HP Windows/9000 Programmer's Manual

HP 9000 Series 300 Computers

HP Part Number 97069-90002

HEWLETT PACKARD

Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Table of Contents

Overview 1

HP Windows/9000 supports subroutine libraries in the Window library, */usr/lib/libwindow.a.*

Window routines can be called from user programs to do tasks performed by window commands—for example, creating, moving, or changing the size of a window. In addition, library routines allow users to do tasks unachievable through the window commands—for example, adding scroll bars to a window's border. This manual discusses the use of Window routines in user programs.

Example Source Code

Throughout this manual, you will find several source code examples on how to use Window library routines. Most of these example programs and functions are stored in */usr/lib/hpwindows/man_examples*, hereafter referred to as the *man_examples* directory.

Also, the examples found throughout this manual use constant definitions defined in the header file */usr/include/window.h*, referred to as the *window.h* header file. Using these definitions (via the `#include` C-compiler directive) in your windowing programs will help ensure portability.

For details on compiling window programs, see the Appendix A, "Compiling Window Programs."

Conventions

The following typeface conventions are used throughout this manual:

- *Italic text* is used for the names of files and HP-UX commands, system calls, subroutines, etc. found in the *HP-UX Reference*. Italics is also used to denote window commands (e.g., *wsh(1)*) and Window library routines (e.g., *winit(3W)*).
- **Boldface text** is used when a word is first defined (as **term0**) and for strong emphasis (**never do this**).
- `Computer text` denotes literal text, either typed by the user or displayed by the system. For example,

  `wmstart` Return

  means to type "wmstart" at the keyboard, and press the Return key.
- Environment variables, such as WMDIR and WMIATIMEOUT, are represented in upper-case letters.

Prerequisites

Before reading this guide, you should know:

1. How to use the window system. You should feel comfortable with the window commands; in particular, you must understand window concepts presented in the *HP Windows/9000 User's Manual.*

2. The C programming language and its standard input/output routines. All program examples in this guide are presented in C, and all input/output is done in C. For more information on C, consult:

    - the C programming manual shipped with HP-UX documentation,
    - *HP-UX Concepts and Tutorials: Programming Environment.*

3. If you intend to use graphics windows, then become familiar with the Starbase Graphics library routines documented in *HP-UX Concepts and Tutorials: Starbase Graphics Techniques.*

    The *Starbase Device Drivers Library* may also be helpful; it contains information about the Graphics Window Device Driver.

Programming Manual Contents

The *HP Windows/9000 Programmer's Manual* consists of the following tabbed sections:

- Overview
- Concepts
- Window Library
- Appendices

Descriptions of each tabbed section follow.

Overview

This tabbed section contains **Chapter 1: Overview**, which describes the organization and contents of the *Programmer's Manual*, explains conventions used, and identifies information you should know before using the manual.

Concepts

This tabbed section contains **Chapter 2: Concepts**, which describes window system concepts and HP Windows/9000 system architecture and data flow. This chapter is useful if you wish to know the intrinsic structure of the window system. If you like to know how things work before you use them, then read this chapter.

Window Library

This tabbed section of the manual contains chapters three through ten, all dealing with Window library routines.

Chapter 3: Window Management defines the programming model used in programs that call window management routines. It discusses how to start and stop communication with the window manager, how to create term0 and graphics windows, how to destroy windows, how to shuffle windows and repaint the display screen, and how to kill the window manager.

Chapter 4: Window Manipulation defines how to start communication with a term0 or graphics window. Once this is done, a program can call other window manipulation routines to change window attributes such as size, location, and label.

Chapter 5: Icons shows how to use the icon manipulation routines. These routines display icons and change windows to icons (and vice versa). In addition, you can make your own custom icons by using these routines.

Chapter 6: Event Detection describes how to use event detection routines, which allow a program to be signaled (interrupted) when window system events (such as moving a window, selecting a window, or activating a hotspot) occur.

Chapter 7: Locator and Echo Routines describes the use of routines that read from and manipulate the locator devices and echo. For example, with these routines you can define your own echo to appear over a window's user area. You can also determine the locator's position via these routines.

Chapter 8: Arrows and Elevators shows how to create elevators in a graphics window's border. It also shows how to enable user mode, in which a graphics window's elevators and arrows send signals to programs that have enabled event detection for the window.

Chapter 9: Graphics Window Hotspots illustrates how to create hotspots in a graphics window's user area. A hotspot is a sensitive rectangle which can be activated either by the user pressing a button, or simply by the locator entering or exiting its boundary. When activated, a hotspot can signal an application.

Chapter 10: User-Definable Menus describes how to define and read from pop-up menus, which you can use from your applications.

Chapter 11: Graphics Window Input Routines shows how to read input from graphics windows. Graphics window input routines provide several different, powerful ways to read input from graphics windows. Each method has its own benefits, which you may find useful for your application development needs.

Chapter 12: Graphics Window Softkeys describes the use of routines that edit graphics window softkeys and turn them on and off.

Chapter 13: Term0 Windows describes term0 window features, term0 escape sequences, user-definable softkeys, term0 font management routines, term0 window colors, and using raw mode with term0 windows.

Appendices

The "Appendices" tab contains the appendices described below and a subject index.

Appendix A: Compiling Window Programs describes how to compile programs that call routines from the Window libraries.

Appendix B: HP Windows/9000 Files lists files associated with the window system and briefly describes their function.

Notes

Concepts 2

Before you begin using Window routines, you should understand basic window system concepts. This chapter provides an overview of essential window concepts and a description of HP Windows/9000 system architecture.

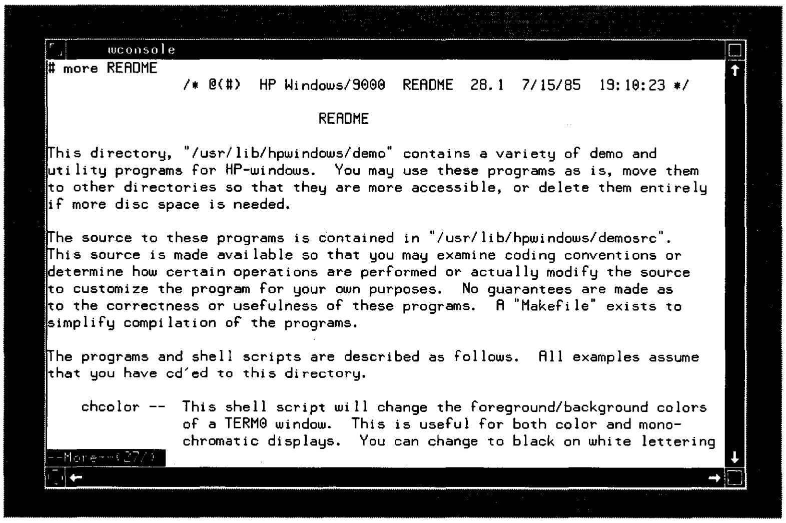
Window Types

To maximize the use of software developed for non-window systems on HP Widows/9000, two **window types** are supported: **term0** and **graphics**.

Term0 Window Type

Term0 windows (pronounced "term-zero") emulate HP 2622 terminals without block or format mode and also support HP 2627 color escape sequences. Therefore, programs written for HP 2622 terminals are easily ported to term0 windows. Chapter 10, "Term0 Windows," describes how to manage fonts in term0 windows, how to use term0 window escape sequences, and how to input information from term0 windows.

