XVT Platform-Specific
Guide-Motif

xvt

XVT/XM

PREFACE

About This Manual

XVT takes pride in its documentation, and continually seeks to improve it. If you find a documentation error, please contact Customer Support. They will forward your suggestion to XVT's documentation team.

Conventions Used in This Manual

In this manual, the following typographic and code conventions indicate different types of information.

General Conventions

code
: This typestyle is used for code and code elements (names of functions, data types and values, attributes, options, flags, events, and so on). It also is used for environment variables and commands.

code bold
: This typestyle is used for elements that you see in the user interface of applications, such as compilers and debuggers. An arrow separates each successive level of selection that you need to make through a series of menus, e.g., **Edit=>Font=>Size**.

bold
: Bold type is used for filenames, directory names, and program names (utilities, compilers, and other executables).

italics
: Italics are used for emphasis and the names of documents.

Tip: This symbol marks the beginning of a procedure. These symbols can help you quickly locate "how-to" information.

Note: An italic heading like this marks a standard kind of information: a Note, Caution, Example, Tip, or See Also (cross-reference).

This symbol and typestyle highlight information specific to using XVT-Design, XVT's visual programming tool and code generator.

Code Conventions

<non-literal element> or non_literal_element
: Angle brackets or italics indicate a non-literal element, for which you would type a substitute.

[optional element]
: Square brackets indicate an optional element.

...
: Ellipses in data values and data types indicate that these values and types are opaque. You should *not* depend upon the actual values and data types that may be defined.

XVT/XM

CONTENTS

1

INTRODUCTION

Welcome to XVT/XM. This platform-specific book (PSB) contains information about using the latest release of the XVT Portability Toolkit (XVT/XM) on your particular platform. If you had an earlier version of XVT/XM, this manual replaces the previous platform-specific book.

Installing the XVT Development Solution for C for Motif gives instructions for installing XVT/XM. Once you have XVT/XM installed, XVT recommends that you read this book and try the sample programs that come with the product.

Note: Before writing your application, read the *XVT Portability Toolkit Guide*. The *Guide* focuses on strategies for developing portable applications.

See Also: For an alphabetical listing of all the XVT functions and other API elements, refer to the online *XVT Portability Toolkit Reference*. For additional information not documented in this platform-specific book, see the **readme** file in the **doc** directory.

1.1. Changes to Existing Features

1.2. Compilers Supported by XVT/XM

XVT/XM supports the following platforms and compilers:

- AIX with the VisualAge C++ compiler
- Linux (Red Hat) with the GNU C++ compiler
- HPUX with the aC++ compiler
- Solaris with the Sun One Studio compiler

See Also: Changes to compiler support may be listed in the **readme** file in the **doc** directory.

1.3. Viewing the Online XVT Portability Toolkit Reference

The online *XVT Portability Toolkit Reference* can be accessed by changing to the **bin** directory and entering the following command:

```
helpview
```

When the portable XVT help viewer appears, a dialog will prompt you for a help file. Select the file **refman.csc** from the **doc** directory.

1.4. XVT Implementations and Operating Systems

The XVT library is currently available for several different window systems and operating systems:

XVT Product:	Window Systems:	Operating Systems:
XVT/Mac	MacOS	MacOS
XVT/Win32/64	Win32/64	Windows (all)
		Windows (all)
XVT/XM	X and Motif	UNIX

2

USING XVT/XM

2.1. Introduction

This chapter addresses various platform-specific issues that you may need to consider while using XVT/XM. The information here assumes you are familiar with developing Motif applications from a general standpoint. If not, see the *OSF/Motif Programmer's Guide* and the *OSF/Motif Programmer's Reference* for more information.

2.2. Extensibility

2.2.1. Conditional Compilation

If, in your application, you need to provide some native-platform GUI functionality not available in the XVT Portability Toolkit, then the small percentage of your code that provides that functionality will be non-portable. In this case, you must compile your code conditionally, based on the compilers, the window systems, and the operating systems on which your non-portable code will run.

The XVT Portability Toolkit automatically determines the environment in which the application is compiled.

See Also: For more information on conditional compilation, see the "Symbols for Conditional Compilation" section in the "About the XVT API" chapter in the *XVT Portability Toolkit Guide*, and the file **xvt_env.h** in your **include** directory.

Tip: It's best to consolidate any non-portable code into a few separate files so that most of your application will be portable XVT code. Separating your non-portable code makes it easier to change your program when the capability you need is added to a future version of XVT.

Tip: To compile Motif-specific code conditionally:

Use the following preprocessor statements to compile window system-specific code:

```
#if XVTWS == MTFWS
/* window-system-specific code goes here */
#endif
```

Use the following preprocessor statements to compile file system-specific code:

```
#if    XVT_FILESYS_UNIX
       /* UNIX file-system-specific code here */
#endif
```

2.2.2. Accessing Window Device Contexts and Handles

Given an XVT WINDOW, your application can access its native window handle (X type, Window) or graphics context (GC).

Tip: To get the X Window associated with an XVT WINDOW (excluding windows of type W_PIXMAP, W_PRINT and W_SCREEN):

Call:

```
(Window) xvt_vobj_get_attr(win,
                                   ATTR_NATIVE_WINDOW)
```

To get the X graphics context associated with an XVT WINDOW (includes only drawable windows of type W_DOC, W_PLAIN, W_DBL, W_MODAL, WTASK if drawable, and W_NO_BORDER):

Call:

```
(GC) xvt_vobj_get_attr(win,
                                 ATTR_NATIVE_GRAPHIC_CONTEXT)
```

2.3. Invoking an Input Method Editor

An Input Method Editor (IME) is provided by Motif to allow application users to enter multibyte or other non-ASCII characters from a keyboard that does not support these characters. On UNIX, users may select an environment variable for the language.

Character events are sent at appropriate times as determined by the IME. If the user composes a character, a character event is sent *only*

after the conversion—only the composed character is sent in the event. This means that in some cases there may be a delay between when characters are typed and when your application receives an event. Several characters may be typed before any character event is received.

To start the IME on Motif platforms requires some special setup (this setup is not related to XVT requirements). The following sections describe some suggested setup steps—the actual steps may vary for each platform.

HP-UX

Tip: To use an IME under HP-UX, follow these steps:

1. Start the **kks input server** as part of the system startup by entering the following command lines:

```
# Set the executable pathname
setenv PATH "/usr/lib/nlio/bin:${PATH}"
#
# Set the kks input server port (look in
# /etc/services for the proper port). For example:
kks 6897/tcp
```

2. For each shell in which you need to execute an Input Method Editor, set the LANG environment variable for the locale and character codeset. For example, to set the locale and character codeset for Japanese (EUC):

```
setenv LANG japanese.euc
```

3. For each shell, enter the command that starts the IME:

```
ximsstart -env -shell csh
```

4. Execute your XVT application from the shell.

Users invoke the IME by pressing the Kanji key (the key just to the left of the space bar). The IME appears as an entry field just below the shell window that has keyboard focus.

IBM AIX

Normally, the AIX server requires no special startup for users wishing to invoke an Input Method Editor.

Tip: To set up an **aixterm** window to use an IME, follow these steps:

1. Set the LANG environment variable for the locale and character codeset. For example, to set the locale and character codeset for Japanese (Shift-JIS):

   setenv LANG ja_JP

2. Enter a command to start the **aixterm** window:

   aixterm &

3. Enter a command to remap the keys on the keyboard for Shift-JIS input:

   xmodmap *keyboard_file*

   where *keyboard_file* is the file that contains the desired key bindings. The file *keyboard_file* also defines a key or sequence of keys that enables and disables keyboard input using a particular codeset. Enter man xmodmap or check your system documentation for more details on how to remap the keyboard.

4. From the **aixterm** window, start your application.

XVT invokes the IME automatically when users type characters into input fields or text edit objects. The IME appears as a thin window across the bottom of the screen.

Sun SPARC Solaris

Tip: To use an IME under Solaris, follow these steps:

1. Configure the X11 server as part of the system startup by entering the following command lines:

```
# Set font paths for server
xset +fp $OPENWINHOME/lib/locale/ja/X11/fonts/
75dpi,$OPENWINHOME/lib/locale/ja/X11/fonts/
F3,$OPENWINHOME/lib/locale/ja/X11/fonts/
F3bitmaps
#
# Rebuild font directory cache after setting fonts
xset fp rehash
#
# Start the IME server
htt &
```

2. For each shell in which you need to execute an Input Method Editor, set the LANG environment variable for the locale and character codeset. For example, to set the locale and character codeset for Japanese (EUC):

```
setenv LANG ja
```

3. Execute your XVT application from the shell.

Users invoke the IME by simultaneously pressing the Control and Space keys. The IME appears as an entry field just below the shell window that has keyboard focus.

2.4. XVT/XM Resource Specifics

If you use XVT-Design, you probably won't need to code native resources directly. XVT-Design and the ***xrc*** *compiler code resources automatically. (XVT-Design can be configured to invoke* ***xrc*** *for you, either directly as part of the code generation process or via a generated makefile.) The information here is provided for reference purposes only.*

This section provides information on using URL (XVT's Universal Resource Language) with XVT/XM. It also tells you how to code XVT/XM-specific resources.

See Also: Before creating any Motif-specific resources, see the "Resources and URL" chapter in the *XVT Portability Toolkit Guide*.

2.4.1. Creating Portable Resources with URL

The XVT-Design tag SPCL:User_Url *in the Action Code Editor (ACE) lets you add platform-specific resources.*

Motif's native resource language is User Interface Language (UIL). The **xrc** compiler produces a UIL script file, which the Motif UIL compiler uses to produce the binary resource UID file. The process is analogous to coding a program in C and compiling it into object code.

Tip: XVT recommends that you use **xrc** to generate the UIL resource file. If necessary, you can then change the UIL file to add Motif-specific resources. However, you should embed such resources in the URL script using the #transparent statement.

You can code all menus, dialogs, and strings in URL. Or, you can create them directly in UIL. If you decide to code them in UIL, study the output of **xrc** (in UIL format) for the resources in the XVT Example Set (**../design/examples**) to see how they are coded.

Some resources, such as file selection and error dialog boxes, *must* be present in the UID file for XVT/XM to work properly.

Resource File Location

At runtime, XVT/XM applications look for resources in a file named **your_app.uid**. In the filename **your_app.uid**, **your_app** matches the base_appl_name field of the XVT_CONFIG structure.

To find **your_app.uid**, the application looks in the current directory unless you have set the Motif environment variable UIDPATH on UNIX.

See Also: See the document *Installing XVT Development Solution for C for Motif* for more information on setting environment variables or logical names.

2.4.2. Cursors and Drawn Icons

You cannot define icons (needed by xvt_dwin_draw_icon) or cursors (needed by xvt_win_set_cursor) directly in URL. Create your icons and/or cursors using **bitmap**, the X Window System icon editor. The **bitmap** editor generates a file containing the image of an icon, in the form of C definitions and declarations.

Example: If you create an icon in a file named **icon1**, the file will contain text similar to this:

```
#define icon1_width 16
#define icon1_height 16
static char icon1_bits[] = {0x40,0x00,0xf0,0x03,0x50,0x00,
                            0x50,0x00,0x50,0x00,0xf0,0x03,0x40,0x02,
                            0x40,0x02,0x40,0x02,0x40,0x02,0xf0,0x03,
                            0x40,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                            0x00,0x00};
```

Do *not* include this file in your portable XVT source files. Instead, include it in a special file you write yourself, called a *resource manager* file, which is used only in the Motif version of XVT. The resource manager file gives XVT access to your icons and cursors by numeric ID.

Example: If your XVT/XM application were named **sample.c**, you might call your resource manager file **rmsample.c.** That file should contain only the declarations and statements discussed below.

You'll compile your resource manager file (e.g., **rmsample.c**) normally and link it with the rest of your XVT application.

Using XVT-Design, you can specify a resource manager file in the Extended Files window. The XVT-Design-generated makefile builds and links in the resource file manager.

2.4.2.1. Sample Resource Manager File (rmsample.c)

This example shows a complete resource manager file:

```
#include "xvt.h"                        /* standard XVT header */
#include "xvt_xres.h"                   /* needed for all rm*.c files */
#include "sample.h"                     /* this app's resource IDs */

#include "icon1"                        /* created with bitmap editor */
#include "icon2"
#include "cur1"
#include "cur2"

static ICON_RESOURCE icon1, icon2;              /* icon objects */
static CURSOR_RESOURCE cur1, cur2;              /* cursor objects */

RESOURCE_INFO rtable[] = {                  /* resource table */
        { "ICON", ICON1_RID, (char *)&icon1 },
        { "ICON", ICON2_RID, (char *)&icon2 },
        { "CURSOR", CUR1_RID, (char *)&cur1 },
        { "CURSOR", CUR2_RID, (char *)&cur2 },
        { 0 }
};

RESOURCE_INFO *xvt_xres_create_table
{
        xvt_xres_build_icon(&icon1, icon1_height, icon1_width,
                        (DATA_PTR)icon1_bits);
        xvt_xres_build_icon(&icon2, icon2_height, icon2_width,
                        (DATA_PTR)icon2_bits);
        xvt_xres_build_cursor(&cur1, cur1_height, cur1_width,
                        (DATA_PTR)cur1_x_hot,
                        cur1_y_hot,cur1_bits);
        xvt_xres_build_cursor(&cur2, cur2_height, cur2_width,
                        (DATA_PTR)cur2_x_hot,
                                cur2_y_hot, cur2_bits);
        return(rtable);
}
```

The following sections discuss the important sections of this file.

Include Files for the Resource Manager File

At the top of your resource manager file, use #include to include these files:

- **xvt.h**
- **xvt_xres.h**
- Your application's header file that defines resource ID numbers
- Each bitmap file that you created with **bitmap**

Declarations and Statements for the Resource Manager File

For each individual icon or cursor, declare an object of type ICON_RESOURCE or CURSOR_RESOURCE. (These types are defined in **xvt_xres.h.**) For example, if you had two icons and two cursors, the declarations might look like this:

```
static ICON_RESOURCE icon1, icon2;
static CURSOR_RESOURCE cur1, cur2;
```

Then, for each icon or cursor, initialize an element of an array of type RESOURCE_INFO. Each element of this array is a structure with three members:

- The string "ICON" or "CURSOR"
- The ID used by xvt_dwin_draw_icon or xvt_win_set_cursor
- A pointer to the corresponding ICON_RESOURCE or CURSOR_RESOURCE

For example:

```
RESOURCE_INFO rtable[] = {
        { "ICON", ICON1_RID, (char *)&icon1 },
        { "ICON", ICON2_RID, (char *)&icon2 },
        { "CURSOR", CUR1_RID, (char *)&cur1 },
        { "CURSOR", CUR2_RID, (char *)&cur2 },
        { 0 }
};
```

Notice that the table is terminated with an element of all zeros.

The resource IDs must be unique within their type (icon or cursor). Define constants for these numbers in your application-specific header file **sample.h**. For example:

```
#define ICON1_RID 101
#define ICON2_RID 102
#define CUR1_RID 101
#define CUR2_RID 102
```

Additional Required Initialization

Even though the declarations have set up the data structures, the ICON_RESOURCEs and CURSOR_RESOURCEs require further initialization. You must create a function named xvt_xres_create_table, which XVT calls. The function
has this prototype:

```
RESOURCE_INFO *xvt_xres_create_table(void)
```

Put this function in the resource manager file.

XVT supplies two functions, xvt_xres_build_icon and xvt_xres_build_cursor, that your xvt_xres_create_table function must call to initialize the resource objects.

Here are the prototypes for these functions:

```
void xvt_xres_build_icon(iconp, height, width, data)
        ICON_RESOURCE *iconp;       /* pointer to ICON_RESOURCE */
        unsigned height;            /* height from bitmap editor */
        unsigned width;             /* width from bitmap editor */
        DATA_PTR data;              /* data from bitmap editor */

void xvt_xres_build_cursor(curp, height,
                                    width, x_hot, y_hot, data)
        CURSOR_RESOURCE *curp       /* pointer to CURSOR_RESOURCE */
        unsigned height;            /* height from bitmap editor */
        unsigned width;             /* width from bitmap editor */
        unsigned x_hot;             /* hotspot x from bitmap editor */
        unsigned y_hot;             /* hotspot y from bitmap editor */
        DATA_PTR data;              /* data from bitmap editor */
```

These functions use the data from the files you created with **bitmap** to initialize your ICON_RESOURCE and CURSOR_RESOURCE objects.

For example, you might call xvt_xres_build_icon to initialize one of your icon resources as follows:

```
xvt_xres_build_icon(&icon1, icon1_height,
                            icon1_width, (DATA_PTR) icon1_bits);
```

Finally, after initializing the resources, your function must return the address of the RESOURCE_INFO array (rtable in our example).

Do not put any other statements in your resource manager file. Compile the file normally and link it with the rest of your XVT application.

At runtime, XVT calls xvt_xres_create_table so that your calls to xvt_xres_build_icon and xvt_xres_build_cursor are executed. This function returns the address of the resource table (rtable in our example).

Then, whenever it needs an icon or cursor by ID (for instance, when your program calls xvt_dwin_draw_icon), XVT scans the resource table (rtable) for the element with the desired ID and type ("ICON" or "CURSOR"). That element points directly to the ICON_RESOURCE or CURSOR_RESOURCE object that contains the data produced by the icon editor.

2.4.2.2. Cursor Masks

X cursors consist of a shape and a mask. The mask determines which pixels on the screen are modified by the cursor. By default, XVT uses a mask equal to the dimensions of the cursor bitmap. You can create your own cursor mask by creating another cursor (as described above) with a resource ID equal to 100 plus the ID of the cursor for which you are creating the mask.

See Also: For more information about creating a cursor mask, see the **cursor.txt** file in the **doc** directory.

