VMS DECwindows Guide to Xlib (Release 4) Programming: MIT C Binding

Order Number: AA–PGZCA–TE

August 1991

This manual is a guide to programming Xlib routines.

Revision/Update Information:	This is a new manual.
Operating System:	VMS Version 5.4
Software Version:	VMS DECwindows Motif Version 1.0

Digital Equipment Corporation
Maynard, Massachusetts

August 1991

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

ZK5642

This document was prepared using DECdocument, Version 3.3-1b.

Contents

4 Defining Graphics Characteristics

5 Using Color

6 Drawing Graphics

7 Using Pixmaps and Images

8 Writing Text

9 Handling Events

Examples

Figures

Tables

Preface

This manual describes how to program Xlib routines using the MIT C binding. VMS DECwindows includes the MIT binding for Xlib programmers using the C programming language and other languages that support pointers.

The manual includes an overview of Xlib and tutorials that show how to use Xlib routines.

Note

This manual uses a generic format when referring to Xlib routine names in text. Routine names are represented in all uppercase letters with separating spaces. In addition, the X prefix has been omitted. For example, in text the routine name is written as OPEN DISPLAY; however, the MIT C binding format of the same routine is XOpenDisplay.

See the *X Window System* for a complete reference and description of all MIT C Binding Xlib routines.

Intended Audience

This manual is intended for experienced programmers who need to learn graphics programming using Xlib routines. Readers should be familiar with a high-level language. The manual requires minimal knowledge of graphics programming.

Document Structure

This manual is organized as follows:

- Chapter 1 provides an overview of Xlib, a sample Xlib program, and a guide to debugging Xlib programs.
- Chapters 2 through 12 provide tutorials that show how to use Xlib routines and include descriptions of predefined Xlib data structures and code examples that illustrate the concepts described.

This manual also includes the following appendixes:

- Appendix A is a guide to using the VMS DECwindows font compiler.
- Appendix B provides information about VMS DECwindows named colors.
- Appendix C lists VMS DECwindows fonts.

Associated Documents

The following documents contain additional information:

- *X Window System*—Provides detailed descriptions of each Xlib routine, as well as, the Inter-Client Communication Conventions Manual (ICCCM), the X Logical Font Description Conventions, and the X Window System Protocol.
- *DECwindows Motif for OpenVMS Guide to Non-C Bindings*—Describes non-C bindings for Xlib, Intrinsics, Motif Toolkit, and Digital extension routines.
- *DECwindows Extensions to Motif*—Provides reference information on the Digital extensions to Motif.
- *DECwindows Companion to the OSF/Motif Style Guide*—Covers style issues for Digital extensions to Motif and topics not addressed in the *OSF/Motif Style Guide*.
- *DECwindows Motif Guide to Application Programming*—Describes how to program with the Digital extensions to the Motif Toolkit. It supplements the *OSF/Motif Programmer's Guide*.
- *X and Motif Quick Reference Guide*—Provides quick reference information on Xlib, Intrinsics, and the Motif Toolkit.
- *OSF/Motif Style Guide*—Describes style guidelines for applications based on the Motif Toolkit.
- *OSF/Motif Programmer's Guide*—Describes how to program with the Motif Window Manager, Motif Toolkit, and the Motif User Interface Language (UIL).
- *OSF/Motif Programmer's Reference*—Provides reference information on the Motif Toolkit.

Conventions

The following conventions are used in this manual:

mouse	The term *mouse* is used to refer to any pointing device, such as a mouse, a puck, or a stylus.
MB1 (Select) MB2 (Drag) MB3 (Menu)	MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button. (The buttons can be redefined by the user.)
Ctrl+x	A sequence such as Ctrl+x (or Ctrl/x) indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
[ ]	In format descriptions, brackets indicate that whatever is enclosed within the brackets is optional; you can select none, one, or all of the choices. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)

boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason. Boldface text is also used to show user input in online versions of the book.
italic text	Italic text represents information that can vary in system messages (for example, Internal error *number*).
UPPERCASE TEXT	Uppercase letters indicate that you must enter a command (for example, enter OPEN/READ), or they indicate the name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege.
-	Hyphens in coding examples indicate that additional arguments to the request are provided on the line that follows.
numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1
Programming Overview of Xlib

The VMS DECwindows programming environment includes Xlib, a library of low-level routines that enable the VMS DECwindows programmer to perform windowing and graphics operations.

This chapter provides the following:

- An overview of the library
- A description of error handling conditions
- Xlib debugging techniques

Additionally, the chapter includes an introductory Xlib program. The program includes annotations that are explained more completely in the programming descriptions in later chapters of this guide.

1.1 Overview of Xlib

The VMS DECwindows programming environment enables application programs, called **clients**, to interact with workstations using the X Window System, Version 11 protocol software. The program that controls workstation devices such as screens and pointing devices is the **server**. Xlib is a library of routines that enables a client to communicate with the server to create and manage the following:

- Connections between clients and the server
- Windows
- Colors
- Graphics characteristics such as line width and line style
- Graphics
- Cursors
- Fonts and text
- Pixmaps and offscreen images
- Windowing and sending graphics between clients
- Client notification of windowing and graphics operations

Xlib processes some client requests, such as requests to measure the width of a character string, within the Xlib library. It sends other client requests, such as those pertaining to putting graphics on a screen or receiving device input, to the server.

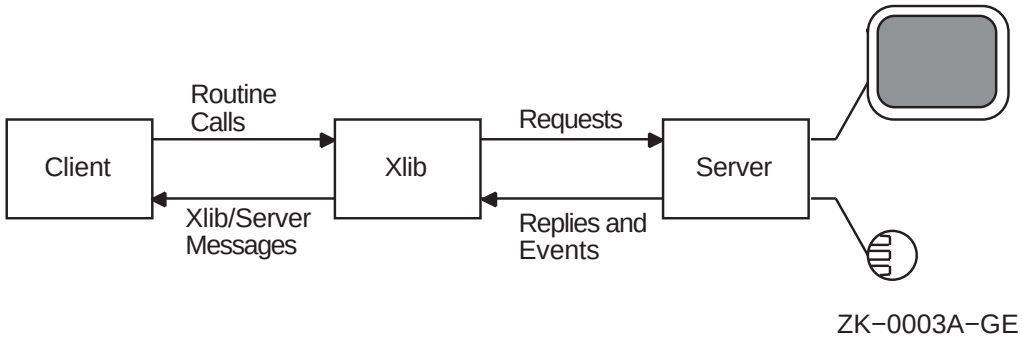
The server returns information to clients through either replies or events. Replies and events both return information to clients; the server returns replies synchronously and events asynchronously.

See the *X Window System* for a list of routines that cause Xlib to send requests to the server.

Figure 1–1 illustrates the relationships among client, Xlib, and server. The client calls Xlib routines, which always reside on the client system. If possible, Xlib processes calls internally and returns information to the client when appropriate. When an Xlib routine requires server intervention, Xlib generates a request and sends the request to the server.

The server may or may not reside on the same system as the client and Xlib. In either case, Xlib communicates with the server through a transport protocol, which can be either local shared memory or DECnet networking software.

Figure 1–1 Client, Xlib, and Server



1.2 Sample Xlib Program

The introductory Xlib program described in Example 1–1 illustrates the structure of a typical client program that uses Xlib windowing and graphic operations. The program creates two windows, draws text into one of them, and exits if the user clicks any mouse button while the cursor is in the window containing text.

The main loop of the program comprises two client-defined routines: *doInitialize* and *doHandleEvents*.

This section describes these routines and introduces fundamental concepts about Xlib resources, windowing, and event-handling.

1.2.1 Sample Initialization Routine

The sample program begins by calling a client-defined routine, *doInitialize*. The routine creates the resources the client needs to perform tasks. Xlib resources include windows, fonts, pixmaps, cursors, color maps, and data structures that define the characteristics of graphics objects. The sample program uses a default font, default cursor, default color map, client-defined windows, and a client-defined data structure that specifies the characteristics of the text displayed.

The *doInitialize* routine makes a connection between the client and the server. The client-server connection is called the **display**. After making the connection, or opening the display, the client can get display information from the server. For example, immediately after opening the display, the program calls the DEFAULT SCREEN OF DISPLAY routine to get the identifier of the default screen. The program uses the identifier as an argument in a variety of routines it calls later.

1.2.1.1 Creating Windows

A window is an area of the screen that either receives input or both receives input and displays graphics.

Windows in the X Window System are hierarchically related. At the base of the hierarchy is the **root window**. All windows that a client creates after opening a display are **inferiors** of the root window. The sample program includes two inferiors of the root window. First generation inferiors of a window are its **children**. The root window has one child, identified in the sample as *window1*. The window named *window2* is an inferior of the root window and a child of *window1*.

To complete the window genealogy, all windows created before a specified window and hierarchically related to it are its ancestors. In the sample program, *window1* has one ancestor (the root window); *window2* has two ancestors (the root window and *window1*).

1.2.1.2 Defining Colors

Defining background and foreground colors is part of the process of creating windows in the sample program. The *doDefineColor* routine allocates named VMS DECwindows colors for client use in a way that permits other clients to share the same color resource. For example, the routine specifies the VMS DECwindows color named "light grey" as the background color of *window2*. If other clients were using VMS DECwindows color resources, they too could access the VMS DECwindows data structure that defines "light grey." Sharing enables clients to use color resources efficiently.

The sample program calls the *doDefineColor* routine again in the next step of initialization, creating the graphics context that defines the characteristics of a graphics object. In this case, the program defines foreground and background colors used when writing text.

1.2.1.3 Working with the Window Manager

Most clients run on systems that have a window manager, which is an Xlib application that controls conflicts between clients. Clients provide the window manager with information about how it should treat client resources, although the manager can ignore the information. The sample program provides the window manager with information about the size and placement of *window1*. Additionally, the program assigns a name that the window manager displays in the title bar of *window1*.

1.2.1.4 Making Windows Visible on the Screen

Creating windows does not make them visible on the screen. To make its windows visible, a client must **map** them, painting the windows on a specified screen. The last step of initializing the sample program is to map *window1* and *window2*.

1.2.2 Sample Event-Handling Routine

The core of an Xlib program is a loop in which the client waits for the server to notify it of an **event**, which is a report of either a change in the state of a device or the execution of a routine call by another client. The server can report 30 types of events associated with the following occurrences:

- Key presses and releases
- Pointer motion
- Window entries and exits

- Changes of keyboards receiving input
- Changes in keyboard configuration
- Window and graphics exposures
- Changes in window hierarchy and configuration
- Requests by other clients to change windows
- Changes in available color resources
- Communication from other clients

When an event occurs, the server sends information about the event to Xlib. Xlib stores the information in a data structure. If the client has specified an interest in that kind of event, Xlib puts the data structure on an event queue. The *doHandleEvents* routine polls the event queue to determine if it contains an event of interest to the client. When the routine finds an event that is of interest to the client, the *doHandleEvents* routine calls one or more other routines.

Because Xlib clients do their essential work in response to events, they are considered event driven.

The sample program continually checks its event queue to determine if a window has been made visible or a button has been clicked. When the server informs it of either kind of event, the program performs its real work, as follows.

If the event is a window exposure, the program calls the *doExpose* routine. This routine determines whether the window exposed is *window2* and if the event is the first instance of the exposure. If both conditions are true, the program writes a message into the window.

If the event is a button press, the program calls the *doButtonPress* routine. This routine checks to make certain the cursor is in *window2* when the user clicks the mouse button. If the user clicks the mouse button when the cursor is in *window1*, the program reminds the user to click on *window2*. Otherwise, the program initiates a series of shutdown routines.

The shutdown routines unmap *window1* and *window2*, free resources allocated for the windows, break the connection between the sample program and its server, and exit the system. On the VMS operating system, clients only need to call SYS$EXIT. Exiting the system causes the other shutdown operations to occur. The call to SYS$EXIT breaks the connection between client and server, which frees resources allocated for client windows, and so forth.

See Example 1–1 for the sample Xlib program.

Note that using the #include directives in the sample program requires that the .h files were extracted during VAX C installation. In addition, note that the .h files reside in the SYS$COMMON:DECW$INCLUDE directory. For programs that use the X11 logical in include directives, users can redefine the logical X11 to DECW$INCLUDE with the following format:

```
DEFINE X11 DECW$INCLUDE
```

Example 1–1 Sample Xlib Program

(continued on next page)

Example 1–1 (Cont.) Sample Xlib Program

```
#include <decw$include/Xlib.h>
#include <decw$include/Xutil.h>

#define FontName\
    "-Adobe-New Century Schoolbook-Medium-R-Normal--*-140-*-*-P-*-ISO8859-1"
#define WindowName "Sample Xlib Program"

Display *dpy;
Window window1,window2;
GC gc;
Screen *screen;
int n, state = 0;
char *message[]= {
    "Click here to exit",
    "Click HERE to exit!"
    };

static void doInitialize( );
static int doDefineColor( );
static void doCreateWindows( );
static void doCreateGraphicsContext( );
static void doLoadFont( );
static void doExpose( );
static void doMapWindows( );
static void doHandleEvents( );
static void doButtonPress( );

/***** The main program *****/

static int main()
{
    doInitialize( );
    doHandleEvents( );
}

/***** doInitialize *****/
static void doInitialize( )
{

❶   dpy = XOpenDisplay(0);
    if (!dpy){
        printf("Display not opened!\n");
        exit(-1);
    }
    screen = XDefaultScreenOfDisplay(dpy);

❷   XSynchronize(dpy,1);

    doCreateWindows( );

    doCreateGraphicsContext( );

    doLoadFont( );

    doMapWindows( );
}
```

(continued on next page)

Example 1–1 (Cont.) Sample Xlib Program

```
/***** Create the windows *****/
❸static void doCreateWindows( )
{
    int window1W = 400;
    int window1H = 300;
    int window1X = (XWidthOfScreen(screen)-window1W)>>1;
    int window1Y = (XHeightOfScreen(screen)-window1H)>>1;
    int window2X = 50;
    int window2Y = 75;
    int window2W = 300;
    int window2H = 150;
    XSetWindowAttributes xswa;

    /* Create the window1 window */

    xswa.event_mask = ExposureMask | ButtonPressMask;
    xswa.background_pixel = doDefineColor(1);

    window1 = XCreateWindow(dpy, XRootWindowOfScreen(screen),
        window1X, window1Y, window1W, window1H, 0,
        XDefaultDepthOfScreen(screen), InputOutput,
        XDefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);

    /* Create the window2 window */

    xswa.event_mask = ExposureMask | ButtonPressMask;
    xswa.background_pixel = doDefineColor(2);

    window2 = XCreateWindow(dpy, window1, window2X, window2Y, window2W,
        window2H, 4, XDefaultDepthOfScreen(screen), InputOutput,
        XDefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);

    XStoreName(dpy, window1, WindowName);
}

/***** Create the graphics context *****/
❹static void doCreateGraphicsContext( )
{
    XGCValues xgcv;

    /* Create graphics context. */

    xgcv.foreground = doDefineColor(3);
    xgcv.background = doDefineColor(2);

    gc = XCreateGC(dpy, window2, GCForeground | GCBackground, &xgcv);
}

/***** Load the font for text writing *****/
❺static void doLoadFont( )
{
    Font font;

    font = XLoadFont(dpy, FontName);
    XSetFont(dpy, gc, font);
}
```

(continued on next page)

Example 1–1 (Cont.) Sample Xlib Program

```
/***** Create color *****/

❻static int doDefineColor(n)
{
    int pixel;
    XColor exact_color,screen_color;
    char *colors[] = {
        "dark slate blue",
        "light grey",
        "firebrick"
        };

    if ((XDefaultVisualOfScreen(screen))->class == TrueColor
        || (XDefaultVisualOfScreen(screen))->class == PseudoColor
        || (XDefaultVisualOfScreen(screen))->class == DirectColor
        || (XDefaultVisualOfScreen(screen))->class == StaticColor)

        if (XAllocNamedColor(dpy, XDefaultColormapOfScreen(screen),
           colors[n-1], &screen_color, &exact_color))
               return screen_color.pixel;
           else
               printf("Color not allocated!");
     else
        switch (n) {
            case 1:             return XBlackPixelOfScreen(screen); break;
            case 2:             return XWhitePixelOfScreen(screen); break;
            case 3:             return XBlackPixelOfScreen(screen); break;
        }
}

/***** Map the windows *****/
❼static void doMapWindows( )
{
    XMapWindow(dpy, window1);
    XMapWindow(dpy, window2);
}

/***** Handle the events *****/
❽static void doHandleEvents( )
{
    XEvent event;
    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case ButtonPress:           doButtonPress(&event); break;
        }
    }
}

/***** Write the message in the window *****/
static void doExpose(eventP)
XEvent *eventP;
{
    /* If this is an expose event on our child window, then write the text. */

    if (eventP->xexpose.window != window2) return;
    XClearWindow(dpy, window2);
    XDrawImageString(dpy, window2, gc, 75, 75, message[state],
        strlen(message[state]));
}
```

(continued on next page)

Example 1–1 (Cont.) Sample Xlib Program

```
/***** Shutdown *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
    if (eventP->xexpose.window != window2) {
        state = 1;
        XDrawImageString(dpy, window2, gc, 75, 75, message[state],
            strlen(message[state]));
        return;
    }

    /* Destroy the windows */

❾   XDestroyWindow(dpy, window1);
    XCloseDisplay(dpy);
    sys$exit (1);
}
```

❶ For information about connecting client and server, see Chapter 2.

❷ Xlib buffers client requests and sends them to the server asynchronously. This sequence causes clients to receive errors after they have occurred. When debugging a program, call the SYNCHRONIZE routine to enable synchronous error reporting. Using the SYNCHRONIZE routine has a serious negative effect on performance. Clients should call the routine only when debugging. For more information about debugging, see Section 1.4.

❸ For information about creating windows, see Chapter 3.

❹ Before drawing a graphics object on the screen, clients must define the characteristics of the object. The *doCreateGraphicsContext* routine defines the foreground and background values for writing text. For information about defining graphics characteristics, see Chapter 4.

❺ The sample program loads a VMS DECwindows font, New Century Schoolbook Roman 14, which the program uses to write the text in *window2*. For information about loading fonts, see Chapter 8.

❻ VMS DECwindows includes named colors for the convenience of clients. The sample program uses the named colors “dark slate blue,” “light grey,” and “firebrick.” It shares the named colors it uses with other clients. For information about sharing colors, whether named or client-defined, see Chapter 5. For information about defining colors for exclusive use, see Section 5.4. For a list of named colors, see the *X and Motif Quick Reference Guide*.

❼ Mapping windows makes them visible on the screen. For information about window mapping, see Chapter 3

❽ For more information about event handling, see Chapter 9.

❾ When a client exits a VMS DECwindows program on the VMS operating system, the series of calls to unmap, destroy windows, and close the display occurs automatically.

1.3 Handling Error Conditions

Xlib differs from most VMS programming libraries in the way it handles error conditions. In particular, Xlib does not perform any validation of input arguments when an Xlib routine is called.

If the input arguments are incorrect, the server usually generates an error event when it receives the Xlib request. Unless the client has specified an error handler, the server invokes the default Xlib error handler, which prints out a diagnostic message and exits. For more information about the Xlib error handler, refer to Section 9.13.2.

In some cases, Xlib signals a fatal access violation (SYS-F-ACCVIO) when passed incorrect arguments. This occurs when arguments are missing or are passed using the wrong addressing mode (passed by value instead of passed by reference).

1.4 Debugging Xlib Programs

As noted in Section 1.1, Xlib handles client requests asynchronously. Instead of dispatching requests as it receives them, Xlib buffers requests to increase communication efficiency.

Buffering contributes to delays in error reporting. Asynchronous reporting enables Xlib and the server to continue processing client requests despite the occurrence of errors. However, buffering contributes to the delay between the occurrence and client notification of an error.

As a result, programmers who want to step through routines to locate errors must override the buffering that causes asynchronous communication between client and server. To override buffering, use the SYNCHRONIZE routine. Example 1–1 includes a SYNCHRONIZE call as a debugging tool. Use the SYNC routine if you are interested in a specific call. The SYNC routine flushes the output buffer and then waits until all requests have been processed.

2

Managing the Client-Server Connection

A client requires one or more servers to process requests and return keyboard and mouse input. The server can be located either on the same system as the client or at a remote location where it is accessed across a network.

This chapter describes the following topics related to managing the client-server connection:

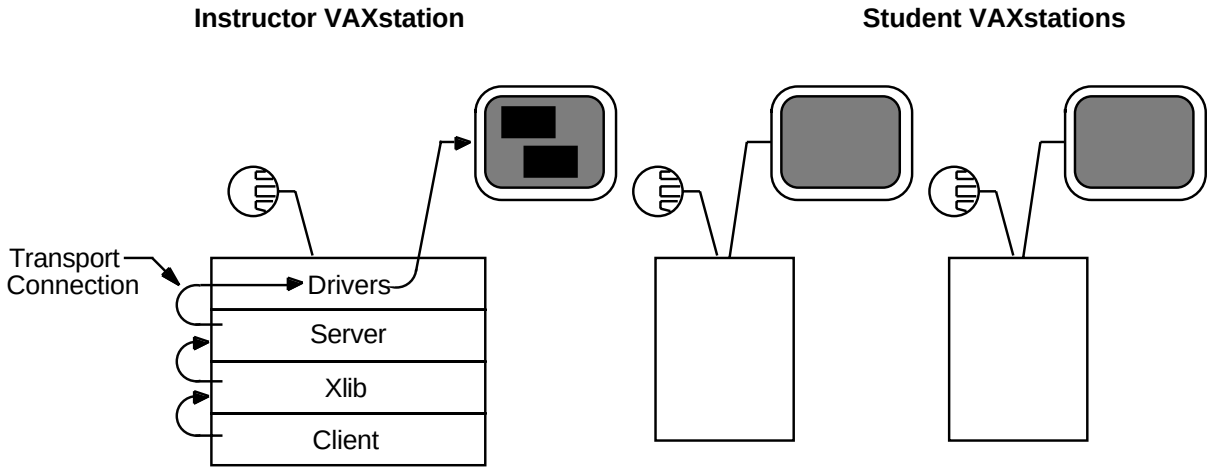
- Overview of the client-server connection
- Opening and closing a display
- Getting information about a display
- Managing sending requests to the server

2.1 Overview of the Client-Server Connection

A client using Xlib makes its first call to open a display. After opening a display, the client can get display information from and send requests to the server. To increase the efficiency of the client-server connection, Xlib buffers client requests.

To understand the relationship between a display and hardware, consider the classroom illustrated in Figure 2–1. The server and an instructor client program are running on the instructor VAXstation, which includes a screen, a keyboard, and a mouse. When the instructor opens a display, Xlib establishes a connection between the instructor client program and the server. The instructor can output graphics on the instructor VAXstation screen.

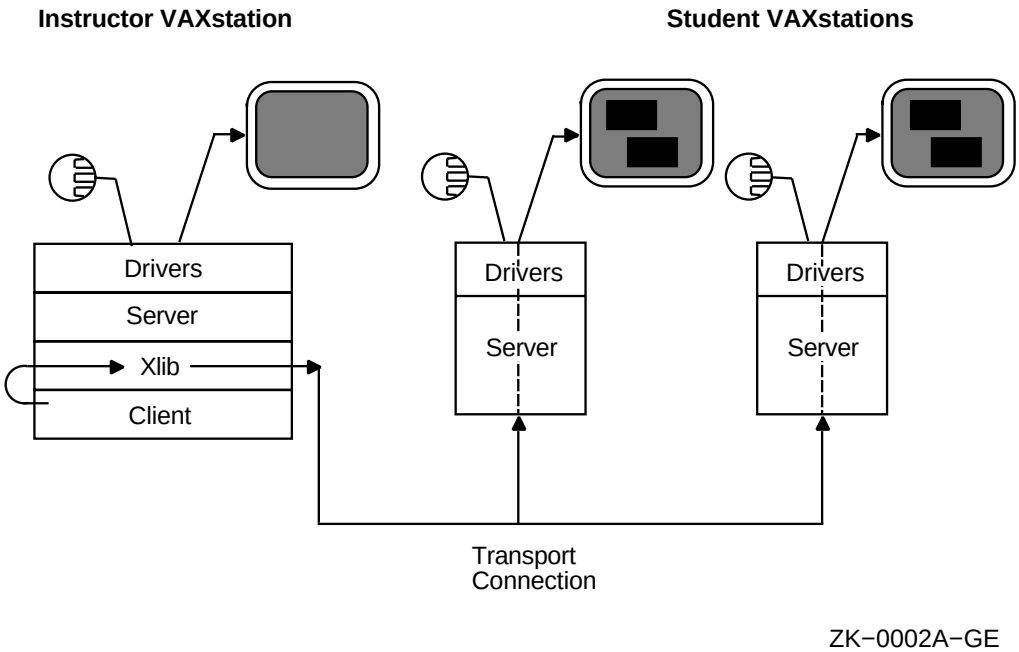
Figure 2–1 Graphics Output to Instructor VAXstation



If the instructor wants to output graphics to student screens, each student VAXstation must be running a server, and the client program must be connected to each server, as Figure 2–2 illustrates. Unlike the prior example, where the client program opened one display by making an internal connection with the server running on the VAXstation, here the client program establishes connections with multiple servers.

Xlib also enables multiple clients to establish connections with one server. For example, to output student work on the instructor screen, each student must open a display with the server running on the instructor VAXstation.

Figure 2–2 Graphics Output to Student VAXstations



2.2 Establishing the Client-Server Connection

The OPEN DISPLAY routine establishes a connection between the client and the server. The OPEN DISPLAY routine call has the following format:

```
display=XOpenDisplay(display_name)
```

In this call, **display_name** is a string that specifies the node on which the server is running. The **display_name** argument has the following format:

```
hostname::number.screen
```

The elements of the argument are as follows:

Elements	Description
hostname	The host on which the server is running. If the client and server are physically running in the same CPU, clients can specify a display number of zero.
number	The number of the display on the host machine.
screen	The screen on which client input and output is handled.

Passing a null argument to the OPEN DISPLAY routine causes Xlib to search for the definition of the logical DECW$DISPLAY. If successful, OPEN DISPLAY returns a unique identifier of the display. See Example 1–1 for an example of defining a display with this method.

A display can also be defined by using the DCL command SET DISPLAY, which sets the logical name DECW$DISPLAY. Refer to the *Using DECwindows Motif for OpenVMS* for more information about specifying a display.

2.3 Closing the Client-Server Connection

Although Xlib automatically destroys windows and resources related to a process when the process exits the server, clients should close their connection with a server explicitly. Clients can close the connection using the CLOSE DISPLAY routine. CLOSE DISPLAY destroys all windows associated with the display and all resources the client has allocated. The CLOSE DISPLAY routine call has the following format:

```
XCloseDisplay(display)
```

For an example of closing a display, see Example 1–1.

After closing a display, clients should not refer to windows, identifiers, and other resources associated with that display.

For more information about closing the X server connection, refer to the *X Window System*.

2.4 Getting Information About the Client-Server Connection

After opening a display, clients can get information about the client-server connection, client screens, and images created on client screens by using the routines listed in this section. These routines are useful for supplying arguments to other routines. See the *X Window System* for more information about these routines.

Note

This manual uses a generic format when referring to Xlib routine names in text. Routine names are represented in all uppercase letters with separating spaces. In addition, the X prefix has been omitted. For example, in text the routine name is written as OPEN DISPLAY; however, the MIT C binding format of the same routine is XOpenDisplay.

See the *X Window System* for a complete reference and description of all MIT C Binding Xlib routines.

Clients can get client-server information by using the following routines:

ALL PLANES
BLACK PIXEL
CONNECTION NUMBER
DEFAULT COLORMAP
DEFAULT DEPTH
DEFAULT GC
DEFAULT ROOT WINDOW
DEFAULT SCREEN
DEFAULT VISUAL
DISPLAY CELLS
DISPLAY KEYCODES
DISPLAY MOTION BUFFER SIZE
DISPLAY PLANES
DISPLAY STRING
IMAGE BYTE ORDER
MAX REQUEST SIZE
PROTOCOL REVISION
PROTOCOL VERSION
Q LENGTH
ROOT WINDOW
SCREEN COUNT
SERVER VENDOR
VENDOR RELEASE
WHITE PIXEL

Clients can get information about client screens using the following routines:

BLACK PIXEL OF SCREEN
CELLS OF SCREEN
DEFAULT COLORMAP OF SCREEN
DEFAULT DEPTH OF SCREEN
DEFAULT GC OF SCREEN
DEFAULT SCREEN OF DISPLAY
DEFAULT VISUAL OF DISPLAY
DOES BACKING STORE
DOES SAVE UNDERS
DISPLAY OF SCREEN
EVENT MASK OF SCREEN
HEIGHT OF SCREEN
HEIGHT MM OF SCREEN
MAX CMAPS OF SCREEN
MIN CMAPS OF SCREEN
PLANES OF SCREEN
ROOT WINDOW OF SCREEN
SCREEN OF DISPLAY
VISUAL ID FROM VISUAL
WHITE PIXEL OF SCREEN
WIDTH OF SCREEN
WIDTH MM OF SCREEN

Clients can get information about images created on screens using the following routines:

BITMAP BIT ORDER
BITMAP PAD
BITMAP UNIT
DISPLAY HEIGHT
DISPLAY HEIGHT MM
DISPLAY WIDTH
DISPLAY WIDTH MM

2.5 Managing Requests to the Server

Instead of sending each request to the server as the client specifies the request, Xlib buffers requests and sends them as a block to increase the efficiency of client-to-server communication. The routines listed in Table 2–1 control how requests output from the buffer.

Table 2–1 Output Buffer Routines

Routine	Description
FLUSH	Flushes the buffer.
SET AFTER FUNCTION	Specifies the function the client calls after processing each protocol request.
SYNC	Flushes the buffer and waits until the server has received and processed all events, including errors. Use SYNC to isolate one call when debugging.
SYNCHRONIZE	Causes the server to process requests in the buffer synchronously. SYNCHRONIZE causes Xlib to generate a return after each Xlib routine completes. Use it to debug an entire client or block.

Most clients do not need to call the FLUSH routine because the output buffer is automatically flushed by calls to event management routines. Refer to Chapter 9 for more information about event handling.

3
Working with Windows

Windows receive information from users; they display graphics, text, and messages. Xlib routines enable a client to create multiple windows and define window size, location, and visual appearance on one or more screens.

Conflicts between clients about displaying windows are handled by a window manager, which controls the size and placement of windows and, in some cases, window characteristics such as title bars and borders. The window manager also keeps clients informed about what it is doing with their windows. For example, the window manager might tell a client that one of its windows has been resized so that the client can reformat information displayed in the window.

This chapter describes the following topics related to windows and the window manager:

- Window fundamentals—A description of window type, hierarchy, position, and visibility
- Creating and destroying windows—How to create and destroy windows
- Mapping and unmapping windows—How to make windows visible on the screen
- Changing window characteristics—How to change the size, position, stacking order, and attributes of windows
- Getting information about windows—How to get information about window hierarchies, attributes, and geometry

3.1 Window Fundamentals

A window is an area of the screen that either receives input or receives input and displays graphics.

One type of window only receives input. Because an input-only window does not display text or graphics, it is not visible on the screen. Clients can use input-only windows to control cursors, manage input, and define regions in which the pointer is used exclusively by one client. A second type of window both receives input and displays text and graphics.

Clients can make input-output windows visible on the screen. To make a window visible, a client first creates the window and then maps it. Mapping a window allows it to become visible on the screen. When more than one window is mapped, the windows may overlap. Window hierarchy and position on the screen determine whether or not one window hides the contents of another window.

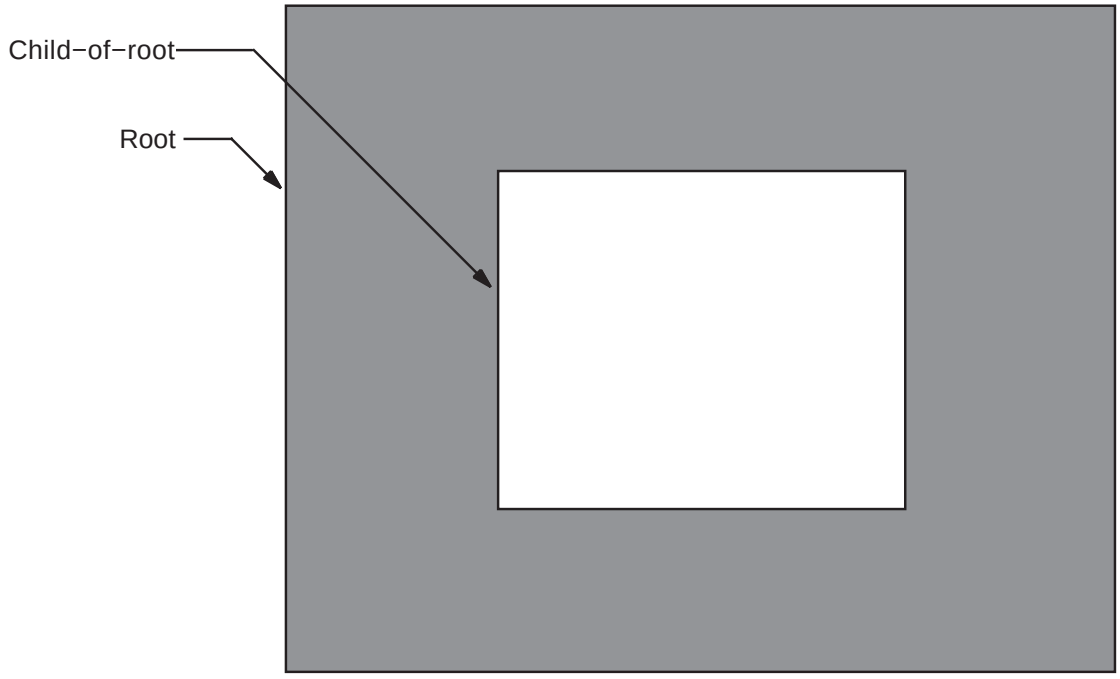
3.1.1 Window Hierarchy

Windows that clients create are part of a window hierarchy. The hierarchy determines how windows are seen. At the base of the hierarchy is the root window, which covers the entire screen when the client opens a display. All windows created after opening a display are subwindows of the root window.

When a client creates one or more subwindows of the root window, the root window becomes a **parent**. Children of the root window become parents when clients create subwindows of the children.

The hierarchy is structured like a stack of papers. At the bottom of the stack is the root window. Windows that clients create after opening a display are stacked on top of the root window, overlapping parts of it. For example, the window named **child-of-root** overlaps parts of the root window in Figure 3–1. The child-of-root window always touches the root window. Xlib always stacks children on top of the parents.

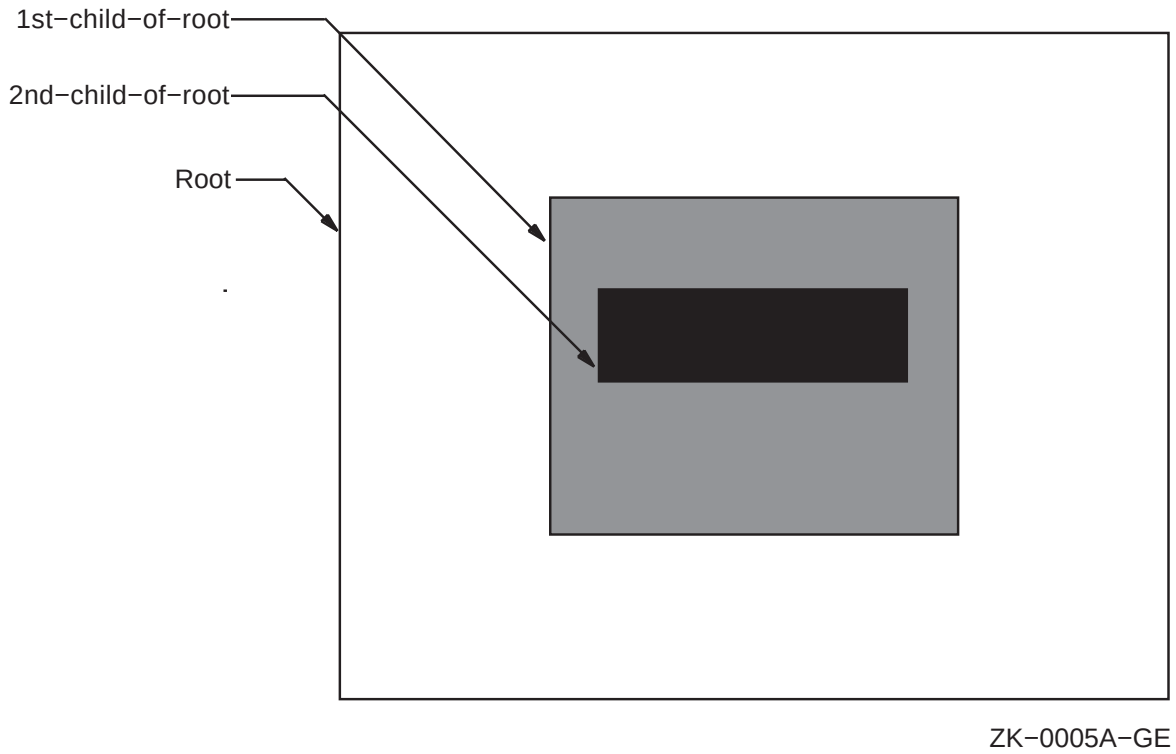
Figure 3–1 Root Window and One Child



ZK−0004A−GE

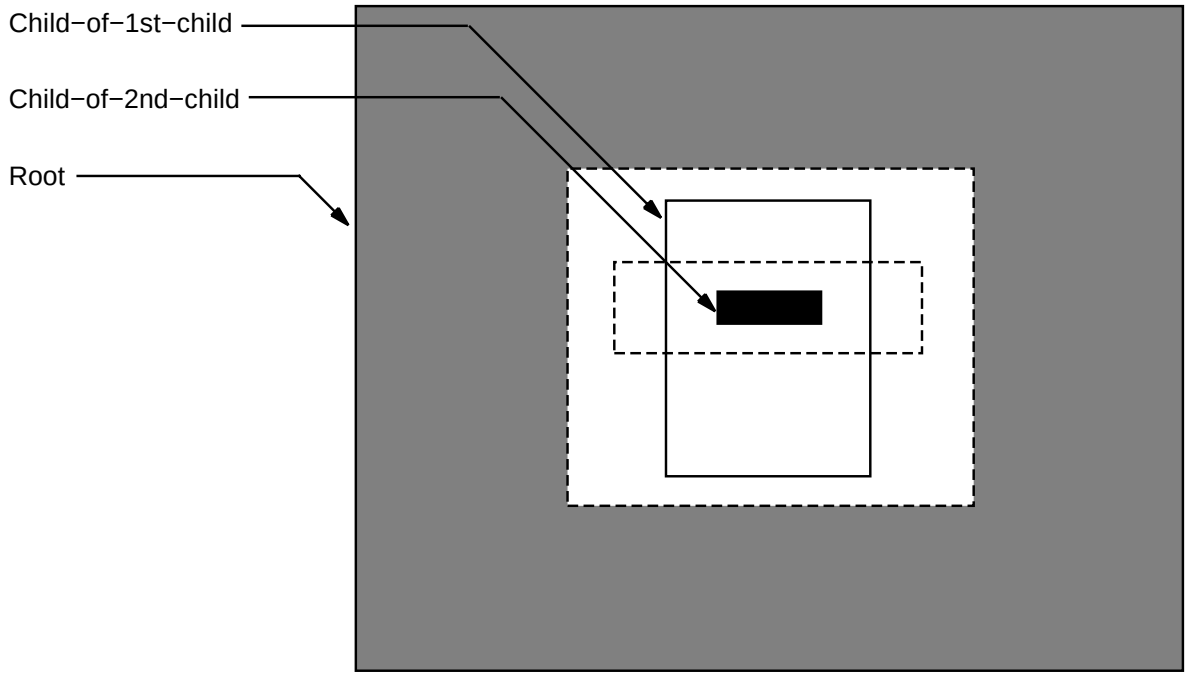
If a window has more than one child and if their borders intersect, Xlib stacks siblings in the order the client creates them, with the last sibling on top. For example, the second-level window named **2nd-child-of-root**, which was created last, overlaps the second-level window named **1st-child-of-root** in Figure 3–2.

Figure 3–2 Relationship Between Second-Level Windows



Third-level windows maintain the hierarchical relationships of their parents. The **child-of-1st-child-of-root** window overlaps **child-of-2nd-child-of-root** in Figure 3–3.

Figure 3–3 Relationship Between Third-Level Windows



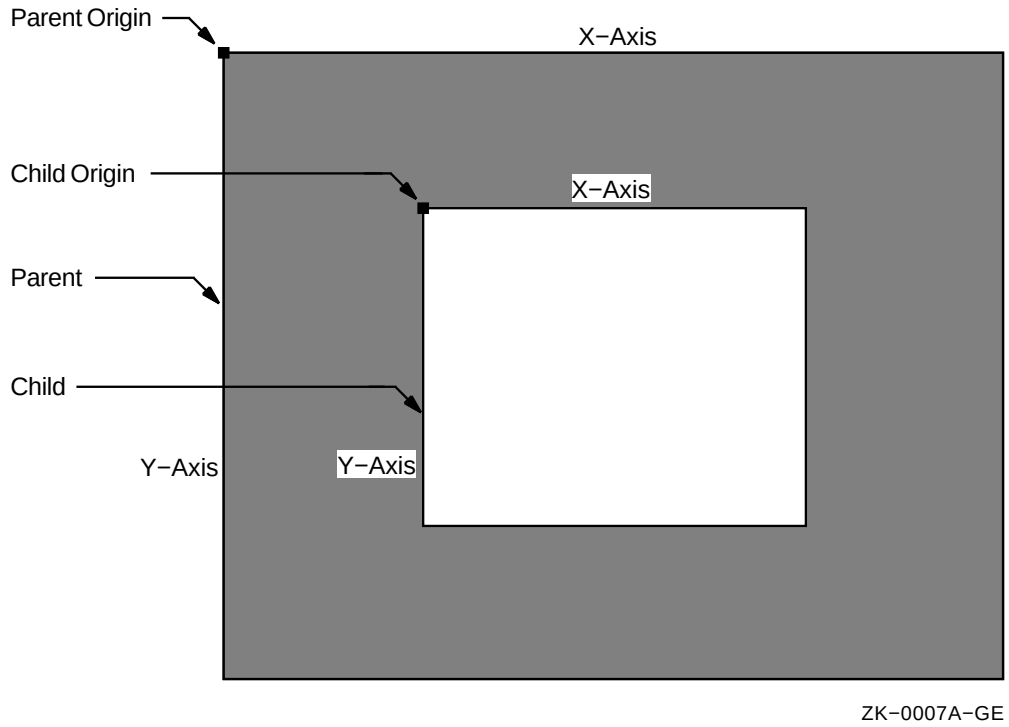
Windows created before a specified window and hierarchically related to it are ancestors of that window. For example, the root window and the window named **1st-child-of-root** are ancestors of **child-of-1st-child-of-root**.

3.1.2 Window Position

Xlib coordinates define window position on a screen and place graphics within windows. Coordinates that specify the position of a window are relative to the **origin**, the upper left corner of the parent window. Coordinates that specify the position of a graphic object within a window are relative to the origin of the window in which the graphic object is displayed.

Xlib measures length along the x-axis from the origin to the right; it measures length along the y-axis from the origin down. Xlib specifies coordinates in units of **pixels**, the smallest unit the server can display on a screen. Figure 3–4 illustrates the Xlib coordinate system.

Figure 3–4 Coordinate System



For more information about positioning windows, see Section 3.2. For more information about positioning graphics, see Chapter 6.

3.1.3 Window Visibility and Occlusion

A window is **visible** if one can see it on the screen. To be visible, a window must be an input-output window, it must be mapped, its ancestors must be mapped, and it must not be totally hidden by another window. When a window and its ancestors are mapped, the window is considered **viewable**. A viewable window that is totally hidden by another window is not visible.

Even though input-only windows are never visible, they can overlap other windows. An input-only window that overlaps another window is considered to **occlude** that window. Specifically, window A occludes window B if both are mapped, if A is higher in the stacking order than B, and if the rectangle defined by the outside edges of A intersects the rectangle defined by the outside edges of B.

A viewable input-output window that overlaps another window is considered to **obscure** that window. Specifically, window A obscures window B if A is a viewable input-output window, if A is higher in the stacking order than B, and if the rectangle defined by the outside edges of A intersects the rectangle defined by the outside edges of B.

3.2 Creating Windows

After opening a display, clients can create windows. As noted in the description of window fundamentals (Section 3.1), creating a window does not make it visible on a screen. To be visible, the window must meet the conditions described in Section 3.1.3.

Clients can either create windows that inherit most characteristics not relating to size or shape from their parents or define all characteristics when creating windows.

3.2.1 Using Attributes of the Parent Window

An **attribute** is a characteristic of a window not relating to size or shape, such as the window background color. The CREATE SIMPLE WINDOW routine creates an input-output subwindow that inherits the following attributes from its parent:

- Method of moving the contents of a window when the parent is moved or resized
- Instructions for saving window contents when the window obscures or is obscured by another window
- Instructions to the server regarding information that ancestors should know when a window change occurs
- Instructions to the window manager concerning map requests
- Color
- Cursor

For more information about these attributes, see Section 3.2.2.

If the parent is a root window, the new window created with the CREATE SIMPLE WINDOW routine has the following attributes:

- The server discards window contents if the window is reconfigured.
- The server discards the contents of obscured portions of the window.
- The server discards the contents of any window that the new window obscures.
- No events are specified as being of interest to the window ancestors.
- No restrictions are placed on the window manager.
- The color is identical to the parent color.
- No cursor is specified.

In addition to creating a window with attributes inherited from the parent window, the CREATE SIMPLE WINDOW routine enables clients to define the border and background attributes of the window and to determine window position and size.

Example 3–1 illustrates creating a simple window. To make the window visible, the example includes mapping and event handling functions, which are described in Section 3.4 and Chapter 9.

Example 3–1 Creating a Simple Window

```
Window win1;
                    .
                    .
                    .
static void doCreateWindows( )
{
❶   int win1W = 600;
    int win1H = 600;
❷   int win1X = (XWidthOfScreen(screen)-window1W)>>1;
    int win1Y = (XHeightOfScreen(screen)-window1H)>>1;

    /* Create the window */

❸   win1 = XCreateSimpleWindow(dpy, XRootWindowOfScreen(screen),
        win1X, win1Y, win1W, win1H, 10, XBlackPixelOfScreen(screen),
        XWhitePixelOfScreen(screen);
}
```

❶ The client assigns window width and height the value of 600 (pixels) each.

❷ The client specifies the position of the window using two display information routines, WIDTH OF SCREEN and HEIGHT OF SCREEN. The **x** and **y** coordinates define the top left outside corner of the window borders relative to the inside of the parent border. In this case, the parent is the root window, which does not have a border.

❸ The CREATE SIMPLE WINDOW routine call has the following format:

window_id = XCreateSimpleWindow(display, parent_id, x_coord,
y_coord, width, height, border_width, border_id,
background_id)

The client specifies a black border ten pixels wide, a white background, and a size of 600 by 600 pixels.

The window manager overrides border width and color.

CREATE SIMPLE WINDOW returns a unique identifier, *win1*, used in subsequent calls related to the window.

3.2.2 Defining Window Attributes

To create a window whose attributes are different from the parent window, use the CREATE WINDOW routine. The CREATE WINDOW routine enables clients to specify the following window attributes when creating an input-output window:

- Default contents of an input-output window
- Border of an input-output window
- Treatment of the window when it or its relative is obscured
- Treatment of the window when it or its relative is moved
- Information the window receives about operations associated with other windows
- Color
- Cursor

Clients creating input-only windows can define the following attributes:

- Treatment of the window when it or its relative is moved

- Information the window receives about operations associated with other windows
- Cursor

Specifying other attributes for an input-only window causes the server to generate an error. Input-only windows cannot have input-output windows as children.

Use the following method to define window attributes:

1. Assign values to the relevant members of a set window attributes data structure.
2. Indicate the defined attribute by specifying the appropriate flag in the **value_mask** argument of the CREATE WINDOW routine. If more than one attribute is to be defined, indicate the attributes by doing a bitwise OR on the appropriate flags.

The following illustrates the set window attributes data structure:

```
typedef struct {
    Pixmap background_pixmap;
    unsigned long background_pixel;
    Pixmap border_pixmap;
    unsigned long border_pixel;
    int bit_gravity;
    int win_gravity;
    int backing_store;
    unsigned long backing_planes;
    unsigned long backing_pixel;
    Bool save_under;
    long event_mask;
    long do_not_propagate_mask;
    Bool override_redirect;
    Colormap colormap;
    Cursor cursor;
} XSetWindowAttributes;
```

Table 3–1 describes the members of the data structure.

Table 3–1 Set Window Attributes Data Structure Members

Member Name	Contents
background_pixmap	Defines the window The background_pixmap member can assume one of three possible values: pixmap identifier, the constant None (default), or the constant ParentRelative.
background_pixel	Causes the server to override the specified value for the background_ pixmap member. This is equivalent to specifying a pixmap of any size filled with the background pixel and used to paint the window background.
border_pixmap	Defines the window border.
border_pixel	Causes the server to override the border_pixmap member.
bit_gravity	Defines how the contents of the window should be moved when the window is resized. By default, the server does not retain window contents. For more information about bit gravity, see Section 3.7.

(continued on next page)

Table 3–1 (Cont.) Set Window Attributes Data Structure Members

Member Name	Contents
win_gravity	Defines how the server should reposition the newly created window when its parent window is resized. By default, the server does not move the newly created window. For more information about window gravity, see Section 3.7.
backing_store	Provides a hint to the server about how the client wants it to manage obscured portions of the window.
backing_planes	Indicates (with bits set to one) which bit planes of the window hold dynamic data that must be preserved if the window obscures or is obscured by another window.
backing_pixel	Defines what values to use in planes not specified by the backing_planes member. The server is free to save only specified bit planes and to regenerate the remaining planes with the specified pixel value. Bits that extend beyond the number per pixel of the window are ignored.
save_under	Informs the server that the client would like the contents of the screen saved when the window obscures them.
event_mask	Defines which types of events associated with the window the server should report to the client. For more information about defining event types, see Chapter 9.
do_not_propagate_mask	Defines which kinds of events should not be propagated to ancestors. For more information about managing events, see Chapter 9.
override_redirect	Specifies whether calls to map and configure the window should override a request by another client to redirect those calls. For more information about redirecting calls, see Chapter 9.
color map	Specifies the color map, if any, that best reflects the colors of the window. The color map must have the same visual type as the window. If it does not, the server issues an error. For more information about the color map and visual types, see Chapter 5.
cursor	Causes the server to use a particular cursor when the pointer is in the window.

Table 3–2 lists default values for the set window attributes data structure.

Table 3–2 Default Values of the Set Window Attributes Data Structure

Member Name	Default Value
background_pixmap	None
background_pixel	Undefined
border_pixmap	Copied from the parent window
border_pixel	Undefined
bit_gravity	Window contents not retained
win_gravity	Window not moved
backing_store	Window contents not retained
backing_planes	All 1s
backing_pixel	0
save_under	False

(continued on next page)

Table 3–2 (Cont.) Default Values of the Set Window Attributes Data Structure

Member Name	Default Value
event_mask	Empty set
do_not_propagate_mask	Empty set
override_redirect	False
colormap	Copied from parent
cursor	None

Xlib assigns a flag for each member of the set window attributes data structure to facilitate referring to the members, as listed in Table 3–3.

Table 3–3 Set Window Attributes Data Structure Flags

Flag Name	Set Window Attributes Member
CWBackPixmap	background_pixmap
CWBackPixel	background_pixel
CWBorderPixmap	border_pixmap
CWBorderPixel	border_pixel
CWBitGravity	bit_gravity
CWWinGravity	win_gravity
CWBackingStore	backing_store
CWBackingPlanes	backing_planes
CWBackingPixel	backing_pixel
CWSaveUnder	save_under
CWEventMask	event_mask
CWDontPropagate	do_not_propagate
CWOverrideRedirect	override_redirect
CWColormap	colormap
CWCursor	cursor

Example 3–2 illustrates how clients can define window attributes while creating input-output windows with the CREATE WINDOW routine. The program creates a parent window and two children windows. The hierarchy of the subwindows is determined by the order in which the program creates them. In this case, *subwin1* is superior to *subwin2*, which is created last.

Example 3–2 Defining Attributes When Creating Windows

(continued on next page)

Example 3–2 (Cont.) Defining Attributes When Creating Windows

```
Window window, subwindow1,subwindow2;
int n;
                    .
                    .
                    .
static void doCreateWindows( )
{
    int windowW = 600;
    int windowH = 600;
    int windowX = (WidthOfScreen(screen)-windowW)>>1;
    int windowY = (HeightOfScreen(screen)-windowH)>>1;
    int subwindow1X = 150;
    int subwindow1Y = 100;
    int subwindow1W = 300;
    int subwindow1H = 400;
    int subwindow2X = 275;
    int subwindow2Y = 125;
    int subwindow2W =  50;
    int subwindow2H = 150;
❶   XSetWindowAttributes xswa;
                    .
                    .
                    .
    /* Create the window window */
❷   xswa.event_mask = ExposureMask | ButtonPressMask;
    xswa.background_pixel = doDefineColor(1);
❸   window = XCreateWindow(dpy, XRootWindowOfScreen(screen),
        windowX, windowY, windowW, windowH, 0,
        XDefaultDepthOfScreen(screen), InputOutput,
        XDefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);
    /* Create the window subwindow1 */
    xswa.background_pixel = doDefineColor(3);
    subwindow1 = XCreateWindow(dpy, window, subwindow1X, subwindow1Y, subwindow1W,
        subwindow1H, 4, XDefaultDepthOfScreen(screen), InputOutput,
        XDefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);
    /* Create the window subwindow2 */
    xswa.background_pixel = doDefineColor(3);
    subwindow2 = XCreateWindow(dpy, window, subwindow2X, subwindow2Y, subwindow2W,
        subwindow2H, 4, XDefaultDepthOfScreen(screen), InputOutput,
        XDefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);
}
                    .
                    .
                    .
static int doDefineColor(n)
{
                    .
                    .
                    .
```

❶ Allocate storage for a set window attributes data structure used to define window attributes.

❷ Set the attributes of the parent window. The client indicates an interest in window exposure and button press events. For more information about events, see Chapter 9.

The client defines the window background by calling the client-defined *doDefineColor* routine. For more information about defining colors, see Chapter 5.

❸ The CREATE WINDOW routine call has the following format:

```
window_id_return=XCreateWindow(display, parent_id, x_coord,
    y_coord, width, height, border_width, depth, class, visual_struc,
    attributes_mask, attributes)
```

The depth of a window is its number of bits per pixel. The call passes a display information routine to indicate that the client wants the parent window depth to be identical to the display depth.

The window class can be either input only or input-output, specified by the following constants:

- InputOnly
- InputOutput

If the window is the same class as the parent, pass the constant **CopyFromParent**.

The visual type indicates how the window displays color values. For more information about visual types, see Chapter 5.

3.3 Destroying Windows

When a client no longer needs a window, the client should destroy it using either the DESTROY WINDOW or the DESTROY SUBWINDOWS routine. DESTROY WINDOW destroys a specified window and all its subwindows. DESTROY SUBWINDOWS destroys all subwindows of a specified window in bottom-to-top stacking order.

Destroying a window frees all storage allocated for that window. If the window is mapped to the screen, the server notifies all applications that the window has been destroyed.

3.4 Mapping and Unmapping Windows

After creating a window, the client can map it to a screen using the MAP WINDOW or MAP SUBWINDOWS routine. Mapping generally makes a window visible at the location the client specified when creating it. Part or all of the window is not visible when the following conditions occur:

- One or more windows higher in the stacking order obscure it
- One or more window ancestors are not mapped
- The new window extends beyond the boundary of its parent

MAP WINDOW maps a window. If the window is an inferior, and one or more of its ancestors have not been mapped, the server considers the window to be mapped after the call, even though the window is not visible on the screen. The window becomes visible when its ancestors are mapped.

To map all subwindows of a specified window in top-to-bottom order, use MAP SUBWINDOWS. Using the MAP SUBWINDOWS routine to map several windows may be more efficient than calling the MAP WINDOW routine to map each window. The MAP SUBWINDOWS routine enables the server to map all of the windows at one time instead of mapping a single window with the MAP WINDOW routine.

To ensure that the window is completely visible, use the MAP RAISED routine. MAP RAISED reorders the stack with the window on top and then maps the window. Example 3–3 illustrates how a window is mapped and raised to the top of the stack.

Example 3–3 Mapping and Raising Windows

```
Window window,subwindow1,subwindow2;

    /* Create windows in the following order: window, subwindow2, subwindow1 */
                    .
                    .
                    .
static void doMapWindows( )
{
    XMapWindow(dpy, window);
❶   XMapWindow(dpy, subwindow2);
❷   XMapRaised(dpy, subwindow1);
}
```

❶ In this example, the client created *subwindow1* after *subwindow2*, putting *subwindow1* at the top of the stack.

Consequently, whether *subwindow2* were to be mapped before or after *subwindow1*, *subwindow1* would obscure *subwindow2*.

The effect is illustrated in Figure 3–5.

❷ Mapping and raising *subwindow2* moves it to the top of the stack. It is now visible, as Figure 3–6 illustrates.

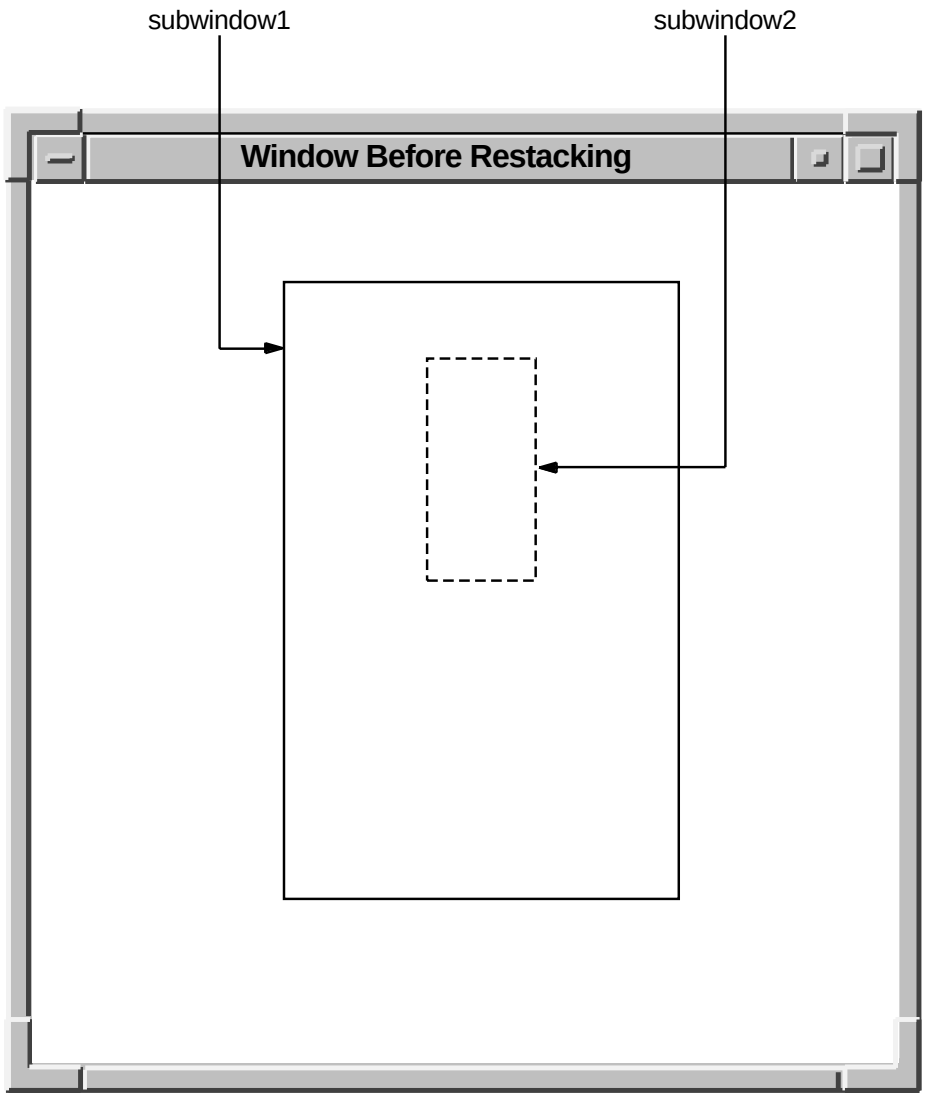
When the client no longer needs a window mapped to the screen, call UNMAP WINDOW. If the window is a parent, its children are no longer visible after the call, although they are still mapped. The children become visible when the parent is mapped again.

To unmap all subwindows of a specified window, use UNMAP SUBWINDOWS. UNMAP SUBWINDOWS results in an UNMAP WINDOW call on all subwindows of the parent, from bottom-to-top stacking order.

3.5 Associating Properties with Windows

Xlib enables clients to associate data with a window. This data is considered a **property** of the window. For example, a client could store text as a window property. Although a property must be data of only one type, it can be stored in 8-bit, 16-bit, and 32-bit formats.

Figure 3–5 Window Before Restacking

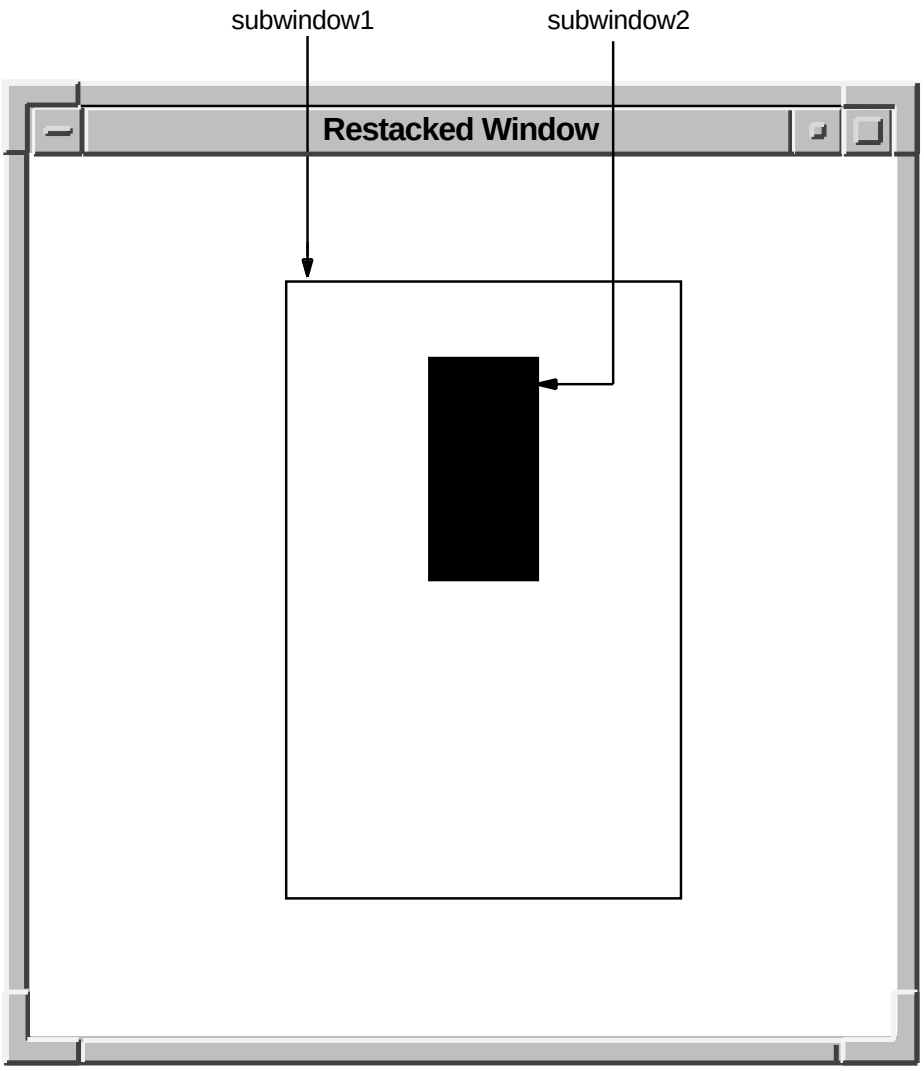


ZK–2505A–GE

Xlib uses **atoms** to uniquely identify properties. An atom is a string paired with an identifier. For example, a client could use the atom XA_WM_ICON_NAME to name a window icon stored for later use. The atom XA_WM_ICON_NAME pairs the string XA_WM_ICON_NAME with a value, 37, that uniquely identifies a property.

In DECW$INCLUDE:XATOMS.H, VMS DECwindows includes predefined atoms such as XA_WM_ICON_NAME for commonly used properties. Table 3–4 lists by function all predefined atoms except those used to identify font properties and atoms used to communicate with the window manager. See Chapter 12 for a list of atoms related to window management. See Chapter 8 for a list of atoms related to fonts.

Figure 3–6 Restacked Window



ZK–2504A–GE

Table 3–4 Predefined Atoms

For Global Selection	
XA_PRIMARY	XA_SECONDARY
For Cut Buffers	
XA_CUT_BUFFER0	XA_CUT_BUFFER1
XA_CUT_BUFFER2	XA_CUT_BUFFER3
XA_CUT_BUFFER4	XA_CUT_BUFFER5
XA_CUT_BUFFER6	XA_CUT_BUFFER7
For Color Maps	
XA_RGB_COLOR_MAP	XA_RGB_BEST_MAP
XA_RGB_BLUE_MAP	XA_RGB_RED_MAP
XA_RGB_GREEN_MAP	XA_RGB_GRAY_MAP
XA_RGB_DEFAULT_MAP	
For Resources	
XA_RESOURCE_MANAGER	XA_ARC
XA_ATOM	XA_BITMAP
XA_CARDINAL	XA_COLORMAP
XA_CURSOR	XA_DRAWABLE
XA_FONT	XA_INTEGER
XA_PIXMAP	XA_POINT
XA_RECTANGLE	XA_STRING
XA_VISUALID	XA_WINDOW

Note

The Inter-Client Communications Convention (ICCC) discourages the use of cut buffer atoms. Use the primary and secondary atoms as the selection mechanism.

In addition to providing predefined atoms, Xlib enables clients to create atom names of their own. To create an atom name, use the INTERN ATOM routine, as in the following example:

```
            .
            .
            .
Atom atom_id;
char *name = "MY_ATOM";
Bool if_exists;
```

```
atom_id =  XInternAtom(dpy, name, if_exists);
           .
           .
           .
```

The routine returns an identifier associated with the string MY_ATOM. If the atom does not exist in the atom table, Xlib returns the value none. Note that any atom identifier and its associated name remain defined until the server is reset. Example 3–4 uses the INTERN ATOM routine to exchange properties between two windows.