Figure 2-1 shows a typical term0 window. The user of this window executed the *more(1)* command to display the contents of the *README* file (found in the */usr/lib/hpwindows/demo* directory).



```
# more README
              /* @(#)  HP Windows/9000  README  28.1  7/15/85  19:10:23 */

                              README

This directory, "/usr/lib/hpwindows/demo" contains a variety of demo and
utility programs for HP-windows.  You may use these programs as is, move them
to other directories so that they are more accessible, or delete them entirely
if more disc space is needed.

The source to these programs is contained in "/usr/lib/hpwindows/demosrc".
This source is made available so that you may examine coding conventions or
determine how certain operations are performed or actually modify the source
to customize the program for your own purposes.  No guarantees are made as
to the correctness or usefulness of these programs.  A "Makefile" exists to
simplify compilation of the programs.

The programs and shell scripts are described as follows.  All examples assume
that you have cd'ed to this directory.

    chcolor --  This shell script will change the foreground/background colors
                of a TERM0 window.  This is useful for both color and mono-
                chromatic displays.  You can change to black on white lettering
--More--(27%)
```

Figure 2-1. A Term0 Window

Graphics Window Type

Graphics windows emulate the bit-mapped graphics displays supported by HP Windows/9000. They support the Starbase, Font Manager, and Fast Alpha libraries. Starbase graphics applications are easily ported to graphics windows.

Special see-thru and IMAGE graphics windows are simply special cases of the graphics window type. See the appendix "Accelerated 3D Graphics Display Stations" in the *HP Windows/9000 User's Manual.*

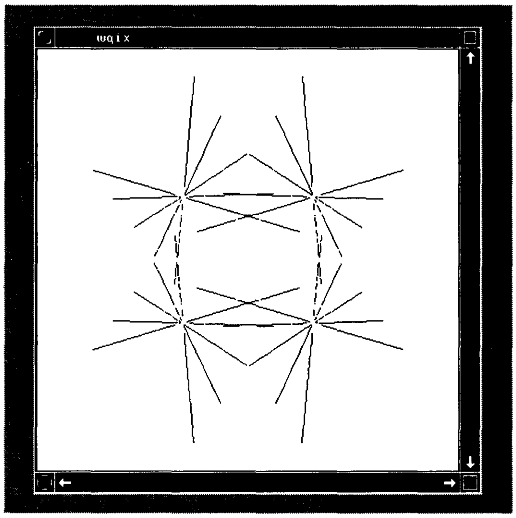


Figure 2-2. A Graphics Window

Programs

Two kinds of programs run in the window system: **window-dumb** and **window-smart**.

Window-Dumb Programs

Window-dumb programs do *not* require windows to run. The *vi(1)* editor is a good example of a window-dumb program: although *vi* can run in windows, it was not created only for windows and can run outside the window system. Window-dumb programs never call Window library routines.

Window-Smart Programs

Unlike window-dumb programs, window-smart programs take advantage of windowing capabilities. In other words, window-smart programs call Window library routines to do windowing tasks. The purpose of this manual is to show you how to use windowing routines to write window-smart programs.

Note

Window-dumb applications are normally invoked only from windows in which a shell is running. Window-smart applications are not easily ported to non-window systems.

Window Type Device Interface

Each window has its own special file (device file) known as the window's **window type device interface**. To identify a window, many routines require, as a parameter, the file descriptor returned from opening (via *open(2)*) the window's window type device interface. The file descriptor identifies which window the routine should work on. For example, the *wmove* Window library routine has the following syntax:

```
wmove(fd, x,y)
```

The *fd* parameter is an integer file descriptor for the opened window type device interface of the window to move. The *x* and *y* parameters are integers specifying the new location for the window.

A window type device interface is a pseudo-terminal special file, or **pty** for short. (For details on *pty* special files, see the *pty(7)* page in the *HP-UX Reference.*) Programs read and write through the slave side of the *pty*, and the *pty*'s master side is controlled by the window system.

Window Name

Some Window library routines require, as a parameter, a window's name. A window's name is simply the basename (see *basename(1)*) of the window's window type device interface.

Each window's window type device interface is stored in the directory specified by the WMDIR window system environment variable. For example, if you create a window named *bunion*, and $WMDIR is */dev/screen*, then the window type device interface for the window is */dev/screen/bunion*.

Note

To determine the value of WMDIR on your system, look in the *wmstart* shell script, which assigns the default value for WMDIR.

To verify this, simply create a few windows via the *wsh* command, and list the WMDIR directory. For example, you might enter:

```
wsh win_1 [Return]
wsh [Return]
wsh win_2 [Return]
ls -l $WMDIR [Return]
```

You should see a special file for every window displayed on the screen.

Note that some of the files listed in the $WMDIR directory may not belong to a window. For example, the window manager has a file in this directory, $WMDIR/*wm*, but has no window.

Window Manager

The **window manager** (*wm*) is a special server created when you start the window system via the *wmstart* command. The window manager is simply a program that manages the window system. It communicates with windows and hardware devices such as the keyboard, locator, and CRT. Only one instance of the window manager is allowed per physical display: you can't have two or more *wm* processes running simultaneously.

Examples of some of the tasks done by the window manager are:

- managing pop-up menus
- repainting the display screen
- channeling communication through windows
- moving and changing the size of windows.

Window Manager Device Interface

The window manager has a **window manager device interface** (special file). Many window routines require, as a parameter, the file descriptor returned from opening the window manager device interface. You can see the window manager special file by listing the $WMDIR directory; the window manager special file is always named *wm.*

The window manager device interface is a *pty* whose master side is attached to the window manager; programs can communicate with the window manager through the slave side.

Window Group

The window manager process heads a **window group** that typically consists of numerous process groups (see *setpgrp(2)*). A window group is all the processes associated with a single instance of the window manager—all processes connected to windows on a single physical display. Window groups are relevant because signals can propagate through them (see *wmstop(1)*).

The window manager runs as a single, setuid (super-user) process. It has no knowledge of login security. Once it is running, anyone can interactively get a shell using a pop-up menu. Running *getty(1M)* within a window is both difficult and useless; in other words, it doesn't make sense to log in users in a window. (Note that you can still use *su(1)* within a window.)

Input Devices

The keyboard and mouse buttons or tablet stylus switch are **attached** to one window at a time. The *wselect* Window library routine **selects** a window, making it attached to the keyboard, mouse buttons, and/or tablet stylus switch. Any process in a selected window can read **locator** information (such as from an optional mouse or graphics tablet) at any time.

Window Attributes

To write window-smart applications that manipulate and manage windows requires some knowledge of **window attributes**. Each window has many attributes which describe characteristics of the window. Window commands and library routines can change window attributes and thus change a window's characteristics. Descriptions of each attribute follow.

Window Name

A window's name, specified when the window is created, is the basename (see *basename(1)*) of the window's window type device interface. By default, the window name is displayed as the window's label (in the border), and in its icon, softkeys, and pop-up menu. The name is set when the window is created and cannot be changed. (Note, however, that a window's *label* can be changed to something other than the window's name.)

Window Type

The type of the window—either term0 or graphics—is set when the window is created. The type cannot be changed.

Representation

Any window may be displayable (in normal or icon form) or concealed. A displayable window may be located partly or totally off screen, or occluded by other windows so that it is not actually visible at all. A concealed window, however, is never displayed, regardless of location or other attributes.

Location

Each window has two locations on the display screen: one for its normal form, the other for its iconic form. Both may be set to defaults determined by the window manager. Both the normal and iconic location can be changed.

Size

Each window has a displayable size for its normal form. This is the number of displayable *x,y* pixels (for graphics windows) or rows and columns (for term0 windows). The size can be set when the window is created, and modified after creation, if necessary.

Place in Stack

Each window, if not concealed, has a place in the display stack. Windows toward the top of the stack occlude those lower in the stack. Windows can be shuffled (rotated) up or down through the display stack, and any window can be made visible as the top or bottom window in the stack.

View

Each graphics window (with a retained raster) provides a view into its corresponding virtual device (an underlying area of memory that may be larger than the window size). The view of a window into its virtual device is just that part of the virtual device seen through the window (as long as it is displayable and not occluded). Panning is the operation of moving the view position over the virtual device. Panning does not change the location of the window; rather, the virtual device appears to move under the window.