2.4.3. Icons as Controls

You can treat icons as controls in windows and dialogs. The two different ways to do this are:

- By specifying the icon as a control in your URL file
- By specifying the icon from within your program using xvt_ctl_create_def, xvt_win_create_def, or xvt_dlg_create_def

Tip: To specify icons as controls belonging to a window or dialog in your URL file:

1. Create your icon using the X Window System icon editor, as explained in section 2.4.2.
2. Add an ICON statement in your URL file. A sample ICON statement follows:

   ```
   ICON ICON_RID 200, 300, 75, 50 123
   ```

   where the statement consists of:

   - The keyword ICON
   - A resource ID (ICON_RID)
   - The x, y, width, and height specifications
   - A number that refers to the native icon resource ID (123)

3. To your **.url** file, add a #transparent statement that tells the UIL compiler about your icon. Here is an example of a #transparent statement used to define an icon:

```
#transparent $$$
        value icon_123 : exported
        xbitmapfile('myicon.icon');
$$$
```

   where the statement consists of:

   - The keyword value
   - Your icon's special ID (the word icon followed by an underscore followed by the number at the end of your URL ICON statement)
   - A colon
   - The keywords exported xbitmapfile
   - Your bitmap filename in parentheses and single quotes

When you create a dialog or window, XVT adds the icon to the window or dialog just like it would for any other control.

You can also create icons in windows using xvt_ctl_create_def. (The function xvt_ctl_create does not allow creation of icons.)

Tip: To create an icon control using xvt_ctl_create_def:

1. Create the icon using the X Window System icon editor, as explained in section 2.4.2.
2. Add a #transparent statement to your URL file (see step 3 above).
3. Create a WIN_DEF structure with the type specified as WC_ICON, and initialize its elements as follows:

rct:	The location of the rectangle bounding the icon
units:	U_PIXELS
wtype:	WC_ICON
v.ctl.icon_id:	The icon's resource ID number (the same number that was appended to the "special ID" in the #transparent statement)

For the #transparent statement above, setting the icon_id in the WIN_DEF structure looks like this:

```
windef[0].v.ctl.icon_id = 123;
```

The file containing the icon ('myicon.icon') must be in the local directory, or you must specify a pathname to the file. If you prefer not to use an external file to specify your icon, or if you would like to use colors in your icon, you can use UIL's icon function in place of the xbitmapfile function.

See Also: For more information about UIL's icon function, see "Color and Pixmaps" in the *OSF/Motif Programmer's Guide Release 1.2*. For general UID details, see "UIL" in the *OSF/Motif Programmer's Reference Release 1.2*.

Note: If you plan to use Motif-specific icons or cursors in your DSC++ application, you must edit the **XResMgr.cxx** file and add the appropriate definitions. For example, the resources corresponding to the retired class CIcon objects cannot be specified via XVT's Universal Resource Language (URL). You must, instead, define them in the **XResMgr.cxx** file with a small amount of C code. **XResMgr.cxx** is generated by XVT-Architect.

2.5. XVT's Encapsulated Font Model

2.5.1. Font Terminology

This section uses the following XVT-defined terms to describe XVT's encapsulated font model:

Physical font
A particular *implementation* of a font as installed on the window system on which an application is running.

Logical font
A *description* of a desired physical font, to a degree of specificity ranging from just a typeface family name or size to a complete description that specifies a particular physical font. A logical font has both portable and non-portable attributes. It is identified by an object of type XVT_FNTID.

2.5.2. Native Font Descriptors for Screen Display

To specify a particular physical font, your application can use a native font descriptor, which is a string of data fields. You can include this string as a parameter to xvt_font_set_native_desc, or in URL as part of a FONT or FONT_MAP statement.

The native font descriptor string contains the following fields:

- The native window system and version of the XVT encapsulated font model (the current version is "01").
- Platform-specific fields that the XVT Portability Toolkit decodes and uses to uniquely specify a native font. The fields describe specific attributes of a native font. Each field is separated by a slash, "/".

The native font descriptor string, then, has this format:

"<system and version>/<field1>/<field2>/<field3>/
...<fieldn>"

See Also: For more information about specifying fonts, see the "Fonts and Text" chapter in the *XVT Portability Toolkit Guide*.

2.5.2.1. XVT/XM Font Descriptor Version Identifier

For XVT/XM, the screen font descriptor version identifier format is X11<vers>. In this release of XVT/XM, the font descriptor version number is "01," so the screen font descriptor version identifier is X1101.

2.5.2.2. XVT/XM Font Fields

For Motif platforms, the native font descriptor string must contain enough information to populate an "X logical font description," which is a 13-part font specification. The following table shows the information used to map a logical font.

foundry	"Manufacturer" of typeface, e.g., Adobe
family	Name of typeface
weight	Density of typeface
slant	Roman (regular), italic, oblique, etc.
set_width	Normal, condensed, narrow, etc.
addl_styl	Additional style information, e.g., "sans"
pixel_size	Height in pixels at specified resolution
point_size	Nominal size of typeface in points
h_res	Horizontal resolution of font
v_res	Vertical resolution of font
spacing	Character spacing, e.g., "p" to indicate a proportional font
avg_width	Average width of font in tenths of a pixel
encoding	Character set encoding, e.g., "iso8859-1"

For XVT/XM, the native font descriptor string has this structure:

"X1101/<foundry>/<family>/<weight>/<slant>/<set_width>/<addl_style>/<pixel_size>/<point_size>/<h_res>/<v_res>/<spacing>/<avg_width>/<encoding>"

Example: This string shows a valid XVT/XM native screen font descriptor string:

"X1101/*/times/bold/i/normal//*/*/100/100/*/*/iso8859-1"

Note: An asterisk (*) in a native font descriptor string indicates a wildcard condition in which any value is acceptable for that particular field. An empty field in a native font descriptor string matches only a null value in that particular field.

Tip: For a list of fonts available on your particular platform, enter a command similar to the following:

/usr/bin/X11/xlsfonts on many UNIX platforms

/usr/X11/bin/xlsfonts on some UNIX platforms

$OPENWINHOME/bin/xlsfonts on SPARC platforms

Note: These are the most common paths to **xlsfonts**. The pathname may be different on other platforms. This font utility may not exist on all platforms.

2.5.3. Native Font Descriptors for Printing

When using XVT/XM, print fonts are specified differently than screen fonts.

2.5.3.1. XVT/XM Font Descriptor Version Identifier

For XVT/XM, the print font descriptor version identifier format is POS<vers>. In this release of XVT/XM, the font descriptor version number is “01,” so the print font descriptor version identifier is POS01.

2.5.3.2. XVT/XM Print-specific Font Fields

Because the XVT/XM Portability Toolkit uses PostScript software to print, the native print font descriptor string must contain enough information to completely specify a PostScript font. The following table shows the information required to specify a PostScript font.

family	Name of font
style	Combination of font characteristics
size	Size of typeface, measured in points

The PostScript font descriptor string has this structure:

"POS01/<family>/<style>/<size>/"

Example: This string shows a valid PostScript font descriptor string:

"POS01/Times/BoldItalic/*"

2.5.3.3. Standard PostScript Native Descriptor Strings

The following standard combinations are available by default:

“Fixed” portable family:

"POS01/Courier/Normal/*"
"POS01/Courier/Bold/*"
"POS01/Courier/BoldOblique/*"
"POS01/Courier/Oblique/*"

“Helvetica” portable family:

"POS01/Helvetica/Normal/*"
"POS01/Helvetica/Bold/*"
"POS01/Helvetica/BoldOblique/*"
"POS01/Helvetica/Oblique/*"

“System” portable family:

"POS01/Times/Normal/*"
"POS01/Times/Bold/*"
"POS01/Times/BoldItalic/*"
"POS01/Times/Italic/*"

“Times” portable family:

"POS01/Times/Normal/*"
"POS01/Times/Bold/*"
"POS01/Times/BoldItalic/*"
"POS01/Times/Italic/*"

2.6. Printing in XVT/XM

Printing from XVT/XM generates a PostScript file. Print output is placed in a file created by the C library function tmpnam.

See Also: To set a specific output filename, see the description of the ATTR_PS_PRINT_FILE_NAME attribute in Appendix A: Non-portable Attributes, Escape Codes, and Menu Fields on page A-1.

2.6.1. Print Files and XVTPATH

XVT/XM requires a number of files, located in the **print** directory of the XVT/XM installation, to generate a PostScript file. These files include the font metrics files and the file **xvtprolg.ps**, which contains PostScript driver functions. For printing to work properly, these files must all be located in the same directory.

See Also: For a string pointer that points to the command to execute on the PostScript print file, see Appendix A: Non-portable Attributes, Escape Codes, and Menu Fields on page A-1.

To print in XVT/XM, you must set the environment variable to include the **print** directory of the XVT PTK installation.

See Also: See the document *Installing XVT Development Solution for C for Motif* for more information on setting UNIX environment variables.

2.6.2. Fonts Used in Printing

For information about how to specify PostScript fonts for XVT/XM to use when printing, refer to section 2.5.3.

2.7. Using X Resources

X toolkit resources specify GUI object (widget) attributes. Resources are specified in the **.Xdefaults** file or in application class-specific files.

Note: The X resource specification allows either global (loosely) bound specifications (*XmPushButton.foreground: green) or named per-widget instance specifications (*button.foreground: black). Since XVT has

chosen not to document the algorithm by which instance names are assigned to Motif widgets instantiated by the XVT Portability Toolkit, only global specifications can be used reliably.

UNIX

The **.Xdefaults** file is (typically) loaded into the X server at the start of the session. Any changes the user makes to **.Xdefaults** take effect only in a new session, or after an invocation of xrdb reloads the resource database.

Application class resource files use the base_appl_name field of the XVT_CONFIG structure, and must be present either in **$HOME** or in the **app-defaults** directory. The **app-defaults** directory is typically located under **/usr/lib/X11** (on SPARC, the **app-defaults** directory is located under **$OPENWINHOME/lib**). X recognizes various environment variables for specifying paths to the application resource files.

See Also: For more information about making changes to widgets used by XVT/XM, see XVT Technical Note 117.
For more information about X resources, consult the X resources chapter in O'Reilly's *X Toolkit Intrinsics Programming Manual* and *Programming Reference.*

2.8. Making Changes to the Widgets Used by XVT/XM

This section presents three methods for making changes to the widgets used by XVT/XM. At the end of this section is a list of widgets used by XVT for particular objects, as well as a list of reference manuals that are helpful when extending the XVT/XM toolkit.

These methods are not appropriate for changing fonts and colors of XVT top-level and child windows since the XVT drawing routines (xvt_dwin_draw_text(), xvt_dwin_draw_rect(), and so on) do not use the Motif widgets.

When you make changes to the widgets outside of XVT, the XVT library doesn't know that these changes have been made. Consequently, you may see some unexpected results. For example, if you changed the font of a push-button or oblong button after is was created, the button would not resize based on the new font. To modify the button, you have to call xvt_vobj_move() or resize it with the Intrinsics.

2.8.1. Method One: Using Files under App-Defaults

XVT/XM is an Intrinsics-based toolkit. It takes advantage of the Intrinsics method of changing widget attributes by using the files under the **app-defaults** directory. This method uses files associated with the application. These files contain names and values of attributes that affect the appearance of a widget.

The **/usr/lib/X11** directory contains a directory titled **app-defaults**. The resource manager looks in the **app-defaults** directory for files that directly affect the look of particular applications.

Note: For Sparc users, Sun changes the X directory structure and uses the $OPENWINHOME environment variable to point to the location of the X installation. Although Sparc machines do not have a **/usr/lib/X11** directory, they look for this directory to find the **app-defaults** files. Therefore, you need to create the **/usr/lib/X11** directory. Then, you can either create the **app-defaults** directory under the **/usr/lib/X11** directory, or you can simply put a symbolic link in the **/usr/lib/X11** directory to point to the **app-defaults** directory in the path, **$OPENWINHOME/lib/app-defaults**.

2.8.1.1. Name of the Resource File in the App-Defaults Directory

The name of the file in the **app-defaults** directory used by XVT applications is specified by the base_appl_name field of the XVT_CONFIG structure.

2.8.1.2. Names of Resources

You can change the look of dialogs, controls, and menubars by prepending the name of the resource with an asterisk. For example, to change the background color in dialogs, controls, and menubars, add the following line to your file in **app-defaults**:

```
*background:  pink
```

You can be more specific by using the class name of the widget you want to change. For example, to change the background color of only WC_PUSHBUTTON controls, add the following line to your file in **app-defaults**:

```
*XmPushButton.background:    pink
```

2.8.1.3. Sample App-Defaults File to Change Fonts and Colors

The following is a sample **app-defaults** file that would change the foreground and background color of all dialogs, controls, and menubars to green and pink, respectively. This code also changes the font of pushbuttons and editable text to an rk24 font:

```
*foreground:  green
*background:  pink
*XmPushButton.fontList:  rk24
*XmText.fontList:  rk24
```

2.8.2. Method 2: Using Intrinsics to Change Widget Attributes

The second method programmatically changes the attributes of a particular widget by using direct Intrinsics routines. This is the most difficult of the three methods, but it gives you the most control over individual widgets.

2.8.2.1. Getting the Widget

You can get the widget for most XVT objects if you have an XVT WINDOW for the object. For dialogs, you already have a WINDOW, but for controls, you have to get the WINDOW before you can get the widget. To get the WINDOW for a control, use the following code:

```
WINDOW window;
window = xvt_win_get_ctl(parent_win, CONTROL_ID);
```

Before you can get the widget, you need to include the following header file:

```
#include <X11/Intrinsic.h>
```

Without this #include file, your compiler will not know what a widget is. Once you have the WINDOW, use the following code:

```
Widget widget;
widget = (Widget)xvt_vobj_get_attr(window,
        ATTR_X_WIDGET);
```

Now you have the widget for the XVT object.

2.8.2.2. Setting Up Attributes to Change

Although changing the height or width, or other simple changes, does not require much work, changing fonts or colors requires more work. The Motif reference manual provides some information, but it is likely you'll have to refer to the O'Reilly manuals to get all the information you need.

2.8.2.3. Using XtVaSetValues to Change Attributes

Once the attribute(s) are ready to be set, you can use a simple Intrinsics call to make the change. The following changes the height and width of a dialog, without calling xvt_vobj_move():

```
XtVaSetValues(widget, XmNheight, (XtArgVal)322,
        XmNwidth, (XtArgVal)477, NULL);
```

The attributes to be changed are paired with the value to which they are changing. The last parameter in this routine must always be NULL.

2.8.2.4. Sample C Code to Change Fonts

The following sample C code will change fonts using the Intrisics method:

```
#include <X11/Xlib.h>
#include <X11/Intrinsic.h>
#include <Xm/Xm.h>

Widget widget;
Display *display;
XFontStruct *Xfont;
XmFontList fontlist;

widget = (Widget)xvt_vobj_get_attr(
        xvt_win_get_ctl(win, PUSHBUTTON_ID),
        ATTR_X_WIDGET);
display = (Display *)xvt_vobj_get_attr(win,
        ATTR_X_DISPLAY);
if ((Xfont = XLoadQueryFont(display,
        "rk24")) == NULL)xvt_dm_post_error(
        "couldn't open rk24 font\n");
else {
        fontlist = XmFontListCreate(Xfont,
                XmSTRING_DEFAULT_CHARSET);
        XtVaSetValues(widget, XmNfontList, fontlist,
                NULL);
}
```

2.8.3. Method 3: Using UIL

UIL is Motif's resource language. It requires only adding (or changing) attributes in the proper section of the UIL file. This method is easy to use, but you can use it only on a subset of widgets created by XVT.

2.8.3.1. Restrictions for Using UIL to Make Non-portable Changes

This method works only for dialogs and controls in dialogs created by xvt_dlg_create_res(). It will not work for controls in windows or for dialogs or controls in dialogs created from xvt_dlg_create_def().

2.8.3.2. Basic UIL Syntax

The following is a sample of generic UIL code generated by the xrc compiler:

```
object
    XVT_DLG_256 : XmBulletinBoardDialog {
        arguments {
            XmNdialogStyle = XmDIALOG_MODELESS;
            XmNdialogTitle = "List Box Exerciser";
            .
            .
            .
        };
        controls {
            XmForm control_256_1;
            XmText control_256_16;
            .
            .
            .
        };
        callbacks {
            MrmNcreateCallback =
            procedure xvt_create_dlg_cb(256);
        };
    };
```

The only place you need to make changes is in the "arguments" section of the UIL file of the control or dialog that is changing.

2.8.3.3. Syntax for Changing Colors and Fonts

Let xrc create the UIL file for you, and then make the changes. When the changes are done, you need to put the entire dialog code back into the **URL** file in a #transparent statement. You cannot put just a portion of the dialog into a #transparent statement.

To change the background and foreground colors of a particular object in UIL, add the following lines to the “arguments” section of the object you are changing:

```
XmNbackground = color('pink');
XmNforeground = color('green');
```

To change the font of a particular object in UIL, add the following line to the “arguments” section of the object you are changing:

```
XmNfontList = font('rk24');
```

Remember to put the dialog back into a #transparent statement after making the changes.

2.8.3.4. Sample UIL Code to Change Fonts and Colors

The following shows the UIL code for changing the background and foreground colors and the font for the "Add" push-button in the "List Box Exerciser” dialog (in the dlg example):

```
object
    button_256_1 : XmPushButton {
        arguments {
            XmNx = 0;
            XmNy = 0;
            XmNwidth = 60;
            XmNheight = 20;
            XmNmarginBottom = 1;
            XmNhighlightOnEnter = true;
            XmNhighlightThickness = 1;
            XmNrecomputeSize = false;
            XmNshowAsDefault = 1;
            XmNlabelString = "Add";
            XmNalignment = XmALIGNMENT_CENTER;
            XmNbackground = color('pink');
                ! user added line
            XmNforeground = color('green');
                ! user added line
            XmNfontList = font('rk24');
        };
        callbacks {
            MrmNcreateCallback = procedure xvt_create_ctl_cb(1);
        };
    };
```

2.8.4. Widgets Used by XVT/XM

The following table shows the relevant widgets used by XVT/XM:

Windows:	
Top-level	XmMainWindow => XmDrawingArea
Child	XmScrolledWindow => XmDrawingArea
Dialogs:	
WD_MODAL	XmBulletinBoard
WD_MODELESS	XmBulletinBoard
Controls:	
WC_CHECKBOX	XmToggleButton
WC_RADIOBUTTON	XmToggleButton
WC_TEXT	XmLabel
WC_EDIT	XmText
WC_LBOX	XmScrolledWindow => XmList
WC_HSCROLL	XmScrollBar
WC_VSCROLL	XmScrollBar
WC_PUSHBUTTON	XmPushButton
WC_LISTBUTTON	XmText & XmArrowButton & XmScrolledWindow => XmList

WC_LISTEDIT	XmText & XmArrowButton & XmScrolledWindow => XmList
WC_GROUPBOX	XmFrame => XmLabel
WC_ICON	XmLabel w/ pixmap

2.8.5. Helpful Reference Manuals

The following is a list of helpful reference manuals:

Open Software Foundation. *OSF/Motif Programmer's Reference*, Revision 1.1. Englewood Cliffs, NJ: Prentice Hall. ISBN 0-13-640681-5.

