized for each locale. However, in the U. S., Solaris 2.x is shipped with the default C locale database.

Because the word “internationalization” contains 20 letters, it is often abbreviated as I18N.

Solaris 2.x supports the X input method (`Ximp`) protocol implementation of I18N.

To support internationalization, locale management functions and routines for handling text drawing, such as font set handling, metric, and string measuring functions, and string drawing functions, have been added to Solaris 2.x.

Many Xlib functions have been respecified to clarify their handling of strings in internationalized applications. Functions have been added to support internationalized window manager and text properties for inter-client communication, and new resource manager functions exist to support local databases.

In addition, many resource manager functions have been respecified to clarify their behavior in internationalized applications. There is also new support for internationalized text input.

For further information in this document on I18N, Chapter 7, “Internationalization,” explains some basic concepts of I18N, as well as ANSI-C I18N.

In addition, the bibliography at the end of this chapter has sources which contain detailed explanations on character encoding. This may help you to understand how character sets are represented in different languages.

Using X11 Release 5 to Internationalize

The internationalization techniques discussed here have been sufficient only to internationalize terminal-based applications. Many applications use multibyte strings in an encoding that contains shift sequences and nonuniform width characters. Some X clients need to communicate with other clients. Therefore, an internationalized interclient communication method is required because internationalized clients can run in different locales. Other X clients make heavy use of resource files and databases; these clients need a mechanism for determining the correct localization of resources.

The function `setlocale` by itself is insufficient for locale management in an X application. Xlib contains two new functions which are used with `setlocale` when an X application starts up. Changes to the Xt architecture are also needed to support the new internationalization features of Xlib.

The following describes some of the internationalization changes and issues in X11 Release:

- **Internationalized Text Output in X11 Release 5:** The new text output routines are based on an Xlib extraction called the `XFontSet()`, which contains all the fonts needed to display text in the locale for which it was developed. See *Programmer’s Supplement For Release 5 of the X Window System, Release 11*, for more information.

Note – X11 Release 5 does not have provisions for text displayed from right to left, such as Hebrew or Arabic.

- **String Encoding Changes for Internationalization**
  The encoding of strings is a major concern regarding internationalizing X applications. An X client must be able to communicate with the X server, and often with a window manager, a session manager, and other clients. There are a variety of ways to encode colors, strings, bitmap files, resource names and values. There are also many ways to pass names to the X server and to copy strings selected in one client to another. Strings can be in the encoding of the locale.

- **Internationalized Interclient Communication**
  X11 Release 5 provides functions that do not assume internationalized interclient communication will be done with ASCII strings.

- **Localization of Resource Databases**
  Sometimes methods of encoding data are implementation-dependent. For example, a Japanese-speaking end user who wishes to customize an application written by an English-speaking programmer may use symbolic representations of English words meaningless to another user.

- **Internationalized Text Input**
  Unlike a program that only runs in one language, an internationalized program cannot assume a one-to-one correspondence between the keyboard and input characters. For example, the keyboard correspondence between keystrokes and characters is much more complex in Greek than in English. When the codeset of a locale contains more characters than keystrokes, an *input method* is required to map keystrokes to characters. Solaris 2.x supports the X Input Method (XIM) and X Input Context (XIC) keyboard input abstractions, as well as the functions `XmbLookupString` and `XwcLookupString`; these functions return a string in the encoding of the specified locale.

The XIM designers envision their I18N text input mechanism will be incorporated within toolkits and thus hidden from developers. It is therefore strongly recommended you use the OLIT toolkit for your I18N interface.

Note – For more information on XIM and X11 Release 5 internationalization, see the *Programmer's Supplement for Release 5 of the X Window System, Version 11*, by David Flanagan (O'Reilly & Associates, Inc., 1991).

The X Color Management System (Xcms)

The X Color Management System (Xcms) is a device-independent color application programming interface (API). Xcms is a set of Xlib functions that allow you to allocate device-independent colors. Xcms provides precise control over the allocation of colors. Xcms also comes with a new syntax for the color strings used throughout Xlib.

The following components of X11 Release 5 are new:

- syntax for device-independent color strings
- Xlib provides a mechanism for device-independent string names for colors
- `xcmsdb`, a database read by Xlib that maps color names to device-independent color specifications
- the Xcms API
- the X Device Color Characterization Conventions (XDCCC)

Before you proceed with how to use Xcms API, it is important to understand the fundamentals. While it is not the focus of this chapter, consult the bibliography to get more information on the science of colorimetry.

When you use a device-independent color specification, the resulting display has identical colors on all devices. X11 Release 5 supports an international standard known as CIEXYZ. In this format, as is the case with RGB, a color is described by the value of three coordinates; the associated color space is usually referred to by those coordinates.

Solaris supports the CIEXYZ color space, the related spaces known as CIExyY, CIEuvY, CIELuv, CIELab, and the Tektronix-designed color space, TekHVC. TekHVC is a perceptually uniform color space designed to be easy to understand; it is mathematically related to the CIE spaces. The TekHVC colors are represented by Hue, Value, and Chroma. *Hue* is what distinguishes one color family from another—red, blue, or green. *Value* describes how light or dark a color is. *Chroma* describes the color saturation. Together, the HVC values describe a color solid.

The Client Color Name Database

Although the X client supports device-independent colors, the X protocol and X server still only use device-dependent RGB colors. Therefore, you cannot use device-independent color specifications in the database of color names read by the server.

You can give symbolic names to device-independent colors, and Solaris supports a client-side color database that maps color names to device-independent or device-dependent color specifications.

While Solaris supports this color database, the MIT release does not provide one. The file `$OPENWINHOME/lib/X11/Xcms.txt` contains the client-side database X clients use by default. To specify a different file, set the `XCMSDB` environment variable.

X Device Color Characterization

The X Device Color Characterization Conventions (XDCCC) specify how Xcms converts from device-independent color used by X clients to the device-dependent colors used by the X server.

Included in X11 Release 5 is a new client, `xcmsdb`.

Device-Independent Color API

The X Color Management System (`Xcms`) API is part of Xlib. It contains many new functions and data types, all of which are prefixed by `Xcms`. Many of these functions are similar to earlier X functions. A group of `Xcms` functions are used to manipulate the *color conversion context* (CCC) `Xcms` datatype. The CCC contains attributes that control the details of color conversion between color spaces. `Xcms` automatically handles default CCCs.

What follows is a brief description and introduction to the `Xcms` API. For more information about `Xcms`, refer to the resources listed at the end of this chapter.

`Xcms` functions require more computation than device-dependent functions because the former perform trigonometric and other floating-point operations. These floating-point operations are directly implemented in Xlib, and they are relatively slow because they do not have a highly-optimized math library and cannot take advantage of floating-point hardware. Therefore, adding `Xcms` functions to a program can make it run slower, especially during gamut compression.

Note – For more information, see the *Programmer's Supplement for Release 5 of the X Window System, Version 11*, by David Flanagan (O'Reilly & Associates, Inc., 1991).

The Xcms Functions

The `XcmsColor` structure is a device-independent version of the `XColor` structure; it is used by the following `Xcms` functions. `XcmsColor` contains a pixel value, a format value denoting the colorspace used to describe the color, and a union of structures that specify parameters for all formats supported by Solaris 2.x.

The following functions are provided within `Xcms` for manipulating color map cells. Although they use the `XcmsColor` structure to specify device-independent color, they are similar to device-dependent color allocation functions.

- `XcmsAllocColor`
  This function is similar to `XAllocColor()` except the color can be specified in any format. The `XcmsAllocColor()` function ultimately calls `XAllocColor()` to allocate a read-only color cell (color map entry) with the specified color. It returns the pixel value of the color cell and the color specification allocated.

- XcmsAllocNamedColor
  This function is similar to XAllocNamedColor() except the color returned can be in any format specified. XcmsAllocNamedColor() ultimately calls XAllocColor() to allocate a readonly color cell with the color specified by a color string. It returns both the color specification (exact specification) and the actual color specification (screen specification) stored.

- XcmsLookupColor
  This function looks up the string name of a color with respect to the screen associated with the specified color map. It returns both the exact color values and the closest values provided by the screen with respect to the visual type of the specified color map.This function performs gamut compression. It is similar to XLookupColor(), but has an argument to specify the desired color space for the return values, and returns a device-independent color specification.

- XcmsQueryColor
  This function obtains the RGB value for the pixel value in the pixel member of the specified XcmsColor structure, and then converts the value to the target format as specified by the result_format argument.

5. XcmsQueryColors
   This function obtains the RGB values to the target format as specified by the result_format argument.

- XcmsStoreColor
  This function converts the color specified in the XcmsColor structure into RGB values and then uses this RGB specification in an XColor structure, whose three flags (DoRed, DoGreen, and DoBlue) are set, in a call to XStoreColor(), to change the color cell specified by the pixel member of the XcmsColor structure. This pixel value must be a valid index for the specified color map, and the color cell specified by the pixel value must be a read/write cell. If the color map is an installed map for its screen, the changes are visible immediately. Due to the screen's hardware limitations or gamut compression, the color stored in the color map may not be identical to the color specified.

- XcmsStoreColors
  This function converts the colors specified in the array of XcmsColor structures into RGB values and then uses these RGB specifications in XColor structures, whose three flags (DoRed, DoGreen, and DoBlue) are set, in a call to XStoreColors(), to change the color cells specified by the pixel member of the corresponding XcmsColor structure. Each pixel value must be a valid index for the specified color map, and the color cell specified by each pixel value must be a read/write cell. If the color map is an installed map for its screen, the changes are visible immediately. Due to the screen's hardware limitations or gamut compression, the colors stored in the color map may not be identical to the colors specified.

Note – When using the TekHVC color space, you should verify that the minimum and maximum Values passed to the `Xcms` functions are valid. If either of the specified Values is outside the TekHVC colorspace, gamut compression takes place.

Color Conversion Context

The *color conversion context* (CCC) is a structure of type `XcmsCCC`. In X11 Release 5, a CCC is automatically associated with each color map. The CCC contains the attributes controlling the details of color conversion from one space to another, such as the procedure that is called to perform gamut compression when a device-independent color specification is outside the range of displayable colors for the specified device.

Gamut Querying Functions

This section describes the gamut querying functions that are provided by Xlib. These functions allow the client to query the boundary of the screen's color gamut in terms of the CIE L*a*b*, CIE L*u*v*, and TekHVC color spaces.

Gamut querying functions are also provided that allow you to query the color specification of:

- white (full intensity red, green, and blue)
- red (full intensity red while green and blue are zero)
- green (full intensity green while red and blue are zero)
- blue (full intensity blue while red and green are zero)
- black (zero intensity red, green, and blue)

The white point associated with color specifications passed to and returned from these gamut querying functions are assumed to be the Screen White Point. This is a reasonable assumption, because the client is trying to query the screen's color gamut. Note that the following naming convention is used for the "Max" and "Min" functions:

- `Xcms`<*color_space*>`QueryMax`<*dimensions*>
- `Xcms`<*color_space*>`QueryMin`<*dimensions*>

Note that the <*dimensions*> consists of letter or letters that identify the dimension or dimensions of the color space that are not fixed.

The following functions can be used to verify that requested color specifications are legal for a specified TekHVC color space:

- `XcmsTekHVCQueryMaxC`
  Use this function to obtain the maximum Chroma for a specified Hue and Value.

- `XcmsTekHVCQueryMaxV`
  Use this function to obtain the maximum Value for a given Hue and Chroma.
- `XcmsTekHVCQueryMinV`
  Use this function to obtain the minimum Value for a specified Hue and Chroma.
- `XcmsTekHVCQueryMaxVC`
  Use this function to determine, given a Hue, the maximum Chroma in TekHVC color space displayable by the screen and the Value at which that maximum Chroma is reached.
- `XcmsTekHVCQueryMaxVSamples`
  Use this function to obtain a specified number of TekHVC specifications such that they contain a maximum Value for a specified Hue and Chroma at which the maximum Values are reached.

Note – See *The X Color Management System*, by Al Tabayoyon et al for more information on functions for color spaces (O'Reilly & Associates, Inc.).

Other Changes to Window Support in X11 Release 5

The following changes to the X11 Release 5 X Window System are documented in the *Programmer's Supplement for Release 5 of the X Window System, Version 11*, by David Flanagan (O'Reilly & Associates, Inc., 1991):

- To facilitate ANSI and POSIX compliance, X11 Release 5 has new standard header files.
- Minor changes have been made to Xlib and Xt.
- The `editres` client. has been added to enable you to interactively edit the widget resources of any running application that uses the Editres protocol.
- Four Xaw widgets have been added: `Panner`, `Porthole`, `Repeater` and `Tree`. Many other Xaw widgets have been modified. The Editres protocol is supported by the Xaw `VendorShell` widget.
- Many Xmu library routines have been modified. The Editres protocol is supported by the Xmu library.
- The X11 Input Device Extension, which allows input from devices other than a keyboard or mouse, is now a standard of the X Consortium.
- The MIT Implementation of X11 Release 5 has two new secure authorization protocols: XDM-AUTHORIZATION-1 and SUN-DES-1.
- Support for Portable Compiled Fonts across architectures is provided via the *pcf* format.
- Hebrew, Kanji (Japanese), and Hangul (Korean) fonts are provided.
- Clients have been added to the MIT X11 Release 5 implementation; others have been modified or replaced.
- The MIT X server runs faster. Reimplementation of the Xt Translation Manager results in less memory use at runtime.

Note – None of these are required. They have been put here for your information and therefore can be used to enhance the capabilities of your applications.

Suggested Reading

The following books contain additional information you may find useful when programming for windows under Solaris 2.x.

X Toolkit Intrinsics Reference Manual for X11 Release 4 and Release 5, David Flanagan, O'Reilly & Associates, Inc., Sebastopol, CA.

X Windowing System: The Complete Reference to Xlib, X Protocol ICCCM, XLFD, Robert W. Sheifler et al, Digital Press, Burlington, MA.

Bibliography

American National Standard for Information Systems - Programming Language C, ANSI X3.159-1989, Published by American National Standard Institute.

Color Extensions, Adobe Systems Incorporated, Mountain View, CA.

Client Library Reference Manual, Adobe Systems Incorporated, Mountain View, CA.

Definition of the Porting Layer for the X v11 Sample Server, Susan Angebranndt, Raymond Drewry, Phillip Karlton, Todd Newman, Bon Scheifler, and Keith Packard. MIT X Consortium.

Display PostScript Programmer's Guide, Adobe Systems Incorporated, Mountain View, CA.

Extensions for the Display PostScript System, Adobe Systems Incorporated, Mountain View, CA.

OLIT 3.2 Reference Manual, Part Number 801-4246-10, SunSoft, Inc., Mountain View, CA.

OLIT Internationalization Features in Asian OpenWindows, Part Number 801-2415-10, SunSoft, Inc., Mountain View, CA.

OpenWindows Version 3.2 Programmer's Guide, Part Number 801-xxxx-xx, SunSoft, Inc., Mountain View, CA.

Programmer's Supplement for Release 5 of the X Window System, Version 11, David Flanagan, O'Reilly & Associates, Inc., Sebastopol, CA.

Strategies for Porting the X v11 Sample Server, Susan Angebranndt, Raymond Drewry, Phillip Karlton, Todd Newman, Bon Scheifler, and Keith Packard. MIT X Consortium.

The X Color Management System, Al Tabayoyon et al, O'Reilly & Associates, Sebastopol, CA.

The X Window System Programming & Applications with Xt, OPEN LOOK Edition, Douglas A. Young and John A. Pew, Prentice Hall, Englewood Cliffs, NJ.

X Server Internals: A Tutorial On the MIT Sample Server at Release 4, Keith Packard, MIT X Consortium.

X Window System, Version 11, Release 5. Release Notes, MIT X Consortium Staff, MIT Laboratory for Computer Science.

X Window System, Version 11 Input Method Specifications, Draft 3.0, October 1990, Bill McMahon.

The X Window System Server, Elias Israel and Erik Fortune, Digital Press.

X11 Input Extension Library Specification, X11 Release 5, Mark Patrick and George Sachs, MIT X Consortium, MIT Laboratory for Computer Science.

X11 Input Extension Porting Document, X11 Release 5, George Sachs, MIT X Consortium, MIT Laboratory for Computer Science.

X11 Input Extension Protocol Specification Version 1.0, Mark Patrick and George Sachs, MIT X Consortium, MIT Laboratory for Computer Science.

An Overview of the Display PostScript System, Adobe Systems Incorporated, Mountain View, CA.

X Window Systems Programmer's Supplement to the Client Library Reference Manual, Adobe Systems Incorporated, Mountain View, CA.

pswrap Reference Manual, Adobe Systems Incorporated, Mountain View, CA.

Perspective for Software Developers, Adobe Systems Incorporated, Mountain View, CA.

OpenWindows 4

Introduction

The OpenWindows product has changed significantly in Solaris 2.x. The strategic direction for the OpenWindows product was to build it on standards. That is, OpenWindows provides developers with Application Program Interfaces (APIs) that are widely used, making the product source compatible and, if possible, binary compatible with other MIT-based X Window System products. Of course, binary compatibility requires use of similar hardware and system software.

The OpenWindows product is based on the following components:

- OPEN LOOK Window Manager
  The familiar Inter Client Communication Conventions Manual (ICCCM) compliant window manager.

- OLIT/DevGuide
  OpenWindows is focusing on the MIT Xt-based OLIT toolkit. This will be the premier toolkit promoted by SunSoft, Inc. DevGuide, the graphical interface builder, has been enhanced to provide full support for the OLIT user interface.

- X11 Server
  OpenWindows Version 4 will be based on MIT X11 Release 5 server; it will adhere to all of MIT's standard APIs with SunSoft value-added components.

- Display PostScript
  OpenWindows Version 4 will support the Display PostScript® System (DPS) as an extension to the server.

In addition, all the desktop integration has been preserved from earlier releases to standardize communication protocols. Figure 4-1 illustrates the OpenWindows components.

New features of the OpenWindows server include:

- standardized graphics hardware support through the device-dependent X (DDX) layer
- support for multiple hardware color maps (MHC) and multiple plane groups (MPG)

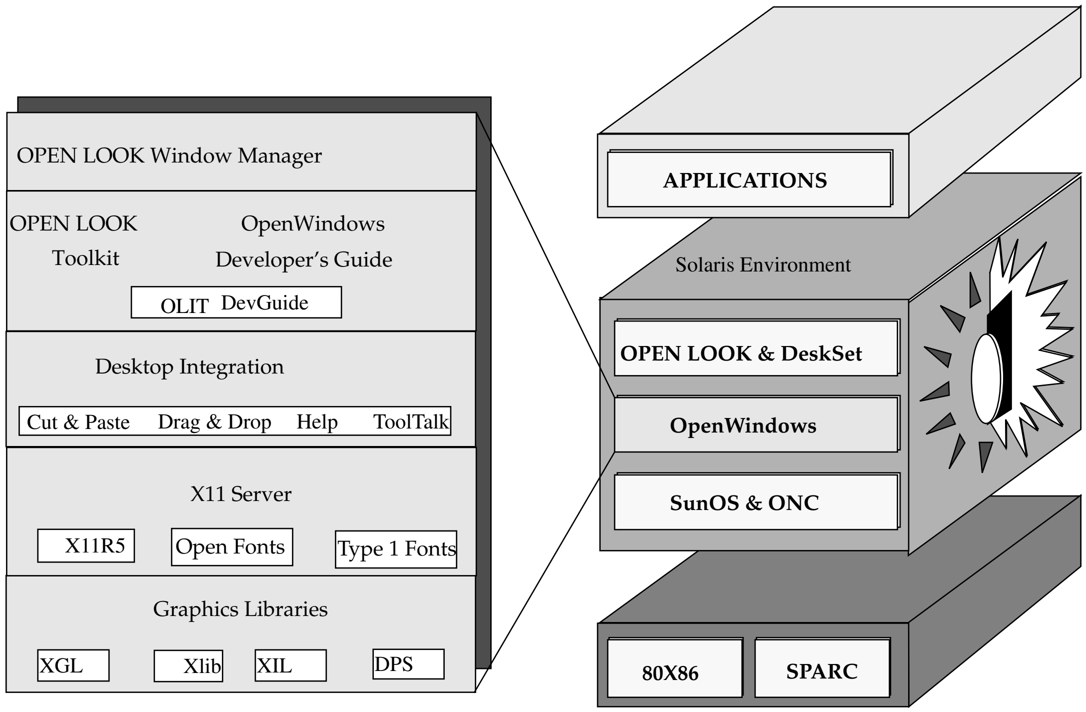

Figure 4-1 OpenWindows Components

- Direct Graphics Access (DGA)
- support for OpenFonts, Type 1 fonts, and PostScript® fonts
- support for dynamically-loadable extensions
- standardized input device interface using X Input Extensions
- enabling technology
- new server options
- internationalization
- device-independent color support
- Display PostScript System (DPS) extension to provide *de facto* standard PostScript imaging

Server performance will be comparable to MIT's Sun Sample Server. There will be no support for SunView™ or NeWS®/tNt with this release.