Keyboard Selection

At any time, only one window is selected, meaning it is attached to the keyboard for input. The currently selected window has a line through its border and an asterisk (*) to the left of its label. When a window becomes un-selected, the line and asterisk disappear.

Softkeys

Each window has up to sixteen softkeys whose displayed labels and values may be set using escape sequences or library routines. (See the "Graphics Window Softkeys" and "Term0 Windows" chapters for details.) When a window is selected, its softkey labels are displayed at the bottom of the display screen.

Pop-Up Menu Type

The type and status of a window affect the choices available when you pop up an interactive menu.

Border Type

A term0 or graphics window's border may be:

- **normal**, in which case the label and manipulation areas are present
- **thin**, in which case the border is just a thin line surrounding the window

In addition to these two types, graphics windows can have **null** borders—that is, no border.

Label

A window's label is a text string displayed in the window's border. By default, the label is identical to the window's name (window spec) used when the window was created. The *wborder(1)* command or the *wsetlabel(3W)* Window library routine can change a window's label to a string other than the default name. The maximum length of a window's label depends on the window's type.

Some commands or routines may require a window's name to manipulate it. If a window's label is different from its name, then attempting to use the window's label for such commands or routines will result in a window system error.

Colors

Each window has various colors attributes. Most are controlled through writing or plotting to the window. However, colors can be controlled through library routines. Note also that **all colors are indices into the display device's color map; also, there is only one color map per physical display**.

The window system maintains and manages the following colors per window:

- font foreground
- font background
- border foreground
- border background
- user foreground
- user background

Note that eight different color combinations of font foreground and background colors can appear simultaneously in term0 windows; they can be changed non-retroactively via Fast Alpha and Font Management routines.

Raster or Scroll Buffer Size

Graphics windows have a maximum raster size; term0 windows have a maximum scroll buffer size. This size is set when the window is created and cannot be changed. The raster/buffer size denotes the maximum size to which the window can grow. For term0 windows the maximum size is measured in rows and columns of characters; for graphics windows, rows and columns of pixels.

Raster Retention

For **graphics** windows, the raster may be *retained*, in which case memory is allocated at window creation time to save occluded areas. It can also be *non-retained*, in which case no memory is allocated for the image. A graphics window may be retained in a **byte-per-pixel** or **bit-per-pixel** format (see appendix for linking information).

Windows that are not retained may not be properly repainted by the window manager because no memory for the image is allocated from which the windows can be redrawn. Programs that use non-retained windows must take care of repainting windows.

For **term0** windows, no raster is allocated, but the window acts as though it were retained, because it is redrawn using its scroll buffer, which contains any characters displayed in the window.

Retention is set when the window is created, and cannot be changed.

Window Paused

Output to **term0** windows can be paused, i.e., temporarily suspended. The window paused attribute denotes whether a term0 window's output is currently paused.

Autotop (Term0 Only)

On Term0 windows, the autotop attribute allows a window to be "marked" to be displayed automatically as the top window in the display stack whenever output is sent to its window type device interface.

Auto-Selection (Term0 Only)

On Term0 windows, a window can also be "marked" to be selected when output is sent to its device interface if this attribute is on.

Writing Window-Smart Programs

Using the Window library routines described in this manual, you can write window-smart programs that use windows and windowing capabilities. Figure 2-3 illustrates how window-smart programs interact with the window system.

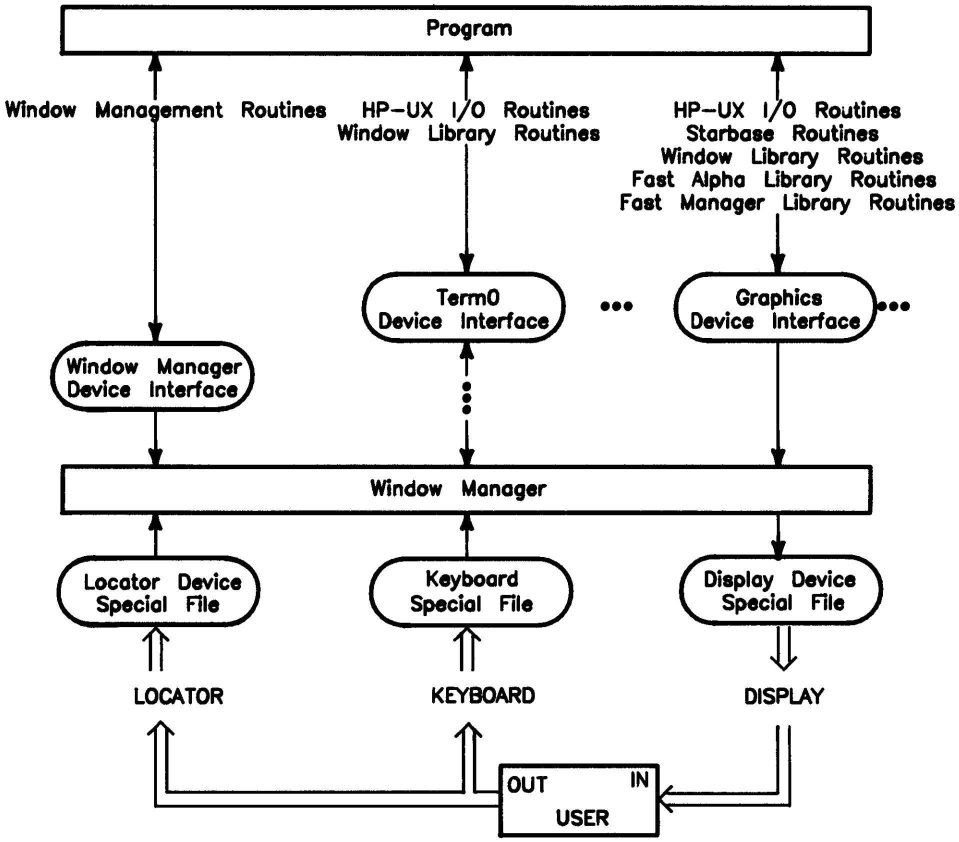


Figure 2-3. Program Interaction with Window System

Using HP-UX system calls (e.g., *open(2)*, *close(2)*, *read(2)*), HP-UX subroutines (e.g., *getc(3C)*, *putc(3C)*, etc), or both, a program can read input from and write output to a given window through the window's window type device interface. (Note, however, that programs cannot use *write(2)* to write to **graphics** windows; instead, they must use Fast Alpha, Font Manager and/or Starbase library routines to do graphics output to a graphics window.)

A program can call Window routines to change a window's attributes—e.g., change the window's size, location, or representation. Like HP-UX system calls and subroutines, library routines manipulate a window through the window's device interface. Routines that manipulate a given window require the file descriptor returned from **starting communication with the window.** (See the "Window Manipulation" chapter for details on starting window communication.)

Additionally, a program can call Window routines to do window manager functions, such as repainting the display screen, shuffling windows, and creating or destroying windows. These routines communicate with the window manager the window manager's device interface. Routines that do window manager functions require the file descriptor returned from opening (via *open(2)*) the window manager's device interface.

Window routines are described in detail in the remainder of this manual.

Writing Window-Dumb Programs

When the window system is not running, programs interact with the bit-mapped display and its keyboard through an **internal terminal emulator** (**ITE**). The ITE causes the bit-mapped display to appear to be a simple terminal; thus, programs not written for bit-mapped displays can still run on the bit-mapped display through the ITE.

The Fast Alpha, Font Manager, and Starbase libraries can also be used to write to a bit-mapped display when Windows/9000 is not running.

/dev/console is a typical path name of the special file for the terminal emulated by the ITE. The ITE accepts input only from the keyboard and ignores the optional mouse or graphics tablet if they are present. Figure 2-4 illustrates the ITE architecture.