The Definitive Guides to the X Window System, by O'Reilly & Associates, Inc.:

Volume 1: Xlib Programming Manual. ISBN 0-937175-11-0.

Volume 2: Xlib Reference Manual. ISBN 0-937175-12-9.

Volume 4: X Toolkit Intrinsics Programming Manual. ISBN 0-937175-62-5.

Volume 5: X Toolkit Intrinsics Reference Manual, Second Edition. ISBN 0-937175-57-9.

3

DEVELOPMENT ENVIRONMENT

3.1. Introduction

If you are using XVT-Design or XVT-Architect, you will rarely, if ever, need to deal directly with makefiles, include files, compiler options, libraries and linkers. Unless you need to modify the makefile templates supplied by XVT-Design, you'll only need to refer to the information in this section (3.1).

This chapter gives detailed information on building XVT/XM applications.

Your compiled XVT/XM application consists of the following files:

- The binary executable file **your_app** for UNIX.
- The binary resource file **your_app.uid**, which contains descriptions of your windows, dialogs, controls, menubars, and strings.
- XVT portable help binary files (***.csc**).
- Portable image files (***.bmp**).

See Also: See the document **Install.txt** for Motif for more information on setting environment variables and logical names.

3.2. UNIX Development Environment

3.2.1. Executing Makefiles

XVT-Design and XVT-Architect generate makefile templates that you can use in your UNIX environment.

For UNIX, use the **make** utility to process makefiles. You can adapt the makefiles that accompany the sample programs for use in building your application.

3.2.2. Include Files

XVT-Design and XVT-Architect generate code that automatically includes all necessary header files.

To build XVT applications, you must include the XVT-specific header file **xvt.h** in addition to any other application-specific header files.

3.2.3. Compiler Flags

XVT provides a compiler optimization flag, XVT_OPT, for runtime optimization of the PTK. This flag is described further in the *XVT Portability Toolkit Guide*. To use the flag with your UNIX compiler, you must add a define for the XVT_OPT symbol on the compiler line.

See Also: For details about how to define a symbol in a compile statement, see the user's manual for your particular operating system's compiler.

3.2.4. Libraries

XVT-Design's and XVT-Architect's makefile templates supply a default configuration that links the appropriate libraries automatically.

3.2.4.1. Shared and Static XVT Libraries

See Also: For up-to-date information on library names, see the **Readme** file in the **doc** directory.

XVT/XM provides the following libraries for building applications:

libxvtxmapi*.a	XVT API library
libxvtxmba*.a	XVT Base library
libxvtxmhb*.a	Bound help viewer library
libxvtxmhi*.a	Standalone help viewer library

You *must* link the libraries on the link line in the following order:

libxvtxmapi*.a <libxvtxmhb*.a or libxvtxmhi*.a>
libxvtxmba*.a

XVT/XM provides static versions of **libxvtxmapi*.a**, **libxvtxmba*.a**, **libxvtxmhb*.a.** and **libxvtxmhi*.a**. On some platforms, XVT/XM also provides shared versions of **libxvtxmba*.a**, **libxvtxmhb*.a**, and **libxvtxmhi*.a**. If you link with the shared versions of the libraries, they must be present at runtime.

If you link your applications with the *static* XVT libraries, you do not have to ship the libraries to your customers. However, if you link your applications with the *shared* XVT libraries, then you must ship the shared XVT libraries with your product.

Tip: To link with the static versions of the libraries, give a full pathname to the libraries in your makefile.

Tip: To link with the shared versions of the libraries:

1. Use the -L flag to indicate the path to the XVT libraries.
2. To use the bound help viewer, specify:

   -lxvtxmapi* -lxvtxmhb* -lxvtxmba*

   as part of the link command.

   To use the standalone help viewer, specify:

   -lxvtxmapi* -lxvtxmhi* -lxvtxmba*

   as part of the link command.

Note: If you are using XVT/XM on an AIX platform, append _shr to the name of a library to get the shared version, e.g., xvtxmba*_shr.

See Also: For more information about linking libraries, see the user's manuals for your particular operating system and linker.

3.2.4.2. System Libraries

In addition to the XVT libraries, XVT/XM applications require linking with the following system libraries (or shared versions of them):

libMrm.a	Motif resource manager library
libXm.a	Motif widget libary
libXt.a	X toolkit instrinsics library
libX11.a	X lib functions library
libm.a	Math functions library

See Also: Some platforms may require additional system libraries. See the makefiles that accompany the sample programs for further information.

3.2.4.3. Linking helpview

XVT/XM provides **helpview** object files in binary format. You may link **helpview** using either static or dynamic libraries. The sample makefile **Makefile.lnk** in the **src/helpview** directory shows the combination of static and shared libraries used by XVT to link **helpview**.

See Also: To use XVT's hypertext online help system, see section 3.4 of this manual and the "Hypertext Online Help" chapter in the *XVT Portability Toolkit Guide*.

3.2.5. Building Utility Programs

All XVT/XM customers receive a command line version of **errscan** in the **bin** directory and the source code in the **src/errscan** directory. You can build either the command line version of this utility or you can build it with a GUI interface (named **errscan_app**).

Tip: To build the **errscan** utility program:

1. Move to the **src/errscan** subdirectory.
2. To build the command line version of **errscan**:

   Type: make errscan
3. To build the GUI version of **errscan**:

   Type: make errscan_app

Executables are placed in the **bin** directory.

3.2.6. For Source Customers Only: XVT/XM Development Environment

This section contains information pertinent to XVT/XM source customers. If you are using the XVT/XM binary product, you can skip this section.

For source customers, XVT/XM supplies makefiles for the utility programs **xrc**, **helpc**, **helpview**, and **maptabc**.

Building Utility Programs

Tip: To build the **xrc** or **maptabc** applications:

1. Move to the **src/xrc** directory.
2. Type: make

Executables are placed in the **bin** directory.

See Also: For further information about the **maptabc** application, see the *XVT Portability Toolkit*.

Tip: To build the help compiler **helpc**, the help viewer **helpview**, and the supporting help libraries:

1. Move to the **src/helpc** directory.
2. Type: make

Executables are placed in the **bin** directory, and libraries are placed in the **lib** directory.

Building Libraries

Tip: To build the XVT/XM libraries:

1. Move to the **src/ptk** directory.
2. Type: make

Libraries are placed in the **lib** directory.

3.3. Compiling Resources

*If you use XVT-Design or XVT-Architect, you will probably never need to deal directly with resource compiler options. XVT-Design and the **xrc** compiler code resources automatically. The information here is provided for reference purposes only.*

3.3.1. Using xrc

*XVT-Design can be configured to invoke **xrc** for you, either directly as part of the code generation process or via a generated makefile.*

Tip: To compile XVT URL resources with **xrc**, use command lines similar to the following:

UNIX:

```
//If using Motif 1.2.x
xrc -r mtf12 -I../include -DLIBDIR=../lib sample.url

//If using Motif2.x
xrc -r mtf20 -I../include -DLiBDIR=../lib sample.url
```

xrc complies URL resource files into Motif UIL (User Interface Language) resource scripts (***.uil** files).

See Also: For more information about using **xrc**, see the "Resources and URL" chapter in the *XVT Portability Toolkit Guide*.
For a list of **xrc** options, see the online *XVT Portability Toolkit Reference*.

3.3.2. Using the Native Resource Compiler (uil)

The Motif Toolkit includes tools that you can use to make Motif resources. The native resource compiler **uil** compiles resource scripts into resource files that have an extension of **.uid**.

Compile your resource scripts (***.uil** files) with the **uil** command, just as you do for non-XVT applications.

Here's a typical command:

```
uil sample.uil
```

Entering this command creates a file named **sample.uid** that can be bound with your application at runtime.

Normally, you use **xrc** to compile menus, dialogs, windows, and strings. Its output is a series of UIL statements, which you then compile with **uil**. If a resource can't be written in URL, you'll have to code it directly in the UIL language and embed it in your URL script with a #transparent statement.

The XVT-Design tag SPCL:User_Url *lets you add resource definitions to your application resource file.*

See Also: For more information on using **uil**, see the *OSF/Motif Programmer's Guide*.
For more information on creating Motif-specific resources, refer to section 2.4.1.
Refer to ATTR_RESOURCE_FILENAME in the online *XVT Portability Toolkit Reference* for information on setting the base name of the resource file.

3.4. Building Your Application with the Help System

XVT-Design supplies a default configuration in its makefile template that links with the bound help viewer. If necessary, you can modify this configuration to suit your needs.

XVT's hypertext online help system requires a help viewer. For XVT/XM, you can bind the portable viewer to the application. An application should link with only *one* of the two help libraries discussed below.

Note: XVT/XM currently supports only the XVT bound help viewer and the standalone help viewer, **helpview**.

See Also: For information on the help viewers, see the "Hypertext Online Help" chapter in the *XVT Portability Toolkit Guide*.
For information on the portable help compiler command options, refer to the online *XVT Portability Toolkit Reference*.

3.4.1. Portable Viewer

See Also: For up-to-date information about library names, see the **Readme** in the **doc** folder.

XVT/XM provides the XVT portable hypertext help viewer in bound and standalone forms. You must use the XVT help compiler **helpc** to produce XVT-portable binary help files for the help viewer to use.

Tip: To compile help text source files for use with the portable viewer, use command lines similar to the following:

UNIX:

```
helpc -f xvt -i../../include sample.csh
```

Tip: To bind the help viewer to your application or to use the standalone help viewer, link with *one* of the following libraries (in addition to the base XVT libraries):

libxvtxmhb*.a	Bound help viewer library
libxvtxmhi*.a	Standalone help viewer library

Caution: If you are providing context-sensitive help from *modal* XVT windows or dialogs, XVT strongly recommends that you use the portable standalone viewer. The bound viewer is a *modeless* window in XVT. Opening a modeless window from a modal object may result in undefined behavior.

3.4.2. Object Click Mode

Object click mode for XVT's hypertext online help system is *not* standard look-and-feel for Motif. Therefore, XVT/XM does not automatically provide an application menu item which enables this feature for users. If your XVT/XM application requires context sensitive help using object click mode, you must define the symbol, XVT_HELP_OBJCLICK, for compiling URL resources. When this symbol is defined, XVT/XM includes the resource for the **Object Click Mode** menu item in the standard menu. Use a command line similar to the following for compiling URL resources:

UNIX:

```
//If using Motif 1.2.x
xrc -r mtf12 -I../../include
            -DLIBDIR=../../lib -DXVT_HELP_OBJCLICK
            sample.url

//If using Motif 2.x
xrc -r mtf20 -I../../include DLIBDIR=
      ../../lib -DXVT_HELP_OBJCLICK Sample.url
```

Note: Although the command shown here is printed on several lines, you should enter a command line as a single line.

Alternatively, you may define the symbol prior to including **url.h** in your application URL resources:

```
#define XVT_HELP_OBJCLICK
#include "url.h"
```

A

APPENDIX A: NON-PORTABLE ATTRIBUTES, ESCAPE CODES, AND MENU FIELDS

A.1. Non-portable Attributes

The xvt_vobj_set_attr and xvt_vobj_get_attr functions allow you to manipulate XVT attributes. Non-portable attributes let you fine-tune your application to make it more closely adhere to the look-and-feel of the underlying platform, or to add functionality not provided by the XVT application interface. This section provides a list of the non-portable attributes for use with XVT/XM.

See Also: Additional non-portable attributes may be listed in the **readme** file in the **doc** directory.

γ

XVT-Design provides a special tag, SPCL:Main_Code*, that lets you supply code in the Action Code Editor (ACE) before calling xvt_app_create. This enables you to set or get system attributes before the XVT library assumes control.*

ATTR_IME_USE_STATUSAREA

Note: This attribute is for AIX only.

Description: This attribute allows the window's status bar to indicate that the IME is associated with the window.When this attribute is set to FALSE, the IME appears to be global, and there is no visual way to identify the window to which the IME belongs.Setting this attribute to TRUE allows the window's status bar to indicate that the IME belongs to that particular window.

Uses win argument:	No
xvt_vobj_get_attr returns:	BOOLEAN
xvt_vobj_set_attr effect:	Window status bar when the IME is present
xvt_app_create use:	Must use before
Default value:	FALSE

ATTR_PS_PRINT_COMMAND

Description: This attribute is a string pointer that points to the command to execute on the PostScript print file after the print file has been generated.

If this attribute is *not set* or is set to NULL, no command is executed.

If this attribute is set, it should be a shell command line containing the substring %s which will be replaced with the current printer output filename. After the substitution, the command is passed to the system for execution. The command lp %s, for example, will send the print file to the default printer.

The portable help viewer (standalone and bound) sets the print filename to helpview.ps via ATTR_PS_PRINT_FILE_NAME.

If the print command is *not set*, the portable help viewer sets the print filename to lp %s.

After printing, the help viewer code restores the original values of these attributes. If you are linking your application with the portable bound help viewer and do not want it to use this default command, you should set the ATTR_PS_PRINT_COMMAND attribute before the help viewer is called. To avoid having any command executed, set this attribute to a blank (*not* NULL) command string.

See Also: For more information about PostScript printing in XVT/XM, see section 2.6.

Uses win argument:	No
xvt_vobj_get_attr returns:	char*
xvt_vobj_set_attr effect:	Causes subsequent printing to construct and execute print command
xvt_app_create use:	Must use after
Default value:	NULL

ATTR_PS_PRINT_FILE_NAME

Description: This attribute is a string pointer that points to the name of the printer output file used by the PostScript printing feature. If the attribute is set, print output is placed in the specified filename. If this attribute is not set or is set to NULL, the function tmpnam is called to get the name of the file for the print output.

Be aware that, by default, XVT uses the same name for the PostScript file each time you print something from within the same application. To generate unique names for your PostScript print files, set the attribute ATTR_PS_PRINT_FILE_NAME to a different filename before each call to xvt_print_create_win.

See Also: For more information about PostScript printing in XVT/XM, see section 2.6.

Uses win argument:	No
xvt_vobj_get_attr returns:	char*
xvt_vobj_set_attr effect:	Causes the printing function to use the name specified in the attribute if set, or calls tmpnam to create the print filename if NULL
xvt_app_create use:	Must use after
Default value:	NULL

ATTR_X_DISPLAY

Description: This attribute gets the X display pointer corresponding to the display on which the XVT application is running.

Uses win argument:	No
xvt_vobj_get_attr returns:	Display*
xvt_vobj_set_attr effect:	Illegal
xvt_app_create use:	Must use after
Default value:	None

ATTR_X_DISPLAY_TASK_WIN

Description: This attribute controls whether the task window menubar appears when there are no other menubar-carrying windows visible on the screen.

Caution: Setting this attribute to FALSE can effectively hang your application if the application doesn't create a dialog or window at startup or doesn't terminate the application when the last dialog or window is closed.

Uses win argument:	No
xvt_vobj_get_attr returns:	BOOLEAN
xvt_vobj_set_attr effect:	Task menubar appears if TRUE, does not appear if FALSE
xvt_app_create use:	Can use either before or after
Default value:	TRUE

ATTR_X_DLG_PARENT

Description: This attribute allows the user to set the parent of subsequently created dialogs to be the specified window. The window must be a top-level window, child window, or a dialog. If NULL_WIN is specified, the standard XVT/XM method of dialog parenting is used (i.e., dialogs are parented to the screen window).

Uses win argument:	No (window is passed in as the value)
xvt_vobj_get_attr returns:	WINDOW (XVT type)
xvt_vobj_set_attr effect:	Specifies parent WINDOW for dialogs; specifying NULL_WIN causes dialogs to be parented to SCREEN_WIN
xvt_app_create use:	Must use after
Default value:	NULL_WIN

ATTR_X_MASK_SERVER_EVENTS

Description: This attribute determines the level at which events are masked.

If the attribute is TRUE, setting an event mask for an XVT window or dialog also masks the events at the server. This reduces message traffic between the server and the application. Not all events are masked at the server level, but at the very minimum, E_MOUSE_MOVE events are masked. At the server level, E_MOUSE_DBL events are masked if other E_MOUSE button events are masked.

If the attribute is FALSE, events are masked only at the XVT level.

Uses win argument:	No
xvt_vobj_get_attr returns:	BOOLEAN
xvt_vobj_set_attr effect:	Masks events at server level if TRUE or at XVT level if FALSE
xvt_app_create use:	Can use either before or after
Default value:	FALSE

ATTR_X_PLACE_WINDOW_EXACT

Description: This attribute controls top-level window placement as follows:

- If TRUE, the client area of all subsequently created top-level windows is placed at the coordinates specified in the RCT structure passed to the window creation function.
- If FALSE, the window manager can place a top-level window wherever it chooses. The placement of the window varies according to the user's environment.

Uses win argument:	No
xvt_vobj_get_attr returns:	BOOLEAN
xvt_vobj_set_attr effect:	Places all subsequently created top-level windows at exact coordinates if TRUE or wherever window manager chooses if FALSE
xvt_app_create use:	Can use either before or after
Default value:	FALSE

ATTR_X_SELECTION_BUFF

Description: This attribute allows your application to select the X buffer for its cut, copy, and paste functions. The value of this attribute is one of three integers:

0 - Use the Motif clipboard buffer.

1 - Use the X primary selection buffer (which is used by **xterm**).

2 - Use the Motif clipboard for *cut* and *copy*, and if the primary selection buffer contains data, the primary selection buffer is cleared. Use the primary selection buffer for *paste* (if the primary selection buffer contains data) otherwise, paste from the Motif clipboard.