New OLIT features include:

- new widgets (`NumericField`, `TextLine`, `FileChooser`, `FontChooser`)
- bug fixes
- internationalization support
- new input focus mode based on supercaret
- support for ANSI C and C++
- unified Desktop look and feel (use of OLGX)
- improved programmer's interface for ScrollingList

OPEN LOOK Intrinsics Toolkit

The OPEN LOOK Intrinsics Toolkit (OLIT) is based on MIT's X Toolkit Intrinsics (Xt) and was jointly developed by AT&T and Sun Microsystems, Inc. Beginning with OpenWindows Version 3.0, SunSoft, Inc. became responsible for the development and enhancement of the OLIT toolkit.

OpenWindows Version 3.1, bundled with Solaris 2.x, is the last fully featured release of the three toolkits currently supported by OpenWindows. The three toolkits are OLIT, XView™, and tNt.

Note – For the next major OpenWindows release, all efforts will go towards the development and enhancement of one toolkit, OLIT. At the same time, XView and tNt will begin to be phased out. Please contact your SunSoft marketing representative for more information on the dates and times for this phase-out.

This section of the porting guide is aimed at:

- Developers who are evaluating the OLIT toolkit and would like to know what new features have been added to the latest release of OLIT—in particular, developers who are thinking of moving from other toolkits (XView, tNt, Motif) to OLIT.
- OLIT developers who would like to enhance their existing product with new features of OLIT.

This section discusses some of the new OLIT features introduced in the latest releases of OpenWindows. Each feature is listed and then described in detail, with some sample programs to give you an understanding of how to utilize the feature. This section describes new features in OLIT 3.1 and 3.2.

Bug Fixes

Although bug fixes are not features, they are one of the most critical points in evaluating the toolkit. Since the introduction of 3D OLIT (OLIT 2.5) in early 1991, OLIT has gone through several revisions, and heavy testing, and a large number of bugs have been fixed. The toolkit has matured rapidly to become stable with state of the art features of GUI toolkits. In addition to bug fixing, a significant amount of work has been done to improve OLIT performance and fix memory leaks in the widgets.

ANSI C Function Prototypes

With Solaris 2.x, ANSI C has become the default compiler. One of the things ANSI C requires is function prototypes. All OLIT header files are now ANSI C-compliant and have function prototypes. The header files have been updated to work with C++. Therefore, you are strongly encouraged to use either the ANSI C or C++ compiler to be compatible with the Solaris 2.x environment.

Internationalization

Internationalization (I18N) in the context of Graphical User Interface (GUI) software, refers primarily to adding the flexibility needed to operate in situations where English and the associated subset of the Roman script are not the only way you can interact textually with the interface. That is, the toolkit is flexible about entering and rendering text in different scripts and writing systems.

Localization (L10N) refers to providing support for a particular *locale*. This entails storing information about the locale's script, then writing a system and cultural environment so the internationalized software can "operate" in that locale. Locales for which information is stored are called *supported* locales. Each locale has supporting information provided as a Feature Package by SunSoft.

An application developed and compiled with international OLIT will be able to operate in any of the supported languages and process data according to the rules of that language. The first version of international OLIT supports Japanese, Korean, Chinese, and traditional Chinese, with other Asian locales to follow. European locale handling is also supported, if supported by the operating system. Adding new locales to an international OLIT application consists primarily of providing a new set of messages, labels, and other strings which can be non-intrusively stored in application default files in parallel directory hierarchies, one for each locale. OLIT automatically picks up these defaults from the appropriate hierarchy, depending on the locale. The only actual changes to code can be minimized to the following actions:

- Removing hard-coded values for string resources in the program; delegating these to defaults files.
- Changing the default text format from its single-byte default.

- Specifying the style of interaction with the input method (for Asian locales).

The *input method* is the method by which text is entered into the system. OLIT uses the portable Xlib API to access the X Input Method. Although OLIT can use the default input method implementation provided by SunSoft for each supported locale, OLIT operates equally well with input method implementations from OEMs or independent software vendors (ISVs).

System Requirements

OLIT supports Level 4 internationalization. The first release of internationalized OLIT, Asian OpenWindows, enables developers to localize to several Asian languages. To run this release of international OLIT, you need Solaris 2.x and the Feature Package for the locale in which you intend to run it. For example, you need the Japanese Feature Package (JFP) for Japan and the Korean Feature Package (KFP) for Korea. These packages consist of extensions to Solaris and incorporate numerous facilities for handling local linguistic and cultural conventions.

Note – Level 4 is a Sun Microsystems internal definition of a level of internationalization. For more information, see the chapter on “Internationalization.”

Overview of Internationalizing Applications

To internationalize your application, you must address the following issues:

- **Locale Setting**
  Locale setting is the method by which the language or cultural environment is set.
- **Character Encoding**
  Character encoding is the method by which a language’s character set is represented. Conventional applications use 7-bit ASCII encoding to represent each character. However, some languages have larger character sets that require more than the 128 character range permitted by 7-bit encoding.
- **Font Set Handling**
  Font handling is simple for the languages conventional OLIT supports because they only use one character set. However, some languages use multiple character sets and therefore require multiple fonts or a *font set*.
- **Localized Text Handling**
  The international developer should be able to use error messages, menu text, and button text in the native language and have those strings retrieved in the language specified by the locale.

- **Input Method:**
  The input method is the method by which end users enter text in a particular language. To enter data into a conventional application, the end user types in the information to be processed. Some languages, however, use multiple alphabets which require several keystrokes to create one character.

- **Standards**
  Software internationalization is supported by many standards organizations, including IEEE (POSIX), ANSI, X/Open, and the MIT X Consortium. To make applications portable across a wide variety of hardware platforms, it is important to use a toolkit, such as international OLIT, that follows these standards as much possible.

Locale Setting

The X Toolkit Intrinsics, on which OLIT is layered, provides an application resource, called `xnlLanguage` (class `XnlLanguage`), which announces the end user's locale to the toolkit and the operating system.

Resource	Type	Default	Access
`XtNxnlLanguage`	`XtNstring`	`NULL`	I

OLIT supports all locales that are supported by SunOS on any given system. As mentioned earlier, locale support is added by installing the locale's Feature Package. Sun currently supports most European locales (such as `de`, `fr`, and `it`) and four Asian locales: `ko`, `ja`, `zh`, and `zh_TW`. There are four ways to establish the locale:

1. Specify `-xnlLanguage` on the command line of an OLIT application. For example, to set the locale for an application called `myapplication` to Korean, you type:

```
% myapplication -nxlLanguage ko
```

2. Specify `*xnlLanguage: <language>` in an X11 Resource Manager database file. For example, to set the locale to traditional Chinese, add the following line to your `.Xdefaults` file:

```
*xnlLanguage: zh_TW
```

3. Set the OpenWindows locale category `BasicLocale` through the Workspace Prop Sheet and optionally the four categories `DisplayLang`, `Numeric`, `InputLang`, and `Time`.

4. Set the `LANG` environment variable in the shell that you are starting the application from. For example, to set the locale to Japanese with `csh`, type:

```
% setenv LANG ja
```

To set the locale to Chinese, type:

```
% setenv LANG zh
```

These four ways of establishing locale are listed in order of precedence. Establishing the OS locale is the responsibility of OLIT and Xt. You should not use the OS function `setlocale()` in your OLIT applications because OLIT already calls this function internally and depends on that state to configure its internals.

Character Representation and Text Formats

International OLIT supports three character representation types, or *text formats:*

- Single byte, which represents each character with one byte.
- Multibyte, which represents characters with a variable number of bytes. Each locale has a unique definition of variable width multibyte characters.
- Wide character, which represents each character with a fixed number of bytes (4 in SVR4).

You can create single-byte, multibyte, or wide character OLIT objects, or a combination of these in a single application. If you are writing an application intended for single-byte locales only, you may want to use single-byte text format to avoid the performance overhead incurred by processing multibyte text. However, if you use single-byte format, you lose the flexibility of handling locales that contain multibyte characters, such as Japanese, Chinese, and Taiwanese.

International OLIT supports the single-byte text format primarily for backward compatibility.

The multibyte text format supports the use of ASCII. While programming with multibyte text can be difficult because each character can potentially be a different size, on the positive side, the multibyte format uses memory efficiently. This format is, therefore, useful for storing display-only text which needs to be manipulated infrequently.

Wide character text format, on the other hand, is easier to program since all characters are represented with the same number of bytes. However, the wide character format consumes more storage because it represents all characters with a fixed number of bytes. If all characters are ASCII, many of the bytes will be superfluous.

When deciding which text format is appropriate for a user interface, you should take the following considerations into account:

- Conversion between formats reduces performance.
- OLIT honors the requested text format which can be bound to each widget.
- Processing multibyte data is inherently more time-consuming than processing single-byte or wide character data. Objects that perform intensive data manipulation (for example, a text-editing object such as OLIT's `TextEdit` or `TextField` widgets) will perform better if created as wide character objects (or as single-byte objects as an optimization, if it is certain that only 8-bit data will be handled by the object).

You should design your application so that conversion between formats is minimized. For example, you may want to decide on an object-by-object basis if textual data will be processed intensively and use wide character format if it is.

Note – The widget makes assumptions based on the text format. For example, if you specify the text format of a widget to be single byte and supply a wide character label, an error results.

OLIT widget resources for presentation text are of type `OlStr`. For more information on this type, see "Setting the Default Text Format for an Entire Application".

To provide support for multiple text formats, international OLIT introduces the `XtNtextFormat` resource. This resource allows you to inform widgets what text format to expect for resources for presentation text. Note that this is an interface for programmers only; there is no equivalent interface for end users to specify widget text formats. This allows you to decide the text format of objects and freeze them; so you can then use this information to decide whether you need to convert the text format when you transfer data from one object to another, thereby improving the performance of your applications. Also note that the text format of an object is persistent for the lifetime of the object; it is established at object creation time and cannot be changed.

There are several ways to set the text format:

- Do nothing. If you do nothing, the text format defaults to single byte.
- Set the default text format for the entire application with `OlSetDefaultTextFormat()`.
- Set the text format for an individual widget by changing the widget's `XtNtextFormat` resource.

Setting the Default Text Format for an Entire Application

OLIT maintains a default text format that widgets inherit when they are created without a specified text format by one of the `XtCreateWidget()` family of functions.

For compatibility with previous OLIT releases, the default text format is single byte unless you change it. To change it, use the OlSetDefaultTextFormat() function, which is defined as follows:

```
    void OlSetDefaultTextFormat(OlStrRep format);
```

where format specifies the character representation of the text.

The format argument can have the following values:

format Value	Meaning
OL_SB_STR_REP	Single-byte character representation
OL_WC_STR_REP	Wide character representation
OL_MB_STR_REP	Multibyte character representation

Your application should call OlToolkitInitialize(), then call OlSetDefaultTextFormat(), and then create the widget hierarchy. Objects subsequently created will have the text format specified in the most recent call to OlSetDefaultTextFormat(), unless overridden by explicit arguments. Note that if the application is single byte only, it does not need to call OlSetDefaultTextFormat().

You can create a widget hierarchy consisting of a combination of multibyte, wide character and single-byte objects. To do this, either change the default text format between widget creation calls or specify an object's text format explicitly when creating it.

Setting the Text Format for an Individual Widget

OLIT provides a resource, XtNtextFormat, that allows you to set the text format for individual widgets.

Resource	Type	Default	Access
XtNtextFormat	XtROlStrRep	OL_SB_STR_REP	GI

The XtNtextFormat resource can have any of the following values:

Value	Meaning
OL_SB_STR_REP	Single-byte character representation (for backward compatibility)
OL_WC_STR_REP	Wide character representation
OL_MB_STR_REP	Multibyte character representation

You can only specify the `XtNtextFormat` resource once for each widget at creation time. You cannot set it with `XtSetValues()`. The `XtNtextFormat` resource automatically propagates to the component widgets of a compound widget. For example, if you set it for a `TextField` widget, it will automatically be set for the `TextEdit` widget component of the `TextField` widget.

Font Set Handling

To represent data that consists of multiple character sets, Xlib provides the notion of a *Font Set*. An `XFontSet` is an X11 Release 5 data structure that supports this notion. From the user's perspective, an `XFontSet` represents a list of X11 Logical Font Description (XLFD) fonts that allows the application to fully represent the characters used in a particular locale. For full details on Font Sets, refer to the X11 Release 5 documentation.

To specify a font set in a resource file, use a comma-separated list of fonts. For example, you can enter the following in your `.Xdefaults` file for a Japanese fontset:

```
*font: -misc-fixed-medium-r-normal--20-200-75-75-c-100-iso8859-1, \
 -sun-gothic-medium-r-normal--14-120-75-75-c-120-jisx0208.1983-0, \
 -sun-gothic-medium-r-normal--14-120-75-75-c-60-jisx0201.1976-0
```

When you internationalize your application, you should specify fonts in a resource file. If you specify them within your application, it will be impossible for others to localize it. OLIT handles `XFontStruct` and `XFontSet` in a unified way, as is described in the next section.

OlFont

OLIT unifies support for both the font and font set notions by introducing the `OlFont` type for objects that display text. `OlFont` is an opaque pointer type whose interpretation depends on the setting of the `XtNtextFormat` resource of the associated object. If you create an object as a multibyte or wide character, the value of the `OlFont` field will be a valid `XFontSet` identifier. If you create an object as single byte, the `OlFont` field will be a valid pointer to an `XFontStruct`.

If the text format is `OL_SB_STR_REP` and a font set has been specified (using a comma-separated list), the first font in the list will be used to construct an `Xfont` structure.

Figure 4-2 illustrates `OlFont` overloading.

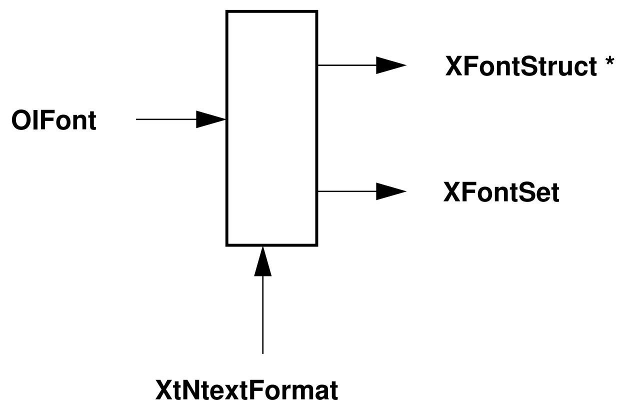

Figure 4-2 OlFont Overloading

OlStr

OLIT has introduced the data type `OlStr` to represent strings. The `OlStr` data type behaves similarly to the `OlFont` data type in the way `XtNtextFormat` is used to overload it:

- If the text format is `OL_SB_STR_REP`, then `OlStr` is interpreted as a string of single-byte characters.
- If the text format is `OL_MB_STR_REP`, then `OlStr` is interpreted as a string of multibyte characters.
- If the text format is `OL_WC_STR_REP`, then `OlStr` is interpreted as a string of wide characters.

Values for `OlStr` resources in resource files should always be specified in single-byte or multibyte format and never in wide character format. OLIT converts the fields to wide character format internally if the object's text format is `OL_WC_STR_REP`.

Figure 4-3 illustrates `OlStr` overloading.

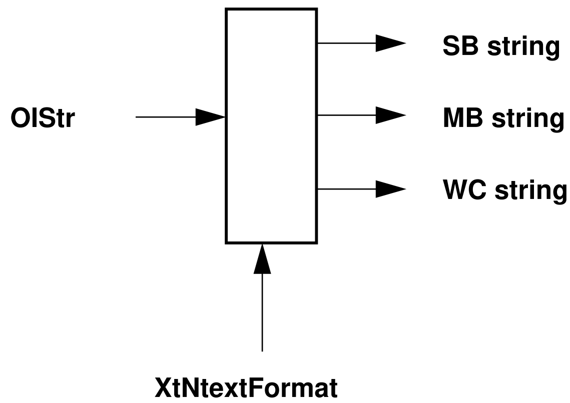

Figure 4-3 OlStr Overloading

Getting the Default Font or Font Set

The OLIT 3.1 widget set provides an `XtNolDefaultFont` resource which specifies the default font or font set for an application. This resource is discussed in the "Common Resources" section of the *OLIT 3.1 Reference Manual* and the on-line man pages.

If no font is specified for a widget's `XtNfont` resource, then `XtNolDefaultFont` determines the widget's font. In international OLIT, the default value of `XtNolDefaultFont` is determined as follows:

- In the C `locale`, the default value of `XtNolDefaultFont` is lucida sans serif with `Resolution_X` and `Resolution_Y` set to 75.
- In other locales, the default value of `XtNolDefaultFont` is a font set required to display all the characters in the code set of the locale.

OLIT provides a convenience routine, `OlGetDefaultFont()`, which enables you to get the font or font set that will be used if you do not set the `XtNfont` resource for a widget. The syntax of `OlGetDefaultFont()` is:

```
OlFont OlGetDefaultFont(widget w);
```

where `widget` is a widget in the application for which you want to get the default font.

Note that the returned value will not be the same as the value for `XtNolDefaultFont` in the following cases:

- If the XtNdefaultFont resource is set.
  OlGetDefaultFont() returns the XtNdefaultFont value instead of XtNolDefaultFont.
- If XtNolDefaultFont specifies a font which is not available, OlGetDefaultFont() returns a null font name.

Localized Text Handling

OLIT provides resources that enable you to set various text strings (messages, menu labels, button labels, etc.) in an application. For information on the text string resources for a particular widget, see the reference section for the widget in the *OLIT 3.1 Reference Manual*. When you create an internationalized application, you should remove all these resources from your application and keep them in locale-specific resource files.

Using Internationalized Help

You can register multibyte help text for a widget with OLIT's OlRegisterHelp() function. To register single-byte help, use the same mechanism. Single-byte is a subset of multibyte text; therefore, it works correctly and without overhead because help is display-only. The syntax for OlRegisterHelp() is as follows:

```
void OlRegisterHelp(
    OlDefine id_type,
    XtPointer id,
    String tag,
    OlDefine source_type,
    XtPointer source);
```

To use OlRegisterHelp() with international OLIT, specify one of the following values for the id_type argument:

id_type Value	Meaning
OL_WIDGET_HELP	Specifies multibyte format help text for an individual widget (for backward compatibility)
OL_FLAT_HELP	Specifies multibyte format help text for a flat widget (for backward compatibility)

If you specify OL_DISK_SOURCE for the source_type argument, you must specify a single-byte or multibyte filename for the source argument.

Help searches the directories specified by XFILESEARCHPATH for the specified filename. For a description of how XFILESEARCHPATH is used, see the description of the function XtResolvePathname in the *Xt Reference Manual*. If you set the filename from within your program as a full path, help ignores XFILESEARCHPATH.

For a description of the other arguments of OlRegisterHelp(), see the section entitled “Help Functions” in the *OLIT 3.1 Reference Manual* or the on-line man pages.

Locale Announcement

To announce locale, OLIT registers a private language procedure with the Intrinsics during OlToolkitInitialize(). For information on this interface, see XtLanguageProc in the *Xt Reference Manual*.

Do not register an application-specific language procedure. If you do, it will interfere with OLIT’s locale-announcement mechanism and you will have to take responsibility for it in your application. If your application requires processing before OLIT’s language procedure is called, you can provide your own procedure and call OLIT’s procedure from it. To get the language procedure registered by OLIT, use the call

```
    olit_proc = XtSetLanguageProc(NULL, NULL, NULL);
```

New Multibyte and Wide Character Text Buffer Utility Functions

International OLIT provides multibyte and wide character equivalents to the single-byte OLIT 3.1 text buffer functions. A list of these functions and their syntax is provided in Appendix C.

Input Method

The input method (IM) is the algorithm by which users enter the text of a language. The input method for each language may be different, depending on the linguistic structure and conventions of that language. OLIT interfaces with Xlib objects that form the top layer of this functionality. Internally, Xlib may communicate with another process, commonly known as the IM server, to do the actual work.

International OLIT follows the X Window System Version 11, Input Method Specification, Draft 3.0 [XIM90]. This specification was derived as a result of discussions among X Consortium members on standardizing the input handling of characters in various languages by X clients.

In English, end users enter the desired text by typing in a sequence of letters to create a word. However, for languages based on ideographic characters, input is more complicated. For example, there are two phonetic alphabets in Japanese—Hiragana and Katakana—and the traditional ideogrammatic alphabet, Kanji. In any piece of writing, all three alphabets may be used. Japanese words can also be spelled out phonetically in English; this is called Romaji.

To handle languages which do not have a one-to-one key to character mapping, input methods provide features such as the following:

- A control key sequence, which selects the *input mode.*
- A pre-edit region, which displays characters as the end user enters them but before the end user *commits* them.
- A lookup choice region, which displays ideographic characters and allows the user to choose one.
- A status region, which provides information such as whether conversion is activated and the state or mode of the input method.