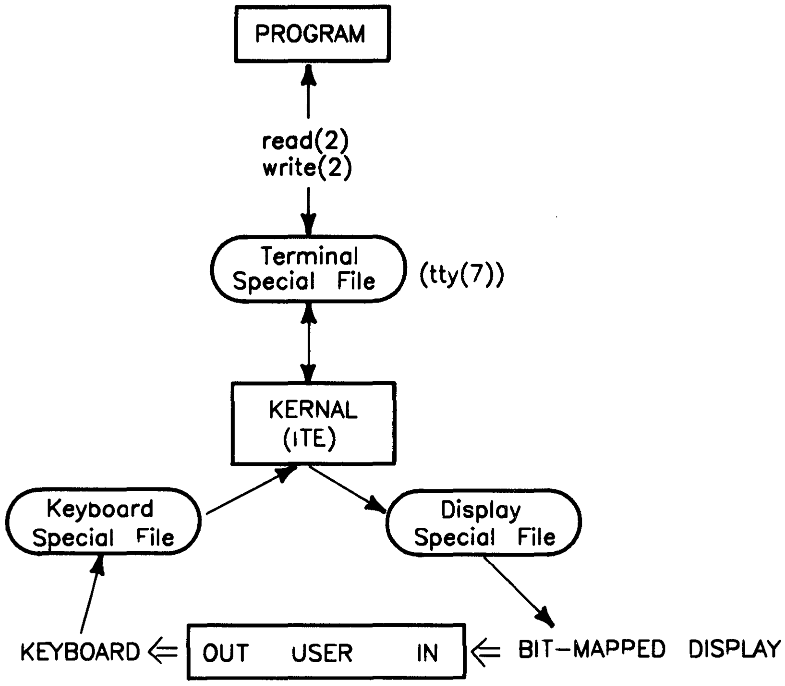


Figure 2-4. Architecture without Windows

The following references provide more information on this topic:

- For more details on the internal terminal emulator, see the ITE article in *HP-UX Concepts and Tutorials: Facilities for Series 200, 300, and 500.* Also see *tty(7).*
- To use graphics with the ITE, see *HP-UX Concepts and Tutorials: Starbase Graphics Techniques.*

Notes

Window Management 3

This chapter describes how to use **window management** routines found in the Window library. By calling window management routines, a program can:

- create term0 or graphics windows
- destroy windows
- shuffle windows
- repaint the screen
- kill the window manager.

These tasks are described in the following sections.

Concepts

This section discusses concepts essential to using window management routines. Be sure to read this section before any others in this chapter.

The Window Manager

Window management routines communicate directly with the window manager. Therefore, before a program can call window management routines, the window manager must be running and the window manager device interface ($WMDIR/*wm*) must exist.

There are two ways to be sure of this:

1. Start the window system via the *wmstart(1)* command before running the program that calls window management routines.
2. Start the window system from the program itself, using the *system(3)* HP-UX subroutine to invoke *wmstart* from the program.

   NOTE: If you start the window system this way, be sure to invoke *wmstart* as a background process (with a trailing ampersand), e.g., `system("wmstart &")`. If you do not invoke *wmstart* as a background process, then your program will hang indefinitely because *wmstart* does not return until the window system is exited.

It may take a few seconds time for the window manager device interface to be created after *wmstart* is executed, so be sure to wait until it really exists before trying to start communication. (You can use the *wmready(1)* command to determine whether the window manager is running.)

Starting Window Manager Communication

Before doing any other window management task, a program must start communication with the window manager, which entails the following steps:

1. Build the path name of the window manager device interface.
2. Open the window manager device interface.
3. Call *winit(3W)* on the window manager device interface.

Each step is described in detail next.

Build the Path Name of the Window Manager Device Interface

The path name of the window manager device interface is normally $WMDIR/*wm*. To build the path name, you can use the *wmpathmake(3W)* routine, which builds a path name from an environment variable and a user-supplied file name; its syntax is:

`wmpathmake(`*environ, suffix, target*`)`

The *environ* parameter points to a null-terminated character string containing the name of an environment variable; *suffix* points to a base name that will be appended to the value of the environment variable pointed to by *environ*. The resulting path name is pointed to by the *target* parameter.

Open the Window Manager Device Interface

Using the *open(2)* HP-UX system call, open the window manager device interface with read/write permission (`O_RDWR`). The returned file descriptor is required as a parameter to other window management routines.

Call winit(3W) on the Window Manager Device Interface

Initialize the window manager via the *winit(3W)* routine; its syntax is:

`winit(`*wmfd*`)`

The *wmfd* parameter is the file descriptor returned from opening the window manager device interface.

At this point, window manager communication is started, and the program can begin doing other window management tasks.

Stopping Window Manager Communication

When finished doing window management tasks, a program must stop communication with the window manager, which entails the following steps:

1. Call *wterminate(3W)* on the window manager device interface.
2. Close the window manager device interface.

These steps are described in detail next.

Call wterminate(3W) on the Window Manager Device Interface

Call the *wterminate(3W)* routine; its syntax is:

```
wterminate(wmfd)
```

The *wmfd* parameter is the file descriptor obtained when the program started communication with the window manager.

Note that *wterminate* does **not** kill the window system as does the *wmstop* command and the *Exit WS* option of the pop-up menu.

Close the Window Manager Device Interface

Finally, you must close the window manager device interface using *close(2)* HP-UX system call.

Example

The following program starts and stops communication with the window manager. Any program that calls window management routines must conform to the structure of this program. That is, the program must:

1. Start communication with the window manager.
2. After starting window manager communication, the program can call other window management routines.
3. When the program is finished doing window management tasks, it must stop communication with the window manager.

```
#include <fcntl.h>      /* system call i/o definitions */
#include <stdio.h>      /* system subroutine i/o definitions */
#include <window.h>     /* window library definitions  */
main()
{
    int     wmfd;                       /* window manager file descriptor */
    char    wm_path[WINNAMEMAX];        /* window manager path name */

/*
 * START WINDOW MANAGER COMMUNICATION:
 *
 * STEP 1:  Create the path name of the window manager device interface:
 */
        wmpathmake("WMDIR", "wm", wm_path);

/*
 * STEP 2:  Open the window manager device interface:
 */
        if ((wmfd = open(wm_path, O_RDWR)) < 0) {
             perror("open window manager device interface failed");
             exit(1);
        }

/*
 * STEP 3:  Call winit on the window manager device interface:
 */
        if (winit(wmfd) < 0) {
             perror("winit on window manager device interface failed");
             exit(2);
        }

        printf("Communication with wm started successfully\n");


/*
 * The program can now call other window management routines in here.
 *          .
 *          .
 *          .
 */
/*
 * STOP WINDOW MANAGER COMMUNICATION:
 *
 * STEP 1:  Call wterminate on the window manager device interface:
 */
        if (wterminate(wmfd) < 0) {
            perror("wterminate failed on window manager device interface.");
            exit(3);
        }

/*
 * STEP 2:  Close the window manager device interface:
 */
        if (close(wmfd) < 0) {
            perror("close failed on window manager device interface.");
            exit(3);
        }

        printf("Communication with wm stopped successfully\n");
        exit(0);
}
```

est_wm_com and term_wm_com

To shorten source code examples throughout this manual, two routines are provided in the *man_examples* directory: **est_wm_com.c** and **term_wm_com.c**. They **are not Window library routines**; they are provided here only to shorten examples. Nevertheless, you may find them useful, and you can compile them separately and link them with your programs.