Uses win argument:	No
xvt_vobj_get_attr returns:	Previously set value
xvt_vobj_set_attr effect:	Changes the selection buffer; see description above
xvt_app_create use:	May use before or after
Default value:	0

ATTR_X_SET_FOCUS_DEICONIZE

Description: This attribute (TRUE or FALSE) tells xvt_scr_set_focus whether to do anything to an iconized window:

- If TRUE, when the window is iconized, xvt_scr_set_focus deiconizes (opens) the window and gives it focus. When the window is not iconized, xvt_scr_set_focus works normally.
- If FALSE, when the window is iconized, xvt_scr_set_focus does nothing. When the window is not iconized, xvt_scr_set_focus works normally.

Uses win argument:	No
xvt_vobj_get_attr returns:	BOOLEAN
xvt_vobj_set_attr effect:	Changes the way xvt_scr_set_focus assigns focus to iconized windows; see description above
xvt_app_create use:	Must use after
Default value:	FALSE

ATTR_X_USE_USERS_STRING

Description: The functions xvt_dm_post_ask, xvt_dm_post_error, xvt_dm_post_fatal_exit, xvt_dm_post_message, xvt_dm_post_note, and xvt_dm_post_warning normally format their string arguments using printf-like conventions. Because of this formatting, a 256-character limit is imposed on the length of the formatted strings. Setting this attribute to TRUE removes this limit, but the strings are not formatted. If you need to format them, use sprintf and pass the formatted string to the xvt_* function.

- If TRUE, the strings are used verbatim, without parsing. The string's length is unlimited.
- If FALSE, the strings are formatted. The maximum length of the formatted string is 256 characters.

Uses win argument:	No
xvt_vobj_get_attr returns:	BOOLEAN
xvt_vobj_set_attr effect:	Parses string if FALSE or uses it verbatim if TRUE
xvt_app_create use:	Can use either before or after
Default value:	FALSE

ATTR_X_WIDGET

Description: This attribute gets the client widget of an XVT window. For the task window, the client widget is the menubar; for the screen window, the client widget is not defined.

Uses win argument:	Yes
xvt_vobj_get_attr returns:	Widget
xvt_vobj_set_attr effect:	Illegal
xvt_app_create use:	Must use after
Default value:	None

ATTR_XOR_REF_COLOR

Description: On X color workstations having a color palette (those that use the "PseudoColor" visual), you draw in XOR mode by XORing the pixel value corresponding to the foreground color (as set in the drawing tools) with the existing pixel values in the display buffer. The resulting pixel value may index into an undefined entry in the color palette; the color might therefore be indistinguishable from the background. The blinking caret might behave the same way if displayed over colored backgrounds.

When you set the per-window attribute ATTR_XOR_REF_COLOR to a particular color, XVT/XM calculates the pixel value used for XOR-mode drawing so that if you XOR it with the pixel value corresponding to the attribute, the application produces a pixel with the foreground color set in the drawing tools. That is, if you set ATTR_XOR_REF_COLOR to the assumed background color of a window on which you expect to perform XOR-mode drawing, you get the desired foreground color.

ATTR_XOR_REF_COLOR is set to COLOR_WHITE by default for each window.

Since the caret is a rectangle drawn in XOR-mode, it inherits the XOR-mode difficulties. To make the caret display properly against non-white backgrounds, set ATTR_XOR_REF_COLOR before calling any xvt_*_set_caret_visible function.

Note: Text edit objects use the ATTR_XOR_REF_COLOR attribute to display selections and when setting the caret. Since text edit objects manage their own events independent of the contained window, you should assume that this attribute can change any time you call any text edit function, including xvt_tx_process_event.

Example: Suppose that you have a window whose background is blue. Before doing any XOR-mode drawing, set ATTR_XOR_REF_COLOR to COLOR_BLUE. Since the attribute is implemented only on some platforms, ifdef the code as follows:

```
#ifdef ATTR_XOR_REF_COLOR
        set_value(win, ATTR_XOR_REF_COLOR, COLOR_BLUE);
#endif
... code to draw in XOR mode ...
```

Uses win argument:	Yes
xvt_vobj_get_attr returns:	Reference color
xvt_vobj_set_attr effect:	Desired foreground color (depending on background color of window to be XOR'd)
xvt_app_create use:	Must use after
Default value:	COLOR_WHITE

A.2. Variations on Portable Attributes

These portable attributes have slight variations in meaning in order to support differences on the native Motif platform.

ATTR_EVENT_HOOK

Description: A pointer to a hook function that is called whenever a native X event is generated for a window or dialog in your application. XVT calls the hook function after XtAppNextEvent and before XtDispatchEvent.

Your application can process this message data in any appropriate manner—however, modifying this data will have no effect on any default processing by XM. If your hook function returns FALSE, XVT does not process the event further. If your hook function returns TRUE, XVT processes the event normally.

Prototype: BOOLEAN event_hook_function(XEvent * xevent);

XEvent * xevent
Native event.

Uses win argument:	No
xvt_vobj_get_attr returns:	Pointer to function returning BOOLEAN or NULL
xvt_vobj_set_attr effect:	Installs hook or uninstalls hook if value is NULL
xvt_app_create use:	Can use either before or after
Default value:	NULL

See Also: You can see the internal ATTR_EVENT_HOOK function in the file **hook.c** in the **samples/hook** directory. You might want to use the code in this file as a template for your own custom function. The **X11/Xlib.h** file contains the XEvent definition.

ATTR_KEY_HOOK

Multibyte-nonaware Application

If your application uses a single-byte character code set and you have set the value of ATTR_MULTIBYTE_AWARE as FALSE (default), then ATTR_KEY_HOOK behaves as follows:

Description: A pointer to a hook function that is called *after* native xKeyEvent key events are received and *before* E_CHAR events are sent to your application. The xevent parameter is a pointer to data passed internally to Motif message procedures by XVT/XM.

If you need to perform key translation, you must modify data in the xevent parameter. Cast xevent to a variable of type XKeyEvent. Modify the values in the XKeyEvent variable. You may also modifiy the value of ch. The values of shift and control are currently ignored. XVT uses the xevent and ch parameters to construct an E_CHAR event.

If your hook function returns FALSE, XVT does not process the event further. If your hook function returns TRUE, XVT processes the event normally. In either case, the E_CHAR event is passed to your application's XVT event handler.

Prototype:

```
BOOLEAN XVT_CALLCONV1 key_hook(XEvent * xevent, int * ch,
    BOOLEAN * shift, BOOLEAN * control);
```

XEvent * xevent
: Native event.

int * ch
: Resulting character code.

BOOLEAN * shift
: Resulting shift state (ignored).

BOOLEAN * control
: Resulting control key state (ignored).

Uses win argument:	No
xvt_vobj_get_attr returns:	Pointer to function returning BOOLEAN or NULL
xvt_vobj_set_attr effect:	Replaces internal key hook function with a custom function or re-enables internal function if NULL
xvt_app_create use:	Can use either before or after
Default value:	NULL

Multibyte-aware Application

If your application is multibyte-aware (in other words, you have set the value of ATTR_MULTIBYTE_AWARE as TRUE), then ATTR_KEY_HOOK behaves as follows:

Description: A pointer to a hook function that is called *after* native xKeyEvent key events are received and *before* E_CHAR events are sent to your application. The xevent parameter is a pointer to data passed internally to Motif message procedures by XVT/XM.

If you need to perform key translation, you must modify data in the xevent parameter. Cast xevent to a variable of type XKeyEvent. Modify the values in the XKeyEvent variable. You may also modify the values of xvtevent. XVT uses the xevent parameter to construct an E_CHAR event. If your key hook function translates a character to a virtual key, then it should also set the event xvtevent->v.chr.virtual_key field to TRUE.

Multibyte characters are composed of one or more key presses. Your key hook function must return a value depending on the status of composed characters. If your hook function returns 0, then an error occured and XVT does not process the event further. If your hook function returns 1, then your hook function is done composing a character event and XVT processes the event normally. If your hook function returns 2, then your hook function is at an intermediate state in composing a character event and XVT should not dispatch the event yet.

Prototype:

```
int XVT_CALLCONV1 key_hook(XEvent * xevent,
    EVENT * xvtevent, WINDOW win);
```

XEvent * xevent
: Native event.

EVENT * xvtevent
: Resulting XVT event.

WINDOW win
: Event window.

Uses win argument:	No
xvt_vobj_get_attr returns:	Pointer to key hook function or NULL
xvt_vobj_set_attr effect:	Replaces internal key hook function with a custom function or re-enables internal function if NULL
xvt_app_create use:	Can use either before or after
Default value:	NULL

See Also: You can see the internal ATTR_KEY_HOOK function in the file **hook.c** in the **samples/hook** directory. You might want to use the code in this file as a template for your own custom function. The **X11/Xlib.h** file contains the XEvent definition.

ATTR_NATIVE_GRAPHIC_CONTEXT

Description: A value that represents the underlying graphics context (i.e., a GC) used by the native window system for a particular window. The window must be a valid XVT WINDOW that is not a control.

Uses win argument:	Yes
xvt_vobj_get_attr returns:	GC
xvt_vobj_set_attr effect:	Illegal
xvt_app_create use:	Must use after
Default value:	None

ATTR_NATIVE_WINDOW

Description: A value that represents the underlying window object (i.e., an X Window) used by the native window system, for a particular window. The window must be a valid XVT WINDOW that is not a control.

Note: For the XVT SCREEN_WIN, the value of this attribute is the X Window that corresponds to the root window (i.e., the desktop). For the XVT TASK_WIN, the value of this attribute is zero.

Uses win argument:	Yes
xvt_vobj_get_attr returns:	Window (X type)
xvt_vobj_set_attr effect:	Illegal
xvt_app_create use:	Must use after
Default value:	None

A.3. Non-portable Escape Codes

The xvt_app_escape function enables you to set or get XVT/XM-specific information that you cannot set or get using the xvt_vobj_set_attr or xvt_vobj_get_attr functions. The function xvt_app_escape's escape codes and the associated parameter lists are given below, with a brief explanation of types and values. The escape code is an integer whose value is defined internally by XVT.

XVT_ESC_XM_GET_COMBO_WIDGETS

Description: This escape gives you non-portable access to the individual widgets that compose the WC_LISTEDIT and WC_LISTBUTTON controls.

Prototype:
```
xvt_app_escape(XVT_ESC_XM_GET_COMBO_WIDGETS, WINDOW win,
    Widget * text_wid, Widget * list_wid,
    Widget * arrow_button_wid, Widget * row_column_wid);
```

WINDOW win
: The WINDOW of a combo control.

Widget * text_wid
: A pointer to the widget that is the edit or label portion of the control to the left of the arrow button.

Widget * list_wid
: A pointer to the widget that is the list (XmList) portion of the control.

Widget * arrow_button_wid
: A pointer to the widget that is the arrow button portion of the control.

Widget * row_column_wid
: A pointer to the XmRowColumn widget that contains the text_wid and the arrow_button_wid.

Tip: If you need only one of the widgets, pass NULL pointers for the items you don't need when calling xvt_app_escape.

XVT_ESC_XM_GET_GRP_BOX_WIDGETS

Description: This escape retrieves the widgets within a WC_GROUPBOX.

Prototype:
```
xvt_app_escape(XVT_ESC_XM_GET_GRP_BOX_WIDGETS,
    WINDOW win, Widget * label_wid, Widget * frame_wid);
```

WINDOW win
: The WINDOW of a group box control.

Widget * label_wid
: A pointer to the widget that is the label on the group box.

Widget * frame_wid
: A pointer to the widget that is the area that encloses label_wid and any controls that are in it. Note that the frame_wid is not the parent of the controls that are in the group box.

Tip: If you need only one of the widgets, pass NULL pointers for the items you don't need when calling xvt_app_escape.

XVT_ESC_XM_LOWER_GRP_BOX_FRAME

Description: This escape lowers the group box so that it doesn't obscure any controls that may be underneath it. This function ensures that the group box appears lowest in the stacking order of its sibling controls so that it seems to contain them.

Prototype:
```
xvt_app_escape(XVT_ESC_XM_LOWER_GRP_BOX_FRAME,
    WINDOW win);
```

WINDOW win
: The WINDOW of a group box control.

XVT_ESC_XM_PICT_TO_XIMAGE

Description: This escape extracts a pointer to an XImage (XImage *) from an XVT PICTURE. XVT/XM provides this escape specifically for backward compatibility with Release 3. Your application must explicitly destroy the XImage (see XDestroyImage in O'Reilly's *Xlib Reference Manual*).

Prototype:
```
xvt_app_escape(XVT_ESC_XM_PICT_TO_XIMAGE,
        PICTURE pict, XImage ** ximage);
```

PICTURE pict
: The picture from which to extract the XImage.

XImage ** ximage
: A pointer to an XImage pointer. The XImage * is returned through this argument.

Example: This example shows how to use the XVT_ESC_XM_PICT_TO_IMAGE attribute:

```
PICTURE pict;
XImage * ximage;
...
/* create an empty XImage of the correct size */
xvt_pict_open(win, &rct);
xvt_dwin_clear(win, COLOR_WHITE);
pict = xvt_pict_close(win);

xvt_app_escape(XVT_ESC_XM_PICT_TO_XIMAGE, pict,
                        &ximage);
xvt_pict_destroy(pict);
app_function_to_draw_into_ximage(ximage);
xvt_app_escape(XVT_ESC_XM_XIMAGE_TO_PICT, * ximage,
                        &pict);
xvt_dwin_draw_pict(win, pict, &drawrct);
xvt_pict_destroy(pict);
...
```

XVT_ESC_XM_SET_CTL_BKG_COLOR

Description: This escape sets the background color for a control. It is particularly useful for controls created in windows, because you can use it to set the control's color to blend in with the window.

Prototype: xvt_app_escape(XVT_ESC_XM_SET_CTL_BKG_COLOR, WINDOW win, COLOR color);

WINDOW win
: The WINDOW of a control.

COLOR color
: An XVT COLOR value.

See Also: For more information about the portable way to set the background color for a control, see the description of xvt_ctl_set_colors in the online *XVT Portability Toolkit Reference*.

XVT_ESC_XM_XIMAGE_TO_PICT

Description: This escape converts an XImage to an XVT PICTURE. XVT provides this escape specifically for backward compatibility with Release 3. When your application is finished with the PICTURE returned by this escape, the application must use xvt_pict_destroy to destroy the PICTURE.

Prototype: xvt_app_escape(XVT_ESC_XM_XIMAGE_TO_PICT, XImage * ximage, PICTURE * pict);

XImage * ximage
: A pointer to an XImage structure. The XImage is converted to an XVT PICTURE.

PICTURE * pict
: A pointer to an XVT PICTURE. The PICTURE produced from the XImage is returned through this argument.

Example: See the example for the escape code XVT_ESC_XM_PICT_TO_XIMAGE.

A.4. Non-portable MENU_ITEM Fields

The MENU_ITEM structure contains some non-portable, platform-specific fields that supplement the portable fields described in the online *XVT Portability Toolkit Reference*. This section describes the non-portable fields for XVT/XM.

Prototype:

```
typedef struct s_mitem {
    ...
    char * accel;           /* menu accelerator */
    char * acceltext;       /* accelerator string */
    ...
} MENU_ITEM;
```

accel String

accel is a string that describes the MENU_ITEM accelerator. The format is the same as the translation table syntax outlined in O'Reilly's *X Toolkit Intrinsics Programming Manual.* XVT/XM uses the modifiers Shift, Control and Meta (XVT's Alt).

Example: An accelerator of Control-Shift-F5 is "CTRL Shift <key>F5". Similarly, an accelerator of Alt-a is "META <key>a".

acceltext String

acceltext is a string that replaces the default string generated by any accelerators. This string appears to the right of the button label.

Note: When a MENU_ITEM is returned from xvt_menu_get_tree, both the accel and acceltext fields are pointers to space that has already been allocated. As a result, the application cannot concatenate characters onto the end of an accel or acceltext string. If the application modifies the MENU_ITEM structure to contain different accel or acceltext strings, adequate space must be allocated for the strings. The function xvt_res_free_menu_tree attempts to free these strings if the pointers are non-NULL.

ATTR_X_PROPAGATE_ECHAR

Description: This attribute controls whether character events are passed to a window's event handler when a control has focus. When this attribute is FALSE, only navigational characters, such as TAB, are sent the the window's event handler as an E_CHAR event. Setting this attribute to TRUE will send all character events generated.

Uses win argument:	No
xvt_vobj_get_attr returns:	BOOLEAN
xvt_vobj_set_attr effect:	Window receives all character events for controls
xvt_app_create use:	Must use before
Default value:	FALSE

ATTR_X_R45_MODALITY

This attribute allows for the XVT release 4.5 modal model to be instated. The model for modality changed after XVT release 4.5. In release 4.5 all objects in the application were sensitized when a modal window or dialog was created. In releases after 4.5, only menubars are sensitized.

Uses win argument:	No
xvt_vobj_get_attr returns:	BOOLEAN
xvt_vobj_set_attr effect:	Changes the modality model
xvt_app_create use:	Must use before
Default value:	FALSE

ATTR_X_EXPOSE_COMPRESSION_TYPE

Description: This attribute allows three compression types for exposure events: XVT_COMPRESS_NONE, XVT_COMPRESS_ALL, and XVT_COMPRESS_OPT. XVT_COMPRESS_NONE does not compress any exposure events. XVT_COMPRESS_ALL will compress all qued exposure events into a single rectangular area. XVT_COMPRESS_OPT will compress all queued exposure events into the smallest single rectangular area.

Uses win argument:	No
xvt_vobj_get_attr returns:	long
xvt_vobj_set_attr effect:	Changes compression type
xvt_app_create use:	Can use before or after
Default value:	XVT_COMPRESS_ALL

ATTR_X_TABLE_PROPORTIONAL_THUMB

Description: This allows an XPO table (DSC) or CCTable (DSC++) thumb to be set to proportional or fixed.

Uses win argument:	No
xvt_vobj_get_attr returns:	BOOLEAN
xvt_vobj_set_attr effect:	Changes table thumb
xvt_app_create use:	Must use before
Default value:	TRUE

A

APPENDIX B: THE XVT/XM LOOK-AND-FEEL

A.1. Focus Models

This section provides information on Motif focus models and how the focus models affect the interpretation of keyboard events.