To get the name of an atom, use the GET ATOM NAME routine, as in the following example:

```
           .
           .
           .
char *name;
Atom atom_id = 39;

name = XGetAtomName(dpy, atom_id);
           .
           .
           .
```

The routine returns a string associated with the atom identifier.

Xlib enables clients to change, obtain, update, and interchange properties. Example 3–4 illustrates exchanging properties between two subwindows. The example uses the CHANGE PROPERTY routine to set a property on the parent window and the GET PROPERTY routine to get the data from the parent window. In addition, the example uses the INTERN ATOM routine.

Example 3–4 Exchanging Window Properties

```
#define windowWidth      600
#define windowHeight     600
#define subwindowWidth   300
#define subwindowHeight  150
#define true 1

display *dpy;
window win, subwin1, sunwin2;
gc gc;
screen *screen;
int n;
atom atom_id;
char *name = "my_atom";
bool if_exists;
                    .
                    .
                    .
```

(continued on next page)

Example 3–4 (Cont.) Exchanging Window Properties

```
static void doCreateWindows( )
{
    int winW = windowWidth;
    int winH = windowHeight;
    int winX = 100;
    int winY = 100;
    int subwindow1X = 150;
    int subwindow1Y = 100;
    int subwindow2X = 150;
    int subwindow2Y = 350;
    XSetWindowAttributes xswa;
```

(continued on next page)

Example 3–4 (Cont.) Exchanging Window Properties

```
    /* Create the win window */

    xswa.event_mask = ExposureMask | ButtonPressMask | PropertyChangeMask;
    xswa.background_pixel = doDefineColor(1);

    win = XCreateWindow(dpy, RootWindowOfScreen(screen),
        winX, winY, winW, winH, 0,
        DefaultDepthOfScreen(screen), InputOutput,
        DefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);

    /* Create the subwindows */
    xswa.event_mask = ExposureMask| ButtonPressMask;
    xswa.background_pixel = doDefineColor(2);

    subwin1 = XCreateWindow(dpy, win, subwindow1X, subwindow1Y, subwindowWidth,
        subwindowHeight, 0, DefaultDepthOfScreen(screen), InputOutput,
        DefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);
    subwin2 = XCreateWindow(dpy, win, subwindow2X, subwindow2Y, subwindowWidth,
        subwindowHeight, 0, DefaultDepthOfScreen(screen), InputOutput,
        DefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);
                    .
                    .
                    .
/***** Handle events ******/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case ButtonPress:           doButtonPress(&event);break;
            case PropertyNotify:        doPropertyNotify(&event);break;
        }
    }
}
                    .
                    .
                    .
/***** Handle button presses *******/
static void doButtonPress(eventP)
XEvent *eventP;
{
    char *property_data = "You clicked MB1";

    if (eventP->xbutton.button == Button2) sys$exit(1);
    if (eventP->xbutton.window == subwin1 && eventP->xbutton.button == Button1)
❶       XChangeProperty(dpy, win, atom_id, XA_STRING, 16,
            PropModeReplace, property_data, 15);
    return;
}
/***** Get the property and draw text into the subwindow *******/
static void doPropertyNotify(eventP)
XEvent *eventP;
{
    long offset = 0;
    long length = 1000;
    Atom type_returned;
    int format_returned;
    unsigned long num_items_returned, bytes_remaining;
    unsigned char *property_returned;
```

(continued on next page)

Example 3–4 (Cont.) Exchanging Window Properties

```
    if (eventP->xproperty.atom == atom_id){
❷       XGetWindowProperty(dpy, win, atom_id, offset, length,
            true, XA_STRING, &type_returned, &format_returned,
            &num_items_returned, &bytes_remaining, &property_returned);

❸       XDrawString(dpy, subwin2, gc, 75, 75, property_returned,
            num_items_returned);
    }
    return;
}
```

❶ When the user clicks MB1 in subwindow *subwin1*, the client calls the CHANGE PROPERTY routine. CHANGE PROPERTY causes the server to change the property identified by the atom *atom_id* to the value specified by *property_data*. The property is associated with the parent window, *win*.

When changing properties, clients can specify how the server should treat the property. If the client specifies the constant **PropModeReplace**, the server discards the previous property. If the client specifies the constant **PropModePrepend**, the server inserts the new data at the beginning of the existing property data. If the client specifies the constant **PropModeAppend**, the server inserts the new data at the end of the existing property data.

Changing the property causes the server to send a property notify event to *win*. For information about event handling, see Chapter 9.

❷ After checking to ensure that the changed property is the one to obtain, the client calls the GET WINDOW PROPERTY routine.

❸ After getting the string data from the parent window, the client uses it to write text in *subwin2*. For information about writing text, see Chapter 8.

In addition to the GET WINDOW PROPERTY routine, Xlib includes the property-management routines described in Table 3–5.

Table 3–5 Routines for Managing Properties

Routine	Description
LIST PROPERTIES	Returns a list of properties defined for a specified window.
ROTATE WINDOW PROPERTIES	Rotates the properties of a specified window and generates a property notify event. For more information about property notify events, see Chapter 9.
DELETE PROPERTY	Deletes a specified property.

3.6 Exchanging Properties Between Clients

Xlib provides routines that enable clients to exchange properties. The properties, which are global to the server, are called **selections**. Text cut from one window and pasted into another window exemplifies the global exchange of properties. The text cut in window A is a property owned by client A. Ownership of the property transfers to client B, who then pastes the text into window B.

Properties are exchanged between clients by a series of calls to routines that manage the selected text. When a user drags the pointer cursor, client A responds by calling the SET SELECTION OWNER routine. SET SELECTION OWNER identifies client A as the owner of the selected text. The routine also identifies the window of the selection, associates an atom with the text, and puts a time-stamp on the selection. The atom, XA_PRIMARY, names the selection. The time-stamp enables any clients competing for the selection to determine selection ownership.

Clients can determine the owner of a selection by calling the GET SELECTION OWNER routine. This routine returns the identifier of the window that currently owns the specified selection.

By calling the CONVERT SELECTION routine, clients ask the owner of a selection to convert it to a particular data type. If conversion is possible, the client converting the selection notifies the client requesting the conversion that the selection is available. The property is then exchanged.

For example, when a user decides to paste the selected text in window B, client B, who owns window B, sends client A a selection request. The request identifies the window requesting the cut text and the format in which the client would like the property transferred.

In response to the request, client A first checks to ensure that the time of the request corresponds to the time in which client A owns the selection. If the time coincides and if client A can convert the selection to the data type requested by client B, client A notifies client B that the text is stored and available. Client B then retrieves the data by calling the GET WINDOW PROPERTY routine.

Clients request and notify other clients of selections by using events. For information about using events to request, convert, and notify clients of selections, see Chapter 9. For style guidelines about using selections, see the *OSF/Motif Style Guide*.

3.7 Changing Window Characteristics

Xlib provides routines that enable clients to change window position, size, border width, stacking order, and attributes.

This section describes how to use Xlib routines to do the following:

- Change multiple window characteristics in one call
- Change position, size, or border width
- Change stacking order
- Change window attributes

3.7.1 Reconfiguring Windows

Xlib enables clients either to change window characteristics using one call or to use individual routines to reposition, resize, or change border width.

The CONFIGURE WINDOW routine enables clients to change window position, size, border width, and place in the hierarchy. To change these window characteristics in one call, use the CONFIGURE WINDOW routine, as follows:

1. Set values of relevant members of a window changes data structure.
2. Indicate what is to be reconfigured by specifying the appropriate flag in the CONFIGURE WINDOW **value_mask** argument.

The window changes data structure enables clients to specify one or more values for reconfiguring a window. The following illustrates the window changes data structure:

```
typedef struct {
    int x, y;
    int width, height;
    int border_width;
    Window sibling;
    int stack_mode;
} XWindowChanges;
```

Table 3–6 describes the members of the data structure.

Table 3–6 Window Changes Data Structure Members

Member Name	Contents
x	Defines, with the y member, the new location of the window relative to the origin of its parent.
y	Defines, with the x member, the new location of the window relative to the origin of its parent.
width	Defines the new width of the window, excluding the border.
height	Defines the new height of the window, excluding the border.
border_width	Specifies the new window border in pixels.
sibling	Specifies the sibling window for stacking order.
stack_mode	Defines how the window is restacked. Table 3–7 lists constants and definitions for restacking windows.

The client can change the hierarchical position of a window in relation to all windows in the stack or to a specified sibling. If the client changes the size, position, and stacking order of the window by calling CONFIGURE WINDOW, the server restacks the window based on its final, not initial, size and position. Table 3–7 lists constants and definitions for restacking windows.

Table 3–7 Stacking Values

Constants	Relative to All Windows	Relative to Siblings
Above	Top of stack.	Just above the sibling.
Below	Bottom of stack.	Just below the sibling.
TopIf	If any sibling obscures a window, the server places the obscured window on top of the stack.	If the specified sibling obscures a window, the server places the obscured window at the top of the stack.
BottomIf	If a window obscures any sibling, the server places the obscuring window at the bottom of the stack.	If a window obscures the specified sibling, the server places the obscuring window at the bottom of the stack.
Opposite	If any sibling obscures a window, the server places the obscured window on top of the stack. If a window obscures any window, the server places the obscuring window at the bottom of the stack.	If the specified sibling obscures a window, the server places the obscuring window on top of the stack. If a window obscures the specified sibling, the server places the obscuring window on the bottom of the stack.

Xlib assigns a symbol to the flag associated with each member of the data structure (Table 3–8).

Table 3–8 Window Changes Data Structure Flags

Flag Name	Window Changes Member
CWX	x
CWY	y
CWWidth	width
CWHeight	height
CWBorderWidth	border_width
CWSibling	sibling
CWStackMode	stack_mode

Example 3–5 illustrates using CONFIGURE WINDOW to change the position, size, and stacking order of a window when the user presses a button.

Example 3–5 Reconfiguring a Window Using the CONFIGURE WINDOW Routine

```
    /* This program changes the position, size, and stacking
       order of subwindow1 */

static void doButtonPress(eventP)
XEvent *eventP
{
    XWindowChanges xwc;

❶   xwc.x = 200;
    xwc.y = 350;
    xwc.width = 200;
    xwc.height = 50;
    xwc.sibling = subwindow2;
    xwc.stack_mode = Above;

❷   XConfigureWindow(dpy, subwindow1, CWX | CWY | CWWidth | CWHeight | CWSibling
            | CWStackMode, &xwc);
}
```

❶ Assign values to relevant members of the window changes data structure. Because the client identifies a sibling (*subwindow1*), it must also choose a mode for stacking operations.

❷ The call to reconfigure *subwindow1*. The CONFIGURE WINDOW routine call has the following format:

XConfigureWindow(display, window_id, change_mask, values)

Create a mask by performing a bitwise OR operation on relevant flags that indicate which members of WINDOW CHANGES the client has defined.

Figure 3–7 illustrates how the windows look after being reconfigured.

Figure 3–7 Reconfigured Window

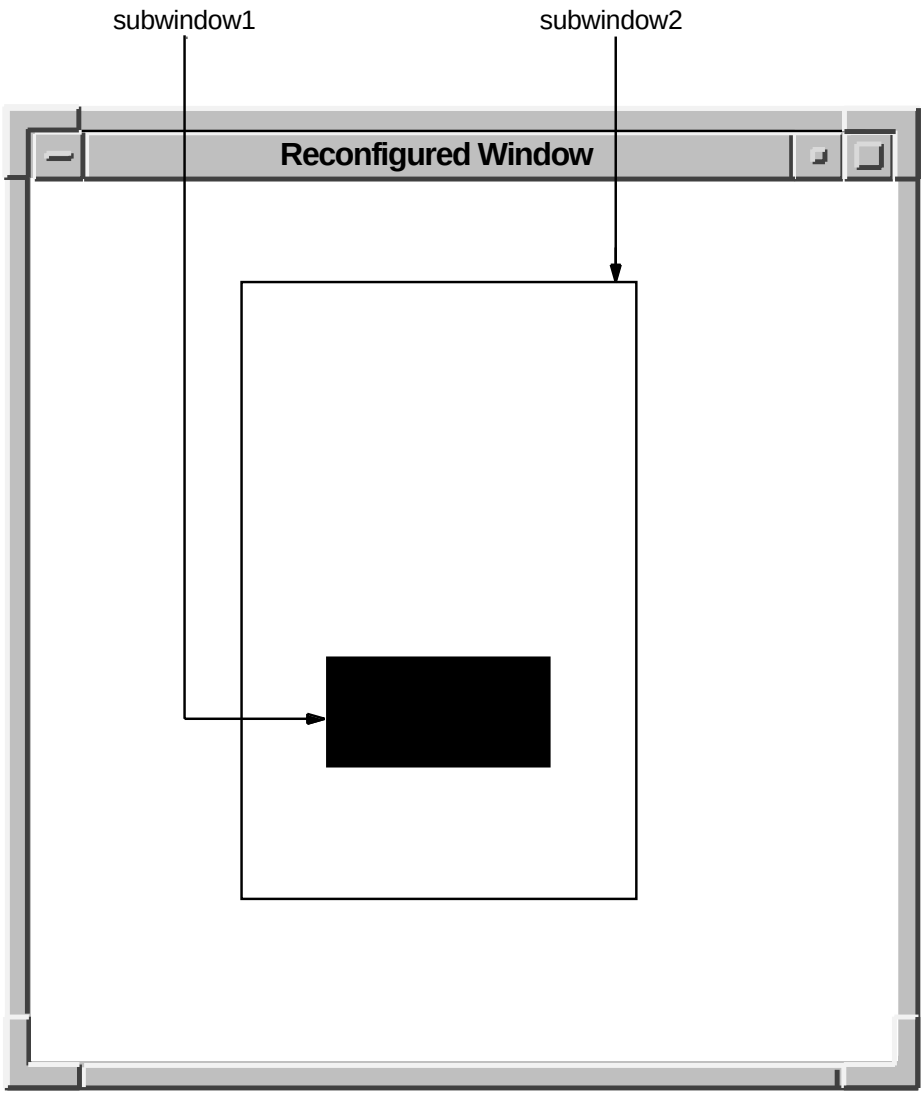


Table 3–9 lists routines to change individual window characteristics.

Table 3–9 Window Configuration Routines

Routine	Description
MOVE WINDOW	Moves a window without changing its size.
RESIZE WINDOW	Changes the size of a window without moving it. The upper left window coordinate does not change after resizing.
MOVE RESIZE WINDOW	Moves and changes the size of a window.
SET WINDOW BORDER WIDTH	Changes the border width of a window.

3.7.2 Effects of Reconfiguring Windows

It is important to know how reconfiguring windows affects graphics and text drawn in them by the client. (See Chapter 6 for a description of working with graphics and Chapter 8 for a description of writing text.) When a client resizes a window, window contents are either moved or lost, depending on the **bit gravity** of the window. Bit gravity indicates that a designated region of the window should be relocated when the window is resized. Resizing also causes the server to resize children of the changed window.

To control how the server moves children when a parent is resized, set the **window gravity** attribute. Table 3–10 lists choices for retaining window contents and controlling how the server relocates children.

Table 3–10 Gravity Definitions

Constant Name	Movement of Window Contents and Subwindows
ForgetGravity	The server always discards window contents and tiles the window with its selected background. If the client has not specified a background, existing screen contents remain the same.
NorthWestGravity	Not moved.
NorthGravity	Moved to the right half of the window width.
NorthEastGravity	Moved to the right, the distance of the window width.
WestGravity	Moved down half the window height.
CenterGravity	Moved to the right half of the window width and down half of the window height.
EastGravity	Moved to the right, the distance of the window width and down half the window height.
SouthWestGravity	Moved down the distance of the window height.
SouthGravity	Moved to the right half of the window width and down the distance of the window height.
SouthEastGravity	Moved to the right, the distance of the window width and down the distance of the window height.
StaticGravity	Contents or origin not moved relative to the origin of the root window. Static gravity only takes effect with a change in window width and height.
UnmapGravity	Window should not be moved; the child should be unmapped when the parent is resized.

The client can change the hierarchical position of a window in relation to either all windows in the stack or to a specified sibling. If the client changes the size, position, and stacking order of the window by calling CONFIGURE WINDOW, the server restacks the window based on its final, not initial, size and position. Table 3–7 lists constants and definitions for restacking windows.

Figure 3–8 illustrates how the server moves the contents of a reconfigured window when the bit gravity is set to the constant **EastGravity**.

Figure 3–9 illustrates how the server moves a child window if its parent is resized and its window gravity is set to the constant **NorthwestGravity**.

Figure 3–8 East Bit Gravity

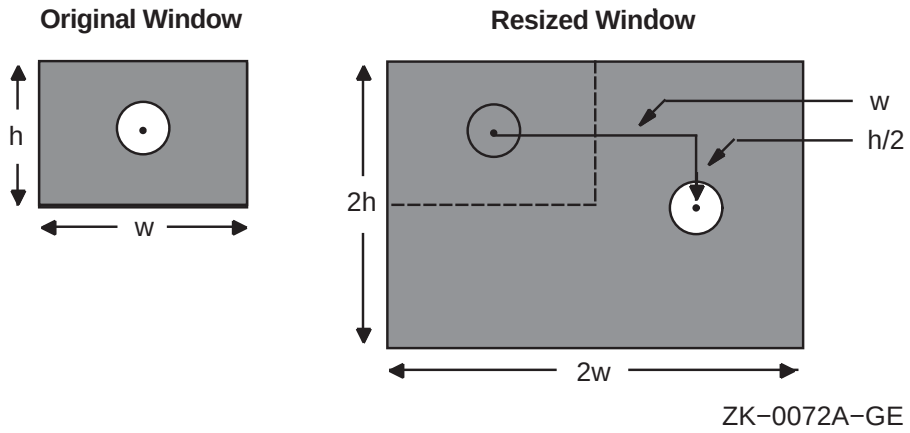
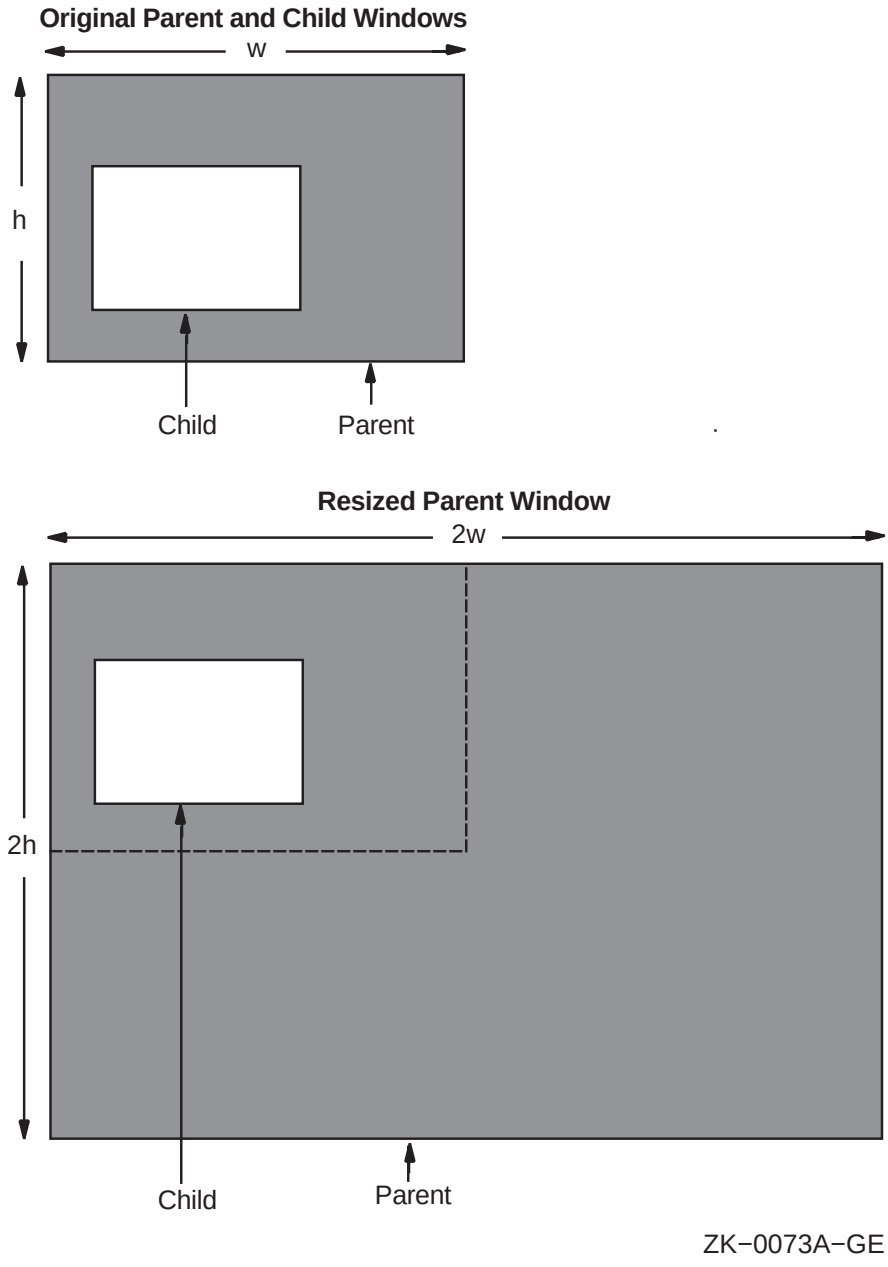


Figure 3–9 Northwest Window Gravity



3.7.3 Changing Stacking Order

Xlib provides routines that alter the window stacking order in the following ways:

- A specified window moves to either the top or the bottom of the stack.
- The lowest mapped child obscured by a sibling moves to the top of the stack.
- The highest mapped child that obscures a sibling moves to the bottom of the stack.

Use the RAISE WINDOW and LOWER WINDOW routines to move a specified window to either the top or the bottom of the stack, respectively.

To raise the lowest mapped child of an obscured window to the top of the stack, call CIRCULATE SUBWINDOWS UP. To lower the highest mapped child that obscures another child, call CIRCULATE SUBWINDOWS DOWN. The CIRCULATE SUBWINDOWS routine enables the client to perform these operations by specifying either the constant **RaiseLowest** or the constant **LowerHighest**.

To change the order of the window stack, use RESTACK WINDOW, which changes the window stack to a specified order. Reordered windows must have a common parent. If the first window the client specifies has other unspecified siblings, its order relative to those siblings remains unchanged.

3.7.4 Changing Window Attributes

Xlib provides routines that enable clients to change the following:

- Default contents of an input-output window
- Border of an input-output window
- Treatment of the window when it or its relative is obscured
- Treatment of the window when it or its relative is moved
- Information the window receives about operations associated with other windows
- Color
- Cursor

Section 3.2.2 includes descriptions of window attributes and their relationship to the set window attributes data structure.

This section describes how to change any attribute using the CHANGE WINDOW ATTRIBUTES routine. In addition to CHANGE WINDOW ATTRIBUTES, Xlib includes routines that enable clients to change background and border attributes. Table 3–11 lists these routines and their functions.

Table 3–11 Routines for Changing Window Attributes

Routine	Description
SET WINDOW BACKGROUND	Sets the background pixel
SET WINDOW BACKGROUND PIXMAP	Sets the background pixmap
SET WINDOW BORDER	Sets the window border to a specified pixel
SET WINDOW BORDER PIXMAP	Sets the window border to a specified pixmap

To change any window attribute, use CHANGE WINDOW ATTRIBUTES as follows:

1. Assign a value to the relevant member of a set window attributes data structure.
2. Indicate the attribute to change by specifying the appropriate flag in the CHANGE WINDOW ATTRIBUTES **value_mask** argument. To define more than one attribute, indicate the attributes by doing a bitwise OR on the appropriate flags.

See Table 3–3 for symbols Xlib assigns to each member to facilitate referring to the attributes.

Example 3–6 illustrates using CHANGE WINDOW ATTRIBUTES to redefine the characteristics of a window.

Example 3–6 Changing Window Attributes

```
    XSetWindowAttributes  xswa;

❶   xswa.background_pixel = BlackPixelOfScreen(dpy);
    xswa.border_pixel = WhitePixelOfScreen(dpy);

❷   XChangeWindowAttributes(dpy, win, CWBorderPixel | CWBackPixel, &xswa);
                        .
                        .
                        .
```

❶ Assign new values to a set window attributes data structure.

❷ Call CHANGE WINDOW ATTRIBUTES to change the window attributes. The CHANGE WINDOWS ATTRIBUTES routine has the following format:

XChangeWindowAttributes(display, window_id, attributes_mask, attributes)

Specify the attributes to change with a bitwise inclusive OR of the relevant symbols listed in Table 3–3. The **values** argument passes the address of a set window attributes data structure.

3.8 Getting Information About Windows

Using Xlib information routines, clients can get information about the parent, children, and number of children in a window tree; window geometry; the root window in which the pointer is currently visible; and window attributes.

Table 3–12 lists and describes Xlib routines that return information about windows.

Table 3–12 Window Information Routines

Routine	Description
QUERY TREE	Returns information about the window tree
GET GEOMETRY	Returns information about the root window identifier, coordinates, width and height, border width, and depth
QUERY POINTER	Returns the root window that the pointer is currently on and the pointer coordinates relative to the root window origin
GET WINDOW ATTRIBUTES	Returns information from the window attributes data structure

To get information about window attributes, use the GET WINDOW ATTRIBUTES routine. The client receives requested information in the window attributes data structure. See the *X Window System* for more information about the window attributes data structure.

4

Defining Graphics Characteristics

After opening a display and creating a window, clients can draw lines and shapes, create cursors, and draw text. Creating a graphics object is a two-step process. Clients first define the characteristics of the graphics object and then create it. For example, before creating a line, a client first defines line width and style. After defining the characteristics, the client creates the line with the specified width and style.

This chapter describes how to define the graphics characteristics prior to creating them, including the following topics:

- The graphics context (GC)—A description of the graphics characteristics a client can define and the GC values data structure used to define them
- Defining graphics characteristics—How to define graphics characteristics using the CREATE GC routine
- Copying, changing, and freeing attributes—How to copy, change, and undefine graphics characteristics
- Defining graphics characteristics efficiently—How to work efficiently with several sets of graphics characteristics

Chapter 6 describes how to create graphics objects. Chapter 8 describes how to work with text.

4.1 The Graphics Context

The characteristics of a graphics object make up its **graphics context**. As with window characteristics, Xlib provides a data structure and routine to enable clients to define multiple graphics characteristics easily. By setting values in the GC values data structure and calling the CREATE GC routine, clients can define all characteristics relevant to a graphics object.

Xlib also provides routines that enable clients to define individual or functional groups of graphics characteristics.

Xlib always records the defined values in a GC data structure, which is reserved for the use of Xlib and the server only. This occurs when clients define graphic characteristics using either the CREATE GC routine or one of the individual routines. Table 4–1 lists the default values of the GC data structure.

Table 4–1 GC Data Structure Default Values

Member Name	Default Value
Function	GXcopy
Plane mask	All ones
Foreground	0
Background	1
Line width	0
Line style	Solid
Cap style	Butt
Join style	Miter
Fill style	Solid
Fill rule	Even odd
Arc mode	Pie slice
Tile	Pixmap of unspecified size filled with foreground pixel
Stipple	Pixmap of unspecified size filled with ones
Tile or stipple x origin	0
Tile or stipple y origin	0
Font	Varies with implementation
Subwindow mode	Clip by children
Graphics exposures	True
Clip x origin	0
Clip y origin	0
Clip mask	None
Dash offset	0
Dashes	4 (the list [4,4])

4.2 Defining Multiple Graphics Characteristics in One Call

Xlib enables clients to define multiple characteristics of a graphics object in one call. To define multiple characteristics, use the CREATE GC routine as follows:

1. Assign values to the relevant members of the GC values data structure.
2. Indicate the attributes to define by specifying the appropriate flag in the **value_mask** argument of the routine. To define more than one attribute, perform a bitwise OR on the appropriate attribute flags.

The following illustrates the GC values data structure:

```
typedef struct {
        int function;
        unsigned long plane_mask;
        unsigned long foreground;
        unsigned long background;
        int line_width;
        int line_style;
        int cap_style;
        int join_style;
        int fill_style;
        int fill_rule;
        int arc_mode;
        Pixmap tile;
        Pixmap stipple;
        int ts_x_origin;
        int ts_y_origin;
        Font font;
        int subwindow_mode;
        Bool graphics_exposures;
        int clip_x_origin;
        int clip_y_origin;
        Pixmap clip_mask;
        int dash_offset;
        char dashes;
} XGCValues;
```

Table 4–2 describes the members of the data structure.

Table 4–2 GC Values Data Structure Members

Member Name	Contents
function	Defines how the server computes pixel values when the client updates a section of the screen.
plane_mask	Specifies the planes on which the server performs the bitwise computation of pixels, defined by the function member.
foreground	Specifies an index to a color map entry for foreground color.
background	Specifies an index to a color map entry for background color.
line_width	Defines the width of a line in pixels. Pixels with centers on a horizontal edge are a special case and are part of the line if, and only if, the interior is immediately below the bounding box (y increasing direction). See Figure 4–1.

(continued on next page)

Table 4–2 (Cont.) GC Values Data Structure Members

Member Name	Contents
line_style	Defines which sections of the line the server draws. The following lists available line styles and the constants that specify them.

Constant Name	Description
LineSolid	The full path of the line is drawn.
LineDoubleDash	The full path of the line is drawn, but the even dashes are filled differently than the odd dashes, with cap butt style used where even and odd dashes meet.
LineOnOffDash	Only the even dashes are drawn. The cap_style member applies to all internal ends of dashes. Specifying the constant CapNotLast is equivalent to specifying CapButt.

Figure 4–2 illustrates the line styles.

cap_style — Defines how the server draws the endpoints of a path. The following lists available cap styles and the constants that specify them.

Constant Name	Description
CapButt	Square at the endpoint (perpendicular to the slope of the line) with no projection beyond the endpoint.
CapNotLast	Equivalent to CapButt, except that the final endpoint is not drawn if the line width is zero or one.
CapRound	A circular arc with the diameter equal to the line width, centered on the endpoint (equivalent to the value specified by CapButt for a line width of zero or one).
CapProjecting	Square at the end, but the path continues beyond the endpoint for a distance equal to half the width of the line (equivalent to the value specified by the constant CapButt for a line width of zero or one).

Figure 4–3 illustrates the butt, round, and projecting cap styles.
Figure 4–4 illustrates the style specified by the constant CapNotLast.

join_style — Defines how the server draws corners for wide lines. Available join styles and the constants that specify them are as follows.

Constant Name	Description
JoinMiter	The outer edges of the two lines extend to meet at an angle
JoinRound	A circular arc with diameter equal to the line width, centered at the join point
JoinBevel	Cap butt endpoint style, with the triangular notch filled

Figure 4–5 illustrates the join styles.

fill_style — Specifies the contents of the source for line, text, and fill operations.

(continued on next page)

Table 4–2 (Cont.) GC Values Data Structure Members

Member Name	Contents
fill_rule	Defines what pixels the server draws along a path when a polygon is filled (see Section 6.5.2). The two available choices are EvenOddRule and WindingRule. The EvenOddRule constant defines a point to be inside a polygon if an infinite ray with the point as origin crosses the path an odd number of times. If the point meets these conditions, the server draws a corresponding pixel. The WindingRule constant defines a point to be inside the polygon if an infinite ray with the pixel as origin crosses an unequal number of clockwise-directed and counterclockwise-directed path segments. A clockwise-directed path segment is one that crosses the ray from left to right as observed from the pixel. A counterclockwise-directed segment is one that crosses the ray from right to left as observed from that point. When a directed line segment coincides with a ray, choose a different ray that is not coincident with a segment. If the point meets these conditions, the server draws a corresponding pixel. For both even/odd rule and winding rule, a point is infinitely small and the path is an infinitely thin line. A pixel is inside the polygon if the center point of the pixel is inside and the center point is not on the boundary. If the center point is on the boundary, the pixel is inside, if and only if, the polygon interior is immediately to its right (x increasing direction). Pixels with centers along a horizontal edge are a special case and are inside, if and only if, the polygon interior is immediately below (y increasing direction). Figure 4–6 illustrates fill rules. Figure 4–7 illustrates rules for filling a pixel when it falls on a boundary.
arc_mode	Controls how the server fills an arc. The available choices are the values specified by the ArcPieSlice and ArcChord constants. Figure 4–8 illustrates the two modes.
tile	Specifies the pixmap that the server uses for tiling operations.
stipple	Specifies the pixmap that the server uses for stipple operations.
ts_x_origin	Defines the origin for tiling and stipple operations. Origins are relative to the origin of whatever window or pixmap is specified in the graphics request.
ts_y_origin	Defines the origin for tiling and stipple operations. Origins are relative to the origin of whatever window or pixmap is specified in the graphics request.
font	Specifies the font that the server uses for text operations.
subwindow_mode	Specifies whether or not inferior windows clip superior windows.
graphics_exposures	Specifies whether or not the server informs the client when the contents of a window region are lost.
clip_x_origin	Defines the x-coordinate of the clip origin. The clip origin specifies the point within the clip region that is aligned with the drawable origin.
clip_y_origin	Defines the y-coordinate of the clip origin. The clip origin specifies the point within the clip region that is aligned with the drawable origin.
clip_mask	Identifies the pixmap that the server uses to restrict write operations to the destination drawable. When a client specifies the value of clip mask as None, the server draws all pixels.

(continued on next page)

Table 4–2 (Cont.) GC Values Data Structure Members

Member Name	Contents
dash_offset	Specifies the pixel within the dash length sequence, defined by the dashes member, to start drawing a dashed line. For example, a dash offset of zero starts a dashed line as the beginning of the dash line sequence. A dash offset of five starts the line at the fifth pixel of the line sequence. Figure 4–9 illustrates dashed offsets.
dashes	Specifies the length, in number of pixels, of each dash. The value of this member must be nonzero or an error occurs.

Figure 4–1 illustrates how a bounding box affects line width.

Figure 4–1 Bounding Box

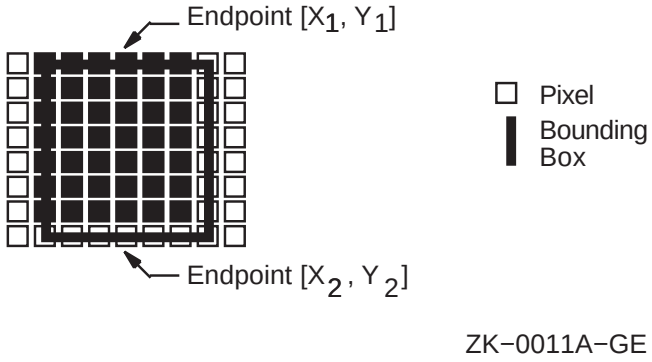


Figure 4–2 illustrates line styles.

Figure 4–2 Line Styles

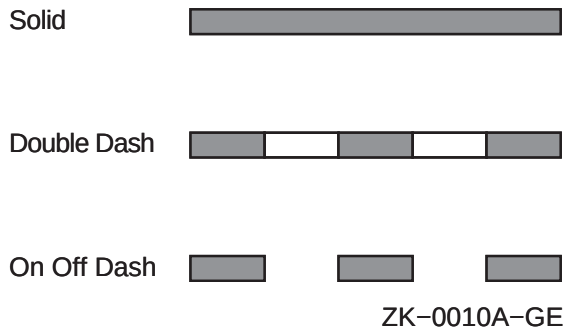


Figure 4–3 illustrates line cap styles.

Figure 4–3 Butt, Round, and Projecting Cap Styles

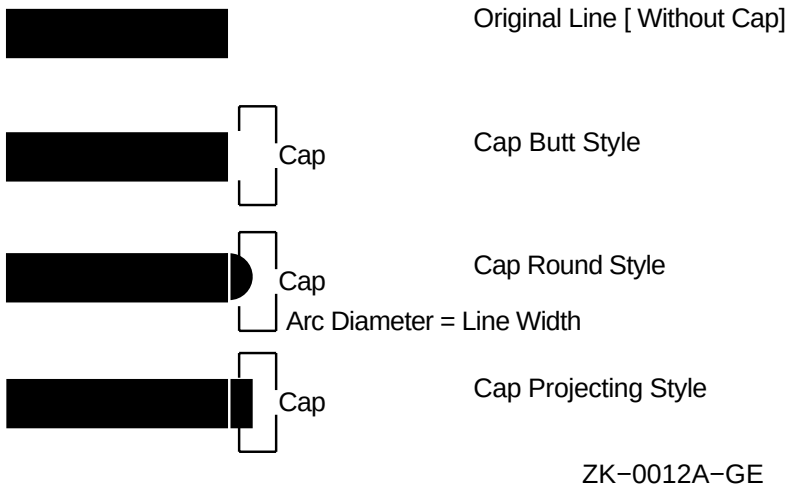


Figure 4–4 illustrates the line specified by the CapNotLast constant.

Figure 4–4 Cap Not Last Style

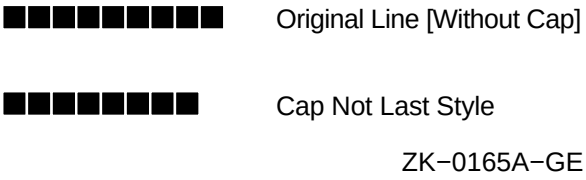


Figure 4–5 illustrates the join styles.

Figure 4–5 Join Styles

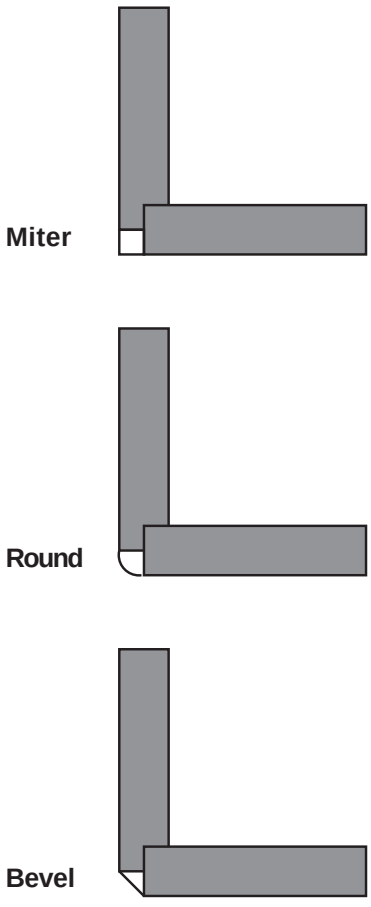


Figure 4–6 illustrates the fill rules.

Figure 4–6 Fill Rules

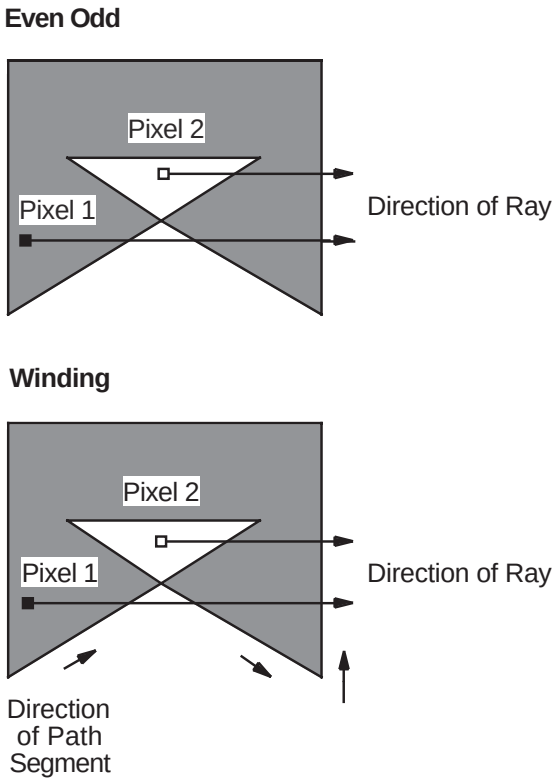


Figure 4–7 illustrates the rules for filling a pixel when it falls on a boundary.

Figure 4–7 Pixel Boundary Cases

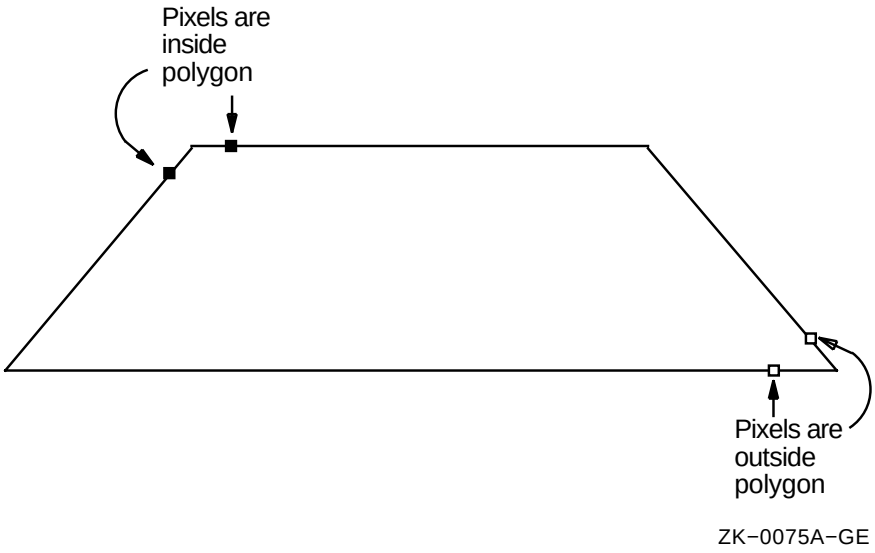


Figure 4–8 illustrates how an arc is filled.

Figure 4–8 Styles for Filling Arcs

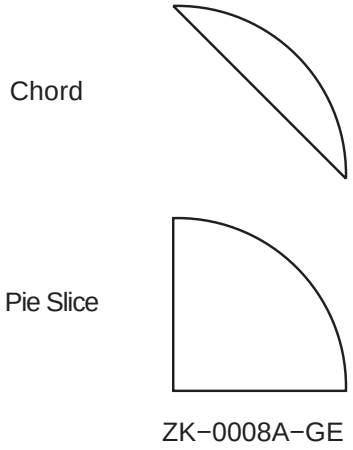
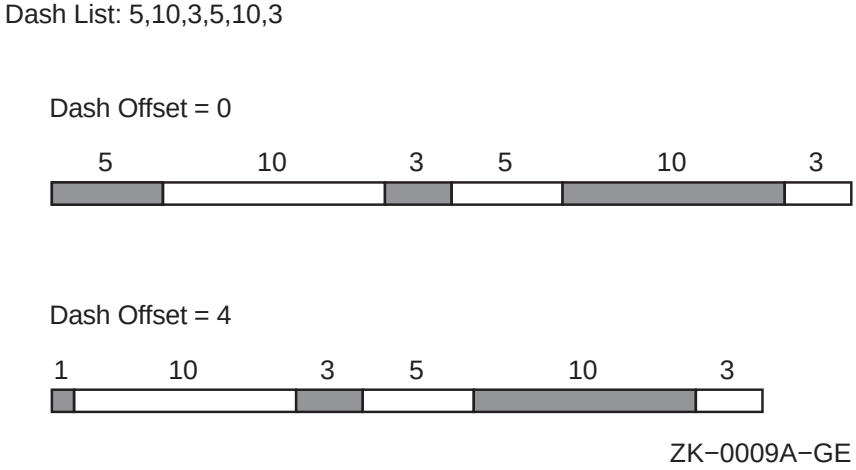


Figure 4–9 illustrates dash offsets.

Figure 4–9 Dashed Line Offset



Xlib assigns a flag for each member of the GC values data structure to facilitate referring to members (Table 4–3).

Table 4–3 GC Values Data Structure Flags

Flag Name	GC Values Member
GCFunction	function
GCPlaneMask	plane_mask
GCForeground	foreground
GCBackground	background

(continued on next page)

Table 4–3 (Cont.) GC Values Data Structure Flags

Flag Name	GC Values Member
GCLineWidth	line_width
GCLineStyle	line_style
GCCapStyle	cap_style
GCJoinStyle	join_style
GCFillStyle	fill_style
GCFillRule	fill_rule
GCTile	tile
GCStipple	stipple
GCTileStipXOrigin	ts_x_origin
GCTileStipYOrigin	ts_y_origin
GCFont	font
GCSubwindowMode	subwindow_mode
GCGraphicsExposures	graphics_exposures
GCClipXOrigin	clip_x_origin
GCClipYOrigin	clip_y_origin
GCXClipMask	clip_mask
GCDashOffset	dash_offset
GCDashList	dash_list
GCArcMode	arc_mode

Example 4–1 illustrates how a client can define graphics context values using the CREATE GC routine. Figure 4–10 shows the resulting output.

Example 4–1 Defining Graphics Characteristics Using the CREATE GC Routine

```
    /* Create window win on                                  *
     *  display dpy, defined as follows:                     *
     *     Position: x = 100,y = 100                         *
     *     Width = 600                                       *
     *     Height = 600                                      *
     *  gc refers to the graphics context                    */
                    .
                    .
                    .
❶GC gc;
                    .
                    .
                    .
static void doCreateGraphicsContext( )
{
❷   XGCValues xgcv;
    /* Create graphics context. */
```

(continued on next page)

Example 4–1 (Cont.) Defining Graphics Characteristics Using the CREATE GC Routine

```
❸  xgcv.foreground = doDefineColor(3);
    xgcv.background = doDefineColor(4);
    xgcv.line_width = 4;
    xgcv.line_style = LineDoubleDash;
    xgcv.dash_offset = 0;
    xgcv.dashes = 25;

❹  gc = XCreateGC(dpy, win, GCForeground | GCBackground
        | GCLineWidth | GCLineStyle | GCDashOffset | GCDashList, &xgcv);
}
                    .
                    .
                    .
static void doButtonPress(eventP)
XEvent *eventP;
{
    x1 = y1 = 100;
    x2 = y2 = 550;

❺  XDrawLine(dpy, win, gc, x1, y1, x2, y2);
}
```

❶ Assign storage for a graphics context (GC) data structure. The scope of *gc* is global to enable windowing and graphics routines in other modules to refer to it.

❷ Once the client defines characteristics with the GC values data structure, Xlib does not have to refer to the data structure again.

❸ Specify the foreground, background, line width, line style, dash offset, and dashes for line drawing.

The dashed line is four pixels wide. A dash offset value of zero starts dashes at the beginning of the line. The dashes value, referred to by GCDashList, specifies that dashes be 25 pixels long.

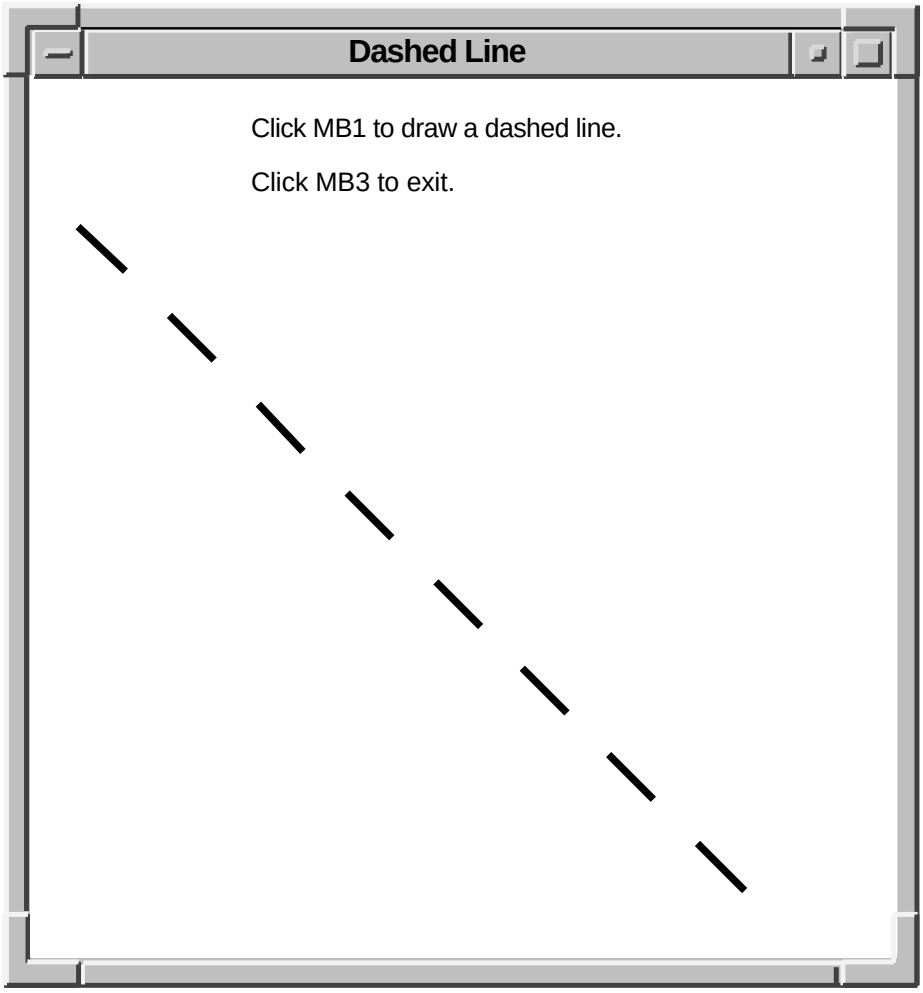
❹ The CREATE GC routine loads values into a GC data structure. The CREATE GC routine has the following format:

gc_id = XCreateGC (display, drawable_id, gc_mask, values_struc)

Indicate defined attributes with a bitwise OR that uses symbols listed in Table 4–3.

❺ See Chapter 6 for information about drawing lines.

Figure 4–10 Dashed Line



ZK–2511A–GE

4.3 Defining Individual Graphics Characteristics

Xlib offers routines that enable clients to define individual or functional groups of graphics characteristics. Table 4–4 lists and briefly describes these routines. For more information about the components, see Section 4.1.

Table 4–4 Routines That Define Individual or Functional Groups of Graphics Characteristics

Routine	Description
Foreground, Background, Plane Mask, and Function Routines	
SET STATE	Sets the foreground, background, plane mask, and function
SET FOREGROUND	Sets the foreground
SET BACKGROUND	Sets the background
SET PLANE MASK	Sets the plane mask
SET FUNCTION	Sets the function
Line Attribute Routines	
SET LINE ATTRIBUTES	Sets line width, line style, cap style, and join style
SET DASHES	Sets the dash offset and dash list of a line
Fill Style and Rule Routines	
SET FILL STYLE	Sets fill style to solid, tiled, stippled, or opaque stippled
SET FILL RULE	Sets fill rule to either even and odd or winding rule
Fill Tile and Stipple Routines	
QUERY BEST SIZE	Queries the server for the size closest to the one specified
QUERY BEST STIPPLE	Queries the server for the closest stipple shape to the one specified
QUERY BEST TILE	Queries the server for the closest tile shape to the one specified
SET STIPPLE	Sets the stipple pixmap
SET TILE	Sets the tile pixmap
SET TS ORIGIN	Sets the tile or stipple origin
Font Routine	
SET FONT	Sets the current font
Clip Region Routines	
SET CLIP MASK	Sets the mask for bitmap clipping
SET CLIP ORIGIN	Sets the origin for clipping
SET CLIP RECTANGLES	Changes the clip mask from its current value to the specified rectangles

(continued on next page)

Table 4–4 (Cont.) Routines That Define Individual or Functional Groups of Graphics Characteristics

Routine	Description
Arc, Subwindow, and Exposure Routines	
SET ARC MODE	Sets the arc mode to either chord or pie slice
SET SUBWINDOW MODE	Sets the subwindow mode to either clip by children or include inferiors
SET GRAPHICS EXPOSURES	Specifies whether exposure events are created when calling COPY AREA or COPY PLANE

Example 4–2 illustrates using individual routines to set background, foreground, and line attributes. Figure 4–11 illustrates the resulting output.

Example 4–2 Using Individual Routines to Define Graphics Characteristics

```
GC gc;
                    .
                    .
                    .
static void doButtonPress(eventP)
XEvent *eventP;
{
❶   char dash_list[] = {20,5,10};
    x1 = y1 = 100;
    x2 = y2 = 550;

    XSetBackground(dpy, gc, doDefineColor(4));
❷   XSetLineAttributes(dpy, gc, 0, LineDoubleDash, 0, 0);
❸   XSetDashes(dpy, gc, 0, dash_list, 3);
    XDrawLine(dpy, win, gc, x1, y1, x2, y2);
}
```

❶ The *dash_list* variable defines the length of odd and even dashes. The first and third elements of the initialization list specify even dashes; the second element specifies odd dashes.

❷ The SET LINE ATTRIBUTES routine enables the client to define line width, style, cap style, and join style in one call.

The SET LINE ATTRIBUTES routine has the following format:

XSetLineAttributes(display, gc_id, line_width, line_style, cap_style, join_style)

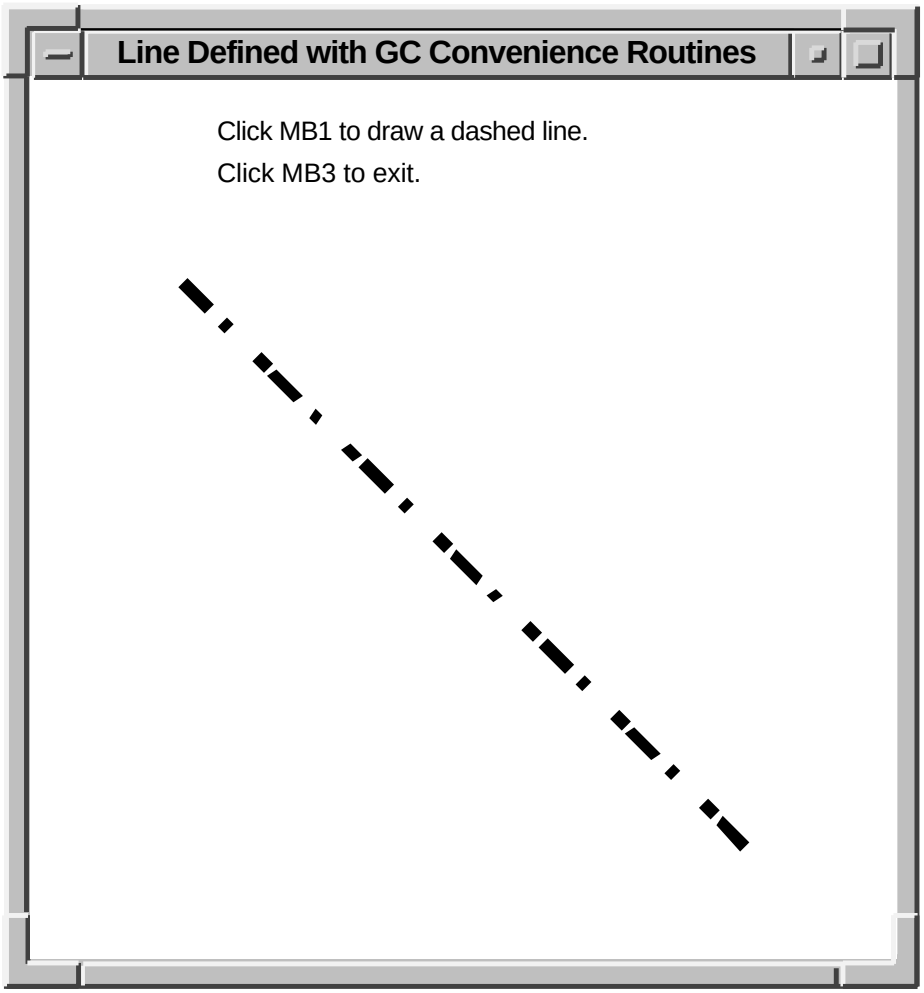
The zero **cap_style** argument specifies the default cap style.

❸ If using the CREATE GC routine to set line dashes, odd and even dashes must have equal length. The SET DASHES routine enables the client to define dashes of varying length. The SET DASHES routine has the following format:

XSetDashes(display, gc_id, dash_offset, dash_list, dash_list_len)

The **dash_list_len** argument specifies the length of the dash list.

Figure 4–11 Line Defined Using GC Routines



ZK–2570A–GE

4.4 Copying, Changing, and Freeing Graphics Contexts

In addition to defining a graphics context, clients can copy defined characteristics from one GC data structure into another. To copy a GC data structure, use COPY GC. The COPY GC routine has the following format:

XCopyGC(display, src_gc_id, gc_mask, dst_gc_id)

The **gc_mask** argument selects values to be copied from the source graphics context (**src_gc_id**). Use the method described in Section 4.2 for assigning values to a GRAPHICS CONTEXT.

The **dst_gc_id** argument specifies the new graphics context into which the server copies values.

After creating a graphics context structure, change values as needed using CHANGE GC. The following code fragment, which alters the values of the line drawn by Example 4–1, illustrates changing a graphics context structure:

```
                .
                .
                .
xgcv.line_width = 10;
xgcv.line_style = LineSolid;

XChangeGC(dpy,gc,GCLineWidth | GCLineStyle,&xgcv);
                .
                .
                .
```

The previous example illustrates defining a new line style and width, and changing the graphics context to reflect the new values.

4.5 Using Graphics Characteristics Efficiently

The server must revalidate a graphics context whenever a client redefines it. Causing the server to revalidate a graphics context unnecessarily can seriously degrade performance.

The server revalidates a graphics context when one of the following conditions occurs:

- A client associates the graphics context with a different window.
- The graphics context clip list changes. Changes in the clip list can happen either when a client changes the graphics context clip origin or when the server modifies the clip list in response to overlapping windows.
- Any member of the graphics context changes.

To minimize revalidating the graphics context, submit as a group the requests to the server that identify the same window and graphics context. Grouping requests enables the server to revalidate the graphics context once instead of many times.

When it is necessary to change the value of graphics context members frequently, creating a new graphics context is more efficient than redefining an existing one, provided the client creates no more than 50 graphics contexts.

5
Using Color

Color is one of many attributes that clients can define when creating a window or a graphics object. Depending on display hardware, clients can define color as black or white, as shades of gray, or as a spectrum of hues. Section 5.2 describes color definition in detail.

Xlib offers clients the choice of either sharing colors with other clients or, when hardware supports it, allocating colors for exclusive use.

A client that does not have to change colors can share them with other clients. By sharing colors, the client saves color resources.

When a client needs to change colors, the client must allocate them for its exclusive use. For example, the client might indicate the flow through a pipeline by changing colors, rather than redrawing the entire pipeline schematic. In this case, the client would allocate for exclusive use colors that represent pipeline flow.

This chapter introduces color management using Xlib and describes how to share and allocate color resources. The chapter includes the following topics:

- Color fundamentals—A description of pixels and planes, color indices, cells, and maps
- Matching color requirements to display types—How display types affect color presentation
- Sharing color resources—How to share color resources with other clients
- Allocating colors for exclusive use—How to reserve colors for a single client
- Querying color resources—How to return values of color map entries
- Freeing color resources—How to release color resources

The concepts presented in this chapter apply to managing the color of both windows and graphic objects.

5.1 Pixels and Color Maps

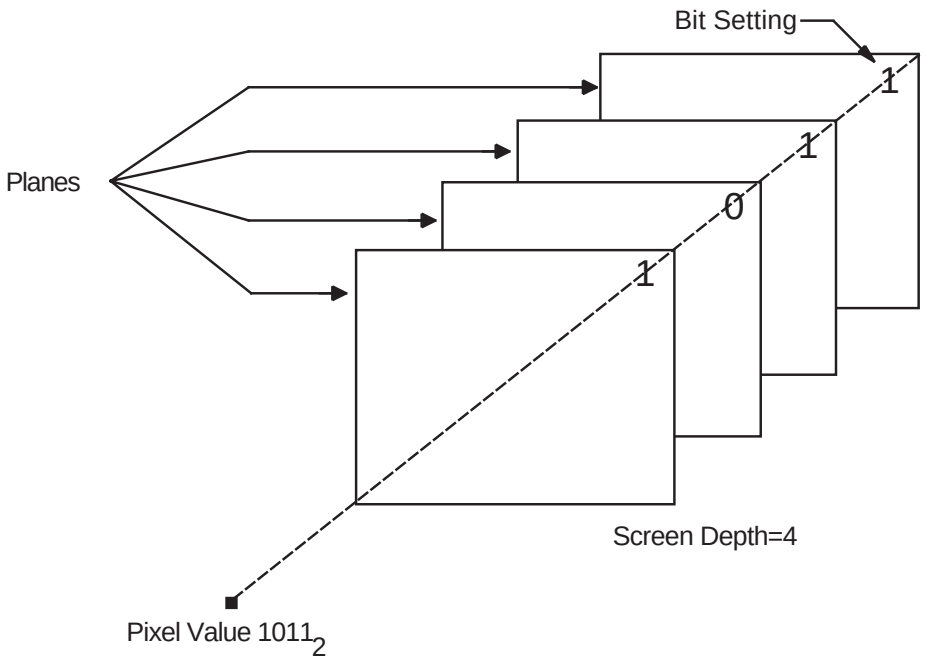
The color of a window or graphics object depends on the values of pixels that constitute it. The number of bits associated with each pixel determines the number of possible pixel values. On a monochrome screen, one bit corresponds to each pixel. The number of possible pixel values is 2. Pixels are either zero or one, black or white.

On a monochrome screen, all bits that define an image reside on one **plane**. A plane is an allocation of memory with a one-to-one correspondence between bits and pixels. The number of planes is the **depth** of the screen.

The depth of intensity of color screens is greater than one. More than one bit defines the value of a pixel. Each bit associated with the pixel resides on a different plane.

The number of possible pixel values increases as depth increases. For example, if the screen has a depth of four planes, the value of each pixel comprises four bits. Clients using a four-plane intensity display can produce up to sixteen levels of brightness. Clients using a four-plane color display can produce as many as sixteen colors. The number of colors possible on any system is equal to 2^n, where n is the number of planes. Figure 5–1 illustrates the relationship between pixel values and planes.

Figure 5–1 Pixel Values and Planes

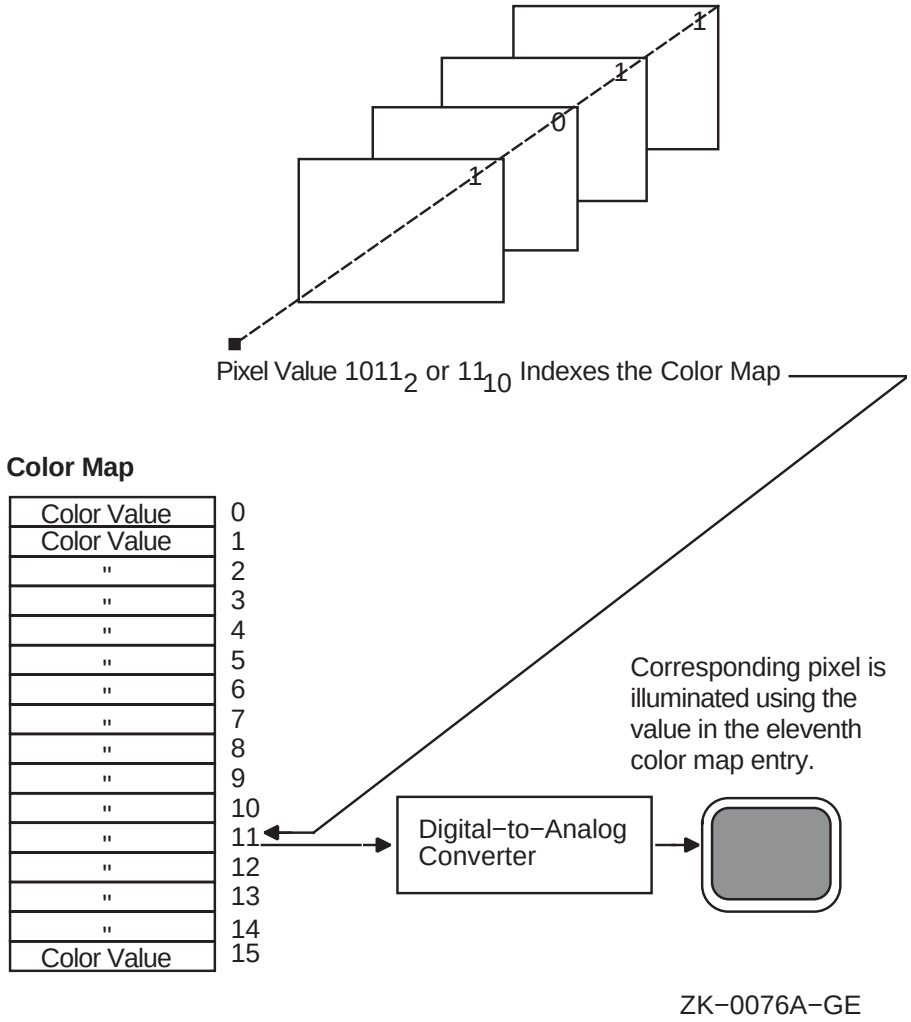


ZK−0074A−GE

Xlib uses **color maps** to define the color of each pixel value. A color map contains a collection of **color cells**, each of which defines the color represented by a pixel value in terms of its red, green, and blue (RGB) components. Red, green, and blue components range from zero (off) to 65535 (brightest) inclusively. By combining the RGB components, many colors can be produced.

Each pixel value refers to a location in a color map or is an **index** into a color map. For example, the pixel value illustrated in Figure 5–1 indexes color cell 11 in Figure 5–2.

Figure 5–2 Color Map, Cell, and Index



Most color workstations have a hardware color map that translates pixel values into colors for the entire workstation screen. When the color definitions from a client's color map are stored in the hardware color map, that color map is said to be installed. If a client's color map is not installed, the client's windows will display in the wrong color.

For example, an image processing program that requires 128 colors might allocate and store a color map of these values. To alter some colors, another client may invoke a color palette program that chooses and mixes colors. The color palette program itself requires a color map, which the program allocates and installs.

Because both programs have allocated different color maps, undesirable results can be produced. The color palette image may be incorrectly displayed when the image processing program runs. The incorrect display results because only the image processing color map is installed. Conversely, when the color palette program runs, the image processing program may be incorrectly displayed because only the color palette color map is installed.

Xlib reduces the problem of contending for color resources in two ways:

- Xlib provides a default color map to which all clients have access.

- Clients can allocate either color cells for exclusive use or colors for shared use from the default color map.

By sharing colors, a client can use the same color cells as other clients. This method conserves space in the default color map.

In cases where the client cannot use the default color map and must use a new color map, Xlib creates virtual color maps. The use of virtual color maps is analogous to the use of virtual memory in a multiprogramming environment where many processes must access physical memory. When concurrent processes collectively require more color map entries than exist in the hardware color map, the color values are swapped in and out of the hardware color map. However, swapping virtual color maps in and out of the hardware color map causes contention for color resources. Therefore, the client should avoid creating color maps whenever possible.

5.1.1 Installing Color Maps

The process of loading or unloading color values of the virtual color map into the hardware lookup table occurs when a client calls the INSTALL COLORMAP or UNINSTALL COLORMAP routine. Typically, the privilege to install or to remove color maps is restricted to the window manager. The window manager installs a color map when a window is given focus. The user gives a window focus by clicking on it with the mouse. The window manager then installs the color map for that window.

On a system with a single hardware color map, only one window can have color map focus at a time. Giving the focus to a new window will cause the previous window that had the focus to display in the wrong color.