est_wm_com

The *est_wm_com* routine starts communication with the window manager. This routine, if used, should be called before a program does any window management tasks. If it cannot start communication, it returns −1; otherwise, it returns the file descriptor for the window manager device interface.

```
#include    <fcntl.h>   /* system call i/o definitions */
#include <window.h>     /* window library definitions  */
est_wm_com()
{
    int     wmfd;                        /* window manager file descriptor */
    char    wm_path[WINNAMEMAX];         /* window manager path name       */
/*
 * STOP WINDOW MANAGER COMMUNICATION:
 *
 *      return -1 if an error occurs
 *      return the window manager file descriptor (wmfd) if successful
 *
 * STEP 1:  Create the path name of the window manager device interface:
 */
        wmpathmake("WMDIR", "wm", wm_path);
/*
 * STEP 2:  Open the window manager device interface:
 */
        if ((wmfd = open(wm_path, O_RDWR)) < 0) return(-1);
/*
 * STEP 3:  Call winit; set this function's return value accordingly.
 */
        if (winit(wmfd) < 0) {
                close(wmfd);
                return(-1);
        } else
                return(wmfd);
}
```

term_wm_com

The *term_wm_com* routine stops communication with the window manager. This routine, if used, should be called only after a program is finished with all window management tasks. If it fails, it returns −1; otherwise, it returns 0.

```
#include <fcntl.h>      /* system i/o call definitions */
#include <window.h>     /* window library definitions  */
term_wm_com(wmfd)
int    wmfd;            /* file descriptor for wm device interface */
{
/*
 * STOP WINDOW MANAGER COMMUNICATION:
 *
 *         return -1 if an error occurs
 *         return  0 if termination is successful
 *
 * STEP 1:  Call wterminate:
 */
        if (wterminate(wmfd) < 0) {
                close(wmfd);
                return(-1);
        }
/*
 * STEP 2:  Close the window manager device interface:
 */
        if (close(wmfd) < 0)
                return(-1);
        else
                return(0);
}
```

Creating a Term0 Window

The *wcreate_term0(3W)* routine creates a term0 window. It creates the window's window type device interface in the $WMDIR directory.

Note that **creating a window does not make the window visible on the display screen**. For details on making term0 and graphics windows visible, see "Displaying and Concealing a Window" in the chapter "Window Manipulation."

All newly created windows inherit their run-time environment from that which existed when the *wmstart(1)* command was executed. For example, if you invoke *wmstart* from the */usr/lib/hpwindows/demo* directory, the first window created, *wconsole*, will have its current directory set to the same, and so will any windows created thereafter.

Procedure

To create a term0 window, follow these steps:

1. Build the path name of the window's window type device interface.
2. Compute the window's anchor point.
3. Call *wcreate_term0(3W)*.

Details on each step come next.

Build the Path Name of the Window's Window Type Device Interface

Determine an unused name for the window—a name no other window is using—and build the path name for its window type device interface. This path name should be $WMDIR/*name*, where *name* is the window name. The path name is required as a parameter to *wcreate_term0.*

The easiest way to build the path name is by using the *wmpathmake(3W)* routine, which builds a path name from an environment variable and a user-supplied suffix; its syntax is:

`wmpathmake(`*environ, suffix, target*`)`

After calling this routine, the *target* parameter will point to a path name created by appending the *suffix* string to the environment variable pointed to by *environ* (in this case, "WMDIR").

You can also obtain a window name from the *wdfltpos(3W)* routine, which returns default name and anchor point values for windows created interactively through the window manager.

Compute the Window's Anchor Point

Compute the window's anchor point in *x,y* pixel coordinates. The upper-left corner of the screen is the origin *0,0*. *X* coordinates increase as you move to the right on the screen; *y* coordinates increase as you move down on the screen. Three methods can be used to compute the anchor point:

- You can specify **absolute** coordinates. This simply means that if you want the window to appear at a specific *x,y* location on the screen, you specify the exact *x,y* values as parameters to the *wcreate_term0* routine. For example, if you want a term0 window's anchor point to appear at 300 pixels over and 200 pixels down from the upper-left corner, simply supply *300,200* as the *x,y* coordinates to *wcreate_term0.*
- You can use **default** coordinates supplied by *wdfltpos.* This routine returns the round-robin default position for the next window (or icon). When using this routine, window coordinates stair-step down from the upper-left corner of the screen. Note that the window manager calls this routine to determine new window coordinates when you interactively create windows.
- To specify coordinates **relative** to an existing window, use the *wgetcoords(3W)* routine, which allows you to obtain coordinates (and other important information) for any existing window.

  Compute relative coordinates by adding to or subtracting from the *x,y* coordinates returned from *wgetcoords.* For example, if you want a new window to appear 50 pixels up and 70 pixels to the right of an existing window, simply use *wgetcoords* to get the existing window's coordinates, add 70 to the *x*-coordinate, subtract 50 from the *y*-coordinate, and supply these new coordinates to *wcreate_term0.*

Call wcreate_term0

Finally, you must call *wcreate_term0(3W)*, which will create a window type device interface for the window; the syntax for this routine is:

```
wcreate_term0(wmfd, name, x,y,
              wincols, winrows,
              scrncols, scrnrows,
              bufcols, bufrows,
              basefont, altfont,
              colormode, border)
```

The *wmfd* parameter is the file descriptor returned from starting window manager communication; *name* points to the window's path name (determined above); *x,y* specify the window's anchor point.

Note that the window's device interface path name cannot contain more characters than the value of `WINNAMEMAX` − 2. The last character must be NULL (`'\0'`).

The *wincols*, *winrows* parameters specify the width and height (in columns and rows) of the window to create; the window can become no larger than these values.

The *scrncols*, *scrnrows* parameters specify the number of columns and rows in the terminal being emulated. For term0 windows, these values should be 80 columns by 24 rows.

The *bufcols*, *bufrows* parameters specify the size of the scroll buffer to be used with the window. These values should be at least as large as the *scrncols*, *scrnrows* parameters; they can be larger if desired. Typically, *bufrows* is given as two times the *scrnrows* value; this way, you can buffer up to two screens of window information.

The *basefont* and *altfont* parameters specify the fonts to use for the base and alternate fonts respectively. These parameters point to the path names of font files found under the font directory specified by the WMFONTDIR environment variable. For more details on fonts in term0 windows, see the "Term0 Windows" chapter.

The *colormode* parameter enables color mode on color systems; it should always be set to `COLORMODE` as defined in the header file *window.h.*

The *border* parameter determines whether the border is normal (interactive manipulation areas present) or thin (no manipulation areas). If *border* is `SETBANNER`, the window will have a normal border, if *border* is `SETNOBANNER`, no banner will be displayed, and if *border* is `SETNULLBANNER`, the window will have no border.

Precautions

- Creating a window does not make the window visible. In fact, nothing can be done with a newly created window until communication is started with the window. For details on window communication, see the "Concepts" section of the "Window Manipulation" chapter.
- The name of a newly created window must be unique: it cannot have the same name as any existing window.

Example

The following program creates a term0 window named *flebnee*; its anchor point is determined by *wdfltpos*. The program does *not* display the window; it merely creates the window's device interface. To display the window use the *wdisp(1)* window command as:

```
wdisp flebnee
```