A.1.1. Window Managers and Input Focus

The window manager is responsible for moving input focus among the various windows on the screen. Most window managers give the user a choice of focus policies. Motif's window manager, **mwm**, offers two choices, "explicit" and "pointer":

Explicit focus (click-to-type model)
The user must click the mouse in a window or on one of its borders to switch the focus to that window.

Pointer focus (pointer-driven model)
The window manager gives focus to a window when the mouse pointer enters a window and removes focus when the pointer leaves.

The window manager is always in control of distributing focus among the windows it manages. Thus XVT top-level windows and dialogs always receive focus from the window manager based on the focus policy defined in the user's environment.

A.1.2. Child Windows or Controls

The window manager has no control over the distribution of input focus within a window hierarchy. If a window contains child windows or controls, the window manager's focus policy does not

extend to child windows. Within the hierarchy, the application is responsible for moving focus.

A.1.3. Keyboard Events

Even though the window manager can't give focus to child windows, X can still send keyboard events to them. This is possible because X does not require that a window receive a FocusIn event prior to receiving keyboard events. The X server delivers keyboard events to the window with focus or to any of its descendants if they contain the mouse pointer.

Figure A.1 shows a window hierarchy in which a window A has a child B, and window B has a child C.

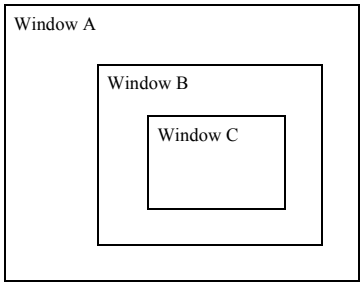


Figure A.1. A window hierarchy

For the windows illustrated above, Table A.1 shows which window receives keyboard events, based on the position of the mouse pointer. Each cell indicates the window that will receive characters from the keyboard, given the location of the pointer and the window that has focus.

Keyboard Focus in a Window Hierarchy

		Location of Pointer		
		A	**B**	**C**
Window with focus	**A**	A	B	C
	B	B	B	C
	C	C	C	C

Table A.1. Windows receiving keyboard events

Window A is a top-level window, and thus it can get focus from the window manager. If the application does nothing to move the focus (i.e., does not call xvt_scr_set_focus), X delivers keyboard events as if a pointer-driven model is in effect, even though the window manager is not changing input focus.

A.2. XVT/XM Focus Policy

The Motif Window Manager always gives focus to the outermost widget (or TopLevelShell widget) of the window hierarchy. Motif then uses a virtual focus model to deliver keyboard events. In other words, Motif uses the X Intrinsics toolkit to redirect keyboard events to the object (window or control) that has been assigned as the focus object (even though the X server's idea of who has focus hasn't changed).

Consequently, if your application is using ATTR_EVENT_HOOK to watch for KeyPress or KeyRelease events, they can be delivered to any X Window in your application. The Xt Intrinsics library will redirect the events appropriately.

A.2.1. Default Focus Model

In XVT/XM (as well as Motif), the focus model defaults to a click-to-type model. For example, in Figure A.1, if window A has focus, all character events are delivered to window A, regardless of the position of the mouse pointer.

See Also: For a comparison of the two focus models used by Motif, refer to section A.1.1.

A.2.2. Changing Focus

To give focus to one of the other windows, the user must move the pointer and click the mouse button in it. The application then receives an E_FOCUS event with the active field equal to FALSE for the previous focus window, and a subsequent E_FOCUS event with the active field equal to TRUE for the window in which the mouse was clicked. The new focus window receives all subsequent E_CHAR events. Clicking on a control also gives it focus and ensures that its parent receives the correct E_FOCUS event, if necessary.

If focus leaves the window hierarchy and then returns, the focus is restored to the last window or control that had the focus. (The focus is not returned to the client area of the top-level window.)

A.3. Keyboard Navigation

Keyboard navigation in dialog controls is handled automatically by XVT/XM. No special processing of characters is required.

On the other hand, keyboard navigation is not automatic in XVT window controls. You may elect to use the XVT navigation object to handle E_CHAR events for keyboard navigation in windows, or you may implement your own navigation mechanism.

Caution: If you implement you own navigation scheme, there are some considerations you must keep in mind—these are discussed in the following two subsections, next.

When a control in a window has focus and the user types characters, characters not processed internally by the control are passed as E_CHAR character events to the control's parent (container) window. Your application event handler then must process these characters for the desired behavior (focus change, selection, etc.).

The portable attribute ATTR_PROPAGATE_NAV_CHARS controls the delivery of those character events necessary for navigation to windows (including modal windows). This attribute is automatically set if you have chosen to use the XVT navigation object.

See Also: For more information about XVT's navigation object, XVT_NAV, see the "Keyboard Navigation" section of the "Windows" chapter in the *XVT Portability Toolkit Guide*.

A.3.1. Controls and Navigation Keys

Different controls propagate different navigation keys as E_CHAR events to their parent window:

- All controls send the following keys: Tab, Back Tab (XVT's K_BTAB), and osfCancel (usually Escape).
- All controls except WC_LBOX also pass osfActivate (usually Return) to the application. A WC_LBOX treats osfActivate as a double-click.
- The following controls also dispatch arrow keys as E_CHAR events: WC_PUSHBUTTON, WC_RADIOBUTTON, WC_CHECKBOX, WC_TEXT, WC_ICON, and WC_LISTBUTTON. The other controls interpret arrow keys as traversal within the widget.

A.3.2. Application Focus Traversal Lists

Note: This section applies only if you are not using XVT's navigation object, XVT_NAV, and instead, have chosen to implement your own mechanism for keyboard navigation.

Since the user can change focus to controls by clicking on them, the control that currently contains the focus can be "out of sync" with any application traversal list. If the application relied on an internal list to decide the control that should receive focus when the user presses the Tab key, the focus would likely jump to a control that the user didn't expect.

Instead, your application should use xvt_scr_get_focus_vobj to "inquire" which control (if any) has focus, and hence to which control the focus should be directed when the user traverses via the keyboard.

A.4. Task Window Menubar

In Motif applications, when no visible window with a menubar is displayed, a "ghost window" (or ghost menu) appears. The ghost window exists solely to display the task window menubar.

The ghost window is necessary because in Motif the task window has no physical representation, but simply corresponds to the screen. A user would be unable to make menu selections (or open windows) if no window (in this case, the ghost window) containing a menubar were present.

As soon as the user opens a visible window containing a menubar, the ghost window disappears.

A

APPENDIX C: FREQUENTLY ASKED QUESTIONS

Q: *I've just upgraded from an earlier release of XVT/XM. When I recompile my application, I get an XVT Fatal Error. What's wrong?*

A: Each time you upgrade to a newer version of XVT/XM, you must not only recompile your programs with the new libraries, but also rerun **xrc** on your URL file. The resource IDs of the strings that your application is looking for change slightly from release to release. Delete your **.uid** file and rerun **xrc** on your URL file.

Q: *Why do I get an XVT Fatal Error followed by "Can't open prolog file" when I try to print?*

A: This problem is the result of an invalid XVTPATH variable. In order to print, you must set XVTPATH to include the **print** directory of the XVT library installation.

Example: This example shows how to set the environment variable (inside a C-shell) to a fictitious path for printing:

```
setenv XVTPATH /xvtdsc45/xm12_x86_sco/print
```

See Also: For more information about printing, see section 2.6.

Q: *Why am I getting the message: "Warning: Couldn't open file xxx.uid - MrmNOT_FOUND" followed by an XVT Fatal Error?*

A: This message appears when your application cannot find its resources. For an application to recognize the location of its resources, you need to either start the application from the directory that contains the UID file, or you need to set the UIDPATH environment variable.

See Also: For more information on resource filenames and UIDPATH, see section 2.4.1.

Q: *How do I use multiple UIL/UID files with XVT/XM?*

A: Follow these two basic rules if you create multiple UIL files for your application:

- Divide your UIL file into multiple files
- Change **xxinit.c** to recognize multiple UID files

The easiest way to divide your large UIL file into multiple UIL files is to cut-and-paste your dialogs into another file. Move only *your* dialogs. Leave all the menubars, strings, help, standard dialogs and any information at the bottom of the file, such as value MWS_xxxxx, in the first UIL file.

You can find the beginning of any of your dialogs by looking for XVT_DLG_xxx (where xxx is the resource ID assigned to the dialog). Then cut-and-paste the dialog and all of its controls. The controls are labeled control_xxx_yy or sssss_xxx_yy, where sssss is the type of control that is being created, xxx is the dialog ID, and yy is the control ID (e.g., list_256_18 for a listbox, label_256_20 for a static text, and so forth). Be sure to cut-and-paste *all* the controls of a particular dialog, since you can't have a dialog split between UIL files.

You must include the first 40 lines of the first UIL file in the second and any subsequent UIL files.

Note: The first 14 lines are comments; you can omit them if desired.

The required lines from the first UIL file (without the optional comments) are shown here:

```
module APPL1
        names = case_sensitive value
! new line "macro" for creating multi-line strings
        NL          : compound_string('', separate = true);
! dialog/control creation flags
        dlg_disabled    : 1;
        dlg_invisible   : 2;
        ctl_readonly    : 1;
        arrow_button_height : 29;
        arrow_button_width  : 23;
        frame_shadow_width  : 3;

procedure
        xvt_create_dlg_cb(integer);
        xvt_create_travs_ctl_cb(integer);
        xvt_create_ctl_cb(integer);
        xvt_create_special_cb(integer);
        xvt_kbd_traversal(integer);
        xvt_motif_dlg_button_cb(integer);
        xvt_create_menu();
        xvt_do_menu();
        xvt_scroll_cb(integer);
```

Add the following lines to the bottom of the second and subsequent UIL files:

```
end module;

! ------------------------------------------------
!                   End of file
! ------------------------------------------------
```

The last three lines are comments and can be omitted if desired.

Once you have split your UIL file, you need to change the **xxinit.c** file to recognize all of your UID files. Remember—there is a one-to-one correspondence between UIL and UID files.

Tip: To change the **xxinit.c** file to recognize multiple UID files:

Change the array size of the variable uidfiles in the function xvtxm_app_init in **xxinit.c**.

The default array of one char * is already declared. Add additional strings to the uidfiles array to indicate you are using multiple UID files. Change the number being passed to MrmOpenHierarchy to reflect the exact number of UID files you are using.

Example: The existing xvtxm_app_init code shown here opens the UID file:

```
/* open the Mrm hierarchy */
uidfiles[0] = xvt_malloc ((strlen(name) + 5)
                                    * sizeof(char));
sprintf (uidfiles[0], "%s.uid", name);
if (MrmOpenHierarchy ((MrmCount)1, uidfiles, NULL, hier)
                                    != MrmSUCCESS)
    return FALSE;
```

After splitting the example UIL file into two files, **examp.uid** and **examp1.uid**, you would change the code in xvtxm_app_init as follows:

```
/* open the Mrm hierarchy */
uidfiles[0] = xvt_malloc ((strlen(name) + 5) *
                                    sizeof(char));
sprintf (uidfiles[0], "%s.uid", name);
uidfiles[1] = xvt_malloc ((strlen("examp1") + 5) *
                                sizeof(char));
strcpy(uidfiles[1], "examp1.uid");
if (MrmOpenHierarchy ((MrmCount)2, uidfiles, NULL, hier)
                                != MrmSUCCESS)
    return FALSE;
```

Q: *How do I use color with controls in my application?*

A: You can use the following two Portability Toolkit functions to set colors for controls in your application:

```
void xvt_ctl_set_colors(WINDOW ctl_win,
                        // WINDOW ID of the control
  XVT_COLOR_COMPONENT *colors,
                        // colors to set or unset
  XVT_COLOR_ACTION action)
                        // set or unset the colors
```

and

```
void xvt_win_set_ctl_colors(WINDOW win,
                        // WINDOW ID of the window or dialog
  XVT_COLOR_COMPONENT *colors,
                        // colors to set or unset
  XVT_COLOR_ACTION action)
                        // set or unset the colors
```

xvt_ctl_set_colors sets or unsets the colors for a single control. This function overrides any color values you set previously for the control, but only for the XVT_COLOR_COMPONENT of the colors array. All other colors used by the specified control are not affected. To set the default colors for a control, use NULL for the value of colors. An action value of XVT_COLOR_ACTION_SET sets the control

colors for the color components specified in the colors parameter. An action value of XVT_COLOR_ACTION_UNSET sets the control colors for the color components specified in the colors parameter to colors inherited from the control's container, the colors owned by the application, or the system default.

xvt_win_set_ctl_colors sets or unsets the colors for all existing controls in window win and all controls that you create after setting the colors. It will not change the colors of controls in other windows. This function overrides any color values you set previously for the controls in the window, but only for the XVT_COLOR_COMPONENT of the colors array. All other colors used by the window's control are not affected.

Note: For controls with color components set individually, the components that *were* set will not be affected by this color change. The components that *were not* set will be affected. For example, if a pushbutton has set blue for the foreground color and then the window has set a background component of red, the background of the pushbutton will be red.

To set the default colors for controls in a window, use NULL for the value of colors. XVT_COLOR_ACTION_SET and XVT_COLOR_ACTION_UNSET work as described above. Note that this function does not affect the colors of the container decorations or any other colors that appear in the container itself.

The following Portability Toolkit functions allow you to get the currently-defined color settings:

```
XVT_COLOR_COMPONENT *xvt_ctl_get_colors(
      WINDOW ctl_win)

XVT_COLOR_COMPONENT *xvt_win_get_ctl_colors(
      WINDOW win)
```

Q: *When compiling my application with the XVT PTK 4.x release, I sometimes get the following XVT internal warning:*

```
WARNING: API function already marked in frame
Category: Error messaging facility (Error Message Frame problems)
Function: xvt_app_process_pending_events
xvt_app_process_pending_events
File: ./verrmsg.c line: 405
```

What does this mean and how can I correct the problem?

A: With error handling in XVT PTK 4.x, each time a function call is made, it is “marked.” When it returns, it is “unmarked,” so that XVT can report which call caused the error.

Since 4.x was released, we have learned that the marking and unmarking of function calls does not happen correctly in certain cases, particularly, in cases where the application interacts with the X toolkit directly in such a way that causes recursion. Thus, the warning occurs. The warning is harmless and should not affect the operation of the application at all.

To prevent the message, however, you can override the error message handler by creating one that filters out the warning message. You can install such a message handler from the window event handler for C customers, as follows:

```
static BOOLEAN                  XVT_CALLCONV1
ErrHandler                      XVT_CALLCONV2
#ifdef XVT_CC_PROTO
(
XVT_ERRMSG   err,               /* Error Message Object */
DATA_PTR     context            /* Context, (not used here) */
)
#else
( err, context )
XVT_ERRMSG   err;
DATA_PTR     context;
#endif
{
        /* Check for error signal(s) we want to ignore */
                if (xvt_errmsg_get_msg_id(err) ==
                ERR_EMF_FRAME_MARKED)
      return TRUE;     /* forget this message,
                        it's OK */
        /* Pass the remaining signals to the default
                handler */
   return FALSE;
}
```

You can also override the warning in the task window event handler or prior to calling xvt_app_create do, as follows:

```
case E_CREATE:
xvt_vobj_set_attr (win, ATTR_ERRMSG_HANDLER,
        (long)ErrHandler);
```

If you are using C++, you can install an error handler by placing the following line in header file:

```
BOOLEAN ErrHandler(XVT_ERRMSG err, DATA_PTR context);
```

Place the following line in an implementation file after the #include statements:

```
extern BOOLEAN ErrHandler(XVT_ERRMSG err,
        DATA_PTR context);
```

Place the following function definition in the implementation file:

```
BOOLEAN             XVT_CALLCONV1
ErrHandler          XVT_CALLCONV2
#ifdef XVT_CC_PROTO

(
XVT_ERRMSG   err,           /* Error Message Object */
DATA_PTR     context        /* Context, (not used here) */
)
#else
( err, context )
XVT_ERRMSG   err;
DATA_PTR     context;
#endif
{
   /* Check for error signal(s) we want to ignore */
   if (xvt_errmsg_get_msg_id(err) == ERR_EMF_FRAME_MARKED)
      return TRUE;    /* forget this message, it's OK */

   /* Pass the remaining signals to the default
            handler */
   return FALSE;
}
```

Then place the following line in **cstartup.cxx** after the #includes (You need to have a header file with the function prototype included):

```
extern BOOLEAN ErrHandler(XVT_ERRMSG err,
      DATA_PTR context);
```

Finally, in CApplication() theApplication:

```
CApplication0 theApplication;
// add the following ATTR statement here
xvt_vobj_set_attr (NULL_WIN, ATTR_ERRMSG_HANDLER,
      (long)ErrHandler);
theApplication.Go(
```

In summary, install the event handler so that it ignores the warning message.

Q: *How do I use color with controls in my application?*

A: You can use the following two Portability Toolkit functions to set colors for controls in your application:

```
void xvt_ctl_set_colors(WINDOW ctl_win,
      // WINDOW ID of the control
XVT_COLOR_COMPONENT *colors,
      // colors to set or unset
XVT_COLOR_ACTION action)
      // set or unset the colors
```

and

```
void xvt_win_set_ctl_colors(WINDOW win,
        // WINDOW ID of the window or dialog
XVT_COLOR_COMPONENT *colors,
        // colors to set or unset
XVT_COLOR_ACTION action)
        // set or unset the colors
```

xvt_ctl_set_colors sets or unsets the colors for a single control. This function overrides any color values you set previously for the control, but only for the XVT_COLOR_COMPONENT of the colors array. All other colors used by the specified control are not affected. To set the default colors for a control, use NULL for the value of colors. An action value of XVT_COLOR_ACTION_SET sets the control colors for the color components specified in the colors parameter. An action value of XVT_COLOR_ACTION_UNSET sets the control colors for the color components specified in the colors parameter to colors inherited from the control's container, the colors owned by the application, or the system default.

xvt_win_set_ctl_colors sets or unsets the colors for all existing controls in window win and all controls that you create after setting the colors. It will not change the colors of controls in other windows. This function overrides any color values you set previously for the controls in the window, but only for the XVT_COLOR_COMPONENT of the colors array. All other colors used by the window's control are not affected.