Some systems provide multiple color maps in hardware. Multiple windows can have color map focus simultaneously. Each window, however, must be clicked on to install the correct color map and to get the correct colors.

Applications that have a window manager running should not make direct calls to install color maps. The window manager may reinstall different color maps if the client attempts to install a private color map. However, on a system with multiple color maps, the window manager will not remove the private color map. Thus, the client will display in correct colors without getting color map focus.

Applications that require subwindows to have color maps separate from the top-level window can use the SET WM COLORMAP WINDOWS routine. This routine provides a hint to the window manager to install the specified color map. Normally, window managers install color maps only for the top-level window. Some applications are designed to run without a window manager. In this case, the application must issue requests to install its own color map.

5.2 Matching Color Requirements to Display Types

The basic philosophy, when using color, is to determine the color needs of the client and then to determine how the system can best support those needs.

This section defines the different visual display types available and describes methods to choose the appropriate type for the client.

5.2.1 Visual Types

Each screen has a list of **visual types** associated with it. The visual type identifies the characteristics of the screen, such as color or monochrome capability. Visual types partially determine the appearance of color on the screen and determine how a client can manipulate color maps for a specified screen.

Color maps can be manipulated in a variety of ways on some hardware, in a limited way on other hardware, and not at all on yet other hardware. For example, a screen may be able to display a full range of colors or a range of grays only, depending on its visual type.

VMS DECwindows defines the following visual types:

- Pseudocolor
- Gray scale
- Direct color
- True color
- Static gray
- Static color

Pseudocolor is a full-color device. A pixel value indexes a color map composed of red, green, and blue definitions. Each definition in the color map stores the red, green, and blue component values for one color. The color index refers directly to a single entry in the color map. RGB values can be changed dynamically if a pixel has been allocated for exclusive use. Pseudocolor is the default visual type on Digital 4-plane and 8-plane systems.

In Figure 5–3, the pseudocolor illustration shows a pixel value of 2 (00000010 in binary) that indexes entry 2 in the color map.

Gray scale is a black and white device. Gray scale is the same as pseudocolor except that a pixel value indexes a color map that produces shades of gray only. The gray shades are defined in a color map with each definition having just one component that defines the level of the white intensity.

Refer to Figure 5–3 for an illustration of the gray scale visual type.

Direct color is a full-color device. Both the pixel value and the color map are separated into three independent parts, one each for red, green, and blue. The red part of the pixel indexes the red color map, the green indexes the green color map, and the blue indexes the blue color map. A complete color definition comprises the three components in each color map. RGB values can be changed dynamically if a pixel has been allocated for exclusive use.

In Figure 5–3, the direct color illustration shows that a pixel value of 90 (01011010 in binary) is separated into three values by using color masks, which are defined in the visual info data structure. (Refer to Section 5.2.3 for information about the visual info data structure.) Each color mask indicates which bits of the pixel value reference which color map. Each value is then used to index one of the three structures. In this case, entry 2 is indexed in the red color map, entry 6 in the green color map, and entry 2 in the blue color map.

True color is a full-color device. True color is the same as direct color except that the color map has predefined read-only RGB values in ascending order. True color is the default visual type on a Digital 24-plane system.

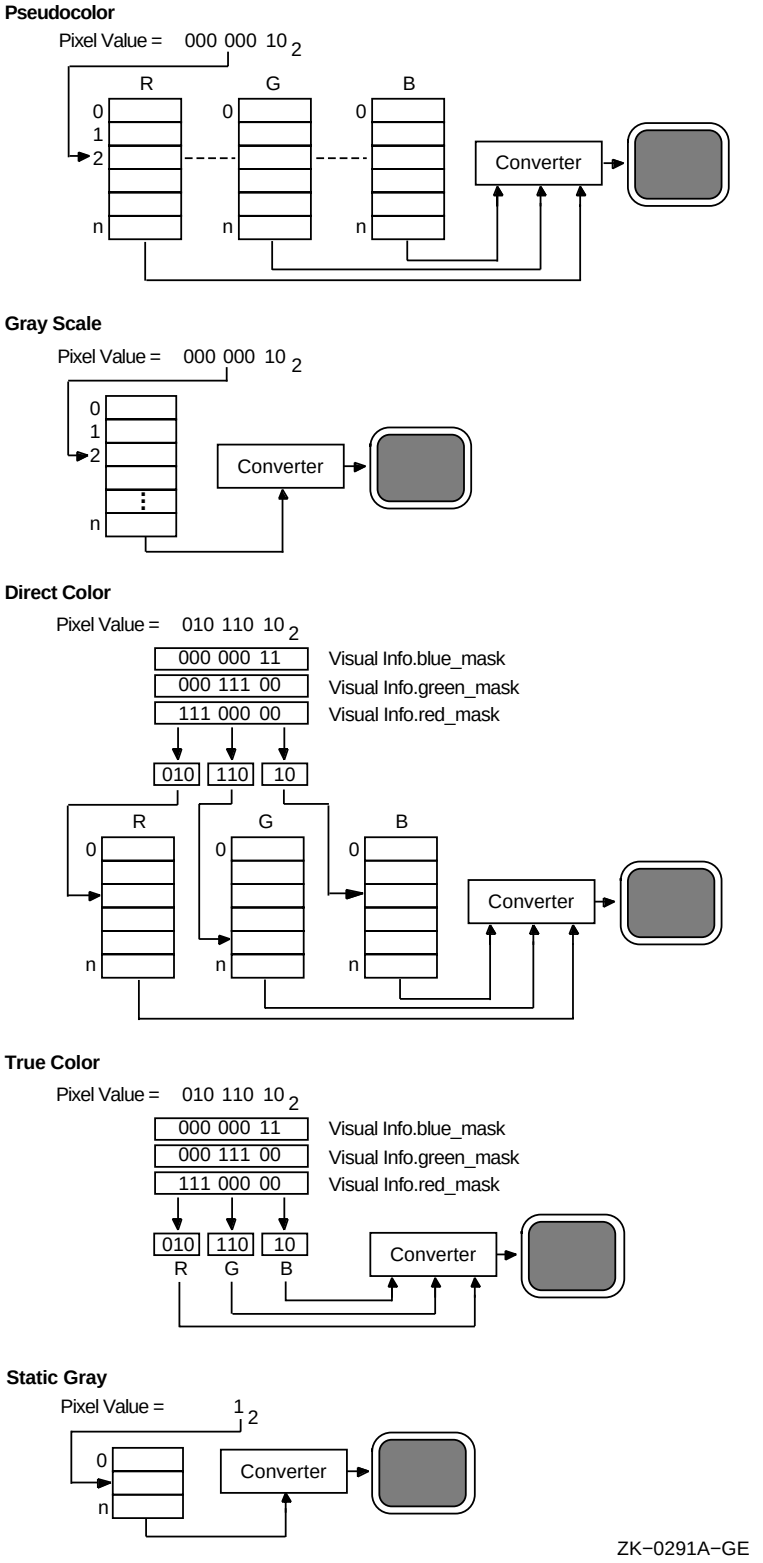
Refer to Figure 5–3 for an illustration of the true color visual type.

Static gray is a black and white device. Static gray is the same as gray scale except that the values in the color map are read-only. Static gray with a two-entry color map can be thought of as monochrome.

Refer to Figure 5–3 for an illustration of the static gray visual type.

Static color is a full-color device and is the same as pseudocolor except that the color map has predefined, read-only, server-dependent values in an undefined, server-dependent order.

Figure 5–3 Visual Types and Color Map Characteristics



5.2.2 Determining the Default Visual Type

Before defining colors, use the following method to determine the default visual type of a screen:

1. Use the DEFAULT VISUAL OF SCREEN routine to determine the identifier of the visual. Xlib returns the identifier to a visual data structure.
2. Refer to the class member of the data structure to determine the visual type.

The following example illustrates one method to determine the default visual type of a screen:

```
                        .
                        .
                        .
    if ((XDefaultVisualOfScreen(screen))->class == TrueColor
         ||  (XDefaultVisualOfScreen(screen))->class ==
              PseudoColor
         ||  (XDefaultVisualOfScreen(screen))->class ==
              DirectColor
         ||  (XDefaultVisualOfScreen(screen))->class ==
              StaticColor)
                        .
                        .
                        .
```

5.2.3 Determining Multiple Visual Types

On some systems, a single display can support multiple screens. Each screen can have several different visual types supported at different depths. Xlib provides routines that allow a client to search and choose the appropriate visual type on the system by using the visual info data structure.

The following illustrates the visual info data structure:

```
typedef struc {
        Visual *visual;
        VisualID *visualid;
        int screen;
        int depth;
        int class;
        unsigned long red_mask;
        unsigned long green_mask;
        unsigned long blue_mask;
        int colormap_size;
        int bits_per_rgb;
}XVisualInfo;
```

Table 5–1 describes the members of the visual info data structure.

Table 5–1 Visual Info Data Structure Members

Member Name	Contents
visual	A pointer to a visual data structure that is returned to the client.
visualid	The id of the visual that is returned by the server.
screen	The specified screen of the display.
depth	The depth in planes of the screen.

(continued on next page)

Table 5–1 (Cont.) Visual Info Data Structure Members

Member Name	Contents
class	The class of the visual (PseudoColor, GrayScale, DirectColor, TrueColor, StaticGray, StaticColor).
red_mask	Definition of the red mask.[1]
green_mask	Definition of the green mask.[1]
blue_mask	Definition of the blue mask.[1]
colormap_size	Number of available color map entries.
bits_per_rgb	Number of bits that specifies the number of distinct red, green and blue values. Actual RGB values are unsigned 16-bit numbers.

[1]The red mask, green mask, and blue mask are defined only for the direct color and true color visual types.

Use the GET VISUAL INFO routine to return a list of visual structures that match a specified template.

The GET VISUAL INFO routine has the following format:

XGetVisualInfo(display, vinfo_mask, vinfo_template, num_items_return)

Example 5–4 illustrates using the GET VISUAL INFO routine.

Use the MATCH VISUAL INFO routine to return the visual information for a visual type that matches the specified depth and class for a screen. Because multiple visual types that match the specified depth and class can exist, the exact visual chosen is undefined.

Note that the MATCH VISUAL INFO routine is a convenience routine that matches one visual of a particular class and depth. The GET VISUAL INFO routine, however, can find any number of visuals that match any combination of characteristics.

Example 5–1 illustrates using the MATCH VISUAL INFO routine to find a pseudocolor visual type on a 24-plane system.

Example 5–1 Matching Visual Information

```
            .
            .
            .
❶#define max_supported_planes 24
❷XVisualInfo vInfo;
            .
            .
            .
screen = XDefaultScreenOfDisplay(dpy);
scrNum = XDefaultScreen(dpy);
            .
            .
            .
/***** Match the visual *****/
static void doMatchVisual( )
{
    for (i = 4; (i <= max_supported_planes && !status); i++)
❸    status = XMatchVisualInfo(dpy, scrNum, i, PseudoColor, &vInfo);
```

(continued on next page)

Example 5–1 (Cont.) Matching Visual Information

```
    if (!status){
        printf ("Could not find a Pseudocolor visual on this system");
        exit(1);
        }
}
/***** Create the color *****/
static void doCreateColor( )
{
❹   if (vInfo.visual != DefaultVisual(dpy,scrNum))
        map = XCreateColormap(dpy,RootWindow(dpy,scrNum),vInfo.visual,
           AllocNone);
    else map = XDefaultColormapOfScreen(screen);
           .
           .
           .
```

❶ The client defines the maximum number of planes, or depth, supported on the system.

❷ Storage is assigned for the visual info data structure.

❸ The MATCH VISUAL INFO routine searches for a pseudocolor visual type beginning at the fourth plane. If a match is found, MATCH VISUAL INFO returns the visual information to the visual info data structure. If a match is not found, the next depth is checked for a pseudocolor visual.

The MATCH VISUAL INFO routine has the following format:

XMatchVisualInfo(display, screen_number, depth, class,
vinfo_return)

❹ The client compares the visual member of the visual info data structure returned by the MATCH VISUAL INFO routine with the default visual. If the default visual is a pseudocolor type, then the client uses the default color map. If the visual is not the default visual, the client creates a color map.

Refer to Section 5.4.1 for more information about creating color maps.

5.3 Sharing Color Resources

Xlib provides the following ways to share color resources:

- Using named VMS DECwindows colors
- Specifying exact color values

The choice of using a named color or specifying an exact color depends on the needs of the client. For instance, if the client is producing a bar graph, specifying the named VMS DECwindows color "Red" as a color value may be sufficient, regardless of the hue that VMS DECwindows names "Red". However, if the client is reproducing a portrait, specifying an exact red color value might be necessary to produce accurate skin tones. For a list of named colors, see the SYS$MANAGER:DECW$RGB.COM file.

Note that because of differences in hardware, no two monitors display colors exactly the same, even though the same named colors are specified.

5.3.1 Using Named Colors

VMS DECwindows includes named colors that clients can share. To use a named color, call the ALLOC NAMED COLOR routine. ALLOC NAMED COLOR determines whether the color map defines a value for the specified color. If the color exists, the server returns the index to the color map. If the color does not exist, the server returns an error.

Example 5–2 illustrates specifying a color using ALLOC NAMED COLOR.

Example 5–2 Using Named VMS DECwindows Colors

```
static int doDefineColor(n)
{
    int pixel;
❶   XColor exact_color,screen_color;
❷   char *colors[ ] = {
        "dark slate blue",
        "light grey",
        "firebrick"
        };

    if ((XDefaultVisualOfScreen(screen))->class == TrueColor
         ||  (XDefaultVisualOfScreen(screen))->class ==
             PseudoColor
         ||  (XDefaultVisualOfScreen(screen))->class ==
             DirectColor
         ||  (XDefaultVisualOfScreen(screen))->class ==
             StaticColor)
        {
❸       if (XAllocNamedColor(dpy, DefaultColormapOfScreen(screen),
            colors[n-1], &screen_color, &exact_color))
                return screen_color.pixel;
            else
                printf("Color not allocated!");

        }
    else
        printf("Not a color device!");
               .
               .
               .
```

❶ The client allocates storage for two color data structures: *exact_color* defines the RGB values specified by the VMS DECwindows named color. *Screen_color* defines the closest RGB values supported by the hardware.

For an illustration of the color data structure, see Section 5.3.2.

❷ An array of characters stores the names of the predefined VMS DECwindows colors that the client uses.

❸ The ALLOC NAMED COLOR routine has the following format:

XAllocNamedColor(display, colormap_id, color_name,
screen_def_return, exact_def_return)

The client passes the names of VMS DECwindows colors by referring to the array *colors*.

5.3.2 Specifying Exact Color Values

To specify exact color values, use the following method:

1. Assign values to a color data structure.
2. Call the ALLOC COLOR routine, specifying the color map from which the client allocates the definition. ALLOC COLOR returns a pixel value and changes the RGB values to indicate the closest color supported by the hardware.

Xlib provides a color data structure enabling clients to specify exact color values when sharing colors. (Routines that allocate colors for exclusive use and that query available colors also use the color data structure. For information about using the color data structure for these purposes, see Section 5.4.)

The following illustrates the color data structure:

```
typedef struct {
        unsigned long pixel;
        unsigned short red, green, blue;
        char flags;
        char pad;
} XColor;
```

Table 5–2 describes the members of the color data structure.

Table 5–2 Color Data Structure Members

Member Name	Contents	
pixel	Pixel value	
red	Specifies the red value of the pixel [1]	
green	Specifies the green value of the pixel [1]	
blue	Specifies the blue value of the pixel [1]	
flags	Defines which color components are to be changed in the color map. Possible flags are as follows:	
	DoRed	Sets red values
	DoGreen	Sets green values
	DoBlue	Sets blue values
pad	Makes the data structure an even length	

[1]Color values are scaled between 0 and 65535. "On full" in a color is a value of 65535, independent of the number of planes of the display. Half brightness in a color is a value of 32767; off is a value of 0. This representation gives uniform results for color values across displays with different color resolution.

Example 5–3 illustrates how to specify exact color definitions.

Example 5–3 Specifying Exact Color Values

(continued on next page)

Example 5–3 (Cont.) Specifying Exact Color Values

```
/***** Create color *****/
static int doDefineColor(n)
{
    int pixel;
    XColor colors[3];

    if ((XDefaultVisualOfScreen(screen))->class == TrueColor
         ||  (XDefaultVisualOfScreen(screen))->class ==
              PseudoColor
         ||  (XDefaultVisualOfScreen(screen))->class ==
              DirectColor
         ||  (XDefaultVisualOfScreen(screen))->class ==
              StaticColor)
        switch (n){
                 case 1:{
❶                    colors[n - 1].red = 59904;
                     colors[n - 1].green = 44288;
                     colors[n - 1].blue = 59904;
❷                    if (XAllocColor(dpy, XDefaultColormapOfScreen(screen),
                          &colors[n - 1]))
                               return colors[n - 1].pixel;
                          else
                               printf("Color not allocated!");
                     return;
                     }
                 case 2:{
                     colors[n - 1].red = 65280;
                     colors[n - 1].green = 0;
                     colors[n - 1].blue = 32512;
                     if (XAllocColor(dpy, XDefaultColormapOfScreen(screen),
                          &colors[n - 1]))
                               return colors[n - 1].pixel;
                          else
                               printf("Color not allocated!");
                     return;
                     }
                 case 3:{
                     colors[n - 1].red = 37632;
                     colors[n - 1].green = 56064;
                     colors[n - 1].blue = 28672;
                     if (XAllocColor(dpy, XDefaultColormapOfScreen(screen),
                         &colors[n - 1]))
                               return colors[n - 1].pixel;
                          else
                               printf("Color not allocated!");
                     return;
                     }
        }
     else
        switch (n) {
            case 1:             return XBlackPixelOfScreen(screen); break;
            case 2:             return XWhitePixelOfScreen(screen); break;
            case 3:             return XBlackPixelOfScreen(screen); break;
        }
}
```

❶ Define color values in the first of three color data structures.

❷ After defining RGB values, call the ALLOC COLOR routine. ALLOC COLOR allocates shared color cells on the default color map and returns a pixel value for the color that matches the specified color most closely.

5.4 Allocating Colors for Exclusive Use

If a client does not need to change color values, it should share colors by using the methods described in Section 5.3. Sharing colors saves resources. However, a client that changes color values must allocate them for its exclusive use.

Xlib provides two methods for allocating colors for a client's exclusive use. First, the client can allocate cells and store color values in the default color map. Second, if the default color map does not contain enough storage, or if the default color map is read-only (such as true color), the client can create its own color map using a writable visual type and store color values in it. In addition, when creating a color map, the client can allocate all entries in the color map for its exclusive use. Refer to the CREATE COLORMAP routine in Section 5.4.1 for more information about allocating all entries in a color map.

This section describes how to specify a color map, how to allocate cells for exclusive use, and how to store values in the color cells.

5.4.1 Specifying a Color Map

Clients can either use the default color map and allocate its color cells for exclusive use or create their own color maps.

If possible, use the default color map. Although a client can create color maps for its own use, the hardware color map storage is limited. When a client creates its own color map, the map must be installed into the hardware color map before the client map can be used. If the client color map is not installed, the client may use a different color map and possibly display the wrong color. Using the default color map eliminates this problem. See Section 5.1 for information about how Xlib handles color maps.

To specify the default color map, use the DEFAULT COLORMAP routine. DEFAULT COLORMAP returns the identifier of the default color map.

If the default color map does not contain enough resources, the client can create its own color map.

To create a color map, use the following method:

1. Using one of the methods described in Section 5.2, determine the visual type of a specified screen.
2. Call the CREATE COLORMAP routine.

The CREATE COLORMAP routine creates a color map for the specified window and visual type. Note that CREATE COLORMAP can only be used with pseudocolor, gray scale, and direct color visual types.

The CREATE COLORMAP routine has the following format:

```
XCreateColormap(display, window_id, visual_struc, alloc)
```

The **alloc** argument specifies whether the client creating the color map allocates all of the color map entries for its exclusive use or creates a color map with no defined color map entries. To allocate all entries for exclusive use, specify the constant **AllocAll**. To allocate no defined map entries, specify the constant **AllocNone**. The latter is useful when two or more clients are to share the newly created color map.

See Section 5.4.2 for information about allocating colors. See Example 5–4 for an example of creating a color map.

5.4.2 Allocating Color Cells

After specifying a color map, allocate color cells in it.

Use the ALLOC COLOR CELLS routine or ALLOC COLOR PLANES to allocate color resources. Either routine can be used; however, ALLOC COLOR CELLS allocates colors according to the pseudocolor model. The ALLOC COLOR PLANES routine allocates color resources according to a direct color model. See Section 5.2 for information about these color models.

Example 5–4 illustrates how to allocate colors for exclusive use. The program creates a color wheel that rotates when the user presses MB1.

Example 5–4 Allocating Colors for Exclusive Use

```
#include <decw$include/Xlib.h>
#include <decw$include/Xutil.h>
#include <stdio.h>
#include math;

#define winW 600
#define winH 600
#define backW 800
#define backH 800

Display *dpy;
Window win;
Pixmap pixmap;
Colormap map;
GC gc;
Screen *screen;
int scrNum;
XColor *colors;
int offsetX, offsetY;
int fullcount;
int ButtonIsDown = 0;
int n, exposeflag = 0;
int ihop=1;
int whiteValue;
XSetWindowAttributes xswa;
XVisualInfo *pVisualInfo;
static void doInitialize( );
static void doGetVisual();
static void doCreateWindows( );
static void doCreateGraphicsContext( );
static void doCreatePixmap( );
static void doCreateColor( );
static void doCreateWheel( );
static void doMapWindows( );
static void doHandleEvents( );
static void doExpose( );
static void doButtonPress( );
static void doButtonRelease( );
static void doChangeColors( );
static void doLoadColormap( );
static void doHLS_to_RGB( );
static void doConfigure( );
```

(continued on next page)

Example 5–4 (Cont.) Allocating Colors for Exclusive Use

```
/***** The main program *****/

static int main()
{
    doInitialize( );
    doHandleEvents( );
}

/***** doInitialize *****/
static void doInitialize( )
{
    dpy = XOpenDisplay(0);

    screen = DefaultScreenOfDisplay(dpy);   /* This is the screen structure */
    scrNum = DefaultScreen(dpy);     /* This is the screen index number*/

    doGetVisual();

    doCreateColor( );

    doCreateWindows( );

    doCreateGraphicsContext( );

    doCreatePixmap( );

    doCreateWheel( );

    doMapWindows( );
}
/***** doGetVisual *****/
static void doGetVisual( )
{
❶   XVisualInfo vInfoTemplate;
    int usableClasses[3] = {PseudoColor,DirectColor,GrayScale},i,nVis;

    vInfoTemplate.screen = scrNum;
    for (i = 0;  i < 3;  i++)
        {
        vInfoTemplate.class = usableClasses[i];
❷       pVisualInfo = XGetVisualInfo(dpy,VisualClassMask|VisualScreenMask,
                                     &vInfoTemplate,&nVis);
        if (pVisualInfo) break;
        }
    if (!pVisualInfo)
        {
        fprintf(stderr,"Unable to find a dynamic visual class");
        exit(1);
        }
}
/***** Create the windows *****/
static void doCreateWindows( )
{
    int winX = 100;
    int winY = 100;

    /* Create the win window */

    xswa.event_mask = ExposureMask | ButtonPressMask |
        ButtonReleaseMask | StructureNotifyMask;
    xswa.background_pixel = whiteValue;
    xswa.border_pixel = whiteValue; /* Note: you must set this for a non-
           default depth and class! */
```

(continued on next page)

Example 5–4 (Cont.) Allocating Colors for Exclusive Use

```
    win = XCreateWindow(dpy, RootWindowOfScreen(screen),
        winX, winY, winW, winH, 0,
        pVisualInfo->depth, InputOutput,pVisualInfo->visual,
        CWBorderPixel |CWEventMask | CWBackPixel | CWColormap, &xswa);

    /* Create the name of the window */
    XStoreName(dpy, win, "Color Wheel: Press MB1 to Rotate or MB2 to Exit.");
}

/***** Create the graphics context *****/
static void doCreateGraphicsContext( )
{
    gc = XCreateGC(dpy, win, 0, 0);
}

/***** Create the pixmap *****/
static void doCreatePixmap( )
{
❸   pixmap = XCreatePixmap(dpy, XRootWindow(dpy, XDefaultScreen(dpy)),
        backW, backH, pVisualInfo->depth);
    XSetForeground(dpy,gc,whiteValue);
    XFillRectangle(dpy, pixmap, gc, 0, 0, backW, backH);
}

/***** Create the color ******/

❹static void doCreateColor( )
{
int *pixels;
int contig;
int *plane_masks;

        if (pVisualInfo->visual != DefaultVisual(dpy,scrNum))
❺           map=XCreateColormap(dpy,RootWindow(dpy,scrNum),pVisualInfo->visual,
                AllocNone);
        else map = XDefaultColormapOfScreen(screen);


        xswa.colormap = map;
        fullcount = XDisplayCells(dpy, scrNum)/2;
        if (fullcount > 128) fullcount = 128;
        pixels = malloc(sizeof(int)*fullcount);
        colors = malloc(sizeof(XColor)*fullcount);

        /* Get a value for white (Use colors[0] temporarily) */

        colors[0].red = colors[0].blue = colors[0].green = 0xffff;
        XAllocColor(dpy,map,&colors[0]);
        whiteValue = colors[0].pixel;

        /* Now get writable pixels for the color wheel */

        if (!XAllocColorCells(dpy, map, contig, plane_masks,
            0, pixels, fullcount))
            {
                sys$exit(1);
            }
        doLoadColormap(pixels);
}
```

(continued on next page)

Example 5–4 (Cont.) Allocating Colors for Exclusive Use

```
/***** Create the wheel *****/

❻static void doCreateWheel( )
{
int pixel, i, j;
XPoint *pgon;
int xcent, ycent;

    /* Now set up wheel.  It is really a set of triangles*/
        pgon = malloc(sizeof(XPoint)*3*fullcount+1);
        xcent=backW/2;
        ycent=backH/2;
❼        pgon[0].x = backW;
        pgon[0].y = backH/2;

    /* Fill in coordinate for center point in all triangles */

        for (i=0;i<fullcount*3;i+=3)
            {
            pgon[i+1].x = xcent;
            pgon[i+1].y = ycent;
            }

     /* Calculate the triangle points on the outer circle */

         for (pixel=0,i=0;pixel<fullcount;i+=3, pixel++)
             {
             double x,y,xcent_f,ycent_f;
             xcent_f = (double)xcent;
             ycent_f = (double)ycent;
             x=cos( (((double)pixel+1.)/(double)fullcount)*2.*3.14159);
             y=sin( (((double)pixel+1.)/(double)fullcount)*2.*3.14159);
             pgon[i+2].x = (int)(x*xcent_f)+xcent;
             pgon[i+2].y = (int)(y*ycent_f)+ycent;
             pgon[i+3].x = pgon[i+2].x;
             pgon[i+3].y = pgon[i+2].y;
             XSetForeground(dpy, gc, colors[i/3].pixel);
             XFillPolygon(dpy, pixmap, gc, &pgon[i], 3, Convex, CoordModeOrigin);
             }
    offsetX = (backW - winW)/2;
    offsetY = (backH - winH)/2;
    return;
}

/***** Map the windows *****/
static void doMapWindows( )
{
    XMapWindow(dpy, win);
}
```

(continued on next page)

Example 5–4 (Cont.) Allocating Colors for Exclusive Use

```
/***** Handle the events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);

        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case ButtonPress:           doButtonPress(&event); break;
            case ButtonRelease:         doButtonRelease(&event); break;
            case ConfigureNotify:       doConfigure(&event); break;
        }
    }
}

/***** Handle window exposures *****/
static void doExpose(eventP)
XEvent *eventP;
{

❽  XCopyArea(dpy, pixmap, win, gc, offsetX + eventP->xexpose.x,
        offsetY + eventP->xexpose.y, eventP->xexpose.width,
        eventP->xexpose.height, eventP->xexpose.x, eventP->xexpose.y);
}

/***** Button Press ******/
static void doButtonPress(eventP)
XEvent *eventP;
{
       if (eventP ->xbutton.button == Button2) {
           sys$exit (1);
       }
       ButtonIsDown = 1;
       if (ButtonIsDown) doChangeColors( );
       return;
}

/***** Button Release *****/
static void doButtonRelease(eventP)  /* Quit rotate when MB1 released */
XEvent *eventP;
{
       ButtonIsDown = 0;
       return;
}

/***** Configure notify *****/
static void doConfigure(eventP)
XEvent *eventP;
{
❾  offsetX = (backW - eventP->xconfigure.width)/2;
    offsetY = (backH - eventP->xconfigure.height)/2;
}
```

(continued on next page)

Example 5–4 (Cont.) Allocating Colors for Exclusive Use

```
/***** Change the colors *****/
⓾static void doChangeColors( )
{
      for (;!(XPending(dpy));){
                unsigned int i,temp;
                double h,r,g,b;

                temp = colors[0].pixel;
                for (i=0;i<fullcount-1;i++)
                      colors[i].pixel = colors[i+1].pixel;
                colors[fullcount-1].pixel = temp;
                XStoreColors(dpy, map, colors, fullcount);
      }
}

/***** Load the colormap *****/
⓫static void doLoadColormap(pPixels)
int *pPixels;
{

    unsigned int i,j;
    double h,r,g,b;

        for (i=0;i < fullcount;i++) {
            colors[i].pixel=pPixels[i];
            colors[i].flags = DoRed | DoGreen |DoBlue;
        }
        for (i=0; i < fullcount ; i++) {
⓬          h = (double)i*360/((double)fullcount+1);
            doHLS_to_RGB(&h,&.5,&.5,&r,&g,&b);
            colors[i].red = r * 65535.0;
            colors[i].green = g * 65535.0;
            colors[i].blue = b * 65535.0;
        }
      XStoreColors(dpy, map, colors, fullcount);
}

/***** Convert to RGB *****/
static void doHLS_to_RGB (h,l,s, r,g,b)
double *h,*l,*s,*r,*g,*b;
{
     double m1,m2;
     double value();

     m2 = (*l < 0.5) ? (*l)*(1+*s) : *l + *s- (*l)*(*s) ;
     m1 = 2*(*l) - m2;
        if ( *s == 0 )
          { (*r)=(*g)=(*b)=(*l); }      /*Gray shade*/
        else
          { *r=value(m1,m2,(double)(*h+120.));
            *g=value(m1,m2,(double)(*h+000.));
            *b=value(m1,m2,(double)(*h-120.));
          }
        return;
}
```

(continued on next page)

Example 5–4 (Cont.) Allocating Colors for Exclusive Use

```
double value (n1,n2,hue)
double n1,n2,hue;
{
        double val;

        if (hue>360.)  hue -= 360.;
        if (hue<0.  )  hue += 360.;

        if (hue<60)
          val = n1+(n2-n1)*hue/60.;
        else if (hue<180.)
          val = n2;
        else if (hue<240.)
          val = n1+(n2-n1)*(240.-hue)/60.;
        else
          val = n1;
        return (val);
 }
```

❶ The client allocates storage for a visual info data structure and creates a template consisting of the screen index number and one of the dynamic visual classes specified in the array *usableClasses*.

❷ The GET VISUAL INFO routine checks each visual on the system for a match of the visual attributes specified in **vInfoTemplate**. Each attribute corresponds to a bit set in the argument **vinfo_mask**. If a match occurs, the client continues. If a suitable visual is not found, the client exits.

The GET VISUAL INFO routine has the following format:

```
XGetVisualInfo(display, vinfo_mask, vinfo_template,
      num_items_return)
```

❸ The client uses a pixmap as a backing store for the color wheel. When a user reconfigures the color wheel window, the client copies the color wheel from the pixmap into the resized window. For information about creating and using pixmaps, see Chapter 7.

❹ After creating the pixmap for backing store, the client creates colors for the wheel and the wheel itself. The client-defined *doCreateColor* routine allocates color cells for the exclusive use of the client and stores initial color values in the color map.

❺ The client compares the default visual id with the visual id in the visual info data structure. If they are equal, the client allocates colors from the default color map. Otherwise, the client creates a color map using the visual information in the visual info data structure. In each case, the client specifies that only 128 color cells be allocated. After allocating color cells, the client calls the client-defined *doLoadColormap* routine to define color values. For a description of the routine, see callouts 7, 8, 9, and 10.

❻ The client-defined *doCreateWheel* routine defines the wheel used to display colors and specifies initial color values.

❼ The wheel is composed of polygons. Each polygon is defined by three points, one in the center of the wheel and two at the circumference. After the initial polygon is specified, each polygon shares one point with the polygon previously defined, as Figure 5–4 illustrates.

To define each point, the client uses a point data structure, which is described in Chapter 6. After defining a polygon, the client fills it with a specified foreground color.

❽ When the user reconfigures the window, the server generates an expose event. In response to the event, the client copies the pixmap into the exposed area, which is calculated using the offset from the original to the new position of the window. For information about handling exposure events, see Chapter 9.

❾ The client calculates the offset from the original window position in response to a configure notify event. The server issues a configure notify event each time the user resizes the color wheel window. For information about handling configure notify events, see Chapter 9.

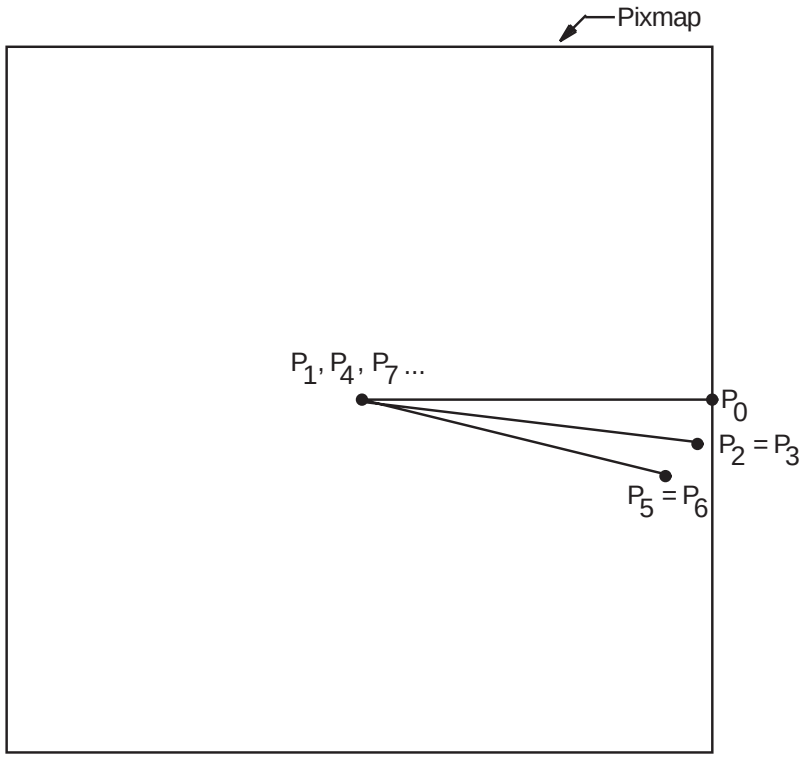
❿ The rotation of the color wheel is accomplished by changing values in the color map. As long as there are no pending events and the user is pressing MB1, the client-defined *doChangeColors* routine shifts color values by one.

⓫ The *doLoadColormap* routine initializes the color wheel by defining 128 colors and storing them in the color map.

⓬ Colors are defined initially using the Hue, Light, Saturation (HLS) system. The values of color hues vary, while values for light and saturation remain constant. After a color has been defined using HLS, the color is converted into RGB values by the client-defined *doHLS_to_RGB* routine. When all colors are defined, the client stores them in the color map by calling the STORE COLORS routine.

Figure 5–4 illustrates how the color wheel in Example 5–4 is composed of a set of polygons.

Figure 5–4 Polygons That Define the Color Wheel



ZK−0518A−GE

When allocating colors from any shared color map, the client may exhaust the resources of the color map. In this case, Xlib provides a routine for copying the default color map entries into a new client-created color map.

To create a new color map when the client exhausts the resources of a previously shared color map, use the COPY COLORMAP AND FREE routine. The routine creates a color map of the same visual type and for the same screen as the previously shared color map. The previously shared color map can be either the default color map or a client-created color map. The COPY COLORMAP AND FREE routine has the following format:

```
XCopyColormapAndFree(display, colormap_id)
```

COPY COLORMAP AND FREE copies all cells allocated by the client from the previously shared color map to the new color map, keeping color values intact. The new color map is created with the same value of the argument **alloc** as the previously shared color map and has the following effect on the new color map entries.

Value of Alloc On Old Color Map	Effect
AllocAll	All entries are copied from the old color map and are then freed
AllocNone	The entries moved are all pixels and planes that have been allocated by the client using the following routines and that have not been freed since they were allocated: ALLOC COLOR, ALLOC NAMED COLOR, ALLOC COLOR CELLS, ALLOC COLOR PLANES

5.4.3 Storing Color Values

After allocating color entries in the color map, store RGB values in the color map cells using the following method:

1. Assign color values to the color data structure and set the flags member to indicate the components to be changed. Normally, all flags should be set.
2. Call the STORE COLOR routine to store one color, the STORE COLORS routine to store more than one color, and the STORE NAMED COLOR routine to store a named color.

The STORE COLOR routine has the following format:

XStoreColor(display, colormap_id, screen_def_return)

The STORE COLORS routine has the following format:

XStoreColors(display, colormap_id, screen_defs_return, num_colors)

The STORE NAMED COLOR routine has the following format:

XStoreNamedColor(display, colormap_id, color_name, pixel, flags)

Refer to Example 5–4 for an example of using the STORE COLORS routine.

5.5 Freeing Color Resources

To free storage allocated for client colors, call the FREE COLORS routine. FREE COLORS releases all storage allocated by the following color routines: ALLOC COLOR, ALLOC COLOR CELLS, ALLOC NAMED COLORS, and ALLOC COLOR PLANES.

To delete the association between the color map ID and the color map, use the FREE COLORMAP routine. FREE COLORMAP has no effect on the default color map of the screen. If the color map is an installed color map, FREE COLORMAP removes it.

5.6 Querying Color Map Entries

Xlib provides routines to return the RGB values of both the color map index and a named color.

To query the RGB values of a specified pixel in the color map, use the QUERY COLOR routine. The pixel value to look up is specified in the pixel member of the color data structure. The RGB components of the color value are returned in the red, green, and blue members of the data structure.

To query the RGB values of an array of pixel values, use the QUERY COLORS routine. The values returned are the values passed in the pixel member of the color data structure. Note that if the color map entry being queried is undefined, the value returned by QUERY COLOR will not necessarily correspond to the color displayed on the screen.

To look up the values associated with a named color, use the LOOKUP COLOR routine. LOOKUP COLOR uses the specified color map to find out the values with respect to a specific screen. It returns both the exact RGB values and the closest RGB values supported by hardware.

6
Drawing Graphics

Xlib provides clients with routines that draw graphics into windows and pixmaps. This chapter describes how to create and manage graphics drawn into windows, including the following topics:

- Drawing points, lines, rectangles, and arcs
- Filling rectangles, polygons, and arcs
- Copying graphics
- Limiting graphics to a region of a window or pixmap
- Clearing graphics from a window
- Creating cursors

Chapter 7 describes drawing graphics into pixmaps.

6.1 Graphics Coordinates

Xlib graphics coordinates define the position of graphics drawn in a window or pixmap. Coordinates are either relative to the origin of the window or pixmap in which the graphics object is drawn or relative to a previously drawn graphics object.

Xlib graphics coordinates are similar to the coordinates that define window position. Xlib measures length along the x-axis from the origin to the right. Xlib measures length along the y-axis from the origin down. Xlib specifies coordinates in units of pixels.

6.2 Using Graphics Routines Efficiently

If clients use the same drawable and graphics context for each call, Xlib handles back-to-back calls of DRAW POINT, DRAW LINE, DRAW SEGMENT, DRAW RECTANGLE, FILL ARC, and FILL RECTANGLE in a batch. Batching increases efficiency by reducing the number of requests to the server.

When drawing more than a single point, line, rectangle, or arc, clients can also increase efficiency by using routines that draw or fill multiple graphics (DRAW POINTS, DRAW LINES, DRAW SEGMENTS, DRAW RECTANGLES, DRAW ARCS, FILL ARCS, and FILL RECTANGLES). Clipping negatively affects efficiency. Consequently, clients should ensure that graphics they draw to a window or pixmap are within the boundary of the drawable. Drawing outside the window or pixmap decreases performance. Clients should also ensure that windows into which they are drawing graphics are not occluded.

The most efficient method for clearing multiple areas is using the FILL RECTANGLES routine. By using the FILL RECTANGLES routine, clients can increase server performance. For information about using FILL RECTANGLES to clear areas, see Section 6.6.1.

6.3 Drawing Points and Lines

Xlib includes routines that draw points and lines. When clients draw more than one point or line, performance is affected. Performance is most efficient if clients use Xlib routines that draw multiple points or lines rather than calling single point and line-drawing routines many times.

This section describes using routines that draw both single and multiple points and lines.

6.3.1 Drawing Points

To draw a single point, use the DRAW POINT routine, specifying x-axis and y-axis coordinates, as in the following:

```
                .
                .
                .
int x,y=100;
XDrawPoint(display, window, gc, x, y);
```

If drawing more than one point, use the following method:

1. Define an array of point data structures.
2. Call the DRAW POINTS routine, specifying the array that defines the points, the number of points the server is to draw, and the coordinate system the server is to use. The server draws the points in the order specified by the array.

Xlib includes the point data structure to enable clients to define an array of points easily. The following illustrates the data structure:

```
typedef struct {
    short x, y;
} XPoint;
```

Table 6–1 describes the members of the point data structure.

Table 6–1 Point Data Structure Members

Member Name	Contents
x	Defines the x value of the coordinate of a point
y	Defines the y value of the coordinate of a point

The server determines the location of points according to the following:

- If the client specifies the constant **CoordModeOrigin**, the server defines all points in the array relative to the origin of the drawable.
- If the client specifies the constant **CoordModePrevious**, the server defines the coordinates of the first point in the array relative to the origin of the drawable and the coordinates of each subsequent point relative to the point preceding it in the array.

The server refers to the following members of the GC data structure to define the characteristics of points it draws:

Function	Plane mask
Foreground	Subwindow mode
Clip x origin	Clip y origin
Clip mask	

Chapter 4 describes GC data structure members.

Example 6–1 uses the DRAW POINTS routine to draw a circle of points each time the user clicks MB1.

Figure 6–1 illustrates sample output from the program.

Example 6–1 Drawing Multiple Points

```
                    .
                    .
                    .
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case ButtonPress:           doButtonPress(&event); break;
        }
    }
}

/***** Write a message *****/
❶static void doExpose(eventP)
XEvent *eventP;
{
    char message1[ ] = {"To create points, click MB1"};
    char message2[ ] = {"Each click creates a new circle of points"};
    char message3[ ] = {"To exit, click MB2"};

    XDrawImageString(dpy, win, gc, 150, 25, message1, strlen(message1));
    XDrawImageString(dpy, win, gc, 150, 50, message2, strlen(message2));
    XDrawImageString(dpy, win, gc, 150, 75, message3, strlen(message3));
}

/***** Draw the points *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
#define POINT_CNT 100
#define RADIUS 50
    XPoint point_arr[POINT_CNT];
    int i;
```

(continued on next page)

Example 6–1 (Cont.) Drawing Multiple Points

```
❷ int x = eventP->xbutton.x;
  int y = eventP->xbutton.y;

  if (eventP->xbutton.button == Button2) sys$exit (1);

  for (i=0;i<POINT_CNT;i++) {
      point_arr[i].x = x + RADIUS*cos(i);
      point_arr[i].y = y + RADIUS*sin(i);
  }

❸ XDrawPoints(dpy, win, gc, &point_arr, POINT_CNT, CoordModeOrigin);
}
```

❶ When the client receives notification that the server has mapped the window, the *doExpose* routine writes three messages into the window. For information about using the DRAW IMAGE STRING routine, see Chapter 8.

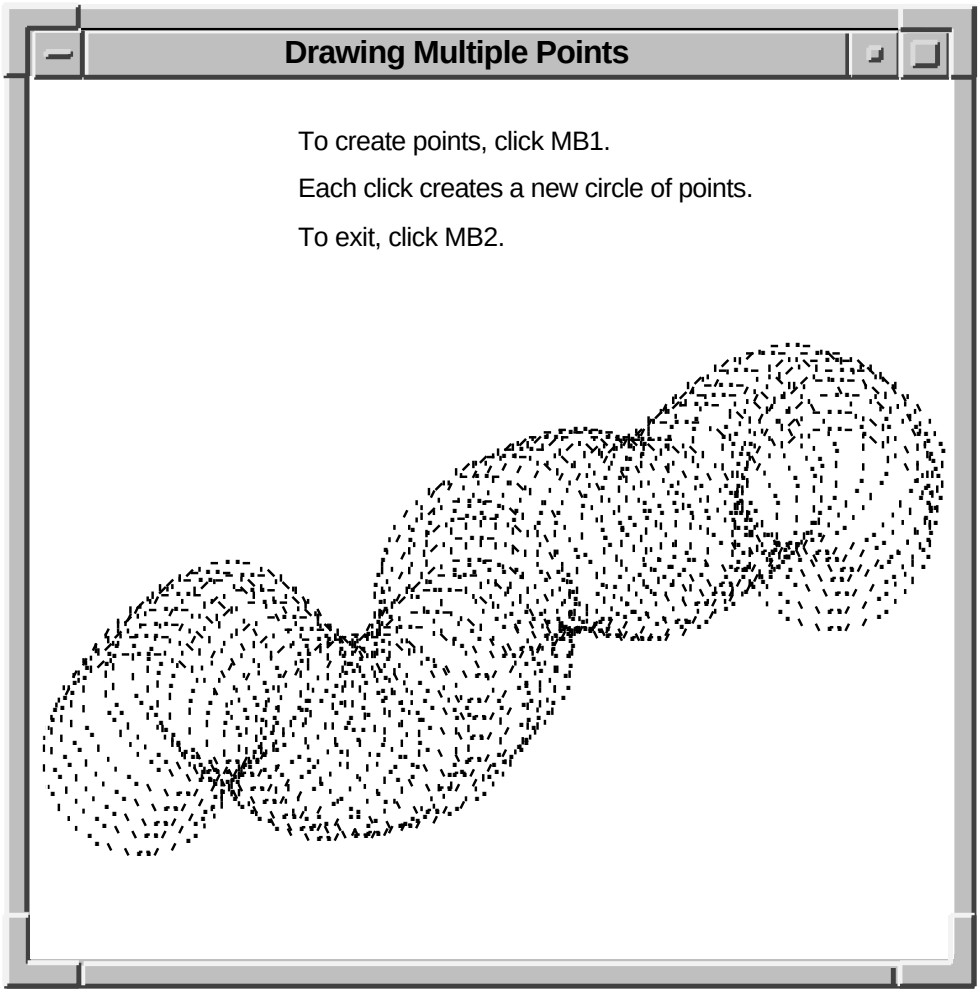
❷ If the user clicks any mouse button, the client initiates the *doButtonPress* routine. If the user clicks MB1, the client draws 50 points. If the user clicks MB2, the client exits the system. The client determines which button the user pressed by referring to the button member of the button event data structure. For more information about the button event data structure, see Chapter 9.

❸ The DRAW POINTS routine has the following format:

XDrawPoints(display, drawable_id, gc_id, points, num_points,
point_mode)

The **point_mode** argument specifies whether coordinates are relative to the origin of the drawable or to the previous point in the array.

Figure 6–1 Circles of Points Created Using the DRAW POINTS Routine



ZK–2571A–GE

6.3.2 Drawing Lines and Line Segments

Xlib includes routines that draw single lines, multiple lines, and line segments. To draw a single line, use the DRAW LINE routine, specifying beginning and ending points, as in the following:

```
                .
                .
                .
int x1,y1=100;
int x2,y2=200;
XDrawLine(display, window, gc, x1, y1, x2, y2);
```

To draw multiple lines, use the following method:

1. Define an array of points using the point data structure described in Section 6.3.1 to specify beginning and ending line points. The server interprets pairs of array elements as beginning and ending points. For example, if the array that defines the beginning point is $point[i]$, the server reads $point[i+1]$ as the corresponding ending point.

2. Call the DRAW LINES routine, specifying the following:
   - The array that defines the points.
   - The number of points that define the line.
   - The coordinate system the server uses to locate the points. The server draws the lines in the order specified by the array.

Clients can specify either the **CoordModeOrigin** or the **CoordModePrevious** constant to indicate how the server determines the location of beginning and ending points. The server uses the methods described in Section 6.3.1.

The server draws lines in the order the client has defined them in the point data structure. Lines join correctly at all intermediate points. If the first and last points coincide, the first and last line also join correctly. For any given line, the server draws pixels only once. The server draws intersecting pixels multiple times if zero-width lines intersect; it draws intersecting pixels of wider lines only once.

Example 6–2 uses the DRAW LINES routine to draw a star when the server notifies the client that the window is mapped.

Example 6–2 Drawing Multiple Lines

```
                    .
                    .
                    .
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
        }
    }
}

/***** Expose event *****/
static void doExpose(eventP)
XEvent *eventP;
{
    XPoint pt_arr[6];

❶   pt_arr[0].x = 75;
    pt_arr[0].y = 500;
    pt_arr[1].x = 300;
    pt_arr[1].y = 100;
    pt_arr[2].x = 525;
    pt_arr[2].y = 500;
    pt_arr[3].x = 50;
    pt_arr[3].y = 225;
    pt_arr[4].x = 575;
    pt_arr[4].y = 225;
    pt_arr[5].x = 75;
    pt_arr[5].y = 500;
```

(continued on next page)

Example 6–2 (Cont.) Drawing Multiple Lines

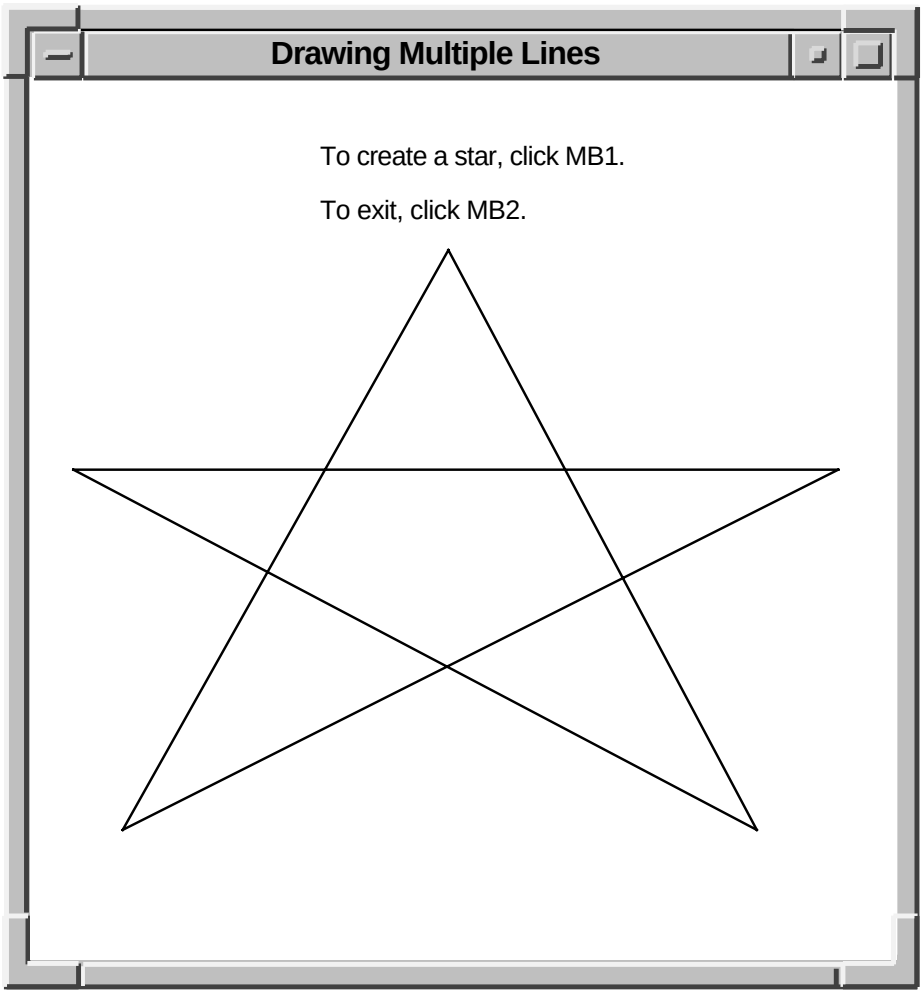
```
❷   XDrawLines(dpy, win, gc, &pt_arr, 6, CoordModeOrigin);
}
                    .
                    .
                    .
```

❶ The *doExpose* routine uses point data structures to define beginning and ending points of lines.

❷ The call to draw lines refers to a graphics context (*gc*), which the client has previously defined, and an array of point data structures. The constant **CoordModeOrigin** indicates that all points are relative to the origin of *win* (100,100).

Figure 6–2 illustrates the resulting output.

Figure 6–2 Star Created Using the DRAW LINES Routine



ZK–2512A–GE

Use the DRAW SEGMENTS routine to draw multiple, unconnected lines, defining an array of segments in the segment data structure. The following illustrates the data structure:

```
typedef struct {
    short x1, y1, x2, y2;
} XSegment;
```

Table 6–2 describes the members of the data structure.

Table 6–2 Segment Data Structure Members

Member Name	Contents
x1	The x value of the coordinate that specifies one endpoint of the segment
y1	The y value of the coordinate that specifies one endpoint of the segment
x2	The x value of the coordinate that specifies the other endpoint of the segment
y2	The y value of the coordinate that specifies the other endpoint of the segment

The DRAW SEGMENTS routine functions like the DRAW LINES routine, except the routine does not use the coordinate mode.

The DRAW LINE and DRAW SEGMENTS routines refer to all but the join style, fill rule, arc mode, and font members of the GC data structure to define the characteristics of lines. The DRAW LINES routine refers to all but the fill rule, arc mode, and font members of the data structure.

Chapter 4 describes the GC data structure.

6.4 Drawing Rectangles and Arcs

As with routines that draw points and lines, Xlib provides clients the choice of drawing either single or multiple rectangles and arcs. If a client is drawing more than one rectangle or arc, use the multiple-drawing routines for most efficiency.

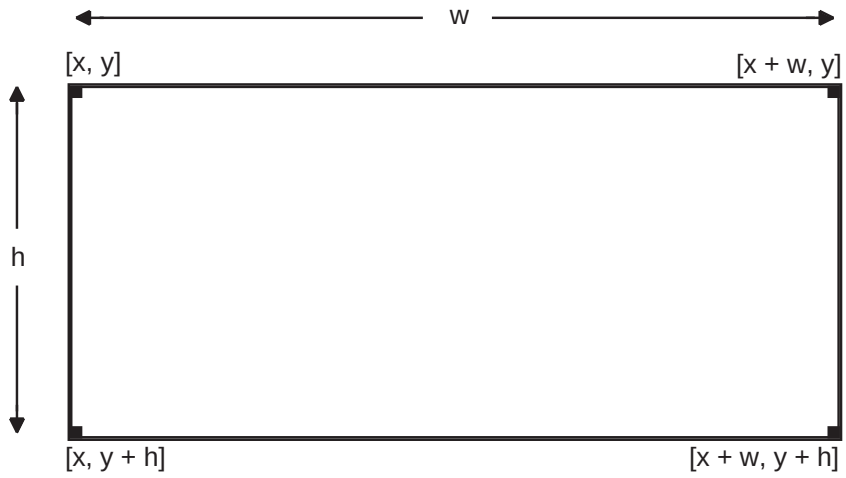
6.4.1 Drawing Rectangles

To draw a single rectangle, use the DRAW RECTANGLE routine, specifying the coordinates of the upper left corner and the dimensions of the rectangle, as in the following:

```
int x=50
int y=100;
int width=25;
int length=50;
                    .
                    .
                    .
XDrawRectangle(display, window, gc, x, y, width, length);
```

Figure 6–3 illustrates how Xlib interprets coordinate and dimension parameters. The x- and y-coordinates are relative to the origin of the drawable.

Figure 6–3 Rectangle Coordinates and Dimensions

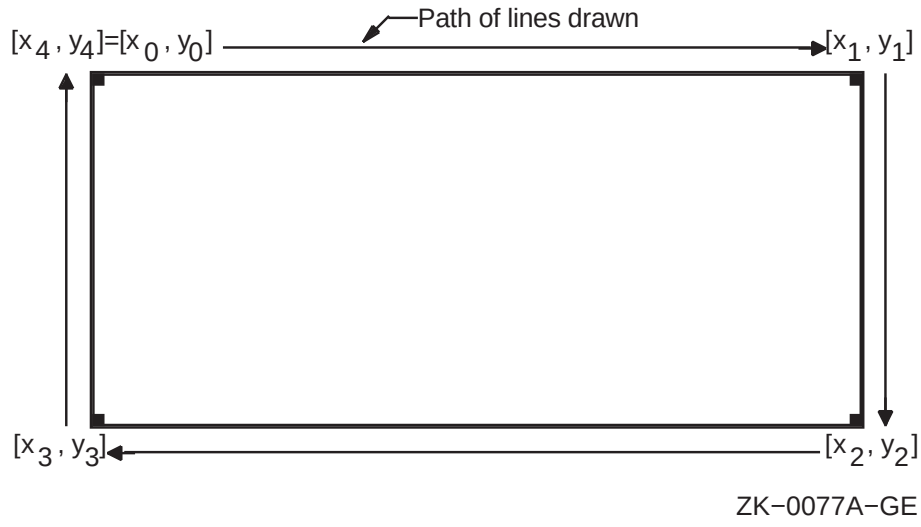


To draw multiple rectangles, use the following method:

1. Define an array of rectangles using the rectangle data structure.
2. Call the DRAW RECTANGLES routine, specifying the array that defines rectangle origin, width, and height, and the number of array elements.

The server draws each rectangle as shown in Figure 6–4.

Figure 6–4 Rectangle Drawing



For a specified rectangle, the server draws each pixel only once. If rectangles intersect, the server draws intersecting pixels multiple times.

Xlib includes the rectangle data structure to enable clients to define an array of rectangles easily. The following illustrates the data structure:

```
typedef struct {
    short x, y;
    unsigned short width, height;
} XRectangle;
```

Table 6–3 describes the members of the rectangle data structure.

Table 6–3 Rectangle Data Structure Members

Member Name	Contents
x	Defines the x value of the rectangle origin
y	Defines the y value of the rectangle origin
width	Defines the width of the rectangle
height	Defines the height of the rectangle

When drawing either single or multiple rectangles, the server refers to the following members of the GC data structure to define rectangle characteristics:

Function	Plane mask
Foreground	Background
Line width	Line style
Join style	Fill style
Tile	Stipple
Tile/stipple x origin	Tile/stipple y origin
Subwindow mode	Clip x origin
Clip y origin	Clip mask
Dash offset	Dashes

Chapter 4 describes the GC data structure members.

Example 6–3 illustrates using the DRAW RECTANGLES routine. Figure 6–5 shows the resulting output.

Example 6–3 Drawing Multiple Rectangles

```
                        .
                        .
                        .
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case ButtonPress:           doButtonPress(&event); break;
        }
    }
}
```

(continued on next page)

Example 6–3 (Cont.) Drawing Multiple Rectangles

```
/***** Write a message *****/
❶static void doExpose(eventP)
XEvent *eventP;
{
    char message1 [ ] = {"To draw multiple rectangles, click MB1"};
    char message2 [ ] = {"To exit, click MB2"};

    XDrawImageString(dpy, win, gc, 150, 25, message1, strlen(message1));
    XDrawImageString(dpy, win, gc, 150, 50, message2, strlen(message2));
}

/***** Draw the rectangles  *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
#define REC_CNT 40
#define STEP 15
    XRectangle rec_arr[REC_CNT];
    int i;
❷   if (eventP->xbutton.button == Button2) sys$exit (1);

    for (i=0;i<REC_CNT;i++) {
        rec_arr[i].x = STEP * i;
        rec_arr[i].y = STEP * i;
        rec_arr[i].width = STEP*2;
        rec_arr[i].height = STEP*3;
    }

❸    XDrawRectangles(dpy, win, gc, &rec_arr, REC_CNT);
}
```

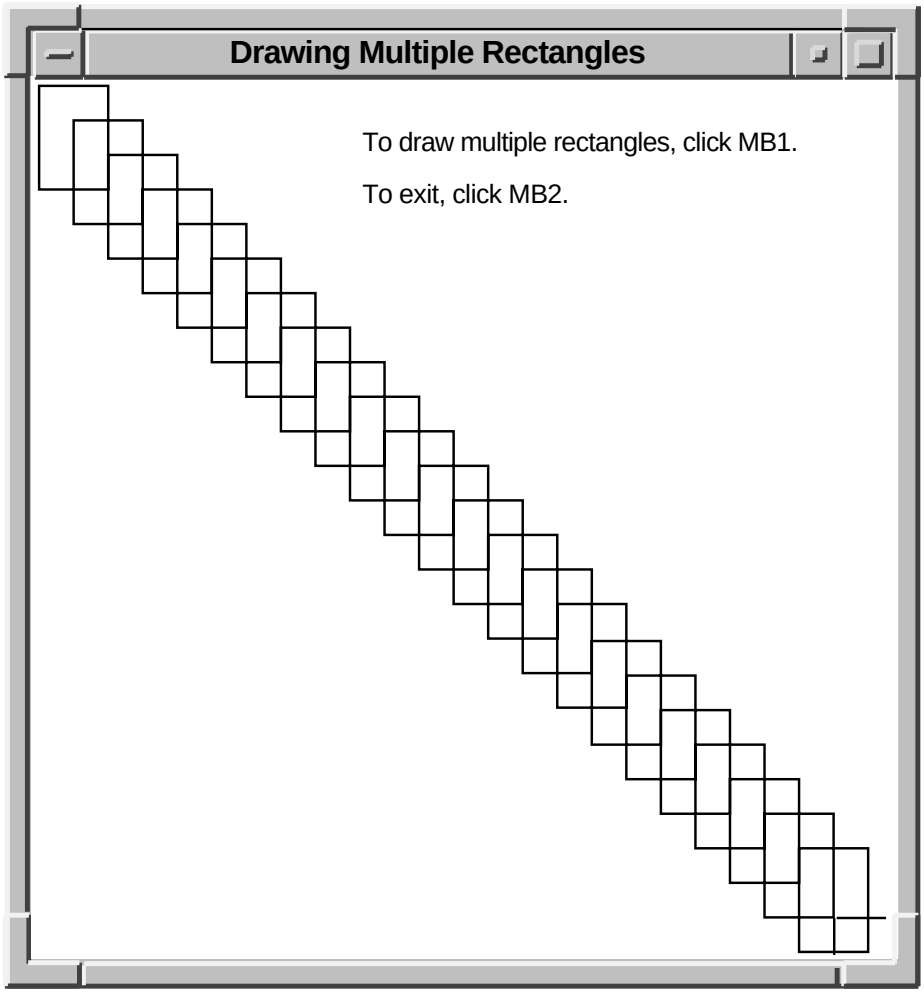
❶ When the client receives notification that the server has mapped the window, the *doExpose* routine writes two messages into the window. For information about using the DRAW IMAGE STRING routine, see Chapter 8.

❷ If the user clicks any mouse button, the client calls the *doButtonPress* routine. If the user clicks MB1, the client draws rectangles defined in the initialization loop. If the user clicks MB2, the client exits the system. The client determines which button the user has clicked by referring to the button member of the button event data structure. For more information about the button event data structure, see Chapter 9.

❸ The DRAW RECTANGLE routine has the following format:

XDrawRectangles(display, drawable_id, gc_id, rectangles,
num_rectangles)

Figure 6–5 Rectangles Drawn Using the DRAW RECTANGLES Routine



ZK–2510A–GE

6.4.2 Drawing Arcs

Xlib routines enable clients to draw either single or multiple arcs. To draw a single arc, use the DRAW ARC routine, specifying a rectangle that defines the boundaries of the arc and two angles that determine the start and extent of the arc, as in the following:

```
int x=50
int y=100;
int width=25;
int length=50;
int angle1=5760;
int angle2=5760;
                    .
                    .
                    .
XDrawArc(display, window, gc, x, y, width, height,
     angle1, angle2);
```

The server draws an arc within a rectangle. The client specifies the upper left corner of the rectangle, relative to the origin of the drawable. The center of the rectangle is the center of the arc. The width and height of the rectangle are the major and minor axes of the arc, respectively.

Two angles specify the start and extent of the arc. The angles are signed integers in degrees scaled up by 64. For example, a client would specify a 90-degree arc as 64 * 90 or 5760. The start of the arc is specified by the first angle, relative to the three o'clock position from the center of the rectangle. The extent of the arc is specified by the second angle, relative to the start of the arc. Positive integers indicate counterclockwise motion; negative integers indicate clockwise motion.

To draw multiple arcs, use the following method:

1. Define an array of arc data structures.
2. Call the DRAW ARCS routine, specifying the array that defines the arcs and the number of array elements.

The following illustrates the arc data structure:

```
typedef struct {
    short x, y;
    unsigned short width, height;
    short angle1, angle2;
} XArc;
```

Table 6–4 describes the members of the arc data structure.

Table 6–4 Arc Data Structure Members

Member Name	Contents
x	Defines the x-coordinate value of the rectangle in which the server draws the arc
y	Defines the y-coordinate value of the rectangle in which the server draws the arc
width	Defines the major axis of the arc
height	Defines the minor axis of the arc
angle1	Defines the starting point of the arc relative to the 3-o'clock position from the center of the rectangle
angle2	Defines the extent of the arc relative to the starting point

When drawing either single or multiple arcs, the server refers to the following members of the GC data structure to define arc characteristics:

Function	Plane mask
Foreground	Background
Line width	Line style
Join style	Cap style
Fill style	Tile
Tile/stipple x origin	Tile/stipple y origin
Clip x origin	Clip y origin
Clip mask	Dash offset
Dashes	Stipple
Subwindow mode	

Chapter 4 describes the GC data structure members.

If the last point in one arc coincides with the first point in the following arc, the two arcs join. If the first point in the first arc coincides with the last point in the last arc, the two arcs join.

If two arcs join, the line width is greater than zero, and the arcs intersect, the server draws all pixels only once. Otherwise, it may draw intersecting pixels multiple times.

Example 6–4 illustrates using the DRAW ARCS routine.