The source is named *create_t0.c* and is found in the *man_examples* directory. The program calls the functions *est_wm_com* and *term_wm_com* described in the "Concepts" section. (See the appendix "Compiling Programs" for details on compiling window programs.)

```
#include <window.h>   /* window library CONSTANT definitions are kept here */
main()
{
    int wmfd;                     /* window manager file descriptor        */
    char wt_path[WINNAMEMAX];     /* path name for window type             */
    int wx,wy,ix,iy;              /* parameters for wdfltpos routine       */
    char *dflt_name;              /* dummy name parameter for wdfltpos     */
    int  est_wm_com();            /* routine to start wm communication */
    int  term_wm_com();           /* routine to stop wm communication */

/*
 * START WINDOW MANAGER COMMUNICATION:
 */
    if ((wmfd = est_wm_com()) == -1)
    {
            perror("est_wm_com");
            exit(1);
    }
/*
 * CREATE THE TERMO WINDOW:
 *
 * STEP 1:  Create the path name for the window.
 */
    wmpathmake("WMDIR", "flebnee", wt_path);

/*
 * STEP 2:  Get default coordinate from wdfltpos.
 */
    if (wdfltpos(wmfd, DFLT_WPOS, &wx,&wy, &ix,&iy, dflt_name) < 0)
    {
        perror("wdfltpos wmfd");
        exit(1);
    }

/*
 * STEP 3:  Determine remaining parameters and call wcreate_term0:
 *
 *                  Initial window size (wincols, winrows) is 80 columns
 *                          by 24 rows of characters.
 *                  Character width and height of the emulated terminal
 *                          (scrncols, scrnrows) is 80 columns by 24 rows.
 *                  The standard 8x16-pixel font will be used as the base
 *                          font (*basefont).
 *                  No alternate font is used (*altfont = ALTFONTNULL).
 *                  Enable color mode (colormode = COLORMODE).
 *                  Normal border (border = SETBANNER).
 */
    if (wcreate_term0(wmfd, wt_path, wx,wy, 80,24, 80,24, 80,48,
        "/usr/lib/raster/8x16/lp.8U", ALTFONTNULL, COLORMODE, SETBANNER) < 0)
    {
        perror("wcreate_term0 wmfd");
        exit(1);
    }

/*
 * STOP WINDOW MANAGER COMMUNICATIONS:
 */
    if (term_wm_com(wmfd) == -1)
    {
        perror("term_wm_com wmfd");
        exit(1);
    }
    exit(0);
}
```

Creating a Graphics Window

Creating a graphics window is similar to creating a term0 window. However, the routine that creates the graphics window—*wcreate_graphics(3W)*—requires some different parameters than *wcreate_term0.*

All newly created windows inherit their run-time environment from that which existed when the *wmstart(1)* command was executed. For example, if you invoke *wmstart* from the */usr/lib/hpwindows/demo* directory, the first window created, *wconsole*, will have its current directory set to the same, and so will any windows created thereafter.

Procedure

To create a graphics window, a program must:

1. Build the path name of the window's window type device interface.
2. Compute the window's anchor point.
3. Call wcreate_graphics(3W).

Build the Path Name of the Window's Window Type Device Interface

Determine an unused name for the window—a name no other windows are using. Use this name to build the path name of the window type device interface for the window to create. Normally, the path name will be $WMDIR/*name*, where *name* is the window name. Use the *wmpathmake* routine to build the path name. *Wdfltpos* can be used to get a default window name.

Compute the Anchor Point

Compute the graphics window's anchor point in x,y pixel coordinates. There are three methods for computing the anchor point coordinates:

- You can specify **absolute** coordinates.
- You can use **default** coordinates supplied by *wdfltpos.*
- You can specify coordinates **relative** to another window.

See the section "Creating a Term0 Window" for details on computing coordinates using these three methods.

Call wcreate_graphics

Finally, you must determine the remaining parameters and call *wcreate_graphics*, which creates the window's device interface. The syntax for this routine is:

```
wcreate_graphics(wmfd, wname, x,y,
                 w, h,
                 rasterw, rasterh,
                 attributes, border)
```

The *wmfd* parameter is the file descriptor returned from starting communication with the window manager. The *wname* parameter points to the path name for the window's device interface; *x,y* specify the window's anchor point.

Note that the window's device interface path name cannot exceed `WINNAMEMAX` characters in length; the last character must be NULL.

The *w, h* parameters specify the initial width and height (in pixels) of the view into the virtual raster.

The *rasterw, rasterh* parameters define the width and height of the window's virtual raster. The window's size (specified by *w* and *h*) can grow no larger than the raster size.

The *attributes* parameter specifies certain attributes of the graphics window. If *attributes* is `SETRETAIN`, the raster is retained as byte/pixel (`SETRETAIN` and `SETRETAINBYTE` are defined to be the same value). If *attributes* is `SETRETAINBIT`, the raster is retained as bit/pixel (this only applies to monochrome displays). If *attributes* is set to `SETNORETAIN`, the raster is not retained. Finally, if *attributes* is `SETIMAGE`, the raster is not retained and the user area of the window is mapped into the image planes of the display (this only applies on the HP 98730).

The *border* parameter determines whether the border is normal (interactive manipulation areas present), thin (no manipulation areas), or null (no border). If *border* is `SETBANNER`, the window will have a normal border; if *border* is `SETNOBANNER`, the window will have a thin border; if *border* is `SETNULLBANNER`, the window will have no border.

Precautions

- Remember that creating a window does not make the window visible. In fact, a program cannot manipulate a window unless it first starts communication with the window. For details, see the "Concepts" section of the "Window Manipulation" chapter.
- The name of a newly created window must be unique: it cannot have the same name as any existing window.

Example

The following program, *create_gr.c*, creates a graphics window named *solipsist*. The program is found in the *man_examples* directory.

The anchor point of *solipsist* is at *350,100*; its virtual raster is 512 pixels wide and 398 pixels high; and the window is created with an initial width and height of 300 by 200 pixels. In addition the raster is not retained, and it has a normal border.

Note that the program does *not* display the window; it merely creates the window's device interface. To display the window use the *wdisp(1)* window command as:

```
wdisp solipsist
```

The program calls the *est_wm_com* and *term_wm_com* routines defined in the "Concepts" section. Therefore, for the program to work properly, it must be compiled with those functions.

```
#include <window.h>        /* window library defintions               */
main()
{
    int wmfd;                   /* window manager file descriptor    */
    char wt_path[WINNAMEMAX];   /* path name for window type         */
    int wx,wy;                  /* x,y parameters for window location */
    int est_wm_com();           /* start wm communication        */
    int term_wm_com();          /* stop wm communication        */
/*
 * START WINDOW MANAGER COMMUNICATION:
 */
    if ((wmfd = est_wm_com()) == -1)
    {
        perror("est_wm_com");
        exit(1);
    }

/*
 * CREATE THE GRAPHICS WINDOW:
 *
 * STEP 1:  Create the path name:
 */
    wmpathmake("WMDIR", "solipsist", wt_path);

/*
 * STEP 2:  Assign the window coordinates (350,100):
 */
    wx = 350; wy = 100;

/*
 * STEP 3:  Determine the remaining parameters and call wcreate_graphics:
 *
 *          The width and height (w,h) of the view into the virtual raster
 *                  is 300 by 200 pixels.
 *          The virtual raster (rasterw,rasterh) is 512 pixels wide by
 *                  398 pixels high.
 *          The raster will not be retained (retained = SETNORETAIN).
 *          The window will have a normal border (border = SETBANNER).
 */
    if (wcreate_graphics(wmfd, wt_path, wx,wy, 300,200, 512,398,
                         SETNORETAIN, SETBANNER) < 0)
    {
        perror("wcreate_graphics wmfd");
        exit(1);
    }
/*
 * STOP WINDOW MANAGER COMMUNICATION:
 */
    if (term_wm_com(wmfd) == -1)
    {
        perror("term_wm_com wmfd");
        exit(1);
    }
    exit(0);
}
```

Destroying a Window

A program can destroy any window—those created by commands and your programs, or those created interactively via the pop-up menu. The *wdestroy(3W)* routine destroys windows.

A destroyed window is immediately concealed and the window type device interface will be removed. But the window will not really be destroyed until it is closed by every process that has it open.

When a window is destroyed, all processes affiliated with the window which do not catch the SIGHUP signal will be killed.

Procedure

To destroy a window, a program must:

1. Build the path name of the window's window type device interface.
2. Call *wdestroy.*

Build the Path Name of the Window's Window Type Device Interface

To destroy a window you created, simply use the path name you supplied as a parameter to the *wcreate_term0* or *wcreate_graphics* routine. To destroy a window created externally to your program, build the path name using the *wmpathmake(3W)* routine.