Note: For controls with color components set individually, the components that were set *will not* be affected by this color change. The components that *were not* set *will* be affected. For example, if a pushbutton has blue set for the foreground color and the window has red set for the background color, the background of the pushbutton will be red.

To set the default colors for controls in a window, use NULL for the value of colors. XVT_COLOR_ACTION_SET and XVT_COLOR_ACTION_UNSET work as described above. Note that this function does not affect the colors of the container decorations or any other colors that appear in the container itself.

The following Portability Toolkit functions allow you to get the currently-defined color settings:

```
XVT_COLOR_COMPONENT *xvt_ctl_get_colors(
        WINDOW ctl_win)
```

and

```
XVT_COLOR_COMPONENT *xvt_win_get_ctl_colors(
        WINDOW win)
```

Q: *Where are all new features of the PTK documented?*

A: New functionality is outlined in the *XVT Portability Toolkit Reference* and in the *XVT Portability Toolkit Guide*.

In addition to documenting new functionality, the online *XVT Portability Toolkit Reference* contains sections on each of the following topics:

- XVT Portable Attributes
- XVT Events
- XVT Data Types
- XVT Constants
- XVT Functions
- URL Statements
- Help File Statements
- Tools

Q: *How do standard fonts map to multibyte fonts?*

A: XVT does not automatically map to multibyte fonts. In order for your application to use multibyte fonts, you must first Internationalize and Localize your application, using the methods detailed in Chapter 19 of the *XVT Portability Toolkit Guide*. You must also install the multibyte fonts appropriate for the language you intend to use, according to your system guidelines. This will allow the fonts to be available to your XVT application.

Presumably, you will be translating your application to one or more languages. If you have properly internationalized your application, all your text and font references exist only in your resource file. When you translate your text, you should also setup the font and font_map resource appropriate for each language.

To set a multibyte font, you must modify the URL font or font_map statements of your application to contain native fonts appropriate for the language.

XVT supplies the following LANG_* xrc compiler options (files in your **ptk/include** directory):

- LANG_JPN_SJIS supports Japanese in Shift-JIS code (file **ujapsjis.h**)
- LANG_GER_IS1 supports German in ISO Latin 1

- LANG_GER_W52 supports German in Windows 1252
- Files for English, French, and Italian are also provided

These options and others are listed and discussed further in the *XVT Portability Toolkit Guide* and the *Guide to XVT Development Solution for C++*.

XVT cannot guarantee which character set your customers will use. There is more than one set available for many languages. Because the font to which you map must be available on your customer's system in order for your application to run, a survey of your proposed customer base may be in order.

The availability of these fonts and other system setup issues should become part of the installation requirements for your application, or the fonts should be installed with your application.

Q: *I've completed development and thoroughly tested my application. I understand the XVT Portability Toolkit has compile time optimization. How do I enable it?*

A: In order to understand how XVT compile time optimization works, some knowledge of the XVT Portability Toolkit implementation is required. The XVT Portability Toolkit is implemented in two layers. The top API layer, the functions of which are listed in the *XVT Portability Toolkit Reference*, is called directly by your application. This layer performs error checking of all input parameters and sometimes other validation before calling the internal layer. It is the internal layer that contains the implementation of the functionality.

XVT provides a compile time symbol, XVT_OPT, which, when defined during application compilation, redefines the top level function names to directly call the internal API functions through macros. This bypasses the parameter checking provided by the top layer and eliminates an extra stack level for each XVT API function. You can also leave XVT_OPT undefined, allowing for the specific optimization of your application code. The header file **xvt_opt.h** contains the macro definitions of the XVT API functions that are optimized.

The optimization will not eliminate all error checking from the XVT Portability Toolkit. Rather, it will eliminate only those errors related to XVT API function parameters. Also, because the top layer sets up the error frames for function information, any errors that do occur may have fictitious results for the function stack trace.

XVT recommends this option be used only after you have completed development and have thoroughly tested your application. Attempting to use this option too early in your development process may result in application crashes and other odd behavior due to improperly called functions that would otherwise have been checked and diagnosed by the top API layer.

Q: *How are the text edit objects after version 4.5 different from the text-edit objects in previous versions?*

A: Text-edit controls have been enhanced to work more like other objects in the XVT Portability Toolkit. The text edits after 4.5 have two improvements over those in previous versions. First, they have been placed inside a child window. Second, you can now use some of the same routines to manipulate controls and text-edits. Additionally, if you find you still need the previous types of text-edits, you can continue to use them.

In releases after 4.5, text-edits have been placed inside a child window, ensuring that they have more consistent behavior with other controls. For instance, the insertion point that appears in editable controls now acts more consistently. You can be assured that only one control will possess the insertion point at any one time. Also, to maintain backward compatibility, the previous text-edit functions will still work. That is, you may continue to use text-edit controls that are not contained in a child window. To use old text-edit features, read about using the attribute ATTR_R40_TXEDIT_BEHAVIOR in the *XVT Portability Toolkit Guide*.

Since the old-style text-edit is fundamentally different from other controls, it required specialized text-edit functions. However, you can manipulate the new text-edit features with many of the generic control and window functions. For instance, with the old-style text-edit, code would have to decide whether a text-edit or some other control should receive focus, requiring the use of two functions, xvt_tx_set_active() and xvt_scr_set_focus_vobj(). The new text-edit lets you use xvt_scr_set_focus_vobj() to clean up some code. xvt_scr_set_focus_vobj() can be helpful if an application needs to handle arrays of native controls mixed with text-edits.

Note: Not all the ctl functions work on text-edits. Check the *XVT Portability Toolkit Guide* for specific functions you can use.

Q: *I'm not sure I understand the M_* values for DRAW_MODE as stated in the XVT Portability Toolkit Reference. What exactly am I supposed to see?*

A: The following "Draw Mode Definitions" section shows the different drawing modes supported by XVT. There is also an explanation of what these modes will do if you are drawing in black or white on either a black or a white source pixel.

See Also: For more information, see "Draw_Mode" under "XVT Data Types" in the XVT Portability Toolkit Reference.

Note: On systems that use a 256-color palette, and not 24 bit color, information in the charts will hold true only for black and white because the palette indices are used for ORing and XORing, not the color values themselves. Because there are no definitive (or at least portable) rules about what color is held in a given index, there are absolutely no guarantees as to what your results will be. 129 xor 1 will always be 128, but index 129 might be yellow, 1 might be white, and 128 might be off-puce. The application can attempt to force a palette, but the colors present will be a random mix based on what applications are currently running and what applications have run in the past in the same session.

The following code and draw mode definitions demonstrate the problem more clearly:

```
typedef enum {   /* drawing (transfer) modes */
        M_COPY,
        M_OR,
        M_XOR,
        M_CLEAR,
        M_NOT_COPY,
        M_NOT_OR,
        M_NOT_XOR,
        M_NOT_CLEAR
} DRAW_MODE;
```

Draw Mode Definitions

M_COPY:

- If you draw black, source pixel will be forced to black
- If you draw white, source pixel will be forced to white

M_OR:

- If you draw black, source pixel will be forced to black
- If you draw white, source pixel will be left as is

M_XOR:

- If you draw black, source pixel will be inverted
- If you draw white, source pixel will be left as is

M_CLEAR:

- If you draw black, source pixel will be forced to white
- If you draw white, source pixel will be left as is

M_NOT_OR:

- If you draw black, source pixel will be left as is
- If you draw white, source pixel will be forced to black.

M_NOT_CLEAR:

- If you draw black, source pixel will be left as is
- If you draw white, source pixel will be forced to white

M_NOT_COPY:

- If you draw black, source pixel will be forced to white
- If you draw white, source pixel will be forced to black

M_NOT_XOR:

- If you draw black, source pixel will be left as is
- If you draw white, source pixel will be inverted

Q: *What is the difference between NText and CText?*

A: NText and CText each display a single line of text and provide alignment options within their frames. Although their basic functions are similar, each class has unique characteristics that make it better than the other in different situations.

The NText class is derived from the CNativeView class. Native views have the look-and-feel of objects provided by the native window manager. They look slightly different from platform to platform. Visually and functionally they fit in with the analogous graphical items on the target platform. They are not implemented by XVT-DSC++, but by the native toolkit, so you have less flexibility in manipulating them. Native views don't know how to print themselves. Since native views are derived from CView, they have all of the capabilities of other objects at the view level. As a native view, NText is defined by platform-specific resources. For example, it uses the system font and color as defined by the window manager.

You should use NText when you want your application, or parts of your application (certain dialog boxes, for example), to have the look-and-feel of objects created by the native window manager.

The CText class is derived from the CView class. Unlike NText, which is drawn by the native window manager, CText creates drawn text which looks the same across all platforms. It allows user and program control over its font properties and colors. For example, it allows you to choose from a variety of font families (Times, Helvetica) and styles (italics, boldface). It can dynamically change its size as its contents change. It can change its placement and alignment at runtime. It can also output itself to a printer.

You should use CText when you want more creative control over the appearance of your text, when you want your text to appear the same across all platforms, or when you want to give the user creative control over the appearance of text in your application.

See Also: For more information, see "CText" and "NText" in the *XVT Development Solution for C++ Reference* and also look for references to CText and NText in the *Guide to XVT Development Solution for C++*.

The "Textual Views" chapter in *Introduction to C++ for Developers* is also helpful.

Q: *Is there a way to implement zooming in DSC++?*

A: The following solution does not use CUnits and will result in correctly updated wireframes, scrolling, sizing, dragging, and so on.

Create a new class called ViewInfo, for example. The purpose of ViewInfo is to keep track of the location where the view was created. Each time that a new view is inserted in the CScroller, create an associated ViewInfo. Fill the associated ViewInfo with the view's creation-frame and a pointer to this view. This ViewInfo instance is then appended to a RWOrdered.

When the zoom factor changes, for example, to 150%, iterate through the RWOrdered, and tell the view, which is pointed to size 1.5 times its original frame. Once all views have processed, call xvt_dwin_invalidate_rect on the CScroller. Everything should successfully redraw. If a CWireFrame has been moved, it generates a WFSizeCmd, and the DoCommand looks up in the RWOrdered to update the creation coordinates according to the actual zooming factor.

The following code illustrates:

```
class ViewInfo : public RWCollectable
{
public:
        ViewInfo( CView* theView, const CRect& theRect ) ;
        ~ViewInfo( ) ;
        virtual CRect& SetFrame( const CRect& theRect ) ;
        virtual CRect GetFrame( void ) ;
        virtual CView* GetView( void ) ;
protected:
        CRect itsCreationFrame;
        CView* itsView;
private:
} ;
```

A fundamental problem is equating the Size() method with zooming. Here are the issues:

- What happens when a view is resized in the usual way? For example, as a pane in a splitter window, a subview may be resized to be twice as wide. Is this equivalent to zooming by 200%?
- What happens when a view is moved in the usual way? Will the associated ViewInfo object need to refresh itsCreationFrame? How would this be done?
- What happens when a native control is zoomed? For example, if a NListButton is told to zoom (resize), the edit box will remain the same height.
- What happens when a CPicture (or a CPictureButton, etc.) is told to resize? Will the picture stay intact?
- What happens to subviews within subviews? The splitter will be resized, but the oval will stay the same.

It should be possible to resolve all of these issues without the need to subclass everything. Expand on what has been started in the ViewInfo class above, and envision a type of visitor attached to the switchboard called a CZoomHandler.

A CZoomHandler will have a zooming factor attribute. If this is set to 100%, it will not do anything. A CZoomHandler will intercept E_UPDATE events at the Switchboard and perform a deep traversal through the window's object hierarchy, via DoDraw(). The CZoomHandler will render each view as it sees fit: On some views, it may just temporarily reset its size attributes and then call its Draw() method. On others, it may do its own drawing to handle some of the tougher issues listed above.

Q: *How do you create global variables for use in a DSC++ application?*

A: The best way to use variables that can be accessed globally from your application is to use them in a real global object, such as the CApplication- derived object. The application object should encapsulate the variables and make them accessible only through member functions. For instance, if the application object has a private variable named theVariable, then the application object might have a member function named SetTheVariable() and another called GetTheVariable(). This approach is a standard mode of operation in most object-oriented applications.

Some prefer to use the CGlobalUser class. This class, however, does not encapsulate and protect data as well as using a more object-oriented approach as described above. In case you choose to use the CGlobalUser class, the following paragraphs describe how.

The CGlobalUser class object has application global scope and can be used to access any global variables you may need. You can find documentation for this class in the *XVT Development Solution for C++ Reference*.

The CGlobalUser class is utilized as follows:

1. Copy the file **CGlobalUser.h** from the **pwr/include** directory to your development directory. You should rename the original file so that the compiler will see your own copy.
2. Add public class variables to your copy of the header file as follows:

```
//////////////////////
//Add items as needed//
//////////////////////

class CGlobalUser : public CNotifier
{
        public:
                CGlobalUser(void) {}
                XVT_HELP_INFO xd_help_info;
                FILE_SPEC* initFile;
                SECURITY_LEVEL userLevel;
};
```

3. In your application's startup member function, create an instance of CGlobalUser and pass it to CBoss: IBoss as follows:

```
////////////////////////////////
// Call IBoss to instantiate //
// the CGlobalUser object    //
////////////////////////////////

void CDEMOApp::StartUp();
{
        CApplication::StartUp();
        IBoss(new CGlobalUser);
        DoNew();
}
```

4. Access the global variables through the CBoss's GU pointer, as follows:

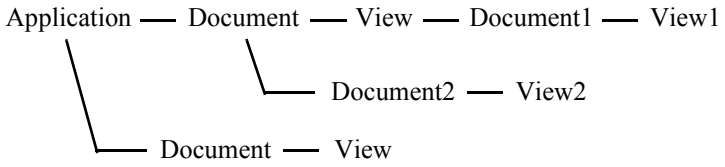
```
...
// Access the global userLevel
GU->userLevel = SUPER_USER;
...
```

5. Destroy the GU pointer in the application's ShutDown member function, as follows:

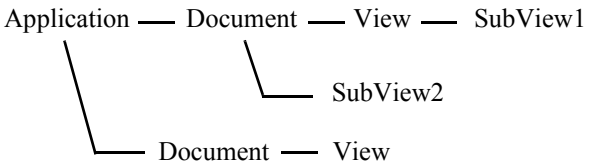
```
////////////////////////////////////
// Destroy GU and set it to NULL  //
////////////////////////////////////

void CDEMOApp::ShutDown(void)
{
        delete GU;
        GU = NULL;
        CApplication::ShutDown();
}
```

Q: *In the Application-Document-View hierarchy, can I have more levels of Document-View? For example, can I have a hierarchy like the following:*



In other words, if there are only three levels in the hierarchy, I have to put all data access/management code in one document and then use this single document to maintain all its views, as illustrated below?



A: No, you cannot have multiple levels of documents using the DSC++ framework. CDocument objects must be parented to one CApplicaton instance, just as CWindow objects are parented to a single CDocument instance. However, this arrangement gives you plenty of power for managing document data.

It might help to make a distinction between two different concepts that are used in the DSC++ framework. One is the "Application-Document-View" concept, and another is known as the "Model-View-Controller" design pattern. These two patterns can be used separately or together to build your application's data-flow structure.

It is true that you have only one level of views that are windows into the data in a document. However, it makes sense that there is only one level of complexity in this model. The real purpose of the App-Doc-View idea is to help the developer visualize which windows are looking at which separate groups of data.

In the App-Doc-View paradigm, it is the document's role to be the conduit of data flow between the data level and the presentation level of a two- or three-tier architecture. A document represents, in all its complexity, an entire, independent data set. Even if your presentation draws its data from several different sources, it can still be thought of as one data set, managed by a single document.

The App-Doc-View concept helps in laying out applications that have many windows that look into one data set, and a separate collection of windows that look into an entirely different data set. In your case, you may not have this type of complexity. More complex documents probably should be broken up into more manageable models, where the document manages (creates and destroys) these models. Each model is designed to solve one piece of the overall project.

In some cases, you may have a single window that looks into two separate data sets. In such a situation, the "Model-View-Controller" design pattern will be more appropriate. This design is borrowed

from the Smalltalk programming environment to help keep all the windows into a data set in sync so they all have the same data at the same time.

An MVC object structure can be as complicated as needed. When the state of one model changes, all other dependent models may be automatically notified and updated via the controllers to which the models are registered.

You can implement this with a document that owns many data models (use the CModel class). Each model has a controller that decides whether windows can change or read the data. You register each of the views with the data controller (use CController). These views can be implemented as CViews, CSubviews, or CWindows. When the data in the model changes, the controller will send a message to the appropriate views so that they can update themselves with the data. The document would manage both the data models and the views themselves.

In Architect, you can visualize the layout with the Application-Document-View graph. However there is no visual way to represent the Model-View-Controller idea in Architect because this design pattern has less to do with the layout of the application, and more to do with the internal data structures.

These two separate concepts have their own unique uses as generic design patterns. Thinking about object-oriented programs in terms of abstract design patterns has proven quite useful to many object-oriented programmers. A good book on the topic is *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides.

Q: *How do you print hidden views, multiple pages, or native controls in DSC++?*

A: The default behavior of Architect-generated code is to print the contents of a window on a single print page. DSC++ has built-in functionality for printing the screen images of most drawn or rendered objects, including CSubView-derived objects. Anything which lies beyond the boundaries of the window is clipped in the printed output. There is no functionality in DSC++ or the XVT Portability Toolkit (PTK) for printing images of native controls, specifically anything that inherits from the class CNativeView. If your application only needs to be able to produce simple screen shots of custom views drawn on a single page, you probably do not need to override any printing methods.

However, many applications need to be able to print text or graphics on multiple pages. Others may need to print portions of the view that demonstrate how to print the contents of a text editor object on multiple pages.

To understand Printing in DSC++, consider that there may be several overloaded versions of DoPrint acting in a single print process. CPrintManager has the DoPrint member function, CDocument has DoPrint, and CView also has DoPrint. These three implementations coexist, and they do different tasks. If you look at the DSC++ hierarchy, you can see they cannot override one another.

Printing is started when the user selects the Print option from the File menu. This generates the standard M_FILE_PRINT menu command. The menu command goes to your window's DoMenuCommand method. It propagates up from there to the default CWindow menu, then the default CDocument menu command. CDocument will then call the CDocument-derived DoPrint Method.