Example 6–4 Drawing Multiple Arcs

```
                        .
                        .
                        .
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case ButtonPress:           doButtonPress(&event); break;
        }
    }
}

/***** Write a message *****/
static void doExpose(eventP)
XEvent *eventP;
{
    char message1[ ] = {"To create arcs, click MB1"};
    char message2[ ] = {"Each click creates a new circle of arcs."};
    char message3[ ] = {"To exit, click MB2"};

    XDrawImageString(dpy, win, gc, 150, 25, message1, strlen(message1));
    XDrawImageString(dpy, win, gc, 150, 50, message2, strlen(message2));
    XDrawImageString(dpy, win, gc, 150, 75, message3, strlen(message3));
}

/***** Draw the arcs *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
#define ARC_CNT 16
#define RADIUS 50
#define INNER_RADIUS  20
    XArc arc_arr[ARC_CNT];
    int i;
❶   int x = eventP->xbutton.x;
    int y = eventP->xbutton.y;

    if (eventP->xbutton.button == Button2) sys$exit (1);
```

(continued on next page)

Example 6–4 (Cont.) Drawing Multiple Arcs

```
    for (i=0;i<ARC_CNT;i++) {
        arc_arr[i].angle1 = (64*360)/ARC_CNT * i;
        arc_arr[i].angle2 = (64*360)/ARC_CNT*3;
        arc_arr[i].width = RADIUS*2;
        arc_arr[i].height = RADIUS*2;
        arc_arr[i].x = x - RADIUS + sin(2*3.14159/ARC_CNT*i) * INNER_RADIUS;
        arc_arr[i].y = y - RADIUS + cos(2*3.14159/ARC_CNT*i) * INNER_RADIUS;
    }
❷   XDrawArcs(dpy, win, gc, &arc_arr, ARC_CNT);
}
```

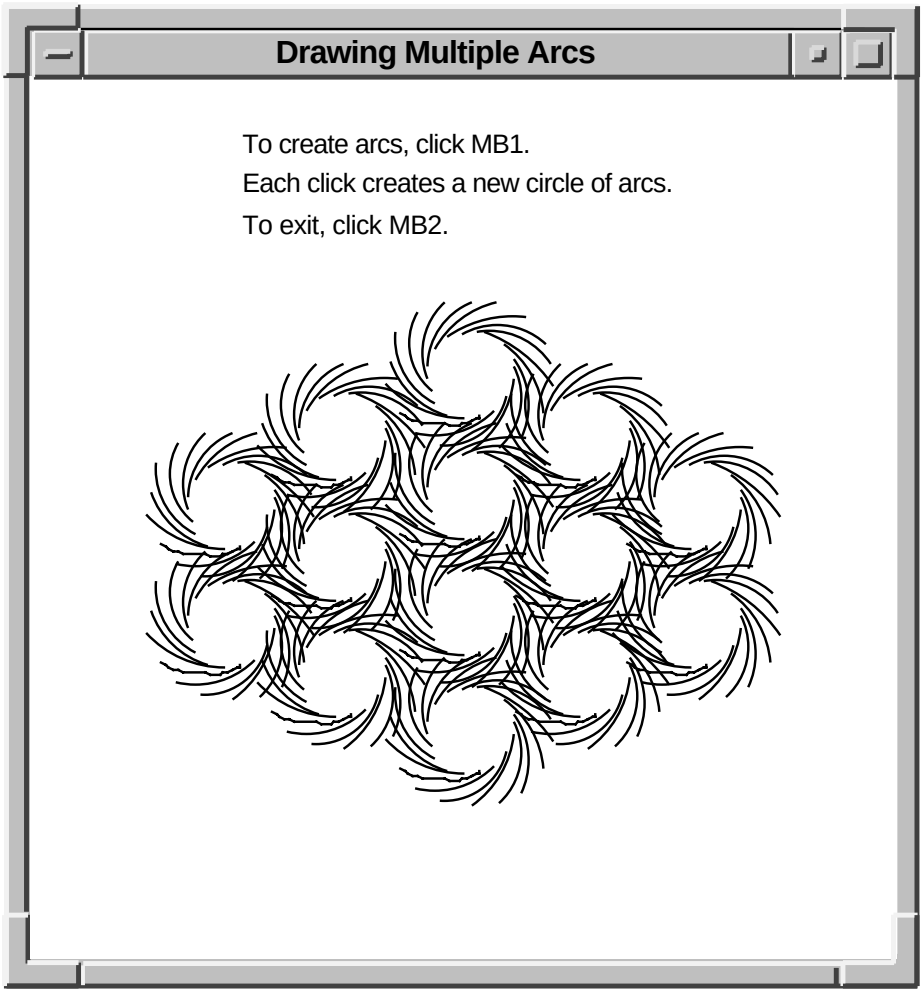
❶ The *x* and *y* variables specify the upper left corner of the rectangle that defines the boundary of the arc. The client determines the rectangle coordinates by taking the values of the **x** and **y** arguments from the button event data structure. Because these values indicate the position of the cursor when the user clicks the mouse button, the server draws the arcs relative to the position of the cursor. For more information about the button event data structure, see Chapter 9.

❷ The DRAW ARCS routine has the following format:

XDrawArcs(display,drawable_id,gc_id,arcs,num_arcs)

Figure 6–6 illustrates the resulting output.

Figure 6–6 Multiple Arcs Drawn Using the DRAW ARCS Routine



ZK–2568A–GE

6.5 Filling Areas

This section describes using Xlib routines to fill single rectangles, arcs, and polygons, and multiple rectangles and arcs.

6.5.1 Filling Rectangles and Arcs

The FILL RECTANGLE, FILL RECTANGLES, FILL ARC, and FILL ARCS routines create single and multiple rectangles or arcs and fill them using the fill style that the client specifies in a graphics context data structure.

The method of calling the fill routines is identical to that for drawing rectangles and arcs. For example, to create rectangles filled solidly with foreground color in Example 6–3, the client needs only to call the FILL RECTANGLES routine instead of DRAW RECTANGLES. The default value of the GC data structure fill style member is solid. If the client were to specify a tile or stipple for filling the rectangles, the client would have to change the graphics context used by the FILL RECTANGLES routine.

The server refers to the following members of the GC data structure to define characteristics of the rectangles and arcs it fills:

Function	Plane mask
Foreground	Background
Fill style	Tile
Stipple	Subwindow mode
Tile/stipple x origin	Tile/stipple y origin
Clip x origin	Clip y origin
Clip mask	

Additionally, the server refers to the arc mode member if filling arcs.

For information about using graphics context, see Chapter 4.

6.5.2 Filling a Polygon

To fill a polygon, use the following method:

1. Define an array of point data structures.
2. Call the FILL POLYGON routine, specifying the array that defines the points of the polygon, the number of points the server is to draw, the shape of the polygon, and the coordinate system the server is to use. The server draws the points in the order specified by the array.

See Section 6.3.1 for an illustration of the point data structure.

To improve performance, clients can specify whether the shape of the polygon is complex, convex, or nonconvex, as follows:

- Specify the constant **Complex** as the **shape** argument if the path that draws the polygon may intersect itself.
- Specify the constant **Convex** as the **shape** argument if the path that draws the shape is wholly convex. If a client specifies **Convex** for a path that is not convex, the results are undefined.
- Specify the constant **Nonconvex** as the **shape** argument if the path does not intersect itself, but the shape is not wholly convex. If a client specifies **Nonconvex** for a path that intersects itself, the results are undefined.

When filling the polygon, the server draws each pixel only once.

The server determines the location of points as follows:

- If the client specifies the constant **CoordModeOrigin**, the server defines all points in the array relative to the origin of the drawable.
- If the client specifies the constant **CoordModePrevious**, the server defines the coordinates of the first point in the array relative to the origin of the drawable, and the coordinates of each subsequent point relative to the point preceding it in the array.

If the last point does not coincide with the first point, the server closes the polygon automatically.

The server refers to the following members of the GC data structure to define the characteristics of the polygon it fills:

Function	Plane mask
Foreground	Fill style
Fill rule (if polygon is complex)	Tile
Tile/stipple x origin	Tile/stipple y origin
Clip x origin	Clip y origin
Subwindow mode	Clip mask
Stipple	Background

Chapter 4 describes GC data structure members.

Example 6–5 uses the FILL POLYGON routine to draw and fill the star created in Example 6–2.

Example 6–5 Filling a Polygon

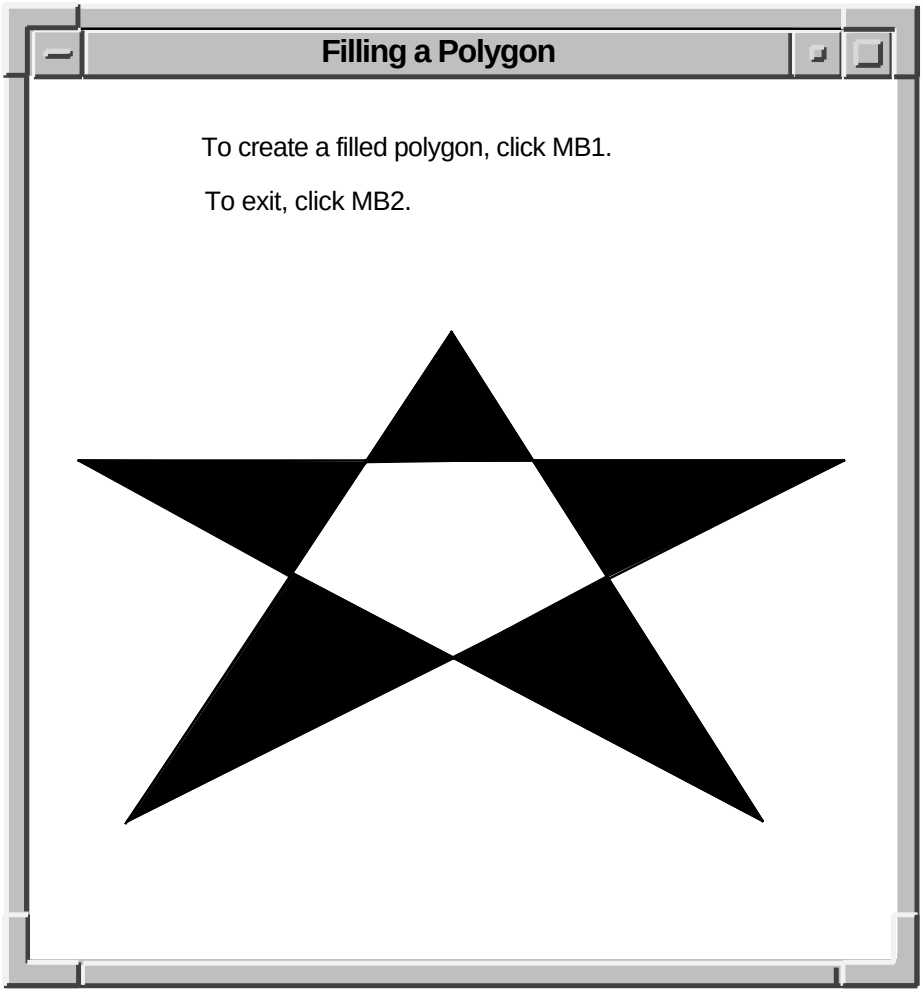
```
                        .
                        .
                        .
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
        }
    }
}
/***** Expose event ****/
static void doExpose(eventP)
XEvent *eventP;
{
    XPoint pt_arr[6];

❶   pt_arr[0].x = 75;
    pt_arr[0].y = 500;
    pt_arr[1].x = 300;
    pt_arr[1].y = 100;
    pt_arr[2].x = 525;
    pt_arr[2].y = 500;
    pt_arr[3].x = 50;
    pt_arr[3].y = 225;
    pt_arr[4].x = 575;
    pt_arr[4].y = 225;
    pt_arr[5].x = 75;
    pt_arr[5].y = 500;

❷    XFillPolygon(dpy, win, gc, &pt_arr, 6, Complex, CoordModeOrigin);
}
                        .
                        .
                        .
```

❶ Use an array of point data structures to specify the points that define the polygon.

❷ The call to fill the polygon refers to a graphics context (*gc*), which the client has previously defined, and an array of point data structures. The constant **Complex** indicates that the path of the line that draws the polygon intersects itself. The constant **CoordModeOrigin** indicates that all points are relative to the origin of *win* (100,100).

Figure 6–7 illustrates the resulting output.

Figure 6–7 Filled Star Created Using the FILL POLYGON Routine



ZK–2569A–GE

6.6 Clearing and Copying Areas

Xlib includes routines that enable clients to clear or copy a specified area of a drawable. Because pixmaps do not have defined backgrounds, clients clearing an area of a pixmap must use the FILL RECTANGLE routine described in Section 6.5.1. For more information about pixmaps, see Chapter 7.

This section describes how to clear windows and copy areas of windows and pixmaps.

6.6.1 Clearing Window Areas

To clear an area of a window, use the CLEAR AREA or CLEAR WINDOW routine. The CLEAR AREA routine clears a specified area and generates an expose event, if the client directs the server to do so.

The CLEAR WINDOW routine clears the entire area of the specified window. If the window has a defined background tile, the window is retiled. If the window has no defined background, the server does not change the window contents.

Example 6–6 illustrates clearing a window.

Example 6–6 Clearing a Window

```
                    .
                    .
                    .
/***** Draw multiple arcs  *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
#define ARC_CNT 16
#define RADIUS 50
#define INNER_RADIUS  20
    XArc arc_arr[ARC_CNT];
    int i;
    int x = eventP->xbutton.x;
    int y = eventP->xbutton.y;

    if (eventP->xbutton.button == Button2) sys$exit (1);
    if (eventP->xbutton.button == Button3)
    {
        XClearWindow(dpy, win);
        return;
    }

    for (i=0;i<ARC_CNT;i++) {
        arc_arr[i].angle1 = (64*360)/ARC_CNT * i;
        arc_arr[i].angle2 = (64*360)/ARC_CNT*3;
        arc_arr[i].width = RADIUS*2;
        arc_arr[i].height = RADIUS*2;
        arc_arr[i].x = x - RADIUS + sin(2*3.14159/ARC_CNT*i) * INNER_RADIUS;
        arc_arr[i].y = y - RADIUS + cos(2*3.14159/ARC_CNT*i) * INNER_RADIUS;
    }

    XDrawArcs(dpy, win, gc, &arc_arr, ARC_CNT);
}
```

The example modifies the *doButtonPress* routine of Example 6–4 to clear the window when the user clicks MB3.

To clear multiple areas, using the FILL RECTANGLES routine is faster than using the CLEAR WINDOW or CLEAR AREA routine. To clear multiple areas on a monochrome screen, first set the function member of the GC data structure to the value specified by the constant **GXclear**. Then call the FILL RECTANGLES routine. If the screen is a color type, set the value of the background to the background of the window before calling FILL RECTANGLES.

6.6.2 Copying Areas of Windows and Pixmaps

Xlib includes the COPY AREA and COPY PLANE routines to enable clients to copy a rectangular area defined on one window or pixmap (the source) to an area of another window or pixmap (the destination). COPY AREA copies areas between drawables of the same root and depth. COPY PLANE copies a single bit plane of the specified drawable to another drawable, regardless of their depths. The bit plane is treated as a stipple with a fill style of **FillOpaqueStippled**. Both drawables must have the same root window.

The server refers to the following members of the GC data structure when copying areas and planes:

Function	Plane mask
Clip x origin	Clip y origin
Subwindow mode	Clip mask
Graphics exposures	

If the client calls the COPY PLANE routine, the server additionally refers to the foreground and background members.

6.7 Defining Regions

A **region** is an arbitrarily defined area within which graphics drawing is clipped. In other words, clipping regions are portions of either windows or pixmaps in which clients can restrict output. As Chapter 4 notes, the SET CLIP MASK, SET CLIP ORIGIN, and SET CLIP RECTANGLES routines define clipping regions. Xlib provides other, more convenient, routines that enable clients to define regions and associate them with drawables without having to change graphics context values directly.

This section describes how to create and manage clipping using Xlib region routines.

6.7.1 Creating Regions

Xlib includes the CREATE REGION and POLYGON REGION routines for creating regions. CREATE REGION creates an empty region. POLYGON REGION creates a region defined by an array of points.

Example 6–7 illustrates using POLYGON REGION to create a star-shaped region. Using the DRAW ARCS routine of Example 6–4, the program limits arc drawing to the star region.

Example 6–7 Defining a Region Using the POLYGON REGION Routine

```
                    .
                    .
                    .
/***** Create the graphics context *****/
static void doCreateGraphicsContext( )
{
    XPoint pt_arr[NUM_PTS];
    XGCValues xgcv;
```

(continued on next page)

Example 6–7 (Cont.) Defining a Region Using the POLYGON REGION Routine

```
❶  pt_arr[0].x = 75;
    pt_arr[0].y = 500;
    pt_arr[1].x = 300;
    pt_arr[1].y = 100;
    pt_arr[2].x = 525;
    pt_arr[2].y = 500;
    pt_arr[3].x = 50;
    pt_arr[3].y = 225;
    pt_arr[4].x = 575;
    pt_arr[4].y = 225;
    pt_arr[5].x = 75;
    pt_arr[5].y = 500;

    /* Create graphics context. */

    xgcv.foreground = doDefineColor(2);
    xgcv.background = doDefineColor(3);

    gc = XCreateGC(dpy, win, GCForeground | GCBackground, &xgcv);
❷  star_region = XPolygonRegion(&pt_arr, NUM_PTS, WindingRule);
}
                    .
                    .
                    .
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case ButtonPress:           doButtonPress(&event); break;
        }
    }
}

/***** Write a message *****/
static void doExpose(eventP)
XEvent *eventP;
{
    char message1[ ] = {"To create arcs in a region, click MB1"};
    char message2[ ] = {"Each click creates a new circle of arcs."};
    char message3[ ] = {"To exit, click MB2"};

    XDrawImageString(dpy, win, gc, 150, 25, message1, strlen(message1));
    XDrawImageString(dpy, win, gc, 150, 50, message2, strlen(message2));
    XDrawImageString(dpy, win, gc, 150, 75, message3, strlen(message3));
}

/***** Draw the arcs *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
#define ARC_CNT 16
#define RADIUS 50
#define INNER_RADIUS  20
    XArc arc_arr[ARC_CNT];
    int i;
    int x = eventP->xbutton.x;
    int y = eventP->xbutton.y;
```

(continued on next page)

Example 6–7 (Cont.) Defining a Region Using the POLYGON REGION Routine

```
    if (eventP->xbutton.button == Button2) sys$exit (1);

❸   XSetRegion(dpy, gc, star_region);
    for (i=0;i<ARC_CNT;i++) {
        arc_arr[i].angle1 = (64*360)/ARC_CNT * i;
        arc_arr[i].angle2 = (64*360)/ARC_CNT*3;
        arc_arr[i].width = RADIUS*2;
        arc_arr[i].height = RADIUS*2;
        arc_arr[i].x = x - RADIUS + sin(2*3.14159/ARC_CNT*i) * INNER_RADIUS;
        arc_arr[i].y = y - RADIUS + cos(2*3.14159/ARC_CNT*i) * INNER_RADIUS;
    }

    XDrawArcs(dpy, win, gc, &arc_arr, ARC_CNT);
}
```

❶ Define an array of point data structures to define the clipping region.

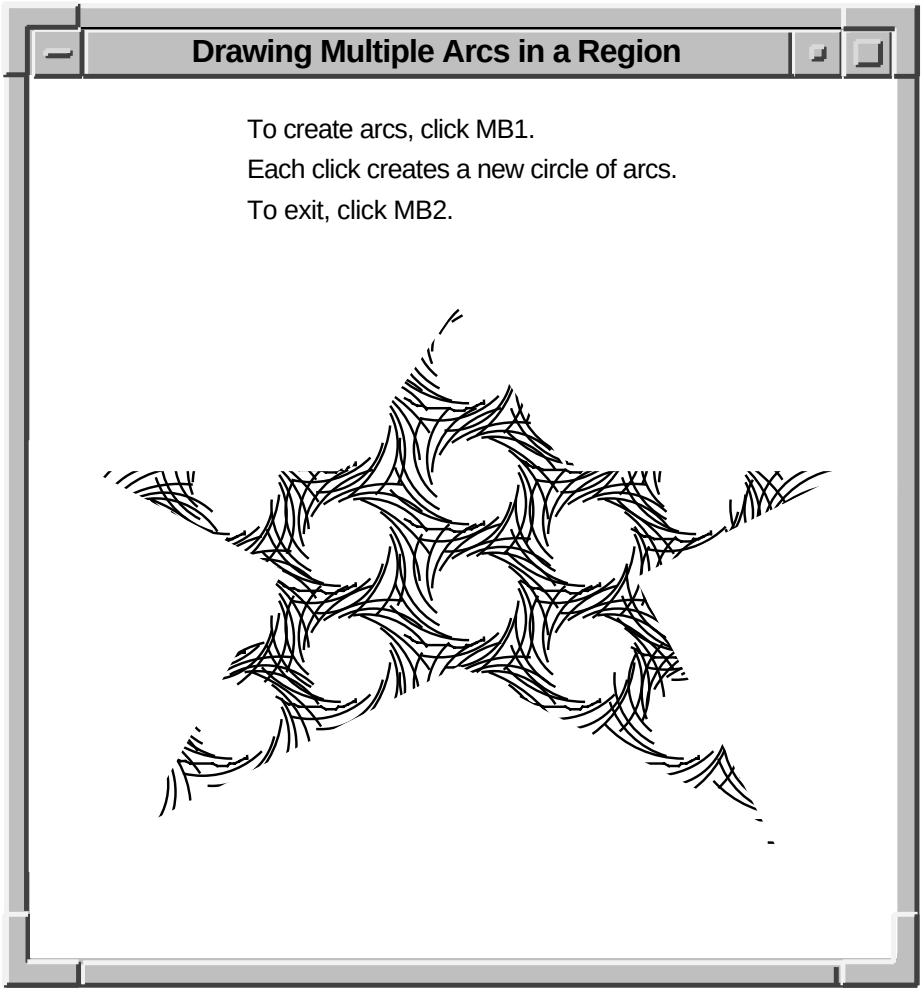
❷ Define the clipping region. Note that defining the region does not associate it with a graphics context.

Fill rule can be either even/odd rule or winding rule. For more information about fill rule, see Chapter 4.

❸ Associate the region with a graphics context. The association sets fields in the specified GC data structure that control clipping. Drawables that refer to the GC data structure have output clipped to the region.

Figure 6–8 illustrates sample output from the program.

Figure 6–8 Arcs Drawn Within a Region



ZK−2507A−GE

6.7.2 Managing Regions

Xlib includes routines that enable clients to do the following:

- Move and shrink a region
- Compute the intersection, union, and results of two regions
- Determine if regions are empty or equal
- Locate a point or rectangle within a region

Table 6–5 lists and describes Xlib routines that manage regions.

Table 6–5 Routines for Managing Regions

Routine	Description
Creating, Copying, and Destroying	
CREATE REGION	Creates a new empty region
SET REGION	Sets the clip mask of a GC to a region
DESTROY REGION	Deallocates storage associated with a specified region
Moving and Shrinking	
OFFSET REGION	Moves a region a specified amount
SHRINK REGION	Reduces a region a specified amount
Computing	
INTERSECT REGION	Computes the intersection of two regions
UNION REGION	Computes the union of two regions
UNION RECT WITH REGION	Creates a union of a source region with a rectangle
SUBTRACT REGION	Subtracts two regions
XOR REGION	Calculates the difference between the union and intersection of two regions
Determining If Regions Are Empty or Equal	
EMPTY REGION	Determines if a region is empty
EQUAL REGION	Determines if two regions have the same offset, size, and shape
Locating a Point or Rectangle Within a Region	
POINT IN REGION	Determines if a point is within a region
RECT IN REGION	Determines if a rectangle is within a region

Example 6–8 illustrates creating a region from the intersection of two others.

Example 6–8 Defining the Intersection of Two Regions

(continued on next page)

Example 6–8 (Cont.) Defining the Intersection of Two Regions

```
Pixmap pixmap1, pixmap2, pixmap3;
Region region1, region2, region3;
                    .
                    .
                    .
/***** doInitialize *****/
static void doInitialize( )
{
    dpy = XOpenDisplay(0);

    screen = XDefaultScreenOfDisplay(dpy);

    doCreateWindows( );

    doCreateGraphicsContext( );

    doCreatePixmap( );

    doCreateRegion( );

    doMapWindows( );
}
                    .
                    .
                    .
/***** Create the pixmap *****/

❶static void doCreatePixmap( )
{
    pixmap1 = XCreatePixmap(dpy, win, pixWidth, pixHeight,
              DefaultDepthOfScreen(screen));
    pixmap2 = XCreatePixmap(dpy, win, pixWidth, pixHeight,
              DefaultDepthOfScreen(screen));
    pixmap3 = XCreatePixmap(dpy, win, pixWidth, pixHeight,
              DefaultDepthOfScreen(screen));

    /* Set the pixmap background */
    XFillRectangle(dpy, pixmap1, gc, 0, 0, pixWidth, pixHeight);
    XFillRectangle(dpy, pixmap2, gc, 0, 0, pixWidth, pixHeight);
    XFillRectangle(dpy, pixmap3, gc, 0, 0, pixWidth, pixHeight);

    /* Redefine foreground value for line drawing and text */
    XSetForeground(dpy, gc, doDefineColor(2));

    /* Draw Line into the pixmap */
    XDrawLine(dpy, pixmap1, gc, 0, 4, 0, 8);
    XDrawLine(dpy, pixmap2, gc, 4, 0, 8, 0);
    XDrawLine(dpy, pixmap3, gc, 0, 4, 0, 8);
    XDrawLine(dpy, pixmap3, gc, 4, 0, 8, 0);

}
/***** Create the region *****/
static void doCreateRegion( )
{
❷   XPoint pt_arr_1[num_pts], pt_arr_2[num_pts];

    pt_arr_1[0].x = 200;
    pt_arr_1[0].y = 100;
    pt_arr_1[1].x = 50;
    pt_arr_1[1].y = 300;
    pt_arr_1[2].x = 200;
    pt_arr_1[2].y = 500;
    pt_arr_1[3].x = 350;
    pt_arr_1[3].y = 300;
```

(continued on next page)

Example 6–8 (Cont.) Defining the Intersection of Two Regions

```
    pt_arr_2[0].x = 400;
    pt_arr_2[0].y = 100;
    pt_arr_2[1].x = 250;
    pt_arr_2[1].y = 300;
    pt_arr_2[2].x = 400;
    pt_arr_2[2].y = 500;
    pt_arr_2[3].x = 550;
    pt_arr_2[3].y = 300;

    region1 = XPolygonRegion(pt_arr_1, num_pts, WindingRule);
    region2 = XPolygonRegion(pt_arr_2, num_pts, WindingRule);

}
                    .
                    .
                    .
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case ButtonPress:           i++; doButtonPress(&event); break;
        }
    }
}

/***** Write a message *****/
static void doExpose(eventP)
XEvent *eventP;
{
    char message1[ ] = {"To map regions click MB1 three times."};
    char message2[ ] = {"To exit, click MB2."};

    XDrawImageString(dpy, win, gc, 150, 25, message1, strlen(message1));
    XDrawImageString(dpy, win, gc, 150, 50, message2, strlen(message2));
}
/***** Map the regions when the button is pressed *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
    char message3[ ] = {"That's it! Click MB2 to exit."};

    if (eventP->xbutton.button == Button2) sys$exit (1);
    if (i == 1){
```

(continued on next page)

Example 6–8 (Cont.) Defining the Intersection of Two Regions

```
        /* Redefine the fill style for stippling */
❸           XSetFillStyle(dpy, gc, FillTiled);

        XClearWindow(dpy, win);
        XSetTile(dpy, gc, pixmap1);
❹       XSetRegion(dpy, gc, region1);
❺       XFillRectangle(dpy, win, gc, xOrigin, yOrigin, winW, winH);
        }
    else if (i == 2){
❻       XClearWindow(dpy, win);
        XSetTile(dpy, gc, pixmap2);
        XSetRegion(dpy, gc, region2);
        XFillRectangle(dpy, win, gc, xOrigin, yOrigin, winW, winH);
        }
    else if (i == 3){
        XClearWindow(dpy, win);
❼       region3 = XCreateRegion();
        XIntersectRegion(region1, region2, region3);
        XSetTile(dpy, gc, pixmap3);
        XSetRegion(dpy, gc, region3);
        XFillRectangle(dpy, win, gc, xOrigin, yOrigin, winW, winH);
        }
    else{
        /* To draw text, redefine the fill style as solid */
❽       XSetFillStyle(dpy, gc, FillSolid);
        XDrawImageString(dpy, win, gc, 150, 50, message3, strlen(message3));
        }
}
```

❶ Pixmaps are used to tile the window with horizontal, vertical, and cross-hatched lines. For information about pixmaps, see Chapter 7.

❷ Arrays of point data structures define two regions.

❸ After writing messages in the window, the fill style defined in the GC data structure is changed to tile the window with pixmaps. The subsequent call to SET TILE defines one of the three pixmaps created earlier as the window background pixmap. For information about fill styles and tiling, see Chapter 4.

❹ The SET REGION routine specifies the clipping region in the graphics context. The region defined by *pt_arr1* is first specified.

❺ FILL RECTANGLE repaints the window, filling it with the tiling pattern defined in *pixmap1*. Tiling is restricted to the region defined by *region1*.

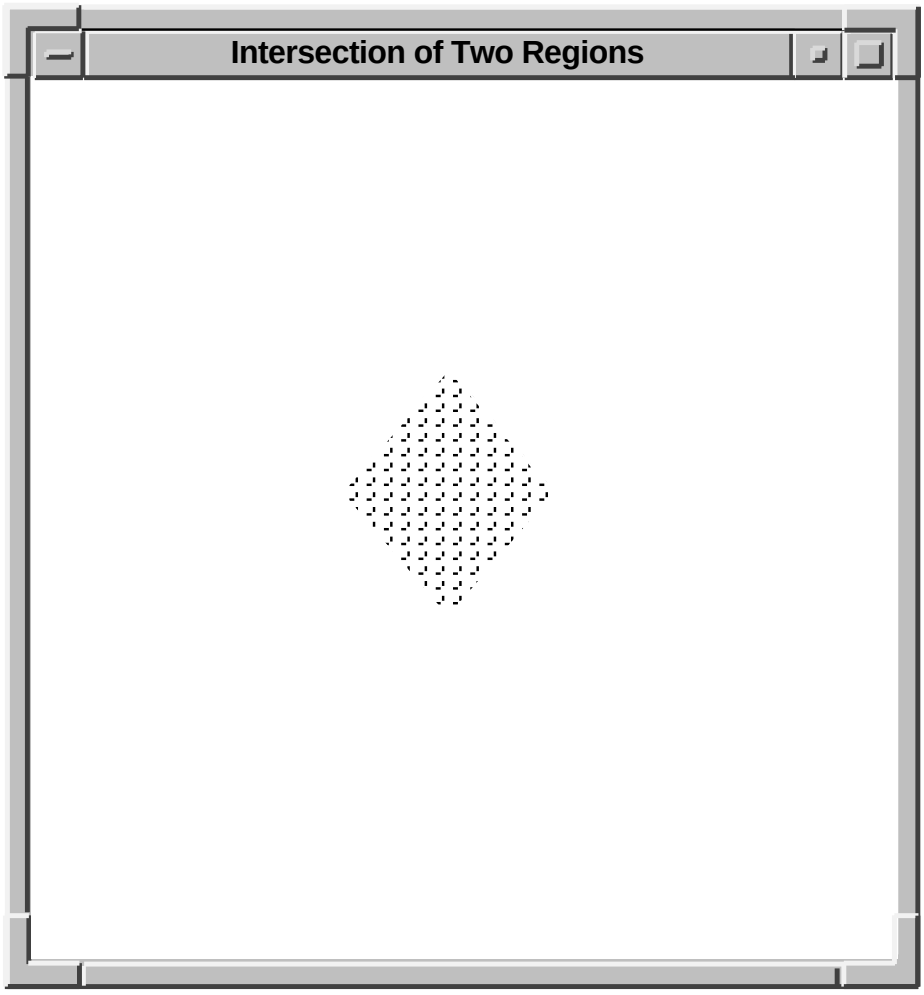
❻ Before specifying a new tiling pattern and region, the window is cleared.

❼ CREATE REGION creates an empty region and returns an identifier, *region3*. Xlib returns the results of intersecting *region1* and *region2* to *region3*.

❽ Before displaying a final message in the window, the fill style is redefined to solid to enable text writing.

Figure 6–9 illustrates the output from the program.

Figure 6–9 Intersection of Two Regions



ZK–2508A–GE

6.8 Defining Cursors

A **cursor** is a bit image on the screen that indicates either the movement of a pointing device or the place where text will next appear. Xlib enables clients to associate a cursor with each window they create. After making the association between cursor and window, the cursor is visible whenever it is in the window. If the cursor indicates movement of a pointing device, the movement of the cursor in the window automatically reflects the movement of the device.

Xlib and VMS DECwindows provide fonts of predefined cursors. Clients that want to create their own cursors can either define a font of shapes and masks or create cursors using pixmaps.

This section describes the following:

- Creating cursors using the Xlib cursor font, a font of shapes and masks, and pixmaps
- Associating cursors with windows

- Managing cursors
- Freeing memory allocated to cursors when clients no longer need them

6.8.1 Creating Cursors

Xlib enables clients to use predefined cursors or to create their own cursors. To create a predefined Xlib cursor, use the CREATE FONT CURSOR routine. Xlib cursors are predefined in DECW$INCLUDE:CURSORFONT.H. See the *X and Motif Quick Reference Guide* for a list of the constants that refer to the predefined Xlib cursors.

The following example creates a sailboat cursor, one of the predefined Xlib cursors, and associates the cursor with a window:

```
Cursor fontcursor;
                .
                .
                .
fontcursor = XCreateFontCursor(dpy, XC_sailboat);
XDefineCursor(dpy, win, fontcursor);
```

The DEFINE CURSOR routine makes the sailboat cursor automatically visible when the pointer is in window *win*.

In addition to the standard Xlib cursors, VMS DECwindows provides another set of cursors. VMS DECwindows cursors are predefined in SYS$LIBRARY:DECW$CURSOR.H. Table 6–6 lists the constants that refer to the predefined VMS DECwindows cursors.

Table 6–6 Predefined VMS DECwindows Cursors

decw$c_select_cursor	decw$c_leftselect_cursor
decw$c_help_select_cursor	decw$c_wait_cursor
decw$c_inactive_cursor	decw$c_resize_cursor
decw$c_vpane_cursor	decw$c_hpane_cursor
decw$c_text_insertion_cursor	decw$c_text_insertion_bl_cursor
decw$c_cross_hair_cursor	decw$c_draw_cursor
decw$c_pencil_cursor	decw$c_rpencil_cursor
decw$c_center_cursor	decw$c_rightselect_cursor
decw$c_wselect_cursor	decw$c_eselect_cursor
decw$c_x_cursor	decw$c_circle_cursor
decw$c_mouse_cursor	decw$c_lpencil_cursor
decw$c_leftgrab_cursor	decw$c_grabhand_cursor
decw$c_rightgrab_cursor	decw$c_leftpointing_cursor
decw$c_uppointing_cursor	decw$c_rightpointing_cursor

To create a predefined VMS DECwindows cursor, use the CREATE GLYPH CURSOR routine. CREATE GLYPH CURSOR selects a cursor shape and cursor mask from the VMS DECwindows cursor font, defines how the cursor appears on the screen, and assigns a unique cursor identifier. The following example illustrates creating the select cursor and associating the cursor with a window:

```
Font cursorfont
Cursor glyphcursor;
XColor forecolor, backcolor;
                  .
                  .
                  .
cursorfont = XLoadFont(dpy, "decw$cursor");
XSetFont(dpy, gc, cursorfont);

glyphcursor = XCreateGlyphCursor(dpy, cursorfont, cursorfont,
     decw$c_select_cursor, decw$c_select_cursor + 1,
     &forecolor, &backcolor);
XDefineCursor(dpy, win, glyphcursor);
```

To create client-defined cursors, either create a font of cursor shapes or define cursors using pixmaps. In each case, the cursor consists of the following components:

- Shape—Defines the cursor as it appears without modification in a window
- Mask—Acts as a clip mask to define how the cursor actually appears in a window
- Background color—Specifies RGB values used for the cursor background
- Foreground color—Specifies RGB values used for the cursor foreground
- Hotspot—Defines the position on the cursor that reflects movements of the pointing device

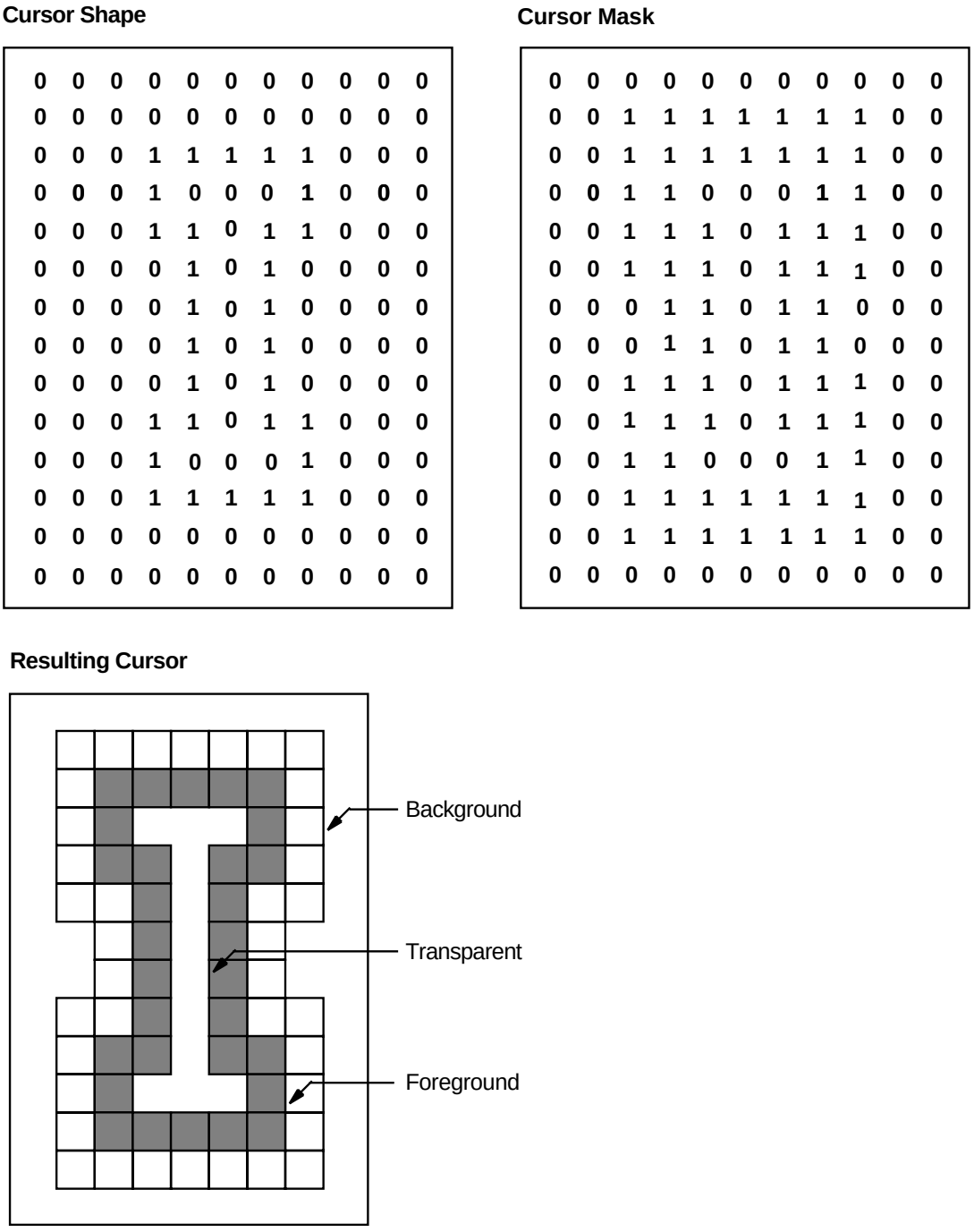
Figure 6–10 illustrates the relationship between the cursor shape and the cursor mask. The cursor shape defines the cursor as it would appear on the screen without modification. The cursor mask bits that are set to 1 select which bits of the cursor shape are actually displayed. If the mask bit has a value of 1, the corresponding shape bit is displayed whether it has a value of 1 or 0. If the mask bit has a value of 0, the corresponding shape bit is not displayed.

In the resulting cursor shape, bits with a 0 value are displayed in the specified background color; bits with a 1 value are displayed in the specified foreground color.

To create a client-defined cursor from a font of glyphs, use the CREATE GLYPH CURSOR routine, specifying the cursor and mask fonts that contain the glyphs. To create a cursor from pixmaps, use the CREATE PIXMAP CURSOR routine. The pixmaps must have a depth of one. If the depth is not one, the server generates an error.

The size of the pixmap cursor must be supported by the display on which the cursor is visible. To determine the supported size closest to the size the client specifies, use the QUERY BEST CURSOR routine. Example 6–9 illustrates creating a pencil pointer cursor from two pixmaps.

Figure 6–10 Cursor Shape and Cursor Mask



ZK−0154A−GE

Example 6–9 Creating a Pixmap Cursor

(continued on next page)

Example 6–9 (Cont.) Creating a Pixmap Cursor

```
#include <decw$include/Xlib.h>
#include <decw$include/Xutil.h>

#define winW 600
#define winH 600
#define pencil_width 16
#define pencil_height 16
#define pencil_xhot 1
#define pencil_yhot 15

Display *dpy;
Window win;
Pixmap pixmap, pencil;
Pixmap pencil_mask;
Cursor pencil_cursor;
                        .
                        .
                        .
static char pencil_bits[] = {
   0x0000, 0x0070, 0x0000, 0x0088, 0x0000, 0x008C, 0x0000, 0x0096,
   0x0000, 0x0069, 0x0080, 0x0030, 0x0040, 0x0010, 0x0020, 0x0008,
   0x0010, 0x0004, 0x0008, 0x0002, 0x0008, 0x0001, 0x0094, 0x0000,
   0x0064, 0x0000, 0x001E, 0x0000, 0x0006, 0x0000, 0x0000, 0x0000};

static char pencil_mask_bits[] = {
   0x00, 0xF8, 0x00, 0xFC, 0x00, 0xFE, 0x00, 0xFF, 0x80, 0xFF, 0xC0, 0x7F,
   0xE0, 0x3F, 0xF0, 0x1F, 0xF8, 0x0F, 0xFC, 0x07, 0xFC, 0x03, 0xFE, 0x01,
   0xFE, 0x00, 0x7F, 0x00, 0x1F, 0x00, 0x07, 0x00};
                        .
                        .
                        .
/***** Create the cursor *****/
static void doCreateCursor( )
{
    XColor dummy, cursor_foreground, cursor_background;

     /*Create pixmaps for cursor */
❶   pixmap = XCreatePixmap(dpy, XDefaultRootWindow(dpy), 1, 1, 1);

❷   XLookupColor(dpy, XDefaultColormapOfScreen(screen), "black",
        &dummy, &cursor_foreground);
    XLookupColor(dpy, XDefaultColormapOfScreen(screen), "white",
        &dummy, &cursor_background);
❸   pencil = XCreatePixmapFromBitmapData(dpy, pixmap, pencil_bits,
        pencil_width, pencil_height, 1, 0, 1);
    pencil_mask = XCreatePixmapFromBitmapData(dpy, pixmap, pencil_mask_bits,
        pencil_width, pencil_height, 1, 0, 1);

❹   pencil_cursor = XCreatePixmapCursor(dpy, pencil, pencil_mask,
        &cursor_foreground, &cursor_background, pencil_xhot, pencil_yhot);
    XDefineCursor(dpy, win, pencil_cursor);
}
```

❶ The client first creates a pixmap into which it will draw bit images for the cursor and cursor mask. Note that the depth of the pixmap must be one. For information about creating pixmaps, see Chapter 7.

❷ The LOOKUP COLOR routine returns the color value associated with the named color to the *cursor_foreground* and *cursor_background* variables. For information about LOOKUP COLOR, see Chapter 5.

❸ The CREATE PIXMAP FROM BITMAP DATA routine writes an image into a specified pixmap. The client uses the routine to write images for the cursor and the cursor mask into two pixmaps with depths of one.

❹ The CREATE PIXMAP CURSOR routine uses the two pixmaps to create the pixmap cursor.

6.8.2 Managing Cursors

To dissociate a cursor from a window, call the UNDEFINE CURSOR routine. After a call to UNDEFINE CURSOR, the cursor associated with the parent window is used. If the window is a root window, UNDEFINE CURSOR restores the default cursor. UNDEFINE CURSOR does not destroy a cursor. Using its identifier, the client can still refer to the cursor and associate it with a window.

To change the color of a cursor, use the RECOLOR CURSOR routine. If the cursor is displayed on the screen, the change is immediately visible. For information about defining foreground and background colors, see Chapter 5. For information about loading fonts, see Chapter 8.

6.8.3 Destroying Cursors

To destroy a cursor, use the FREE CURSOR routine. FREE CURSOR deletes the association between the cursor identifier and the specified cursor. It also frees memory allocated for the cursor.

7
Using Pixmaps and Images

Xlib enables clients to create and work with both on-screen graphics, such as lines and cursors, and off-screen images, such as pixmaps. Chapter 4 and Chapter 6 describe how to work with on-screen graphics objects.

This chapter describes how to work with off-screen graphics resources, including the following topics:

- Creating and freeing pixmaps
- Creating and managing bitmaps
- Working with images

7.1 Creating and Freeing Pixmaps

A **pixmap** is an area of memory into which clients can either define an image or temporarily save part of a screen. Pixmaps are useful for defining cursors and icons, for creating tiling patterns, and for saving portions of a window that have been exposed. Additionally, drawing complicated graphics sequences into pixmaps and then copying the pixmaps to a window are often faster than drawing the sequences directly to a window.

Use the CREATE PIXMAP routine to create a pixmap. The routine creates a pixmap of a specified width, height, and depth. If the width or height is zero or the depth is not supported by the drawable root window, the server returns an error. The pixmap must be associated with a window, which can be either an input-output or an input-only window.

Example 7–1 illustrates creating a pixmap to use as a backing store for drawing the star of Example 6–5.

Example 7–1 Creating a Pixmap

```
                    .
                    .
                    .
Pixmap pixmap;
int n, exposeflag = 0;
                    .
                    .
                    .
/***** Create the graphics context *****/
static void doCreateGraphicsContext( )
{
    XGCValues xgcv;

    /* Create graphics context. */
```

(continued on next page)

Example 7–1 (Cont.) Creating a Pixmap

```
❶  xgcv.foreground = doDefineColor(1);
    xgcv.background = doDefineColor(1);

    gc = XCreateGC(dpy, win, GCForeground | GCBackground, &xgcv);
}

/***** Create the pixmap *****/
static void doCreatePixmap( )
{
    XPoint pt_arr[6];

    pt_arr[0].x = 75;
    pt_arr[0].y = 500;
    pt_arr[1].x = 300;
    pt_arr[1].y = 100;
    pt_arr[2].x = 525;
    pt_arr[2].y = 500;
    pt_arr[3].x = 50;
    pt_arr[3].y = 225;
    pt_arr[4].x = 575;
    pt_arr[4].y = 225;
    pt_arr[5].x = 75;
    pt_arr[5].y = 500;

❷  pixmap = XCreatePixmap(dpy, win, winW, winH, DefaultDepthOfScreen(screen));
❸  XFillRectangle(dpy, pixmap, gc, 0, 0, winW, winH);
    XSetForeground(dpy, gc, doDefineColor(2));
❹  XFillPolygon(dpy, pixmap, gc, &pt_arr, 6, Complex, CoordModeOrigin);
}
                        .
                        .
                        .
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case ButtonPress:           doButtonPress(&event); break;
        }
    }
}

/***** Write a message *****/
static void doExpose(eventP)
XEvent *eventP;
{
    char message1[ ] = {"To create a filled polygon, click MB1."};
    char message2[ ] = {"To exit, click MB2."};

    XDrawImageString(dpy, win, gc, 150, 25, message1, strlen(message1));
    XDrawImageString(dpy, win, gc, 150, 50, message2, strlen(message2));
❺  if (!exposeflag)
        exposeflag = 1;
    else
        XCopyArea(dpy, pixmap, win, gc, 0, 0, winW, winH, 0, 0);
        XDrawImageString(dpy, win, gc, 150, 50, message2, strlen(message2));
}
```

(continued on next page)

Example 7–1 (Cont.) Creating a Pixmap

```
/***** Draw the polygon in the window *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
    char message2[ ] = {"To exit, click MB2."};

    if (eventP->xbutton.button == Button2) sys$exit (1);

    XCopyArea(dpy, pixmap, win, gc, 0, 0, winW, winH, 0, 0);
    XDrawImageString(dpy, win, gc, 150, 50, message2, strlen(message2));
}
```

❶ Pixmaps use only the foreground member of the graphics context to define color. Because the client is using the pixmap as backing store, which is copied into the window to repaint exposed areas, both foreground and background members of the graphics context are first defined as the window background color.

❷ The pixmap has the width, height, and depth of the window.

❸ FILL RECTANGLE fills the pixmap with the background color of the window. After filling the pixmap to ensure that pixel values of both the pixmap and window background are the same, the foreground color is redefined for graphics operations.

❹ After redefining foreground color, the client draws the polygon into the pixmap. For description of specifying and filling the polygon, see Example 6–5.

❺ At the first window exposure, the client draws only the text into the window. On subsequent exposures, the client copies the pixmap into the window to repaint exposed areas. For a description of handling exposure events, see Chapter 9.

Note that the CREATE PIXMAP routine is not a synchronous routine and does not return an error if the routine fails to create a pixmap. Although Xlib returns a resource ID for this routine, it does not indicate that a valid resource was created by the server. Refer to Section 9.13.3 for a method to check if a pixmap, or any X resource, has been created.

When a client no longer needs a pixmap, use the FREE PIXMAP routine to free storage associated with it. FREE PIXMAP first deletes the association between the pixmap identifier and the pixmap and then frees pixmap storage.

7.2 Creating and Managing Bitmaps

Xlib enables clients to create files of bitmap data and then to use those files to create either bitmaps or pixmaps. To create a bitmap data file, use the WRITE BITMAP FILE routine. Example 7–2 illustrates creating a pixmap and writing the pixmap data into a bitmap data file.

Example 7–2 Creating a Bitmap Data File

(continued on next page)

Example 7–2 (Cont.) Creating a Bitmap Data File

```
                    .
                    .
                    .
/***** Create the pixmap *****/
static void doCreatePixmap( )
{
    XPoint pt_arr[5];

    pt_arr[0].x = 20;
    pt_arr[0].y = 0;
    pt_arr[1].x = 20;
    pt_arr[1].y = 5;
    pt_arr[2].x = 20;
    pt_arr[2].y = 10;
    pt_arr[3].x = 20;
    pt_arr[3].y = 15;
    pt_arr[4].x = 20;
    pt_arr[4].y = 20;

    pixmap = XCreatePixmap(dpy, win, pixW, pixH, DefaultDepthOfScreen(screen));
    XFillRectangle(dpy, pixmap, gc, 0, 0, pixW, pixH);
    XSetForeground(dpy, gc, doDefineColor(2));
    XDrawLines(dpy, pixmap, gc, &pt_arr, 5, CoordModeOrigin);
    status = XWriteBitmapFile(dpy, "bitfile.dat", pixmap, 20, 20, 0, 0);
}
```

The client first creates a pixmap using the method described in Section 7.1 and then calls the WRITE BITMAP FILE routine to write the pixmap data into the BITFILE.DAT bitmap file.

To create a bitmap or pixmap from a bitmap data file, use either the CREATE BITMAP FROM DATA or CREATE PIXMAP FROM DATA routine. Example 7–3 illustrates creating a pixmap from the bitmap data stored in BITFILE.DAT.

Example 7–3 Creating a Pixmap from Bitmap Data

```
                        .
                        .
                        .
/***** Create the pixmap *****/
static void doCreatePixmap( )
{
   static char LINES[] = {
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x3f, 0x06, 0x00,
      0x03, 0x0c, 0x00, 0x03, 0x18, 0x02, 0x03, 0x30, 0x00, 0xf3, 0x7f, 0x05,
      0x03, 0x30, 0x00, 0x03, 0x18, 0x00, 0x03, 0x0c, 0x00, 0x3f, 0x06, 0x00,
      0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00,
      0xaa, 0xaa, 0x0a, 0x55, 0x55, 0x05, 0xaa, 0xaa, 0x0a, 0x55, 0x55, 0x05};

    pixmap = XCreatePixmapFromBitmapData(dpy, win, LINES, pixW, pixH,
        xgcv.foreground, xgcv.background, XDefaultDepthOfScreen(screen));
    XSetWindowBackgroundPixmap(dpy, win, pixmap);
}
                        .
                        .
                        .
```

The client uses the pixmap to define window background.

7.3 Working with Images

Instead of managing images directly, clients perform operations on them by using the image data structure, which includes a pointer to data such as the LINES array defined in Example 7–3. In addition to the image data, the image data structure includes pointers to client-defined functions that perform the following operations:

- Destroying an image
- Getting a pixel from the image
- Storing a pixel in the image
- Extracting part of the image
- Adding a constant to the image

If the client has not defined a function, the corresponding Xlib routine is called by default.

The following illustrates the data structure:

```
typedef struct _XImage {
    int width, height;
    int xoffset;
    int format;
    char *data;
    int byte_order;
    int bitmap_unit;
    int bitmap_bit_order;
    int bitmap_pad;
    int depth;
    int bytes_per_line;
    int bits_per_pixel;
    unsigned long red_mask;
    unsigned long green_mask;
    unsigned long blue_mask;
    char *obdata;
    struct funcs {
        struct _XImage *(*create_image)();
        int (*destroy_image)();
        unsigned long (*get_pixel)();
        int (*put_pixel)();
        struct _XImage *(*sub_image)();
        int (*add_pixel)();
        } f;
} XImage;
```

Table 7–1 describes the members of the data structure.

Table 7–1 Image Data Structure Members

Member Name	Contents
width	Specifies the width of the image.
height	Specifies the height of the image.
offset	Specifies the number of pixels offset in the x direction. Specifying an offset permits the server to ignore the beginning of scanlines and rapidly display images when Z pixmap format is used.
format	Specifies whether the data is stored in XY pixmap or Z pixmap format. The following flags facilitate specifying data format:

Flag Name	Description
XYBitmap	A single bitmap representing one plane
XYPixmap	A set of bitmaps representing individual planes
ZPixmap	Data organized as a list of pixel values viewed as a horizontal row

data	Indicates the address of the image data.
byte_order	Indicates whether the least significant or the most significant byte is first.
bitmap_unit	Specifies whether the bitmap is organized in units of 8-, 16-, or 32-bits.

(continued on next page)

Table 7–1 (Cont.) Image Data Structure Members

Member Name	Contents
bitmap_bit_order	Specifies whether the bitmap order is least or most significant.
bitmap_pad	Specifies whether padding in XY format or Z format should be done in units of 8-, 16-, or 32-bits.
depth	Specifies the depth of the image.
bytes_per_line	Specifies the bytes per line to be used as an accelerator.
bits_per_pixel	Indicates for Z format the number of bits per pixel.
red_mask	Specifies red values for Z format.
green_mask	Specifies green values for Z format.
blue_mask	Specifies blue values for Z format.
obdata	Specifies the address of a data structure that contains object routines.
create_image	Specifies a client-defined function that creates an image.
destroy_image	Specifies a client-defined function that destroys an image.
get_pixel	Specifies a client-defined function that gets the value of a pixel in the image.
put_pixel	Specifies a client-defined function that changes the value of a pixel in the image.
sub_image	Specifies a client-defined function that creates a new image from an existing one.
add_pixel	Specifies a client-defined function that increments the value of each pixel in the image by a constant.

To create an image, use either the CREATE IMAGE or the GET IMAGE routine. CREATE IMAGE initializes an image data structure, including a reference to the image data. For example, the following call creates an image data structure that points to the image data LINES, illustrated in Example 7–3:

```
#define pixW 16
#define pixH 16
#define bitmap_pad 16
#define bytes_per_line 16

XImage *image;
                          .
                          .
                          .
     image = XCreateImage(dpy, XDefaultVisualOfScreen(screen),
          XDefaultDepthOfScreen(screen), ZPixmap, 0, &LINES,
          pixW, pixH, bitmap_pad, bytes_per_line);
                          .
                          .
                          .
```

Note that the CREATE IMAGE routine does not allocate storage space for the image data.

To create an image from a drawable, use the GET IMAGE routine. In the following example, the client creates an image from a pixmap:

```
#define xOrigin 0
#define yOrigin 0
#define pixW 16
#define pixH 16
                        .
                        .
                        .
    image = XGetImage(dpy, pixmap, xOrigin, yOrigin, pixW,
         pixH, AllPlanes, ZPixmap);
                        .
                        .
                        .
```

When the client calls the GET IMAGE routine and the drawable is a window, the window must be mapped. In addition, if there are no inferiors or overlapping windows, the specified rectangle of the window should be fully visible on the screen and wholly contained within the outside edges of the window. In other words, an error results if the GET IMAGE routine is called to get a portion of a window that is off-screen.

To transfer an image from memory to a drawable, use the PUT IMAGE routine. In the following example, the client transfers the image from memory to a window:

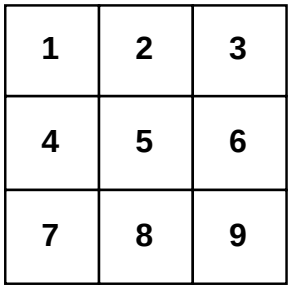
```
#define pixW 16
#define pixH 16
#define srcX 0
#define srcY 0
#define dstX 200
#define dstY 200
                        .
                        .
                        .
    XPutImage(dpy, win, gc, image, srcX, srcY, dstX, dstY,
        pixW, pixH);
                        .
                        .
                        .
```

The call transfers the entire image, which was created in the call to GET IMAGE, from memory to coordinates (200, 200) in the window.

As the description of the image data structure indicates, Xlib enables clients to store an image in the following ways:

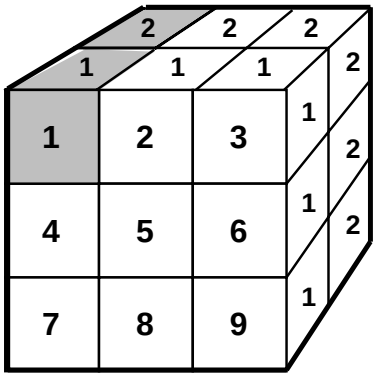
- As a bitmap—XY bitmap format stores the image as a two-dimensional array. Figure 7–1 illustrates XY bitmap format.
- As a set of bitmaps—XY pixmap format stores the image as a stack of bitmaps. Figure 7–2 illustrates XY pixmap format.
- As a list of pixel values—Z pixmap format stores the image as a list of pixel values viewed as a horizontal row. Each example of creating an image uses Z pixmap format. Figure 7–3 illustrates scanline order.

Figure 7–1 XY Bitmap Format



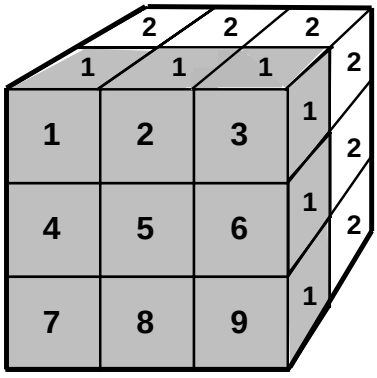
ZK–0157A–GE

Figure 7–2 XY Pixmap Format



ZK–0155A–GE

Figure 7–3 Z Format



ZK–0156A–GE

Xlib includes routines to change images by manipulating their pixel values and creating new images out of subsections of existing images. Table 7–2 lists these routines and their use. Clients can override these routines by defining functions referred to in the image data structure.

Table 7–2 Routines That Change Images

Routine	Description
ADD PIXEL	Increments each pixel in an image by a constant value
GET PIXEL	Returns the pixel value of an image
PUT PIXEL	Sets the pixel value of an image
SUB IMAGE	Creates a new image out of a subsection of an existing image

When a client no longer needs an image, use the DESTROY IMAGE routine to deallocate memory associated with the image data structure.

8
Writing Text

This chapter describes writing text using Xlib. The chapter includes the following topics:

- Characters and fonts—A description of the composition of characters and types of fonts and their components
- Specifying fonts—How to load a font and associate it with a graphics context
- Getting information about fonts—How to get information about fonts and text
- Freeing font resources—How to free memory associated with fonts
- Computing text size—How to determine the size of text
- Drawing text—How to write text on the screen
- Using fonts efficiently—How to speed up font searches and other hints

VMS DECwindows provides a font compiler to enable programmers to convert ASCII files into binary form. For a guide to using the font compiler, see Appendix A.

8.1 Characters and Fonts

The smallest unit of text the server displays on a screen is a **character**. Pixels that form a character are enclosed within a **bounding box** that defines the number of pixels the server turns on or off to represent the character on the screen. For example, Figure 8–1 illustrates the bounding box that encloses the character y.

The server turns each pixel within the bounding box either on or off, depending on the character. Consequently, bounding box size affects performance. Larger bounding boxes require more server time to process than do smaller boxes.

The character is positioned relative to the **baseline** and the character origin. The baseline is logically viewed as the x-axis that runs just below nondescending characters. The **character origin** is a point along the baseline. The **left bearing** of the character is the distance from the origin to the left edge of the bounding box; the **right bearing** is the distance from the origin to the right edge. **Ascent** and **descent** measure the distance from the baseline to the top and bottom of the bounding box, respectively. **Character width** is the distance from the origin to the next character origin $(x + width, y)$.

Figure 8–1 Composition of a Character

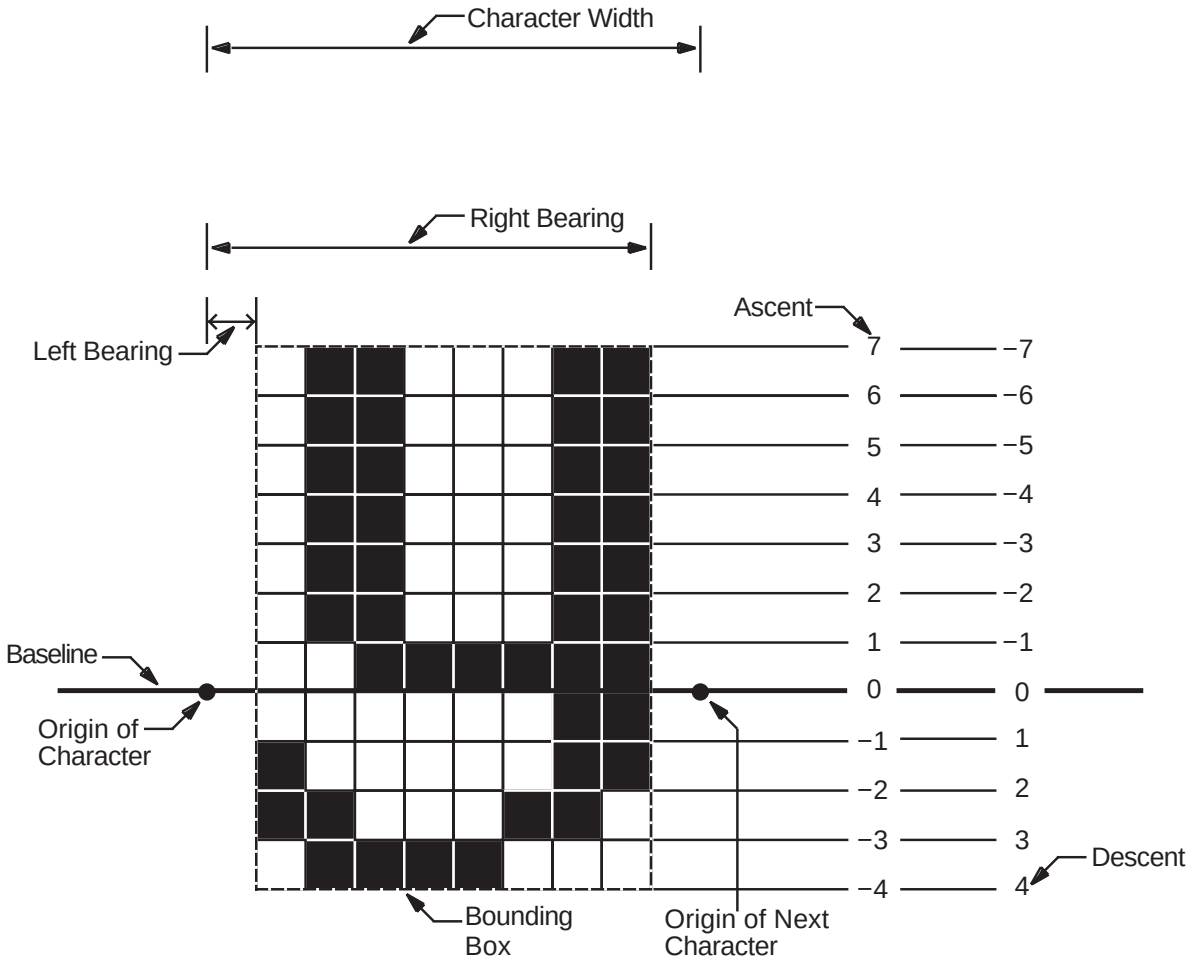
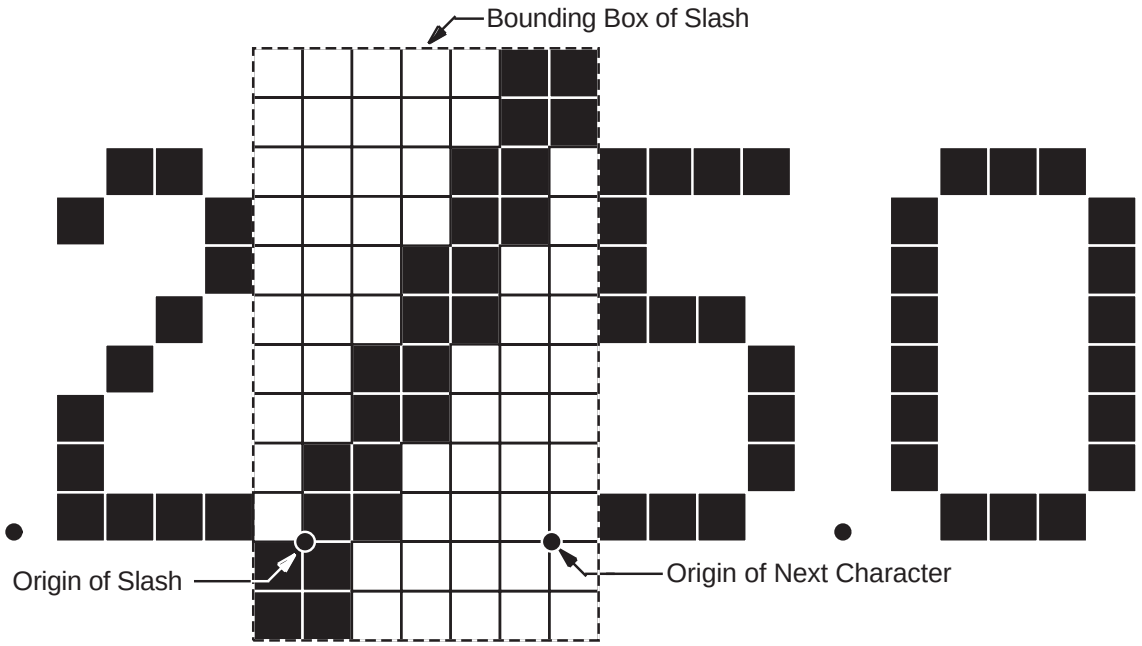


Figure 8–2 illustrates that the bounding box of a character can extend beyond the character origin. The bounding box of the slash extends one pixel to the left of the origin of the slash, giving the character a left bearing of −1. The slash is also unusual because its bounding box extends to the right of the next character. The width of the slash, measured from origin to origin, is 5; the right bearing, measured from origin to the right edge of the bounding box, is 6.