The use of these features varies with the input method. For more information, see the documentation for the input method you are using. OpenWindows provides input methods for all locales it supports.

Figure 4-4 shows the input method screen regions.

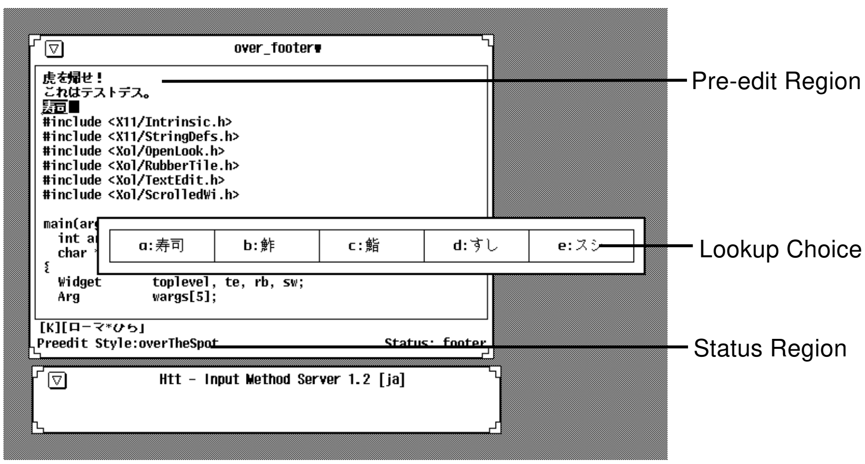

Figure 4-4 Input Method Screen Regions for "ja" Locale

Setting the Input Method Pre-edit and Status Styles

There are two aspects of the input method that an OLIT application can control:

- The *pre-edit style*, which specifies where and how pre-edit data is presented.
- The *status style*, which specifies where status feedback is presented.

OLIT provides two new resources that specify the pre-edit and status styles: `XtNimPreeditStyle` and `XtNimStatusStyle`.

XtNimPreeditStyle

The `TextEdit` resource, `XtNimPreeditStyle` (`XtCImPreeditStyle` class) selects the pre-edit style.

Resource	Type	Default	Access
XtNimPreeditStyle	XtROlImPreeditStyle	OL_NO_PREEDIT	GI

If the specified style is not supported by the input method, the ability to pre-edit is lost. The currently supported pre-edit styles are listed in the following table.

Table 4-1 Supported Pre-edit Styles

XtNimPreeditStyle Value	Meaning
OL_ON_THE_SPOT/"onTheSpot"	IM directs the application to display the pre-edit data.
OL_OVER_THE_SPOT/"overTheSpot"	IM displays pre-edit data in its own window over the application window.
OL_ROOT_WINDOW/"rootWindow"	IM displays pre-edit data outside the application in a window that is visually separate from the application.
OL_NO_PREEDIT/"none"	IM does not display pre-edit data.

Figure 4-5 shows an example of the `onTheSpot` pre-edit style. Note that the pre-edit data is shown in reverse video. When the user commits the data, it is sent to the client and displayed in normal video.

Figure 4-5 `onTheSpot` *Pre-edit Style for "ko" Locale*

Figure 4-6 shows an example of the `overTheSpot` pre-edit style. The difference between this style and `onTheSpot` is that `overTheSpot` obscures data in the application during pre-edit, while `onTheSpot` inserts data into the application without obscuring data.

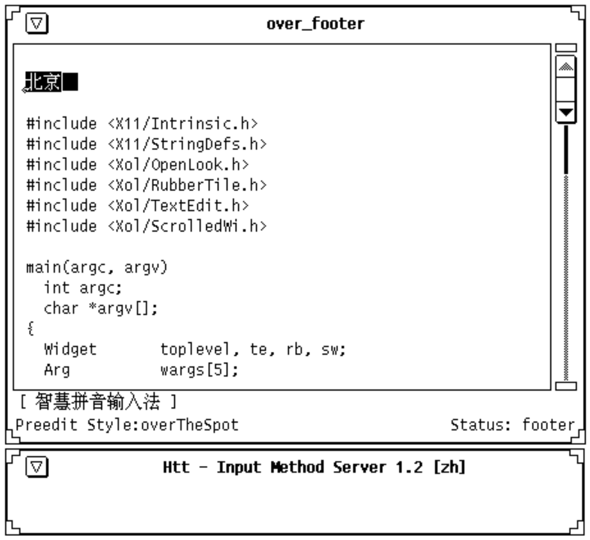

Figure 4-6 `overTheSpot` *Pre-edit Style for "zh" Locale*

Figure 4-7 shows an example of the `rootWindow` pre-edit style.

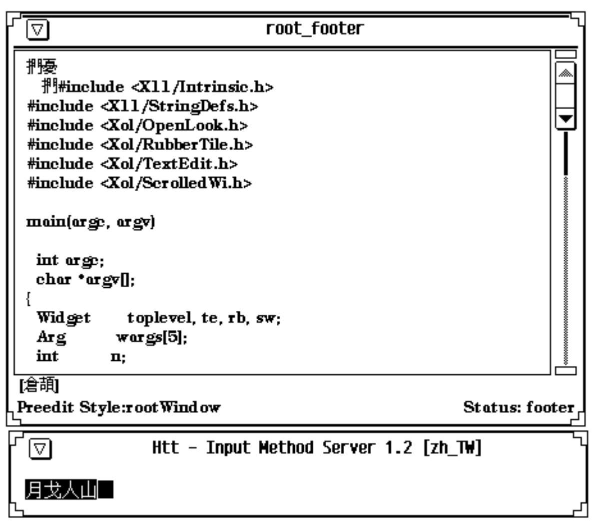

Figure 4-7 `rootWindow` *Preedit Style for "zh_TW" Locale*

See the X11 Release 5 documentation for a full description of each of the pre-edit styles.

XtNimStatusStyle

The application resource, `XtNimStatusStyle` (`XtCImStatusStyle` class), determines the style of IM status feedback.

Resource	Type	Default	Access
XtNimStatusStyle	XtROlImStatusStyle	OL_IM_STATUS_NONE	GI

The supported styles are:

XtNimStatusStyle Value	Meaning
OL_IM_STATUS_FOOTER/"footer"	IM generates status feedback in the footer of the shell window.
OL_IM_STATUS_ROOT/"rootWindow"	IM generates status feedback in a separate window visually separate from the application.
OL_IM_STATUS_NONE/"none"	IM doesn't generate status feedback.

When `XtNimStatusStyle` is set to `OL_IM_STATUS_FOOTER`, a number of other resources are available to set characteristics of the application footer. See "Application Resources for the IM Footer" below.

Figure 4-8 shows an example of the `footer` status style.

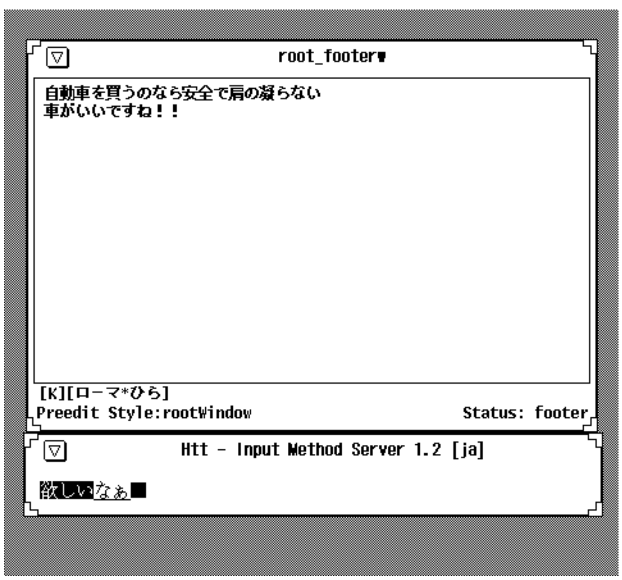

Figure 4-8 `footer` *Status Style for "ja" Locale*

Figure 4-9 shows an example of the rootWindow IM status style.

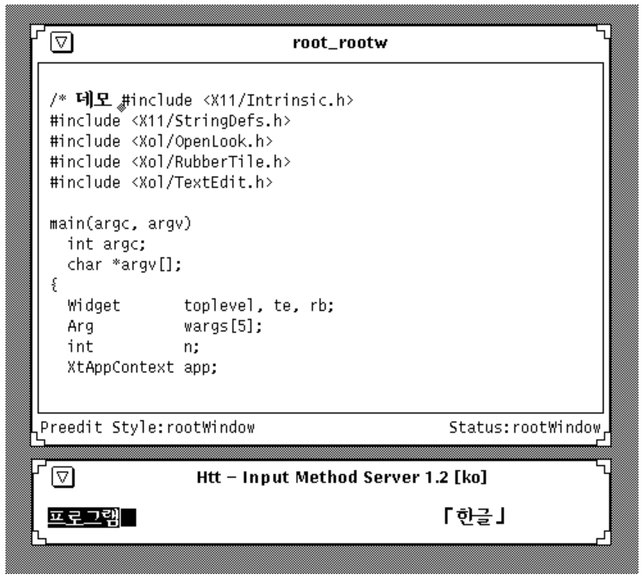

Figure 4-9 rootWindow *Status Style for "ko" Locale*

Application Resources for the IM Footer

Table 4-2 Summary of Footer Resources

Resource	Type	Default	Access
XtNimForeground	Pixel	XtDefaultForeground	SGI
XtNimBackground	Pixel	XtDefaultBackground	SGI
XtNimBackgroundPixmap	Pixmap	NULL	SGI
XtNimFontSet	OlFont	XtDefaultFont	SGI
XtNdefaultImName	String	NULL	SGI

- XtNimFontSet
  The XtNimFontSet resource specifies the font set to be used by the IM to display pre-edit and status feedback.

- XtNdefaultImName
  The XtNdefaultImName resource specifies the string to identify the IM Server. This is equivalent to specifying XMODIFIERS="@im=<string>" where you need to specify <string> in the resource.

OLIT Internationalization Example

To internationalize a simple OLIT application:

- Remove any resources that contain display text from the application code and put them in a resource file.
- Specify the text format for widgets that display text (by specifying a default text format for the application or by specifying the text format for individual widgets).
- Specify a font set for resources that contain `OlFont` values (such as `XtNfont`) in the resource file.

Suppose you want to write an internationalized application that displays a `StaticText` widget with some wide character text in it. To do this, you might write code similar to the following:

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/StaticText.h>

main(argc, argv)
    int   argc;
    char *argv[];
{
    Widget toplevel, msg_widget;
    XtAppContext app;

    /* Initialize the OLIT toolkit */
    OlToolkitInitialize((XtPointer)NULL);

    /* Set the default text format to wide character */
    OlSetDefaultTextFormat(OL_WC_STR_REP);

    toplevel = XtAppInitialize(&app, "Memo",
                    (XrmOptionDescList)NULL,
                    0, &argc, argv, NULL,
                    (ArgList) NULL, 0);

    /* Create a staticText widget. */
    msg_widget = XtVaCreateManagedWidget("msg",
                        staticTextWidgetClass,
                        toplevel,
                        NULL);

  /* Realize the widgets and enter the event loop. */
  XtRealizeWidget(toplevel);
  XtAppMainLoop(app);
}
```

This application does not specify the string that appears in the `StaticText` widget. You must specify this text in the resource file, as follows:

```
*StaticText.string: <text>
```

Notice that this application differs from a conventional OLIT application; it calls `OlSetDefaultTextFormat()` to set the default text format. `OlSetDefaultTextFormat()` sets the default text format for any widgets in the application that display text.

To set the text format for an individual widget, set its `XtNtextFormat` resource. For example, in the previous application, create the `StaticText` widget as follows:

```
msg_widget = XtVaCreateManagedWidget("msg",
                        staticTextWidgetClass,
                        toplevel,
                        XtNtextFormat, OL_WC_STR_REP,
                        NULL);
```

For some locales, you may also need to specify a font set for the application. To do this, add the following line to the resources file:

```
    *StaticText.font:   <font set>
```

Figure 4-10 shows the application with Korean text. Figure 4-11 shows the application with Japanese text.

Figure 4-10 Hello World in Korean

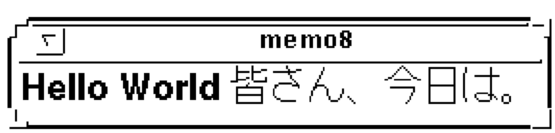

Figure 4-11 Hello World in Japanese

Standards

Internationalized OLIT is based on three standards: Internationalized UNIX, MIT X11 Internationalization Standard, and Internationalized Extension of OPEN LOOK Specification.

For more information on support of internationalization in the toolkit, see the “Internationalization Tutorial” and “Text Buffer Utility Functions for Internationalization in OLIT” sections of Appendix C.

Unified Desktop Look And Feel

With the release of OLIT 3.2, all OLIT objects that require rendering use OLGX . This will make OLIT objects look more like XView objects, such as the DeskSet applications. For most developers, this transition will be automatic. Developers will benefit from a more consistent look for their applications on the the OPEN LOOK desktop. However, if you are subclassing widgets and want to do custom rendering, you will need knowledge of the OLGX API and how to migrate your code from OLG to OLGX or write new rendering code. A white paper addressing this topic will soon be available through public distribution channels. Look for this paper to learn more about using OLGX to develop applications with custom rendering.

ScrollingList Widget (New Features)

Developers have lamented the lack of support from the OLIT toolkit in managing the `ScrollingList` widget. In Solaris 2.x, OLIT has an enhanced API for this widget. Resources and convenience routines have been added to help you manage the `ScrollingList` widget. What follows is an overview of the widget and a sample program demonstrating the associated changes.

The `ScrollingList` widget can now be created in one of the following modes:

`OL_EXCLUSIVE`	Lets you choose one item from a list of items; at any given time one item is current or chosen.
`OL_EXCLUSIVE_NONESET`	Variation of the “exclusive” scrollinglist. Here you can choose one or none of the items on the list.
`OL_NONEXCLUSIVE`	Lets you choose none, one, or multiple items from the list.
`OL_NONE`	None of the preceding three modes. For backward compatibility. This is the *default* mode for the scrollinglist. In this mode, you may *not* use any of the new functions or resources that follow.

Other features include making an item insensitive, double-clicking, item list management now done by widget, and resources and convenience functions.

To avoid confusion, all the new resources and convienence functions use the new data type `OlSlistItemPtr` as the scrollinglist item, as opposed to `OlListToken`.

Note – Do **not** mix the old and new API. The new API is sufficient in the sense that it provides equivalent routines for the old ones. For example, you cannot use the old API method of doing an `XtGetValues()` of the `XtNapplAddItem` resource and then use new convenience functions. Instead, you must use `OlSlistAddItem()` specifying the `OlSlistItemAttrs` structure, as shown in the following section.

New Item Attribute Structure

There now exists a new item data structure, `OlSlistItemAttrs`, which is used in the `OlSlistAddItem()` function to indicate the attributes of interest for a scrollinglist item. Its source code is shown in Code Example 4-1.

Code Example 4-1 OlSlistAddItem Attributes

```
#define OlItemLabelType (1L << 0)
#define OlItemLabel (1L << 1)
#define OlItemImage (1L << 2)
#define OlItemSensitive (1L << 3)
#define OlItemCurrent (1L << 4)
#define OlItemMnemonic (1L << 5)
#define OlItemUserData (1L << 6)
```

The following structure exists for each `scrollinglist` item:

```
typedef struct {
        long    flags; /* indicates which fields in this
                          struct are valid */
        OlDefine  label_type;
        XtPointer item_label;
        XImage    *item_image;
        Boolean   item_sensitive;
        Boolean   item_current;
        unsigned char item_mnemonic;
        XtPointer user_data;
} OlSlistItemAttrs, *OlSlistItemAttrsPtr;
```

The following table describes the attributes of the `OlSlistAddItem` structure:

Attribute	Function
`flags`	Marks which fields in this structure are valid or to be used.
`label_type`	Identifies the type of label to display for the item. It can have one of the following values: `OL_STRING`, `OL_IMAGE`, or `OL_BOTH` (for string and image).
`item_label`	The text label for this item

Attribute	Function
item_image	The glyph associated with this item.
item_sensitive	Indicates whether item active (receive events) or inactive.
item_current	Indicates whether the item is current (chosen).
item_mnemonic	A single character that is used as a mnemonic accelerator.
user_data	For application use

New Callback Structures for ScrollingLists

The following callbacks are new to OLIT and are used with the `ScrollingList` widget:

Code Example 4-2 OlSlistCallbackStruct

```
typedef struct {
 int reason;
 XEvent *event;
 OlDefine mode;
 OlSlistItemPtr item;
 int item_pos;
 OlSlistItemPtr *cur_items;
 int num_cur_items;
 int *cur_items_pos; /* not implemented */
} OlSlistCallbackStruct;
```

Code Example 4-3 OlSlistUserDeleteCallbackStruct

```
typdef struct {
 int reason;
 XEvent *event;
 OlDefine mode;
 OlSlistItemPtr *user_del_items;
 int num_del_items;
} OlSlistUserDeleteCallbackStruct;
```

The following table describes the parameters of these structures:

Parameter	Definition
reason	Indicates the reason for the callback.
mode	Indicates what mode the scrollinglist is in. Can be `OL_EXCLUSIVES`, `OL_EXCLUSIVES_NONESET`, or `OL_NONEXCLUSIVES`.
event	Points to the XEvent that triggered the callback. Can be NULL.
item	Last current item at the time of the event that triggered the callback. This points to temporary storage that the application must copy into its own space as needed.
item_pos	Specifies the position of the item in the list.

Parameter	Definition
cur_items	Specifies a list of current items at the time of the event that triggered the callback. When the mode is OL_NONEXCLUSIVE, this gives a list of all the current items.
OL_EXLCUSIVE and OL_EXCLUSIVE_NONESET	This variable Has the same value as the field 'item.' This points to temporary storage that the application must copy into its own space if needed.
num_cur_items	Gives the number of current items in the list.
cur_items_pos	An array of integers, representing one for each current item. This points to temporary storage that the application must copy into its own space if needed.

Changes in Existing Resources for ScrollingList Widget

The following change has been implemented in existing resources to support the `ScrollingList` widget:

- `XtNuserDeleteItems`

Type:	`XtCallbackList`
Default:	`NULL`
Access:	SGIO
Action:	Defines the callbacks issued when the end user tries to delete items from the list. Currently, the `ScrollingList` widget handles deletions through a `cut` operation. The `call_data` parameter in `OL_NONE` mode is `OlListDelete`, as documented in the *OLIT Reference Manual*; for all other modes it is `OlSlistUserDelCallbackStruct`, and the reason field is set to `OL_REASON_USER_DELETE_ITEMS`. It is currently up to the application to act on each item by calling `OlSlistDeleteItem` or `OlSlistDeleteItems` and providing necessary feedback.

New Resources for ScrollingLists

The following X resources have been added to the OLIT toolkit:

- `XtNscrollingListMode`

Type:	`Oldefine`
Default:	`OL_NONE`
Access:	GI
Action:	Defines the scrollinglist's mode—exclusive or non-exclusive.

Values:

`OL_EXCLUSIVE`	Exclusive mode—one item current at all times.
`OL_EXCLUSIVE_NONESET`	Exclusive mode—one item or none current at all times.
`OL_NONEXCLUSIVE`	Non-exclusive mode—one or many current.
`OL_NONE`	Nothing—for backward compatibility.

- `XtNitemCurrentCallback`

Type:	`XtCallbackList`
Default:	`NULL`
Access:	SGIO
Action:	Defines the callbacks to be invoked when an item is made current by an end user pressing the mouse button to select an item when the mode is anything other than `OL_NONE`, the reason field in `OlSlistCallbackStruct` is set to `OL_REASON_ITEM_CURRENT`. If the mode is `OL_NONE` (default value), then only `XtNuserMakeCurrent` callbacks will be called.

- `XtNmultiClickCallback`

Type:	`XtCallbackList`
Default:	`NULL`
Access:	SGIO
Action:	Defines the callbacks to be invoked when the SELECT mouse button is pressed on an item multiple times. The reason field in `OlSlistCallbackStruct` is set to `OL_REASON_DOUBLE_CLICK`.

The time span for a button click followed by another in milliseconds is determined by the `XtNmultiClickTimeout` resource. This activates `XtNitemCurrentCallback` callbacks.

- `XtNfirstViewableItem`

Type:	`OlSlistItemPtr`
Default:	calculated
Access:	SG
Action:	Specifies which item is the first viewable item in the list. The default is the first item added to the list.

- `XtNlastViewableItem`

Type:	`OlSlistItemPtr`
Default:	calculated
Access:	SG
Action:	Specifies which item is the last viewable item in the list.

- `XtNviewableItems`

Type:	`OlSlistItemPtr *`
Default:	`NULL`
Access:	G
Action:	Gives the viewable list of items in the widget. The number of items in this list is determined by the resource `XtNviewHeight`. If the widget is empty, `XtNviewableItems` it returns `NULL`.

- `XtNitems`

Type:	`OlSlistItemPtr *`
Default:	`NULL`
Access:	G
Action:	Gives the entire list of items in the widget. The number of items in this list is determined by the resource `XtNnumItems`. If the widget is empty `XtNitems` returns NULL.

- `XtNnumItems`

Type:	`int`
Default:	`0` (zero)
Access:	G
Action:	Defines the total number of items in the list at any given time. It is automatically updated by the `scrollinglist` whenever an item is added or deleted.

- XtNcurrentItems

Type: OlSlistItemPtr *

Default: NULL

Access: G

Action: Points to an array of OlSlistItemPtr that are to be set current in the list. XtGetValues for this resource returns the items that are currently current and the application must not free these items.

- XtNnumCurrentItems

Type: int

Default: NULL

Access: G

Action: Specifies the number of items current. The value must be the number of items in XtNcurrentItems and must not be negative.

- XtNitemNotCurrentCallback

Type: XtCallbackList

Default: NULL

Access: SGIO

Action: Defines the list of callbacks to be invoked when an end user presses the SELECT mouse button on a current item. The reason field in OlSlistCallbackStruct is set to OL_REASON_ITEM_NOT_CURRENT.

New Convenience Functions

The following convenience functions have been added to OLIT:

```
void OlSlistTouchItem(Widget widget, OlSlistItemPtr item)
```

A function that the application should use when it changes the attributes of an item.