If your application overrides the CView::DoPrint, note that your overridden DoPrint function will not get called with the default Architect code. This is because the CView class does not inherit from CDocument, and it is the CDocument DoPrint that gets called by default. You will need to add your own code to call the proper print method. Usually this code is added at the window object level.

At the document level, the DoPrint method inserts each of the document's windows as an entry (or page) in the print queue, and calls the CPrintManager::DoPrint. The CPrintManager's DoPrint starts a PTK print thread. If you override CView::DoPrint, your function should also call CPrintManager::DoPrint.

The PrintThread function looks at every item in the print queue and opens a print page for each one. It calls an item's DoPrintDraw and then closes the page. This way each view in the queue gets exactly one page.

The default DoPrintDraw, generally at the CView level, simply sets the output device to be the printer, prepares the clipping region of the view, and calls the view's PrintDraw. PrintDraw is not called if the view is invisible. PrintDraw then does the drawing to the print page. In many cases, PrintDraw just calls Draw, the same routine that draws to the screen. Drawing to the screen or to the print page works interchangeably, depending on how the output device is set.

The secret to printing multiple pages it to override the DoPrint method that inserts pages into the print queue. Every time the CPrintManager's Insert method is called results in a page of printed output. If you want

a view to appear on several pages, call Insert once for each page with the same view as its parameter. If the objects to be printed are within a virtual frame, you can scroll hidden views into the visible portion of the frame before you enter the views in the print queue.

The overridden DoPrint also needs to figure out how many pages a view will occupy. Often times, this requires converting from printer dot units to pixel units. For this, you need to create a units object with dynamic mapping for your application.

Q: *How do I deallocate unused colors in an X application?*

A: Deallocation of unused colors is provided through XVT_PALETTE objects introduced in Release 4. Palettes of type XVT_PALETTE_USER will allow the application to allocate any available color cells. These color cells are deallocated when the palette object is destroyed. This gives applications some control over color cell allocation/ deallocation in a shared colormap. Currently, the palette implementation does not use private X colormaps, so complete "portable" control over all server colors is not possible. (XVT apps are "good X apps." They share all available color resources with all other X apps on the same server.)

Q: *I want to use an 8-bit character set from another country. How do I get XVT to modify this?*

A: Do the following:

1. Get a font in the new character set for X.
2. Set the locale for your new country using either setlocale(LC_ALL, "") or XtSetLanguageProc(NULL, NULL, NULL). (The locale setting changes certain items because of cultural differences, such as the way times and dates are displayed, the placement of decimals in numbers, etc.)

Note: The parameters may vary from Motif machine to machine.

   If you use setlocale(LC_ALL, ""), also use #include <locale.h>.

3. Set your LANG environment variable to the new locale. For example, for Norway: setenv LANG NORWAY.
4. Modify **xsstrings.h** to recognize the new font by altering the old FF_* font setup. You'll need to replace (not add to) one of the existing XVT defined fonts.

This will not affect programs like Design because they need to be rebuilt with the modifications in place in order for the changes to occur.

Note: This method will *not* work for multibyte character sets, like Japanese Kanji.

Q: *How can I change native widget characteristics of objects created by XVT?*

A: The following code will make an edit control non-editable and remove the blinking cursor:

```
Arg args[15];
int n;

n = 0;
  XtSetArg(args[n], XmNeditable, False);          n++;
  XtSetArg(args[n], XmNblinkRate, 0);             n++;
  XtSetArg(args[n], XmNcursorPositionVisible,
                    False); n++;
  XtSetValues(<WIDGET>,args,n);
```

where <WIDGET> is determined by using ATTR_X_WIDGET.

To change other Motif object characteristics check your Motif manual for specific names and settings:

Q: *When a palette is used in XVT for Motif platforms, the colormap is filled and other applications cannot allocate colors. Allocation of as few as sixteen colors can result in 200 colors being placed in the colormap, which can contain only 256 colors. The colormap may also be filled by use of a pixmap or image. How can I use colors in XVT for Motif and not fill the colormap?*

If you do not set attribute ATTR_DEFAULT_PALETTE_TYPE prior to starting your application, a default of XVT_PALETTE_STOCK is used. On Motif, this leads to problems with filling the colormap. To avoid the problem, set ATTR_DEFAULT_PALETTE_TYPE to XVT_PALETTE_CURRENT or XVT_PALETTE_USER prior to calling xvt_app_create.

See Also: For information on palette types, see 12.5.3 Color Palettes in the XVT Portability Toolkit Guide.

Note: XVT allocates colors on an application basis rather than a window basis in order to prevent flashing when different windows receive focus. All windows, therefore, share their palettes with each other and with all other applications. If some application has already

allocated a large number of colors, the XVT application will have fewer available to it for modification (in the event of XVT_PALETTE_USER). Your application should always check how many colors are available to it to modify, and when not enough are available, it should gracefully handle the condition either by reducing requirements or by alerting the user to close some applications and restart.

Q: *When an XVT error occurs, how do I stop the error from displaying to a dialog, since I would like to retrieve the information and act upon it?*

A: Make an error callback function and assign it to ATTR_ERRMSG_HANDLER. This callback should trap all error codes and assign them to a global variable defined by the application. You can then check the global after each function call.

This implementation allows the programmer to completely change how the error is handled, once it is signaled, by overloading the error handler callbacks. The implementation also allows you to isolate the error handling in just a few functions so that if you ever need to change how a particular error is handled, you only need to change the one function. There is no need to search through your code looking for all the possible instances of the error code.

Additionally, with this implementation, you can completely modify the error message text assigned to any error code in the error message file, as well as retrieve the entire XVT API calling stack to where the error occurred, using xvt_errmsg_get_api_name. You can also get the source file and line number where the error was signaled. The application can create its own error codes and signals to get the same information in the application source files.

Q: *ATTR_PRINTER_WIDTH and ATTR_PRINTER_HEIGHT don't work on Motif. How can I get this information?*

A: ATTR_PRINTER_WIDTH and ATTR_PRINTER_HEIGHT represent the default dimensions of the printer, independent of page orientation. Therefore, these values do *not* change in response to a change in page orientation for a specific print record.

The application can determine whether the orientation has changed for a specific print record in response to a print setup dialog if you use either of the following methods:

1. Call xvt_app_escape(XVT_ESC_GET_PRINTER_INFO, ...) to determine the printable area's height, width, and resolution. The height and width returned reflect the page orientation. This method *does* work for XM.
2. Get the client rect of the print window and examine the rect's height and width, which also reflect the page orientation. (This method works only if the print window exists, of course.)

Note: The return value from xvt_dm_post_page_setup() indicates whether the user may have changed the print record as a result of the dialog.

Q: *When compiling with the PWRNoError option, all assertions from the XVT application framework are suppressed. What should I do about my code, which expects these assertions?*

A: Since the testing of the condition is suppressed by this option, make sure your program does not depend on any side effects that are a product of evaluating the condition.

For example, PwrAssert(itsImageId!=NULL_IMAGE, kImageErr, "Unable to construct image.") would not be evaluated, so a NULL_IMAGE could be returned.

This option is described in the **Error.h** file in the **pwr/include** directory.

Q: *When running Design and our application, we get the message, "Warning: Name: XVT_MB_1, Class: XmRowColumn, XtGrabKeyboard failed" and the system crashes if we click on the task window while a file is loading. Why is this?*

A: Unfortunately, there's nothing you can do about this situation with respect to Design. This error could occur in any application that uses the task menubar.

Following is a description of what is happening and why, and suggestions for application code you may be developing:

When a window that carries a menubar is created, the task window is no longer needed, so XVT hides it. However, the user may try to make a menu selection from the task window while XVT is attempting to hide it. The net affect is that while the new window is coming up, the task window is being removed. The menu the user selected from the task window menu bar eventually pops up briefly. After it pops up and then goes away, the application attempts to

restore focus to the menu button that was pressed on the task menu bar. The task menu bar, however, is now hidden so it is no longer valid to put focus on it. This is when the error occurs.

XVT has reviewed this situation in detail and has determined that there is no satisfactory solution to the problem. Following are some ideas for you to consider in your application:

1. Don't use the task menubar. Instead, create a W_DOC style window initially when your application starts up (similar to several of the XVT examples). As long as you always have a window with a menubar, the task window will not be displayed. Also, you can completely eliminate the use of the task window with the attribute ATTR_X_DISPLAY_TASK_WIN.

2. You could add some non-portable code that would make the task window insensitive just before creating the new window. You would then have to change it back when the task window was displayed again. For example, in response to choosing M_FILE_NEW from the task menu, you could make the following calls:

```
Widget widget;
    widget = (Widget)get_value(TASK_WIN,
                ATTR_X_WIDGET);
    XtSetSensitive(widget, FALSE);
    if (!create_res_window(WIN_101, TASK_WIN, EM_ALL,
                win_101_eh, 0L))
        xvt_error("Can't open window");
```

   Then in the E_DESTROY of the window, you could make these calls:

```
Widget widget;
widget = (Widget)get_value(TASK_WIN,
                ATTR_X_WIDGET);
XtSetSensitive(widget, TRUE);
```

   When we used this method, the crash did not occur.

3. Another idea is to install an event_handler, before creating the new window, that would discard all ButtonPress and KeyPress events. You would then need to set the event_handler back to NULL in the E_CREATE of the new window.

Q: *How does XVT get my machine's fonts?*

A: XVT uses the X fonts defined in **xsstrings.h** as "possible" fonts for its four types of XVT fonts, System, Fixed, Times and Helvetica.

When you run an XVT program, the XListFonts function gets called to load up a table that tells exactly which X fonts (out of the ones listed in **xsstrings.h**) to use for each of the four XVT fonts and to verify that at least one of the SYSTEM_NORMAL fonts exists. For each of the four XVT font families, XListFonts gets called with each of the font name strings from **xsstrings.h**. The first one to be found for each font is the one XVT uses for the font family. Those font strings contain "*" characters for things like point size. So XlistFonts returns to XVT all actual fonts that match the font string with the "*" wildcards. This means that it could return several fonts of the same family, but of different sizes.

If one of the system normal fonts is *not* found, an XVT error #36894 is generated. If XVT can't find any of the X fonts defined for one of the non-system fonts like Times, for example, it will automatically use the System normal font when a program requests a Times font.

XListFonts (and xlsfonts) looks in a file called **Families.list** or **fonts.dir** in the font directory (xset -q will tell you the font directory) for the fonts available on the machine. **Families.list** or **fonts.dir** has a mapping between a font name and an actual file that contains the font. (**Families.list** is generated by a Unix program called bldfamily. **fonts.dir** is created by mkfontdir.)

It is possible that if **Families.list**/**fonts.dir** is corrupt that XListFonts might find a font in **Families.list**/**fonts.dir** that doesn't really exist. In that case, the XVT program will work just fine, thinking it has the fonts it needs and won't generate an error until XLoadQueryFont is called to actually load the font. XVT will then generate the error, "XVT internal error #46016 - xvtxi_win_font: Current font is not defined." In this situation, you may have to run **bldfamily**/**mkfontdir**. After running **bldfamily**/**mkfontdir**, you must run xset fp rehash so that the server rereads the font databases.

Q: *How do I specify accelerators on the fly in my XVT code?*

A: The only platform for which on-the-fly accelerator specification is possible is Motif. For all other platforms, you cannot add or change accelerators after your program has started.

See Also: For information on how to specify accelerators on the fly, see the X Toolkit Intrinsics documentation.

Q: *Running an XVT program on Dec Alpha (OSF1) results in errors like:*

```
X Toolkit Warning:
    Name: mb1001_i32026
    Class: XmPushButtonGadget
    Illegal mnemonic character;  Could not convert X
    KEYSYM to a keycode
```

Why is this?

A: The problem is that on Dec Alpha, the key you press to erase or delete a character generates the keysym "Delete." On the Alpha, unlike most Unix keyboards, there is no key that generates "BackSpace." XVT attempts to define an accelerator that uses the keysym "BackSpace." Because there is no such keysym, you get the "Could not convert X KEYSYM to a keycode" error.

There are two possible solutions:

1. modify the **xrc**-generated UIL file, changing the BackSpace accelerator to Delete. For example, change

   ```
   XmNaccelerator = "Meta <Key>BackSpace";
   ```

   to

   ```
   XmNaccelerator = "Meta <Key>Delete";
   ```

2. Use xmodmap to change the Delete key to generate BackSpace with the following command:

   ```
   xmodmap -e "keysym Delete = Backspace"
   ```

Q: *How do I use the CGlobabUser class for Motif?*

A: To use the CGlobalUser class for Motif, you need to make a copy of **CGlobalUser.h** and put it in your local directory. Then you need to remove the one from **pwr/include** so that at compile time, yours will be used.

You can then make references in CGlobalUser for variables you desire to be global. You access these with the GU pointer, as follows:

```
void MyApp::StartUp(void)
{
    CApplication::StartUp();
    IApplication(new CGLobalUser);
    ...

GU->itsHelp = xvt_help_open_helpfile(&hf, 0);
```

Q: *How can I change the color of a menubar in XM?*

The following is the native code you need to change the color of a menubar. The E_CREATE of the window is a good place to put it.

```
#include <X11/Xlib.h>
#include <X11/Intrinsic.h>
#include <Xm/Xm.h>

Display *dsp;
XColor   rgb_def, hardware_def;
Colormap cmap;

Widget wid;
Widget parent;
Widget menu_wid;

int    status1;

/* Get the X widget handle for the XVT window */
wid = (Widget)get_value(win, ATTR_X_WIDGET);

/* Get the X widget handle for the parent of the XVT
            window */
parent = XtParent(wid);

/* Get the X widget handle for the menubar */
XtVaGetValues(parent, XmNmenuBar, &menu_wid, NULL);

/* Get the X display handle */
dsp = (Display *) get_value(win,ATTR_X_DISPLAY);

/* Get the X default colormap handle */
cmap = XDefaultColormap(dsp,0);

/* Get the XColor structures for a given color
            string */
status1 = XLookupColor(dsp, cmap, "yellow", &rgb_def,
            &hardware_def);

/* Get the actual XColor structure from the default
            color map */
status1 = XAllocColor(dsp, cmap, &hardware_def);

/* Set the background color for the menubar */
XtVaSetValues(menu_wid, XmNbackground,
            hardware_def.pixel, NULL);
```

Q: *How can I change the font on the menubar via an app-defaults resource file in Motif?*

A: The buttons across the menubar are XmCascadeButton widgets and the pulldown menu is an XmMenuShell widget. You can change the font (which will modify the size to match) of these two items by adding the following code to the app-defaults resource file:

```
*XmCascadeButton*fontList: rk24
```

```
*XmMenuShell*fontList: rk24
```

Both must be used.

Q: *How can I specify the title and icon used when a window is iconized in DSC?*

A: There is no portable way in XVT to change the icon and title associated with a window. You can do it non-portably, however, in Motif and OpenLook.

The following code fragment, to be placed in the E_CREATE of the window you want the icon to be associated with, demonstrates how to change the name of the icon associated with a given window and how to change the icon itself.

Note: This is extensible non-portable code. We intend for this to be a starting place for you in making this extension. If you have further questions, please refer to the X manuals for details on these functions.

```
/* includes */
#include <X11/Xlib.h>
#include <X11/Intrinsic.h>

/* top of file, global data */
Display *display;
XWMHints * wmhints_p;
        .
        .
        .
window_event_handler (win, ep)
WINDOW win;
EVENT *ep;
{
    char                    *data;
    static char             name[4];
    unsigned int            width, height;

    Widget                  widget;
    XTextProperty text_prop_p;

    switch (ep->type) {
    case E_CREATE:

        /* the following lines set the icon pixmap */
        display = (Display *) xvt_vobj_get_attr (
                            NULL_WIN, ATTR_X_DISPLAY);
        widget  = (Widget) xvt_vobj_get_attr (
                            win, ATTR_X_WIDGET);

        while(XtParent(widget) != NULL)
            widget = XtParent(widget);

        wmhints_p = XGetWMHints (
                        display, XtWindow (widget));
        if (wmhints_p == NULL) {
            xvt_dm_post_error ("XGetWMHints failed");
        }
```

```
/* function def below */
        data = get_data (&width, &height);

        wmhints_p->icon_pixmap =
                XCreateBitmapFromData(display,
                        XtWindow(widget), data, width, height);

        if (wmhints_p->icon_pixmap == 0)
                xvt_dm_post_error (
                        "XCreateBitmapFromData failed");

        wmhints_p->flags = IconPixmapHint;

        XSetWMHints(display, XtWindow(widget),
                wmhints_p);


        /* the following lines set the icon name */

        /* get the current WMIconName structure */
        if (XGetWMIconName(display, XtWindow(widget),
                &text_prop_p) == 0) {
        xvt_dm_post_error ("XGetWMIconName failed");
        }

        /* set the new name -- this obviously should
        *be something more meaningful than
        *"New"...maybe some combination of
        * get_title and stripping off all of the
        *characters before the - then strcpy the rest
        *of the string into text_prop_p.value.
        */

        sprintf(name, "New");

        text_prop_p.value = name;

        XSetWMIconName(display, XtWindow(widget),
                &text_prop_p);

        .
        .
        .
} /* event handler */



static char *
#if XVT_CC_PROTO
get_data(unsigned int *width, unsigned int *height)
#else
get_data(width, height)
unsigned int *width;
unsigned int *height;
#endif
{
/*This function will just return a few hard coded
*values, but would be the one that opened a file, read
*it in, put it into the proper format etc. The
*information here is produced by the x utility bitmap.
*We could also use XReadBitmapFile. */
```

```
#define bmapfile_width 64
#define bmapfile_height 64
static char bmapfile_bits[] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00,0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                .
                .
                .
        0xc0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0x03,
        0x38, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xfc,
        0x07, 0x00, 0x00, 0x00};

    *width = bmapfile_width;
    *height = bmapfile_height;
    return (bmapfile_bits);
}
```

Note: The contents of the file (bmapfile_bits) were obtained from an xbitmap program.