Figure 8–2 Composition of a Slash



ZK−0289A−GE

The left bearing, right bearing, ascent, descent, and width of a character are **character metrics**. Xlib maintains information about character metrics in a char struct data structure. The following illustrates the data structure:

```
typedef struct {
    short       lbearing;
    short       rbearing;
    short       width;
    short       ascent;
    short       descent;
    unsigned short attributes;
} XCharStruct;
```

Table 8–1 describes members of the char struct data structure. Any member of the data structure can have a negative value, except the attributes member.

Table 8–1 Char Struct Data Structure Members

Member Name	Contents
lbearing	Distance from the origin to the left edge of the bounding box. When the value of this member is zero, the server draws only pixels whose x-coordinates are less than the value of the origin x-coordinate.
rbearing	Distance from the origin to the right edge of the bounding box.
width	Distance from the current origin to the origin of the next character. Text written right-to-left, such as Arabic, uses a negative width to place the next character to the left of the current origin.

(continued on next page)

Table 8–1 (Cont.) Char Struct Data Structure Members

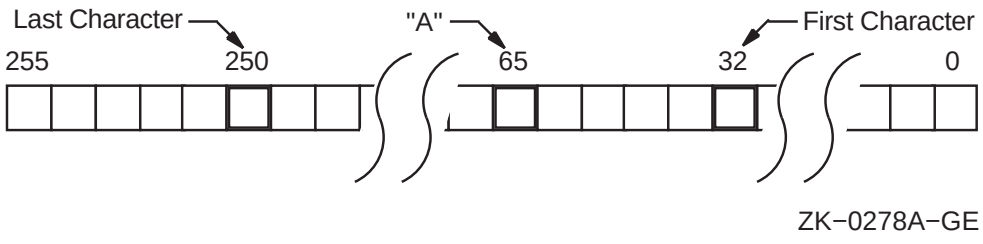
Member Name	Contents
ascent	Distance from the baseline to the top of the bounding box.
descent	Distance from the baseline to the bottom of the bounding box.
attributes	Attributes of the character defined in the bitmap distribution format (BDF) file. A character is not guaranteed to have any attributes.

A **font** is a group of characters that have the same style and size. Xlib supports both fixed and proportional fonts. A **fixed font** has equal metrics. For example, all characters in the font have the same value for left bearing. Consequently, the bounding box for all characters is the same. All metrics in a **proportional font** can vary from character to character. A **monospaced font** is a special type of proportional font in which only the width of all characters must be equal. Bounding boxes of characters in a monospaced font vary depending on the size of characters. If the same font is compiled as a monospaced font and a fixed font, the bounding boxes of the monospaced font are typically smaller than the bounding box that encloses fixed-font characters. For information about compiling fonts, see Appendix A.

Xlib uses indexes to refer to characters that compose a font. The indexes, each defined by a byte, are arranged in one or more rows of up to 256 indexes. A font can contain as many as 256 rows of character indexes, used contiguously. Fonts seldom use all possible indexes.

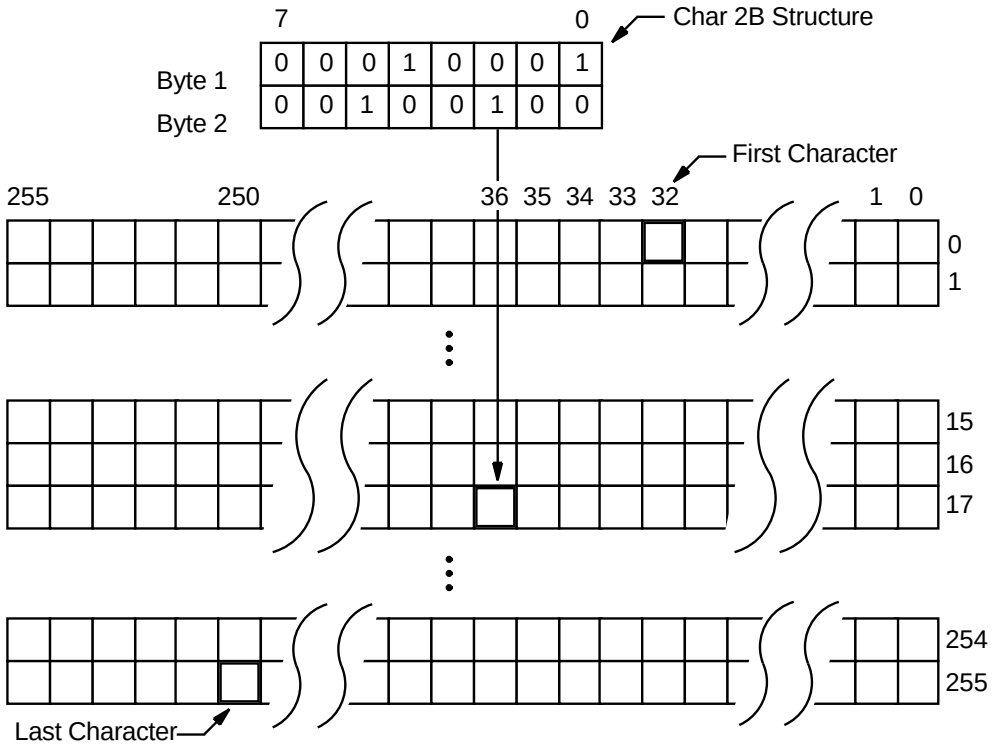
For example, the font illustrated in Figure 8–3, comprises 219 characters in columns 32 through 250, one column for each character index. Columns 0 through 31 and 251 through 255 are undefined. The first character of the font is located at column 32; the last character is located at column 250. Because all characters are defined in one row of 256 indexes, the font is a **single-row font**. In the illustration, character “A” is located at column 65.

Figure 8–3 Single-Row Font



Multiple-row fonts, such as Kanji, comprise more characters than can be indexed by a single row of 256 bytes. Figure 8–4 illustrates the configuration of a multiple-row font. Byte 1 refers to the row. Byte 2 refers to the column in the row. In Figure 8–4, the character is located at column 36 in row 17. Note that each row of a multiple-row font has the same number of undefined bytes at the beginning and end. In each row, characters begin at column 32 and end at column 250.

Figure 8–4 Multiple-Row Font



Xlib provides a char 2B data structure to enable clients to index multiple-row fonts easily. The following illustrates the data structure:

```
typedef struct {
    unsigned char byte1;
    unsigned char byte2;
} XChar2b;
```

Table 8–2 describes members of the data structure.

Table 8–2 Char 2B Data Structure Members

Member Name	Contents
byte1	Row in which the character is indexed
byte2	Position of the character in the row

Xlib maintains a record of the characteristics of a font in the font struct data structure. The following illustrates the font struct data structure:

```
typedef struct {
    XExtData    *ext_data;
    Font        fid;
    unsigned    direction;
    unsigned    min_char_or_byte2;
    unsigned    max_char_or_byte2;
    unsigned    min_byte1;
    unsigned    max_byte1;
    Bool        all_chars_exist;
    unsigned    default_char;
    int         n_properties;
    XFontProp   *properties;
    XCharStruct min_bounds;
    XCharStruct max_bounds;
    XCharStruct *per_char;
    int         ascent;
    int         descent;
} XFontStruct;
```

Table 8–3 describes members of the data structure.

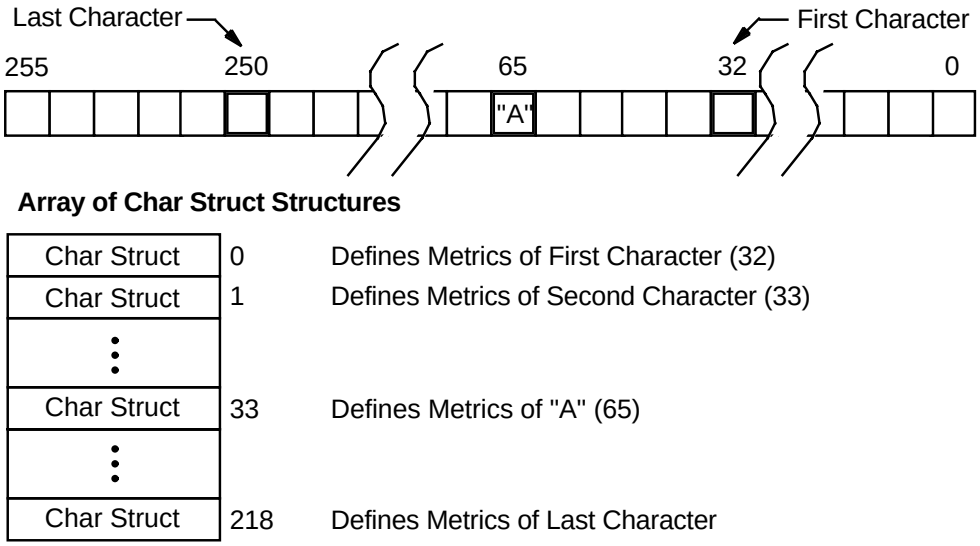
Table 8–3 Font Struct Data Structure Members

Member Name	Contents
ext_data	Data used by extensions.
fid	Identifier of the font.
direction	Hint about the direction in which the font is painted. The direction can be either left-to-right, specified by the constant FontLeftToRight, or right-to-left, specified by the constant FontRightToLeft. The core protocol does not support vertical text.
min_char_or_byte2	First character in the font.
max_char_or_byte2	Last character in the font.
min_byte1	First row that exists.
max_byte1	Last row that exists.
all_chars_exist	If the value of this member is true, all characters in the array pointed to by per_char have nonzero bounding boxes.
default_char	Character used when an undefined or nonexistent character is printed.
n_properties	Number of properties associated with the font.
properties	Address of an array of additional font properties.
min_bounds	The minimum bounding box value of all the elements in the array of char struct data structures that define each character in the font. For a description of the use of min_bounds, see max_bounds.
max_bounds	Maximum metrics values of all the characters in the font.
per_char	Address of an array of char struct data structures that define each character in the font.
ascent	Distance from the baseline to the top of the bounding box. With descent, ascent is used to determine line spacing.
descent	Distance from the baseline to the bottom of the bounding box. With ascent, descent is used to determine line spacing.

As Table 8–3 indicates, Xlib records metrics for each character in an array of char struct data structures specified by the font struct per_char member. The array

comprises as many char struct data structures as there are characters in the font. However, the indexes that refer to the location of characters in the array differ from the indexes to characters in the font. For example, 32 indexes the first character of the font illustrated in Figure 8–5, whereas 0 indexes its char struct data structure in the array.

Figure 8–5 Indexing Single-Row Font Character Metrics



To locate the char struct data structure that defines the metrics of any character in a single-row font, subtract the value of the column that indexes the first character in the font, specified by min_char_or_byte2, from the position of the character. For instance, in Figure 8–5 the metrics of character “A” are located at index 33 in the array of char struct data structures specified by the per_char member.

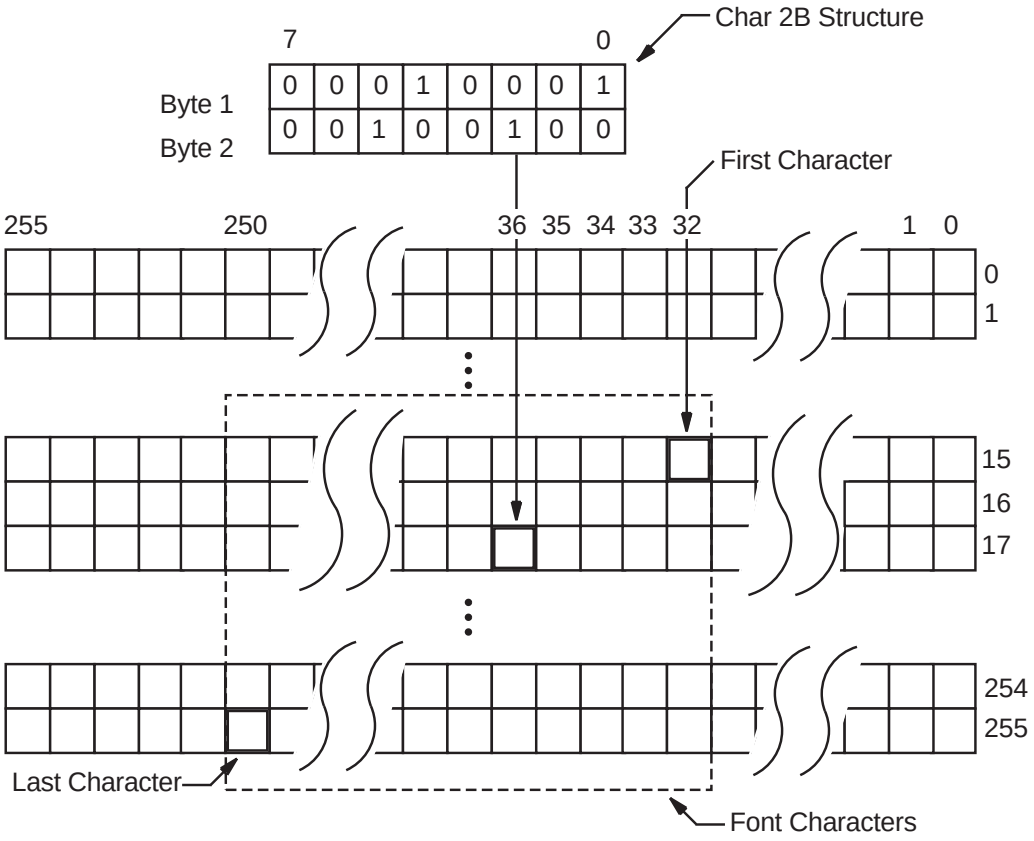
To locate the char struct data structure that defines the metrics of a character of a multiple-row font, use the following formula to adjust for both the number of rows in the font and the position of the character in a row:

$$(row - first\ row\ of\ characters) * N + (position\ in\ column - first\ column)$$

N is equal to the last column minus the first column plus 1.

For example, the array index of the character specified in Figure 8–6 is 442.

Figure 8–6 Indexing Multiple-Row Font Character Metrics



Array of Char Struct Structures

Char Struct	0	Defines Metrics of Character at Row 15, Column 32
⋮		
Char Struct	218	Defines Metrics of Character in Row 15, Column 250
Char Struct	219	Defines Metrics of Character in Row 16, Column 32
⋮		
Char Struct	442	Defines Metrics of Char 2B Character
⋮		
Char Struct	52778	Defines Metrics of Last Character

ZK−0277A−GE

Like windows, fonts may have properties associated with them. However, font properties differ from window properties. Window properties are data associated with windows; font properties describe font characteristics, such as spacing between words. When the font is compiled, its properties are defined in an array of font prop data structures.

Just as atoms name window properties, atoms name font properties. If the atoms are predefined, they have associated literals. For example, the predefined atom that identifies the height of capitalized letters is referred to by the literal XA_CAP_HEIGHT.

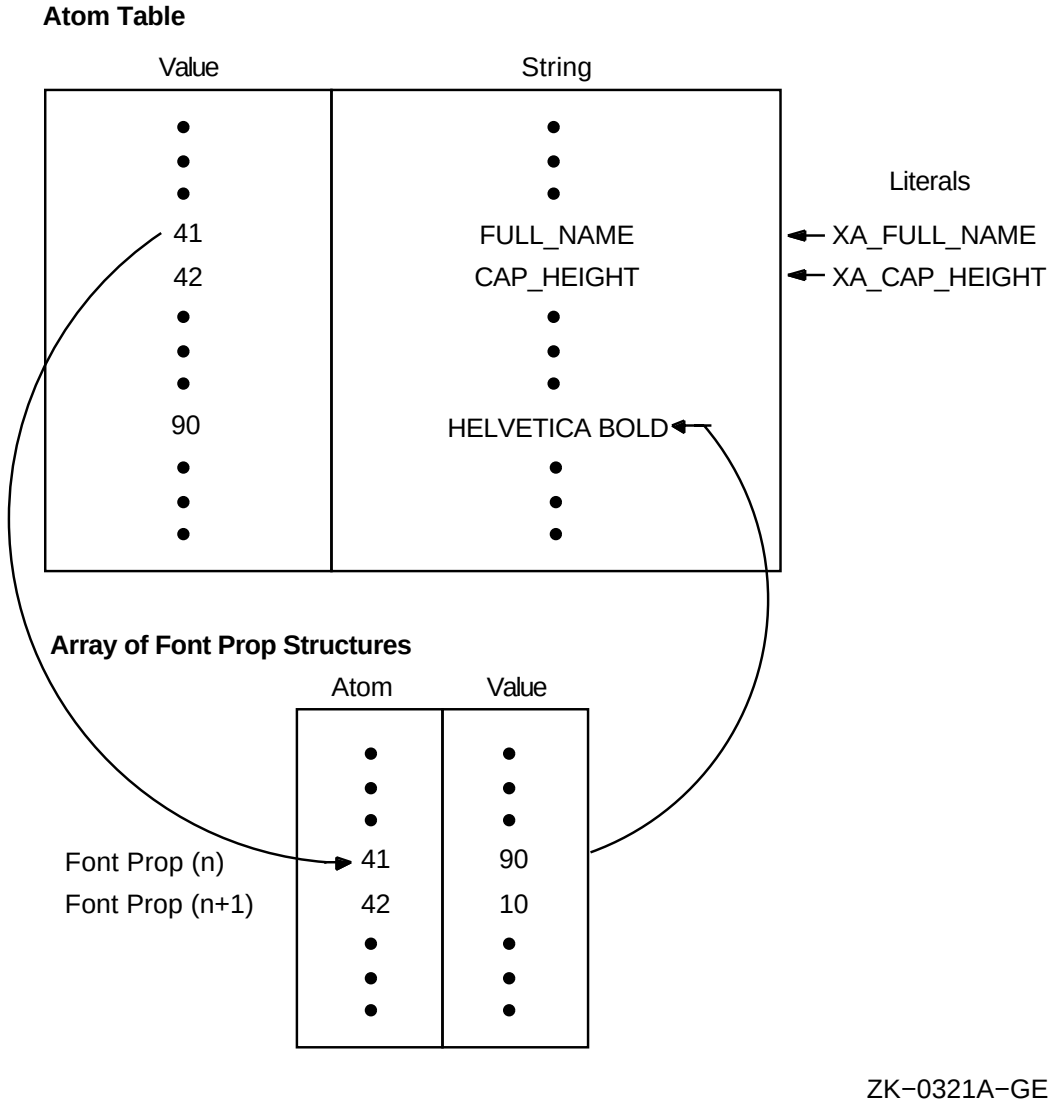
When working with properties, clients must know beforehand how to interpret the font property identified by an atom. Figure 8–7 illustrates this concept.

The server maintains an atom table for font properties. The table associates values with strings. For example, the atom table illustrated in Figure 8–7 defines two atoms. One associates the string FULL_NAME with the value 41. The other associates the string CAP_HEIGHT with the value 42. Notice that the string in the atom table is different from XA_FULL_NAME, the literal that refers to the atom.

Both atoms uniquely identify different types of data. FULL_NAME identifies string data that names the font. CAP_HEIGHT identifies integer data that defines the height of capitalized letters.

Although the atoms identify different types of data, the property table illustrated in Figure 8–7 associates both atoms with integers. The integer associated with CAP_HEIGHT defines without further interpretation the height of capitalized letters. However, the integer listed with FULL_NAME is an atom value. This integer, 90, corresponds to a value in the atom table that has an associated string, HELVETICA BOLD. To use the string, the client must know that the value associated with the atom is itself an atom value.

Figure 8–7 Atoms and Font Properties



Xlib lists each font property and its corresponding atom in a font prop data structure. The property value table in Figure 8–7 is an array of font prop data structures.

The following illustrates the data structure:

```
typedef struct {
    Atom name;
    unsigned long card32;
} XFontProp;
```

Table 8–4 describes members of the data structure.

Table 8–4 Font Prop Data Structure Members

Member Name	Contents
name	String of characters that names the property
card32	A 32-bit value that defines the font property

8.2 Specifying Fonts

To specify a font for writing text, first load the font and then associate the loaded font with a graphics context. The font files are stored in:

- DECW$SYSCOMMON:[SYSFONT.DECW.75DPI]
- DECW$SYSCOMMON:[SYSFONT.DECW.100DPI]
- DECW$SYSCOMMON:[SYSFONT.DECW.COMMON]

Appendix C lists VMS DECwindows font names.

To load a font, use either the LOAD FONT or the LOAD QUERY FONT routine. LOAD FONT loads the specified font and returns a font identifier. LOAD QUERY FONT loads the specified font and returns information about the font to a font struct data structure.

Because LOAD QUERY FONT returns information to a font struct data structure, calling the routine takes significantly longer than calling LOAD FONT, which returns only the font identifier.

When using either routine, pass the display identifier and font name. Xlib font names consist of the following fields, in left-to-right order:

1. Foundry that supplied the font, or the font designer
2. Typeface family of the font
3. Weight (Book, Demi, Medium, Bold, Light)
4. Slant (R (roman), I (italic), O (oblique))
5. Width per horizontal unit of the font (Normal, Wide, Double Wide, Narrow)
6. Additional style font identifier
7. Pixel font size
8. Point size (80, 100, 120, 140, 180, 240) in decipoints (for example, 120 means 12.0 points)
9. X resolution in pixels (dots) per inch
10. Y resolution in pixels (dots) per inch
11. Spacing (M (monospaced), P (proportional), or C (character cell))
12. Average width of all characters in the font in decipixels
13. Character set registry
14. Character set encoding

For more information about font names, see the X Logical Font Description (XLFD) in the *X Window System*.

The full XLFD name of a representative font is as follows:

```
-Adobe-ITC Avant Garde Gothic-Book-R-Normal--14-100-100-100-P-80-ISO8859-1
```

The font foundry is Adobe. The font family is ITC Avant Garde Gothic. Font weight is Book, font slant is R (roman), and width between font units is Normal.

The pixel size is 14 and the decipoint size is 100. (The actual point size is 10.)

Horizontal and vertical resolution in dots per inch (dpi) is 100. When the dpi is 100, 14 pixels are required to be a 10-point font.

The font is proportionally spaced. Average width of characters is 80 decipixels. Character encoding is ISO Latin-1.

The following designates the full XLFD name of the comparable font designed for a 75 dpi monitor:

```
-Adobe-ITC Avant Garde Gothic-Book-R-Normal--10-100-75-75-P-59-ISO8859-1
```

Unlike the previous font, this font requires only 10 pixels to be 10 points. Note that this font differs from the previous font only in pixel size, resolution, and average character width.

Xlib enables clients to substitute a question mark for a single character and an asterisk (*) for one or more fields in a font name. The following illustrates using the asterisk to specify a 10-point ITC Avant Garde Gothic font of book weight, roman style, and normal spacing for display on either 75 or 100 dpi monitors:

```
-Adobe-ITC Avant Garde Gothic-Book-R-Normal--*-100-*-*-P-*-ISO8859-1
```

See Section 8.7 for more information about using asterisks in font names.

The following example illustrates loading the 10-point font:

```
#define FontName\
"-Adobe-ITC Avant Garde Gothic-Book-R-Normal--*-100-*-*-P-*-ISO8859-1"
                        .
                        .
                        .
font = XLoadFont(dpy, FontName);
                        .
                        .
                        .
```

After loading a font, associate it with a graphics context by calling the SET FONT routine. Specify the font identifier that either LOAD FONT or LOAD QUERY FONT returned and a graphics context, as in the following example:

```
XSetFont(dpy, gc, font);
```

The call associates *font* with *gc*.

When loading fonts, note that the LOAD FONT routine is an asychronous routine and does not return an error if the call is unsuccessful. Use one of the following three methods to determine the validity of the font id:

- Force the error by calling the SYNC routine and using an error handler. (For more information about this method, refer to Section 9.13.3.)
- Check that the font exists by calling the LIST FONTS routine, and load the font by calling the LOAD FONT routine.

- Use the LOAD QUERY FONT routine. LOAD QUERY FONT is a synchronous routine that loads the font, returns a pointer to a font struct data structure, and checks that the call is successful. However, note that because LOAD QUERY FONT returns information to a font struct data structure, calling the routine takes significantly longer than calling LOAD FONT, which returns only the font identifier.

8.3 Getting Information About a Font

Xlib provides clients with routines that list available fonts, get font information with or without character metrics, and return the value of a specified font property.

To get a list of available fonts, use the LIST FONTS routine, specifying the font searched for.

LIST FONTS returns a list of available fonts that match the specified font name. When the client no longer needs the list of font names, call the FREE FONT NAMES routine to free storage allocated for the font list.

To receive both a list of fonts and information about the fonts, use the LIST FONTS WITH INFO routine. LIST FONTS WITH INFO returns both a list of fonts that match the font specified by the client and the address of a font struct data structure for each font listed. Each data structure contains information about the font. The data structure does not include character metrics in the per_char member. For a description of the information returned, see Table 8–3.

To receive information about a font, including character metrics, use the QUERY FONT routine. Because the server returns character metrics, calling QUERY FONT takes approximately eight times longer than calling LIST FONTS WITH INFO. To get the value of a specified property, use the GET FONT PROPERTY routine.

Although a font is not guaranteed to have any properties, it should have at least the properties described in Table 8–5. The table lists properties by atom name and data type. For information about properties, see Section 3.5.

Table 8–5 Atom Names of Font Properties

Atom	Data Type	Description of the Property
XA_MIN_SPACE	unsigned	Minimum spacing between words, in pixels.
XA_NORMAL_SPACE	unsigned	Normal spacing between words, in pixels.
XA_MAX_SPACE	unsigned	Maximum spacing between words, in pixels.
XA_END_SPACE	unsigned	Additional spacing at the end of a sentence, in pixels.

XA_SUPERSCRIPT_X	signed	With XA_SUPERSCRIPT_Y, the offset from the character origin where superscripts should begin, in pixels. If the origin is [x, y], superscripts should begin at the following coordinates:
		`x + XA_SUPERSCRIPT_X,`
		`y - XA_SUPERSCRIPT_Y`

(continued on next page)

Table 8–5 (Cont.) Atom Names of Font Properties

Atom	Data Type	Description of the Property
XA_SUPERSCRIPT_Y	signed	With XA_SUPERSCRIPT_X, the offset from the character origin where superscripts should begin, in pixels. See the description under XA_ SUPERSCRIPT_X.

XA_SUBSCRIPT_X	signed	With XA_SUBSCRIPT_Y, the offset from the character origin where subscripts should begin, in pixels. If the origin is [x, y], subscripts should begin at the following coordinates:
		
		`x + XA_SUBSCRIPT_X,`
		`y + XA_SUBSCRIPT_Y`

XA_SUBSCRIPT_Y	signed	With XA_SUBSCRIPT_X, the offset from the character origin where subscripts should begin, in pixels. See the description under XA_ SUBSCRIPT_X.
XA_UNDERLINE_POSITION	signed	The y offset from the baseline to the top of an underline, in pixels. If the baseline y-coordinate is y, then the top of the underline is at y + XA_ UNDERLINE_POSITION.
XA_UNDERLINE_THICKNESS	unsigned	Thickness of the underline, in pixels.

XA_STRIKEOUT_ASCENT	signed	With XA_STRIKEOUT_DESCENT, the vertical extent for boxing or voiding characters, in pixels. If the baseline y-coordinate is y, the top of the strikeout box is y - XA_STRIKEOUT_ASCENT. The height of the box is as follows:
		
		`XA_STRIKEOUT_ASCENT +`
		`XA_STRIKEOUT_DESCENT`

XA_STRIKEOUT_DESCENT	signed	With XA_STRIKEOUT_ASCENT, the vertical extent for boxing or voiding characters, in pixels. See the description under XA_STRIKEOUT_ ASCENT.
XA_ITALIC_ANGLE	signed	The angle of the dominant staffs of characters in the font, in degrees scaled by 64, relative to the 3 o'clock position from the character origin. Positive values indicate counterclockwise motion.
XA_X_HEIGHT	signed	One ex, as in TeX, but expressed in units of pixels. Often the height of lowercase x.
XA_QUAD_WIDTH	signed	One em, as in TeX, but expressed in units of pixels. Often the width of the digits 0 to 9.
XA_CAP_HEIGHT	signed	The y offset from the baseline to the top of capital letters, ignoring ascents. If the baseline y-coordinate is y, the top of the capitals is at y - XA_CAP_HEIGHT.
XA_WEIGHT	unsigned	Weight or boldness of the font, expressed as a value between 0 and 1000.
XA_POINT_SIZE	unsigned	Point size of the font at ideal resolution, expressed in 1/10 points.
XA_RESOLUTION	unsigned	Number of pixels per point, expressed in 1/100, at which the font was created.

(continued on next page)

Table 8–5 (Cont.) Atom Names of Font Properties

Atom	Data Type	Description of the Property
XA_COPYRIGHT	unsigned	Copyright date of the font.
XA_NOTICE	unsigned	Copyright date of the font name.
XA_FONT_NAME	atom	Font name. For example: `-Adobe-Helvetica-Bold-R-Normal--10-100-75-75-P-60-ISO8859-1`
XA_FAMILY_NAME	atom	Name of the font family. For example: `Helvetica`
XA_FULL_NAME	atom	Full name of the font. For example: `Helvetica Bold`

8.4 Freeing Font Resources

Because allocating fonts requires large amounts of memory, it is important to deallocate these resources when the client no longer needs them. Table 8–6 lists complimentary font routines and the result when the deallocating routine is called.

Table 8–6 Complimentary Font Routines

Allocating Routine	Deallocating Routine	Result
LOAD FONT	UNLOAD FONT	Deletes the association between the font resource ID and the specified font and unloads it from server memory
LOAD QUERY FONT	FREE FONT	Frees the client-side storage used by the font structure
	UNLOAD FONT	Unloads the font from server memory
LIST FONTS	FREE FONT NAMES	Frees the array and strings returned by LIST FONTS
LIST FONTS WITH INFO	FREE FONT NAMES	Frees the array and strings returned by LIST FONTS WITH INFO
	FREE FONT INFO	Frees the font information array

8.5 Computing the Size of Text

Use the TEXT WIDTH and TEXT WIDTH 16 routines to compute the width of 8-bit and 2-byte strings, respectively. The routines return the sum of the width of each character in the specified string. To compute the bounding box of a specified 8-bit string, use either the TEXT EXTENTS or QUERY TEXT EXTENTS routine. Both TEXT EXTENTS and QUERY TEXT EXTENTS return the direction hint, ascent, descent, and overall size of the character string being queried.

TEXT EXTENTS passes to Xlib the font struct data structure returned by a previous call to either LOAD QUERY FONT or QUERY FONT. QUERY TEXT EXTENTS queries the server for font information, which the server returns to a font struct data structure. Because Xlib can process TEXT EXTENTS locally, without querying the server for font metrics, calling TEXT EXTENTS is significantly faster than calling QUERY TEXT EXTENTS.

To compute the bounding boxes of a specified 2-byte string, use either the TEXT EXTENTS 16 or the QUERY TEXT EXTENTS 16 routine. Both routines return information identical to information returned by TEXT EXTENTS and QUERY TEXT EXTENTS. As with TEXT EXTENTS, calling TEXT EXTENTS 16 is significantly faster than calling QUERY TEXT EXTENTS 16 because Xlib can process the call without making the round-trip to the server.

8.6 Drawing Text

Xlib enables clients to draw text stored in text data structures, text whose foreground bits only are displayed, and text whose foreground and background bits are displayed.

To draw 8-bit or 2-byte text stored in data structures, use either the DRAW TEXT or the DRAW TEXT 16 routine. Xlib includes text item and text item 16 data structures to enable clients to store text. The following illustrates the text item data structure:

```
typedef struct {
    char *chars;
    int nchars;
    int delta;
    Font font;
} XTextItem;
```

Table 8–7 describes members of the text item data structure.

Table 8–7 Text Item Data Structure Members

Member Name	Contents
chars	Address of a string of characters.
nchars	Number of characters in the string.
delta	Horizontal spacing before the start of the string. Spacing is always added to the string origin and is not dependent on the font used.
font	Identifier of the font used to print the string. If the value of this member is None, the server uses the current font in the GC data structure. If the member has a value other than None, the specified font is stored in the GC data structure.

The following illustrates the text item 16 data structure:

```
typedef struct {
    XChar2b *chars;
    int nchars;
    int delta;
    Font font;
} XTextItem16;
```

Table 8–8 describes members of the text item 16 data structure.

Table 8–8 Text Item 16 Data Structure Members

Member Name	Contents
chars	Address of a string of characters stored in a char 2B data structure. For a description of the char 2B data structure, see the description immediately following this table.
nchars	Number of characters in the string.
delta	Horizontal spacing before the start of the string. Spacing is always added to the string origin and is not dependent on the font used.
font	Identifier of the font used to print the string. If the value of this member is None, the server uses the current font in the GC data structure. If the member has a value other than None, the specified font is stored in the GC data structure.

Xlib provides a char 2B data structure to enable clients to store 2-byte text. The following illustrates the data structure:

```
typedef struct {
    unsigned char byte1;
    unsigned char byte2;
} XChar2b;
```

Xlib processes each text item in turn. Each character image, as defined by the font in the graphics context, is treated as an additional mask for a fill operation on the drawable. The drawable is modified only where the font character has a bit set to 1.

Example 8–1 illustrates using the DRAW TEXT routine to draw three words in one call.

Example 8–1 Drawing Text Using the DRAW TEXT Routine

```
                        .
                        .
                        .
#define FirstFont "-Adobe-New Century Schoolbook-Bold-R-Normal--*-80-*-*-P-*-ISO8859-1"
#define SecondFont "-Adobe-New Century Schoolbook-Bold-R-Normal--*-140-*-*-P-*-ISO8859-1"
#define ThirdFont "-Adobe-New Century Schoolbook-Bold-R-Normal--*-240-*-*-P-*-ISO8859-1"

Display *dpy;
Window window;
GC gc;
Screen *screen;
int n;
XTextItem text[] = {
        "small", 5, 0, 0,
        "bigger", 6, 0, 0,
        "biggest", 7, 0, 0,
};
                        .
                        .
                        .
/***** Load the font for text writing *****/
static void doLoadFont( )
{
    Font FontOne, FontTwo, FontThree;
```

(continued on next page)

Example 8–1 (Cont.) Drawing Text Using the DRAW TEXT Routine

```
    FontOne = XLoadFont(dpy, FirstFont);
    FontTwo = XLoadFont(dpy, SecondFont);
    FontThree = XLoadFont(dpy, ThirdFont);
    XSetFont(dpy, gc, FontTwo);

    text[0].delta = 0;
    text[0].font = FontOne;

    text[1].delta = 20;
    text[1].font = FontTwo;

    text[2].delta = 20;
    text[2].font = FontThree;
}
                        .
                        .
                        .
/***** Button press *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
    if (eventP->xbutton.button == Button2) sys$exit (1);
    if (eventP->xbutton.button == Button1)
        XDrawText(dpy, window, gc, 100, 300, text, 3);
}
```

To draw 8-bit or 2-byte text, use the DRAW STRING, DRAW STRING 16, DRAW IMAGE STRING, and DRAW IMAGE STRING 16 routines. DRAW STRING and DRAW STRING 16 display the foreground values of text only. DRAW IMAGE STRING and DRAW IMAGE STRING 16 display both foreground and background values.

Example 8–2 illustrates drawing text with the DRAW STRING routine. The example modifies the sample program in Chapter 1 to draw shadow text.

Example 8–2 Drawing Text Using the DRAW STRING Routine

```
                    .
                    .
                    .
/***** Write the message in the window *****/
static void doExpose(eventP)
XEvent *eventP;
{
    if (eventP->xexpose.window != window2) return;
    XClearWindow(dpy, window2);
    XSetForeground(dpy, gc, doDefineColor(3));
    XDrawString(dpy, window2, gc, 35, 75, message[state], strlen(message[state]));
    XSetForeground(dpy, gc, doDefineColor(4));
    XDrawString(dpy, window2, gc, 31, 71, message[state], strlen(message[state]));
}
```

(continued on next page)

Example 8–2 (Cont.) Drawing Text Using the DRAW STRING Routine

```
/***** Shutdown *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
    if (eventP->xexpose.window != window2) {
    state = 1;
    XClearWindow(dpy, window2);
    XSetForeground(dpy, gc, doDefineColor(3));
    XDrawString(dpy, window2, gc, 35, 75, message[state], strlen(message[state]));
    XSetForeground(dpy, gc, doDefineColor(4));
    XDrawString(dpy, window2, gc, 31, 71, message[state], strlen(message[state]));
    return;
    }

    /* Unmap and destroy windows */

    XUnmapWindow(dpy, window1);
    XDestroyWindow(dpy, window1);

    XCloseDisplay(dpy);

    sys$exit (1);
}
```

The server refers to the following members of the GC data structure when writing text with DRAW TEXT, DRAW TEXT 16, DRAW STRING, and DRAW STRING 16:

Function	Plane mask
Foreground	Subwindow mode
Stipple	Font
Background	Tile
Tile stipple x origin	Tile stipple y origin
Clip x origin	Clip y origin
Clip mask	Fill style

To draw both foreground and background values of text, use the DRAW IMAGE STRING and DRAW IMAGE STRING 16 routines. For example, the sample program uses the DRAW IMAGE routine to write the text "Click here to exit," as follows:

```
int n, state = 0;
char *message[]= {
    "Click here to exit",
    "Click HERE to exit!"
    };
                    .
                    .
                    .
 if (eventP->xexpose.window != window2) return;
 XDrawImageString(dpy, window2, gc, 75, 75, message[state],
     strlen(message[state]));
```

The effect is first to fill a rectangle with the background defined in the graphics context and then to paint the text with the foreground pixel. The upper left corner of the filled rectangle is at $75, (75 - font\ ascent)$. The width of the rectangle is equal to the width of the string. The height of the rectangle is equal to $font\ ascent + font\ descent$.

When drawing text in response to calls to DRAW IMAGE STRING and DRAW IMAGE STRING 16, the server ignores the function and fill style the client has defined in the graphics context. The value of the function member of the GC data structure is effectively the value specified by the constant **GXCopy**. The value of the fill style member is effectively the value specified by the constant **FillSolid**.

The server refers to the following members of the GC data structure when writing text with DRAW IMAGE STRING and DRAW IMAGE STRING 16:

Subwindow mode	Plane mask
Foreground	Background
Stipple	Font
Clip x origin	Clip y origin
Clip mask	

8.7 Font Usage Hints

This section includes information about the Digital font fallback strategy and hints for using font names efficiently.

8.7.1 Font Fallback Strategy

When specifying fonts, the client should use fonts that are common to both DECwindows Motif and X Window System, Version 11, Release 4 software. Using common fonts makes a client application interoperable and enables it to display on a wide variety of third-party workstations and X terminals. The following lists the common font families:

- Courier
- Helvetica
- New Century Schoolbook
- Symbol
- Times

If clients use other font families (such as ITC Avant Garde Gothic, ITC Lubalin Graph, or ITC Souvenir), the DECwindows toolkit provides the DxmFindFontFallback routine that supports the Digital font fallback strategy. For more information about this routine, see the *DECwindows Extensions to Motif*.

Digital recommends that clients not use certain fonts. Table 8–9 lists the font families and the reason why. All other font families are for general use.

Table 8–9 Fonts Not Recommended for General Use

Font Family	Reason
Interim DEC Math	For use only by the DECwindows Bookreader. This font will eventually be phased out.
Menu	For use by the DECwindows Toolkit.
Terminal	For use by terminal emulators.
Fixed	Available for compatibility reasons only. Should not be used by new clients.
Variable	Available for compatibility reasons only. Should not be used by new clients.
Fixed Width	Available for compatibility reasons only. Should not be used by new clients.

8.7.2 Speeding Up Font Name Searches

The DECwindows X server uses a heuristic to speed up font name searching. When the client specifies the FAMILY_NAME, WEIGHT_NAME, SLANT, SETWIDTH_NAME, CHARSET_REGISTRY, and CHARSET_ENCODING fields explicitly, the server uses a hash table to speed up font name searching. For example, the following font name is specified correctly to use the heuristic:

```
-*-Times-Medium-R-Normal--*-140-*-*-P-*-ISO8859-1
```

The previous example will be found more quickly than the following because a wildcard has been used in the SLANT field:

```
-*-Times-Medium-*-Normal--*-140-*-*-P-*-ISO8859-1
```

The client can specify other fields, such as the FOUNDRY field; however, all fourteen hyphens in a font name must be specified for the heuristic to work. The ADD_STYLE_NAME field (the field after Normal in the example) should be left empty because this field may be used in the hashing algorithm in the future.

8.7.3 Monitor Density Independence

To choose a particular sized font without regard to the density of the monitor, the client should always use a wildcard for the PIXEL_SIZE field and never use a wildcard for the POINT_SIZE field. In addition, the client should use a wildcard for the RESOLUTION_X and RESOLUTION_Y fields.

8.7.4 Character Set Considerations

The client should always explicitly specify the CHARSET_REGISTRY and CHARSET_ENCODING fields (for example, ISO8859-1), not only because they speed up font name searching, but because they ensure that the client uses the correct character set. ISO8859-1 specifies the Latin-1 character set that is normally used in text files. There are other possible character sets that could match a wildcard search, such as Latin-2 or Latin-3, but they should not be used if the client can only process and display Latin-1 text.

9
Handling Events

An event is a report of either a change in the state of a device (such as a mouse) or the execution of a routine called by a client. An event can be either unsolicited or solicited. Typically, unsolicited events are reports of keyboard or pointer activity. Solicited events are Xlib responses to calls by clients.

Xlib reports events asynchronously. When any event occurs, Xlib processes the event and sends it to clients that have specified an interest in that type of event.

This chapter describes the following concepts needed to manage events:

- Event processing—An overview of types of events
- Event type selection—A description of how clients can specify the types of events Xlib reports to them
- Event handling—A description of handling specific types of events

This chapter provides information for a subset of event types. For a complete reference of event handling routines and data structures, see the *DECwindows Motif for OpenVMS Guide to Non-C Bindings* and the *X Window System*.

9.1 Event Processing

Apart from errors, which Section 9.13 describes, Xlib events issue from operations on either windows or pixmaps. Most events result from operations associated with windows. The smallest window that contains the pointer when a window event occurs is the **source window**.

Xlib searches the window hierarchy upward from the source window until one of the following applies:

- Xlib finds a window that one or more clients have identified as interested in the event. This window is the **event window**. After Xlib locates an event window, it sends information about the event to appropriate clients.
- Xlib finds a window whose do_not_propagate attribute has been set by a client. Setting this attribute specifies that Xlib should not notify ancestors of the window owned by the client about events occurring in the window and its children. For more information about the do_not_propagate attribute, see Chapter 3.
- Xlib reaches the top of the window hierarchy without finding a window that a client has identified as interested in the event.

While there are many types of window events, events associated with pixmaps occur only when a client cannot compute a destination region because the source region is out-of-bounds (see Chapter 6 for a description of source and destination regions). When a client attempts an operation on an out-of-bounds pixmap region, Xlib puts the event on the event queue and checks a list to determine if a client

is interested in the event. If a client is interested, Xlib sends information to the client using an event data structure.

Xlib can report 30 types of events related to keyboards, mice, windowing, and graphics operations. A flag identifies each type to facilitate referring to the event. Table 9–1 lists event types, grouped by category, and the flags that represent them.

Table 9–1 Event Types

Event Type	Flag Name
Keyboard Events	
Key press	KeyPress
Key release	KeyRelease
Pointer Motion Events	
Button press	ButtonPress
Button release	ButtonRelease
Motion notify	MotionNotify
Window Crossing Events	
Enter notify	EnterNotify
Leave notify	LeaveNotify
Input Focus Events	
Focus in	FocusIn
Focus out	FocusOut
Keymap State Event	
Keymap notify	KeymapNotify
Exposure Events	
Expose	Expose
Graphics expose	GraphicsExpose
No expose	NoExpose

(continued on next page)

Table 9–1 (Cont.) Event Types

Event Type	Flag Name
Window State Events	
Circulate notify	CirculateNotify
Configure notify	ConfigureNotify
Create notify	CreateNotify
Destroy notify	DestroyNotify
Gravity notify	GravityNotify
Map notify	MapNotify
Mapping notify	MappingNotify
Reparent notify	ReparentNotify
Unmap notify	UnmapNotify
Visibility notify	VisibilityNotify
Color Map State Events	
Color map notify	ColormapNotify
Client Communication Events	
Client message	ClientMessage
Property notify	PropertyNotify
Selection clear	SelectionClear
Selection notify	SelectionNotify
Selection request	SelectionRequest

Every event type has a corresponding data structure that Xlib uses to pass information to clients. See the sections that describe handling specific event types for a description of the relevant event-specific data structures.

Xlib includes the any event data structure, which clients can use to receive reports of any type of event. The following illustrates the any event data structure:

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
} XAnyEvent;
```

Table 9–2 describes members of the data structure.

Table 9–2 Any Event Data Structure Members

Member Name	Contents
type	Type of event being reported
serial	Number of the last request processed by the server
send_event	Value defined by the constant true if the event came from a SEND EVENT request
display	Display on which the event occurred
window	Window on which the event report was requested

To enable clients to manage multiple types of events easily, Xlib also includes an event data structure, which is composed of the union of individual event data structures.

The following illustrates the event data structure:

```
typedef union _XEvent {
        int type;
        XAnyEvent xany;
        XKeyEvent xkey;
        XButtonEvent xbutton;
        XMotionEvent xmotion;
        XCrossingEvent xcrossing;
        XFocusChangeEvent xfocus;
        XExposeEvent xexpose;
        XGraphicsExposeEvent xgraphicsexpose;
        XNoExposeEvent xnoexpose;
        XVisibilityEvent xvisibility;
        XCreateWindowEvent xcreatewindow;
        XDestroyWindowEvent xdestroywindow;
        XUnmapEvent xunmap;
        XMapEvent xmap;
        XMapRequestEvent xmaprequest;
        XReparentEvent xreparent;
        XConfigureEvent xconfigure;
        XGravityEvent xgravity;
        XResizeRequestEvent xresizerequest;
        XConfigureRequestEvent xconfigurerequest;
        XCirculateEvent xcirculate;
        XCirculateRequestEvent xcirculaterequest;
        XPropertyEvent xproperty;
        XSelectionClearEvent xselectionclear;
        XSelectionRequestEvent xselectionrequest;
        XSelectionEvent xselection;
        XColormapEvent xcolormap;
        XClientMessageEvent xclient;
        XMappingEvent xmapping;
        XErrorEvent xerror;
        XKeymapEvent xkeymap;
} XEvent;
```

The type member specifies the type of event being reported. For descriptions of the other members of the event data structure, see the section that describes the specific event type.

9.2 Selecting Event Types

Xlib sends information about an event only to clients that have specified an interest in that event type. Clients use one of the following methods to indicate interest in event types:

- By calling the SELECT INPUT routine. SELECT INPUT indicates to Xlib which events to report.
- By specifying event masks when creating a window.
- By specifying event masks when changing window attributes.
- By specifying the graphics exposure mask when creating the graphics context. For more information about specifying a graphics exposure mask, see Chapter 4.

Note that Xlib always reports client messages, mapping notifications, selection clearings, selection notifications, and selection requests.

See the description of the SELECT INPUT routine in the *X Window System* for restrictions on event reporting to multiple clients.

9.2.1 Using the SELECT INPUT Routine

Use the SELECT INPUT routine to specify the types of events Xlib reports to a client. Select event types by passing to Xlib one or more of the masks listed in Table 9–3.

Table 9–3 Event Masks

Event Mask	Event Reported (Event Type)
ButtonMotionMask	At least one button on the pointing device is pressed while the pointer moves (MotionNotify).
Button1MotionMask	Button 1 of the pointing device is pressed while the pointer moves (MotionNotify).
Button2MotionMask	Button 2 of the pointing device is pressed while the pointer moves (MotionNotify).
Button3MotionMask	Button 3 of the pointing device is pressed while the pointer moves (MotionNotify).
Button4MotionMask	Button 4 of the pointing device is pressed while the pointer moves (MotionNotify).
Button5MotionMask	Button 5 of the pointing device is pressed while the pointer moves (MotionNotify).
ButtonPressMask	A button on the pointing device is pressed (ButtonPress).
ButtonReleaseMask	A button on the pointing device is released (ButtonRelease).
ColormapChangeMask	A client installs, changes, or removes a color map (ColormapNotify).
EnterWindowMask	The pointer enters a window (EnterNotify).
ExposureMask	A window becomes visible, a graphics region cannot be computed, a graphics request exposes a region or all sources available, and a no expose event is generated (Expose, GraphicsExpose, NoExpose).
LeaveWindowMask	The pointer leaves a window (LeaveNotify).
FocusChangeMask	The keyboard focus changes (FocusIn, FocusOut).

(continued on next page)

Table 9–3 (Cont.) Event Masks

Event Mask	Event Reported (Event Type)
KeymapStateMask	The key map changes (KeymapNotify).
KeyPressMask	A key is pressed or released (KeyPress, KeyRelease).
OwnerGrabButtonMask	Not applicable.
PointerMotionMask	The pointer moves (MotionNotify).
PointerMotionHintMask	Xlib is free to report only one pointer-motion event (MotionNotify) until one of the following occurs: • Either the key or button state changes. • The pointer leaves the window. • The client calls QUERY POINTER or GET MOTION EVENTS.
PropertyChangeMask	A client changes a property (PropertyNotify).
StructureNotifyMask	One of the following operations occurs on a window: • Circulate (CirculateNotify) • Configure (ConfigureNotify) • Destroy (DestroyNotify) • Move (GravityNotify) • Map (MapNotify) • Reparent (ReparentNotify) • Unmap (UnmapNotify)
SubstructureNotifyMask	One of the following operations occurs on the child of a window: • Circulate (CirculateNotify) • Configure (ConfigureNotify) • Create (CreateNotify) • Destroy (DestroyNotify) • Move (GravityNotify) • Map (MapNotify) • Reparent (ReparentNotify) • Unmap (UnmapNotify)
VisibilityChangeMask	The visibility of a window changes (VisibilityNotify).

The following illustrates using the SELECT INPUT routine:

```
                .
                .
                .
XSelectInput(dpy,win,StructureNotifyMask);
}
```

Clients specify the **StructureNotifyMask** mask to indicate an interest in one or more of the following window operations (see Table 9–3):

Circulating	Configuring
Destroying	Reparenting
Changing gravity	Mapping and unmapping
Moving	

9.2.2 Specifying Event Types When Creating a Window

To specify event types when calling the CREATE WINDOW routine, use the method described in Section 3.2.2 for setting window attributes. Indicate the type of event Xlib reports to a client by doing the following:

1. Set the event_mask window attribute to one or more masks listed in Table 9–3.
2. Specify the event mask flag using the **value_mask** argument of the CREATE WINDOW routine.

Example 9–1 illustrates this method of selecting events. The program specifies that Xlib notify the client of exposure events.

Example 9–1 Selecting Event Types Using the CREATE WINDOW Routine

```
Window window;
                    .
                    .
                    .
static void doCreateWindows( )
{
    int windowW = 400;
    int windowH = 300;
    int windowX = (WidthOfScreen(screen)-window1W)>>1;
    int windowY = (HeightOfScreen(screen)-window1H)>>1;
    XSetWindowAttributes xswa;

    /* Create the window1 window */

❶   xswa.event_mask = ExposureMask;

❷   window = XCreateWindow(dpy, RootWindowOfScreen(screen),
        windowX, windowY, windowW, windowH, 0,
        DefaultDepthOfScreen(screen), InputOutput,
        DefaultVisualOfScreen(screen), CWEventMask, &xswa);
}
```

❶ Set the event mask of the set window attributes data structure to indicate interest in exposure events.

❷ The window attribute is referred to by the constant **CWEventMask**, which specifies the attribute.

9.2.3 Specifying Event Types When Changing Window Attributes

To specify one or more event types when changing window attributes, use the method described in Section 3.7 for changing window attributes. Indicate an interest in event types by doing the following:

1. Set the event_mask window attribute to one or more masks listed in Table 9–3.

2. Specify the event mask flag using the **value_mask** argument of the CHANGE WINDOW ATTRIBUTES routine.

The following illustrates this method:

```
                .
                .
                .
xswa.event_mask = StructureNotify;

XChangeWindowAttributes(dpy, win, CWEventMask, &xswa);
```

9.3 Pointer Events

Xlib reports pointer events to interested clients when the button on the pointing device is pressed or released or when the pointer moves.

This section describes how to handle the following pointer events:

- Pressing a button on the pointing device
- Releasing a button on the pointing device
- Moving the pointing device

The section also describes the button event and motion event data structures.

9.3.1 Handling Button Presses and Releases

To receive event notification of button presses and releases, pass the window identifier and either the **ButtonPressMask** mask or the **ButtonReleaseMask** mask when using the selection method described in Section 9.2.

When a button is pressed, Xlib searches for ancestors of the event window from the root window down to determine whether or not a client has specified a passive grab, an exclusive interest in the button. If Xlib finds no passive grab, it starts an active grab, reserving the button for the sole use of the client receiving notification of the event. Xlib also sets the time of the last pointer grab to the current server time. The effect is the same as calling the GRAB BUTTON routine with argument values listed in Table 9–4.

Table 9–4 Values Used for Grabbing Buttons

Argument	Value
window_id	Event window.
event_mask	Client pointer motion mask.
pointer_mode	Value specified by the constant GrabModeAsync.
keyboard_mode	Value specified by the constant GrabModeAsync.
owner_events	True, if the owner has selected OwnerGrabButtonMask. Otherwise, false.
confine_to	None.
cursor	None.

Refer to Chapter 11 for more information about using grabs.

Xlib uses the button event data structure to report button presses and releases. The following illustrates the data structure:

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        Window root;
        Window subwindow;
        Time time;
        int x, y;
        int x_root, y_root;
        unsigned int state;
        unsigned int button;
        Bool same_screen;
} XButtonEvent;
typedef XButtonEvent XButtonPressedEvent;
typedef XButtonEvent XButtonReleasedEvent;
```

Table 9–5 describes members of the button event data structure. Note that Xlib defines the button pressed event and button released event data structures as a type button event.

Table 9–5 Button Event Data Structure Members

Member Name	Contents
type	Type of event reported. The event type can be either ButtonPress or ButtonRelease.
serial	Number of the last request processed by the server.
send_event	Value defined by the constant true if the event came from a SEND EVENT request.
display	Address of the display on which the event occurred.
window	Event window.
root	Root window in which the event occurred.
subwindow	Source window in which the event occurred.
time	Time in milliseconds at which the event occurred.
x	The x value of the pointer coordinates in the source window.
y	The y value of the pointer coordinates in the source window.
x_root	The x value of the pointer coordinates relative to the root window.
y_root	The y value of the pointer coordinates relative to the root window.

(continued on next page)

Table 9–5 (Cont.) Button Event Data Structure Members

Member Name	Contents	
state	State of the button just prior to the event. Xlib can set this member to the bitwise OR of one or more of the following masks:	
	Button1Mask	Button2Mask
	Button3Mask	Button4Mask
	Button5Mask	Mod1Mask
	Mod2Mask	Mod3Mask
	Mod4Mask	Mod5Mask
button	Buttons that changed state. Xlib can set this member to one of the following values:	
	Button1	Button2
	Button3	Button4
	Button5	
screen	Indicates whether or not the event window is on the same screen as the root window.	

Example 9–2 illustrates the button press event handling routine of the sample program described in Chapter 1. The program calls shutdown routines when the user clicks the mouse button in *window2*.

Xlib removes the next event from the event queue and copies it into an event data structure. The program executes one of two routines, depending on the flag returned in the event data structure type field. Xlib indicates an exposure event by setting the Expose flag in the type field; it indicates a button press event by setting the ButtonPress flag.

When creating *window1* and *window2*, the client indicated an interest in exposures and button presses by setting the event mask field of the set window attributes data structure, as follows:

```
XSetWindowAttributes xswa;
                .
                .
                .
xswa.event_mask = ExposureMask | ButtonPressMask;
```

For more information about selecting event types, see Section 9.2.

The event data structure includes other data structures Xlib uses to report information about various kinds of events. The client-defined *doButtonPress* routine checks the window field of one of these data structures (the expose event data structure) to determine whether or not the server has mapped *window2*.

If the server has mapped *window2*, the client calls a series of shutdown routines when the user presses the mouse button.

Example 9–2 Handling Button Presses

```
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case ButtonPress:           doButtonPress(&event); break;
        }
    }
}
                    .
                    .
                    .
static void doButtonPress(eventP)
XEvent *eventP;
{
        if (eventP->xexpose.window != window2) {
        state =1;
        XDrawImageString(dpy, child, gc, 75, 75, message[state],
            strlen(message[state]));
        return;
    }

    /***** Unmap and destroy windows *****/

    XUnmapWindow(dpy, window1);
    XDestroyWindow(dpy, window1);

    XCloseDisplay(dpy);

    sys$exit (1);
}
```

9.3.2 Handling Pointer Motion

To only receive pointer motion events when a specified button is pressed, pass the window identifier and one of the following masks when using the selection method described in Section 9.2:

ButtonMotionMask	Button1MotionMask
Button2MotionMask	Button3MotionMask
Button4MotionMask	Button5MotionMask

Xlib reports pointer motion events to interested clients whenever the pointer moves and the movement begins and ends in the window. Spatial and temporal resolution of the events is not guaranteed, but clients are assured they will receive at least one event when the pointer moves and then rests. The following illustrates the data structure Xlib uses to report these events:

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        Window root;
        Window subwindow;
        Time time;
        int x, y;
        int x_root, y_root;
        unsigned int state;
        char is_hint;
        Bool same_screen;
} XMotionEvent;
typedef XMotionEvent XPointerMovedEvent;
```

Table 9–6 describes members of the motion event data structure. Note that Xlib defines the pointer moved event data structure as type motion event.

Table 9–6 Motion Event Data Structure Members

Member Name	Contents
type	Type of event reported. The member can have only the value specified by the constant MotionNotify.
serial	Number of the last request processed by the server.
send_event	Value defined by the constant true if the event came from a SEND EVENT request.
display	Address of the display on which the event occurred.
window	Event window.
root	Root window in which the event occurred.
subwindow	Source window in which the event occurred.
time	Time in milliseconds at which the event occurred.
x	The x value of the pointer coordinates in the source window.
y	The y value of the pointer coordinates in the source window.
x_root	The x value of the pointer coordinates relative to the root window.
y_root	The y value of the pointer coordinates relative to the root window.
state	State of the button just prior to the event. Xlib can set this member to the bitwise OR of the following masks: Button1Mask Button2Mask Button3Mask Button4Mask Button5Mask Mod1Mask Mod2Mask Mod3Mask Mod4Mask Mod5Mask
is_hint	Indicates that motion hints are active. No other events reported until pointer moves out of window.
same_screen	Indicates whether or not the event window is on the same screen as the root window.

Example 9–3 illustrates pointer motion event handling.

Example 9–3 Handling Pointer Motion

```
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case MotionNotify:          doMotionNotify(&event); break;
            case ButtonPress:           sys$exit(1);
        }
    }
}
                          .
                          .
                          .
static void doMotionNotify(eventP)
XEvent *eventP;
{
    int x = eventP->xmotion.x;
    int y = eventP->xmotion.y;
    int width = 5;
    int length = 5;

    XFillRectangle(dpy, win, gc, x, y, width, length);
}
```

Each time the pointer moves, the program draws a small filled rectangle at the resulting *x* and *y* coordinates.

To receive pointer motion events, the client specifies the **MotionNotify** flag when removing events from the queue. The client indicated an interest in pointer motion events when creating window *win*, as follows:

```
    xswa.event_mask = ExposureMask | ButtonPressMask |
        PointerMotionMask;

    win = XCreateWindow(dpy, RootWindowOfScreen(screen),
        winX, winY, winW, winH, 0,
        DefaultDepthOfScreen(screen), InputOutput,
        DefaultVisualOfScreen(screen), CWEventMask, &xswa);
```

The server reports pointer movement. Xlib records the resulting position of the pointer in a motion data structure, one of the event data structures that constitute the event data structure. The client-defined *doMotionNotify* routine determines the origin of the filled rectangle it draws by referring to the motion event data structure *x* and *y* members.

9.4 Window Entries and Exits

Xlib reports window entries and exits to interested clients when one of the following occurs:

- The pointer moves into or out of a window due to either pointer movement or to a change in window hierarchy. This is normal window entry and exit.
- A client calls WARP POINTER, which moves the pointer to any specified point on the screen.

- A client calls CHANGE ACTIVE POINTER GRAB, GRAB KEYBOARD, GRAB POINTER, or UNGRAB POINTER. This is **pseudomotion**, which simulates window entry or exit without actual pointer movement.

To receive event notification of window entries and exits, pass the window identifier and either the **EnterWindowMask** mask or the **LeaveWindowMask** mask when using the selection method described in Section 9.2.

Xlib uses the crossing event data structure to report window entries and exits. The following illustrates the data structure:

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        Window root;
        Window subwindow;
        Time time;
        int x, y;
        int x_root, y_root;
        int mode;
        int detail;
        Bool same_screen;
        Bool focus;
        unsigned int state;
} XCrossingEvent;
typedef XCrossingEvent XEnterWindowEvent;
typedef XCrossingEvent XLeaveWindowEvent;
```

Table 9–7 describes members of the crossing event data structure. Note that Xlib defines the enter window event and leave window event data structures as type crossing event.

Table 9–7 Crossing Event Data Structure Members

Member Name	Contents
type	Value defined by either the EnterNotify or the LeaveNotify constant.
serial	Number of the last request processed by the server.
send_event	The value defined by the constant true if the event came from a SEND EVENT request.
display	Address of the display on which the event occurred.
window	Event window.
root	Root window on which the event occurred.
subwindow	Source window in which the event occurred.
time	Time in milliseconds at which the key event occurred.
x	The x value of the pointer coordinates in the source window.
y	The y value of the pointer coordinates in the source window.
x_root	The x value of the pointer coordinates relative to the root window.
y_root	The y value of the pointer coordinates relative to the root window.

(continued on next page)

Table 9–7 (Cont.) Crossing Event Data Structure Members

Member Name	Contents
mode	Indicates whether the event is normal or pseudomotion. Xlib can set this member to the value specified by the constants NotifyNormal, NotifyGrab, and NotifyUngrab. See Section 9.4.1 and Section 9.4.2 for descriptions of normal and pseudomotion events.
detail	Indicates which windows Xlib notifies of the window entry or exit event. Xlib can specify one of the following constants:
	NotifyAncestor NotifyVirtual
	NotifyInferior NotifyNonlinear
	NotifyNonlinearVirtual
same_screen	Indicates whether or not the event window is on the same screen as the root window.
focus	Specifies whether the event window or an inferior is the focus window. If true, the event window is the focus window. If false, an inferior is the focus window.
state	State of buttons and keys just prior to the event. Xlib can return values specified by the following constants:
	Button1Mask Button2Mask
	Button3Mask Button4Mask
	Button5Mask Mod1Mask
	Mod2Mask Mod3Mask
	Mod4Mask Mod5Mask
	ShiftMask ControlMask
	LockMask

9.4.1 Normal Window Entries and Exits

A normal window entry or exit event occurs when the pointer moves from one window to another due to either a change in window hierarchy or the movement of the pointer. In either case, Xlib sets the mode member of the crossing event data structure to the constant **NotifyNormal**.

If the pointer leaves or enters a window as a result of one of the following changes in window hierarchy, Xlib reports the event after reporting the hierarchy event:

Mapping Unmapping

Configuring Circulating

Changing gravity

Xlib can report a window entry or exit event caused by changes in focus, visibility, and exposure either before or after reporting these events.

See the *X Window System* for a description of the events that Xlib reports when the pointer moves from window A to window B as a result of normal window entry or exit.

Example 9–4 illustrates window entry and exit event handling. The program changes the color of a window when the pointer enters or leaves the window.

Figure 9–1 shows the resulting output.

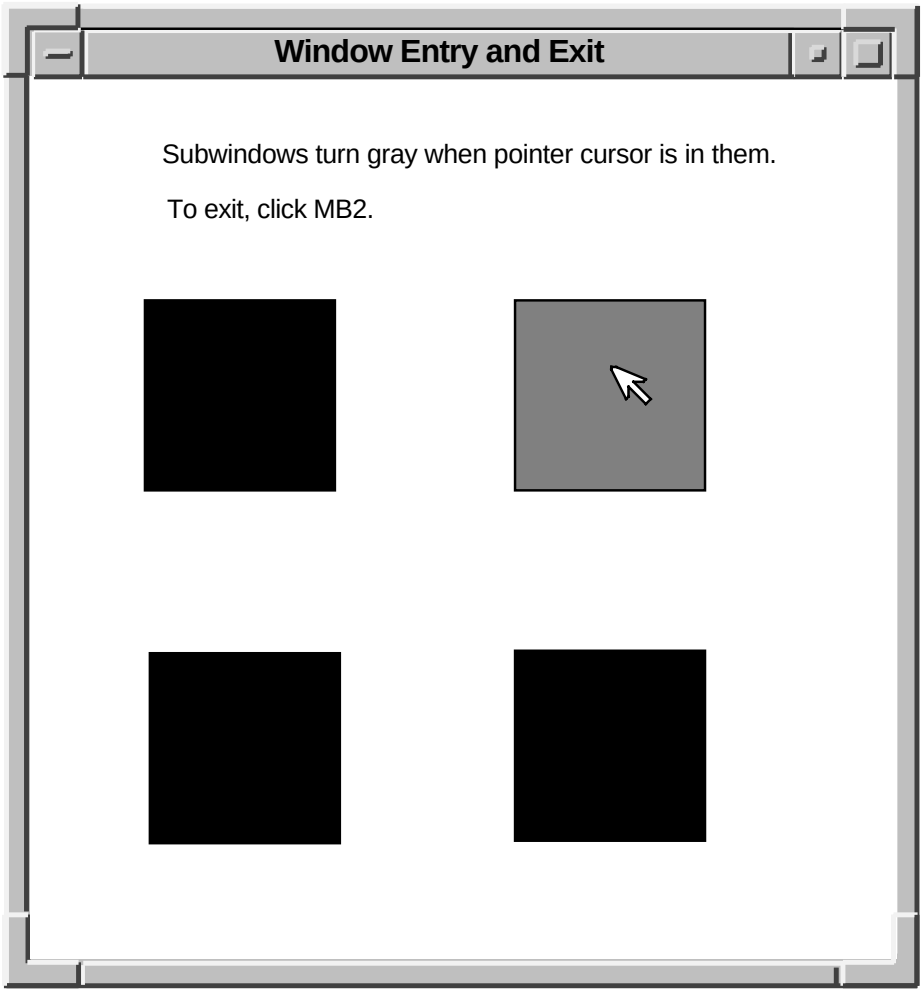
Example 9–4 Handling Window Entries and Exits

```
    /* Create windows win, subwin1,                *
     * subwin2, subwin3, and                       *
     * subwin4  on                                 *
     *  display dpy, defined as follows:           *
     *     #define windowWidth      600            *
     *     #define windowHeight     600            *
     *     #define subwindowWidth  120             *
     *     #define subwindowHeight 120             *
     *     win position: x = 100,y = 100           *
     *     subwin1 position: x = 120,y = 120       *
     *     subwin2 position: x = 360,y = 120       *
     *     subwin3 position: x = 120,y = 360       *
     *     subwin1 position: x = 360,y = 360       */
                          .
                          .
                          .
/**** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case ButtonPress:           sys$exit(1);
            case EnterNotify:           doEnterNotify(&event); break;
            case LeaveNotify:           doLeaveNotify(&event); break;
        }
    }
}
                          .
                          .
                          .
/***** Change window color when pointer enters window *****/
static void doEnterNotify(eventP)
XEvent *eventP;
{
❶   Window window = eventP->xcrossing.window;
    XSetWindowBackground(dpy, window, doDefineColor(4));
❷   XClearArea(dpy, window, 0, 0, subwindowWidth, subwindowHeight, 0);
    return;
}

/***** Change window color when pointer leaves window *****/
static void doLeaveNotify(eventP)
XEvent *eventP;
{
    Window window = eventP->xcrossing.window;
    XSetWindowBackground(dpy, window, doDefineColor(2));
    XClearArea(dpy, window, 0, 0, subwindowWidth, subwindowHeight, 0);
    return;
}
```

❶ Xlib gives the window identifier in the crossing event data structure window field. This occurs when the pointer cursor enters the new window. The program uses the identifier to define the window background and clear the window.