```
void OlSlistUpdateView(Widget widget, Boolean ok)
```

A function that the application can call to keep the `ScrollingList` widget from updating the view or to let it again update the view. If `ok` is `TRUE`, then the `ScrollingList` widget can update the view as it changes; if `ok` is `FALSE`, then no update.

```
OlSlistItemPtr OlSlistAddItem(Widget widget, OlSlistItemAttrsPtr item_attr,
OlSlistItemPtr ref_item)
```

A function that adds an item before `ref_item`, given the attributes of the item and returns a pointer to the item. The value of `ref_item` can be zero to append the new item to the bottom of the list.

```
void OlSlistDeleteAllItems(Widget widget)
```

A function that deletes all items from the scrollinglist.

```
void OlSlistDeleteItem(Widget widget, OlSlistItemPtr item)
```

A function that deletes an item from the scrollinglist.

```
void OlSlistDeleteItems(Widget widget, OlSlistItemPtr *items, int num_items)
```

A function that deletes items from the scrollinglist.

```
void OlSlistMakeAllItemsNotCurrent(Widget widget)
```

A function that makes all items not current.

```
void OlSlistMakeItemNotCurrent(Widget widget, OlSlistItemPtr item, Boolean
issue_callback)
```

A function that makes an item not current in the scrollinglist and also removes it from the current list; issues an `XtNitemNotCurrentCallback` if `issue_callback` is `TRUE`.

```
Boolean OlSlistIsValidItem(Widget widget OlSlistItemPtr item)
```

A function that returns `TRUE` if the specified item is a valid item in the list, else it returns `FALSE`.

```
void OlSlistMakeItemCurrent(Widget widget, OlSlistItemPtr item, Boolean
issue_callback)
```

A function that makes an item/row current and adds it to the list of current items and issues a current callback if the boolean value `issue_callback` is `TRUE`.

```
void OlSlistFirstViewableItem(Widget widget, OlSlistItemPtr item)
```

A function that sets the first viewable item in the list.

```
void OlSlistLastViewableItem(Widget widget, OlSlistItemPtr item)
```

A function that sets the last viewable item in the list.

```
OlSlistItemPtr OlSlistGetNextItem(Widget widget, OlSlistItemPtr item)
```

A function that returns the next item, given a valid `item`. If `item` is the last item in the scrollinglist, then it returns `NULL`.

```
OlSlistItemPtr OlSlistGetPrevItem(Widget widget, OlSlistItemPtr item)
```

A function that returns the previous item, given a valid `item`. If `item` is the first item in the scrollinglist then it returns `NULL`.

```
OlStr OlSlistGetItemLabel(Widget widget, OlSlistItemPtr item)
```

A function that returns the item label given the item; returns `NULL` if no label is present.

```
XImage * OlSlistGetItemImage(Widget widget, OlSlistItemPtr item)
```

A function that returns the item image/glyph given the item; returns `NULL` if no image is present.

```
OlDefine OlSlistGetItemType(Widget widget, OlSlistItemPtr item)
```

A function that returns the item type given the item/row. Valid values are:

OL_STRING	if item just has a string.
OL_IMAGE	if item just has a image/glyph.
OL_BOTH	if item has OL_STRING and OL_IMAGE.
OL_NONE	if item is not a valid item in the list.

A function that returns what mode the scrollinglist is in, such as OL_EXCLUSIVE, if the widget

```
OlDefine OlSlistGetMode(Widget widget)
```

is in exclusive mode, or OL_NONEXCLUSIVE for nonexclusive mode.

```
Boolean OlSlistIsItemCurrent(Widget widget, OlSlistItemPtr item)
```

A function that returns TRUE if the item is current, else FALSE.

```
Boolean OlSlistGetItemSensitivity(Widget widget, OlSlistItemPtr item)
```

A function that returns TRUE if the item is sensitive, else FALSE.

```
XtPointer OlSlistGetItemUserData(Widget widget, OlSlistItemPtr item)
```

A function that returns the item's user data (if any).

```
void OlSlistSetItemAttrs(Widget widget, OlSlistItemPtr item, OlSlistItemAttrs*
set_attrs)
```

Set the attributes pointed to in set_attrs on the specified item.

```
void OlSlistGetItemAttrs(Widget widget, OlSlistItemPtr item, OlSlistItemAttrs*
get_attrs);
```

This function returns the 'item' attributes in get_attrs.

```
void OlSlistEditItem(Widget widget, Boolean insert, OlSlistItemPtr ref_item)
```

This function allows programmatic editing of a scrollinglist item. If `insert` is TRUE then `ref_item` identifies the item before where a new item is to be inserted. If `insert` is TRUE then `ref_item` can be null, in which case a new item is added to the end of the list. If `insert` is FALSE then `ref_item` refers to the item that should be edited and must be a valid item in the list.

```
void OlSlistEndEdit(Widget widget)
```

A function that the application should call when the end user has finished editing an item.

The following program uses some of the widgets described in this section:

Code Example 4-4   Sample Program Using ScrollingList Widget

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/RubberTile.h>
#include <Xol/ScrollingL.h>
#include <Xol/ControlAre.h>
#include <Xol/OblongButt.h>
#include <dirent.h>

main(int argc, char **argv)
{
  XtAppContext app;
  Widget      toplevel, rt, sl, ca, delete, findcurrent;
  String      directory;
  void        initList(Widget, String);
  void      current_callback(Widget, XtPointer, XtPointer);

  (void) OlToolkitInitialize((XtPointer)NULL);
  toplevel = XtAppInitialize(&app, "Scrollinglist",
        (XrmOptionDescList)NULL, 0,
        (Cardinal *)&argc, argv,
        (const char **)NULL,
        (ArgList)NULL,
        0);

  /*
   * Get the command line argument, if any
   */
  directory = argv[1];
  rt = XtVaCreateManagedWidget("rt", rubberTileWidgetClass,
              toplevel,
```

Code Example 4-4 Sample Program Using ScrollingList Widget (Continued)

```
        NULL);
  /*
   * Create the ScrollingList widget
   */
  sl = XtVaCreateManagedWidget("sl", scrollingListWidgetClass,
                 rt,
        XtNviewHeight, 6,
        XtNscrollingListMode, OL_EXCLUSIVE,
        NULL);
  XtAddCallback(sl, XtNitemCurrentCallback, current_callback, NULL);

  XtRealizeWidget(toplevel);
  initList(sl, directory);

  XtAppMainLoop(app);
}

void
initList(Widget sl, String directory)
{
  OlSlistItemAttrs     item;
  DIR           *dirp;
  struct dirent *dp;

  if(directory == NULL)
    directory = ".";
  dirp = opendir(directory);
  if(dirp == NULL) {
    perror(directory);
    exit(-1);
  }
  OlSlistUpdateView(sl, False);
  for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp)) {
    item.flags = OlItemLabelType | OlItemLabel | OlItemMnemonic;
    item.label_type = OL_STRING;
    item.item_label = XtNewString(dp->d_name);
    item.item_mnemonic = NULL;
    OlSlistAddItem(sl, &item, 0);
  }
  closedir(dirp);
  OlSlistUpdateView(sl, True);
}

void
current_callback(Widget w, XtPointer client_data, XtPointer call_data)
```

Code Example 4-4 Sample Program Using ScrollingList Widget (Continued)

```
{
  OlSlistCallbackStruct *token = (OlSlistCallbackStruct *)call_data;
  printf("String selected is %s \n", OlSlistGetItemLabel(w, token->item));
}
```

TextLine Widget

OLIT 3.2 introduces the `TextLine` widget, a light-weight one-line text input widget. The TextLine widget obsoletes the `TextField` widget, providing similar functionality with much less memory consumption. The `TextField` widget remains in the toolkit for compatibility reasons, but its use is discouraged.

The `TextLine` widget is a one-line input field for text data. Once the input focus is moved into the widget, keyboard entry is allowed. If the input value exceeds the length of the input field, the ScrollButtons will appear. Hidden text can then be scrolled into view by pressing the ScrollButtons. Pressing the buttons continuously scrolls the text repeatedly.

The following shows a representation of the `TextLine` widget.

Figure 4-12 TextLine Widget

TextLine Widget Visual Components

The `TextLine` widget has the following visual components:

- right-justified bold label followed by a colon, at the left of the text line
- one point (for Monochrome or 2D) or chiseled underline
- ScrollButtons

For more detail on the programmer's interface for the `TextLine` widget, please consult the man page for the `TextLine` widget in Appendix C "OpenWindows Source Code" in this book.

The following is a sample program using the `TextLine` widget:

Code Example 4-5 TextLine Widget Sample Code

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/ControlAre.h>
```

Code Example 4-5 TextLine Widget Sample Code (Continued)

```
#include <Xol/TextLine.h>
main(int argc, char **argv)
{
    XtAppContext app;
    Widget toplevel, control, tl, tl1;
    void postmodify_cb(Widget, XtPointer, XtPointer);
    void commit_cb(Widget, XtPointer, XtPointer);

    (void) OlToolkitInitialize((XtPointer)NULL);
    toplevel = XtAppInitialize(&app, "Scrollinglist",
        (XrmOptionDescList)NULL, 0,
        (Cardinal *)&argc, argv,
        (const char **)NULL,
        (ArgList)NULL,
        0);

    control = XtVaCreateManagedWidget("control",
        controlAreaWidgetClass,
        toplevel,
        NULL);

    tl = XtVaCreateManagedWidget("textline",
        textLineWidgetClass,
        control,
        XtNcaptionLabel, "GetResults: ",
        XtNcharsVisible, 15,
        XtNmaximumChars, 80,
        XtNstring, "Type something here",
        NULL);

    tl1 = XtVaCreateManagedWidget("textline1",
        textLineWidgetClass,
        control,
        XtNcaptionLabel, "PutResults: ",
        XtNcharsVisible, 15,
        XtNmaximumChars, 80,
        XtNstring, "TextLine Widget Test",
        NULL);

    XtAddCallback(tl, XtNpostModifyCallback, postmodify_cb, (XtPointer) tl1);
    XtAddCallback(tl, XtNcommitCallback, commit_cb, (XtPointer) tl1);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(app);
}
```

Code Example 4-5 TextLine Widget Sample Code (Continued)

```
void
postmodify_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
    char buff[80];
    OlTLPostModifyCallbackStruct *postmodify_cdata =
        (OlTLPostModifyCallbackStruct *)call_data;
    Widget tl = (Widget) client_data;

    if(postmodify_cdata->reason == OL_REASON_POST_MODIFICATION)
    {
        sprintf(buff, "OL_REASON_POST_MODIFICATION: %s",
            postmodify_cdata->buffer);
        XtVaSetValues(tl, XtNstring, buff, NULL);
    }
    else if(postmodify_cdata->reason == OL_REASON_PROG_POST_MODIFICATION)
    {
        sprintf(buff, "OL_REASON__PROG_POST_MODIFICATION: %s",
            postmodify_cdata->buffer);
        XtVaSetValues(tl, XtNstring, buff, NULL);
    }
}

void
commit_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
    char buff[80];
    OlTLCommitCallbackStruct *commit_cdata =
        (OlTLCommitCallbackStruct *)call_data;
    Widget tl = (Widget) client_data;

    sprintf(buff, "OL_REASON_COMMIT: %s", commit_cdata->buffer);
    XtVaSetValues(tl, XtNstring, buff, NULL);
}
```

NumericField Widget

OLIT 3.2 introduces a specialized `TextLine` widget called the `NumericField` widget. This widget allows applications to have a one-line GUI object that controls numeric values. The widget provides delta buttons that increment and decrement the value as specified by the application.

Figure 4-13 contains a representation of the `NumericField` widget.

Figure 4-13 NumericField Widget

NumericField Widget Description

The `NumericField` widget is a one-line input field for numeric text. Once the input-focus is moved into the widget, keyboard entry is allowed. If the input value exceeds the length of the input field, ScrollButtons will appear.

Hidden text can then be scrolled into view by pressing the ScrollButtons. Pressing the buttons continuously scrolls the text repeatedly with a user-adjustable delay.

DeltaButtons can be used to increment or decrement the value in an appropriate manner.

Visual Components

The `NumericField` widget contains the following visual components:
- right-justified bold label followed by a colon at the left of the TextLine
- optional one point (for Monochrome or 2D) or chiseled underline
- ScrollButtons
- Increment and Decrement buttons, otherwise known as DeltaButtons

For more detail on the programmer's interface for the `NumericField` widget, please consult the man page for the `NumericField` widget in Appendix C "OpenWindows Source Code" in this book.

The following is a sample source file using the `NumericField` widget:

Code Example 4-6 NumericField Widget Sample Source

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/ControlAre.h>
#include <Xol/NumericFie.h>
main(int argc, char **argv)
{
XtAppContext app;
Widget toplevel, control, tl;
void delta_cb(Widget, XtPointer, XtPointer);
int intVal, minVal, maxVal, delVal;
```

Code Example 4-6 NumericField Widget Sample Source (Continued)

```
(void) OlToolkitInitialize((XtPointer)NULL);
toplevel = XtAppInitialize(&app, "Scrollinglist",
(XrmOptionDescList)NULL, 0,
(Cardinal *)&argc, argv,
(const char **)NULL,
(ArgList)NULL,
0);
control = XtVaCreateManagedWidget("control",
controlAreaWidgetClass,
toplevel,
NULL);
intVal = 0;
minVal = -100;
maxVal = 100;
delVal = 2;
tl = XtVaCreateManagedWidget("numericfield",
numericFieldWidgetClass,
control,
XtNcaptionLabel, "NumericField: ",
XtNvalue, &intVal,
XtNdelta, &delVal,
XtNmaxValue, &maxVal,
XtNminValue, &minVal,
NULL);
XtAddCallback(tl, XtNdeltaCallback, delta_cb, NULL);
XtRealizeWidget(toplevel);
XtAppMainLoop(app);
}
void
delta_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
OlNFDeltaCallbackStruct *nf_cdata = (OlNFDeltaCallbackStruct *)call_data;
if(nf_cdata->reason == OL_REASON_INCREMENT)
printf("user incremented. new value is: %d\n", (int)(*nf_cdata->value));
else
printf("user decremented. new value is: %d\n", (int)(*nf_cdata->value));
}
```

FileChooser Widget

The `FileChooser` widget is a new widget introduced in the OLIT 3.2. This widget provides an easy interface to the UNIX file systems for end users. The benefit of this functionality inside the toolkit is that now all applications can provide a uniform GUI interface to file systems that

OPEN LOOK users are accustomed to. The `FileChooser` widget has a standard API that is easy to incorporate and yet extensible to provide flexibility to developers. This section contains a technical outline of the widget, along with a small sample program.

The following is a representation of the `FileChooser` widget:

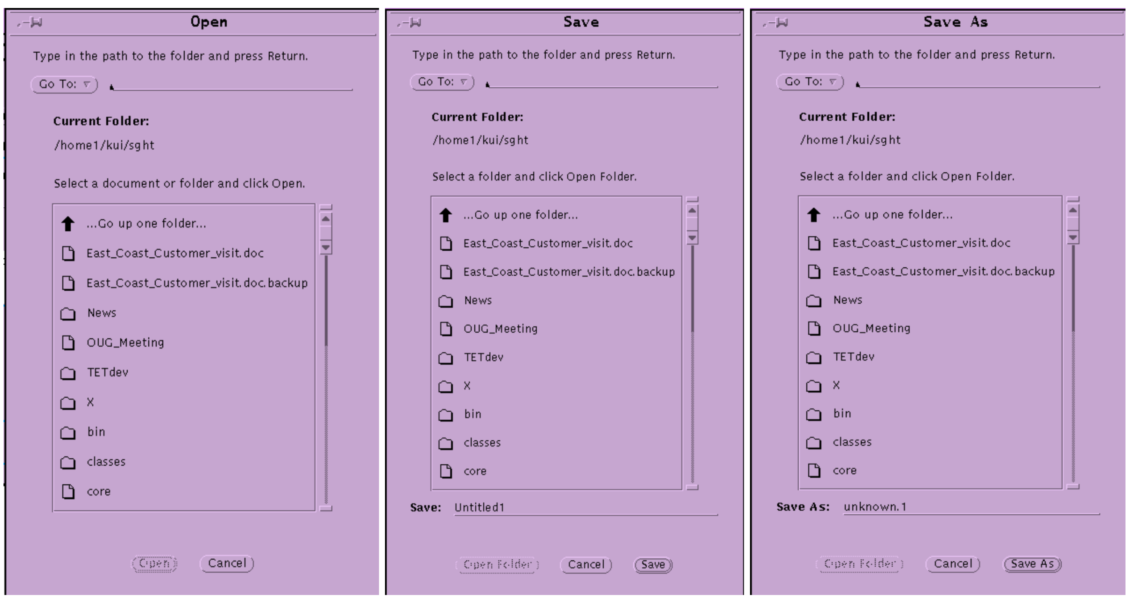

Figure 4-14 FileChooser Widget

For more detail on the programmer's interface for the `FileChooser` widget, please consult the man page for the `FileChooser` widget in Appendix C "OpenWindows Source Code" in this book.