Call wdestroy

Call the *wdestroy* routine; the syntax for this routine is:

```
wdestroy(wmfd, wt_path)
```

The file descriptor returned from starting window manager communication is passed as the *wmfd* parameter. The path name of the window to destroy is pointed to by the *wt_path* parameter; it should be a null-terminated string.

Related Routines

By using the *wrecover(3W)* and *wautodestroy(3W)* routines you can change the way windows are destroyed. See the section "Setting Autodestroy Status" in the "Window Manipulation" chapter for details on using these routines.

Precautions

- Destroying a window can be dangerous: **Once a window is destroyed, it cannot be recovered.** In addition, all processes affiliated with the window which do not catch the SIGHUP signal will be killed. Therefore, be careful when destroying windows.

- If a process is not affiliated to a window and it has the window open, then the process will not receive the SIGHUP signal sent when the window is destroyed. Because of this, the window will continue to exist, unoccluded. You can determine which windows are in this state by using the *wlist(1)* command as follows:

  ```
  wlist \*
  ```

  Wlist will display "`I can't find` *window-name* `window`" for each window in this state.

- If the selected window is destroyed, the keyboard is attached to the resulting top window (if one exists; otherwise it is not attached to any window).

- When a **term0** window is destroyed, it will send the SIGCLD signal to its parent process. For the window to be completely destroyed, the parent process must receive the signal. To receive the signal you should call the *signal(2)* system call as follows:

  ```
  signal(SIGCLD, SIG_IGN);
  ```

  This will cause the signal to be received but ignored, and the window will be completely destroyed. If you don't do this, then the destroyed window will become a defunct process.

- When writing code to create graphics windows on Series 300 HP-UX 5.2 and later systems, you needn't worry about defunct processes for **graphics** windows. However, if you run pre-5.2 **compiled** window source on 5.2 or later systems, then you must handle defunct processes as described above; otherwise, defunct processes will exist until the program exits.

Example

The following program, named *rm_window.c*, is a simplified version of the *wdestroy(1)* command: it destroys a window specified by the user. Its syntax is:

```
rm_window windowspec
```

where *windowspec* is the name of the window type device interface for the window to destroy.

The source for this program is found in the *man_examples* directory; it calls the functions found in the sections "Starting Window Manager Communication" and "Stopping Window Manager Communication."

```
#include <window.h>      /* window library definitions */
main(argc, argv)
int     argc;           /* number of arguments on command line        */
char    *argv[];        /* command line argument list                 */
{
    int wmfd;                  /* window manager file descriptor      */
    char wt_path[WINNAMEMAX]; /* path name for window type            */
    int est_wm_com();   /* starts window manager communication   */
    int term_wm_com();  /* stops window manager communication    */

/*
 * START WINDOW MANAGER COMMUNICATION:
 */
    if ((wmfd = est_wm_com()) == -1)
    {
        perror("est_wm_com");
        exit(1);
    }
/*
 * DESTROY THE WINDOW:
 *
 * STEP 1:  build the path name of the window to destroy:
 */
    wmpathmake("WMDIR", argv[1], wt_path);
/*
 *
 * STEP 2:  Call wdestroy:
 */
    if (wdestroy(wmfd, wt_path) < 0)
    {
        perror("wdestroy wmfd");
        exit(1);
    }

/*
 * STOP WINDOW MANAGER COMMUNICATION:
 */
    if (term_wm_com(wmfd) == -1)
    {
        perror("term_wm_com wmfd");
        exit(1);
    }
    exit(0);
}
```

Shuffling Windows

When one or more windows are visible on the display screen, they can be shuffled via the *wshuffle(3W)* routine. Windows can be shifted (or shuffled) either upward or downward through the display stack. If windows are shuffled upward, the bottom window in the stack becomes the top, and the other windows are shifted down one position in the display stack. If windows are shuffled downward, the top window in the stack is placed on bottom, and the remaining windows are shifted up one position.

Note

When windows are shuffled, the keyboard is attached to the resulting topmost window in the display stack—the top window becomes selected.

You can call *wshuffle* when no windows are visible on the screen; however, the window system and the display screen will not change as a result of using the routine in this state, and the selected window will not change either.

Procedure

To shuffle windows, call *wshuffle*; its syntax is:

```
wshuffle(wmfd, value)
```

The *wmfd* parameter specifies the file descriptor of the open window manager device interface.

The *value* parameter gives the direction of the shuffle. If *value* is `SHUFFLEDOWN`, the windows are shuffled down; if *value* is `SHUFFLEUP`, windows are shuffled up.

Example

The following program, named *shuffle_dn.c*, causes the top window to shuffled to the bottom and the remaining windows to be shifted up one position through the display stack.

The source is found in the *man_examples* directory; the program calls the *est_wm_com* and *term_wm_com* functions defined earlier.

```
#include <window.h>
main()
{
    int wmfd;                   /* file descriptor for wm device interface */
    char wm_path[WINNAMEMAX]; /* path name for wm device interface       */
    int est_wm_com();           /* starts wm communication            */
    int term_wm_com();          /* stops wm communication             */

/*
 * START WINDOW MANAGER COMMUNICATION:
 */
    if ((wmfd = est_wm_com()) == -1)
    {
        perror("est_wm_com");
        exit(1);
    }

/*
 * SHUFFLE TOP WINDOW TO BOTTOM AND MOVE OTHERS UP ONE POSITION:
 */
    if (wshuffle(wmfd, SHUFFLEDOWN) < 0)
    {
        perror("wshuffle wmfd");
        exit(1);
    }

/*
 * STOP WINDOW MANAGER COMMUNICATION:
 */
    if (term_wm_com(wmfd) == -1)
    {
        perror("term_wm_com wmfd");
        exit(1);
    }
    exit(0);
}
```

Repainting the Display Screen

Repainting the display screen causes the entire display screen to be redisplayed: all windows, icons, and typing aids are redisplayed on the desk top. This task is required only when part of the display becomes mussed—for example, if a graphics program accidentally writes over the desk top, making the display difficult to understand.

Procedure

To repaint the display screen, call *wmrepaint*; its syntax is:

```
wmrepaint(wmfd)
```

The *wmfd* parameter is the file descriptor returned from starting window manager communication.

Example

The following program, named *wrepaint.c*, causes the display screen to be repainted. The source is found in the *man_examples* directory, and the program calls the *est_wm_com* and *term_wm_com* functions presented earlier.

```
#include <window.h>
main()
{
    int wmfd;           /* file descriptor for wm device interface */
    int est_wm_com();   /* start wm communication              */
    int term_wm_com();  /* stop wm communication              */

/*
 * START WINDOW MANAGER COMMUNICATION:
 */
        if ((wmfd = est_wm_com()) == -1)
        {
            perror("est_wm_com");
            exit(1);
        }
/*
 * REPAINT THE DISPLAY SCREEN:
 */
    if (wmrepaint(wmfd) < 0)
    {
        perror("wshuffle wmfd");
        exit(1);
    }

/*
 * STOP WINDOW MANAGER COMMUNICATION:
 */
    if (term_wm_com(wmfd) == -1)
    {
        perror("term_wm_com wmfd");
        exit(1);
    }
    exit(0);
}
```

Killing the Window Manager

When a program kills the window manager:

- all existing windows are destroyed
- the window manager stops executing
- control of the physical display returns to the ITE.

The *wmkill(3W)* routine "marks" (flags) the window system to be killed when the program stops communication with the window manager.

CAUTION

Extreme caution should be exercised when using this routine. It will destroy all windows and kill all processes affiliated to the window system; in addition, the window manager will stop executing.

Procedure

To kill the window manager, call the *wmkill* routine; its syntax is:

```
wmkill(wmfd)
```

The *wmfd* parameter specifies the file descriptor returned from starting communication with the window manager.

Note

Wmkill should be called immediately before stopping window manager communication. No other calls to window routines should be made between killing the window manager and stopping communication.