❷ The CLEAR AREA routine clears the window and repaints it with the newly defined window background.

Figure 9–1 Window Entries and Exits



ZK–2509A–GE

9.4.2 Pseudomotion Window Entries and Exits

Pseudomotion window entry and exit events occur when the pointer cursor moves from one window to another due to activating or deactivating a pointer grab.

Xlib reports a pseudomotion window entry if a client grabs the pointer, causing the pointer cursor to change from one window to another even though the pointer cursor has not moved. For example, if the pointer cursor is in window A and a client maps window B over window A, the pointer cursor changes from being in window A to being in window B. If possible, the pointer cursor remains in the same position on the screen. When the placement of the two windows prevents the pointer cursor from maintaining the same position, the pointer cursor moves to the location closest to its original position.

Clients can grab pointers actively by calling the GRAB POINTER routine or passively by calling the GRAB BUTTON routine. Whether the grab is active or passive, Xlib sets the following members of the crossing event data structure to the indicated constants after the pointer cursor moves from one window to another:

- Type member—EnterNotify
- Mode member—NotifyGrab

When a client passively grabs the pointer by calling the GRAB BUTTON routine, Xlib reports a button press event after reporting the pointer grab.

Xlib reports a pseudomotion window exit when a client deactivates a pointer grab, causing the pointer cursor to change from one window to another even though the pointer cursor has not moved.

Clients can deactivate pointer grabs either actively by calling the UNGRAB POINTER routine or passively by calling the UNGRAB BUTTON routine. Whether deactivating the grab is active or passive, Xlib sets the following members of the crossing event data structure to the indicated constants after the pointer cursor moves from one window to another:

- Type member—LeaveNotify
- Mode member—NotifyUngrab

When a client passively deactivates a pointer grab by calling the UNGRAB BUTTON routine, Xlib reports a button release event before reporting that the pointer has been released.

9.5 Input Focus Events

Input focus defines the window to which Xlib sends keyboard input. The keyboard is always attached to some window. Typically, keyboard input goes to either the root window or to a window at the top of the stack called the **focus window**. The focus window and the position of the pointer determine the window that receives keyboard input.

When the keyboard input focus changes from one window to another, Xlib reports a focus out event and a focus in event. The window that loses the input focus receives the focus out event. The window that gains the focus receives a focus in event. Additionally, Xlib notifies other windows in the hierarchy of focus in and focus out events.

To receive notification of input focus events, pass the window identifier and the **FocusChangeMask** mask when using the selection method described in Section 9.2.

Xlib uses the focus change event data structure to report keyboard input focus events.

9.6 Exposure Events

Xlib reports an exposure event when one of the following conditions occurs:

- A formerly obscured window or window region becomes visible.
- A destination region cannot be computed.
- A graphics request exposes one or more regions.

This section describes how to handle window exposures and graphics exposures.

9.6.1 Handling Window Exposures

A window exposure occurs when a formerly obscured window becomes visible again. Because Xlib does not guarantee to preserve the contents of regions when windows are obscured or reconfigured, clients are responsible for restoring the contents of the exposed window.

To receive notification of window exposure events, pass the window identifier and the **ExposureMask** mask when using the selection method described in Section 9.2. Xlib notifies clients of window exposures using the expose event data structure.

The following illustrates the data structure:

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        int x, y;
        int width, height;
        int count;
} XExposeEvent;
```

Table 9–8 describes members of the expose event data structure.

Table 9–8 Expose Event Data Structure Members

Member Name	Contents
type	Value defined by the Expose constant.
serial	Number of the last request processed by the server.
send_event	Value defined by the constant true if the event came from a SEND EVENT request.
display	Address of the display on which the event occurred.
window	Event window.
x	The x value of the coordinates that define the upper left corner of the region that is exposed. The coordinates are relative to the origin of the drawable.
y	The y value of the coordinates that define the upper left corner of the region that is exposed. The coordinates are relative to the origin of the drawable.
width	Width of the exposed region.
height	Height of the exposed region.
count	Number of exposure events that are to follow. If Xlib sets the count to zero, no more exposure events follow for this window. Clients that do not want to optimize redisplay by distinguishing between subareas of its windows can ignore all exposure events with nonzero counts and perform full redisplays on events with zero counts.

The following fragment from the sample program in Chapter 1 illustrates window exposure event handling:

```
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:         doExpose(&event); break;
                    .
                    .
                    .
static void doExpose(eventP)
XEvent *eventP;
{
    char message[] = {"Click here to exit"};

    if (eventP->xexpose.window != window2) return;

    XDrawImageString(dpy, window2, gc, 75, 75, message,
         strlen(message));
}
```

The program checks exposure events to verify that the server has mapped the second window. After the window is mapped, the program writes text into it.

The client-defined *doExpose* routine checks the window and count members of the expose event data structure to determine whether or not the server has completed mapping *window2*. If the window is mapped, the program writes the message "Click here to exit" in it.

9.6.2 Handling Graphics Exposures

Xlib reports graphics exposures when one of the following conditions occurs:

- A destination region could not be computed due to an obscured or out-of-bounds source region. For information about destination and source regions, see Chapter 6.
- A graphics request exposes one or more regions. If the request exposes more than one region, Xlib reports them continuously.

Instead of using the SELECT INPUT routine to indicate an interest in graphics exposure events, assign a value of true to the graphics_exposures member of the GC values data structure. Clients can set the value to true at the time they create a graphics context. If a graphics context exists, use the SET GRAPHICS EXPOSURES routine to set the value of the field. For information about creating a graphics context and using the SET GRAPHICS EXPOSURES routine, see Chapter 4.

Xlib uses the graphics expose event data structure to report graphics exposures. The following illustrates the data structure:

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Drawable drawable;
        int x, y;
        int width, height;
        int count;
        int major_code;
        int minor_code;
} XGraphicsExposeEvent;
```

Table 9–9 describes members of the graphics expose event data structure.

Table 9–9 Graphics Expose Event Data Structure Members

Member Name	Contents
type	Value defined by the GraphicsExpose constant.
serial	Number of the last request processed by the server.
send_event	Value defined by the constant true if the event came from a SEND EVENT request.
display	Address of the display on which the event occurred.
drawable	Drawable reporting the event.
x	The x value of the coordinates that define the upper left corner of the exposed region. The coordinates are relative to the origin of the drawable.
y	The y value of the coordinates that define the upper left corner of the exposed region. The coordinates are relative to the origin of the drawable.
width	Width of the exposed region.
height	Height of the exposed region.
count	Number of exposure events that are to follow. If Xlib sets the count to zero, no more exposure events follow for this window.
major_code	Indicates whether the graphics request was a copy area or a copy plane.
minor_code	The value zero. Reserved for use by extensions.

Xlib uses the no expose event data structure to report when a graphics request that might have produced an exposure did not. The following illustrates the data structure:

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Drawable drawable;
        int major_code;
        int minor_code;
} XNoExposeEvent;
```

Table 9–10 describes members of the no expose event data structure.

Table 9–10 No Expose Event Data Structure Members

Member Name	Contents
type	Value defined by the NoExpose constant.
serial	Number of the last request processed by the server.
send_event	Value defined by the constant true if the event came from a SEND EVENT request.
display	Address of the display on which the event occurred.

(continued on next page)

Table 9–10 (Cont.) No Expose Event Data Structure Members

Member Name	Contents
drawable	Window or pixmap reporting the event.
major_opcode	Indicates whether the graphics request was a copy area or a copy plane.
minor_opcode	The value zero. Reserved for use by extensions.

Example 9–5 illustrates handling graphics exposure events. The program checks for graphics exposures and no exposures to scroll up a window.

Figure 9–2 shows the resulting output of the program.

Example 9–5 Handling Graphics Exposures

```
#define scrollPixels    1
#define windowWidth     600
#define windowHeight    600

Display *dpy;
Window win;
GC gc;
Screen *screen;
int n;
int ButtonIsDown;
int vY = 0;
                    .
                    .
                    .

/**** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:                doExpose(&event); break;
            case ButtonPress:           doButtonPress(&event); break;
            case GraphicsExpose:        doGraphicsExpose(&event); break;
            case ButtonRelease:         doButtonRelease(&event); break;
            case NoExpose:              doNoExpose(&event); break;
        }
    }
}

/***** Write a message *****/
static void doExpose(eventP)
XEvent *eventP;
{
    char message1[ ] = {"To scroll, press MB1."};
    char message2[ ] = {"To exit, click MB2."};

    XDrawImageString(dpy, win, gc, 150, 25, message1, strlen(message1));
    XDrawImageString(dpy, win, gc, 150, 50, message2, strlen(message2));
}
```

(continued on next page)

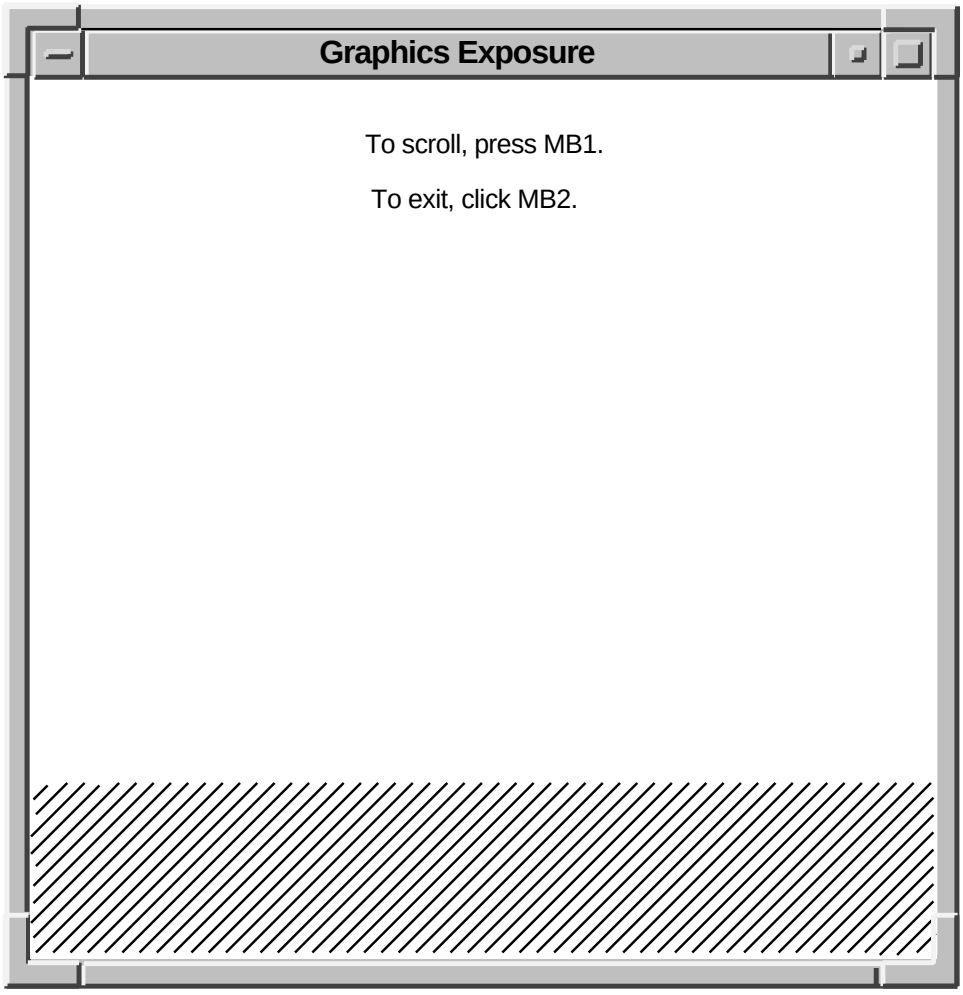
Example 9–5 (Cont.) Handling Graphics Exposures

```
/***** Start a scroll operation *****/
static void startScroll()
{
❶   XCopyArea(dpy, win, win, gc, 0, scrollPixels,
        windowWidth, windowHeight, 0, 0);
    vY += scrollPixels;
}

/***** Copy the area *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
    if (eventP->xbutton.button == Button2) sys$exit(1);
    ButtonIsDown = 1;
    startScroll();
    return;
}

/***** Draw points into the exposed area *****/
static void doGraphicsExpose(eventP)
XEvent *eventP;
{
❷   int x = eventP->xgraphicsexpose.x;
    int y = eventP->xgraphicsexpose.y;
    int width = eventP->xgraphicsexpose.width;
    int height = eventP->xgraphicsexpose.height;
    int px, py;

    for (py=y; py<(y+height); py++)
        for (px=x; px<(x+width); px++)
            if (!((px+py+vY) % 10)) XDrawPoint(dpy, win, gc, px, py);

    if (ButtonIsDown) startScroll();

    return;
}

/***** Quit scrolling when the button is released *****/
static void doButtonRelease(eventP)
XEvent *eventP;
{
    ButtonIsDown = 0;
    return;
}

/***** Draw points in the exposed area when window is obscured *****/
static void doNoExpose(eventP)
XEvent *eventP;
{
    if (ButtonIsDown) startScroll();
    return;
}
```

❶ The client-defined *startScroll* routine copies the window contents, less one row of pixels, to the top of the window. The result leaves an exposed area one pixel high at the bottom of the window.

❷ When a graphics exposure occurs, the client calculates where to draw points into the exposed area by referring to members of the expose event data structure.

Figure 9–2 Window Scrolling



ZK−2513A−GE

9.7 Key Events

Xlib reports key press and key release events to interested clients. To receive event notification of key presses and releases, pass the window identifier and either the **KeyPressMask** mask or the **KeyReleaseMask** mask when using the selection method described in Section 9.2.

Xlib uses a key event data structure to report key presses and releases to interested clients whenever any key changes state, even when the key is mapped to modifier bits.

9.8 Window State Notification Events

Xlib reports events related to the state of a window when a client does one of the following:

- Circulates a window, changing the order of the window hierarchy
- Configures a window, changing its position, size, or border
- Creates a window

- Destroys a window
- Changes the size of a parent, causing Xlib to move a child window
- Maps a window
- Reparents a window
- Unmaps a window
- Changes the visibility of a window

This section describes handling events that result from these operations. For more information about these events, see the *X Window System*.

9.8.1 Handling Window Circulation

To receive notification when a client circulates a window, pass either the window identifier and the **StructureNotifyMask** mask or the identifier of the parent window and the **SubstructureNotifyMask** mask when using a selection method described in Section 9.2.

Xlib reports to interested clients a change in the hierarchical position of a window when a client calls the CIRCULATE SUBWINDOWS, CIRCULATE SUBWINDOWS UP, or CIRCULATE SUBWINDOWS DOWN routine.

Xlib uses the circulate event data structure to report circulate events.

9.8.2 Handling Changes in Window Configuration

To receive notification when window size, position, border, or stacking order changes, pass either the window identifier and the **StructureNotifyMask** mask or the identifier of the parent window and the constant **SubstructureNotifyMask** when using the selection method described in Section 9.2.

Xlib reports changes in window configuration when the following occur:

- Window size, position, border, and stacking order change when a client calls the CONFIGURE WINDOW routine.
- Window position in the stacking order changes when a client calls the LOWER WINDOW, RAISE WINDOW, or RESTACK WINDOW routine.
- Window moves when a client calls the MOVE WINDOW routine.
- Window size changes when a client calls the RESIZE WINDOW routine.
- Window size and location change when a client calls the MOVE RESIZE WINDOW routine.
- Border width changes when a client calls the SET WINDOW BORDER WIDTH routine.

For more information about these routines, see Chapter 3.

Xlib reports changes to interested clients using the configure event data structure.

9.8.3 Handling Window Creations

To receive notification when a client creates a window, pass the identifier of the parent window and the constant **SubstructureNotifyMask** when using the selection method described in Section 9.2.

Xlib reports window creations using the create window event data structure.

9.8.4 Handling Window Destructions

To receive notification when a client destroys a window, pass either the window identifier and the constant **StructureNotifyMask** or the identifier of the parent window and the **SubstructureNotifyMask** mask when using the selection method described in Section 9.2.

Xlib reports window destructions using the destroy window event data structure.

9.8.5 Handling Changes in Window Position

To receive notification when a window is moved because a client has changed the size of its parent, pass the window identifier and the **StructureNotifyMask** mask or the identifier of the parent window and the **SubstructureNotifyMask** mask when using the selection method described in Section 9.2.

Xlib reports window gravity events using the gravity event data structure.

9.8.6 Handling Window Mappings

To receive notification when a window changes state from unmapped to mapped, pass either the window identifier and the **StructureNotifyMask** mask or the identifier of the parent window and the **SubstructureNotifyMask** mask when using the selection method described in Section 9.2.

Xlib reports window mapping events using the map event data structure.

9.8.7 Handling Key, Keyboard, and Pointer Mappings

All clients receive notification of changes in key, keyboard, and pointer mapping. Xlib reports these events when a client has successfully done one of the following:

- Called the SET MODIFIER MAPPING routine to indicate which keycodes are modifiers
- Changed keyboard mapping using the CHANGE KEYBOARD MAPPING routine
- Set pointer mapping using the SET POINTER MAPPING routine

Xlib reports key, keyboard, and pointer mapping events using the mapping event data structure.

9.8.8 Handling Window Reparenting

To receive notification when the parent of a window changes, pass either the window identifier and the **StructureNotifyMask** mask or the identifier of the parent window and the **SubstructureNotifyMask** mask when using the selection method described in Section 9.2.

Xlib reports window reparenting events using the reparent event data structure.

9.8.9 Handling Window Unmappings

To receive notification when a window changes from mapped to unmapped, pass either the window identifier and the **StructureNotifyMask** mask or the identifier of the parent window and the **SubstructureNotifyMask** mask when using the selection method described in Section 9.2.

Xlib reports window unmapping events using the unmap event data structure.

9.8.10 Handling Changes in Window Visibility

All or part of a window is visible if it is mapped to a screen, if all of its ancestors are mapped, and if it is at least partially visible on the screen. To receive notification when the visibility of a window changes, pass the window identifier and the **StructureNotifyMask** mask when using the selection method described in Section 9.2.

Xlib reports changes in visibility to interested clients using the visibility event data structure.

9.9 Key Map State Events

Xlib reports changes in the state of the key map immediately after every enter notify and focus in event.

To receive notification of key map state events, pass the window identifier and the **KeymapStateMask** mask when using the selection method described in Section 9.2.

Xlib uses the keymap event data structure to report changes in the key map state.

9.10 Color Map State Events

Xlib reports a color map event when the window manager installs, changes, or removes the color map.

To receive notification of color map events, pass the window identifier and the **ColormapChangeMask** mask when using the selection method described in Section 9.2.

Xlib reports color map events to interested clients when the following occur:

- A client sets the color map member of the set window attributes data structure by calling CHANGE WINDOW ATTRIBUTES. See Chapter 3 for more information on the data structure and routine.
- A client calls the FREE COLORMAP routine. See Section 5.5 for more information about FREE COLORMAP.
- The window manager installs or removes a color map in response to either a client call of the INSTALL COLORMAP or UNINSTALL COLORMAP routine.

Xlib reports color map events using the color map event data structure.

9.11 Client Communication Events

Xlib reports an event when one of the following occurs:

- One client notifies another client that an event has happened.
- A client changes, deletes, rotates, or gets a property.
- A client loses ownership of a window.
- A client requests ownership of a window.

This section describes how to handle communication between clients.

9.11.1 Handling Event Notification from Other Clients

Clients can notify each other of events by calling the SEND EVENT routine.

Xlib sends notification between clients using the client message event data structure.

9.11.2 Handling Changes in Properties

As Chapter 3 notes, a property associates a constant with data of a particular type. Xlib reports a property event when a client does one of the following:

- Changes a property
- Rotates a window property
- Gets a property
- Deletes a property

To receive information about property changes, pass the window identifier and the **PropertyChangeMask** mask when using the selection method described in Section 9.2.

Xlib reports changes in properties to interested clients using the property event data structure.

9.11.3 Handling Changes in Selection Ownership

Clients receive notification automatically when they lose ownership of a selection in a window. Xlib reports the event when a client takes ownership of a selection by calling the SET SELECTION OWNER routine.

To report the event, Xlib uses the selection clear event data structure.

9.11.4 Handling Requests to Convert a Selection

The server issues a selection request event to the owner of a selection when a client calls the CONVERT SELECTION routine. For information about the CONVERT SELECTION routine, see Section 3.6.

To report the event, Xlib uses the selection request event data structure.

9.11.5 Handling Requests to Notify of a Selection

The server issues a selection notify event after a client calls the CONVERT SELECTION routine. The owner of the selection being converted should initiate this event by calling SEND EVENT when either the selection has been converted and stored as a property or the selection conversion could not be performed. For information about converting selections, see Section 3.6.

To report the event, Xlib uses the selection event data structure.

9.12 Event Queue Management

Xlib maintains an input queue known as the **event queue**. When an event occurs, the server sends the event to Xlib, which places it at the end of an event queue. By using routines described in this section, the client can check, remove, and process the events on the queue. As the client removes an event, remaining events move up the event queue.

Certain routines may **block** or prevent other routine calls from accessing the event queue. If the blocking routine does not find an event that the client is interested in, Xlib flushes the output buffer and waits until an event is received from the server.

9.12.1 Checking the Contents of the Event Queue

To check the event queue without preventing other routines from accessing the queue, use the EVENTS QUEUED routine. Clients can check events already queued by calling the EVENTS QUEUED routine and specifying one of the following constants:

QueuedAlready	Returns the number of events already in the event queue and never performs a system call.
QueuedAfterFlush	Returns the number of events in the event queue if the value is a nonzero. If there are no events in the queue, this routine flushes the output buffer, attempts to read more events out of the client connection, and returns the number read.
QueuedAfterReading	Returns the number of events already in the event queue if the value is a nonzero. If there are no events in the queue, this routine attempts to read more events out of the client connection without flushing the output buffer and returns the number read.

To return the number of events in the event queue, use the PENDING routine. If there are no events in the queue, PENDING flushes the output buffer, attempts to read more events out of the client connection, and returns the number read. The PENDING routine is identical to EVENTS QUEUED with constant **QueuedAfterFlush** specified.

9.12.2 Returning the Next Event on the Queue

To return the first event on the event queue and copy it into the specified event data structure, use the NEXT EVENT and PEEK EVENT routines. NEXT EVENT returns the first event, copies it into an EVENT structure, and removes it from the queue. PEEK EVENT returns the first event, copies it into an event data structure, but does not remove it from the queue. In both cases, if the event queue is empty, the routine flushes the output buffer and blocks until an event is received.

9.12.3 Selecting Events That Match User-Defined Routines

Xlib enables the client to check all the events on the queue for a specific type of event by specifying a client-defined routine known as a **predicate procedure**. The predicate procedure determines if the event on the queue is one that the client is interested in.

The client calls the predicate procedure from inside the event routine. The predicate procedure should determine only if the event is useful and must not call Xlib routines. The predicate procedure is called once for each event in the queue until it finds a match.

Table 9–11 lists routines that use a predicate procedure and indicates whether or not the routine blocks.

Table 9–11 Selecting Events Using a Predicate Procedure

Routine	Description	Blocking/ No Blocking
IF EVENT	Checks the event queue for the specified event. If the event matches, removes the event from the queue. This routine is also called each time an event is added to the queue.	Blocking
CHECK IF EVENT	Checks the event queue for the specified event. If the event matches, removes the event from the queue. If the predicate procedure does not find a match, it flushes the output buffer.	No blocking
PEEK IF EVENT	Checks the event queue for the specified event but does not remove it from the queue. This routine is also called each time an event is added to the queue.	Blocking

9.12.4 Selecting Events Using an Event Mask

Xlib enables a client to process events out of order by specifying a window identifier and one of the event masks listed in Table 9–3 when calling routines listed in Table 9–12.

For example, the following specifies keyboard events on window *window* by using the event mask name constant **KeymapStateMask**.

```
.
.
.
XWindowEvent(dpy, window, KeymapStateMask, &event)
```

Table 9–12 lists routines that use event or window masks and indicates whether the routine blocks.

Table 9–12 Routines to Select Events Using a Mask

Routine	Description	Blocking/ No Blocking
WINDOW EVENT	Searches the event queue and removes the next event that matches both the specified window and event mask	Blocking
CHECK WINDOW EVENT	Searches the event queue, then the events available on the server connection, and removes the first event that matches the specified event and window mask	No blocking
MASK EVENT	Searches the event queue and removes the next event that matches the event mask	Blocking
CHECK MASK EVENT	Searches the event queue, then the events available on the server connection, and removes the next event that matches an event mask	No blocking
CHECK TYPED EVENT	Returns the next event in the queue that matches an event type	No blocking
CHECK TYPED WINDOW EVENT	Searches the event queue, then the events available on the server connection, and removes the next event that matches the specified type and window	No blocking

9.12.5 Putting Events Back on Top of the Queue

To push an event back onto the top of the event queue, use the PUT BACK EVENT routine. PUT BACK EVENT is useful when a client returns an event from the queue and decides to use it later. There is no limit to how many times in succession PUT BACK EVENT can be called.

9.12.6 Sending Events to Other Clients

To send an event to a client, use the SEND EVENT routine. For example, owners of a selection should use this routine to send a SELECTION NOTIFY event to a requestor when a selection has been converted and stored as a property.

9.13 Error Handling

Xlib has two default error handlers. One manages fatal errors, such as when the connection to a display is severed due to a system failure. The other handles error events from the server. The default error handlers print an explanatory message and text and then exit.

Each of these error handlers can be replaced by client error handling routines. If a client-supplied routine is passed a null pointer, Xlib reinvokes the default error handler.

This section describes the Xlib event error handling resources including enabling synchronous operation, handling server errors, and handling input/output (I/O) errors.

9.13.1 Enabling Synchronous Operation

When debugging programs, it is convenient to require Xlib to behave synchronously so that errors are reported at the time they occur.

To enable synchronous operation, use the SYNCHRONIZE routine. The client passes the **display** argument and the **onoff** argument. The **onoff** argument passes either a value of zero (disabling synchronization) or a nonzero value (enabling synchronization).

9.13.2 Using the Default Error Handlers

To handle error events when an error event is received, use the SET ERROR HANDLER routine.

Xlib provides an error event data structure that passes information to the SET ERROR HANDLER routine.

The following illustrates the error event data structure:

```
typedef struct {
        int type
        Display *display
        unsigned long serial;
        char error_code;
        char request_code;
        char minor_code;
        XID resourceid;
} XErrorEvent;
```

Table 9–13 describes the members of the data structure.

Table 9–13 Error Event Data Structure Members

Member Name	Description
type	Type of error event being reported
display	Display on which the error event occurred
serial	Number of requests starting at one sent over the network connection since it was opened
error_code	Identifying error code of the failing routine
request_code	Protocol representation of the name of the procedure that failed and defined in X11/X.h
minor_code	Minor opcode of failed request
resourceid	Resource ID

The routines described in this section return Xlib error codes. For a description of the error codes, see the *X Window System*. The following lists the codes:

BadAccess
BadAlloc
BadAtom
BadColor
BadCursor
BadDrawable
BadFont
BadGC
BadIDChoice
BadImplementation
BadLength
BadMatch
BadName
BadPixmap
BadRequest
BadValue
BadWindow

9.13.3 Confirming X Resource Creation

When creating any X resource, such as a window, pixmap, or gc, it is important to note that these routines are asynchronous and do not return errors if the create operation fails. Although Xlib returns a resource ID for these routines, it does not indicate that a valid resource was created by the server.

Use the following method to check if the client has successfully created a resource:

1. Provide a client-defined error handler and specify it by calling the SET ERROR HANDLER routine.
2. Call the NEXT REQUEST routine. The NEXT REQUEST routine returns the serial number that Xlib is to use for the next request.
3. Call the routine to create the resource, such as CREATE PIXMAP.
4. Call the SYNC routine. The SYNC routine forces all requests in the output buffer to be processed by the server and returns any errors to the error handler.
5. Use the error handler to compare the **serial** member of the error event data structure with the serial number returned by the NEXT REQUEST routine. The value of the **serial** member in the error event data structure reflects the number of the request immediately before the failing call was made. Therefore, if the values are equal, the server has failed to create the resource.

10

Using the X Resource Manager

The X resource manager, also referred to as the resource manager in the rest of this chapter, is a database manager that provides a set of tools for specifying client preferences such as color, fonts, and line width.

This chapter describes using the resource manager and includes the following topics:

- Defining the fundamentals of the resource manager—How the resource manager operates as a database
- Getting the default values — How to return the default values defined for a user environment
- Storing resources—How to create a database and store resources in it
- Retrieving resources from the database—How to look up resources and obtain values
- Merging and storing databases—How to merge database contents and write a database to disk
- Using representations for strings—How to use quarks with the resurce manager

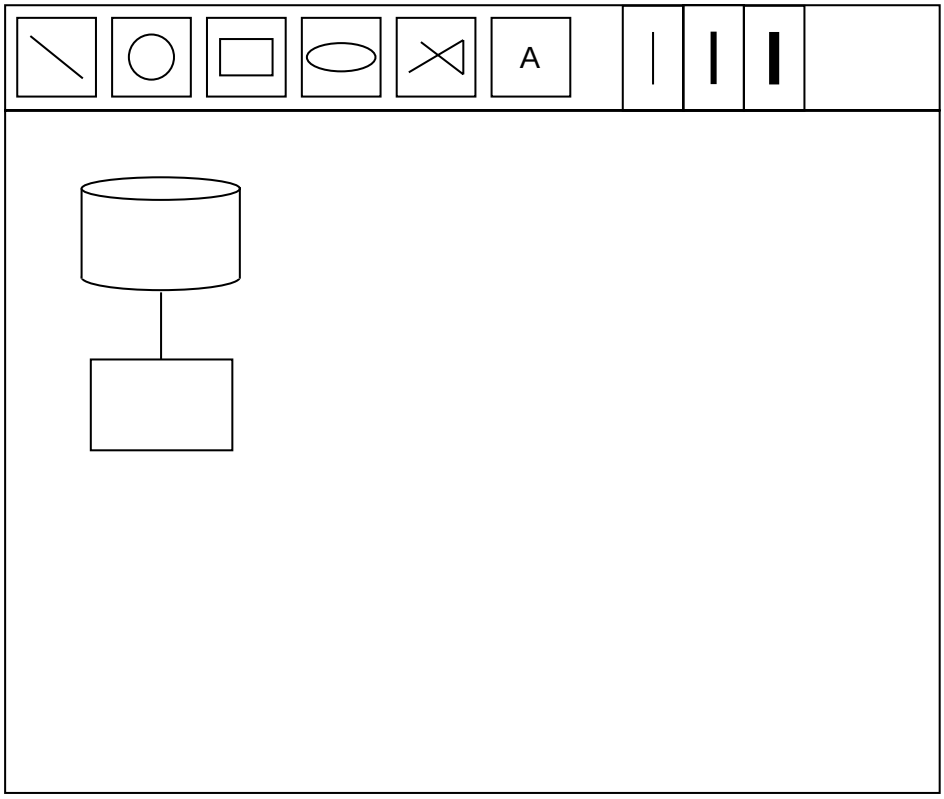
10.1 Defining Resource Manager Fundamentals

The **resource manager** is a database manager; however, it operates differently than most database managers. In most systems, the database contains precise specifications; the client then queries the database using imprecise or broad specifications. With the resource manager, a large set of resources can be specified in the database with one imprecise specification. The client queries the resource manager with a precise specification and one value is returned. Thus, the resource manager can be used by clients to return values for color, font names, or other resources.

For example, a client can specify in the resource manager database that all windows should have a blue background but that all windows used for editing text should have a white background. This specification can be done using just two entries in the database: one entry that sets the text-editing window to white and one entry that sets all other windows to blue.

To illustrate how the resource manager works, consider the following example of a simple graphics client called *xgr*. This client allows the user to draw simple graphic line figures and text using the command buttons at the top of the window. In addition, it provides drawing in three different line widths. Figure 10–1 illustrates the user interface of the client.

Figure 10–1 Interface of Client *xgr*



ZK–1219A–GE

Assume that the interface of the client *xgr* uses a set of windows from the parent window (which includes all client windows) all the way down to the individual command buttons that actually draw the graphics. It is this window hierarchy that provides one method for creating and naming the resources in the database.

It is important to note here that the resource manager imposes no restrictions on the entries in the database. However, if entries in the database are formed logically, then it is easy to specify resources for any portion of the client. Incorrect entries can have unpredictable results.

10.1.1 Names and Classes

Each object that uses the resource manager must have a **name** and a **class**. A name is a more specific way of referring to a particular object or window; whereas, the class is more general and can be used for returning values for an entire set, or class, of windows or objects.

The name and the class are built from components at each level of the window hierarchy. Components within a name and a class are separated by periods. Although names and classes can have an arbitrary number of components, each name and class must be *fully-qualified*. A fully-qualified name has the same number of components as its reciprocal fully-qualified class. Refer to Figure 10–2 for an example of fully-qualified names and classes.

Because names and classes can often contain the same components, the following conventions are used to differentiate between a name and a class:

- The initial letter of each component in a name is lowercase.
- The initial letter of each component in a class is uppercase.
- Where a component is made up of two words, the second letter is capitalized in both cases.

Although the preceding conventions are not rules, they provide one method to differentiate between names and classes.

Note that in the resource manager names and classes are created by the client and are entirely arbitrary. Section 10.1.2 shows how names and classes can be formed and used.

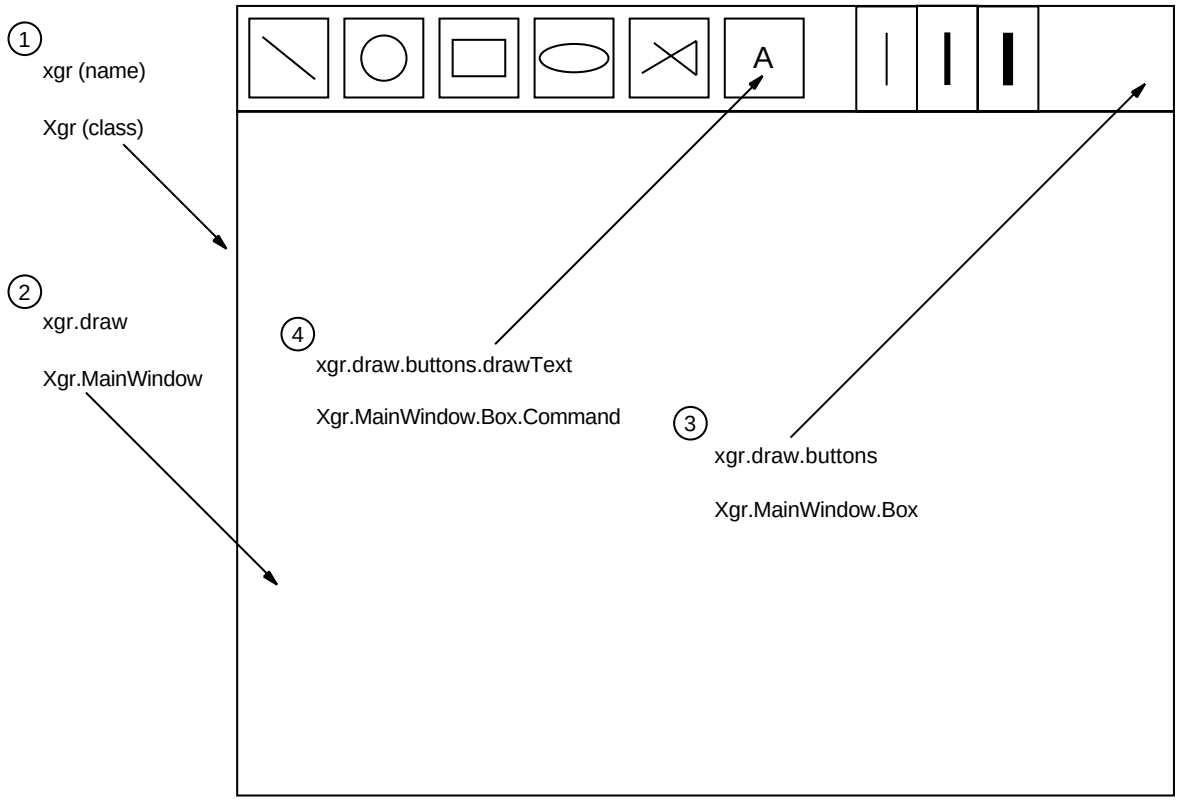
10.1.2 Forming Names and Classes

To illustrate how names and classes are formed, consider the client *xgr* from the preceding example. At the top level is the client *xgr* which forms the first component of the name and the class, *xgr* and *Xgr* respectively.

The parent window forms the second component for the name and the class. This window is named *main*; its class is *MainWindow*. Therefore, its fully-qualified name is the name of its parent followed by its name, or *xgr.main*. Its fully-qualified class is the class of its parent followed by its class, or *Xgr.MainWindow*.

At the next level is the window that contains the buttons that perform the graphic operations. The fully-qualified name of this window is its name, *buttons*, appended to the name of its parent or *xgr.main.buttons*. Its fully-qualified class is its class appended to its parent's class, or *Xgr.MainWindow.Box*.

Figure 10–2 Hierarchy of Names and Classes of the Client *xgr*



Circled values indicate the level of the window in the hierarchy.

ZK−1218A−GE

At the lowest level is the set of windows that actually perform the graphics operations. Each has a name such as *drawText*, or *drawEllipse*, yet each belongs to the class of windows called *Commands*. Therefore, the fully-qualified name of the button that draws the text is *xgr.main.buttons.drawText* and its fully-qualified class is *Xgr.MainWindow.Box.Command*.

Figure 10–2 shows the hierarchy of the windows and the names and classes of each window.

The resources that any object or window needs are called **attributes**, and as such have a name and a class. For example, some of the attributes that the client can set for the window named *main* include height, width, background color, and foreground color. The fully-qualified name of the height attribute for the window *main* is *xgr.main.height*. Its class is *Xgr.MainWindow.Height*.

To illustrate how names and classes differ in their uses, suppose that the client *xgr* needs to have a white background assigned for the window named *draw*, yet needs to have a red background for all command buttons. The following entries in Table 10–1 meet these requirements.

Table 10–1 Example of Using a Name and a Class

Entry	Explanation
xgr.draw.background: white	Within the client named *xgr*, the window named *draw* is assigned a white background.
xgr.draw.buttons.Command.background: red	Within the client named *xgr*, all windows that belong to the class *Command* are assigned a red background. With this one entry, each window that draws the graphics is assigned the color red.

When a client looks up a resource, it passes the complete name and complete class of the resource to a lookup routine. The resource manager then returns the resource value that best matches the name and the class.

10.1.3 Resource Manager Matching Rules

Resources are stored with only partially specified names and classes, using pattern matching constructs. The following rules pertain to individual entries in the database:

- A period (.) is used to separate immediately adjacent components.
- An asterisk (*) is used to represent any number of intervening components (including none).
- A trailing period and asterisk are not removed.
- The library supports 100 components in a name or a class.
- Names and classes can be mixed.

To signal a continuation of a line, use the backslash (\) character. Use the \n characters to indicate a new line, for example when using a multiline table such as a translation table. Refer to the *VMS DECwindows Guide to Application Programming* for more information about translation tables.

The following are examples of resource database entries:

```
xgr.main.color: blue
*MainWindow.Color: red
xgr*Font: -Adobe-New Century Schoolbook-Bold-R-Normal-- \
          *-140-*-*-P-*-ISO8859-1
xgr*lineWidth.thin: 1
xgr*lineWidth.medium: 2
xgr*lineWidth.heavy: 3
```

After the lookup routine passes the fully specified name and class, the lookup algorithm then searches the database for the name that most closely matches the full name and class passed by the lookup routine. The algorithm that determines the resource name that matches a given query is the heart of the database.

Table 10–2 lists the rules for the match in order of precedence and an example of each rule.

Table 10–2 Resource Manager Matching Rules

Rule	Example
The attribute of the name and the class must match.	The queries xgr.main.width (name) and Xgr.MainWindow.Width (class) do not match the database entry: xgr.main: 600. (The entry is missing the attribute, width.)
Database entries with name or class prefixed by a period (.) are more specific than those prefixed by an asterisk (*).	The entry xgr.font is more specific than xgr*font. The entry xgr.font will be fetched before xgr*font.
Names are more specific than classes.	The entry *main.color is more specific than *MainWindow.Color.
Specifying a name or a class is more specific than omitting either.	The entry *MainWindow*Color is more specific than *Color.
Left components are more specific than right components.	The entry *xgr*color is more specific than *main*color for the query xgr.main.color.
If neither a period nor an asterisk is specified at the beginning, a period is implicit.	The entry xgr.main is identical to .xgr.main.

10.2 Getting the Default Values

Xlib provides a convenience routine that makes it easy to find out the fonts, colors, and other defaults to be used for a user environment. The GET DEFAULT routine checks for a database on the display. If it is not present, GET DEFAULT processes the database from the file DECW$XDEFAULTS.DAT and stores it on the display.

Note that the resource file must be named DECW$XDEFAULTS.DAT and that the number of components in both the name and class is 2.

The following illustrates a DECW$XDEFAULTS.DAT file:

```
xgr.background: white
xgr.foreground: black
xgr.textFont: -Adobe-ITC Avant Garde Gothic-Book-R-Normal-- \
              *-100-*-*-P-*-ISO8859-1
```

Example 10–1 shows how to query the DECW$XDEFAULTS.DAT database and return the default property string "black" for the resource entry *xgr.foreground*. Note that the strings returned by GET DEFAULT are owned by Xlib and should not be freed or modified by the client.

Example 10–1 Using the GET DEFAULT Routine

```
Display *dpy;
char *name, *program, *option;

main()
{
     dpy = XOpenDisplay(0);
     program = "xgr";
     option = "foreground";
❶    name = XGetDefault(dpy, program, option);
     printf("The default is '%s'.", name);
}
```

❶ The GET DEFAULT routine has the following format:

property_name_return = XGetDefault(display, program_name,
option_name)

10.3 Storing Resources into a Database

Before the client can retrieve resources from a database, the client must create the database and store resources into it, unless the client is using an application or default database.

Example 10–2 illustrates how to create a database and add a single resource to it using the PUT LINE RESOURCE routine.

Example 10–2 Creating and Storing into the Database

```
                .
                .
                .
Display *dpy;
char *name, *class;
❶XrmDatabase graph_db = 0;
static void doCreateDatabase( );
/***** Create the database ******/
static void doCreateDatabase( )
{
    int i;

❷    char *resource[] ={
       "*Background.Color: light steel blue\n",
       "*Foreground.Color: yellow\n",
       "graph.window.height: 600\n",
       "graph.window.width: 600\n",
       "Graph.Font:
         -Adobe-New Century Schoolbook-Bold-R-Normal--*-140-*-*-p-*-ISO8859-1\n",
       "graph.gc.line.style: 2\n",
       "graph.gc.line.width: 5\n",
       "graph.gc.line.dashOffset: 0\n",
       "graph.gc.line.dashes: 25\n"
       };

❸       for (i = 0; i <= 8; i++)
       XrmPutLineResource (&graph_db, resource[i]);
}
```

❶ The client assigns space for the database *graph_db* and initializes it to 0.

❷ The client creates the array *resources*.

❸ The PUT LINE RESOURCE routine adds a single resource entry that is specified as a string and that contains both a name and a value. If the database contains null, PUT LINE RESOURCE creates a new database and returns a pointer to it.

The PUT LINE RESOURCE routine has the following format:

XrmPutLineResource(database, line)

Any white space before or after the name or colon in the line argument is ignored. The value is terminated by a new-line or null character. To allow values to contain embedded new-line characters, a \n is recognized and replaced by a new-line character.

10.4 Retrieving from the Resource Database

To retrieve a resource from the database, use the GET RESOURCE routine. The GET RESOURCE routine uses the resource manager value data structure. The resource manager value data structure defines database values. Database values consist of a size, an address, and a representation type. The size is specified in bytes. The representation type is a way to store data by some client-defined type, such as a string.

The following illustrates the resource manager value data structure:

```
typedef struct {
        unsigned int size;
        caddr_t addr;
} XrmValue, *XrmValuePtr:
```

Table 10–3 describes the members of the structure.

Table 10–3 Resource Manager Value Data Structure Members

Member Name	Contents
size	Size of the resource
addr	Address of the resource

Example 10–3 illustrates how to create a client-defined routine that retrieves a resource from the database and returns the value to the calling routine. The resources are used to define graphics context values to draw a dashed line. See Chapter 4 for more information about defining graphics contexts.

In the following example, the fully-qualified name of the client is *graph* and its fully-qualified class is *Graph*.

Example 10–3 Retrieving a Resource from the Database

```
#include <decw$include/Xlib.h>
#include <decw$include/Xutil.h>
#include <decw$include/Xresource.h>

Display *dpy;
Window win;
GC gc;
Screen *screen;
int n;
char *name, *class;
❶XrmDatabase graph_db = 0;
XrmValue value;
❷XrmString type;
```

(continued on next page)

Example 10–3 (Cont.) Retrieving a Resource from the Database

```
static void doInitialize( );
static char *doGetResource( );
static char *doDefineColor( );
static void doCreateDatabase( );
static void doCreateWindows( );
static void doCreateGraphicsContext( );
static void doLoadFont( );
static void doExpose( );
static void doMapWindows( );
static void doHandleEvents( );
static void doButtonPress( );

/***** The main program *****/
static int main()
{
    doInitialize( );
    doHandleEvents( );
}

/***** doInitialize *****/
static void doInitialize( )
{
    dpy = XOpenDisplay(0);

    screen = DefaultScreenOfDisplay(dpy);

    doCreateDatabase( );

    doCreateWindows( );

    doCreateGraphicsContext( );

    doLoadFont( );

    doMapWindows( );
}

/***** Create the database *****/
static void doCreateDatabase( )
{
    int i;
    char *resource[] ={
        "*Background.Color: light steel blue\n",
        "*Foreground.Color: yellow\n",
        "graph.window.height: 600\n",
        "graph.window.width: 600\n",
        "Graph.Font:
           -Adobe-New Century Schoolbook-Bold-R-NormaL--*-140-*-*-P-*-ISO8859-1\n",
        "graph.gc.line.style: 2\n",
        "graph.gc.line.width: 5\n",
        "graph.gc.line.dashOffset: 0\n",
        "graph.gc.line.dashes: 25\n"
        };

    for (i = 0; i <= 8; i++)
        XrmPutLineResource (&graph_db, resource[i]);

    XrmPutFileDatabase (graph_db, "graph_db.dat");
}
```

(continued on next page)

Example 10–3 (Cont.) Retrieving a Resource from the Database

```
/***** Get the resource *****/
❸static char *doGetResource (name, class)
{
    if (XrmGetResource (graph_db, name, class, &type, &value)){
        printf("Returning '%s'.\n", value.addr);
        return value.addr;
    }

    else{
        printf("no such entry in database");
        exit (-1);
    }
}

/***** Create the windows *****/
static void doCreateWindows( )
{
❹   int winW = atoi (doGetResource ("graph.window.width",
                                    "Graph.Window.Width"));
    int winH = atoi (doGetResource ("graph.window.height",
                                    "Graph.Window.Height"));
    int winX = (WidthOfScreen(screen)-winW)>>1;
    int winY = (HeightOfScreen(screen)-winH)>>1;
    XSetWindowAttributes xswa;

    /* Create the window */

    xswa.event_mask = ExposureMask | ButtonPressMask;
❺   xswa.background_pixel = doDefineColor(doGetResource(
        "graph.window.background.color", "Graph.Window.Background.Color"));

    win = XCreateWindow(dpy, RootWindowOfScreen(screen),
        winX, winY, winW, winH, 0,
        DefaultDepthOfScreen(screen), InputOutput,
        DefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);
}
```

(continued on next page)

Example 10–3 (Cont.) Retrieving a Resource from the Database

```
/***** Create the graphics context *****/
static void doCreateGraphicsContext( )
{
    XGCValues xgcv;

    xgcv.foreground = doDefineColor(doGetResource("graph.gc.foreground.color",
                                                  "Graph.GC.Foreground.Color"));
    xgcv.background = doDefineColor(doGetResource("graph.gc.background.color",
                                                  "Graph.GC.Background.Color"));
    xgcv.line_width = atoi (doGetResource("graph.gc.line.width",
                                          "Graph.GC.Line.Width"));
❻   xgcv.line_style = atoi (doGetResource("graph.gc.line.style",
                                          "Graph.GC.Line.Style"));
    xgcv.dash_offset = atoi (doGetResource("graph.gc.line.dashOffset",
                                           "Graph.GC.Line.Type"));
    xgcv.dashes = atoi (doGetResource("graph.gc.line.dashes",
                                      "Graph.GC.Line.Dashes"));
    gc = XCreateGC(dpy, win, GCForeground | GCBackground
        | GCLineWidth | GCLineStyle | GCDashOffset | GCDashList, &xgcv);
}

/***** Load the font for text writing *****/
static void doLoadFont( )
{
    Font font;
    char *fontname;

    fontname = doGetResource ("graph.font", "Graph.Font");
    font = XLoadFont(dpy, fontname);
    XSetFont(dpy, gc, font);
}
                .
                .
                .
```

❶ The client assigns storage for a resource manager data structure.

❷ XrmString is a typedef defined in DECW$INCLUDE: XRESOURCE.H and declares *type* as a character string.

❸ The client-defined *doGetResource* routine calls the GET RESOURCE routine and retrieves a resource from the specified database. The client passes a fully-qualified name and fully-qualified class. The address of the value member of the resource manager value data structure is returned. Note that the value returned points into database memory and therefore must not be freed or modified.

The GET RESOURCE routine has the following format:

XrmGetResource(database_id, name_list_string, class_list_string,
repr_type_return, repr_value_return)

❹ To specify the window height and width, the client calls *doGetResource* and passes the fully-qualified name and class of the resource to be returned. Because the value returned is a string, the C library function *atoi* converts it to an integer.

❺ The background pixel is defined by calling the client-defined routine *doGetResource*, which returns the value "light steel blue". The string is then passed to the client-defined routine *doDefineColor*, which allocates the color. See Chapter 5 for more information about defining colors.

❻ Values for the graphics context data structure members are set using strings returned by *doGetResource* and converted to an integer by *atoi*. Because the resource manager does not translate macros and constants, the integer for the constant **LineDoubleDash** is used. Refer to X.H for these values in the DECW$INCLUDE directory.

10.5 Merging and Storing Databases

The resource manager provides routines that merge two or more databases, store a copy of a database to a disk, and retrieve the database from the disk.

Use the MERGE DATABASES routine to merge two or more databases into one database. This routine is used to combine databases; for example, to combine a client-specific database of defaults and a database of user preferences. The MERGE DATABASES routine overwrites an identical database entry.

Use the PUT FILE DATABASE routine to store a copy of the specified database to disk in a specified file. To retrieve a database from the disk, use the GET FILE DATABASE routine.

Example 10–4 shows how to retrieve two databases from disk, to merge each database into a newly created database, and to store the new database in a disk file on the default directory.

Example 10–4 uses the database file *graph_db* created by Example 10–3. Assume that this database contains the default values for the client named *graph*. Assume that another database exists containing user preferences for several clients including values to be used for *graph*. This database is named *user_pref_db* and contains the following entries:

```
window1.background: dark blue
window1.foreground: white
window1.borderWidth: 1
window2.height: 50
window2.width: 400
graph.window.height: 600
graph.window.width: 600
graph.gc.line.width: 1
graph.gc.line.style: 2
graph.gc.line.dashOffset: 0
graph.gc.line.dashes: 25
Graph.Font:
   -Adobe-Helvetica-Bold-R-Normal--*-140-*-*-P-*-ISO8859-1
*Background.Color: light steel blue
*Foreground.Color: yellow
```

Note that in Example 10–4, the entry Graph.Font in “user_pref_db.dat” is identical to the entry in *graph_db*; however, the value of the font name has been changed. When the client merges the two databases, the value of *Graph.Font* in *user_pref_db* overwrites the value from *graph_db*.

Example 10–4 Merging and Storing Databases

(continued on next page)

Example 10–4 (Cont.) Merging and Storing Databases

```
#include <decw$include/Xlib.h>
#include <decw$include/Xresource.h>

main()
{
    Display *dpy;
    XrmDatabase graph_db, user_pref_db, result_db = 0;
    XrmString type;
    XrmValue value;

❶   graph_db = XrmGetFileDatabase("graph_db.dat");
❷   XrmMergeDatabases(graph_db, &result_db);

    user_pref_db = XrmGetFileDatabase("user_pref_db.dat");
    XrmMergeDatabases(user_pref_db, &result_db);

    XrmGetResource(result_db, "graph.font", "Graph.Font", &type, &value);
    printf("The answer is '%s'.", value.addr);

❸   XrmPutFileDatabase(result_db,"result_db.dat");
}
```

❶ The client retrieves a database from disk and loads it into database memory.

The GET FILE DATABASE routine has the following format:

database_id_return = XrmGetFileDatabase(file_name)

❷ The source database *graph_db* is merged with the destination database *result_db*. Note that when using the MERGE DATABASE routine, the source database is destroyed.

The MERGE DATABASE routine has the following format:

XrmMergeDatabases(src_database_id, dst_database_id)

❸ The client calls the PUT FILE DATABASE to write the database *result_db* to disk on the default directory.

The PUT FILE DATABASE routine has the following format:

XrmPutFileDatabase(database_id, file_name)

10.6 Using Representations for Strings

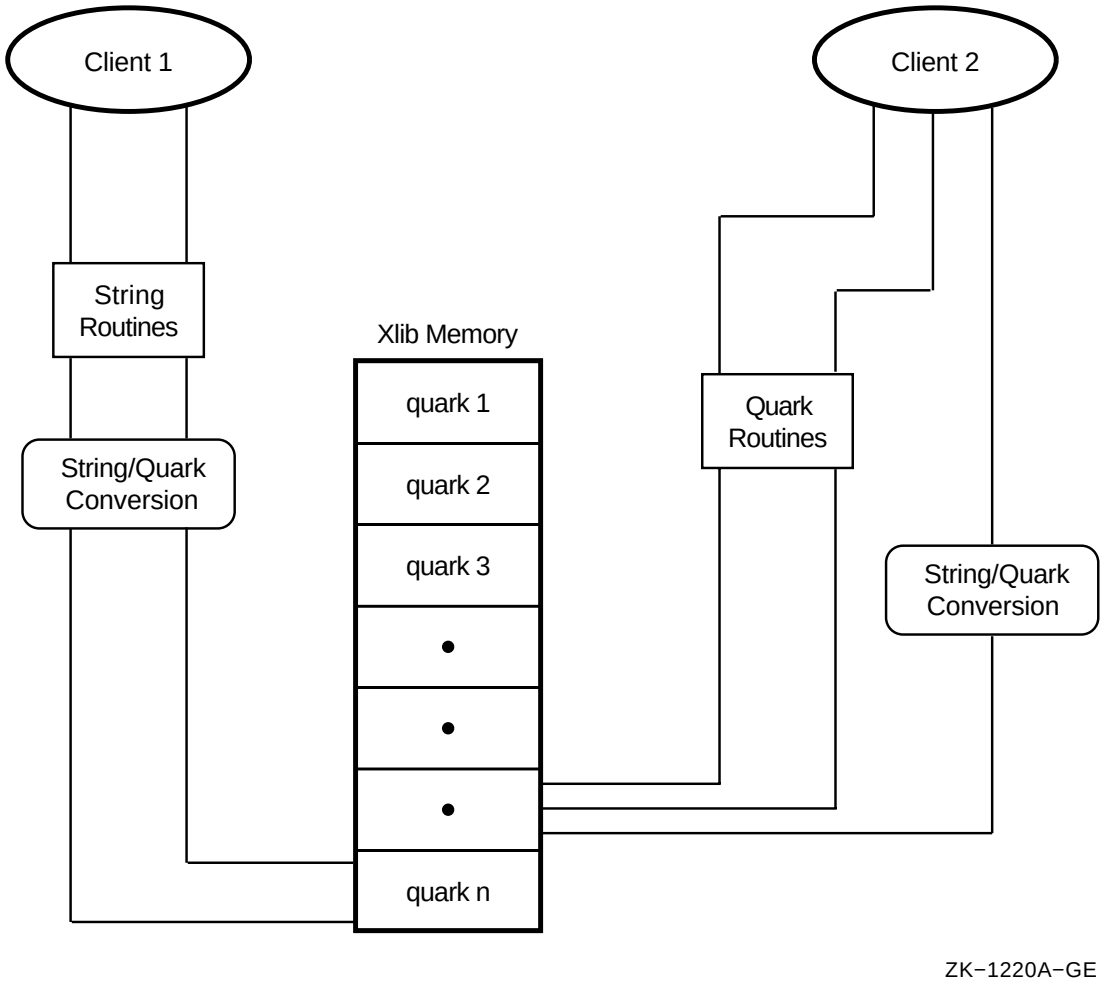
Most uses of the resource manager involve defining names, classes, and types as string constants; however, strings are not stored in the database as ASCII text. All strings are converted from strings to **quarks**. A quark is an integer and is used as a shorthand form of the string constant. A quark is also referred to as a **representation**. Converting from a string to a quark is controlled by the type of resource manager routine used.

The resource manager provides two types of routines for most resource manager functions: string routines and quark routines. String routines are convenience functions that convert all strings to quarks each time the routine is called. For example, when the client calls the PUT RESOURCE routine, the resource manager first converts the string and the value to quarks and then calls Q PUT RESOURCE to add them to the database. Although conversion is transparent to the user when using these routines, performance is slower because a conversion occurs each time a string routine is called. (See Example 10–3 for an illustration of these types of routines.)

Quark routines, on the other hand, operate directly on the quarks that are in the database. Although the client must use conversion routines to change the string to a quark, performance is increased because conversion takes place only once. In addition, quark routines compare integers, which is much faster than string comparisons.

Figure 10–3 illustrates how string and quark routines operate on the database. Client 1 uses string routines that perform a string/quark conversion with each routine call. Client 2 converts the string directly and then uses quark routines to manipulate the database.

Figure 10–3 String and Quark Routines Operation



10.6.1 Converting a String to a Quark

Before using quark routines, the database is created and the strings are converted to quarks. Each string is converted and placed in a **quark list** where each name or class component in the string is converted to an integer. In addition, a **binding list** is created that indicates whether the component is separated by a period or an asterisk. When a component is separated by a period, it is bound tightly. The component is bound loosely when separated by an asterisk.

The following example shows the binding list for the string "graph.window*height". Because a period is implicit if neither a period nor asterisk appears at the beginning of the string, the following binding list is formed from the string ".graph.window*height". Refer to Table 10–2 for more information about resource manager rules.