The following is a sample source file using the `FileChooser` widget:

Code Example 4-7 FileChooser Widget Sample Source

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/ControlAre.h>
#include <Xol/OblongButt.h>
#include <Xol/FileChSh.h>
```

Code Example 4-7 FileChooser Widget Sample Source (Continued)

```
#include <Xol/MenuButton.h>

main(int argc, char **argv)
{
 XtAppContext app;
 Widgettoplevel, ca, menupane, ofilesh, sfilesh, safilesh, dumb;
 void   openfile_cb(Widget, XtPointer, XtPointer);
 void   filechpopup_cb(Widget, XtPointer, XtPointer);

 (void) OlToolkitInitialize((XtPointer)NULL);
 toplevel = XtAppInitialize(&app, "FileChooser",
        (XrmOptionDescList)NULL, 0,
        (Cardinal *)&argc, argv,
        (const char **)NULL,
        (ArgList)NULL,
        0);

 ca = XtVaCreateManagedWidget("controlarea", controlAreaWidgetClass,
        toplevel,
        NULL);

 ofilesh = XtVaCreatePopupShell("openfilechoosershell",
        fileChooserShellWidgetClass,ca,
        XtNoperation,OL_OPEN,
        NULL);

 XtVaGetValues(ofilesh,
        XtNfileChooserWidget,(XtArgVal)&dumb,
        NULL);

 XtAddCallback(dumb, XtNopenDocumentCallback, openfile_cb, NULL);

 sfilesh = XtVaCreatePopupShell("savefilechoosershell",
        fileChooserShellWidgetClass,ca,
        XtNoperation,OL_SAVE,
        NULL);

 safilesh = XtVaCreatePopupShell("saveasfilechoosershell",
        fileChooserShellWidgetClass,ca,
        XtNoperation,OL_SAVE_AS,
        NULL);

 dumb = XtVaCreateManagedWidget("menubutton",
        menuButtonWidgetClass,ca,
        XtNpushpin,(XtArgVal)OL_OUT,
```

Code Example 4-7 FileChooser Widget Sample Source (Continued)

```
        NULL);

 XtVaGetValues(dumb,
        XtNmenuPane,(XtArgVal)&menupane,
        NULL);

 dumb = XtVaCreateManagedWidget("Open...",
        oblongButtonWidgetClass,menupane,
        NULL);

 XtAddCallback(dumb, XtNselect, filechpopup_cb, ofilesh);

 dumb = XtVaCreateManagedWidget("Save...",
     oblongButtonWidgetClass, menupane,
     NULL);

 XtAddCallback(dumb, XtNselect, filechpopup_cb, sfilesh);

 dumb = XtVaCreateManagedWidget("SaveAs...",
     oblongButtonWidgetClass, menupane,
     NULL);

 XtAddCallback(dumb, XtNselect, filechpopup_cb, safilesh);

 XtRealizeWidget(toplevel);
 XtAppMainLoop(app);
}

void
openfile_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
 OlAnyCallbackStruct *any_cd = (OlAnyCallbackStruct *)call_data;

 if (any_cd->reason == OL_REASON_OPEN_DOCUMENT)
 {
 OlFileChDocumentCallbackStruct *cd =
           (OlFileChDocumentCallbackStruct *)call_data;
 }
}

void
filechpopup_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
 XtPopup((Widget)client_data, XtGrabNone);
}
```

FontChooser Widget

The `FontChooser` widget is a new widget introduced in OLIT 3.2. This widget allows you to communicate with an application regarding a font choice. Because the widget is inside the toolkit, all OLIT based applications will be able to popup a font chooser that is OPEN LOOK compliant and presents a consistent and familiar GUI. This section discusses the widget and is followed by a sample program.

Figure 4-15 contains a representation of the `FontChooser` widget:

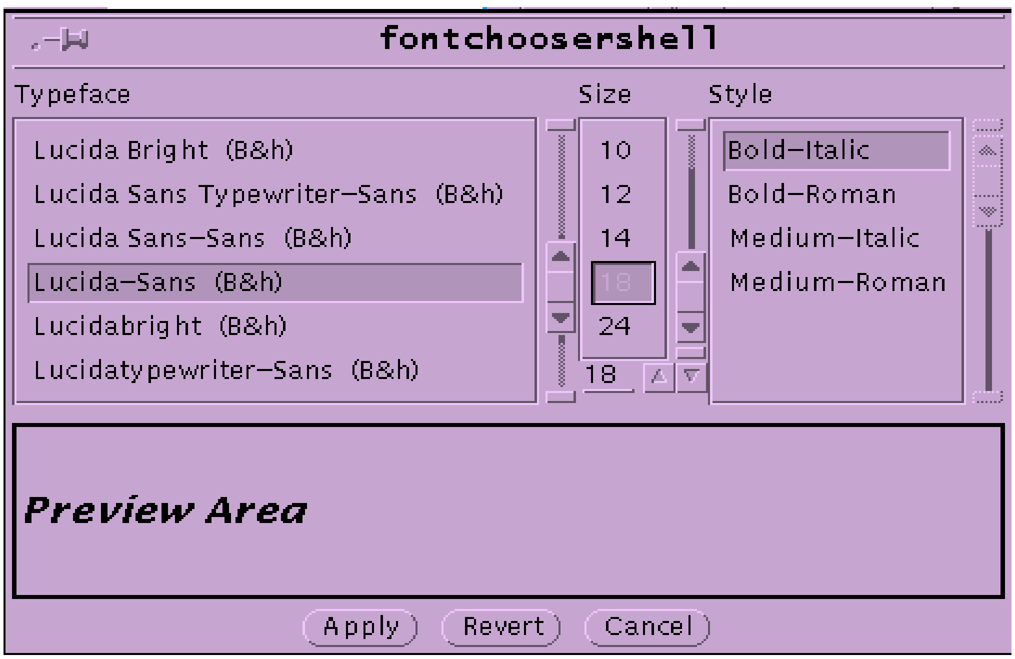

Figure 4-15 FontChooser Widget

For more detail on the programmer's interface for the `FontChooser` widget, please consult the man page for the `FontChooser` widget in Appendix C "OpenWindows Source Code", in this book.

The following is a sample source file using the `FontChooser` widget.

Code Example 4-8 FontChooser Widget Sample Source

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/ControlAre.h>
#include <Xol/OblongButt.h>
#include <Xol/FontChSh.h>

main(int argc, char **argv)
{
 XtAppContext app;
 Widget toplevel, ca, obbutt, tmpwidget, fontchshell;
 void fontchooser_cb(Widget, XtPointer, XtPointer);
 void fontapply_cb(Widget, XtPointer, XtPointer);

 (void) OlToolkitInitialize((XtPointer)NULL);
 toplevel = XtAppInitialize(&app, "FileChooser",
        (XrmOptionDescList)NULL, 0,
        (Cardinal *)&argc, argv,
        (const char **)NULL,
        (ArgList)NULL,
        0);

 ca = XtVaCreateManagedWidget("controlarea", controlAreaWidgetClass,
        toplevel,
        NULL);

 obbutt = XtVaCreateManagedWidget("fontbutton", oblongButtonWidgetClass,
        ca,
        NULL);

 fontchshell = XtVaCreatePopupShell("fontchoosershell",
        fontChooserShellWidgetClass,
        obbutt,
        NULL);

 XtAddCallback(obbutt, XtNselect, fontchooser_cb, (XtPointer)fontchshell);

 XtVaGetValues(fontchshell,
        XtNfontChooserWidget,(XtArgVal)&tmpwidget,
        NULL);

 XtAddCallback(tmpwidget, XtNapplyCallback, fontapply_cb, NULL);
```

Code Example 4-8 FontChooser Widget Sample Source (Continued)

```
 XtRealizeWidget(toplevel);
 XtAppMainLoop(app);
}

void
fontchooser_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
    XtPopup((Widget)client_data, XtGrabNone);
}

void
fontapply_cb(Widget w, XtPointer client_data, XtPointer call_data)
{
    OlFCApplyCallbackStruct *cdata = (OlFCApplyCallbackStruct *)call_data;
    printf("the font chosen is:\n\t%s\n", cdata->current_font_name);
}
```

SuperCaret Support

Developers who have used OLIT in the past know that when the application written in OLIT gets focus, the widget with focus is colored with either RED or the input focus color defined in the resource file. The input focus is used to tell which widget has the focus in a mouseless model where keyboard navigation is possible. The following section discusses the input focus model implemented in OLIT 3.0, 3.0.1, and 3.1 and then discusses the new input focus model based on supercaret.

Figure 4-16 contains a representation of the `SuperCaret` widget:

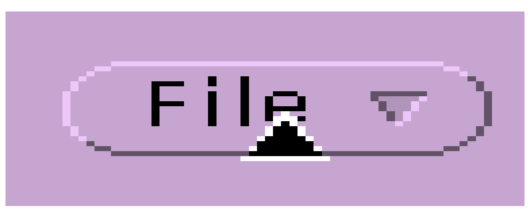

Figure 4-16 SuperCaret Widget

Existing Input Focus Model

The input focus model is available when mouseless mode is on. This is done by:

- setting `XtNmouseless` to `true` on the top-level window of an application.
- setting `*mouseless: true` in resource file.

Note – In OLIT 3.0, `XtNtraversalOn/*traversalOn` did the same thing.

When the input focus is active, the application shows which widget has the focus by using focus color as the foreground for the widget. The application and the end user have the flexibility to change the focus color by setting `XtNinputFocusColor/*inputFocusColor`.

New Focus Mode

With the new input focus model introduced in OLIT 3.2, the input focus feedback will be provided by a *supercaret* in the form of a triangular caret overlaying the GUI component that has keyboard focus. The caret appears to move between objects as OPEN LOOK mouse and mouseless operations cause the focus object to change.

To enable the this feature, the following application resources are available:

Note – Mouseless mode must be on for this to work.

Set `XtNinputFocusFeedback` to:

- `OL_INPUT_FOCUS_COLOR` for the old model with input focus color.
- `OL_SUPERCARET` for the new model with super caret.

The resource, if set by the application, should be done on the top-level window. This resource should be set as an argument to `XtAppInitialize()` or the equivalent of this function.

You can set this resource in your resource file using the line:

```
*inputFocusFeedback: supercaret/inputfocuscolor
```

The default value is for `inputfocuscolor`, although this may change in a future release. If the resource is set to `inputfocuscolor`, the `XtNinputFocusColor` resource is used to determine the input focus color.

`SuperCaret` example:

```
*inputFocusFeedback:  supercaret         <- default
*inputFocusFeedback:  inputfocuscolor    <- old style
```

OpenWindows Server (X11 Release 5 / DPS based)

OpenWindows Version 3.2, as well as earlier releases, contains the X11 Release 4 compliant X11/NeWS server. The next major release, OpenWindows Version 4.x, will include the X11 Release 5 compliant server released by MIT with Display PostScript System (DPS) from Adobe as an extension to the server.

Note – In this section, “OpenWindows Version 4,” “X11 Release 5-compliant server,” and ”Next Generation server” can be used interchangeably and will refer to the redesigned server based on MIT X11 Release 5 and DPS.

This section discusses porting considerations for application and hardware developers. The section focuses on new features of X11 Release 5 from MIT, as well as SunSoft’s value-added features to the server. The performance goals for this server with all the value-added work from SunSoft are similar to the Sun Sampler Server from MIT.

Note – With the OpenWindows Version 4 server, all support of SunView and NeWS will be discontinued.

Changes to OpenWindows Version 4 Server

OpenWindows Version 4 has many new features which are discussed in in this section. These include:

- a porting interface based on DDX (with SunSoft’s value add of loadable DDX, Multiple Plane Groups (MPG) and Multiple Hardware Color maps (MHC) support)
- input extensions for devices (with SunSoft’s value add of allowing dynamically loadable extensions with no server recompile)
- font support for scalable/outline and bitmap fonts (F3 forman, Type 1, Type 3, and Bitmaps)
- new server options
- customization for end users with disabilities
- Display PostScript Extension with support for the DPS toolkit and resources
- internationalization
- device-independent color

The Porting Interface

The OpenWindows Version 4 server uses the DDX interface from MIT as the interface for adding devices to the server. This section assumes that you are is already familiar with the MIT server and its porting interface (DDX). If not, please see the section, “Additional Information”.

SunSoft has added the ability to dynamically load DDX modules at runtime. This will enable third-party developers to develop DDX modules that can be delivered as separate components. SunSoft also provides DDX utility libraries that supply functionality which is common across devices. This functionality will facilitate the task of porting the OpenWindows Version 4 server to new graphics devices.

The DDX interface is quite extensive; the screen structure alone contains approximately 70 function vectors. Fortunately, many of these functions do not need to be specialized for every device. There are general purpose DDX modules that provide functionality that can be used by a device's DDX implementation; these are shown in Figure 4-17.

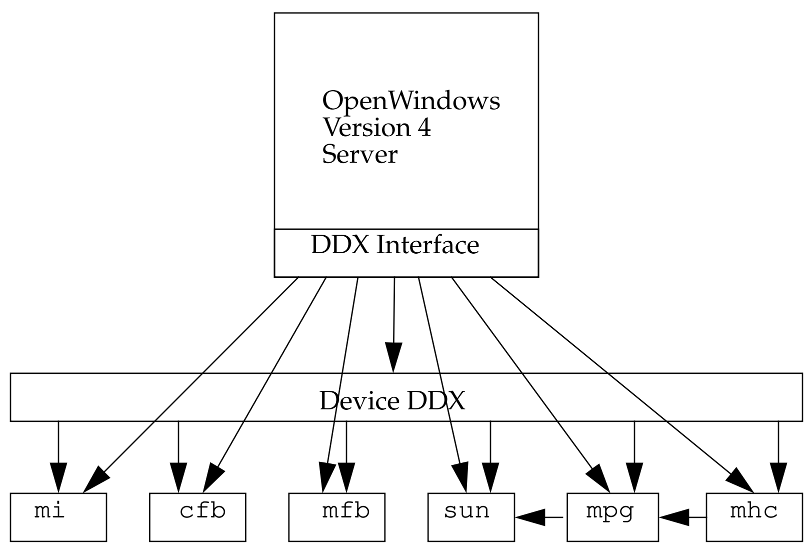

Figure 4-17 DDX module interdependencies

These modules provide different types of functionality. The `mi` module supplies high level, machine independent, functionality; `cfb` and `mfb` supply functionality useful to memory-mapped color and monochrome frame buffers. The `mpg` and `mhc` modules are for devices with multiple plane groups and multiple hardware color maps, and the `sun` module is for frame buffers that have device drivers that respond to specific ioctls.

Note – Although the DDX interface will be exposed to Independent Hardware Vendors (IHVs), it should be clear that it is an uncommitted interface, subject to change in subsequent releases of OpenWindows. There is also a DDX interface for input devices which is not covered in this book.

Loadable DDX

Loadable DDX is a mechanism that allows the server to dynamically load DDX modules at runtime instead of having to re-link the server to add support for new devices. The DDX module is essentially a shared library that is linked into the server at runtime through an explicit call to `dlopen()`. The location of the DDX modules and their mappings between devices is determined by a configuration file.

Dumb Frame Buffer Support

As with the MIT server, OpenWindows Version 4 provides a set of general purpose support routines for dumb frame buffers. This includes `mfb` for monochrome frame buffers and `cfb` for color frame buffers.

Although `cfb` supports depths of 2-, 4-, 8-, 16-, and 32-bits, only the 8-bit and 32-bit depths are currently built and tested.

Sun Device Support

Many frame buffers on the Sun platform have common characteristics. For example, they are accessed through device drivers and these device drivers support a common set of ioctls. The `sun` DDX module provides functionality that takes advantage of these characteristics.

Multiple Plane Group Support

For devices with multiple plane groups there is a DDX module that provides most of the management functions necessary for such devices. The `mpg` support also includes the ability to minimize exposure events between windows that reside in different plane groups. The `mpg` interface is designed so that other layers, such as `cfb` and `mfb`, may be used beneath it to do the actual rendering and window manipulation.

Additional Information

SunSoft will ship an OpenWindows Version 4 Server Device Developer's Kit for IHVs interested in porting their devices to the Solaris platform. The kit contains:

- documentation describing the interfaces
- necessary header files and libraries

- sample implementations (reference ports)
- sufficient server source code for debugging

There are also documents that describe the X11 window server and its porting interface. All of these come on the distribution tape from MIT except the one from Digital Press, which is highly recommended for those new to X11 Server development:

- *Strategies for Porting the X v11 Sample Server*, Susan Angebranndt, Raymond Drewry, Phillip Karlton, Todd Newman, Bon Scheifler, and Keith Packard, MIT X Consortium.
- *Definition of the Porting Layer for the X v11 Sample Server,* Susan Angebranndt, Raymond Drewry, Phillip Karlton, Todd Newman, Bon Scheifler, and Keith Packard, MIT X Consortium.
- *X Server Internals: A Tutorial on the MIT Sample Server at Release 4*, Keith Packard, MIT X Consortium.
- *X Window System, Version 11, Release 5 Release Notes*, MIT X Consortium Staff, MIT Laboratory for Computer Science.
- *The X Window System Server*, Elias Israel and Erik Fortune, Digital Press.

The Loadable DDX Interface

Note – The interfaces described below are subject to change. Please consult the OpenWindows Version 4 Server Device Developer's Kit that accompanies the OpenWindows Version 4 product.

The server interfaces to a loadable DDX routine in the following manner:

1. The module containing the device's DDX module is installed in the configuration directory. This directory is specified to be in the `OW_CONFIG_PATH` which defaults to `/usr/openwin/etc/server`.
2. The device type, device name, device class, the filename, and the name of the initialization function are added to the `ow_config` file (also found in the `OW_CONFIG_PATH`). The frame buffer section looks similar to the following:

# fld	DevName	Class	libName	InitFunc
f	`/dev/cgsix0`	12	`libddxcg6.so.1.0`	`sunCG6Init`
f	`/dev/cgthree0`	6	`libddxcg3.so.1.0`	`sunCG3Init`
f	`/dev/cgfour0`	8	`libddxcg4.so.1.0`	`sunCG4Init`
f	`/dev/bwtwo0`	2	`libddxbw2.so.1.0`	`sunBW2Init`
f	`/dev/cgeight0`	7	`libddxcg8.so.1.0`	`sunCG8init`
f	`/dev/fb`	0	`libddxfb.so.1.0`	`sunFBInit`
f	`/dev/gt0`	18	`libddxgt.so.1.0`	`sunGTInit`

The first field, `# fld`, is the type of device, and it should contain an `f` for all frame buffers or other output devices (other types of modules are described in this file: `i` denotes input handlers and `e` denotes an extension).

The second field, `DevName`, is the filename that corresponds to the device.

The third field, `Class`, is the value returned when the `FBIOGTYPE` ioctl is issued to the device driver. It is only used when the server is started without specifying a device or when the servier is started on `/dev/fb`.

The fourth field, `libName`, is the name of the file that contains the shared library with the DDX module in it. A loadable module may be linked against other loadable modules.

The fifth field, `InitFunc`, is the symbol name for the entry point into the DDX module's initialization function.

3. From this information, the server, at initialization time, loads the appropriate module and calls its `InitFunc`. It is up to the `InitFunc` to initialize the device and data structures so that the server can run.

The Initialization Function

Each type of device has an initialization function that is called once for each device. The initialization function is responsible for initializing the device and the screen data structure associated with it.

```
xxxInit(index, pScreen, argc, argv)
    int        index;      /* The index of pScreen in the ScreenInfo */
    ScreenPtr   pScreen;   /* The Screen to initialize */
    int         argc;     /* The number of the Server's arguments. */
    char       **argv;    /* The arguments themselves. Don't change!*/
```

Note – A mechanism for setting version numbers is being developed that will allow the server and the DDX modules to detect incompatibilities. This may require another entry point or may be a part of the screen initialization function. A proposal for this system is included at the back of this document.

The Sun Interface

The device initialization procedure mentioned is responsible for initializing the screen structure and all of its function vectors. Since many `sun` devices have similar characteristics, there is a set of utility functions that make the porting job easier and reduces the size of the server. The Sun interface provides services to do such things as opening the device, checking its type, updating the color maps, and managing the cursor.

Note – The interfaces described in this section are subject to change. Please consult the OpenWindows Version 4 Server Device Developer's Kit that accompanies the OpenWindows Version 4 product.

- `sunOpenFrameBuffer`

```
int
sunOpenFrameBuffer(expect, pfbType, index, pSunFbs, argc, argv)
    int expect;              /* The expected type of frame buffer */
    struct fbtype *pfbType;  /* Place to store the fb info */
    int index;               /* Screen index */
    fbFd *pSunFbs;           /* Pointer to private fbFd */
    int argc;                /* Command-line arguments... */
    char **argv;             /* ... */
```

`sunOpenFramebuffer()` opens the frame buffer, queries the type of frame buffer using the `FBIOGATTR` and `FBIOGTYPE` ioctls, verifies the device is what was expected, and returns the file descriptor of the device. If an error occurs, `-1` is returned. The structures pointed to by `pfbType` and `pSunFbs` are filled in.

- `sunScreenAllocate`

```
    Bool
    sunScreenAllocate(ScreenPtr pscreen)
```

`SunScreenAllocate()` allocates the sun screen private index if necessary and allocates the `sunScreenPrivate` for the given screen and sets the `devPrivates` entry to point to it. `FALSE` is returned if there is an error.

- `sunScreenInit`

```
    Bool
    sunScreenInit(ScreenPtr pScreen)
```

`sunScreenInit` initializes the screen's cursor functions and `SaveScreen` functions to Sun Specific routines. `sunScreenInit` returns `TRUE` upon success.

- `sunCGScreenInit`

```
    void
    sunCGScreenInit(ScreenPtr pScreen)
```

`sunCGScreenInit` is called if the device is sun Color Frame buffer. It initializes the color map functions (`InstallColormap`, `UninstallColormap`, `ListInstalledColormaps`, `StoreColors`, and `UpdateColormap`) in `pScreen` to use the Sun-specific functions. The underlying device must support the `FBIOPUTCMAP` ioctl for these functions to be used.

- `SunSetPixmapFormat`

```
    Bool
    sunSetPixmapFormat(PixmapFormatRec request)
```

`SunSetPixmapFormat` is used by each device to advertise what pixmap formats it supports. Since there may be multiple screens supporting different depths it is necessary to have a centralized facility for registering the formats. There can only be one format per depth, so any attempt to define a new format for an existing depth will fail and sunSetPixmapFormat will return `FALSE`.

Currently the following formats are known to be supported by some Sun Devices:

Depth	BitsPerPixel	ScanlinePad
1	1	`BITMAP_SCANLINE_PAD`
4	8	`BITMAP_SCANLINE_PAD`
8	8	`BITMAP_SCANLINE_PAD`
24	32	`BITMAP_SCANLINE_PAD`
32	32	`BITMAP_SCANLINE_PAD`

SunSaveScreen

`SunSaveScreen` is the `saveScreen` function for most Sun devices and is installed automatically by `sunScreenInit`. A devices's `ddx` module may need to call this function directly to ensure that video is enabled at start-up and before exiting using the format:

```
sunScreenSave(ScreenPtr pScreen, int on)
```

where the value of `on` has the following meanings:

`SCREEN_SAVER_ON`	Turn on the screen saver. Blanks the screen.
`SCREEN_SAVER_OFF`	Turn off the screensave, Enables video.
`SCREENSAVER_FORCER`	Update the time of the last screen saver mode change.