On the E_DESTROY:

```
printf ("E_DESTROY\n");
XFreePixmap(display, wmhints_p->icon_pixmap);
XFree (wmhints_p);
```

Q: *How can I specify the title and icon used when a window is iconized in DSC++?*

There is no portable way in XVT to change the icon and title associated with a window. You can do it non-portably, however, in Motif and OpenLook. The following commented code demonstrates how to specify the icon and icon name associated with a window. The code contains two files, **pwr_icon.cxx** and **pwr_icon.h**.

Note: This is extensible non-portable code. We intend this to be a starting place for you in making this extension. If you have further questions, please refer to the X manuals for details on these functions.

```
// pwr_icon.cxx

#if (XVTWS == MTFWS)

#include "pwr_icon.h"
#include <X11/Xlib.h>
#include <X11/Intrinsic.h>

// Used locally to incapsulate the actual bitmap.
static unsigned char* get_data (unsigned int* width,
                unsigned int* height) {

#include "pwr.h"

        *width = pwr_width;
        *height = pwr_height;

        return (unsigned char *) pwr_bits;
} // End get_data

void Set_Window_Icon(CWindow* Win) {

        WINDOW itsXVTWindow = Win->GetXVTWindow();

        Display *display;
        XWMHints * wmhints_p;

        unsigned char                   *data;
        static unsigned char name[128];
        unsigned int width, height;

        Widget              widget;
        XTextProperty text_prop_p;

// the following lines set the icon pixmap

display = (Display *) xvt_vobj_get_attr (
                NULL_WIN, ATTR_X_DISPLAY);

widget  = (Widget) xvt_vobj_get_attr(
                itsXVTWindow, ATTR_X_WIDGET);

while(XtParent(widget) != NULL)
        widget = XtParent(widget);

wmhints_p = XGetWMHints (display, XtWindow (widget));
if (wmhints_p == NULL) {
        xvt_dm_post_error ("XGetWMHints failed");
}

data = get_data (&width, &height);

wmhints_p->icon_pixmap =
        XCreateBitmapFromData(display,
                XtWindow(widget), (char *) data, width,
                height);

if (wmhints_p->icon_pixmap == 0)
        xvt_dm_post_error (
                        "XCreateBitmapFromData failed");

wmhints_p->flags = IconPixmapHint;
```

```
XSetWMHints(display, XtWindow(widget), wmhints_p);

// the following lines set the icon name

// get the current WMIconName structure
if (XGetWMIconName(display, XtWindow(widget),
                &text_prop_p) == 0) {
        xvt_dm_post_error ("XGetWMIconName failed");
}

char title[128];
xvt_vobj_get_title(itsXVTWindow,title,128);

// We want the window's title to be: "Debugger -
//Register Window" and we want the icon's title to be
//"Register Window".
int start = 0;
int end = strlen(title) - 1;

while((end > 0) && ((title[end] == ' ') || (
                        title[end] == '\t'))) end--;
while((start < end) && (title[start] != '-')) start++;
if((start < end) && (title[start] == '-')) start++;
while((start < end) && ((title[start] == ' ') || (
                        title[start] == '\t')))
        start++;

// Ideally, at this point, start is less than end.  If
//not, the title did not contain the "-" (dash) we
//were looking for or the title was null.  In the first
//case, use title as is.  In the second case, use PWR
//as the title.

if(strlen(title) == 0) {
        strcpy(title,"PWR");
        start = 0;
        end = strlen(title) - 1; }
else if(start >= end) {
        start = 0;
        end = strlen(title) - 1;
}

// Now put the title into name.
int i, j;
j = 0;
for(i=start; i<=end; i++) {
        name[j++] = title[i];
}

text_prop_p.value = name;
text_prop_p.nitems = strlen((char *) name);

XSetWMIconName(display, XtWindow(widget),
                &text_prop_p);


} // End Set_Window_Icon

#endif
```

```
pwr_icon.h

#if (XVTWS == MTFWS)

//Tested under Sun Sparc 20, running Solaris 2.4
//      and Motif

ifndef ICON_DEFINITION_H
#define ICON_DEFINITION_H

#include "xvt.h"
#include "PwrDef.h"
#include CWindow_i
```

The following routine allows you to associate an Icon with your CWindows in XVT-DSC++, running under Motif:

```
void Set_Window_Icon(CWindow* Win);
```

You should create the actual icon with the X Windows bitmap editor called “bitmap.” The bitmap must be 32 width by 30 tall and be called pwr. It will be included in the body of Set_Window_Icon. Calling it pwr will give a definition similar to the following:

```
#define pwr_width 32
#define pwr_height 30
static char pwr_bits[] = {
      0xf0, 0x00, 0xc0, 0x03, 0xf8, 0x00, 0xc0, 0x07,
      0x0c, 0xf3, 0x33, 0x0c, 0x0c, 0xf7, 0x3b, 0x0c,
      0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
      0x00, 0x0c, 0x0c, 0x00, 0x00, 0x0c, 0x0c, 0x00,
      0x00, 0x0c, 0x0c, 0x00, 0x00, 0xf0, 0x03, 0x00,
      0x00, 0xf0, 0x03, 0x00, 0xfc, 0xff, 0xff, 0x0f,
      0xfc, 0xff, 0xff, 0x0f, 0x0c, 0xff, 0x3f, 0x0c,
      0x0c, 0xff, 0x3f, 0x0c, 0xc0, 0xff, 0xff, 0x00,
      0xc0, 0xff, 0xff, 0x00, 0xfc, 0xff, 0xff, 0x0f,
      0xfc, 0xff, 0xff, 0x0f, 0xcc, 0xff, 0xff, 0x0c,
      0xcc, 0xff, 0xff, 0x0c, 0x00, 0xff, 0x3f, 0x00,
      0x00,0xff, 0x3f, 0x00, 0xfc, 0xff, 0xff, 0x0f,
      0x0c, 0xf0, 0x03, 0x0c, 0x0c, 0xf0, 0x03, 0x0c,
      0x00, 0xc0, 0x00, 0x00, 0x00, 0xc0, 0x00, 0x00,
      0x00, 0xc0, 0x00, 0x00, 0x00, 0xc0, 0x00, 0x00};
```

This definition should be in a file called **pwr.h**. If you call this procedure in all of your windows, they will all have the icon you defined, but the name will correspond to the name you gave the window in Architect. Note that the actual name that shows up at the top of a CWindow consists of two parts, the application name and the window name. These are separated by a dash. For example, “MyApplication - My Primary Window.”

The Icon name in this case would be, “My Primary Window,” and that would be shortened to “My Prim” or something similar. You should keep this in mind when giving the window a name in Architect.

The factory sets the window's name only after the window's constructor has been called. Because of this you need to call Set_Window_Icon after the factory creates it, which means that you must call it in the document after it is created by the document, and not the Window.

Because there are multiple places to create Windows the code must be included in multiple places. First, at the top of your document's body file, insert the following:

```
#if (XVTWS == MTFWS)
//If compiling for Motif
#include "pwr_icon.h"
#endif
```

In your document's BuildWindow routine after it creates all the automatic windows, insert the following:

```
#if (XVTWS == MTFWS)
        //If compiling for Motif
        const RWOrdered* WindowList = GetWindows();
        RWOrderedIterator win_next = RWOrderedIterator(
                        *WindowList);

        CWindow* win;
        while((win = (CWindow *) win_next()) != 0) {
                Set_Window_Icon(win);
        }
#endif
```

Anywhere else in your document's code where you create a Window with something like the following:

```
itsData.itsAnotherWindow  = (AnotherWindow *)
                vbxFactory.CreateWindow(this,AnotherWindow);
```

Follow with:

```
#if (XVTWS == MTFWS)
//If compiling for Motif
        Set_Window_Icon(itsData.itsAnotherWindow);
#endif

void Set_Window_Icon(CWindow* Win);

#endif
#endif
```

Also, at the close of the application, you need to free memory with the following:

```
XFreePixmap(display, wmhints_p->icon_pixmap);
XFree (wmhints_p);
```

Q: *Why won't a Motif application accept the -font command line argument?*

A: Motif uses the FontList resource to determine the font used for displaying text and ignores the "font" resource. So while the Intrinsics recognize the "-font" command-line option, and will place the corresponding value in the resource database, Motif applications will not be affected.

The "-xrm" command-line flag allows users to specify an arbitrary string to be placed in the resource database. This flag can be used to specify any resource on the command line, using the same syntax as a resource file.

Q: *How do I fill a CStringCollectionRWC to use it in an NListBox constructor?*

A: Following is an example of how to insert items into a CStringCollectionRWC for the constructors of CListBox, NListBox, NListButton, etc.:

```
RWOrdered mylist;
CStringRWC teststr("item1");
if (!mylist.insert(&teststr)) printf(
        "item1 failed insert!\n");
CStringRWC teststr2("item2");
if (!mylist.insert(&teststr2)) printf(
        "item2 failed insert!\n");
CStringRWC teststr3("item3");
if (!mylist.insert(&teststr3)) printf(
        "item3 failed insert!\n");

CStringCollectionRWC mycollection(mylist);

NListButton *newlist = new NListButton(
        this,CRect(10,10,100,100),mycollection);
```

Q: *Can we define which pixels on the screen are disturbed by the cursor?*

A: User defined cursors can now include a mask bitmap that defines which pixels on the screen are disturbed by the cursor. These marked pixels should be identical to the marked pixels in the cursor bitmap but one pixel wider at all boundaries.

The previous implementation of user defined cursors is still supported. It uses a default mask bitmap equal to the dimension of the cursor bitmap, typically 16 pixels in both dimensions.

For example, to use a user defined cursor, lasso, with a minimal mask, do the following:

1. Create the lasso cursor bitmap with the X bitmap client:

   ```
   bitmap lasso.h
   ```

2. Create the mask for lasso by starting with the **lasso.h** bitmap and include all pixels that touch the boundary of the marked pixels in **lasso.h**:

   ```
   cp lasso.h lassomask.h
   bitmap lassomask.h
   ```

3. In the file that defines the user-defined cursor resources, insert code that identifies the mask bitmap, lasso_mask, as follows.

Note: The ID symbol for the lasso bitmap mask *must* be equal to the ID symbol for the lasso cursor incremented by exactly 1000.

```
...
#include "lasso.h"
#include "lassomask.h"
...
static CURSOR_RESOURCE lasso_cursor, lasso_mask;
...


static RESOURCE_INFO
resource_table[] = {
        ...
        {"CURSOR", CURSOR_LASSO, (char *)&lasso_cursor},
        {"CURSOR", CURSOR_LASSO + 1000, (
                char *)&lasso_mask}, /* NOTE + 1000 */
        ...
        {       NULL,      0,              NULL}
};

RESOURCE_INFO *
resource_table_init()
{
        ...
        bld_cursor(&lasso_cursor, lasso_height,
                lasso_width, lasso_x_hot, lasso_y_hot,
                lasso_bits);
        bld_cursor(&lasso_mask, lassomask_height,
                lassomask_width, lassomask_x_hot,
                lassomask_y_hot, lassomask_bits);
        ...
        return(resource_table);
}
```

See Also: Refer to XVT/XM Resource Specifics on page 2-5 for more information on Motif resource specifics.

Note: Some platforms such as the IBM RS6000 require that the cursor bitmap be inverted. This is accomplished by creating the cursor and

cursor mask bitmap in the normal manner and selecting the Invert All option before the writing the bitmap to file.

Q: *How do I move project files between platforms and how does it affect the layout of the application?*

A: The following example involves creating a project file on a PC running an XTERM emulator, then moving the project file to an RS6000 machine, running Design from the console. Although in other situations, the numbers and platforms may vary, the concepts remain the same.

When you are running Design from the PC/Xterm and go into the Layout->Grid menu option, you might see something like "currently 7 x 13" next to the Chars RadioButton. Then, when Design is run on the RS6000, the same dialog shows "currently 8 x 13."

This is because when XVT-Design runs, it calculates the average size (in pixels) of a character, based on the FF_SYSTEM 12 point font. It then uses these calculations to display the windows and controls. This calculation is necessary because the font used in controls varies from platform to platform. If Design didn't calculate the character size this way, you would end up with controls that were too small on some platforms and too large on others, in relation to the font used in the controls.

Therefore, in the above scenario, when you run Design on the PC/Xterm, it determines a character is 7x13, but if you run it on the RS6000, it determines a character is 8x13.

Technical Note 122 discusses the scaling macros used by **xrc**. This same concept is used internally by XVT-Design. To generate a layout that will look appropriate on the machine you are moving to, Design determines how to scale the controls by dividing the dimensions on the destination machine by the dimensions on the source machine.

???Is Tech Note 122 still correct???

In the scenario just described, if you create a window 760 pixels wide on the PC and move it to the RS6000 (which uses 8x13), Design calculates 760 * 8 / 7 = 868 (width * destination dimension / source dimension). The resulting width of the same window on the RS6000 is 868.

To prevent resizing of the windows, one option is not to reload the project into Design on the RS6000, but rather just to move the source

code over and recompile. The generated URL code (from the PC/Xterm machine) should show URL_SRC_WIDTH as 7 and URL_SRC_HEIGHT as 13. In the generated URL code, the URL_DEST_WIDTH is set to equal the URL_SRC_WIDTH and the same is true for the HEIGHT. Therefore, if you just recompile the code, the dimensions of the window/controls will not change. However, you may end up with the wrong size windows and controls for your destination machine.

If you end up with the wrong size windows, then you can define the URL_DEST_WIDTH and URL_DEST_HEIGHT macros to 8x13 (or whatever you deem is appropriate) as described in Technical Note 122. Essentially you will be doing the same thing as Design does internally.

Design 3.0 (???Is this part true for the latest version too. Is it Design 5???) has 3 additional configuration file options that you can set to help with converting projects between platforms, objectScaling, charHeight, and charWidth.

objectScaling turns scaling on (TRUE is the default) or off (FALSE). If scaling is turned off, Design will make no attempt to scale the windows/controls/dialogs as discussed just discussed.

With charHeight and charWidth, you can tell Design what size character to use when scaling as discussed above, rather than allowing Design to pick the character size by which to scale.

Q: *Is it possible to hide the horizontal scrollbar of a ListBox, since by default the vertical and horizontal scrollbar appear?*

A: Scrollbars are placed automatically on list boxes. XVT does this because the other option in Motif is to have the scrollbars appear only when they are needed. That is, the existence of the horizontal scrollbar will change based on the width of items in the list box, and the existence of the vertical scrollbar will change based on the number of items in the list box. As the number of items exceeds the size of the list box, a vertical scrollbar will appear, and if the width of any of the items is larger than the width of the list box, a horizontal scrollbar will appear.

If you want the appearance of the list box to remain constant regardless of the size or number of items, you must have both vertical and horizontal scrollbars.

If you are sure the size of the items won't exceed the size of the list box, or if you don't mind the scrollbars appearing and disappearing,

then you can change an attribute of the ListBox widget to allow the list box to appear without scrollbars.

There are two methods of changing the attribute of the ListBox. You can either change the UIL file, or change the attribute of the widget programmatically. However, you can only use the UIL method for ListBoxes that were created in dialogs using create_res_dialog. If your list box is in a window, or if you are creating your dialogs using create_def_dialog or create_dialog, you will have to use the programmatic method.

To change the UIL file, let **xrc** create the UIL file for you, and then find the dialog that contains the ListBox. (Look for XVT_DLG_XXXX where XXXX is the resource ID of your dialog.) Then locate the XmScrolledList widget in this dialog and find the "arguments" section of the widget. In the arguments section, you can find a line that looks something like:

```
XmNscrollBarDisplayPolicy = XmSTATIC;
```

You should change this line to:

```
XmNscrollBarDisplayPolicy = XmAS_NEEDED;
```

Once you have changed the line, you will need to put the *entire* dialog code back into the URL file in a #transparent statement.

To change the attribute of the widget from programmatically, you will need to add some code to your window or dialog event handler, probably in the E_CREATE case, as follows. You will also need to add the two include files.

```
#include <X11/Intrinsic.h>
#include <Xm/Xm.h>

.
.
.
        WINDOW ctl_window;
        Widget listbox;

        switch (ep->type) {
        case E_CREATE:
        .
        .
        .
                ctl_window = xvt_win_get_ctl(win,
                                LISTBOX_ID);
                listbox = (Widget)xvt_vobj_get_attr(
                        ctl_window, ATTR_X_WIDGET);
        XtVaSetValues(listbox,
                        XmNscrollBarDisplayPolicy,XmAS_NEEDED,
                        NULL);
        break;
```

This code will change the behavior of your ListBox to show scrollbars only when they are necessary. Also, the size of the ListBox will remain constant regardless of the number of items it contains.

With the programmatic method, you can remove the horizontal scrollbar. The size of your list box, however, will shrink or grow according to the amount of room it needs to fit the text horizontally in the list box. You can make this change only at creation time, meaning that you can change your UIL code (or put this in a #transparent statement in your URL code) and create the dialog with create_res_dialog.

To remove the horizontal scrollbar if the list box is in a dialog, then you will need to modify the value of the XmNlistSizePolicy resource for the ListBox in the application's UIL file. For instance, the following is one of the ListBox definitions from **dlg.uil**:

```
object
        list_255_3 : XmScrolledList {
                arguments {
                        XmNx = 0;
                        XmNy = 0;
                        XmNwidth = 494;
                        XmNlistSizePolicy = XmCONSTANT;
                        XmNvisibleItemCount = 10;
                        XmNscrollBarDisplayPolicy = XmSTATIC;
                        XmNhighlightOnEnter = true;
                        XmNhighlightThickness = 1;
                };
                callbacks {
                        MrmNcreateCallback =
                        procedure xvt_create_travs_ctl_cb(4);
                };
        };
```

Notice that XmNlistSizePolicy is set to XmCONSTANT. This maintains a horizontal scrollbar and prevents the list from changing in size. If you change this to XmVARIABLE, the list grows or shrinks to match the size of the largest item, and there is no horizontal scrollbar. The third option is to use XmRESIZE_IF_POSSIBLE.

This attribute can be specified only at creation time, so it is best to initially define the dialog in **URL**, generate the UIL code, and then cut-and-paste the UIL definition of the whole dialog into a #transparent statement in your **URL** file. This makes your URL code non-portable but allows you to remove the scrollbar.

XVT/XM

INDEX

D

E

J

K

L

M

Z