```
XrmBindTightly
XrmBindTightly
XrmBindLoosely
```

Example 10–5 shows how to allocate space for the binding and quark lists, to convert resources and values, and to put the resources and values into the specified database.

Example 10–5 Converting a String to a Quark

```
#include <decw$include/Xresource.h>
              .
              .
              .
XrmDatabase graph_db;
❶XrmBinding bindings[5];
❷XrmQuark quarks[5];
              .
              .
              .
/***** Create the database *****/
static void doCreateDatabase( )
{
    int i;
    char *resource[] ={
        "*Background.Color",
        "*Foreground.Color",
    };

    char *value_str[] ={
        "light steel blue",
        "yellow",
    };

    for (i = 0; i <= 1; i++){
❸       XrmStringToBindingQuarkList(resource[i], &bindings, &quarks);
❹       XrmQPutStringResource(&graph_db, &bindings, &quarks, value_str[i]);
    }
```

❶ Allocate space for a binding list.

❷ Allocate space for a quark list.

❸ The client calls the STRING TO BINDING QUARK LIST routine to convert the string to a quark list and a binding list.

The STRING TO BINDING QUARK LIST routine has the following format:

XrmStringToBindingQuarkList(value_name, binding_list_id_ret,
repr_list_id_ret)

❹ The client calls the Q PUT STRING RESOURCE routine to add the binding list, quark list, and a resource value to the specified database. If the database is a null value, a new resource database is created.

The Q PUT STRING RESOURCE routine has the following format:

XrmQPutStringResource(database_id, binding_list_id,
repr_list_id, value_name)

10.6.2 Retrieving Resources with Quarks

Once the string has been converted to a quark, other quark routines can be used to query, to compare, and to return values from the database. Although this causes an increase in code, performance is increased for programs that do many such operations.

Example 10–6 shows how to use the following quark routines: STRING TO NAME LIST, STRING TO CLASS LIST, and Q GET RESOURCE. The following example is modified from the example in Section 10.4 to use these routines.

Example 10–6 Retrieving Quarks

```
#include <decw$include/Xresource.h>
                .
                .
                .
Display *dpy;
Window win;
GC gc;
Screen *screen;
XrmDatabase graph_db = 0;
XrmValue value;
❶ XrmRepresentation type;
XrmBinding bindings[5];
XrmQuark quarks[5];
XrmNameList names[5];
XrmClassList classes[5];

static void doInitialize( );
static char *doQGetResource( );
static char *doDefineColor( );
static void doCreateDatabase( );
static void doCreateWindows( );
static void doCreateGraphicsContext( );
static void doLoadFont( );
static void doExpose( );
static void doMapWindows( );
static void doHandleEvents( );
static void doButtonPress( );

/***** The main program *****/

static int main()
{
    doInitialize( );
    doHandleEvents( );
}

/***** doInitialize *****/
static void doInitialize( )
{
    dpy = XOpenDisplay(0);

    screen = DefaultScreenOfDisplay(dpy);

    doCreateDatabase( );
```

(continued on next page)

Example 10–6 (Cont.) Retrieving Quarks

```
    doCreateWindows( );

    doCreateGraphicsContext( );

    doLoadFont( );

    doMapWindows( );
}

/***** Create the database *******/
static void doCreateDatabase( )
{
    int i;
    char *resource[] ={
        "*Background.Color",
        "*Foreground.Color",
        "graph.window.height",
        "graph.window.width",
        "Graph.Font",
        "graph.gc.line.style",
        "graph.gc.line.width",
        "graph.gc.line.dashOffset",
        "graph.gc.line.dashes"
    };

    char *value_str[] ={
        "light steel blue",
        "yellow",
        "600",
        "600",
        "-Adobe-New Century Schoolbook-Bold-R-Normal--*-140-*-*-P-*-ISO8859-1",
        "2",
        "5",
        "0",
        "25"
    };

    for (i = 0; i <= 8; i++){
        XrmStringToBindingQuarkList(resource[i], &bindings, &quarks);
        XrmQPutStringResource(&graph_db, &bindings, &quarks, value_str[i]);
    }
    XrmPutFileDatabase(graph_db, "graph_db.dat");
}

/*****  Get the resource *****/
static doGetResource ( )
{

❷ if (XrmQGetResource (graph_db, names, classes, &type, &value)){
        printf("Returning '%s'.\n", value.addr);
        return value.addr;
    }

    else{
        printf("no such entry in database");
        exit (-1);
    }
}
```

(continued on next page)

Example 10–6 (Cont.) Retrieving Quarks

```
/***** Create the windows *****/
static void doCreateWindows( )
{
    int winW;
    int winH;
    int winX;
    int winY;
    XSetWindowAttributes xswa;

❸   XrmStringToNameList("graph.window.width", &names);
❹   XrmStringToClassList("Graph.Window.Width", &classes);
❺   winW = atoi (doGetResource ( ) );

    XrmStringToNameList("graph.window.height", &names);
    XrmStringToClassList("Graph.Window.Height", &classes);
    winH = atoi (doGetResource ( ) );

    winX = (WidthOfScreen(screen)-winW)>>1;
    winY = (HeightOfScreen(screen)-winH)>>1;


    /* Create the window */

    xswa.event_mask = ExposureMask | ButtonPressMask;

    XrmStringToNameList("graph.window.background.color", &names);
    XrmStringToClassList("Graph.Window.Background.Color", &classes);
    xswa.background_pixel = doDefineColor(doGetResource( ) );

    win = XCreateWindow(dpy, RootWindowOfScreen(screen),
        winX, winY, winW, winH, 0,
        DefaultDepthOfScreen(screen), InputOutput,
        DefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);
}

/***** Create the graphics context *****/
static void doCreateGraphicsContext( )
{
    XGCValues xgcv;

    XrmStringToNameList("graph.gc.foreground.color", &names);
    XrmStringToClassList("Graph.GC.Foreground.Color", &classes);
    xgcv.foreground = doDefineColor(doGetResource( ) );

    XrmStringToNameList("graph.gc.background.color", &names);
    XrmStringToClassList("Graph.GC.Background.Color", &classes);
    xgcv.background = doDefineColor(doGetResource( ) );

    XrmStringToNameList("graph.gc.line.width", &names);
    XrmStringToClassList("Graph.GC.Line.Width", &classes);
    xgcv.line_width = atoi (doGetResource( ) );

    XrmStringToNameList("graph.gc.line.style", &names);
    XrmStringToClassList("Graph.Line.Style", &classes);
    xgcv.line_style = atoi (doGetResource( ) );

    XrmStringToNameList("graph.gc.line.dashOffset", &names);
    XrmStringToClassList("Graph.GC.Line.Type", &classes);
    xgcv.dash_offset = atoi (doGetResource( ) );

    XrmStringToNameList("graph.gc.line.dashes", &names);
    XrmStringToClassList("Graph.Line.Dashes", &classes);
    xgcv.dashes = atoi (doGetResource( ) );
```

(continued on next page)

Example 10–6 (Cont.) Retrieving Quarks

```
    gc = XCreateGC(dpy, win, GCForeground | GCBackground
        | GCLineWidth | GCLineStyle | GCDashOffset | GCDashList, &xgcv);
}
```

(continued on next page)

Example 10–6 (Cont.) Retrieving Quarks

```
/***** Load the font for text writing *****/
static void doLoadFont( )
{
    Font font;
    char *fontname;

    XrmStringToNameList("graph.font", &names);
    XrmStringToClassList("Graph.Font", &classes);
    fontname = doGetResource ( );

    font = XLoadFont(dpy, fontname);
    XSetFont(dpy, gc, font);
}
                .
                .
                .
```

❶ XrmRepresentation is a typedef defined in DECW$INCLUDE:XRESOURCE.H and declares **type** as a quark.

❷ The client-defined *doGetResource* routine calls the Q GET RESOURCE routine and retrieves a resource from the database. The routine passes a fully-qualified name and class in the form of quarks. If a match occurs, the routine returns the address member of the value data structure.

The Q GET RESOURCE routine has the following format:

XrmQGetResource(database_id, name_list_id, class_list_id,
repr_type_id_return, repr_value_id_return)

❸ The STRING TO NAME LIST routine converts a string with one or more components to a quark list.

The STRING TO NAME LIST routine has the following format:

XrmStringToNameList(repr_name, repr_list_id_return)

❹ The STRING TO CLASS LIST routine converts a string with one or more components to a quark list.

The STRING TO CLASS LIST routine has the following format:

XrmStringToClassList(repr_class, repr_list_id_return)

❺ The client-defined *doGetResource* routine returns the value 600, which is converted to an integer by the C Library routine *atoi*.

11
Using Grabs

When mouse and keyboard events occur, the server usually delivers them to an appropriate client, determined by the window and the input focus. However, by using the functions described in this chapter, the client can control the delivery of pointer and keyboard events independently. See Chapter 9 for more information about event handling.

This chapter describes how to use grabs and includes the following topics:

- Fundamentals of grabs
- Grabbing the pointer and buttons
- Grabbing the keyboard and keys

11.1 Grab Fundamentals

When mouse buttons or keyboard keys are grabbed, events are sent to the grabbing client, rather than the client that owns the window.

This section describes how events are reported and other fundamentals of grabbing routines.

11.1.1 Event Reporting

When calling a grab routine, it is possible to specify that pointer or keyboard events be reported by the server in synchronous or asynchronous mode. In asynchronous mode, event processing continues as usual.

If the client specifies synchronous mode, no further events of the type specified to be synchronous (either pointer, keyboard, or both) are processed. Depending on the type specified, the pointer and keyboard are considered to be frozen during this interval. Actual pointer and keyboard events are not lost, they are simply queued in the server for later processing. Further events are processed only when the grabbing client releases the grab or calls the ALLOW EVENTS routine.

Refer to Section 11.5 for more information about the ALLOW EVENTS routine.

11.1.2 Active and Passive Grabs

There are two kinds of grabs: active and passive. An **active grab** occurs when a single client grabs the pointer or the keyboard explicitly. The routines GRAB POINTER and GRAB KEYBOARD perform active grabs.

A **passive grab** occurs when the client grabs a particular key or mouse button in a window. The routines GRAB BUTTON and GRAB KEY perform passive grabs. With passive grabs, the grab activates when the button or key is actually pressed. The passive grab terminates when the button or key is released. Passive grabs are convenient for implementing reliable popup menus.

For example, you can guarantee that the popup is mapped before the up pointer button event occurs by grabbing a button that requests synchronous behavior. The down event triggers the grab and freezes further processing of pointer events until the client maps the popup window. The client can then allow further event processing. The up event is then correctly processed, relative to the popup window.

When performing an active grab on the pointer or the keyboard, the routines take a time argument. The server maintains the time when the following occurs:

- Input focus has changed
- Keyboard was last grabbed
- Pointer was last grabbed
- Selection was last changed

By using a time-stamp, the client can specify that its request should not occur if another application has in the meanwhile taken control of the keyboard, pointer, or selection. One predefined value called **CurrentTime** is used in requests to represent the current server time.

11.2 Pointer Grabs

To perform an active grab on the pointer, use the GRAB POINTER routine.

Example 11–1 illustrates how to use a pointer grab. The example creates two windows and actively grabs the pointer when MB2 is pressed. This changes the pointer cursor shape and confines the pointer cursor to *window2*.

Example 11–1 Grabbing the Pointer

```
#include <decw$include/cursorfont.h>
                    .
                    .
                    .
/***** Create the windows *****/
static void doCreateWindows( )
{
    int window1W = 400;
    int window1H = 400;
    int window1X = (XWidthOfScreen(screen)-window1W)>>1;
    int window1Y = (XHeightOfScreen(screen)-window1H)>>1;
    int window2W = 200;
    int window2H = 200;
    int window2X = 50;
    int window2Y = 50;
    XSetWindowAttributes xswa;

    /* Create the window1 window */

    xswa.background_pixel = doDefineColor(1);
❶   xswa.event_mask = ButtonPressMask;

    window1 = XCreateWindow(dpy, XRootWindowOfScreen(screen),
        window1X, window1Y, window1W, window1H, 0,
        XDefaultDepthOfScreen(screen), InputOutput,
        XDefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);
```

(continued on next page)

Example 11–1 (Cont.) Grabbing the Pointer

```
    /* Create the window2 window */

    xswa.background_pixel = doDefineColor(2);
    xswa.event_mask = ButtonPressMask;

    window2 = XCreateWindow(dpy, window1, window2X, window2Y, window2W,
        window2H, 4, XDefaultDepthOfScreen(screen), InputOutput,
        XDefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);
}
                    .
                    .
                    .
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
             case ButtonPress:          doButtonPress(&event); break;
             case ButtonRelease:        doButtonRelease(&event); break;
        }
    }
}

/***** Grabbing the Pointer *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
    if (eventP -> xbutton.button == Button2) {

        new_cursor = XCreateFontCursor (dpy, XC_sailboat);
❷       XGrabPointer(dpy, window2, 0, ButtonPressMask | ButtonReleaseMask,
            GrabModeAsync, GrabModeAsync, window2, new_cursor, CurrentTime);

    }

    if (eventP ->xbutton.button == Button3) {
❸       XUngrabPointer(dpy, CurrentTime);
        sys$exit (1);
    }
}

/***** Write the message *****/
static void doButtonRelease( )
Xevent *eventP;
{
❹    if (eventP -> xbutton.button == Button1){

        XDrawImageString(dpy, window2, gc, 40, 75, message[state]
            strlen(message[state]));
    }
}
```

❶ By using the set window attributes data structure, the client specifies an interest in button press events for both *window1* and *window2*. See Chapter 3 for more information about setting window attributes.

❷ When MB2 is pressed, the client calls the GRAB POINTER routine, which actively grabs the pointer, changes the pointer cursor shape and confines it to *window2*. If the pointer cursor is not in *window2*, the pointer cursor is automatically moved to the closest edge of *window2* just before the grab activates.

The GRAB POINTER routine has the following format:

XGrabPointer(display, window_id, owner_events, event_mask,
pointer_mode, keyboard_mode, confine_id, cursor_id, time)

Because the **owner_events** argument is set to 0, pointer events are reported with respect to the grabbing window, and only if selected by an event mask. Therefore, in the example, the grabbing client will receive only button press and button release events. (If **owner_events** were set to 1, pointer events would be reported as usual to the client.)

Because the **pointer_mode** and **keyboard_mode** arguments are set to **GrabModeAsync**, all pointer and keyboard events are unaffected by the grab and processing of both event types continue as usual.

❸ When MB3 is pressed, the client calls the UNGRAB POINTER routine, which terminates the grab, and the program exits.

The UNGRAB POINTER routine has the following format:

XUngrabPointer(display, time)

❹ The grabbing client has specified an interest in button release events so that when MB1 is released, the client writes a message in *window2*. Refer to Chapter 8 for more information about writing text.

Clients can also modify the parameters of an active pointer grab by calling the CHANGE ACTIVE POINTER GRAB routine as long as the following is true:

- The pointer is actively grabbed by the client.
- The specified time is no earlier than the last pointer grab time and no later than the current server time.

11.3 Button Grabs

Use the GRAB BUTTON routine to passively grab control of a single mouse button.

Example 11–2 illustrates how to grab a single mouse button. The example creates two windows. The button grab occurs when MB2 and the shift are pressed simultaneously.

Because the GRAB BUTTON routine performs a passive grab, the grab terminates whenever the button is released. To deactivate a button grab before the button release, call UNGRAB BUTTON. The UNGRAB BUTTON routine does not affect any active grab.

Example 11–2 Grabbing a Button

```
                .
                .
                .
XSetWindowAttributes xswa;
/* Create the window1 window */
❶ xswa.event_mask = ButtonPressMask;
```

(continued on next page)

Example 11–2 (Cont.) Grabbing a Button

```
    window1 = XCreateWindow(dpy, XRootWindowOfScreen(screen),
        window1X, window1Y, window1W, window1H, 0,
        XDefaultDepthOfScreen(screen), InputOutput,
        XDefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);

    /* Create the window2 window */

    xswa.event_mask = ButtonPressMask;

    window2 = XCreateWindow(dpy, window1, window2X, window2Y, window2W, window2H, 4,
        XDefaultDepthOfScreen(screen), InputOutput,
        XDefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);
}
                    .
                    .
                    .
/***** Grab Button 2 *****/
static void doGrabButton( )
{
❷   XGrabButton(dpy, Button2, ShiftMask, window1, 0, Button2MotionMask,
         GrabModeAsync, GrabModeAsync, window1, None);
}

/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
             case MotionNotify:          doMotionNotify(&event); break;
             case ButtonPress:           doButtonPress(&event); break;
        }
    }
}

/***** Motion notify event ******/
static void doMotionNotify(eventP)
XEvent *eventP;
{
❸   int x = eventP->xbutton.x;
    int y = eventP->xbutton.y;
    XMoveWindow(dpy, window2, x, y);
}

/***** Button press event *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
    if (eventP ->xbutton.button == Button3) {
        sys$exit (1);
    }
}
```

❶ By using the set window attributes data structure, the client specifies an interest in button press events for both *window1* and *window2*. See Chapter 3 for more information about setting window attributes.

❷ The client-defined routine *doGrabButton* calls the GRAB BUTTON routine to perform a passive grab on MB2 while the shift key and MB2 are pressed and the pointer cursor is in *window1*.

The GRAB BUTTON routine has the following format:

XGrabButton(display, button, modifiers, window_id, owner_events,
event_mask, pointer_mode, keyboard_mode, confine_id,
cursor_id)

The **event_mask** argument specifies that pointer motion events that occur while MB2 is down are to be reported to the grabbing client.

Because the arguments **pointer_mode** and **keyboard_mode** are set to **GrabModeAsync**, both pointer and keyboard events are reported normally. During the grab, the pointer is confined to *window1*.

❸ The client-defined routine *doMotionNotify* uses the x and y members of the motion event data structure as the coordinates to move *window2*. See Chapter 9 for more information about the motion event data structure.

11.4 Key and Keyboard Grabs

To grab the keyboard, use the GRAB KEYBOARD routine. The GRAB KEYBOARD routine actively grabs the keyboard. Use the UNGRAB KEYBOARD to release the keyboard and any queued events if the client has the keyboard actively grabbed from either GRAB KEY or GRAB KEYBOARD.

To grab a single key, use the GRAB KEY routine. The GRAB KEY routine establishes a passive grab on the keyboard.

Example 11–3 illustrates grabbing a key. The example creates two windows. The grab occurs when the F1 and Ctrl keys are pressed simultaneously.

Example 11–3 Grabbing a Key

```
#include <decw$include/keysym.h>
                .
                .
                .
/***** Create the windows *****/
static void doCreateWindows( )
{
    int window1W = 400;
    int window1H = 400;
    int window1X = (XWidthOfScreen(screen)-window1W)>>1;
    int window1Y = (XHeightOfScreen(screen)-window1H)>>1;
    int window2W = 375;
    int window2H = 375;
    XSetWindowAttributes xswa;

    /* Create the window1 window */

    xswa.event_mask = ButtonPressMask;

    window1 = XCreateWindow(dpy, XRootWindowOfScreen(screen),
        window1X, window1Y, window1W, window1H, 0,
        XDefaultDepthOfScreen(screen), InputOutput,
        XDefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);
```

(continued on next page)

Example 11–3 (Cont.) Grabbing a Key

```
    /* Create the window2 window */

    xswa.event_mask = ButtonPressMask;

    window2 = XCreateWindow(dpy, window1, window2X, window2Y, window2W, window2H, 4,
        XDefaultDepthOfScreen(screen), InputOutput,
        XDefaultVisualOfScreen(screen), CWEventMask | CWBackPixel, &xswa);
}

/***** Map the windows *****/
static void doMapWindows( )
{
    XMapWindow(dpy, window1);
    XMapWindow(dpy, window2);
}

/***** Grab the key ******/
static void doGrabKey( )
{
❶   keycode_f1 = XKeysymToKeycode(dpy, XK_F1);
❷   XGrabKey(dpy, keycode_f1, ControlMask, window2, 1, GrabModeAsync,
          GrabModeAsync);
}

/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
             case KeyPress:             doKeyPress(&event); break;
             case ButtonPress:          doButtonPress(&event); break;
        }
    }
}

/***** Key press event *****/
static void doKeyPress(eventP)
XEvent *eventP;
{
❸   if (eventP ->xkey.keycode == keycode_f1){
        XResizeWindow(dpy, window2, resize_w, resize_h);
        XDrawImageString(dpy, window1, gc, 100, 40, message[state],
            strlen(message[state]));
        XDrawImageString(dpy, window1, gc, 100, 60, message[state + 1],
            strlen(message[state + 1]));
    }
}

/***** Button press event *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
    if (eventP ->xbutton.button == Button3) {
    sys$exit (1);
    }
}
```

❶ The client calls the KEYSYM TO KEYCODE routine that converts a specified key symbol to a defined keycode. If the specified key symbol is not defined for any key, the routine returns zero.

The KEYSYM TO KEYCODE routine has the following format:

keycode_return = XKeysymToKeycode(display, keysym_id)

❷ The client establishes a passive grab when the F1 and Ctrl keys are pressed simultaneously. Because the **pointer_mode** and **keyboard_mode** arguments are set to **GrabModeAsync**, all events are reported normally.

The GRAB KEY routine has the following format:

XGrabKey(display, keycode, modifiers, window_id, owner_events,
pointer_mode, keyboard_mode)

❸ The client tests to see that the keycode member of the key event data structure equals the keycode, as specified by the grabbing client. If so, the window is resized and a message is written. See Chapter 9 for more information about the key event data structure. Refer to Chapter 8 for more information about writing text.

11.5 Allowing Events

To allow events to be processed when the device has been frozen, use the ALLOW EVENTS routine. The ALLOW EVENTS routine is used to release some queued events if the client has caused a device to freeze. A device will freeze when the client performs a pointer or keyboard grab and specifies synchronous reporting of events. It will remain frozen until the client that issued the grab request issues an ALLOW EVENTS request.

The ALLOW EVENTS routine has the following format:

XAllowEvents(display, event_mode, time)

In the following example, the client has established a grab on the pointer and has specified that all events are processed synchronously. By calling the ALLOW EVENTS routine and specifying the predefined value **AsyncPointer** for the **event_mode** argument, pointer event processing continues as usual.

```
                .
                .
                .
XGrabPointer(dpy, window1, 0, ButtonPressMask,
    GrabModeSync, GrabModeSync, none, window1, none,
    CurrentTime)
XAllowEvents(dpy, AsyncPointer, CurrentTime)
```

For more information about the ALLOW EVENTS routine and a list of the predefined arguments, see the *X Window System*.

12

Complying with Inter-Client Communications Conventions

In most cases, a client communicates information about its windows to the window and session managers. For example, the client may want to provide the window manager with names for a specific window and icon. In addition, the client may want to provide hints to the window manager concerning window size and location. The X Consortium approves of certain methods and routines that govern this inter-client communication.

The Inter-Client Communications Conventions Manual (ICCCM) details these methods and, through their use, ensures compatibility in a multi-client environment. The *X Window System* contains the Inter-Client Communications Conventions Manual.

This chapter provides information and examples for communicating with the window manager using Xlib ICCCM-compliant routines and properties. For a reference to all Xlib ICCCM-compliant routines, see the *X Window System*. For more information about properties, see Chapter 8.

12.1 Communicating with Standard Properties

Xlib provides predefined properties to enable clients to communicate with the window manager and session manager about the following:

- Window names
- Icon names
- Pixmaps used to define window icons
- Commands used to start the application
- Position and size of windows in their startup state
- Initial state of windows
- Input that windows accept
- Names used to retrieve application resources

Table 12–1 describes the atom names, data types, and formats of these properties.

Table 12–1 Atom Names of Standard Properties

Atom	Data Type	Format	Description
XA_WM_CLASS	text	32	Application resources from the resource database
XA_WM_CLIENT_MACHINE	text	N/A	String name of the machine on which the client application is running
XA_WM_COLORMAP_ WINDOWS	window	32	List of window IDs that may need a different colormap than that of their top-level window
XA_WM_COMMAND	text	8	Command used to start the client
XA_WM_HINTS	wm_hints	32	Hints about keyboard input, initial state, icon pixmap, icon window, icon position, and icon mask
XA_WM_ICON_NAME	text	8	Icon name
XA_WM_ICON_SIZE	wm_icon_size	32	Icon size supported by the window manager
XA_WM_NAME	string	8	Application name
XA_WM_NORMAL_HINTS	wm_size_hints	32	Size hints for a window in its normal state
XA_WM_PROTOCOLS	atom	32	List of atoms that identify the communications protocols between the client and the window manager
XA_WM_STATE	wm_state	32	Property intended for communicating between window manager and session manager
XA_WM_TRANSIENT_FOR	window	32	Property intended for a window that is transient, such as a dialog box

The remainder of this chapter illustrates how to use many of these properties.

12.2 Manipulating Top-Level Windows

Xlib provides routines that the client can use to change the visibility or size of top-level windows. Example 12–1 illustrates the use of the ICONIFY WINDOW and RECONFIGURE WINDOW routines. In the example, the client shrinks the window to an icon ten seconds after the client exposes the window. The user can also reconfigure the window by pressing MB2. Pressing MB3 unmaps and destroys the window.

It is important to note that the window manager ignores any subwindow of the top-level window. To manipulate subwindows, use the routines described in Chapter 3.

Example 12–1 Reconfiguring a Top-Level Window

(continued on next page)

Example 12–1 (Cont.) Reconfiguring a Top-Level Window

```
#include <decw$include/Xlib.h>
#include <decw$include/Xutil.h>

#define FontName\
     "-Adobe-New Century Schoolbook-Bold-I-NormaL--*-140-*-*-P-*-ISO8859-1"
#define WindowName "Manipulating Top-Level Windows"

Display *dpy;
Window window;
GC gc;
Screen *screen;
int scrNum;
int n, i, y, state = 0;
char *message[]= {
    "This window shrinks to an icon in 10 seconds.",
    "Click MB2 to reconfigure the window.",
    "Click MB3 to exit."
    };
float wait_time=10.0;
               .
               .
               .
/***** Handle events *****/
static void doHandleEvents( )
{
    XEvent event;

    XNextEvent(dpy, &event);
❶   doExpose(&event);
❷   doWait_Iconify();

    for ( ; ; ) {
        XNextEvent(dpy, &event);
        switch (event.type) {
            case Expose:        doExpose(&event); break;
            case ButtonPress:   doButtonPress(&event); break;
        }
    }
}

/***** Expose Event *****/
static void doExpose(eventP)
XEvent *eventP;
{
    if (eventP->xexpose.window != window) return;
       XClearWindow(dpy, window);
    for (y=75, i=0; i<3; y+=25, i++) {
       XDrawString(dpy, window, gc, 25, y, message[i],
           strlen(message[i]));
    }
}

/***** Wait, then shrink the window to an icon *****/
static void doWait_Iconify()
{
❸   lib$wait(&wait_time);
    XIconifyWindow(dpy, window, scrNum);
    return;
}
```

(continued on next page)

Example 12–1 (Cont.) Reconfiguring a Top-Level Window

```
/***** Shutdown *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
    if (eventP->xbutton.button == Button2) {
        XWindowChanges xwc;
        xwc.x = 200;
        xwc.y = 200;
        xwc.width = 600;
        xwc.height = 600;
❹       XReconfigureWMWindow(dpy, window, scrNum, CWX | CWY |
                 CWWidth | CWHeight, &xwc);
         return;
    }
❺   if (eventP->xbutton.button == Button3) {
        XWithdrawWindow(dpy, window, scrNum);
        XCloseDisplay(dpy);
        sys$exit (1);
    }
}
```

❶ After the first expose event, the client calls the client-defined *doExpose* routine, which clears and draws the text in the window.

❷ The client calls the client-defined routine *doWait_Iconify*.

❸ The *doWait_Iconify* routine calls the Run-Time Library routine LIB$WAIT and waits ten seconds before shrinking the top-level window to an icon. The ICONIFY WINDOW routine sends a WM_CHANGE_STATE ClientMessage event to the root window of the specified screen. The routine returns a nonzero status if the client message is sent successfully; otherwise, it returns a zero status.

The ICONIFY WINDOW routine has the following format:

XIconifyWindow(display, window, screen_number)

❹ The client has specified an interest in MB2 or MB3 button press events. If the user presses MB2, then the client reconfigures the window by calling the RECONFIGURE WINDOW routine. The RECONFIGURE routine requests that a top-level window be reconfigured as specified by the window changes data structure.

The RECONFIGURE WINDOW routine has the following format:

XReconfigureWindow(display, window, screen_number,
    value_mask, values)

For more information about reconfiguring windows, refer to Chapter 3.

❺ When the user presses MB3, the client exits.

12.3 Defining Window Manager Properties

This section describes how to communicate with the window manager by defining individual properties. Example 12–2 illustrates how to set these properties by using the following Xlib routines:

SET WM NORMAL HINTS
SET WM HINTS
SET WM WINDOW NAME
SET WM ICON NAME
SET CLASS HINT

Xlib also provides the convenience routine, SET WM PROPERIES, which allows the client to set the standard window manager properties with a single function. Example 12–3 illustrates how to use the SET WM PROPERTIES routine.

12.3.1 Setting Window Manager Hints

Xlib provides routines to set and read the WM_HINTS property. Use the WM hints data structure and the SET WM HINTS routine to provide the window manager with hints about keyboard input, initial window state, icon pixmap, icon window, icon position, icon mask, and window group. Use the GET WM HINTS routine to read the WM_HINTS property. Example 12–2 illustrates using the SET WM HINTS routine to provide hints about managing the initial state and icon pixmap of the window.

Note that each time the WM hints data structure is passed to SET WM HINTS, the flags field specifies only which fields are valid, not which fields are updated. Setting one flag and passing one value states that all other values are no longer valid. For those flags not set, the window manager uses the default values.

Table 12–2 lists the flags.

Table 12–2 Window Manager Hints Data Structure Flags

Flag Name	Description
Input Hint	Input focus model used by the client
StateHint	Initial state of the window
IconPixmapHint	Pixmap used as an icon
IconWindowHint	Window used as an icon
IconPositionHint	Initial position of icon
IconMaskHint	Pixmap to be used as a mask for the icon_pixmap
WindowGroupHint	ID of related window group
AllHints	The bitwise OR of InputHint, StateHint, IconPixmapHint, IconWindowHint, IconPositionHint, IconMaskHint, and WindowGroupHint

Note

The use of the AllHints mask is not recommended because the WM hints data structure may be extended in the future. If additional members are added to the WM hints data structure, the AllHints mask may not contain these new fields.

The following illustrates the WM hints data structure:

```
typedef struct {
        long flags;
        Bool input;
        int initial_state;
        Pixmap icon_pixmap;
        Window icon_window;
        int icon_x, icon_y;
        Pixmap icon_mask;
        XID window_group;
} XWMHints;
```

Table 12–3 defines the members of the data structure.

Table 12–3 WM Hints Data Structure Members

Member Name	Contents
flags	Specifies the members of the data structure that are defined.
input	Indicates whether or not the client relies on the window manager to get keyboard input.
initial_state	Defines how the window should appear in its initial configuration. Possible initial states are as follows:

Constant Name	Description
WithdrawnState	Neither client's top-level window nor icon window is visible.
NormalState	Client's top-level window is visible.
IconicState	Client's top-level window starts as an icon.

icon_pixmap	Identifies the pixmap used to create the window icon.
icon_window	Specifies the window to be used as an icon.
icon_x	Specifies the initial x-coordinate of the icon position.
icon_y	Specifies the initial y-coordinate of the icon position.
icon_mask	Specifies the pixels of the icon pixmap used to create the icon.
window_group	Specifies that a window belongs to a group of other windows.

12.3.2 Providing Size Hints

Xlib provides routines that the client can use to set or read the WM_NORMAL_HINTS property for a given window. These routines use the size hints data structure to communicate to the window manager about the size and position of windows in their normal and iconic startup states.

Use the SET WM NORMAL HINTS routine to set a window's WM_NORMAL_HINTS property. Use the GET WM NORMAL HINTS routine to read a window's WM_NORMAL_HINTS property. Example 12–2 illustrates the use of the SET WM NORMAL HINTS routine.

Table 12–4 lists the flags used in the size hints data structure.

Table 12–4 Size Hints Data Structure Flags

Flag Name	Description
USPosition	User-specified position of the window (obsolete)
USSize	User-specified size of the window (obsolete)
PPosition	Client-specified position of the window (obsolete)
PSize	Client-specified size of the window (obsolete)
PMinSize	Client-specified minimum size of the window
PMaxSize	Client-specified maximum size of the window
PResizeInc	Client-specified increments for resizing the window
PAspect	Client-specified minimum and maximum aspect ratios of the window
PBaseSize	Client-specified desired size of the window
PWinGravity	Client-specified window gravity
PAllHints	The bitwise OR of PPosition, PSize, PMinSize, PMaxSize, PResizeInc, and PAspect

Note

The use of the PAllHints mask is not recommended because this flag does not include the PBaseSize or PWinGravity masks. In addition, the size hints data structure may be extended in the future. If members are added to the data structure, the PAllHints mask may not contain these new fields.

The following illustrates the size hints data structure:

```
typedef struct {
        long flags;
        int x, y;             /*obsolete*/
        int width, height;    /*obsolete*/
        int min_width, min_height;
        int max_width, max_height;
        int width_inc, height_inc;
        struct{
            int x;
            int y;
        }min_aspect, max_aspect;
 int base_width, base_height;
 int win_gravity;
}XSizeHints;
```

Table 12–5 describes the data structure contents.

Note

The x, y, width, height members are obsolete and are left for compatibility reasons only.

Table 12–5 Size Hints Data Structure Members

Member Name	Contents
flags	Defines the members to which the client is assigning values
x	Specifies the x-coordinate that defines window position
y	Specifies the y-coordinate that defines window position
width	Defines the width of the window
height	Defines the height of the window
min_width	Specifies the minimum useful width of the window
min_height	Specifies the minimum useful height of the window
max_width	Specifies the maximum useful width of the window
max_height	Specifies the maximum useful height of the window
width_inc	Defines the increments by which the width of the window can be resized
height_inc	Defines the increments by which the height of the window can be resized
min_aspect_x[1]	Specifies the minimum aspect ratio of the window when used with the min_aspect y member.
min_aspect_y[1]	Specifies the minimum aspect ratio of the window with the min_aspect x member.
max_aspect_x[1]	Specifies the maximum aspect ratio of the window with the max_aspect y member.
max_aspect_y[1]	Specifies the maximum aspect ratio of the window with the max_aspect_x member.
base_width	Defines the desired width of the window
base_height	Defines the desired height of the window
win_gravity	Defines the region of the window that is to be retained when it is resized

[1]Setting the minimum and maximum aspects indicates the preferred range of the size of a window. An aspect is expressed in terms of a ratio between x and y.

By setting the max_width and max_height members, the client can set a meaningful maximum window size to make the maximize operation useful. If these members are not set, then the default maximum size of the window is the screen size.

12.3.3 Setting Window and Icon Names

Xlib includes routines to enable clients to define properties for communicating with the window manager about window names, icon names, and window classes. Use the SET WM NAME routine to set the WM_NAME property to display the name for a given window. Use the SET WM ICON NAME routine to set the WM_ICON_NAME property to set the name of a given window icon name. Example 12–2 illustrates how to use these routines.

Both the SET WM NAME and SET WM ICON NAME routines are convenience functions that use the text property data structure and call the SET TEXT PROPERTY routine. The following illustrates the text property data structure:

```
typedef struct{
    unsigned char *value;
    Atom encoding;
    int format;
    unsigned long items;
}XTextProperty
```

Table 12–6 describes the members of the data structure.

Table 12–6 Text Property Data Structure Members

Member Name	Contents
value	Character string
encoding	Type of property
format	Number of bits: 8, 16, or 32
nitems	Number of items in value

Xlib provides an additional routine to set a window name. The STORE NAME routine assigns a name to a window and displays it in a prominent place such as in the title bar. Example 1–1 in Chapter 1 uses the STORE NAME routine to define the name of the client's parent window, as follows:

```
XStoreName(dpy, window1, WindowName);
```

To define and get the class of a specified window, use the SET CLASS HINT and GET CLASS HINT routines. The routines refer to the class hint data structure. The following illustrates the class hint data structure:

```
typedef struct {
        char *res_name;
        char *res_class;
} XClassHint;
```

Table 12–7 defines the members of the data structure. For more information about a window's class, refer to Chapter 10.

Table 12–7 Class Hint Data Structure Members

Member Name	Contents
res_name	Defines the name of the window
res_class	Defines the class of the window

Note that the name defined in this data structure may differ from the name defined by the XA_WM_NAME property. The XA_WM_NAME property specifies what should be displayed in the title bar. Consequently, it may contain a temporary name, as in the name of a file that a client currently has in a buffer. In contrast to XA_WM_NAME, the res_name member defines the formal window name that clients should use when retrieving resources from the resource database. For more information about using the X resource manager, see Chapter 10.

12.3.4 Example of Setting Properties

This section includes an example that uses the following routines:

SET WM NORMAL HINTS
SET WM HINTS
SET WM WINDOW NAME
SET WM ICON NAME
SET CLASS HINT

Example 12–2 uses the following resource file to return values for such variables as window and icon position, name and size, font name, and colors. Using a resource data file is one method that the client can use to set properties. For more information about using the X resource manager, see Chapter 10.

```
star.window.name: Complying with ICCCM Conventions Example
star.icon.name: Star
star.font: -Adobe-New Century Schoolbook-Bold-R-NormaL--*-140-*-*-P-ISO8859-1
star.background: blue
star.foreground: yellow
star.border.color: black
star.border.width: 2
star.window.height: 400
star.window.width: 600
star.window.x: 60
star.window.y: 70
star.window.maxHeight: 375
star.window.maxWidth: 475
star.window.minHeight: 120
star.window.minWidth: 350
star.icon.x: 800
star.icon.y: 10
```

Example 12–2 maps one window. When the user clicks MB1 on the minimize button, the window manager uses a bitmap to create a star as the icon pixmap. Clicking on MB2 exits the program. See Chapter 7 for more information about pixmaps and bitmaps.

Example 12–2 Setting Window Manager Properties

```
#include <decw$include/Xlib.h>
#include <decw$include/Xutil.h>
#include <decw$include/Xresource.h>
#include <decw$include/Xatom.h>

❶#include "icon_file.dat"

Display *dpy;
Window win;
GC gc;
Screen *screen;
int scrNum;
Pixmap icon_win_pixmap;
```

(continued on next page)

Example 12–2 (Cont.) Setting Window Manager Properties

```
int n;
int winX, winY, winHeight, winWidth;
unsigned depth;
unsigned long bd, fg, bg;
unsigned long bw = 1;
char *name, *class;
char *fontname;
char *message[]={
    "This example demonstrates how applications",
    "can comply with ICCCM conventions."
    };

XrmDatabase star_db = 0;
XrmValue value;
XrmString type;
XWMHints *xwmh;
XSizeHints *xsh;
XClassHint *xch;
XSetWindowAttributes xswa;
XGCValues xgcv;
XTextProperty windowName, iconName;
                    .
                    .
                    .
/***** Create the Window using XSizeHints  *****/
static void doCreateWindows( )
{
❷   xsh = XAllocSizeHints();
    xsh->flags = PMinSize | PMaxSize;
    xsh->min_height = atoi (doGetResource("star.window.minHeight",
                                          "Star.Window.MinHeight"));
    xsh->min_width  = atoi (doGetResource("star.window.minWidth",
                                          "Star.Window.MinWidth"));
    xsh->max_height = atoi (doGetResource("star.window.maxHeight",
                                          "Star.Window.MaxHeight"));
    xsh->max_width  = atoi (doGetResource("star.window.maxWidth",
                                          "Star.Window.MaxWidth"));

    xswa.event_mask = ExposureMask | ButtonPressMask;
    xswa.background_pixel = doDefineColor(doGetResource("star.background",
                                                        "Star.Background"));

    winX = atoi (doGetResource("star.window.x", "Star.Window.X"));
    winY = atoi (doGetResource("star.window.y", "Star.Window.Y"));
    winWidth = atoi (doGetResource("star.window.width",
           "Star.Window.Width"));
    winHeight = atoi (doGetResource("star.window.height",
        "Star.Window.height"));

    win = XCreateWindow(dpy, DefaultRootWindow(dpy),
            winX, winY, winWidth, winHeight, 0,
            DefaultDepthOfScreen(screen), InputOutput,
            DefaultVisualOfScreen(screen), CWEventMask |
               CWBackPixel, &xswa);

❸   XSetWMNormalHints(dpy, win, xsh);
}
```

(continued on next page)

Example 12–2 (Cont.) Setting Window Manager Properties

```
/***** Set the window name and icon name *****/
static void doSetNames( )
{
❹  windowName.value = (doGetResource("star.window.name", "Star.Window.Name"));
    windowName.encoding = XA_STRING;
    windowName.format = 8;
    windowName.nitems = strlen(windowName.value);
❺  XSetWMName(dpy, win, &windowName);

    iconName.value =  (doGetResource("star.icon.name", "Star.Icon.Name"));
    iconName.encoding = XA_STRING;
    iconName.format = 8;
    iconName.nitems = strlen(iconName.value);
    XSetWMIconName(dpy, win, &iconName);
}

/***** Create the graphics context *****/
static void doCreateGraphicsContext( )
{

    xgcv.foreground = doDefineColor(doGetResource("star.foreground",
        "Star.Foreground"));
    xgcv.background = doDefineColor(doGetResource("star.background",
        "Star.Background"));
    gc = XCreateGC(dpy, win, (GCForeground | GCBackground), &xgcv);
}

/**** Use the WM Hints data structure *****/
static void doWMHints ( )
{
❻  xwmh = XAllocWMHints( );
    xwmh->flags = (InputHint | StateHint | IconPixmapHint
                   | IconWindowHint | IconPositionHint);
    xwmh->input = False;
    xwmh->initial_state = NormalState;

 /* Create the bitmap for the icon pixmap */

❼   if (xwmh->icon_pixmap = XCreateBitmapFromData(dpy, win, icon_file_bits,
         icon_file_width, icon_file_height)){
         depth=DefaultDepth(dpy, DefaultScreen(dpy));
         icon_win_pixmap = XCreatePixmap(dpy, DefaultRootWindow(dpy),
         icon_file_width, icon_file_height, depth);
         XCopyPlane(dpy, xwmh->icon_pixmap, icon_win_pixmap, gc, 0, 0,
             icon_file_width, icon_file_height, 0, 0, 1);

   /* Create the icon window */

❽        xswa.background_pixmap = icon_win_pixmap;
         xswa.border_pixel = bd;
         xwmh->icon_window = XCreateWindow(dpy, DefaultRootWindow(dpy),
             0, 0, icon_file_width, icon_file_height, bw,
         CopyFromParent, InputOutput, CopyFromParent,
         (CWBackPixmap | CWBorderPixel), &xswa);

         XFreePixmap (dpy, icon_win_pixmap);
         xwmh->icon_x = atoi (doGetResource("star.icon.x", "Star.Icon.X"));
         xwmh->icon_y = atoi (doGetResource("star.icon.y", "Star.Icon.Y"));
❾        XSetWMHints(dpy, win, xwmh);

     }
     else
         printf("Can't open bitmap file for icon\n");
```

(continued on next page)

Example 12–2 (Cont.) Setting Window Manager Properties

```
    /* Set the class hint data structure */
    xch = XAllocClassHint( );
    xch->res_name = "star";
    xch->res_class = "Star";
⓾   SetClassHint(dpy, win, xch);
}

/***** Load the font for text writing *****/
static void doLoadFont( )
{
    Font font;
    fontname = (doGetResource ("star.font", "Star.Font"));
    font = XLoadFont(dpy, fontname);
    XSetFont(dpy, gc, font);
}

/***** Map the window *****/
static void doMapWindow( )
{
 XMapWindow(dpy, win);
}
                    .
                    .
                    .
/***** Button press & shutdown *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
    if (eventP->xbutton.button == Button2) {
        XWithdrawWindow(dpy, win, scrNum);
        XFree(xwmh);
        XFree(xch);
        XFree(xsh);
        XCloseDisplay(dpy);
        exit(1);
     }
}
```

❶ The client specifies an include file, icon_file.dat, that contains the data to create an icon bitmap.

❷ The client calls the ALLOC SIZE HINTS routine to allocate a size hints data structure and return a pointer. In this example, the client sets the minimum and maximum size of the window. The client-defined *doGetResource* routine accesses the X resource manager database and sets the specified members of the data structure. For more information about the X resource manager, see Chapter 10.

❸ The SET WM NORMAL HINTS routine sets the window's WM_NORMAL_HINTS property with the values from the size hints data structure. The SET NORMAL HINTS routine has the following format:

XSetWMNormalHints(display, window, hints)

❹ The client uses the text property data structure to assign values for the window and icon name. The client sets the value member of the data structure by calling the client-defined *doGetResource* routine, which fetches the name of the window and the icon from the database.

❺ The client calls the SET WM NAME to set the property XA_WM_NAME. SET WM NAME is a convenience function that performs a SET TEXT PROPERTY on the XA_WM_NAME property. The SET WM NAME routine has the following format:

```
XSetWMName(display, window, text_property)
```

The client also sets the icon name in the same manner with the SET WM ICON NAME routine.

❻ By calling the ALLOC WM HINTS routine, the client allocates and sets the values of the WM hints data structure. Because the client does not expect any keyboard input, the input member is set to false. To create a mapped and visible window, the client sets the initial_state member to **NormalState**.

❼ The CREATE BITMAP FROM DATA routine creates a bitmap and returns the pixmap ID to the wm hints data structure. For more information about bitmaps and pixmaps refer to Chapter 7.

❽ The client creates a pixmap and copies the icon pixmap into it. When the client creates the icon window with the CREATE WINDOW routine, the icon pixmap is used as the background.

The example sets both the icon pixmap and the icon window, which displays a window for the client to use as an icon. MWM supports icon windows; however, not all window managers support icon windows. For reasons of portability, it is important that if the client sets the icon window, the client also set the icon pixmap as a backup.

❾ The SET WM HINTS routine sets all of the properties specified in the WM hints data structure.

❿ The name and class of the window can be set with the SET CLASS routine. Note that this is not the window name, but is the resource manager name and class. For more information about names and classes, see Chapter 10.

12.3.5 Using the SET WM PROPERTIES Routine

Xlib also provides a convenience function to set the standard window manager properties. Use the SET WM PROPERTIES routine to set the following window manager properties:

WM_CLASS
WM_CLIENT_MACHINE
WM_COMMAND
WM_HINTS
WM_ICON_NAME
WM_NAME
WM_NORMAL_HINTS

Example 12–3 illustrates how to use the SET WM PROPERTIES routine. This example is identical to Example 12–2 except that all properties are set at the end of the client-defined *doWMHints* routine.

Example 12–3 Using the SET WM PROPERTIES Routine

(continued on next page)

Example 12–3 (Cont.) Using the SET WM PROPERTIES Routine

```
#include <decw$include/Xlib.h>
#include <decw$include/Xutil.h>
#include <decw$include/Xresource.h>
#include <decw$include/Xatom.h>
#include "icon_file.dat"

Display *dpy;
Window win;
GC gc;
Screen *screen;
int scrNum;
Pixmap icon_win_pixmap;

int n;
int winX, winY, winHeight, winWidth;
unsigned depth;
unsigned long bd, fg, bg;
unsigned long bw = 1;
char *name, *class;
char *fontname;
char *message[]={
    "This example demonstrates how applications",
 "can set properties with one routine call."
 };

    "This example demonstrates how applications",
 "can set properties with one routine call."
 };

XrmDatabase star_db = 0;
XrmValue value;
XrmString type;
XWMHints *xwmh;
XSizeHints *xsh;
XClassHint *xch;
XSetWindowAttributes xswa;
XGCValues xgcv;
XTextProperty windowName, iconName;
                    .
                    .
                    .
/***** Create the Window using XSizeHints  *****/
static void doCreateWindows( )
{
    xsh = XAllocSizeHints();
    xsh->flags = PMinSize | PMaxSize;
    xsh->min_height = atoi (doGetResource("star.window.minHeight",
                                          "Star.Window.MinHeight"));
    xsh->min_width  = atoi (doGetResource("star.window.minWidth",
                                          "Star.Window.MinWidth"));
    xsh->max_height = atoi (doGetResource("star.window.maxHeight",
                                          "Star.Window.MaxHeight"));
    xsh->max_width  = atoi (doGetResource("star.window.maxWidth",
                                          "Star.Window.MaxWidth"));

    xswa.event_mask = ExposureMask | ButtonPressMask;
    xswa.background_pixel = doDefineColor(doGetResource("star.background",
                                                        "Star.Background"));
```

(continued on next page)

Example 12–3 (Cont.) Using the SET WM PROPERTIES Routine

```
    winX = atoi (doGetResource("star.window.x", "Star.Window.X"));
    winY = atoi (doGetResource("star.window.y", "Star.Window.Y"));
    winWidth = atoi (doGetResource("star.window.width",
                                   "Star.Window.Width"));
    winHeight = atoi (doGetResource("star.window.height",
                                    "Star.Window.height"));

    win = XCreateWindow(dpy, DefaultRootWindow(dpy), winX, winY,
              winWidth, winHeight, DefaultDepthOfScreen(screen),
              InputOutput, DefaultVisualOfScreen(screen), CWEventMask |
              CWBackPixel, &xswa);
}

/***** Set the window name and icon name *****/
static void doSetNames( )
{
    windowName.value = (doGetResource("star.window.name", "Star.Window.Name"));
    windowName.encoding = XA_STRING;
    windowName.format = 8;
    windowName.nitems = strlen(windowName.value);

    iconName.value =  (doGetResource("star.icon.name", "Star.Icon.Name"));
    iconName.encoding = XA_STRING;
    iconName.format = 8;
    iconName.nitems = strlen(iconName.value);
}

/***** Create the graphics context *****/
static void doCreateGraphicsContext( )
{
    xgcv.foreground = doDefineColor(doGetResource("star.foreground",
                                                  "Star.Foreground"));
    xgcv.background = doDefineColor(doGetResource("star.background",
                                                  "Star.Background"));
    gc = XCreateGC(dpy, win, (GCForeground | GCBackground), &xgcv);
}

/***** Use the WM Hints data structure *****/
static void doWMHints ( )
{
    xwmh = XAllocWMHints( );
    xwmh->flags = (InputHint | StateHint | IconPixmapHint
                   | IconWindowHint | IconPositionHint);
    xwmh->input = False;
    xwmh->initial_state = NormalState;

        /* Create the bitmap for the icon pixmap */
        if (xwmh->icon_pixmap = XCreateBitmapFromData(dpy, win, icon_file_bits,
        icon_file_width, icon_file_height)){

            depth=DefaultDepth(dpy, DefaultScreen(dpy));
            icon_win_pixmap = XCreatePixmap(dpy, DefaultRootWindow(dpy),
                icon_file_width, icon_file_height, depth);
            XCopyPlane(dpy, xwmh->icon_pixmap, icon_win_pixmap, gc, 0, 0,
                icon_file_width, icon_file_height, 0, 0, 1);
```

(continued on next page)

Example 12–3 (Cont.) Using the SET WM PROPERTIES Routine

```
            /* Create the icon window */
            xswa.background_pixmap = icon_win_pixmap;
            xswa.border_pixel = bd;
            xwmh->icon_window = XCreateWindow(dpy, DefaultRootWindow(dpy),
                   0, 0, icon_file_width, icon_file_height, bw,
                   CopyFromParent, InputOutput, CopyFromParent,
                   (CWBackPixmap | CWBorderPixel), &xswa);
            XFreePixmap (dpy, icon_win_pixmap);
            xwmh->icon_x = atoi (doGetResource("star.icon.x", "Star.Icon.X"));
            xwmh->icon_y = atoi (doGetResource("star.icon.y", "Star.Icon.Y"));
    }else
            printf("Can't open bitmap file for icon\n");

        /* Set the class hint data structure */
        xch = XAllocClassHint( );
        xch->res_name = "star";
        xch->res_class = "Star";

❶SetWMProperties(dpy, win, &windowName, &iconName,
    NULL, NULL, xsh, xwmh, xch);
}

/***** Load the font for text writing *****/
static void doLoadFont( )
{
    Font font;
    fontname = (doGetResource ("star.font", "Star.Font"));
    font = XLoadFont(dpy, fontname);
    XSetFont(dpy, gc, font);
}

/***** Map the window *****/
static void doMapWindow( )
{
 XMapWindow(dpy, win);
}
                    .
                    .
                    .
/***** Button press & shutdown *****/
static void doButtonPress(eventP)
XEvent *eventP;
{
    if (eventP->xbutton.button == Button2) {
        XWithdrawWindow(dpy, win, scrNum);
        XFree(xwmh);
        XFree(xch);
        XFree(xsh);
        XCloseDisplay(dpy);
        exit(1);
        }
}
```

❶ The client uses the convenience routine SET WM PROPERTIES to set the standard window manager properties. The SET WM PROPERTIES routine has the following format:

XSetWMProperties(display, window, window_name, icon_name,
argv, argc, normal_hints, wm_hints, class_hints)

A

Compiling Fonts

VMS DECwindows includes a font compiler that enables programmers to convert an ASCII Bitmap Distribution Format (BDF) font into a binary server natural font (SNF). For information about the Bitmap Distribution Format, see the *X Window System*. The server uses an SNF file to display a font. In addition to converting the BDF file to binary form, the compiler provides statistical information about the font and the compilation process.

To invoke the font compiler, use the following DCL format:

```
FONT filename [
              /[NO]OUTPUT[=output_file]
              /[NO]MINBBOX
              /[NO]REPORT[=report_file]
              ]
```

The filename parameter specifies the BDF file to be compiled. A file name is required. The default file type is DECW$BDF.

The optional /OUTPUT qualifier specifies the file name of the resulting SNF file. The default output file name is the file name of the BDF file being compiled. The default output SNF file type is DECW$FONT. The default is /OUTPUT.

Compiler output consists of an SNF file that contains font information, character metrics, and the image of each character in the font. Font information in the SNF file is essentially the same as information stored in the font struct data structure. For a description of the data structure, see Section 8.1.

The optional /MINBBOX qualifier specifies that the compiler produce the minimum bounding box for each character in the font and adjust values for the left bearing, right bearing, ascent, and descent of each character accordingly. Character width is not affected. Specifying the /MINBBOX qualifier is equivalent to converting a fixed font to a monospaced font. For a description of character metrics and fonts, see Section 8.1. The default is /NOMINBBOX.

Using the /MINBBOX qualifier has two advantages. Because the font compiler produces minimum instead of fixed bounding boxes, the resulting SNF file is significantly smaller than the comparable fixed font SNF file. Consequently, both disk requirements for storing the font and server memory requirements when a client loads the font are reduced. In addition, because the resulting font comprises minimum inkable characters, server performance when writing text is increased.

The optional /REPORT qualifier directs the compiler to report information about the font and the compilation process, including BDF information, font properties, compiler generation information, and metrics. The /REPORT qualifier also causes the compiler to illustrate each glyph in the font. The default report file name is the file name of the BDF file being compiled. The default report file type is DECW$REP. The default is /NOREPORT.

B

VMS DECwindows Named Colors

VMS DECwindows provides the X Windows Release 4 named colors. For a list of all VMS DECwindows named colors and their RGB values, see SYS$MANAGER:DECW$RGB.COM. For a description of using named colors, see Section 5.3.1.

In addition to common named colors, VMS DECwindows also provides the following colors that are specific to Digital:

DECWBlue
Screen Background
Border Topshadow
Border Background
Border Bottomshadow
Window Topshadow
Window Background
Window Bottomshadow

Please note that color display is device-dependent. You can use a color mixing dialog box to see how a particular named color displays on your system. The following procedure describes one way to display this dialog box:

1. Choose Screen Background... from the Session Manager's Options Menu.

   The Session Manager displays the Screen Background Options dialog box.

2. In this dialog box, click on the Screen Foreground Color or Screen Background Color buttons.

   The Session Manager displays a color mixing dialog box.

3. Choose Browser from the Color Model menu.

For more information about using the Color Mix dialog box, see the *Using DECwindows Motif for OpenVMS*.

C

VMS DECwindows Fonts

Table C–1 lists VMS DECwindows 75 dpi fonts and their file names. Table C–2 lists VMS DECwindows 100 dpi fonts and their file names. Table C–3 lists VMS DECwindows Common Fonts. These fonts can be used with both 75 dpi and 100 dpi monitors. Table C–3 also lists font aliases for the fixed width fonts. For information about using fonts, see Chapter 8.

Note that a double dash occurs between the fifth and seventh fields of the font name. For example, the full XLFD name of a representative font is as follows:

```
-Adobe-ITC Avant Garde Gothic-Book-R-Normal--11-80-100-100-P-59-ISO8859-1
```

An example that shows how to use a file name as a font alias is provided in the following file: DECW$EXAMPLES:DECW$FONT_ALIAS_FILENAMES.DAT.

Table C–1 VMS DECwindows 75 dpi Fonts

File Name	Font Name
FIXED	fixed
DECW$SESSION	DECW$SESSION
VARIABLE	variable
Avant Garde	
AVANTGARDE_BOOK8	-Adobe-ITC Avant Garde Gothic-Book-R-Normal--8-80-75-75-P-49-ISO8859-1
AVANTGARDE_BOOK10	-Adobe-ITC Avant Garde Gothic-Book-R-Normal--10-100-75-75-P-59-ISO8859-1
AVANTGARDE_BOOK12	-Adobe-ITC Avant Garde Gothic-Book-R-Normal--12-120-75-75-P-70-ISO8859-1
AVANTGARDE_BOOK14	-Adobe-ITC Avant Garde Gothic-Book-R-Normal--14-140-75-75-P-80-ISO8859-1
AVANTGARDE_BOOK18	-Adobe-ITC Avant Garde Gothic-Book-R-Normal--18-180-75-75-P-103-ISO8859-1
AVANTGARDE_BOOK24	-Adobe-ITC Avant Garde Gothic-Book-R-Normal--24-240-75-75-P-138-ISO8859-1
AVANTGARDE_ BOOKOBLIQUE8	-Adobe-ITC Avant Garde Gothic-Book-O-Normal--8-80-75-75-P-49-ISO8859-1
AVANTGARDE_ BOOKOBLIQUE10	-Adobe-ITC Avant Garde Gothic-Book-O-Normal--10-100-75-75-P-59-ISO8859-1
AVANTGARDE_ BOOKOBLIQUE12	-Adobe-ITC Avant Garde Gothic-Book-O-Normal--12-120-75-75-P-69-ISO8859-1
AVANTGARDE_ BOOKOBLIQUE14	-Adobe-ITC Avant Garde Gothic-Book-O-Normal--14-140-75-75-P-81-ISO8859-1
AVANTGARDE_ BOOKOBLIQUE18	-Adobe-ITC Avant Garde Gothic-Book-O-Normal--18-180-75-75-P-103-ISO8859-1

(continued on next page)

Table C–1 (Cont.) VMS DECwindows 75 dpi Fonts

File Name	Font Name
Avant Garde	
AVANTGARDE_BOOKOBLIQUE24	-Adobe-ITC Avant Garde Gothic-Book-O-Normal--24-240-75-75-P-138-ISO8859-1
AVANTGARDE_DEMI8	-Adobe-ITC Avant Garde Gothic-Demi-R-Normal--8-80-75-75-P-51-ISO8859-1
AVANTGARDE_DEMI10	-Adobe-ITC Avant Garde Gothic-Demi-R-Normal--10-100-75-75-P-61-ISO8859-1
AVANTGARDE_DEMI12	-Adobe-ITC Avant Garde Gothic-Demi-R-Normal--12-120-75-75-P-70-ISO8859-1
AVANTGARDE_DEMI14	-Adobe-ITC Avant Garde Gothic-Demi-R-Normal--14-140-75-75-P-82-ISO8859-1
AVANTGARDE_DEMI18	-Adobe-ITC Avant Garde Gothic-Demi-R-Normal--18-180-75-75-P-105-ISO8859-1
AVANTGARDE_DEMI24	-Adobe-ITC Avant Garde Gothic-Demi-R-Normal--24-240-75-75-P-140-ISO8859-1
AVANTGARDE_DEMIOBLIQUE8	-Adobe-ITC Avant Garde Gothic-Demi-O-Normal--8-80-75-75-P-51-ISO8859-1
AVANTGARDE_DEMIOBLIQUE10	-Adobe-ITC Avant Garde Gothic-Demi-O-Normal--10-100-75-75-P-61-ISO8859-1
AVANTGARDE_DEMIOBLIQUE12	-Adobe-ITC Avant Garde Gothic-Demi-O-Normal--12-120-75-75-P-71-ISO8859-1
AVANTGARDE_DEMIOBLIQUE14	-Adobe-ITC Avant Garde Gothic-Demi-O-Normal--14-140-75-75-P-82-ISO8859-1
AVANTGARDE_DEMIOBLIQUE18	-Adobe-ITC Avant Garde Gothic-Demi-O-Normal--18-180-75-75-P-103-ISO8859-1
AVANTGARDE_DEMIOBLIQUE24	-Adobe-ITC Avant Garde Gothic-Demi-O-Normal--24-240-75-75-P-139-ISO8859-1
Courier	
COURIER8	-Adobe-Courier-Medium-R-Normal--8-80-75-75-M-50-ISO8859-1
COURIER10	-Adobe-Courier-Medium-R-Normal--10-100-75-75-M-60-ISO8859-1
COURIER12	-Adobe-Courier-Medium-R-Normal--12-120-75-75-M-70-ISO8859-1
COURIER14	-Adobe-Courier-Medium-R-Normal--14-140-75-75-M-90-ISO8859-1
COURIER18	-Adobe-Courier-Medium-R-Normal--18-180-75-75-M-110-ISO8859-1
COURIER24	-Adobe-Courier-Medium-R-Normal--24-240-75-75-M-150-ISO8859-1
COURIER_BOLD8	-Adobe-Courier-Bold-R-Normal--8-80-75-75-M-50-ISO8859-1
COURIER_BOLD10	-Adobe-Courier-Bold-R-Normal--10-100-75-75-M-60-ISO8859-1
COURIER_BOLD12	-Adobe-Courier-Bold-R-Normal--12-120-75-75-M-70-ISO8859-1
COURIER_BOLD14	-Adobe-Courier-Bold-R-Normal--14-140-75-75-M-90-ISO8859-1
COURIER_BOLD18	-Adobe-Courier-Bold-R-Normal--18-180-75-75-M-110-ISO8859-1

(continued on next page)

Table C–1 (Cont.) VMS DECwindows 75 dpi Fonts

File Name	Font Name
Courier	
COURIER_BOLD24	-Adobe-Courier-Bold-R-Normal--24-240-75-75-M-150-ISO8859-1
COURIER_BOLDOBLIQUE8	-Adobe-Courier-Bold-O-Normal--8-80-75-75-M-50-ISO8859-1
COURIER_BOLDOBLIQUE10	-Adobe-Courier-Bold-O-Normal--10-100-75-75-M-60-ISO8859-1
COURIER_BOLDOBLIQUE12	-Adobe-Courier-Bold-O-Normal--12-120-75-75-M-70-ISO8859-1
COURIER_BOLDOBLIQUE14	-Adobe-Courier-Bold-O-Normal--14-140-75-75-M-90-ISO8859-1
COURIER_BOLDOBLIQUE18	-Adobe-Courier-Bold-O-Normal--18-180-75-75-M-110-ISO8859-1
COURIER_BOLDOBLIQUE24	-Adobe-Courier-Bold-O-Normal--24-240-75-75-M-150-ISO8859-1
COURIER_OBLIQUE8	-Adobe-Courier-Medium-O-Normal--8-80-75-75-M-50-ISO8859-1
COURIER_OBLIQUE10	-Adobe-Courier-Medium-O-Normal--10-100-75-75-M-60-ISO8859-1
COURIER_OBLIQUE12	-Adobe-Courier-Medium-O-Normal--12-120-75-75-M-70-ISO8859-1
COURIER_OBLIQUE14	-Adobe-Courier-Medium-O-Normal--14-140-75-75-M-90-ISO8859-1
COURIER_OBLIQUE18	-Adobe-Courier-Medium-O-Normal--18-180-75-75-M-110-ISO8859-1
COURIER_OBLIQUE24	-Adobe-Courier-Medium-O-Normal--24-240-75-75-M-150-ISO8859-1
DEC Math	
DUTCH801_DECMATH_EXTENSION8	-Bitstream-Dutch 801-Medium-R-Normal--31-80-75-75-P-244-DEC-DECmath_Extension
DUTCH801_DECMATH_EXTENSION10	-Bitstream-Dutch 801-Medium-R-Normal--39-100-75-75-P-307-DEC-DECmath_Extension
DUTCH801_DECMATH_EXTENSION12	-Bitstream-Dutch 801-Medium-R-Normal--46-120-75-75-P-362-DEC-DECmath_Extension
DUTCH801_DECMATH_EXTENSION14	-Bitstream-Dutch 801-Medium-R-Normal--54-140-75-75-P-425-DEC-DECmath_Extension
DUTCH801_DECMATH_ITALIC8	-Bitstream-Dutch 801-Medium-I-Normal--8-80-75-75-P-45-DEC-DECmath_Italic
DUTCH801_DECMATH_ITALIC10	-Bitstream-Dutch 801-Medium-I-Normal--10-100-75-75-P-56-DEC-DECmath_Italic
DUTCH801_DECMATH_ITALIC12	-Bitstream-Dutch 801-Medium-I-Normal--12-120-75-75-P-67-DEC-DECmath_Italic
DUTCH801_DECMATH_ITALIC14	-Bitstream-Dutch 801-Medium-I-Normal--15-140-75-75-P-83-DEC-DECmath_Italic
DUTCH801_DECMATH_SYMBOL8	-Bitstream-Dutch 801-Medium-R-Normal--8-80-75-75-P-62-DEC-DECmath_Symbol
DUTCH801_DECMATH_SYMBOL10	-Bitstream-Dutch 801-Medium-R-Normal--10-100-75-75-P-77-DEC-DECmath_Symbol
DUTCH801_DECMATH_SYMBOL12	-Bitstream-Dutch 801-Medium-R-Normal--12-120-75-75-P-92-DEC-DECmath_Symbol
DUTCH801_DECMATH_SYMBOL14	-Bitstream-Dutch 801-Medium-R-Normal--15-140-75-75-P-115-DEC-DECmath_Symbol

(continued on next page)

Table C–1 (Cont.) VMS DECwindows 75 dpi Fonts

File Name	Font Name
Helvetica	
HELVETICA8	-Adobe-Helvetica-Medium-R-Normal--8-80-75-75-P-46-ISO8859-1
HELVETICA10	-Adobe-Helvetica-Medium-R-Normal--10-100-75-75-P-56-ISO8859-1
HELVETICA12	-Adobe-Helvetica-Medium-R-Normal--12-120-75-75-P-67-ISO8859-1
HELVETICA14	-Adobe-Helvetica-Medium-R-Normal--14-140-75-75-P-77-ISO8859-1
HELVETICA18	-Adobe-Helvetica-Medium-R-Normal--18-180-75-75-P-98-ISO8859-1
HELVETICA24	-Adobe-Helvetica-Medium-R-Normal--24-240-75-75-P-130-ISO8859-1
HELVETICA_BOLD8	-Adobe-Helvetica-Bold-R-Normal--8-80-75-75-P-50-ISO8859-1
HELVETICA_BOLD10	-Adobe-Helvetica-Bold-R-Normal--10-100-75-75-P-60-ISO8859-1
HELVETICA_BOLD12	-Adobe-Helvetica-Bold-R-Normal--12-120-75-75-P-70-ISO8859-1
HELVETICA_BOLD14	-Adobe-Helvetica-Bold-R-Normal--14-140-75-75-P-82-ISO8859-1
HELVETICA_BOLD18	-Adobe-Helvetica-Bold-R-Normal--18-180-75-75-P-103-ISO8859-1
HELVETICA_BOLD24	-Adobe-Helvetica-Bold-R-Normal--24-240-75-75-P-138-ISO8859-1
HELVETICA_ BOLDOBLIQUE8	-Adobe-Helvetica-Bold-O-Normal--8-80-75-75-P-50-ISO8859-1
HELVETICA_ BOLDOBLIQUE10	-Adobe-Helvetica-Bold-O-Normal--10-100-75-75-P-60-ISO8859-1
HELVETICA_ BOLDOBLIQUE12	-Adobe-Helvetica-Bold-O-Normal--12-120-75-75-P-69-ISO8859-1
HELVETICA_ BOLDOBLIQUE14	-Adobe-Helvetica-Bold-O-Normal--14-140-75-75-P-82-ISO8859-1
HELVETICA_ BOLDOBLIQUE18	-Adobe-Helvetica-Bold-O-Normal--18-180-75-75-P-104-ISO8859-1
HELVETICA_ BOLDOBLIQUE24	-Adobe-Helvetica-Bold-O-Normal--24-240-75-75-P-138-ISO8859-1
HELVETICA_OBLIQUE8	-Adobe-Helvetica-Medium-O-Normal--8-80-75-75-P-47-ISO8859-1
HELVETICA_OBLIQUE10	-Adobe-Helvetica-Medium-O-Normal--10-100-75-75-P-57-ISO8859-1
HELVETICA_OBLIQUE12	-Adobe-Helvetica-Medium-O-Normal--12-120-75-75-P-67-ISO8859-1
HELVETICA_OBLIQUE14	-Adobe-Helvetica-Medium-O-Normal--14-140-75-75-P-78-ISO8859-1
HELVETICA_OBLIQUE18	-Adobe-Helvetica-Medium-O-Normal--18-180-75-75-P-98-ISO8859-1
HELVETICA_OBLIQUE24	-Adobe-Helvetica-Medium-O-Normal--24-240-75-75-P-130-ISO8859-1
Interim DEC Math	
INTERIM_DM_EXTENSION14	-Adobe-Interim DM-Medium-I-Normal--14-140-75-75-P-140-DEC-DECMATH_ EXTENSION
INTERIM_DM_ITALIC14	-Adobe-Interim DM-Medium-I-Normal--14-140-75-75-P-140-DEC-DECMATH_ITALIC
INTERIM_DM_SYMBOL14	-Adobe-Interim DM-Medium-I-Normal--14-140-75-75-P-140-DEC-DECMATH_ SYMBOL

(continued on next page)

Table C–1 (Cont.) VMS DECwindows 75 dpi Fonts

File Name	Font Name
Lubalin Graph	
LUBALINGRAPH_BOOK8	-Adobe-ITC Lubalin Graph-Book-R-Normal--8-80-75-75-P-50-ISO8859-1
LUBALINGRAPH_BOOK10	-Adobe-ITC Lubalin Graph-Book-R-Normal--10-100-75-75-P-60-ISO8859-1
LUBALINGRAPH_BOOK12	-Adobe-ITC Lubalin Graph-Book-R-Normal--12-120-75-75-P-70-ISO8859-1
LUBALINGRAPH_BOOK14	-Adobe-ITC Lubalin Graph-Book-R-Normal--14-140-75-75-P-81-ISO8859-1
LUBALINGRAPH_BOOK18	-Adobe-ITC Lubalin Graph-Book-R-Normal--18-180-75-75-P-106-ISO8859-1
LUBALINGRAPH_BOOK24	-Adobe-ITC Lubalin Graph-Book-R-Normal--24-240-75-75-P-139-ISO8859-1
LUBALINGRAPH_ BOOKOBLIQUE8	-Adobe-ITC Lubalin Graph-Book-O-Normal--8-80-75-75-P-50-ISO8859-1
LUBALINGRAPH_ BOOKOBLIQUE10	-Adobe-ITC Lubalin Graph-Book-O-Normal--10-100-75-75-P-60-ISO8859-1
LUBALINGRAPH_ BOOKOBLIQUE12	-Adobe-ITC Lubalin Graph-Book-O-Normal--12-120-75-75-P-70-ISO8859-1
LUBALINGRAPH_ BOOKOBLIQUE14	-Adobe-ITC Lubalin Graph-Book-O-Normal--14-140-75-75-P-82-ISO8859-1
LUBALINGRAPH_ BOOKOBLIQUE18	-Adobe-ITC Lubalin Graph-Book-O-Normal--18-180-75-75-P-105-ISO8859-1
LUBALINGRAPH_ BOOKOBLIQUE24	-Adobe-ITC Lubalin Graph-Book-O-Normal--24-240-75-75-P-140-ISO8859-1
LUBALINGRAPH_DEMI8	-Adobe-ITC Lubalin Graph-Demi-R-Normal--8-80-75-75-P-51-ISO8859-1
LUBALINGRAPH_DEMI10	-Adobe-ITC Lubalin Graph-Demi-R-Normal--10-100-75-75-P-61-ISO8859-1
LUBALINGRAPH_DEMI12	-Adobe-ITC Lubalin Graph-Demi-R-Normal--12-120-75-75-P-73-ISO8859-1
LUBALINGRAPH_DEMI14	-Adobe-ITC Lubalin Graph-Demi-R-Normal--14-140-75-75-P-85-ISO8859-1
LUBALINGRAPH_DEMI18	-Adobe-ITC Lubalin Graph-Demi-R-Normal--18-180-75-75-P-109-ISO8859-1
LUBALINGRAPH_DEMI24	-Adobe-ITC Lubalin Graph-Demi-R-Normal--24-240-75-75-P-144-ISO8859-1
LUBALINGRAPH_ DEMIOBLIQUE8	-Adobe-ITC Lubalin Graph-Demi-O-Normal--8-80-75-75-P-52-ISO8859-1
LUBALINGRAPH_ DEMIOBLIQUE10	-Adobe-ITC Lubalin Graph-Demi-O-Normal--10-100-75-75-P-62-ISO8859-1
LUBALINGRAPH_ DEMIOBLIQUE12	-Adobe-ITC Lubalin Graph-Demi-O-Normal--12-120-75-75-P-74-ISO8859-1
LUBALINGRAPH_ DEMIOBLIQUE14	-Adobe-ITC Lubalin Graph-Demi-O-Normal--14-140-75-75-P-85-ISO8859-1
LUBALINGRAPH_ DEMIOBLIQUE18	-Adobe-ITC Lubalin Graph-Demi-O-Normal--18-180-75-75-P-109-ISO8859-1
LUBALINGRAPH_ DEMIOBLIQUE24	-Adobe-ITC Lubalin Graph-Demi-O-Normal--24-240-75-75-P-144-ISO8859-1
Menu	
MENU10	-Bigelow & Holmes-Menu-Medium-R-Normal--10-100-75-75-P-56-ISO8859-1
MENU12	-Bigelow & Holmes-Menu-Medium-R-Normal--12-120-75-75-P-70-ISO8859-1

(continued on next page)

Table C–1 (Cont.) VMS DECwindows 75 dpi Fonts

File Name	Font Name
New Century Schoolbook	
NEWCENTURYSCHLBK_BOLD8	-Adobe-New Century Schoolbook-Bold-R-Normal--8-80-75-75-P-56-ISO8859-1
NEWCENTURYSCHLBK_BOLD10	-Adobe-New Century Schoolbook-Bold-R-Normal--10-100-75-75-P-66-ISO8859-1
NEWCENTURYSCHLBK_BOLD12	-Adobe-New Century Schoolbook-Bold-R-Normal--12-120-75-75-P-77-ISO8859-1
NEWCENTURYSCHLBK_BOLD14	-Adobe-New Century Schoolbook-Bold-R-Normal--14-140-75-75-P-87-ISO8859-1
NEWCENTURYSCHLBK_BOLD18	-Adobe-New Century Schoolbook-Bold-R-Normal--18-180-75-75-P-113-ISO8859-1
NEWCENTURYSCHLBK_BOLD24	-Adobe-New Century Schoolbook-Bold-R-Normal--24-240-75-75-P-149-ISO8859-1
NEWCENTURYSCHLBK_BOLDITALIC8	-Adobe-New Century Schoolbook-Bold-I-Normal--8-80-75-75-P-56-ISO8859-1
NEWCENTURYSCHLBK_BOLDITALIC10	-Adobe-New Century Schoolbook-Bold-I-Normal--10-100-75-75-P-66-ISO8859-1
NEWCENTURYSCHLBK_BOLDITALIC12	-Adobe-New Century Schoolbook-Bold-I-Normal--12-120-75-75-P-76-ISO8859-1
NEWCENTURYSCHLBK_BOLDITALIC14	-Adobe-New Century Schoolbook-Bold-I-Normal--14-140-75-75-P-88-ISO8859-1
NEWCENTURYSCHLBK_BOLDITALIC18	-Adobe-New Century Schoolbook-Bold-I-Normal--18-180-75-75-P-111-ISO8859-1
NEWCENTURYSCHLBK_BOLDITALIC24	-Adobe-New Century Schoolbook-Bold-I-Normal--24-240-75-75-P-148-ISO8859-1
NEWCENTURYSCHLBK_ITALIC8	-Adobe-New Century Schoolbook-Medium-I-Normal--8-80-75-75-P-50-ISO8859-1
NEWCENTURYSCHLBK_ITALIC10	-Adobe-New Century Schoolbook-Medium-I-Normal--10-100-75-75-P-60-ISO8859-1
NEWCENTURYSCHLBK_ITALIC12	-Adobe-New Century Schoolbook-Medium-I-Normal--12-120-75-75-P-70-ISO8859-1
NEWCENTURYSCHLBK_ITALIC14	-Adobe-New Century Schoolbook-Medium-I-Normal--14-140-75-75-P-81-ISO8859-1
NEWCENTURYSCHLBK_ITALIC18	-Adobe-New Century Schoolbook-Medium-I-Normal--18-180-75-75-P-104-ISO8859-1
NEWCENTURYSCHLBK_ITALIC24	-Adobe-New Century Schoolbook-Medium-I-Normal--24-240-75-75-P-136-ISO8859-1
NEWCENTURYSCHLBK_ROMAN8	-Adobe-New Century Schoolbook-Medium-R-Normal--8-80-75-75-P-50-ISO8859-1

(continued on next page)

Table C–1 (Cont.) VMS DECwindows 75 dpi Fonts

File Name	Font Name
New Century Schoolbook	
NEWCENTURYSCHLBK_ ROMAN10	-Adobe-New Century Schoolbook-Medium-R-Normal--10-100-75-75-P-60-ISO8859-1
NEWCENTURYSCHLBK_ ROMAN12	-Adobe-New Century Schoolbook-Medium-R-Normal--12-120-75-75-P-70-ISO8859-1
NEWCENTURYSCHLBK_ ROMAN14	-Adobe-New Century Schoolbook-Medium-R-Normal--14-140-75-75-P-82-ISO8859-1
NEWCENTURYSCHLBK_ ROMAN18	-Adobe-New Century Schoolbook-Medium-R-Normal--18-180-75-75-P-103-ISO8859-1
NEWCENTURYSCHLBK_ ROMAN24	-Adobe-New Century Schoolbook-Medium-R-Normal--24-240-75-75-P-137-ISO8859-1
Souvenir	
SOUVENIR_DEMI8	-Adobe-ITC Souvenir-Demi-R-Normal--8-80-75-75-P-52-ISO8859-1
SOUVENIR_DEMI10	-Adobe-ITC Souvenir-Demi-R-Normal--10-100-75-75-P-62-ISO8859-1
SOUVENIR_DEMI12	-Adobe-ITC Souvenir-Demi-R-Normal--12-120-75-75-P-75-ISO8859-1
SOUVENIR_DEMI14	-Adobe-ITC Souvenir-Demi-R-Normal--14-140-75-75-P-90-ISO8859-1
SOUVENIR_DEMI18	-Adobe-ITC Souvenir-Demi-R-Normal--18-180-75-75-P-112-ISO8859-1
SOUVENIR_DEMI24	-Adobe-ITC Souvenir-Demi-R-Normal--24-240-75-75-P-149-ISO8859-1
SOUVENIR_DEMIITALIC8	-Adobe-ITC Souvenir-Demi-I-Normal--8-80-75-75-P-57-ISO8859-1
SOUVENIR_DEMIITALIC10	-Adobe-ITC Souvenir-Demi-I-Normal--10-100-75-75-P-67-ISO8859-1
SOUVENIR_DEMIITALIC12	-Adobe-ITC Souvenir-Demi-I-Normal--12-120-75-75-P-78-ISO8859-1
SOUVENIR_DEMIITALIC14	-Adobe-ITC Souvenir-Demi-I-Normal--14-140-75-75-P-92-ISO8859-1
SOUVENIR_DEMIITALIC18	-Adobe-ITC Souvenir-Demi-I-Normal--18-180-75-75-P-115-ISO8859-1
SOUVENIR_DEMIITALIC24	-Adobe-ITC Souvenir-Demi-I-Normal--24-240-75-75-P-154-ISO8859-1
SOUVENIR_LIGHT8	-Adobe-ITC Souvenir-Light-R-Normal--8-80-75-75-P-46-ISO8859-1
SOUVENIR_LIGHT10	-Adobe-ITC Souvenir-Light-R-Normal--10-100-75-75-P-56-ISO8859-1
SOUVENIR_LIGHT12	-Adobe-ITC Souvenir-Light-R-Normal--12-120-75-75-P-68-ISO8859-1
SOUVENIR_LIGHT14	-Adobe-ITC Souvenir-Light-R-Normal--14-140-75-75-P-79-ISO8859-1
SOUVENIR_LIGHT18	-Adobe-ITC Souvenir-Light-R-Normal--18-180-75-75-P-102-ISO8859-1
SOUVENIR_LIGHT24	-Adobe-ITC Souvenir-Light-R-Normal--24-240-75-75-P-135-ISO8859-1
SOUVENIR_LIGHTITALIC8	-Adobe-ITC Souvenir-Light-I-Normal--8-80-75-75-P-49-ISO8859-1
SOUVENIR_LIGHTITALIC10	-Adobe-ITC Souvenir-Light-I-Normal--10-100-75-75-P-59-ISO8859-1
SOUVENIR_LIGHTITALIC12	-Adobe-ITC Souvenir-Light-I-Normal--12-120-75-75-P-69-ISO8859-1
SOUVENIR_LIGHTITALIC14	-Adobe-ITC Souvenir-Light-I-Normal--14-140-75-75-P-82-ISO8859-1
SOUVENIR_LIGHTITALIC18	-Adobe-ITC Souvenir-Light-I-Normal--18-180-75-75-P-104-ISO8859-1
SOUVENIR_LIGHTITALIC24	-Adobe-ITC Souvenir-Light-I-Normal--24-240-75-75-P-139-ISO8859-1

(continued on next page)

Table C–1 (Cont.) VMS DECwindows 75 dpi Fonts

File Name	Font Name
Symbol	
SYMBOL8	-Adobe-Symbol-Medium-R-Normal--8-80-75-75-P-51-ADOBE-FONTSPECIFIC
SYMBOL10	-Adobe-Symbol-Medium-R-Normal--10-100-75-75-P-61-ADOBE-FONTSPECIFIC
SYMBOL12	-Adobe-Symbol-Medium-R-Normal--12-120-75-75-P-74-ADOBE-FONTSPECIFIC
SYMBOL14	-Adobe-Symbol-Medium-R-Normal--14-140-75-75-P-85-ADOBE-FONTSPECIFIC
SYMBOL18	-Adobe-Symbol-Medium-R-Normal--18-180-75-75-P-107-ADOBE-FONTSPECIFIC
SYMBOL24	-Adobe-Symbol-Medium-R-Normal--24-240-75-75-P-142-ADOBE-FONTSPECIFIC
Terminal	
TERMINAL14	-DEC-Terminal-Medium-R-Normal--14-140-75-75-C-80-ISO8859-1
TERMINAL18	-Bitstream-Terminal-Medium-R-Normal--18-180-75-75-C-110-ISO8859-1
TERMINAL28	-DEC-Terminal-Medium-R-Normal--28-280-75-75-C-160-ISO8859-1
TERMINAL36	-Bitstream-Terminal-Medium-R-Normal--36-360-75-75-C-220-ISO8859-1
TERMINAL_BOLD14	-DEC-Terminal-Bold-R-Normal--14-140-75-75-C-80-ISO8859-1
TERMINAL_BOLD18	-Bitstream-Terminal-Bold-R-Normal--18-180-75-75-C-110-ISO8859-1
TERMINAL_BOLD28	-DEC-Terminal-Bold-R-Normal--28-280-75-75-C-160-ISO8859-1
TERMINAL_BOLD36	-Bitstream-Terminal-Bold-R-Normal--36-360-75-75-C-220-ISO8859-1
TERMINAL_BOLD_ DBLWIDE14	-DEC-Terminal-Bold-R-Double Wide--14-140-75-75-C-160-ISO8859-1
TERMINAL_BOLD_ DBLWIDE18	-Bitstream-Terminal-Bold-R-Double Wide--18-180-75-75-C-220-ISO8859-1
TERMINAL_BOLD_ DBLWIDE_DECTECH14	-DEC-Terminal-Bold-R-Double Wide--14-140-75-75-C-160-DEC-DECtech
TERMINAL_BOLD_ DBLWIDE_DECTECH18	-Bitstream-Terminal-Bold-R-Double Wide--18-180-75-75-C-220-DEC-DECtech
TERMINAL_BOLD_ DECTECH14	-DEC-Terminal-Bold-R-Normal--14-140-75-75-C-80-DEC-DECtech
TERMINAL_BOLD_ DECTECH18	-Bitstream-Terminal-Bold-R-Normal--18-180-75-75-C-110-DEC-DECtech
TERMINAL_BOLD_ DECTECH28	-DEC-Terminal-Bold-R-Normal--28-280-75-75-C-160-DEC-DECtech
TERMINAL_BOLD_ DECTECH36	-Bitstream-Terminal-Bold-R-Normal--36-360-75-75-C-220-DEC-DECtech
TERMINAL_BOLD_ NARROW14	-DEC-Terminal-Bold-R-Narrow--14-140-75-75-C-60-ISO8859-1
TERMINAL_BOLD_ NARROW18	-Bitstream-Terminal-Bold-R-Narrow--18-180-75-75-C-70-ISO8859-1
TERMINAL_BOLD_ NARROW28	-DEC-Terminal-Bold-R-Narrow--28-280-75-75-C-120-ISO8859-1
TERMINAL_BOLD_ NARROW36	-Bitstream-Terminal-Bold-R-Narrow--36-360-75-75-C-140-ISO8859-1
TERMINAL_BOLD_ NARROW_DECTECH14	-DEC-Terminal-Bold-R-Narrow--14-140-75-75-C-60-DEC-DECtech

(continued on next page)

Table C–1 (Cont.) VMS DECwindows 75 dpi Fonts

File Name	Font Name
Terminal	
TERMINAL_BOLD_ NARROW_DECTECH18	-Bitstream-Terminal-Bold-R-Narrow--18-180-75-75-C-70-DEC-DECtech
TERMINAL_BOLD_ NARROW_DECTECH28	-DEC-Terminal-Bold-R-Narrow--28-280-75-75-C-120-DEC-DECtech
TERMINAL_BOLD_ NARROW_DECTECH36	-Bitstream-Terminal-Bold-R-Narrow--36-360-75-75-C-140-DEC-DECtech
TERMINAL_BOLD_WIDE14	-DEC-Terminal-Bold-R-Wide--14-140-75-75-C-120-ISO8859-1
TERMINAL_BOLD_WIDE18	-Bitstream-Terminal-Bold-R-Narrow--18-180-75-75-C-140-ISO8859-1
TERMINAL_BOLD_WIDE_ DECTECH14	-DEC-Terminal-Bold-R-Wide--14-140-75-75-C-120-DEC-DECtech
TERMINAL_BOLD_WIDE_ DECTECH18	-Bitstream-Terminal-Bold-R-Narrow--18-180-75-75-C-140-DEC-DECtech
TERMINAL_DBLWIDE14	-DEC-Terminal-Medium-R-Double Wide--14-140-75-75-C-160-ISO8859-1
TERMINAL_DBLWIDE18	-Bitstream-Terminal-Medium-R-Double Wide--18-180-75-75-C-220-ISO8859-1
TERMINAL_DBLWIDE_ DECTECH14	-DEC-Terminal-Medium-R-Double Wide--14-140-75-75-C-160-DEC-DECtech
TERMINAL_DBLWIDE_ DECTECH18	-Bitstream-Terminal-Medium-R-Double Wide--18-180-75-75-C-220-DEC-DECtech
TERMINAL_DECTECH14	-DEC-Terminal-Medium-R-Normal--14-140-75-75-C-80-DEC-DECtech
TERMINAL_DECTECH18	-Bitstream-Terminal-Medium-R-Normal--18-180-75-75-C-110-DEC-DECtech
TERMINAL_DECTECH28	-DEC-Terminal-Medium-R-Normal--28-280-75-75-C-160-DEC-DECtech
TERMINAL_DECTECH36	-Bitstream-Terminal-Medium-R-Normal--36-360-75-75-C-220-DEC-DECtech
TERMINAL_GS14	-DEC-Terminal-Medium-R-Normal-GS-14-140-75-75-C-80-ISO8859-1
TERMINAL_GS18	-Bitstream-Terminal-Medium-R-Normal-GS-18-180-75-75-C-110-ISO8859-1
TERMINAL_NARROW14	-DEC-Terminal-Medium-R-Narrow--14-140-75-75-C-60-ISO8859-1
TERMINAL_NARROW18	-Bitstream-Terminal-Medium-R-Narrow--18-180-75-75-C-70-ISO8859-1
TERMINAL_NARROW28	-DEC-Terminal-Medium-R-Narrow--28-280-75-75-C-120-ISO8859-1
TERMINAL_NARROW36	-Bitstream-Terminal-Medium-R-Narrow--36-360-75-75-C-140-ISO8859-1
TERMINAL_NARROW_ DECTECH14	-DEC-Terminal-Medium-R-Narrow--14-140-75-75-C-60-DEC-DECtech
TERMINAL_NARROW_ DECTECH18	-Bitstream-Terminal-Medium-R-Narrow--18-180-75-75-C-70-DEC-DECtech
TERMINAL_NARROW_ DECTECH28	-DEC-Terminal-Medium-R-Narrow--28-280-75-75-C-120-DEC-DECtech
TERMINAL_NARROW_ DECTECH36	-Bitstream-Terminal-Medium-R-Narrow--36-360-75-75-C-140-DEC-DECtech
TERMINAL_WIDE14	-DEC-Terminal-Medium-R-Wide--14-140-75-75-C-120-ISO8859-1
TERMINAL_WIDE18	-Bitstream-Terminal-Medium-R-Wide--18-180-75-75-C-140-ISO8859-1
TERMINAL_WIDE_ DECTECH14	-DEC-Terminal-Medium-R-Wide--14-140-75-75-C-120-DEC-DECtech
TERMINAL_WIDE_ DECTECH18	-Bitstream-Terminal-Medium-R-Wide--18-180-75-75-C-140-DEC-DECtech

(continued on next page)

Table C–1 (Cont.) VMS DECwindows 75 dpi Fonts

File Name	Font Name
Times	
TIMES_BOLD8	-Adobe-Times-Bold-R-Normal--8-80-75-75-P-47-ISO8859-1
TIMES_BOLD10	-Adobe-Times-Bold-R-Normal--10-100-75-75-P-57-ISO8859-1
TIMES_BOLD12	-Adobe-Times-Bold-R-Normal--12-120-75-75-P-67-ISO8859-1
TIMES_BOLD14	-Adobe-Times-Bold-R-Normal--14-140-75-75-P-77-ISO8859-1
TIMES_BOLD18	-Adobe-Times-Bold-R-Normal--18-180-75-75-P-99-ISO8859-1
TIMES_BOLD24	-Adobe-Times-Bold-R-Normal--24-240-75-75-P-132-ISO8859-1
TIMES_BOLDITALIC8	-Adobe-Times-Bold-I-Normal--8-80-75-75-P-47-ISO8859-1
TIMES_BOLDITALIC10	-Adobe-Times-Bold-I-Normal--10-100-75-75-P-57-ISO8859-1
TIMES_BOLDITALIC12	-Adobe-Times-Bold-I-Normal--12-120-75-75-P-68-ISO8859-1
TIMES_BOLDITALIC14	-Adobe-Times-Bold-I-Normal--14-140-75-75-P-77-ISO8859-1
TIMES_BOLDITALIC18	-Adobe-Times-Bold-I-Normal--18-180-75-75-P-98-ISO8859-1
TIMES_BOLDITALIC24	-Adobe-Times-Bold-I-Normal--24-240-75-75-P-128-ISO8859-1
TIMES_ITALIC8	-Adobe-Times-Medium-I-Normal--8-80-75-75-P-42-ISO8859-1
TIMES_ITALIC10	-Adobe-Times-Medium-I-Normal--10-100-75-75-P-52-ISO8859-1
TIMES_ITALIC12	-Adobe-Times-Medium-I-Normal--12-120-75-75-P-63-ISO8859-1
TIMES_ITALIC14	-Adobe-Times-Medium-I-Normal--14-140-75-75-P-73-ISO8859-1
TIMES_ITALIC18	-Adobe-Times-Medium-I-Normal--18-180-75-75-P-94-ISO8859-1
TIMES_ITALIC24	-Adobe-Times-Medium-I-Normal--24-240-75-75-P-125-ISO8859-1
TIMES_ROMAN8	-Adobe-Times-Medium-R-Normal--8-80-75-75-P-44-ISO8859-1
TIMES_ROMAN10	-Adobe-Times-Medium-R-Normal--10-100-75-75-P-54-ISO8859-1
TIMES_ROMAN12	-Adobe-Times-Medium-R-Normal--12-120-75-75-P-64-ISO8859-1
TIMES_ROMAN14	-Adobe-Times-Medium-R-Normal--14-140-75-75-P-74-ISO8859-1
TIMES_ROMAN18	-Adobe-Times-Medium-R-Normal--18-180-75-75-P-94-ISO8859-1
TIMES_ROMAN24	-Adobe-Times-Medium-R-Normal--24-240-75-75-P-124-ISO8859-1

Table C–2 VMS DECwindows 100 dpi Fonts

File Name	Font Name
FIXED_100DPI	fixed
DECW$SESSION_100DPI	DECW$SESSION
VARIABLE_100DPI	variable

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
Avant Garde	
AVANTGARDE_BOOK8_ 100DPI	-Adobe-ITC Avant Garde Gothic-Book-R-Normal--11-80-100-100-P-59-ISO8859-1
AVANTGARDE_BOOK10_ 100DPI	-Adobe-ITC Avant Garde Gothic-Book-R-Normal--14-100-100-100-P-80-ISO8859-1
AVANTGARDE_BOOK12_ 100DPI	-Adobe-ITC Avant Garde Gothic-Book-R-Normal--17-120-100-100-P-93-ISO8859-1
AVANTGARDE_BOOK14_ 100DPI	-Adobe-ITC Avant Garde Gothic-Book-R-Normal--20-140-100-100-P-104-ISO8859-1
AVANTGARDE_BOOK18_ 100DPI	-Adobe-ITC Avant Garde Gothic-Book-R-Normal--25-180-100-100-P-138-ISO8859-1
AVANTGARDE_BOOK24_ 100DPI	-Adobe-ITC Avant Garde Gothic-Book-R-Normal--34-240-100-100-P-183-ISO8859-1
AVANTGARDE_ BOOKOBLIQUE8_100DPI	-Adobe-ITC Avant Garde Gothic-Book-O-Normal--10-80-100-100-P-59-ISO8859-1
AVANTGARDE_ BOOKOBLIQUE10_100DPI	-Adobe-ITC Avant Garde Gothic-Book-O-Normal--14-100-100-100-P-81-ISO8859-1
AVANTGARDE_ BOOKOBLIQUE12_100DPI	-Adobe-ITC Avant Garde Gothic-Book-O-Normal--17-120-100-100-P-92-ISO8859-1
AVANTGARDE_ BOOKOBLIQUE14_100DPI	-Adobe-ITC Avant Garde Gothic-Book-O-Normal--20-140-100-100-P-103-ISO8859-1
AVANTGARDE_ BOOKOBLIQUE18_100DPI	-Adobe-ITC Avant Garde Gothic-Book-O-Normal--25-180-100-100-P-138-ISO8859-1
AVANTGARDE_ BOOKOBLIQUE24_100DPI	-Adobe-ITC Avant Garde Gothic-Book-O-Normal--34-240-100-100-P-184-ISO8859-1
AVANTGARDE_DEMI8_ 100DPI	-Adobe-ITC Avant Garde Gothic-Demi-R-Normal--11-80-100-100-P-61-ISO8859-1
AVANTGARDE_DEMI10_ 100DPI	-Adobe-ITC Avant Garde Gothic-Demi-R-Normal--14-100-100-100-P-82-ISO8859-1
AVANTGARDE_DEMI12_ 100DPI	-Adobe-ITC Avant Garde Gothic-Demi-R-Normal--17-120-100-100-P-93-ISO8859-1
AVANTGARDE_DEMI14_ 100DPI	-Adobe-ITC Avant Garde Gothic-Demi-R-Normal--20-140-100-100-P-105-ISO8859-1
AVANTGARDE_DEMI18_ 100DPI	-Adobe-ITC Avant Garde Gothic-Demi-R-Normal--25-180-100-100-P-140-ISO8859-1
AVANTGARDE_DEMI24_ 100DPI	-Adobe-ITC Avant Garde Gothic-Demi-R-Normal--34-240-100-100-P-182-ISO8859-1
AVANTGARDE_ DEMIOBLIQUE8_100DPI	-Adobe-ITC Avant Garde Gothic-Demi-O-Normal--11-80-100-100-P-61-ISO8859-1
AVANTGARDE_ DEMIOBLIQUE10_100DPI	-Adobe-ITC Avant Garde Gothic-Demi-O-Normal--14-100-100-100-P-82-ISO8859-1
AVANTGARDE_ DEMIOBLIQUE12_100DPI	-Adobe-ITC Avant Garde Gothic-Demi-O-Normal--17-120-100-100-P-93-ISO8859-1
AVANTGARDE_ DEMIOBLIQUE14_100DPI	-Adobe-ITC Avant Garde Gothic-Demi-O-Normal--20-140-100-100-P-103-ISO8859-1
AVANTGARDE_ DEMIOBLIQUE18_100DPI	-Adobe-ITC Avant Garde Gothic-Demi-O-Normal--25-180-100-100-P-139-ISO8859-1

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
Avant Garde	
AVANTGARDE_ DEMIOBLIQUE24_100DPI	-Adobe-ITC Avant Garde Gothic-Demi-O-Normal--34-240-100-100-P-183-ISO8859-1
Courier	
COURIER8_100DPI	-Adobe-Courier-Medium-R-Normal--11-80-100-100-M-60-ISO8859-1
COURIER10_100DPI	-Adobe-Courier-Medium-R-Normal--14-100-100-100-M-90-ISO8859-1
COURIER12_100DPI	-Adobe-Courier-Medium-R-Normal--17-120-100-100-M-100-ISO8859-1
COURIER14_100DPI	-Adobe-Courier-Medium-R-Normal--20-140-100-100-M-110-ISO8859-1
COURIER18_100DPI	-Adobe-Courier-Medium-R-Normal--25-180-100-100-M-150-ISO8859-1
COURIER24_100DPI	-Adobe-Courier-Medium-R-Normal--34-240-100-100-M-200-ISO8859-1
COURIER_BOLD8_100DPI	-Adobe-Courier-Bold-R-Normal--11-80-100-100-M-60-ISO8859-1
COURIER_BOLD10_100DPI	-Adobe-Courier-Bold-R-Normal--14-100-100-100-M-90-ISO8859-1
COURIER_BOLD12_100DPI	-Adobe-Courier-Bold-R-Normal--17-120-100-100-M-100-ISO8859-1
COURIER_BOLD14_100DPI	-Adobe-Courier-Bold-R-Normal--20-140-100-100-M-110-ISO8859-1
COURIER_BOLD18_100DPI	-Adobe-Courier-Bold-R-Normal--25-180-100-100-M-150-ISO8859-1
COURIER_BOLD24_100DPI	-Adobe-Courier-Bold-R-Normal--34-240-100-100-M-200-ISO8859-1
COURIER_BOLDOBLIQUE8_ 100DPI	-Adobe-Courier-Bold-O-Normal--11-80-100-100-M-60-ISO8859-1
COURIER_ BOLDOBLIQUE10_100DPI	-Adobe-Courier-Bold-O-Normal--14-100-100-100-M-90-ISO8859-1
COURIER_ BOLDOBLIQUE12_100DPI	-Adobe-Courier-Bold-O-Normal--17-120-100-100-M-100-ISO8859-1
COURIER_ BOLDOBLIQUE14_100DPI	-Adobe-Courier-Bold-O-Normal--20-140-100-100-M-110-ISO8859-1
COURIER_ BOLDOBLIQUE18_100DPI	-Adobe-Courier-Bold-O-Normal--25-180-100-100-M-150-ISO8859-1
COURIER_ BOLDOBLIQUE24_100DPI	-Adobe-Courier-Bold-O-Normal--34-240-100-100-M-200-ISO8859-1
COURIER_OBLIQUE8_ 100DPI	-Adobe-Courier-Medium-O-Normal--11-80-100-100-M-60-ISO8859-1

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
COURIER_OBLIQUE10_100DPI	-Adobe-Courier-Medium-O-Normal--14-100-100-100-M-90-ISO8859-1
COURIER_OBLIQUE12_100DPI	-Adobe-Courier-Medium-O-Normal--17-120-100-100-M-100-ISO8859-1
COURIER_OBLIQUE14_100DPI	-Adobe-Courier-Medium-O-Normal--20-140-100-100-M-110-ISO8859-1
COURIER_OBLIQUE18_100DPI	-Adobe-Courier-Medium-O-Normal--25-180-100-100-M-150-ISO8859-1
COURIER_OBLIQUE24_100DPI	-Adobe-Courier-Medium-O-Normal--34-240-100-100-M-200-ISO8859-1
DEC Math	
DUTCH801_DECMATH_EXTENSION8_100DPI	-Bitstream-Dutch 801-Medium-R-Normal--42-80-100-100-P-331-DEC-DECmath_Extension
DUTCH801_DECMATH_EXTENSION10_100DPI	-Bitstream-Dutch 801-Medium-R-Normal--52-100-100-100-P-409-DEC-DECmath_Extension
DUTCH801_DECMATH_EXTENSION12_100DPI	-Bitstream-Dutch 801-Medium-R-Normal--62-120-100-100-P-488-DEC-DECmath_Extension
DUTCH801_DECMATH_EXTENSION14_100DPI	-Bitstream-Dutch 801-Medium-R-Normal--71-140-100-100-P-559-DEC-DECmath_Extension
DUTCH801_DECMATH_ITALIC8_100DPI	-Bitstream-Dutch 801-Medium-I-Normal--11-80-100-100-P-61-DEC-DECmath_Italic
DUTCH801_DECMATH_ITALIC10_100DPI	-Bitstream-Dutch 801-Medium-I-Normal--14-100-100-100-P-78-DEC-DECmath_Italic
DUTCH801_DECMATH_ITALIC12_100DPI	-Bitstream-Dutch 801-Medium-I-Normal--17-120-100-100-P-94-DEC-DECmath_Italic
DUTCH801_DECMATH_ITALIC14_100DPI	-Bitstream-Dutch 801-Medium-I-Normal--19-140-100-100-P-105-DEC-DECmath_Italic
DUTCH801_DECMATH_SYMBOL8_100DPI	-Bitstream-Dutch 801-Medium-R-Normal--11-80-100-100-P-85-DEC-DECmath_Symbol
DUTCH801_DECMATH_SYMBOL10_100DPI	-Bitstream-Dutch 801-Medium-R-Normal--14-100-100-100-P-107-DEC-DECmath_Symbol
DUTCH801_DECMATH_SYMBOL12_100DPI	-Bitstream-Dutch 801-Medium-R-Normal--17-120-100-100-P-130-DEC-DECmath_Symbol
DUTCH801_DECMATH_SYMBOL14_100DPI	-Bitstream-Dutch 801-Medium-R-Normal--19-140-100-100-P-146-DEC-DECmath_Symbol
Helvetica	
HELVETICA8_100DPI	-Adobe-Helvetica-Medium-R-Normal--11-80-100-100-P-56-ISO8859-1
HELVETICA10_100DPI	-Adobe-Helvetica-Medium-R-Normal--14-100-100-100-P-76-ISO8859-1
HELVETICA12_100DPI	-Adobe-Helvetica-Medium-R-Normal--17-120-100-100-P-88-ISO8859-1
HELVETICA14_100DPI	-Adobe-Helvetica-Medium-R-Normal--20-140-100-100-P-100-ISO8859-1
HELVETICA18_100DPI	-Adobe-Helvetica-Medium-R-Normal--25-180-100-100-P-130-ISO8859-1
HELVETICA24_100DPI	-Adobe-Helvetica-Medium-R-Normal--34-240-100-100-P-176-ISO8859-1
HELVETICA_BOLD8_100DPI	-Adobe-Helvetica-Bold-R-Normal--11-80-100-100-P-60-ISO8859-1

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
Helvetica	
HELVETICA_BOLD10_100DPI	-Adobe-Helvetica-Bold-R-Normal--14-100-100-100-P-82-ISO8859-1
HELVETICA_BOLD12_100DPI	-Adobe-Helvetica-Bold-R-Normal--17-120-100-100-P-92-ISO8859-1
HELVETICA_BOLD14_100DPI	-Adobe-Helvetica-Bold-R-Normal--20-140-100-100-P-105-ISO8859-1
HELVETICA_BOLD18_100DPI	-Adobe-Helvetica-Bold-R-Normal--25-180-100-100-P-138-ISO8859-1
HELVETICA_BOLD24_100DPI	-Adobe-Helvetica-Bold-R-Normal--34-240-100-100-P-182-ISO8859-1
HELVETICA_ BOLDOBLIQUE8_100DPI	-Adobe-Helvetica-Bold-O-Normal--11-80-100-100-P-60-ISO8859-1
HELVETICA_ BOLDOBLIQUE10_100DPI	-Adobe-Helvetica-Bold-O-Normal--14-100-100-100-P-82-ISO8859-1
HELVETICA_ BOLDOBLIQUE12_100DPI	-Adobe-Helvetica-Bold-O-Normal--17-120-100-100-P-92-ISO8859-1
HELVETICA_ BOLDOBLIQUE14_100DPI	-Adobe-Helvetica-Bold-O-Normal--20-140-100-100-P-103-ISO8859-1
HELVETICA_ BOLDOBLIQUE18_100DPI	-Adobe-Helvetica-Bold-O-Normal--25-180-100-100-P-138-ISO8859-1
HELVETICA_ BOLDOBLIQUE24_100DPI	-Adobe-Helvetica-Bold-O-Normal--34-240-100-100-P-182-ISO8859-1
HELVETICA_OBLIQUE8_ 100DPI	-Adobe-Helvetica-Medium-O-Normal--11-80-100-100-P-57-ISO8859-1
HELVETICA_OBLIQUE10_ 100DPI	-Adobe-Helvetica-Medium-O-Normal--14-100-100-100-P-78-ISO8859-1
HELVETICA_OBLIQUE12_ 100DPI	-Adobe-Helvetica-Medium-O-Normal--17-120-100-100-P-88-ISO8859-1
HELVETICA_OBLIQUE14_ 100DPI	-Adobe-Helvetica-Medium-O-Normal--20-140-100-100-P-98-ISO8859-1
HELVETICA_OBLIQUE18_ 100DPI	-Adobe-Helvetica-Medium-O-Normal--25-180-100-100-P-130-ISO8859-1
HELVETICA_OBLIQUE24_ 100DPI	-Adobe-Helvetica-Medium-O-Normal--34-240-100-100-P-176-ISO8859-1
Interim DEC Math	
INTERIM_DM_ EXTENSION14_100DPI	-Adobe-Interim DM-Medium-I-Normal--20-140-100-100-P-180-DEC-DECMATH_ EXTENSION
INTERIM_DM_ITALIC14_ 100DPI	-Adobe-Interim DM-Medium-I-Normal--20-140-100-100-P-180-DEC-DECMATH_ ITALIC
INTERIM_DM_SYMBOL14_ 100DPI	-Adobe-Interim DM-Medium-I-Normal--20-140-100-100-P-180-DEC-DECMATH_ SYMBOL

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
Lubalin Graph	
LUBALINGRAPH_BOOK8_ 100DPI	-Adobe-ITC Lubalin Graph-Book-R-Normal--11-80-100-100-P-60-ISO8859-1
LUBALINGRAPH_BOOK10_ 100DPI	-Adobe-ITC Lubalin Graph-Book-R-Normal--14-100-100-100-P-81-ISO8859-1
LUBALINGRAPH_BOOK12_ 100DPI	-Adobe-ITC Lubalin Graph-Book-R-Normal--17-120-100-100-P-89-ISO8859-1
LUBALINGRAPH_BOOK14_ 100DPI	-Adobe-ITC Lubalin Graph-Book-R-Normal--19-140-100-100-P-106-ISO8859-1
LUBALINGRAPH_BOOK18_ 100DPI	-Adobe-ITC Lubalin Graph-Book-R-Normal--24-180-100-100-P-139-ISO8859-1
LUBALINGRAPH_BOOK24_ 100DPI	-Adobe-ITC Lubalin Graph-Book-R-Normal--33-240-100-100-P-180-ISO8859-1
LUBALINGRAPH_ BOOKOBLIQUE8_100DPI	-Adobe-ITC Lubalin Graph-Book-O-Normal--11-80-100-100-P-60-ISO8859-1
LUBALINGRAPH_ BOOKOBLIQUE10_100DPI	-Adobe-ITC Lubalin Graph-Book-O-Normal--14-100-100-100-P-82-ISO8859-1
LUBALINGRAPH_ BOOKOBLIQUE12_100DPI	-Adobe-ITC Lubalin Graph-Book-O-Normal--19-120-100-100-P-89-ISO8859-1
LUBALINGRAPH_ BOOKOBLIQUE14_100DPI	-Adobe-ITC Lubalin Graph-Book-O-Normal--20-140-100-100-P-105-ISO8859-1
LUBALINGRAPH_ BOOKOBLIQUE18_100DPI	-Adobe-ITC Lubalin Graph-Book-O-Normal--24-180-100-100-P-140-ISO8859-1
LUBALINGRAPH_ BOOKOBLIQUE24_100DPI	-Adobe-ITC Lubalin Graph-Book-O-Normal--33-240-100-100-P-181-ISO8859-1
LUBALINGRAPH_DEMI8_ 100DPI	-Adobe-ITC Lubalin Graph-Demi-R-Normal--11-80-100-100-P-61-ISO8859-1
LUBALINGRAPH_DEMI10_ 100DPI	-Adobe-ITC Lubalin Graph-Demi-R-Normal--14-100-100-100-P-85-ISO8859-1
LUBALINGRAPH_DEMI12_ 100DPI	-Adobe-ITC Lubalin Graph-Demi-R-Normal--17-120-100-100-P-92-ISO8859-1
LUBALINGRAPH_DEMI14_ 100DPI	-Adobe-ITC Lubalin Graph-Demi-R-Normal--19-140-100-100-P-109-ISO8859-1

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
Lubalin Graph	
LUBALINGRAPH_DEMI18_100DPI	-Adobe-ITC Lubalin Graph-Demi-R-Normal--24-180-100-100-P-144-ISO8859-1
LUBALINGRAPH_DEMI24_100DPI	-Adobe-ITC Lubalin Graph-Demi-R-Normal--33-240-100-100-P-184-ISO8859-1
LUBALINGRAPH_DEMIOBLIQUE8_100DPI	-Adobe-ITC Lubalin Graph-Demi-O-Normal--11-80-100-100-P-62-ISO8859-1
LUBALINGRAPH_DEMIOBLIQUE10_100DPI	-Adobe-ITC Lubalin Graph-Demi-O-Normal--14-100-100-100-P-85-ISO8859-1
LUBALINGRAPH_DEMIOBLIQUE12_100DPI	-Adobe-ITC Lubalin Graph-Demi-O-Normal--17-120-100-100-P-92-ISO8859-1
LUBALINGRAPH_DEMIOBLIQUE14_100DPI	-Adobe-ITC Lubalin Graph-Demi-O-Normal--19-140-100-100-P-109-ISO8859-1
LUBALINGRAPH_DEMIOBLIQUE18_100DPI	-Adobe-ITC Lubalin Graph-Demi-O-Normal--24-180-100-100-P-144-ISO8859-1
LUBALINGRAPH_DEMIOBLIQUE24_100DPI	-Adobe-ITC Lubalin Graph-Demi-O-Normal--33-240-100-100-P-184-ISO8859-1
Menu	
MENU10_100DPI	-Bigelow & Holmes-Menu-Medium-R-Normal--13-100-100-100-P-77-ISO8859-1
MENU12_100DPI	-Bigelow & Holmes-Menu-Medium-R-Normal--16-120-100-100-P-92-ISO8859-1
New Century Schoolbook	
NEWCENTURYSCHLBK_BOLD8_100DPI	-Adobe-New Century Schoolbook-Bold-R-Normal--11-80-100-100-P-66-ISO8859-1
NEWCENTURYSCHLBK_BOLD10_100DPI	-Adobe-New Century Schoolbook-Bold-R-Normal--14-100-100-100-P-87-ISO8859-1
NEWCENTURYSCHLBK_BOLD12_100DPI	-Adobe-New Century Schoolbook-Bold-R-Normal--17-120-100-100-P-99-ISO8859-1
NEWCENTURYSCHLBK_BOLD14_100DPI	-Adobe-New Century Schoolbook-Bold-R-Normal--20-140-100-100-P-113-ISO8859-1
NEWCENTURYSCHLBK_BOLD18_100DPI	-Adobe-New Century Schoolbook-Bold-R-Normal--25-180-100-100-P-149-ISO8859-1
NEWCENTURYSCHLBK_BOLD24_100DPI	-Adobe-New Century Schoolbook-Bold-R-Normal--34-240-100-100-P-193-ISO8859-1
NEWCENTURYSCHLBK_BOLDITALIC8_100DPI	-Adobe-New Century Schoolbook-Bold-I-Normal--11-80-100-100-P-66-ISO8859-1
NEWCENTURYSCHLBK_BOLDITALIC10_100DPI	-Adobe-New Century Schoolbook-Bold-I-Normal--14-100-100-100-P-88-ISO8859-1
NEWCENTURYSCHLBK_BOLDITALIC12_100DPI	-Adobe-New Century Schoolbook-Bold-I-Normal--17-120-100-100-P-99-ISO8859-1
NEWCENTURYSCHLBK_BOLDITALIC14_100DPI	-Adobe-New Century Schoolbook-Bold-I-Normal--20-140-100-100-P-111-ISO8859-1
NEWCENTURYSCHLBK_BOLDITALIC18_100DPI	-Adobe-New Century Schoolbook-Bold-I-Normal--25-180-100-100-P-148-ISO8859-1

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
New Century Schoolbook	
NEWCENTURYSCHLBK_BOLDITALIC24_100DPI	-Adobe-New Century Schoolbook-Bold-I-Normal--34-240-100-100-P-193-ISO8859-1
NEWCENTURYSCHLBK_ITALIC8_100DPI	-Adobe-New Century Schoolbook-Medium-I-Normal--11-80-100-100-P-60-ISO8859-1
NEWCENTURYSCHLBK_ITALIC10_100DPI	-Adobe-New Century Schoolbook-Medium-I-Normal--14-100-100-100-P-81-ISO8859-1
NEWCENTURYSCHLBK_ITALIC12_100DPI	-Adobe-New Century Schoolbook-Medium-I-Normal--17-120-100-100-P-92-ISO8859-1
NEWCENTURYSCHLBK_ITALIC14_100DPI	-Adobe-New Century Schoolbook-Medium-I-Normal--20-140-100-100-P-104-ISO8859-1
NEWCENTURYSCHLBK_ITALIC18_100DPI	-Adobe-New Century Schoolbook-Medium-I-Normal--25-180-100-100-P-136-ISO8859-1
NEWCENTURYSCHLBK_ITALIC24_100DPI	-Adobe-New Century Schoolbook-Medium-I-Normal--34-240-100-100-P-182-ISO8859-1
NEWCENTURYSCHLBK_ROMAN8_100DPI	-Adobe-New Century Schoolbook-Medium-R-Normal--11-80-100-100-P-60-ISO8859-1
NEWCENTURYSCHLBK_ROMAN10_100DPI	-Adobe-New Century Schoolbook-Medium-R-Normal--14-100-100-100-P-82-ISO8859-1
NEWCENTURYSCHLBK_ROMAN12_100DPI	-Adobe-New Century Schoolbook-Medium-R-Normal--17-120-100-100-P-91-ISO8859-1
NEWCENTURYSCHLBK_ROMAN14_100DPI	-Adobe-New Century Schoolbook-Medium-R-Normal--20-140-100-100-P-103-ISO8859-1
NEWCENTURYSCHLBK_ROMAN18_100DPI	-Adobe-New Century Schoolbook-Medium-R-Normal--25-180-100-100-P-136-ISO8859-1
NEWCENTURYSCHLBK_ROMAN24_100DPI	-Adobe-New Century Schoolbook-Medium-R-Normal--34-240-100-100-P-181-ISO8859-1
Souvenir	
SOUVENIR_DEMI8_100DPI	-Adobe-ITC Souvenir-Demi-R-Normal--11-80-100-100-P-62-ISO8859-1
SOUVENIR_DEMI10_100DPI	-Adobe-ITC Souvenir-Demi-R-Normal--14-100-100-100-P-90-ISO8859-1
SOUVENIR_DEMI12_100DPI	-Adobe-ITC Souvenir-Demi-R-Normal--17-120-100-100-P-94-ISO8859-1
SOUVENIR_DEMI14_100DPI	-Adobe-ITC Souvenir-Demi-R-Normal--20-140-100-100-P-112-ISO8859-1
SOUVENIR_DEMI18_100DPI	-Adobe-ITC Souvenir-Demi-R-Normal--25-180-100-100-P-149-ISO8859-1
SOUVENIR_DEMI24_100DPI	-Adobe-ITC Souvenir-Demi-R-Normal--34-240-100-100-P-191-ISO8859-1
SOUVENIR_DEMIITALIC8_100DPI	-Adobe-ITC Souvenir-Demi-I-Normal--11-80-100-100-P-67-ISO8859-1

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
Souvenir	
SOUVENIR_DEMIITALIC10_100DPI	-Adobe-ITC Souvenir-Demi-I-Normal--14-100-100-100-P-92-ISO8859-1
SOUVENIR_DEMIITALIC12_100DPI	-Adobe-ITC Souvenir-Demi-I-Normal--17-120-100-100-P-98-ISO8859-1
SOUVENIR_DEMIITALIC14_100DPI	-Adobe-ITC Souvenir-Demi-I-Normal--20-140-100-100-P-115-ISO8859-1
SOUVENIR_DEMIITALIC18_100DPI	-Adobe-ITC Souvenir-Demi-I-Normal--25-180-100-100-P-154-ISO8859-1
SOUVENIR_DEMIITALIC24_100DPI	-Adobe-ITC Souvenir-Demi-I-Normal--34-240-100-100-P-197-ISO8859-1
SOUVENIR_LIGHT8_100DPI	-Adobe-ITC Souvenir-Light-R-Normal--11-80-100-100-P-56-ISO8859-1
SOUVENIR_LIGHT10_100DPI	-Adobe-ITC Souvenir-Light-R-Normal--14-100-100-100-P-79-ISO8859-1
SOUVENIR_LIGHT12_100DPI	-Adobe-ITC Souvenir-Light-R-Normal--17-120-100-100-P-85-ISO8859-1
SOUVENIR_LIGHT14_100DPI	-Adobe-ITC Souvenir-Light-R-Normal--20-140-100-100-P-102-ISO8859-1
SOUVENIR_LIGHT18_100DPI	-Adobe-ITC Souvenir-Light-R-Normal--25-180-100-100-P-135-ISO8859-1
SOUVENIR_LIGHT24_100DPI	-Adobe-ITC Souvenir-Light-R-Normal--34-240-100-100-P-174-ISO8859-1
SOUVENIR_LIGHTITALIC8_100DPI	-Adobe-ITC Souvenir-Light-I-Normal--11-80-100-100-P-59-ISO8859-1
SOUVENIR_LIGHTITALIC10_100DPI	-Adobe-ITC Souvenir-Light-I-Normal--14-100-100-100-P-82-ISO8859-1
SOUVENIR_LIGHTITALIC12_100DPI	-Adobe-ITC Souvenir-Light-I-Normal--17-120-100-100-P-88-ISO8859-1
SOUVENIR_LIGHTITALIC14_100DPI	-Adobe-ITC Souvenir-Light-I-Normal--20-140-100-100-P-104-ISO8859-1
SOUVENIR_LIGHTITALIC18_100DPI	-Adobe-ITC Souvenir-Light-I-Normal--25-180-100-100-P-139-ISO8859-1
SOUVENIR_LIGHTITALIC24_100DPI	-Adobe-ITC Souvenir-Light-I-Normal--34-240-100-100-P-177-ISO8859-1
Symbol	
SYMBOL8_100DPI	-Adobe-Symbol-Medium-R-Normal--11-80-100-100-P-61-ADOBE-FONTSPECIFIC
SYMBOL10_100DPI	-Adobe-Symbol-Medium-R-Normal--14-100-100-100-P-85-ADOBE-FONTSPECIFIC
SYMBOL12_100DPI	-Adobe-Symbol-Medium-R-Normal--17-120-100-100-P-95-ADOBE-FONTSPECIFIC
SYMBOL14_100DPI	-Adobe-Symbol-Medium-R-Normal--20-140-100-100-P-107-ADOBE-FONTSPECIFIC
SYMBOL18_100DPI	-Adobe-Symbol-Medium-R-Normal--25-180-100-100-P-142-ADOBE-FONTSPECIFIC
SYMBOL24_100DPI	-Adobe-Symbol-Medium-R-Normal--34-240-100-100-P-191-ADOBE-FONTSPECIFIC

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
Terminal	
TERMINAL10_100DPI	-DEC-Terminal-Medium-R-Normal--14-100-100-100-C-80-ISO8859-1
TERMINAL14_100DPI	-Bitstream-Terminal-Medium-R-Normal--18-140-100-100-C-110-ISO8859-1
TERMINAL18_100DPI	-Bitstream-Terminal-Medium-R-Normal--25-180-100-100-C-150-ISO8859-1
TERMINAL20_100DPI	-DEC-Terminal-Medium-R-Normal--28-200-100-100-C-160-ISO8859-1
TERMINAL28_100DPI	-Bitstream-Terminal-Medium-R-Normal--36-280-100-100-C-220-ISO8859-1
TERMINAL36_100DPI	-Bitstream-Terminal-Medium-R-Normal--50-360-100-100-C-300-ISO8859-1
TERMINAL_BOLD10_100DPI	-DEC-Terminal-Bold-R-Normal--14-100-100-100-C-80-ISO8859-1
TERMINAL_BOLD14_100DPI	-Bitstream-Terminal-Bold-R-Normal--18-140-100-100-C-110-ISO8859-1
TERMINAL_BOLD18_100DPI	-Bitstream-Terminal-Bold-R-Normal--25-180-100-100-C-150-ISO8859-1
TERMINAL_BOLD20_100DPI	-DEC-Terminal-Bold-R-Normal--28-200-100-100-C-160-ISO8859-1
TERMINAL_BOLD28_100DPI	-Bitstream-Terminal-Bold-R-Normal--36-280-100-100-C-220-ISO8859-1
TERMINAL_BOLD36_100DPI	-Bitstream-Terminal-Bold-R-Normal--50-360-100-100-C-300-ISO8859-1
TERMINAL_BOLD_ DBLWIDE10_100DPI	-DEC-Terminal-Bold-R-Double Wide--14-100-100-100-C-160-ISO8859-1
TERMINAL_BOLD_ DBLWIDE14_100DPI	-Bitstream-Terminal-Bold-R-Double Wide--18-140-100-100-C-220-ISO8859-1
TERMINAL_BOLD_ DBLWIDE18_100DPI	-Bitstream-Terminal-Bold-R-Double Wide--25-180-100-100-C-300-ISO8859-1
TERMINAL_BOLD_ DBLWIDE_DECTECH10_ 100DPI	-DEC-Terminal-Bold-R-Double Wide--14-100-100-100-C-160-DEC-DECtech
TERMINAL_BOLD_ DBLWIDE_DECTECH14_ 100DPI	-Bitstream-Terminal-Bold-R-Double Wide--18-140-100-100-C-220-DEC-DECtech
TERMINAL_BOLD_ DBLWIDE_DECTECH18_ 100DPI	-Bitstream-Terminal-Bold-R-Double Wide--25-180-100-100-C-300-DEC-DECtech
TERMINAL_BOLD_ DECTECH10_100DPI	-DEC-Terminal-Bold-R-Normal--14-100-100-100-C-80-DEC-DECtech
TERMINAL_BOLD_ DECTECH14_100DPI	-Bitstream-Terminal-Bold-R-Normal--18-140-100-100-C-110-DEC-DECtech
TERMINAL_BOLD_ DECTECH18_100DPI	-Bitstream-Terminal-Bold-R-Normal--25-180-100-100-C-150-DEC-DECtech
TERMINAL_BOLD_ DECTECH20_100DPI	-DEC-Terminal-Bold-R-Normal--28-200-100-100-C-160-DEC-DECtech
TERMINAL_BOLD_ DECTECH28_100DPI	-Bitstream-Terminal-Bold-R-Normal--36-280-100-100-C-220-DEC-DECtech
TERMINAL_BOLD_ DECTECH36_100DPI	-Bitstream-Terminal-Bold-R-Normal--50-360-100-100-C-300-DEC-DECtech

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
Terminal	
TERMINAL_BOLD_NARROW10_100DPI	-DEC-Terminal-Bold-R-Narrow--14-100-100-100-C-60-ISO8859-1
TERMINAL_BOLD_NARROW14_100DPI	-Bitstream-Terminal-Bold-R-Narrow--18-140-100-100-C-70-ISO8859-1
TERMINAL_BOLD_NARROW18_100DPI	-Bitstream-Terminal-Bold-R-Narrow--25-180-100-100-C-90-ISO8859-1
TERMINAL_BOLD_NARROW20_100DPI	-DEC-Terminal-Bold-R-Narrow--28-200-100-100-C-120-ISO8859-1
TERMINAL_BOLD_NARROW28_100DPI	-Bitstream-Terminal-Bold-R-Narrow--36-280-100-100-C-140-ISO8859-1
TERMINAL_BOLD_NARROW36_100DPI	-Bitstream-Terminal-Bold-R-Narrow--50-360-100-100-C-180-ISO8859-1
TERMINAL_BOLD_NARROW_DECTECH10_100DPI	-DEC-Terminal-Bold-R-Narrow--14-100-100-100-C-60-DEC-DECtech
TERMINAL_BOLD_NARROW_DECTECH14_100DPI	-Bitstream-Terminal-Bold-R-Narrow--18-140-100-100-C-70-DEC-DECtech
TERMINAL_BOLD_NARROW_DECTECH18_100DPI	-Bitstream-Terminal-Bold-R-Narrow--25-180-100-100-C-90-DEC-DECtech
TERMINAL_BOLD_NARROW_DECTECH20_100DPI	-DEC-Terminal-Bold-R-Narrow--28-200-100-100-C-120-DEC-DECtech
TERMINAL_BOLD_NARROW_DECTECH28_100DPI	-Bitstream-Terminal-Bold-R-Narrow--36-280-100-100-C-140-DEC-DECtech
TERMINAL_BOLD_NARROW_DECTECH36_100DPI	-Bitstream-Terminal-Bold-R-Narrow--50-360-100-100-C-180-DEC-DECtech
TERMINAL_BOLD_WIDE10_100DPI	-DEC-Terminal-Bold-R-Wide--14-100-100-100-C-120-ISO8859-1
TERMINAL_BOLD_WIDE14_100DPI	-Bitstream-Terminal-Bold-R-Wide--18-140-100-100-C-140-ISO8859-1
TERMINAL_BOLD_WIDE18_100DPI	-Bitstream-Terminal-Bold-R-Wide--25-180-100-100-C-180-ISO8859-1
TERMINAL_BOLD_WIDE_DECTECH10_100DPI	-DEC-Terminal-Bold-R-Wide--14-100-100-100-C-120-DEC-DECtech
TERMINAL_BOLD_WIDE_DECTECH14_100DPI	-Bitstream-Terminal-Bold-R-Wide--18-140-100-100-C-140-DEC-DECtech
TERMINAL_BOLD_WIDE_DECTECH18_100DPI	-Bitstream-Terminal-Bold-R-Wide--25-180-100-100- C-180-DEC-DECtech
TERMINAL_DBLWIDE10_100DPI	-DEC-Terminal-Medium-R-Double Wide--14-100-100-100-C-160-ISO8859-1
TERMINAL_DBLWIDE14_100DPI	-Bitstream-Terminal-Medium-R-Double Wide--18-140-100-100-C-220-ISO8859-1
TERMINAL_DBLWIDE18_100DPI	-Bitstream-Terminal-Medium-R-Double Wide--25-180-100-100-C-300-ISO8859-1

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
Terminal	
TERMINAL_DBLWIDE_ DECTECH10_100DPI	-DEC-Terminal-Medium-R-Double Wide--14-100-100-100-C-160-DEC-DECtech
TERMINAL_DBLWIDE_ DECTECH14_100DPI	-Bitstream-Terminal-Medium-R-Double Wide--18-140-100-100-C-220-DEC-DECtech
TERMINAL_DBLWIDE_ DECTECH18_100DPI	-Bitstream-Terminal-Medium-R-Double Wide--25-180-100-100-C-300-DEC-DECtech
TERMINAL_DECTECH10_ 100DPI	-DEC-Terminal-Medium-R-Normal--14-100-100-100-C-80-DEC-DECtech
TERMINAL_DECTECH14_ 100DPI	-Bitstream-Terminal-Medium-R-Normal--18-140-100-100-C-110-DEC-DECtech
TERMINAL_DECTECH18_ 100DPI	-Bitstream-Terminal-Medium-R-Normal--25-180-100-100-C-150-DEC-DECtech
TERMINAL_DECTECH20_ 100DPI	-DEC-Terminal-Medium-R-Normal--28-200-100-100-C-160-DEC-DECtech
TERMINAL_DECTECH28_ 100DPI	-Bitstream-Terminal-Medium-R-Normal--36-280-100-100-C-220-DEC-DECtech
TERMINAL_DECTECH36_ 100DPI	-Bitstream-Terminal-Medium-R-Normal--50-360-100-100-C-300-DEC-DECtech
TERMINAL_GS10_100DPI	-DEC-Terminal-Medium-R-Normal-GS-14-100-100-100-C-80-ISO8859-1
TERMINAL_GS14_100DPI	-Bitstream-Terminal-Medium-R-Normal-GS-18-140-100-100-C-110-ISO8859-1
TERMINAL_NARROW10_ 100DPI	-DEC-Terminal-Medium-R-Narrow--14-100-100-100-C-60-ISO8859-1
TERMINAL_NARROW14_ 100DPI	-Bitstream-Terminal-Medium-R-Narrow--18-140-100-100-C-70-ISO8859-1
TERMINAL_NARROW18_ 100DPI	-Bitstream-Terminal-Medium-R-Narrow--25-180-100-100- C-90-ISO8859-1
TERMINAL_NARROW20_ 100DPI	-DEC-Terminal-Medium-R-Narrow--28-200-100-100-C-120-ISO8859-1
TERMINAL_NARROW28_ 100DPI	-Bitstream-Terminal-Medium-R-Narrow--36-280-100-100-C-140-ISO8859-1
TERMINAL_NARROW36_ 100DPI	-Bitstream-Terminal-Medium-R-Narrow--50-360-100-100-C-180-ISO8859-1
TERMINAL_NARROW_ DECTECH10_100DPI	-DEC-Terminal-Medium-R-Narrow--14-100-100-100-C-60-DEC-DECtech
TERMINAL_NARROW_ DECTECH14_100DPI	-Bitstream-Terminal-Medium-R-Narrow--18-140-100-100-C-70-DEC-DECtech

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
Terminal	
TERMINAL_NARROW_ DECTECH18_100DPI	-Bitstream-Terminal-Medium-R-Narrow--25-180-100-100-C-90-DEC-DECtech
TERMINAL_NARROW_ DECTECH20_100DPI	-DEC-Terminal-Medium-R-Narrow--28-200-100-100-C-120-DEC-DECtech
TERMINAL_NARROW_ DECTECH28_100DPI	-Bitstream-Terminal-Medium-R-Narrow--36-280-100-100-C-140-DEC-DECtech
TERMINAL_NARROW_ DECTECH36_100DPI	-Bitstream-Terminal-Medium-R-Narrow--50-360-100-100-C-180-DEC-DECtech
TERMINAL_WIDE10_100DPI	-DEC-Terminal-Medium-R-Wide--14-100-100-100-C-120-ISO8859-1
TERMINAL_WIDE14_100DPI	-Bitstream-Terminal-Medium-R-Wide--18-140-100-100-C-140-ISO8859-1
TERMINAL_WIDE18_100DPI	-Bitstream-Terminal-Medium-R-Wide--25-180-100-100-C-180-ISO8859-1
TERMINAL_WIDE_ DECTECH10_100DPI	-DEC-Terminal-Medium-R-Wide--14-100-100-100-C-120-DEC-DECtech
TERMINAL_WIDE_ DECTECH14_100DPI	-Bitstream-Terminal-Medium-R-Wide--18-140-100-100-C-140-DEC-DECtech
TERMINAL_WIDE_ DECTECH18_100DPI	-Bitstream-Terminal-Medium-R-Wide--25-180-100-100-C-180-DEC-DECtech
Times	
TIMES_BOLD8_100DPI	-Adobe-Times-Bold-R-Normal- -11-80-100-100-P-57-ISO8859-1
TIMES_BOLD10_100DPI	-Adobe-Times-Bold-R-Normal--14-100-100-100-P-76-ISO8859-1
TIMES_BOLD12_100DPI	-Adobe-Times-Bold-R-Normal--17-120-100-100-P-88-ISO8859-1
TIMES_BOLD14_100DPI	-Adobe-Times-Bold-R-Normal--20-140-100-100-P-100-ISO8859-1
TIMES_BOLD18_100DPI	-Adobe-Times-Bold-R-Normal--25-180-100-100-P-132-ISO8859-1
TIMES_BOLD24_100DPI	-Adobe-Times-Bold-R-Normal--34-240-100-100-P-177-ISO8859-1
TIMES_BOLDITALIC8_ 100DPI	-Adobe-Times-Bold-I-Normal--11-80-100-100-P-57-ISO8859-1
TIMES_BOLDITALIC10_ 100DPI	-Adobe-Times-Bold-I-Normal--14-100-100-100-P-77-ISO8859-1
TIMES_BOLDITALIC12_ 100DPI	-Adobe-Times-Bold-I-Normal--17-120-100-100-P-86-ISO8859-1
TIMES_BOLDITALIC14_ 100DPI	-Adobe-Times-Bold-I-Normal--20-140-100-100-P-98-ISO8859-1
TIMES_BOLDITALIC18_ 100DPI	-Adobe-Times-Bold-I-Normal--25-180-100-100-P-128-ISO8859-1
TIMES_BOLDITALIC24_ 100DPI	-Adobe-Times-Bold-I-Normal--34-240-100-100-P-170-ISO8859-1
TIMES_ITALIC8_100DPI	-Adobe-Times-Medium-I-Normal--11-80-100-100-P-52-ISO8859-1
TIMES_ITALIC10_100DPI	-Adobe-Times-Medium-I-Normal--14-100-100-100-P-73-ISO8859-1
TIMES_ITALIC12_100DPI	-Adobe-Times-Medium-I-Normal--17-120-100-100-P-84-ISO8859-1
TIMES_ITALIC14_100DPI	-Adobe-Times-Medium-I-Normal--20-140-100-100-P-94-ISO8859-1
TIMES_ITALIC18_100DPI	-Adobe-Times-Medium-I-Normal--25-180-100-100-P-125-ISO8859-1

(continued on next page)

Table C–2 (Cont.) VMS DECwindows 100 dpi Fonts

File Name	Font Name
Times	
TIMES_ITALIC24_100DPI	-Adobe-Times-Medium-I-Normal--34-240-100-100-P-168-ISO8859-1
TIMES_ROMAN8_100DPI	-Adobe-Times-Medium-R-Normal--11-80-100-100-P-54-ISO8859-1
TIMES_ROMAN10_100DPI	-Adobe-Times-Medium-R-Normal--14-100-100-100-P-74-ISO8859-1
TIMES_ROMAN12_100DPI	-Adobe-Times-Medium-R-Normal--17-120-100-100-P-84-ISO8859-1
TIMES_ROMAN14_100DPI	-Adobe-Times-Medium-R-Normal--20-140-100-100-P-96-ISO8859-1
TIMES_ROMAN18_100DPI	-Adobe-Times-Medium-R-Normal--25-180-100-100-P-125-ISO8859-1
TIMES_ROMAN24_100DPI	-Adobe-Times-Medium-R-Normal--34-240-100-100-P-170-ISO8859-1

Table C–3 VMS DECwindows Common Fonts

CURSOR	Cursor
DECW$C32X32	DECW$CURSOR
DECW$CURSOR	DECW$CURSOR
Fixed Width	
File Name	**Font Name**
5X8	-Misc-Fixed-Medium-R-Normal--8-80-75-75-C-50-ISO8859-1
6X10	-Misc-Fixed-Medium-R-Normal--10-100-75-75-C-60-ISO8859-1
6X12	-Misc-Fixed-Medium-R-SemiCondensed--12-110-75-75-C-60-ISO8859-1
6X13	-Misc-Fixed-Medium-R-SemiCondensed--13-120-75-75-C-60-ISO8859-1
6X13B	-Misc-Fixed-Bold-R-SemiCondensed--13-120-75-75-C-60-ISO8859-1
6X9	-Misc-Fixed-Medium-R-Normal--9-90-75-75-C-60-ISO8859-1
7X13	-Misc-Fixed-Medium-R-Normal--13-120-75-75-C-70-ISO8859-1
7X13B	-Misc-Fixed-Bold-R-Normal--13-120-75-75-C-70-ISO8859-1
7X14	-Misc-Fixed-Medium-R-Normal--14-130-75-75-C-70-ISO8859-1
8X13	-Misc-Fixed-Medium-R-Normal--13-120-75-75-C-80-ISO8859-1
8X13B	-Misc-Fixed-Bold-R-Normal--13-120-75-75-C-80-ISO8859-1
8X16	-Sony-Fixed-Medium-R-Normal--16-120-100-100-C-80-ISO8859-1
9X15	-Misc-Fixed-Medium-R-Normal--15-140-75-75-C-90-ISO8859-1
9X15B	-Misc-Fixed-Bold-R-Normal--15-140-75-75-C-90-ISO8859-1
10X20	-Misc-Fixed-Medium-R-Normal--20-200-75-75-C-100-ISO8859-1
12X24	-Sony-Fixed-Medium-R-Normal--24-170-100-100-C-120-ISO8859-1

(continued on next page)

Table C–3 (Cont.) VMS DECwindows Common Fonts

Fixed Width	
File Name Alias	**Font Name Alias**
5X8	-Misc-Fixed-Medium-R-Normal--8-60-100-100-C-50-ISO8859-1
6X10	-Misc-Fixed-Medium-R-Normal--9-80-100-100-C-60-ISO8859-1
6X12	-Misc-Fixed-Medium-R-Normal--10-70-100-100-C-60-ISO8859-1
6X13	-Misc-Fixed-Medium-R-SemiCondensed--12-90-100-100-C-60-ISO8859-1
6X13B	-Misc-Fixed-Medium-R-SemiCondensed--13-100-100-100-C-60-ISO8859-1
6X9	-Misc-Fixed-Bold-R-SemiCondensed--13-100-100-100-C-60-ISO8859-1
7X13	-Misc-Fixed-Medium-R-Normal--13-100-100-100-C-70-ISO8859-1
7X13B	-Misc-Fixed-Bold-R-Normal--13-100-100-100-C-70-ISO8859-1
7X14	-Misc-Fixed-Medium-R-Normal--14-110-100-100-C-70-ISO8859-1
8X13	-Misc-Fixed-Medium-R-Normal--13-100-100-100-C-80-ISO8859-1
8X13B	-Misc-Fixed-Bold-R-Normal--13-100-100-100-C-80-ISO8859-1
8X16	-Sony-Fixed-Medium-R-Normal--16-150-75-75-C-80-ISO8859-1
9X15	-Misc-Fixed-Medium-R-Normal--15-120-100-100-C-90-ISO8859-1
9X15B	-Misc-Fixed-Bold-R-Normal--15-120-100-100-C-90-ISO8859-1
10X20	-Misc-Fixed-Medium-R-Normal--20-140-100-100-C-100-ISO8859-1
12X24	-Sony-Fixed-Medium-R-Normal--24-230-75-75-C-120-ISO8859-1

Index

A

B

C

D

E

F

G

H

I

K

Q

R

S

T

U

V

W

X

Z