SunDisableCursor

Sometimes, usually before exiting server when the screen's `closeScreen` function is being called, it is necessary for the `ddx` module to disable the `sunCursor`. Usually this is done to handle the case where the cursor is a hardware cursor.

```
sunDisableCursor(ScreenPtr pScreen)
```

SunGetMonitorRes

The `ddx` module calls this routine to get the monitor's resolution. The monitor's resolution is specified on the command line; this routine fills in what the end user specified. If the end user does not specify a monitor resolution, 90 DPI is the default.

```
void sunGetMonitorRes(int index, int *dpix int *dpiy)
```

The integer `index` is the screen's index, and `dpix` and `dpiy` are filled in by this routine.

Note – Future releases of the OpenWindows server may store the monitor resolution in some sort of configuration file associated with the screen.

SunGetVisualInfo

This routine is called to get the specified visual information for the screen. Since the user may specify the default visual class, the default depth, or grayvisual, the `ddx` module must know how to set up its visuals.

```
void
sunGetVisualInfo(int   index,
                 int   *defClass,
                 int   *devDepth,
                 Bool  *grayVis)
```

The integer `index` is the screen's index, `defClass` is the visual class that should be used as the default, and `defDepth` is the default depth. This routine will assign `-1` to either `defClass` or `defDepth` if there was no specification for these on the command line. `GrayVis` is set to indicate the `-grayvis` command line option which should disable all color visuals, as the device may be connected to a gray-scale monitor.

The MPG Interface

Some devices contain multiple plane groups (MPG) to support overlays and visuals of varying depths. The `mpg` DDX module provides the following features:

- *Window Hierarchy Operations*
  MPG provides the necessary screen functions to move windows, and handle exposure events. Typically when a window is moved, all of its plane groups need to be moved. Likewise, when a window is exposed, all of its plane groups need to be repainted.
- *Minimizing Exposure Events*
  MPG provides specialized window hierarchy functions that minimize exposure event between windows that reside in separate plane groups.
- *Leveraging off of existing DDX interfaces*
  MPG is designed to layer on top of existing rendering and window functionality such as `cfb` or `mfb`.

Architectural Overview

The `mpg` module is basically a data driven design. It is up to the device's DDX module to inform the `mpg` DDX module what plane groups are being used by what windows and how it is using them. From this information, the `mpg` windowing functions take care of moving, preparing and calculating exposure to the plane groups.

The `mpg` layer does not actually do any rendering. Instead, it is designed to layer on top of other DDX modules, like `cfb` and `mfb`, which provide rendering and some basic windowing functions. This way, a frame buffer with a 24-bit plane group and a 1 bit overlay can use `cfb32` and `mfb` to do the rendering and most of the window functions. The `mpg` module manages the switching between modules and provides the rest of the windowing functions.

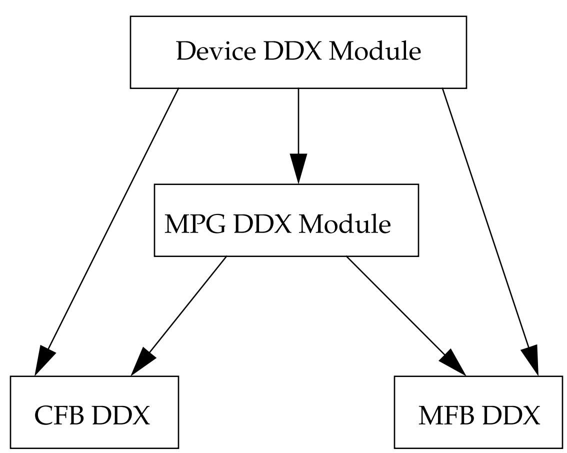

Figure 4-18 MPG DDX Interface Diagram

The `mpg` layer does not explicitly call `cfb` or `mfb`, but can use any device specific screen functions. These are shown as examples of how `mpg` may be configured.

There is also a distinction between window types and plane groups. The example device may have two types of windows—monochrome and color—but may have 3 plane groups: color, overlay and overlay enable. There must be a screen drawable for each plane group and there be certain screen functions for each type of window.

Note – The Functional Interface for MPG can be found in the OpenWindows Version 4 Server Device Developer's Kit that accompanies the OpenWindows Version 4 product.

The Multiple Hardware Color Maps (MHC) Interface

The following terminology is used when discussing multiple hardware color maps:

A *color look-up table* is a hardware look-up table (LUT) that outputs an RGB color triple when a pixel value is input. R, G, and B are the voltage levels to drive the red, green, and blue monitor guns respectively. A LUT is also called a *hardware color map*.

A *software color map* is a software abstraction of the color mapping process that a color LUT provides. Clients can allocate pixels from a software color map by presenting an RGB triple to the server. The server hands back to the client a pixel index where the triple has been stored in the color map. The software color map can be loaded or installed into a hardware color LUT. Software color maps are supported through the X11 protocol as a *color map resource*. A software color map is also called a *color map*.

Note – In the following discussion the word *color map* used alone will refer to a software color map. The acronym *LUT* used alone will refer to a color LUT, or hardware color map.

Color Map Installation–Single LUT Devices

Most devices have only one LUT. Only one color map may be installed in this LUT, where "installed" means the LUT is guaranteed to contain the values of all of the allocated entries in the color map.

In an environment in which there are several clients each having their own color map, only one of these client color maps will be installed at a given time. The windows that are associated with the installed color map will show their correct colors. Windows that are associated with some other color map may show false colors, depending on whether their entries collide with the installed color map. This display of false colors is referred to as *color map flashing* or *technicolor*.

Color Map Loading

The server maximizes the number of valid colors in the hardware color LUT. The server loads all of the allocated entries of the newly installed color map into the LUT. Then, for those that are unallocated, the server looks for corresponding allocated entries in other color maps to load. Slots in the default color map are examined first, followed by slots in the StaticColor color map (only if the OpenWindows option `-favorstatic` is selected). For a given entry, if no allocated entry is found in either of these color maps, the LUT value will not be changed; it will still have the value of the last allocated color map entry that was installed in that LUT.

Color Map Installation–Multiple Look-up Table Devices

The GS and GT frame buffers provided by Sun Microsystems Computer Corporation provide multiple 8-bit indexed look-up tables (LUTs). The purpose of multiple LUTs is to increase the frequency of true colors being displayed and to reduce the occurrence of color map flashing.

The GS has two 8-bit LUTs. The GT has a maximum of eight 8-bit LUTs. These devices also provide separate LUTs for the other supported visual depths. The GS provides a single 4-bit LUT and a single 24-bit LUT. The GT provides a single 24-bit LUT. Because there is only one LUT for the other depths, the installation of color maps for visuals of these depths behaves the same way as for a single LUT device. Therefore, the remainder of this discussion will deal only with 8-bit LUTs.

When there are *n* color maps, each of them may be installed in hardware when there are *n* LUTs. However, when there are fewer than *n* LUTs, only a subset of the color maps can be installed. The decision making process that the server uses to determine which color maps to assign to which LUTs is called the *server color map installation policy*.

Note – Ultimately, the window manager decides which color maps are installed in which LUTs. This is referred to as the *window manager color map installation policy*. This policy is different from the server color map installation policy, which only specifies how the server reacts to various `InstallColormap` and `UninstallColormap` requests. The window manager color map installation policy defines how and when a window manager will send these requests to the server. The window manager policy is outside of the domain of this chapter and will not be mentioned further.

The color map installation policy used by the window server for visuals that provide *n* LUTs is to keep the *n* most recently installed color maps resident in some LUT. “Most recently installed” means that the color maps were installed by the last *n* `InstallColormap` requests sent by the window manager.

For example, if there are two LUTs (L1 and L2) that have two color maps installed (C1 is installed in L1 and C2 is installed in L2), and the server receives an `InstallColormap` request for a third color map (C3), the least recently installed color map (C2) will be *bumped* from L2, and C3 will be installed in L2. If another `InstallColormap` request is received for a fourth color map (C4), C1 will be bumped from L1 (because it is now the least recently installed) and C4 will be installed into L1. Another term used for this bumping is *implicit uninstallation* (this is the term used in the X11 protocol specification).

When a color map is explicitly uninstalled by an `UninstallColormap` or `FreeColormap` request, the server takes advantage of the available LUT and assigns it the most recently installed color map that has been bumped. This is referred to as *unbumping* or *implicit installation*. In the preceding example, if C4 were freed, C1 would be assigned C4's LUT (L1) because C1 is a bumped color map which still wants a LUT if it can get it, and it was requested by the window manager to be installed before the request to install C2.

Note – The Functional Interface for MHC can be found in the OpenWindows Version 4 Server Device Developer's Kit that accompanies the OpenWindows Version 4 product. To see sample source code, see the section "Sample CG3 Source with Sun Interface" in Appendix C titled "OpenWindows Source Code."

Input Extensions for Devices

This section describes the design of how input devices are handled in the OpenWindows Version 4 server. This section discusses how the Consortium's implementation of the Input Extensions has changed and how those changes result in a more flexible implementation, and how to apply the concepts of these concepts to the internals of the server. This section also discusses the interface between the owmux STREAMS module and the server and device libraries.

Client programs use the Input Extension to gain access to non-core devices in the server. The Input Extension is an extension that was adopted as a standard and is a part of the X11 Release 5 distribution. As a part of the distribution there are three documents that are of interest; the X11 Input Extension Library Specification, the X11 Input Extension Protocol Specification, and the X11 Input Extension Porting Document. These documents can be found in the subdirectory `mit/hardcopy/extensions/xinput`. This document is not intended to be a replacement for any of these documents; it intends to present the specifics of how input devices may be dynamically loaded into the server, as well as how this is integrated with the Input Extension.

Current window system servers provide very limited support of input devices. These servers are not dynamic or extensible. The X11/NeWS server is limited to supporting the mouse, the keyboard, SunDials, SunButtons, and the lightpen. It also has the ability to support certain devices that can present themselves as one of the above. The Sun sample server that is a part of the X11 Release 5 distribution is even more limited. It only supports the mouse and keyboard.

With the adoption of the Input Extension as a standard in X11 Release 5, there now exists a protocol specification in X that handles multiple input devices attached to the server and allows multiple clients to receive events from that device. The problem with this extension is that the server must have the code to control each device built into it when the server is created. Once a server is shipped, there is no way to add support for a new device that the server does not know about. The following design for handling input devices in the server solves this problem. It presents a design for dynamically loading input devices, and accessing them through the Input Extension.

Note – Dynamic loading is true for extensions to the server in general.

The client interface for accessing input devices in the server is the Input Extension as defined in X11 Release 5. The design presented here will not change that interface in any way. Any client protocol requests that are used in this document are already defined in the Input Extension.

Overview of Server Configuration and Start-Up

The server reads a configuration file to find out what devices are available and which devices are to be used as the core pointer and core keyboard. Every input device the server wishes to use is dynamically loaded into the server as it is needed. This is done by putting all of the code that is specific to an input device into a shared library. The server may then load this library when it needs to and use the functions in the shared library to control the device.

When the server starts up, it will read the `ow_config` configuration file to find out which devices are to be used as the core keyboard and core pointer. If there are none, the server will exit because these two devices are required by the core X protocol. If the core keyboard and pointer are present, the server loads their shared libraries, initializes them, and starts them. At this point, if no client requests any other devices to be used, via the Input Extension, the server will continue to run just like the X11 Release 5 Sun sample server.

Overview of Client Access to Devices

A client is required by the Input Extension to execute the following steps when gaining access to devices:

- Get a list of the devices that the server supports by issuing a `ListInputDevices` request. The first `ListInputDevices` received by the server causes it to load and initialize all input device found in the `ow_config` file (except the keyboard and mouse which have already been loaded). If the `DeviceControlProc` returns a `!Success`, the server assumes the hardware is not present and unloads the device. All subsequent `ListInputDevices` will cause the server to search the `ow_config` file for any devices that may have been added since the last `ListInputDevices`. If any new devices are found, they are loaded and initialized. During initialization, the device registers all its features with the server. This information is passed back to the client as part of the `ListInputDevices` request. Note that the devices have not been turned on by a `ListInputDevices` request; therefore, the server will not accept input from them.

- Once a client has the list of devices, it may open one of the devices by issuing an `OpenDevice` request. The server tells the device to turn itself on, and when input is pending on that device, the server will read the data and put it into the event stream.
- At this point, the client may issue any of the Input Extension protocol requests to receive events, initiate grabs, and control features of the device. This portion of device handling in the server is exactly as it appears in the X11 Release 5 Input Extension. No changes to the Input Extension protocol have been made.
- When the client is finished with the device, it issues a `CloseDevice` request. If the client issuing the `CloseDevice` request was the last client to have the device open, the server tells the device to turn itself off. If there are other clients that have the device open, the server leaves the device open and continues to read the device until no clients have the device open. Although the server may still read the device, the client that issued the `CloseDevice` request no longer receives events from the device. This is because the event mask for that client has been cleared by the Input Extension as part of the `CloseDevice` processing.

Overview of Server Restart and Shutdown

During server restart and shutdown, all open devices are closed and unloaded. During the close process the device is notified of the shutdown and must free any memory that may have been allocated and close its file descriptor. There is no difference between a server restart and a shutdown.

Overview of the Input Multiplexor Device

The MIT X11 Release 5 server reads the mouse and keyboard file descriptors and then sorts the events from the two devices so that the events are queued in the server in the order they actually happen. The server introduces an input multiplexor that reduces the amount of processing required by the server to handle incoming device events.

The input multiplexor is a M-to-one or lower STREAMS multiplexor. It will multiplex events from multiple devices into a single stream for the server to read. The multiplexor reduces the number of file descriptors you have to read from and simplifies the sorting of events from multiple file descriptors. The multiplexor also removes the need for a read procedure from the device’s shared library.

You do not need the extension devices to use the input multiplexor. When a device is initialized, it must register itself as either having a read procedure or as wanting the input multiplexor to read the device. Only devices that are generating VUID events can use the multiplexor. All others must supply a read procedure to read the device events and format the data.

What follows is a detailed description of the server modifications needed to support dynamic loading of input devices in the OpenWindows Version 4 server and how it interacts with the Input Extension.

Server Configuration and Start-Up

The configuration file `ow_config` is used by the server to read in information about the dynamically loadable extensions available, the different dynamically loadable frame buffer modules, and the dynamically loadable input devices. The server looks for the `ow_config` file in the path as follows:

- `configpathA` comes from the `-cp` command line option
- `configpathB` comes from the `OW_CONFIG_PATH` environment variable
- `configpathC` comes from the `OPENWINHOME` environment variable
- `configpath` = `configpathA:configpathB:configpathC`

The format for the input devices portion of the `ow_config` file is as follows:

Parameter	Meaning
`fld`	`i` (indicates input device)
`DevName`	Name of the device that uniquely identifies the device.
`Class`	`K` indicates the core keyboard device.
	`P` indicates the core pointer device.
	`-` indicates other than core keyboard or pointer device.
`SharedObjectName`	Name of the shared object file.
`CompatFunctionName`	Function that is called to check compatibility between the DDX and the handler.

For example, the core keyboard and mouse would be specified as follows:

# fld	DevName	Class	SharedObjectName	CompatFunctionName
i	`IKBD`	`K`	`ddxSUNWkbd.so.1`	`ddxSUNWkbdHandlerCompatible`
i	`IMOUSE`	`P`	`ddxSUNWmouse.so.1`	`ddxSUNWmouseHandlerCompatible`

When the server is being started or restarted, the routine `InitInput()` gets called. It is the responsibility of this routine to open the muxtiplexor device, initialize the core keyboard and pointer, and register them with the rest of the server. To accomplish this in OpenWindows

Version 4, InitInput() calls a new function called ParseDeviceFile(). If this function is being called from InitInput(), it causes the ow_config file to be parsed while the server looks for the core keyboard and pointer. When it encounters one of these devices it calls the dlopen() function to open the devices shared object in $(OPENWINHOME)/lib/X11. Then the compatibility function specified in the configuration file is retrieved via dlsym() and calls this function. If the device handler is compatible with the server, the handler passes back a pointer to its DeviceControlProc() to the server. The device is then added and registered with the server via AddInputDevice() and RegisterPointerDevice() and RegisterKeyboardDevice(). After ParseDeviceFile() completes, InitInput() registers the core pointer and keyboard with the mieq layer of the server by calling mieqInit(). The second case that ParseDeviceFile() needs to handle is discussed in the "Client Access to Device" section.

The DeviceControlProc()

The DeviceControlProc() is one of the entries in the ow_config file. It is one of the parameters passed to the AddInputDevice() function. The purpose of this function is to allow the server to tell the device to perform some sort of action. This allows the server to control the device without having to know how to perform an action on a specific device. The control function can be called with one of four actions.

The first action is DEVICE_INIT. During the DEVICE_INIT action, the device must initialize any data structures, open up the device, and register all of the features of the device.

The second action that the DeviceControlProc() must support is DEVICE_ON. This tells the device that it needs to be activated. A device that wants the input multiplexor to read its device events links itself to the multiplexor at this time by using the appropriate ioctl. Devices that have a read procedure to handle device events must at least call AddEnabledDevice() and have the device start generating SIGPOLLs when input events are at the STREAM head.

The third action is DEVICE_OFF. This tells the device that it will be turned off for a while. A device that is using the input multiplexor must unlink from the multiplexor at this time. A device that has its own read procedure must at least call RemoveEnabledDevice().

The last action the device must support is DEVICE_CLOSE. This tells the device that the server is about to exit so the device closes itself and cleans up any states. These four actions allow the server to control a loadable device.

Device Features and Reading the Device

The Input Extension provides a number of routines that allow a device to register features that a client program may use. In OpenWindows Version 4, a device needs to use these registry routines during the DEVICE_INIT action of its control function. During DEVICE_INIT, the device may use any of the registry functions that are defined in the *X11 Input Extension Porting*

Document. Because of the dynamic loading of input devices in the server, there are a number of new registry functions that will be available. These may be viewed in two categories. The first category contains functions that help define device features to the server. The second category deals with how the data is to be read from the device.

There are several routines that let a device specify whether it supports a feature. OpenWindows Version 4 has six additional routines that a device may use during `DEVICE_INIT` to register its preferences. The routines and their interfaces are briefly described:

- `RegisterModifierCheckProc()`
  This routine registers a function to be called when the validity of a keycode needs to be checked by the device. This can happen when a client wishes to change the Modifier Mapping of a device.
- `RegisterSetDeviceModeProc()`
  This routine registers a function to be called when a client wishes to set the mode of a device. The two modes that a device may have are Absolute and Relative positioning.
- `RegisterSetDeviceValuatorsProc()`
  This routine registers a function to be called when a client wishes to change the valuators of a device.
- `RegisterChangeDeviceControlProc()`
  This routine registers a function to be called when a client wishes to change the control of a device. The only device control that is currently supported is the `DEVICE_RESOLUTION`.
- `RegisterXKeyboardInterest()`
  This routine registers a device as being a possible core keyboard device. This is used in the `ChangeKeyboardDevice` protocol request.
- `RegisterXPointerInterest()`
  This routine registers a device as being a possible core pointer device. It also registers a function to be called when a client wishes to make this device the core pointer. This is used in the `ChangePointerDevice` protocol request.

The second category of registry functions that are provided deal with how the device wishes to be read:

- `RegisterMuxIo()`
  This routine registers a device as one that will be read by the multiplexor.
- `RegisterFdIo()`
  This routine registers a device as one that will read itself.

The general flow of how data is read from devices is as follows. Figure 4-19 can be used to help in following this flow. As presented earlier, there are two ways that a device may be read. The device can chose to let the multiplexor read its events, or it can read the events itself. In the case were the device is reading its own events, the device driver must generate the signal `SIGPOLL` when data is available to be read. This requirement is mainly in place to enforce consistent flow through the server.

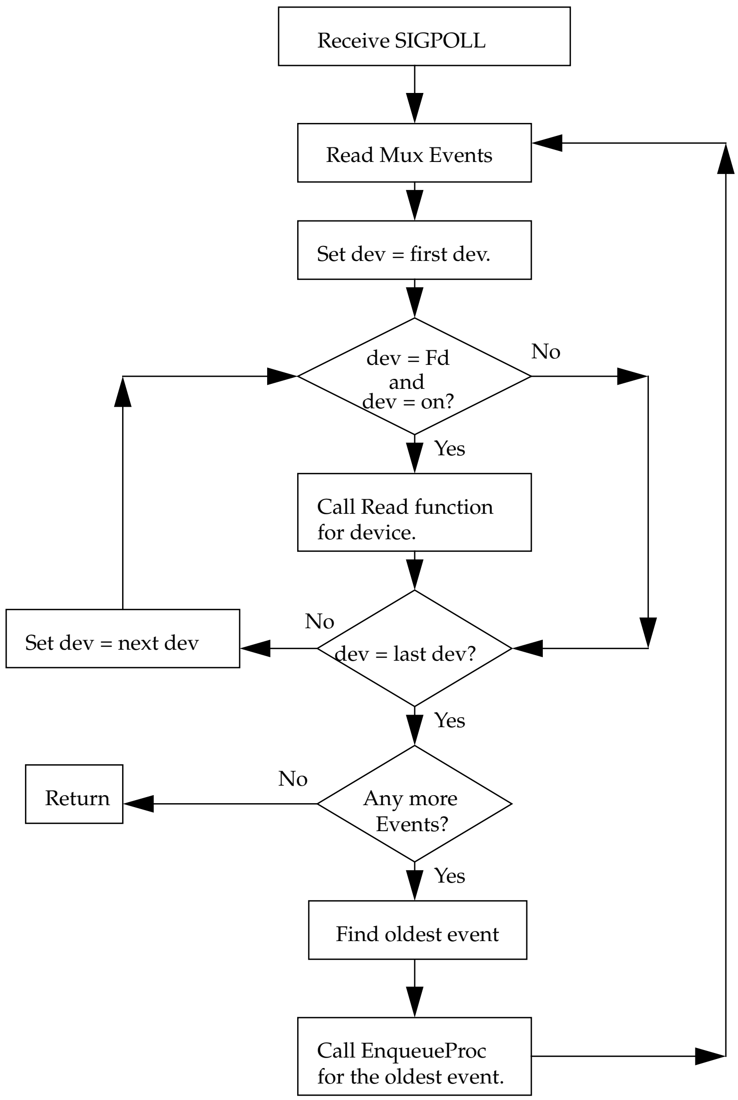

Figure 4-19   Data Flow from Devices

The basic flow is as follows. A `SIGPOLL` is received by the server. When this happens, the server loops through the following steps until no more events are available on any of the input devices.

1. Read all events from the Input Multiplexor.
2. For each device that is on and wishes to read itself, call the `readProc()` function for that device.
3. Check to see if there are any events from all of the sources just read,
   a. If there are no more events, break out of the loop and return.
   b. If there are more events, continue to 4.
4. Find the oldest event.
5. Give the oldest event to the `enqueueProc()` for that device.
6. Loop back up to 1.

The `enqueueProc()` in 5. is present for every device that is being read. The `enqueueProc()` takes an event, does any device specific processing on the event, converts it to an xEvent, and finally places it on the global event queue via `mieqEnqueue()`. Also, it may seem that it is wasteful to attempt to reread devices after every call to `enqueueProc()`, but this is the only way to ensure that events are delivered in the correct order.

Note – This design was taken straight from the X11 Release 5 Sun sample server.

There are two routines that allow a device to specify how the device wishes to be read. Both of the routines take a `DevicePtr`, a file descriptor, and an `enqueueProc()` as arguments. If the device wishes to read itself, it will use `RegisterFdIo()` and supply a `readProc()` as well as the three other arguments. A device that generates `VUID` style events and wishes to be read by the input multiplexor supplies the three arguments to `RegisterMuxIo()`.

That takes care of all of the feature registration portions of the OpenWindows Version 4 Input Extension. This is the most complicated part of handling input devices in the server. It is very important that the interfaces presented are well-defined so dynamic loading of input devices works without any side affects.

Client Access to Devices

At the end of server initialization, the core keyboard and core pointer are the only devices that are initialized and generating events. If a client wishes to use an Input Extension device, the first thing that it needs to do is issue a `ListInputDevices` protocol request. When the server receives this request, it must gather all the features of all the possible devices and pass this information back to the client. At this point the server calls `ParseDeviceFile()`.

The first time `ParseDeviceFile()` is called due to a `ListInputDevices` request, the `ow_config` file is reparsed. Any new devices that have not been initialized will have it's shared library loaded, call `AddInputDevice()` and `RegisterOtherDevice()`, and the control function of the device will be called with `DEVICE_INIT`. This will cause the device to register all of its features with the server, so that the server may send the information back to the client. At this point the device is known by the server, but is not generating any events. This is accomplished through the `OpenDevice` request.

For each additional `ListInputDevices` request `ParseDeviceFile()` will stat the file to see if the user has modified the file since the last `ListInputDevices` request. If the file has not changed, no action is taken. If the file has changed the file is reparsed. New devices are initialized as above. Devices that are no longer in the `ow_config` file are closed.

Each time the `ow_config` file is read due to a `ListInputDevices`, devices that have a class of `K` or `P` and were not the original core pointer and keyboard are treated as extension devices.

When a client wishes to use a device that is in the list of devices returned by the `ListInputDevices` request, it must use the `OpenDevice` request. When the server receives this request, it will tell the device to start generating events by calling the `DeviceControlProc()` with a value of `DEVICE_ON`. If the device is already on, this request adds the client to a list of clients that currently have the device open.

When a client is finished with a device, it uses the `CloseDevice` request to have the server close the device. When the server receives this request, it will clean up all of the references that this client had on this device, such as grabs and event masks. It will also remove the client from the list of clients that currently have the device open. If the client is the only client with the device open, the server will call the `DeviceControlProc()` with a value of `DEVICE_OFF`. This tells the device to stop generating events.

Server Shutdown

When the server is about to exit or restart, it goes through all of the devices that have been initialized and closes them. This is done via a call in the server called `CloseDevice()`. In `CloseDevice()`, the server calls the `DeviceControlProc()` with a value of `DEVICE_CLOSE`. This tells the device to clean up all of its resources because the server is about to exit. In addition to calling the `DeviceControlProc()` in `CloseDevice()`, the server also goes through its data structures and clean up any references to the device. This includes calling `dlclose()` on the shared library handle that is associated with the device. This ensures that if the server is just restarting, the server restarts successfully.

Note – The interfaces for the Device Library Interface Specification will be included in the OpenWindows Version 4 Server Device Developer's Kit that accompanies the OpenWindows Version 4 product.

The owmux STREAMS Module Interface

The owmux STREAMS module is an M-to-one or lower STREAMS multiplexor. The server will uses the owmux to merge events from multiple input devices below it and present a single stream of events for the server to process. The owmux onlys work with devices that generate firm_events as specified by the protocol of a VUID.

The owmux understands the `I_LINK`, `I_UNLINK`, `MUXIOCGETLOWER`, and `MUXIOCSETLOWER` ioctls as defined in `/usr/include/sys/muxio.h` and `/usr/include/sys/stropts.h`. All other ioctls are passed downstream to the device that has been specified by the `MUXIOSETLOWER` ioctl. Table 4-3 describes the ioctls the owmux understands.

Table 4-3 The owmux ioctls

ioctl	Function
`MUXIOCSETLOWER`	Set the device to send all unrecognized ioctls too. The argument is a pointer to an `int` containing the index of the device that is to receive all further unrecognized ioctls.
`MUXIOCGETLOWER`	Get the index of the device that is currently receiving all unrecognized ioctls. The argument is a pointer to an `int`. The index of the device currently receiving unrecognized ioctls is stored in the int pointed to by the argument.
`I_LINK`	Link the stream from the specified device under the multiplexor. The argument is a pointer to an `int`. The device whose file descriptor is pointed to by the argument is linked under the multiplexor. The `I_LINK` ioctl returns an index that is used as the argument of the three other ioctls to refer to the newly linked device.
`I_UNLINK`	Unlink the stream from the specified device under the multiplexor. The argument is a pointer to an `int`. The index of the device to be unlinked is stored in the `int` pointed to by the argument.

The owmux interface (private) in the server is responsible for opening the owmux and linking and unlinking devices in the following situations:

- the server opens the owmux in the `RegisterMuxIo()` function
- the server links a device's stream under the multiplexor in the `LinkDevice()` function

- the server unlinks a device's stream from the multiplexor in the `UnlinkDevice()` function

The only time the owmux interface (public) in the device library references the owmux is when it must send an `ioctl` to the device being controlled by the library.

Note – To see sample source code, see the section "Sample Tablet Device Driver Using the X Input Extension" in Appendix C titled "OpenWindows Source Code."

Font Service and Scalable Fonts

The X11/NeWS server has always offered scalable fonts. A Type 1 font scaler has been incorporated into Solaris 2.x as well. Therefore, you can use F3 format, Type 1, Type 3 (PostScript), and Bitmap fonts in any applications you develop.

Bitmap Fonts

Each size-specific bitmap font is provided in a separate file. The format of these bitmap font files can be any of several industry standard formats. All of the bitmap fonts shipped with OpenWindows "Next Generation server" will be in the Portable Compile Format which has a filename extension of `.pcf`. For smaller sizes, some scalable fonts are provided as hand-tuned bitmaps. The hand-tuned bitmaps provide superior results and are used, if available, rather than scaling the desired font from an outline representation.

Note – Some of the binary bitmap file formats cannot be shared between machines of different architectures, as shown in Table 4-4.

Table 4-4 Bitmap File Formats

File Format	File extension	Binary	Architecture-specific
Bitmap Distribution Format	`.bdf`	No	No
Portable Compiled Format	`.pcf`	Yes	No
Server Natural Font format	`.snf`	Yes	Yes
OpenWindows Font Bitmap	`.fb`	Yes	Yes

Outline Fonts

Each outline font is provided in one of four outline font formats:

- OpenFonts™ F3 format scalable outline fonts are stored in files with the `.f3b` extension

- PostScript® Type 3 fonts are stored in files with the `.ps` extension. Because the outlines for these fonts are described by arbitrary PostScript language procedures, these fonts are only visible to DPS applications.
- Adobe Type™ 1 font format files have the `.pfa` or `.pfb` extension.
- Bitstream Speedo™ font format files have the `.spd` extension.

This means a single outline font can be arbitrarily scaled to a variety of sizes, thus replacing a whole family of bitmap files. The "Next Generation" server supplies a core set of 57 F3 fonts, 8 Speedo fonts, and either 8 or 35 PostScript fonts from Adobe.

The outline fonts can be used to generate arbitrarily transformed and rotated characters from DPS. X11 programs use these fonts to get characters that are arbitrarily scaled, but are not rotated. When you use a character from an outline font for the first time, the bitmap must be created; then it can be cached and reused. If initial display-time for previously undisplayed characters is critical, a bitmap font may be preferable.

With X11 Release 5, the X11 server allows for font scaling in a network environment by obtaining font information from a font server running on the network. The font server parses font files of any of a number of supported formats and outputs them in a standardized X Font Service Protocol bitmap format. Although X servers using font service no longer need to parse fonts, most X terminals and workstation-based X servers will still support file-based fonts.

X11 Release 5 also includes font scaling, allowing you to obtain fonts at any point size and resolution without jagged edges. The X server contains algorithms to scale bitmap fonts, Bitstream Speedo fonts, Type 1 fonts, or OpenFonts F3 format fonts.

While the capability of having a font server is nice, it is not necessary. The server itself has the capability to do anything the font server does. OpenWindows Version 3.x contains the X11 "Next Generation" server with MIT's font scaling capabilities, as well as the value-added OpenFont scaler and Type 1 scaler.

The "Next Generation" server will have Type 1 fonts scaled by the Adobe Scaler provided with DPS. Although the OpenWindows Version 3.x server is capable of working with a font server, OpenWindows Version 3.x does not come with a font server. The fonts discussed in the following sections are included in the X11 Next Generation server.

X11 can access the OpenFonts and Type 1 fonts scaling technology through the X Logical Font Description (XLFD) protocol. The XLFD gives numeric values for pixel size, point size, resolution, and average width. Since these variables have an indefinite range of values for a scalable font, these fonts are reported—through `xlsfonts` or `XListFonts()` Xlib requests—as having 0 in these fields. To use them, take the reported name and replace the pixel size or point size field with the desired value. If pointsize and pixel size and Y resolution are all supplied, then the values must be consistent with each other.

Note – For the server to parse these values correctly, the name must be supplied with all the dashes intact. Using wildcards to leave out sections of the name fails, though wildcards are safe to use for individual fields in the name.

Alternatively, the fonts can be accessed by their PostScript language names, with the point size appended to the name (for example, Times-Roman-23). This method works for any scalable font. Note that this is not a portable way of specifying fonts and should be avoided when writing portable code. This font specification is much easier to type and remember than an XLFD format and may be preferable as a command line argument or `.Xdefaults` entry.

Note – This form of font naming will continue to be supported in the “Next Generation” server, but it is recommended that you use XLFD format.

Type 1 Fonts can be installed in the server by typing:

```
xset +fp <dir>
```

where `dir` is the name of the directory containing Type 1 font files; these files have the extension `.pfa` or `.pfb`.

MIT X11 Release 5 compliant servers can work with the font server. To add a font server to the path that the X server searches, type:

```
xset fp
```

When an X server is named, a font server running on a TCP/IP network uses the naming convention

```
tcp/host:port[/directories]
```

where `directories` is an optional parameter that specifies a list of directory names separated by plus signs. For example, a font path containing a TCP server named `ora` might look like this:

```
tcp/ora:7000
tcp/ora:7000/75dpi, /usr/lib/X11/fonts/misc
tcp/ora:7000/100dpi+misc, /usr/lib/X11/fonts/misc
tcp/ora:7000/100dpi+misc, tcp/ora:7001/new-fonts/misc
```

Note – The above example is taken from the *Programmer’s Supplement for Release 5*, by David Flanagan (O’Reilly & Associates, Inc., 1991).

The X server port number is a relative offset from a hard-coded preassigned base port number. Conversely, a font server name must specify an absolute port number. The default font server port number is 7000, but this can be changed using a command-line option if it conflicts with other services or font servers.

Selecting Scalable Fonts

The addition of scalable fonts to the X Window System changes the way you address the Xlib function `XListFonts()`. Specifically, you need to ensure backwards compatibility with older applications. In earlier releases of the X Window System, you could insert wildcards for many of the fields in the font name, such as:

```
-*-times-bold-r-*-*-*-120-*
```

When `XListFonts()` comes across a scalable font, it returns the string `0` (the number zero) in the `PIXEL_SIZE` (position seven), `POINT_SIZE` (position eight), and `AVERAGE_WIDTH` (position twelve) fields. With X11 Release 5, the X server and font server must be passed a well-formed font that contains all 14 hyphens specified in the XLFD convention. You may use wildcards in place of any field, but you may not use wildcards in lieu of hyphens. The well-formed version of the previous font name is:

```
-*-times-bold-r-*-*-*-120-75-75-*-*-iso8859-1
```

If `XListFonts()` is passed a font name that is not well-formed, it might not search for scalable fonts.

To have `XListFonts()` search exclusively for scalable fonts, insert the string "`0`" in fields seven, eight, and twelve in the font name.

Deriving Instances of Scalable Fonts

Applications that use a large number of fonts, such as text processors or graphics applications, may want to take advantage of scalable fonts. These kinds of applications need to be modified to recognize and utilize scalable fonts. Note that you must always give `XListFonts()` a well-formed and correctly specified font name.

Code Example 4-9 shows a procedure that determines whether a font name returned by `XListFonts()` represents a scalable font. This procedure is intended to be called each time a font is returned by XListFont`()`.

Note – Code Examples 4-9 and 4-10 are taken from the *Programmer's Supplement for Release 5*, by David Flanagan (O'Reilly & Associates, Inc., 1991).

Code Example 4-9 Procedure to determine if a font is scalable

```
/*
 * This routine returns True only if the passed name is a
 * well-formed XLFD style font name with a pixel size, point
 * size, and average width (fields 7,8, and 12) of "0".
 */
Bool IsScalableFont(name)
char *name;
{
    int i, field;

    if ((name == NULL) || (name[0] != '-')) return False;

    for(i = field = 0; name[i] != '\0' && field <= 14; i++) {
        if (name[i] == '-') {
          field++;
          if ((field == 7) || (field == 8) || (field == 12))
                if ((name[i+1] != '0') || (name[i+2] != '-'))
                    return False;
        }
    }

    if (field != 14) return False;
    else return True;
}
```

In Code Example 4-10, the procedure takes a scalable font name and a specified point size and loads the returned font at the requested size and at the screen's precise resolution. This procedure is intended to be called with a scalable font name as returned by `XListFonts()`.

Code Example 4-10 Procedure to load a retrieved scalable font

```
/*
 * This routine is passed a scalable font name and a point
 * size. It returns an XFontStruct for the given font scaled
 * to the specified size and the exact resolution of the
 * screen. The font name is assumed to be a well-formed XLFD
 * name, and to have pixel size, point size, and average
 * width fields of "0" and implementation dependent x-
 * resolution and y- resolution fields.  Size is specified
 * in tenths of points. Returns NULL if the name is malformed
 * or no such font exists.
 */
XFontStruct *LoadQueryScalableFont(dpy, screen, name, size)
```

Code Example 4-10 Procedure to load a retrieved scalable font (Continued)

```
Display *dpy;
int screen;
char *name;
int size;
{
    int i,j, field;
    char newname[500]; /* big enough for a long font name */
    int res_x, res_y; /* resolution values for this screen */

    /* catch obvious errors */
    if ((name == NULL) || (name[0] != '-')) return NULL;

    /* calculate our screen resolution in dots per inch. 25.4mm = 1 inch */
    res_x = DisplayWidth(dpy, screen)/(DisplayWidthMM(dpy, screen)/25.4);
    res_y = DisplayHeight(dpy, screen)/(DisplayHeightMM(dpy, screen)/25.4);

    /* copy the font name, changing the scalable fields as we do so */
    for(i = j = field = 0; name[i] != '\0' && field <= 14; i++) {
       newname[j++] = name[i];
       if (name[i] == '-') {
           field++;
        switch(field) {
            case 7:  /* pixel size */
            case 12: /* average width */
           /* change from "-0-" to "-*-" */
               newname[j] = '*';
               j++;
               if (name[i+1] != '\0') i++;
               break;
           case 8:  /* point size */
           /* change from "-0-" to "-<size>-" */
               (void)sprintf(&newname[j], "%d", size);
               while (newname[j] != '\0') j++;
               if (name[i+1] != '\0') i++;
               break;
         case 9:  /* x resolution */
          case 10: /* y resolution */
           /* change from an unspecified resolution to res_x or res_y */
           (void)sprintf(&newname[j], "%d", (field == 9) ? res_x : res_y);
           while(newname[j] != '\0') j++;
           while((name[i+1] != '-') && (name[i+1] != '\0')) i++;
           break;
         }
       }
    }
```

Code Example 4-10 Procedure to load a retrieved scalable font (Continued)

```
    newname[j] = '\0';

    /* if there aren't 14 hyphens, it isn't a well formed name */
    if (field != 14) return NULL;

    return XLoadQueryFont(dpy, newname);
}
```

New Server Options

- `-unsecure`
  With DPS extensions, any application that is allowed to connect to the X Display can also open and write files that the owner of the X Display has permissions for. This is done with PostScript file operators. This leaves a big security hole in the server. Therefore, by default the server will not allow file operators to function. The end user will have to run the server in unsecure mode to get access to the file system.

Note – `-unsecure` is a placeholder for a server option that will be introduced in the new server do implement this functionality. The option may be renamed.

- `-dev`
  This is the frame buffer name of the device to open. There are device options that can be specified with the server as well.

The device options are:

`[left │ right │ top │ bottom]`	orientation relative to previous screen in a two-monitor configuration.
`dpix #`	dots-per-inch in the x direction
`dpiy #`	dots-per-inch in the y direction

`defclass [GrayScale │`	visual to report as default
	`StaticGray │ PseudoColor │`
	`StaticColor │ DirectColor │`
	`TrueColor]`

`defdepth #`	report only `GrayScale` and/or `StaticGray` visuals
`grayvis`	report only `GrayScale` and/or `StaticGray` visuals

Note – The device options have been added to the server to remain compatible with the X11/NeWS server as much as possible.

Other options new to the server are:

`-cp dir1[:dir2...]`	search path for configuration files and dynamically loaded modules
`-flipPixels`	reverse black and white pixel values
`-pit`	set bell pitch
`-dur`	set bell duration
`-mden`	set pointer acceleration denominator
`-keyboarddev`	specify keyboard device
`-pointerdev`	specify pointer device

For more information, consult the man page for `Xsun`.

Enabling Technology

Enabling technology is technology that contains access solutions for people with physical, visual, and hearing disabilities. In the past, workstations have been deficient in providing access technology. The current technology has not enabled handicapped access. Issues involving ease-of-use impact everyone. Access solutions, such as voice recognition and font size control capabilities, are being marketed on other platforms as productivity boosters for the non-handicapped computer user.

Enabling technology increases system access, system control, and overall task productivity for everyone. The traditional name for this has been *adaptive technology*, but the problem with this term is that the word "adaptive" implies that this technology applies only to users with disabilities. The name "enabling technology" has been chosen because enabling someone implies increased productivity and usability. In other words, everyone benefits from enabling technology.

With OpenWindows Version 4, Sun is attempting to solve the problem of accessibility for all users. The following three enabling technology features are accessed via command-line from OpenWindows Version 4:

- T-Lock (also known as "Sticky Keys")
  Allows you to press modifier keys (Shift, Control) before the keys they modify, rather than requiring that they be pressed simultaneously.

  When T-Lock is enabled, the modifier keys (Shift, Control, Meta, and Alt) will retain state as follows. When a modifier key is pressed and released, it "latches". A latched modifier key will modify the next non-modifier key as though it had been pressed and held normally. When a modifier key is pressed and released twice in a row, it "locks". A locked modifier key will modify any non-modifier key until it is unlocked. When the locked modifier key is pressed a third time it unlocks.

- Slow Keys
  Delays the recognition of keypress events so that users with limited coordination can accidentally bump keys without sending keypress events.

  When the Slow Keys option is enabled, keystroke events are filtered as follows: when a key or keys are pressed then released within the time specified as the Keystroke Recognition Delay, both the keypress(s) and the keyrelease(s) are ignored. When multiple keys are pressed and all but one are released prior to the Keystroke Recognition Delay time only the key press held after this time is recognized as a keystroke event.

- Repeat Keys
  Delays the onset of keys repeating so that users with limited coordination have time to release keys before they start sending multiple characters.

  Enabling and disabling the key repeating is already supported by the servers and by the workspace properties panel. However, the key repeating functionality would be modified to allow the following configurations:
  - The Key Repeat Delay specifies the length of time between the first character being transmitted and the onset of key repeating (that is, the transmission of the second character)
  - The Key Repeat Rate specifies the delay between the character transmissions once repeating starts

The man page for `Xsun` contains further information on these options.

Display PostScript System Extension (DPS)

In the last quarter of 1992, SunSoft, Inc. and Adobe Systems Incorporated made a technology agreement to integrate DPS with SunSoft's new X11 Release 5-compliant server. DPS is implemented as an extension to the X11 server. It allows for clients to do PostScript imaging on X window objects. The following are features of DPS:

- Level II compliant DPS extension
- access to Folio Fonts from DPS using `/findfont` operator
- `pswrap` utility to binary encode PostScript code for better performance
- function calls for all PostScript operators for access from C source code
- access to DPS from any X toolkit and Xlib
- support for DPS toolkit (`libdpstk`)
- support for `libpsres`
- memory management including memory reclamation
- composite fonts for Asian Language support
- ability to cache user-defined paths for fast rendering
- support for generating and storing forms
- device-independent color support (CIE)

- improved halftones

For additional information, you can contact the Adobe Developer's Association at (415) 961-4111.

You can also tap into their file server for information: **ps-file-server@adobe.com**

See Chapter 5 on “Graphics” in this book, or the following books for information on the DPS API. These books are available from the MIT X11 Release 5 distribution.

An Overview of the Display PostScript System, Adobe Systems Incorporated, Mountain View, CA.

X Window Systems Programmer’s Supplement to the Client Library Reference Manual, Adobe Systems Incorporated, Mountain View, CA.

pswrap Reference Manual, Adobe Systems Incorporated, Mountain View, CA.

Perspective for Software Developers, Adobe Systems Incorporated, Mountain View, CA.

Color Extensions, Adobe Systems Incorporated, Mountain View, CA.

Client Library Reference Manual, Adobe Systems Incorporated, Mountain View, CA.

Extensions for the Display PostScript System, Adobe Systems Incorporated, Mountain View, CA.

The following book is available as of March, 1993, from Adobe Systems Incorporated:
Display PostScript Programmer’s Guide, Adobe Systems Incorporated, Mountain View, CA.

Internationalization Support in the Server

This section is being written as OpenWindows Version 4 is developed. Since it will be compliant with Release 5 of MIT, we can examine the internationalization support in the server as available from MIT and be fairly certain that it will not be very different. There may be SunSoft value-added features; these will be optional.

Note – The X Input Method server may differ from the MIT distribution example server.

An *internationalized* application is one that runs, without changes to the binary, in any specified locale. The largest new part of Solaris 2.x is the support for writing internationalized programs.

Solaris 2.x has the ANSI-C internationalization features; the C library reads a “localization database” customized for each locale. However, in the U. S., Solaris 2.x is shipped with the default C locale database.

Because the word “internationalization” contains 20 letters, it is often abbreviated as I18N.

Solaris 2.x supports the X input method (`Ximp`) protocol implementation of I18N.

To support internationalization, locale management functions and routines for handling text drawing, such as font set handling, metric, and string measuring functions, and string drawing functions, have been added to Solaris 2.x.

Many Xlib functions have been respecified to clarify their handling of strings in internationalized applications. Functions have been added to support internationalized window manager and text properties for inter-client communication, and new resource manager functions exist to support local databases.

In addition, many resource manager functions have been respecified to clarify their behavior in internationalized applications. There is also new support for internationalized text input.

For further information in this document on I18N, Chapter 7, "Internationalization," explains some basic concepts of I18N, as well as ANSI-C I18N.

In addition, the bibliography at the end of this chapter has sources which contain detailed explanations on character encoding. This may help you to understand how character sets are represented in different languages.

Using X11 Release 5 to Internationalize

The internationalization techniques discussed here have been sufficient only to internationalize terminal-based applications. Many applications use multibyte strings in an encoding that contains shift sequences and nonuniform width characters. Some X clients need to communicate with other clients. Therefore, an internationalized interclient communication method is required because internationalized clients can run in different locales. Other X clients make heavy use of resource files and databases; these clients need a mechanism for determining the correct localization of resources.

The function `setlocale` by itself is insufficient for locale management in an X application. Xlib contains two new functions which are used with `setlocale` when an X application starts up. Changes to the Xt architecture are also needed to support the new internationalization features of Xlib.

The following describes some of the internationalization changes and issues in X11 Release:

- **Internationalized Text Output in X11 Release 5:** The new text output routines are based on an Xlib extraction called the `XFontSet()`, which contains all the fonts needed to display text in the locale for which it was developed. See *Programmer's Supplement For Release 5 of the X Window System, Release 11*, for more information.

Note – X11 Release 5 does not have provisions for text displayed from right to left, such as Hebrew or Arabic.

- **String Encoding Changes for Internationalization**
  The encoding of strings is a major concern regarding internationalizing X applications. An X client must be able to communicate with the X server, and often with a window manager, a session manager, and other clients. There are a variety of ways to encode colors, strings, bitmap files, resource names and values. There are also many ways to pass names to the X server and to copy strings selected in one client to another. Strings can be in the encoding of the locale.

- **Internationalized Interclient Communication**
  X11 Release 5 provides functions that do not assume internationalized interclient communication will be done with ASCII strings.

- **Localization of Resource Databases**
  Sometimes methods of encoding data are implementation-dependent. For example, a Japanese-speaking end user who wishes to customize an application written by an English-speaking programmer may use symbolic representations of English words meaningless to another user.

- **Internationalized Text Input**
  Unlike a program that only runs in one language, an internationalized program cannot assume a one-to-one correspondence between the keyboard and input characters. For example, the keyboard correspondence between keystrokes and characters is much more complex in Greek than in English. When the codeset of a locale contains more characters than keystrokes, an *input method* is required to map keystrokes to characters. Solaris 2.x supports the X Input Method (XIM) and X Input Context (XIC) keyboard input abstractions, as well as the functions `XmbLookupString` and `XwcLookupString`; these functions return a string in the encoding of the specified locale.

The XIM designers envision their I18N text input mechanism will be incorporated within toolkits and thus hidden from developers. It is therefore strongly recommended you use the OLIT toolkit for your I18N interface.

Note – For more information on XIM and X11 Release 5 internationalization, see the *Programmer's Supplement for Release 5 of the X Window System, Version 11*, by David Flanagan (O'Reilly & Associates, Inc., 1991).

The X Color Management System (Xcms)

The X Color Management System (Xcms) is a device-independent color application programming interface (API). Xcms is a set of Xlib functions that allow you to allocate device-independent colors. Xcms provides precise control over the allocation of colors. Xcms also comes with a new syntax for the color strings used throughout Xlib.

The following components of X11 Release 5 are new:

- syntax for device-independent color strings
- Xlib provides a mechanism for device-independent string names for colors
- `xcmsdb`, a database read by Xlib that maps color names to device-independent color specifications
- the Xcms API
- the X Device Color Characterization Conventions (XDCCC)

Before you proceed with how to use Xcms API, it is important to understand the fundamentals. While it is not the focus of this chapter, consult the bibliography to get more information on the science of colorimetry.

When you use a device-independent color specification, the resulting display has identical colors on all devices. X11 Release 5 supports an international standard known as CIEXYZ. In this format, as is the case with RGB, a color is described by the value of three coordinates; the associated color space is usually referred to by those coordinates.

Solaris supports the CIEXYZ color space, the related spaces known as CIExyY, CIEuvY, CIELuv, CIELab, and the Tektronix-designed color space, TekHVC. TekHVC is a perceptually uniform color space designed to be easy to understand; it is mathematically related to the CIE spaces. The TekHVC colors are represented by Hue, Value, and Chroma. *Hue* is what distinguishes one color family from another—red, blue, or green. *Value* describes how light or dark a color is. *Chroma* describes the color saturation. Together, the HVC values describe a color solid.

The Client Color Name Database

Although the X client supports device-independent colors, the X protocol and X server still only use device-dependent RGB colors. Therefore, you cannot use device-independent color specifications in the database of color names read by the server.

You can give symbolic names to device-independent colors, and Solaris supports a client-side color database that maps color names to device-independent or device-dependent color specifications.

While Solaris supports this color database, the MIT release does not provide one. The file `$OPENWINHOME/lib/X11/Xcms.txt` contains the client-side database X clients use by default. To specify a different file, set the `XCMSDB` environment variable.

X Device Color Characterization

The X Device Color Characterization Conventions (XDCCC) specify how Xcms converts from device-independent color used by X clients to the device-dependent colors used by the X server.

Included in X11 Release 5 is a new client, `xcmsdb`.

Device-Independent Color API

The X Color Management System (`Xcms`) API is part of Xlib. It contains many new functions and data types, all of which are prefixed by `Xcms`. Many of these functions are similar to earlier X functions. A group of `Xcms` functions are used to manipulate the *color conversion context* (CCC) `Xcms` datatype. The CCC contains attributes that control the details of color conversion between color spaces. `Xcms` automatically handles default CCCs.

What follows is a brief description and introduction to the `Xcms` API. For more information about `Xcms`, refer to the resources listed at the end of this chapter.

`Xcms` functions require more computation than device-dependent functions because the former perform trigonometric and other floating-point operations. These floating-point operations are directly implemented in Xlib, and they are relatively slow because they do not have a highly-optimized math library and cannot take advantage of floating-point hardware. Therefore, adding `Xcms` functions to a program can make it run slower, especially during gamut compression.

Note – For more information, see the *Programmer's Supplement for Release 5 of the X Window System, Version 11*, by David Flanagan (O'Reilly & Associates, Inc., 1991).

The Xcms Functions

The `XcmsColor` structure is a device-independent version of the `XColor` structure; it is used by the following `Xcms` functions. `XcmsColor` contains a pixel value, a format value denoting the colorspace used to describe the color, and a union of structures that specify parameters for all formats supported by Solaris 2.x.

The following functions are provided within `Xcms` for manipulating color map cells. Although they use the `XcmsColor` structure to specify device-independent color, they are similar to device-dependent color allocation functions.

- `XcmsAllocColor`
  This function is similar to `XAllocColor()` except the color can be specified in any format. The `XcmsAllocColor()` function ultimately calls `XAllocColor()` to allocate a read-only color cell (color map entry) with the specified color. It returns the pixel value of the color cell and the color specification allocated.

- `XcmsAllocNamedColor`
  This function is similar to `XAllocNamedColor()` except the color returned can be in any format specified. `XcmsAllocNamedColor()` ultimately calls `XAllocColor()` to allocate a readonly color cell with the color specified by a color string. It returns both the color specification (exact specification) and the actual color specification (screen specification) stored.

- `XcmsLookupColor`
  This function looks up the string name of a color with respect to the screen associated with the specified color map. It returns both the exact color values and the closest values provided by the screen with respect to the visual type of the specified color map.This function performs gamut compression. It is similar to `XLookupColor()`, but has an argument to specify the desired color space for the return values, and returns a device-independent color specification.

- `XcmsQueryColor`
  This function obtains the RGB value for the pixel value in the pixel member of the specified `XcmsColor` structure, and then converts the value to the target format as specified by the `result_format` argument.

5. `XcmsQueryColors`
   This function obtains the RGB values to the target format as specified by the `result_format` argument.

- `XcmsStoreColor`
  This function converts the color specified in the `XcmsColor` structure into RGB values and then uses this RGB specification in an `XColor` structure, whose three flags (`DoRed`, `DoGreen`, and `DoBlue`) are set, in a call to `XStoreColor()`, to change the color cell specified by the pixel member of the `XcmsColor` structure. This pixel value must be a valid index for the specified color map, and the color cell specified by the pixel value must be a read/write cell. If the color map is an installed map for its screen, the changes are visible immediately. Due to the screen's hardware limitations or gamut compression, the color stored in the color map may not be identical to the color specified.

- `XcmsStoreColors`
  This function converts the colors specified in the array of `XcmsColor` structures into RGB values and then uses these RGB specifications in `XColor` structures, whose three flags (`DoRed`, `DoGreen`, and `DoBlue`) are set, in a call to `XStoreColors()`, to change the color cells specified by the pixel member of the corresponding `XcmsColor` structure. Each pixel value must be a valid index for the specified color map, and the color cell specified by each pixel value must be a read/write cell. If the color map is an installed map for its screen, the changes are visible immediately. Due to the screen's hardware limitations or gamut compression, the colors stored in the color map may not be identical to the colors specified.

Note – When using the TekHVC color space, you should verify that the minimum and maximum Values passed to the `Xcms` functions are valid. If either of the specified Values is outside the TekHVC colorspace, gamut compression takes place.

Color Conversion Context

The *color conversion context* (CCC) is a structure of type `XcmsCCC`. In X11 Release 5, a CCC is automatically associated with each color map. The CCC contains the attributes controlling the details of color conversion from one space to another, such as the procedure that is called to perform gamut compression when a device-independent color specification is outside the range of displayable colors for the specified device.

Gamut Querying Functions

This section describes the gamut querying functions that are provided by Xlib. These functions allow the client to query the boundary of the screen's color gamut in terms of the CIE L*a*b*, CIE L*u*v*, and TekHVC color spaces.

Gamut querying functions are also provided that allow you to query the color specification of:

- white (full intensity red, green, and blue)
- red (full intensity red while green and blue are zero)
- green (full intensity green while red and blue are zero)
- blue (full intensity blue while red and green are zero)
- black (zero intensity red, green, and blue)

The white point associated with color specifications passed to and returned from these gamut querying functions are assumed to be the Screen White Point. This is a reasonable assumption, because the client is trying to query the screen's color gamut. Note that the following naming convention is used for the "Max" and "Min" functions:

- `Xcms`<*color_space*>`QueryMax`<*dimensions*>
- `Xcms`<*color_space*>`QueryMin`<*dimensions*>

Note that the <*dimensions*> consists of letter or letters that identify the dimension or dimensions of the color space that are not fixed.

The following functions can be used to verify that requested color specifications are legal for a specified TekHVC color space:

- `XcmsTekHVCQueryMaxC`
  Use this function to obtain the maximum Chroma for a specified Hue and Value.

- `XcmsTekHVCQueryMaxV`
  Use this function to obtain the maximum Value for a given Hue and Chroma.
- `XcmsTekHVCQueryMinV`
  Use this function to obtain the minimum Value for a specified Hue and Chroma.
- `XcmsTekHVCQueryMaxVC`
  Use this function to determine, given a Hue, the maximum Chroma in TekHVC color space displayable by the screen and the Value at which that maximum Chroma is reached.
- `XcmsTekHVCQueryMaxVSamples`
  Use this function to obtain a specified number of TekHVC specifications such that they contain a maximum Value for a specified Hue and Chroma at which the maximum Values are reached.

Note – See *The X Color Management System*, by Al Tabayoyon et al for more information on functions for color spaces (O'Reilly & Associates, Inc.).

Other Changes to Window Support in X11 Release 5

The following changes to the X11 Release 5 X Window System are documented in the *Programmer's Supplement for Release 5 of the X Window System, Version 11*, by David Flanagan (O'Reilly & Associates, Inc., 1991):

- To facilitate ANSI and POSIX compliance, X11 Release 5 has new standard header files.
- Minor changes have been made to Xlib and Xt.
- The `editres` client. has been added to enable you to interactively edit the widget resources of any running application that uses the Editres protocol.
- Four Xaw widgets have been added: `Panner`, `Porthole`, `Repeater` and `Tree`. Many other Xaw widgets have been modified. The Editres protocol is supported by the Xaw `VendorShell` widget.
- Many Xmu library routines have been modified. The Editres protocol is supported by the Xmu library.
- The X11 Input Device Extension, which allows input from devices other than a keyboard or mouse, is now a standard of the X Consortium.
- The MIT Implementation of X11 Release 5 has two new secure authorization protocols: XDM-AUTHORIZATION-1 and SUN-DES-1.
- Support for Portable Compiled Fonts across architectures is provided via the *pcf* format.
- Hebrew, Kanji (Japanese), and Hangul (Korean) fonts are provided.
- Clients have been added to the MIT X11 Release 5 implementation; others have been modified or replaced.
- The MIT X server runs faster. Reimplementation of the Xt Translation Manager results in less memory use at runtime.

Note – None of these are required. They have been put here for your information and therefore can be used to enhance the capabilities of your applications.

Suggested Reading

The following books contain additional information you may find useful when programming for windows under Solaris 2.x.

X Toolkit Intrinsics Reference Manual for X11 Release 4 and Release 5, David Flanagan, O'Reilly & Associates, Inc., Sebastopol, CA.

X Windowing System: The Complete Reference to Xlib, X Protocol ICCCM, XLFD, Robert W. Sheifler et al, Digital Press, Burlington, MA.

Bibliography

American National Standard for Information Systems - Programming Language C, ANSI X3.159-1989, Published by American National Standard Institute.

Color Extensions, Adobe Systems Incorporated, Mountain View, CA.

Client Library Reference Manual, Adobe Systems Incorporated, Mountain View, CA.

Definition of the Porting Layer for the X v11 Sample Server, Susan Angebranndt, Raymond Drewry, Phillip Karlton, Todd Newman, Bon Scheifler, and Keith Packard. MIT X Consortium.

Display PostScript Programmer's Guide, Adobe Systems Incorporated, Mountain View, CA.

Extensions for the Display PostScript System, Adobe Systems Incorporated, Mountain View, CA.

OLIT 3.2 Reference Manual, Part Number 801-4246-10, SunSoft, Inc., Mountain View, CA.

OLIT Internationalization Features in Asian OpenWindows, Part Number 801-2415-10, SunSoft, Inc., Mountain View, CA.

OpenWindows Version 3.2 Programmer's Guide, Part Number 801-xxxx-xx, SunSoft, Inc., Mountain View, CA.

Programmer's Supplement for Release 5 of the X Window System, Version 11, David Flanagan, O'Reilly & Associates, Inc., Sebastopol, CA.

Strategies for Porting the X v11 Sample Server, Susan Angebranndt, Raymond Drewry, Phillip Karlton, Todd Newman, Bon Scheifler, and Keith Packard. MIT X Consortium.

The X Color Management System, Al Tabayoyon et al, O'Reilly & Associates, Sebastopol, CA.

The X Window System Programming & Applications with Xt, OPEN LOOK Edition, Douglas A. Young and John A. Pew, Prentice Hall, Englewood Cliffs, NJ.

X Server Internals: A Tutorial On the MIT Sample Server at Release 4, Keith Packard, MIT X Consortium.

X Window System, Version 11, Release 5. Release Notes, MIT X Consortium Staff, MIT Laboratory for Computer Science.

X Window System, Version 11 Input Method Specifications, Draft 3.0, October 1990, Bill McMahon.

The X Window System Server, Elias Israel and Erik Fortune, Digital Press.

X11 Input Extension Library Specification, X11 Release 5, Mark Patrick and George Sachs, MIT X Consortium, MIT Laboratory for Computer Science.

X11 Input Extension Porting Document, X11 Release 5, George Sachs, MIT X Consortium, MIT Laboratory for Computer Science.

X11 Input Extension Protocol Specification Version 1.0, Mark Patrick and George Sachs, MIT X Consortium, MIT Laboratory for Computer Science.

An Overview of the Display PostScript System, Adobe Systems Incorporated, Mountain View, CA.

X Window Systems Programmer's Supplement to the Client Library Reference Manual, Adobe Systems Incorporated, Mountain View, CA.

pswrap Reference Manual, Adobe Systems Incorporated, Mountain View, CA.

Perspective for Software Developers, Adobe Systems Incorporated, Mountain View, CA.