Precautions

Be aware that all windows will be destroyed and the window manager will stop executing when this task is done. Do this task only if you are absolutely sure you want to exit the window system.

Example

When executed from the window system, the following program kills the window manager. It is equivalent in effect to executing the *wmkill(1)* command or selecting the *Exit WS* option of the pop-up menu. Be aware of the consequences of executing this program.

The program is named *kill_wm.c* and is found in the *man_examples* directory. It calls the *est_wm_com* and *term_wm_com* functions described in the "Concepts" section.

```
#include <window.h>
main()
{
    int wmfd; /* file descriptor for wm device interface */
    int est_wm_com();   /* start wm communication  */
    int term_wm_com();  /* stops wm communication   */

/*
 * START WINDOW MANAGER COMMUNICATION:
 */
        if ((wmfd = est_wm_com()) == -1)
        {
            perror("est_wm_com");
            exit(1);
        }

/*
 * KILL THE WINDOW MANAGER:
 */
        if ((wmkill(wmfd) < 0)
        {
            perror("wmkill wmfd");
            exit(1);
        }
/*
 * STOP WINDOW MANAGER COMMUNICATION:
 */
    if (term_wm_com(wmfd) == -1)
    {
        perror("term_wm_com wmfd");
        exit(1);
    }
    exit(0);
}
```

Notes

Window Manipulation 4

This chapter describes how to use **window manipulation** routines, which change window attributes (e.g., size, location, or label). By calling window manipulation routines, a program can:

- select a window
- display or conceal a window
- move a window
- change a graphics window's size
- change a term0 window's size
- change a window's border
- change a window's label
- change a window's border colors
- pause and resume output to a term0 window
- pan a graphics window.

Concepts

This section discusses concepts essential to using window manipulation routines. Be sure to read this section before any others in this chapter.

The Window Type Device Interface

Obviously, before a program can call window manipulation routines to change a window's attributes, the window must exist. There are three ways to create a window, once the window system is running:

1. Via the system pop-up menu, as described in the *HP Windows/9000 User's Manual*;
2. Via window system commands *wsh(1)* and *wcreate(1), as described in the HP Windows/9000 User's Manual*;
3. Via the *wcreate_term0(3W)* and *wcreate_graphics(3W)* routines described in the "Creating a Term0 Window" and "Creating a Graphics Window" sections of the "Window Management" chapter.

Once a window is created, its window type device interface will exist in the $WMDIR directory. The path name of the window type device interface is $WMDIR/*wname*, where *wname* is the window's name.

Program Structure

Any program that calls window manipulation routines must conform to the following structure:

1. First, the program must **start communication with the window**.
2. After starting communication with the window, the program can then call other window manipulation routines.
3. When a program is finished calling window manipulation routines on a window, the program must finally **stop communication with the window.**

If you start communication with a window, be sure to stop it. Stopping window communication before exiting will ensure the proper execution of the window system and your programs.

Starting and stopping window communication is described in detail in the following subsections.

Starting Window Communication

Once a window is created, its window type device interface exists, but a program can do nothing with the window until communication between the program and the window is started. After starting communication with the window, a program can call other library routines to manipulate the window. For instance, routines can be called to make the window visible, to move it on the screen, or to attach the keyboard to it.

Starting window communication entails the following steps:

1. Build the path name of the window's window type device interface.
2. Open the window's window type device interface.
3. Call *winit(3W)*.

Build the Path Name of the Window's Window Type Device Interface

Remember, the path name of a window's window type device interface is $WMDIR/*wname*, where *wname* is the window's name. You can use the *wmpathmake(3W)* routine to build the path name from the WMDIR environment variable and the window's name.

Open the Window's Window Type Device Interface

For graphics windows, open the window's window type device interface using the Starbase *gopen(3G)* routine; for term0 windows, use the *open(2)* system call. The file descriptor returned from opening the window type device interface is required by window manipulation routines.

Note that *gopen* requires the name of the device driver for the physical display. This value is determined by the WMDRIVER environment variable. To get this value for *gopen*, simply use the *wminquire(3W)* routine, which gets the value of a window system environment variable. You can then supply this value to *gopen*. (For details on using *wminquire*, see the example in this section.)

Call winit(3W)

After opening and obtaining a file descriptor for the window type device interface, call *winit(3W)*; its syntax is:

```
winit(fd)
```

The *fd* parameter is the file descriptor returned from opening the window's window type device interface.

At this point, communication with the window has been started, and a program can call window manipulation routines that alter some of the window's attributes.

Examples

The following function, *est_t0.c*, starts communication with a term0 window and returns its file descriptor. It requires the **path** name of the window's window type device interface as a parameter. It returns the window's file descriptor if successful; otherwise, it returns -1. The function's source can be found in the *man_examples* directory:

```
#include <window.h>     /* window library definitions */
#include <fcntl.h>      /* system call i/o defintions */
est_t0(wt_path)
char *wt_path;          /* name of the window     */
{
        int wfd;        /* window file descriptor */
/*
 * START WINDOW COMMUNICATION:
 *
 *      return -1 if an error occurs
 *      return wfd if successful
 *

 * STEP 1:  Build the path name of the window's window type device interface:
 *          [Note:  The path name is determined BEFORE this routine
 *                  is called; the path name is passed as a parameter.]
 *
 * STEP 2:  Open the window type device interface for reading and writing:
 */
        if ((wfd = open(wt_path, O_RDWR)) == -1) return(-1);

/*
 * STEP 3:  Call winit:
 */
        if (winit(wfd) < 0)
                return(-1);
        else
                return(wfd);
}
```

The next function, *est_gr.c*, starts communication with a graphics window and returns its file descriptor. It requires the window manager's file descriptor and the graphics window's **path** name as a parameter. Its source is found in *man_examples*:

```
#include <starbase.c.h> /* starbase library definitions */
#include <window.h>     /* window library definitions   */
est_gr(wmfd, wt_path)
int     wmfd;           /* wm file descriptor           */
char    *wt_path;       /* graphics window's name       */
{
    int  wfd;                   /* window file descriptor      */
    char dr_path[WINNAMEMAX];   /* screen driver name variable */

/*
 * START WINDOW COMMUNICATION:
 *
 *      return  -1 if an error occurs
 *      return wfd if successful
 *
 * STEP 1:  Build the path name of the window's window type device interface:
 *          [NOTE:  The path name is determined before this routine is
 *                  called; the path name is supplied as the parameter.]
 *

 * STEP 2:  Get the physical screen device driver's name, and open the
 *          device interface (for output only) via Starbase gopen:
 */
        if (wminquire(wmfd, "WMDRIVER", dr_path) < 0) return(-1);

        if ((wfd = gopen(wt_path, OUTDEV, dr_path, INIT)) < 0) return(-1);

/*
 * STEP 3:  Initialize the window type device interface:
 */
        if (winit(wfd) < 0)
                return(-1);
        else
                return(wfd);
}
```

Stopping Window Communication

When a program is finished with a term0 or graphics window, communication with the window should be stopped. Stopping window communication does not remove the window from the window system; it merely releases resources allocated at the time communication was started. The window still exists afterward but is under the control of the window manager.

Stopping window communication involves the following steps:

1. Call *wterminate(3W)*.
2. Close the window's window type device interface.

Call wterminate

Call the *wterminate(3W)* routine; its syntax is:

```
wterminate(fd)
```

It requires the device interface's file descriptor—i.e., the descriptor returned from starting communication with the window.

Close the Window's Device Interface

Use the *close(2)* routine on **term0** windows; use the Starbase *gclose(3G)* routine for **graphics** windows.

Examples

The following function stops communication with any graphics window, given the file descriptor of the window's window type device interface. The function is named *term_gr.c* and is found in the *man_examples* directory.

```
#include <starbase.c.h> /* contains starbase library definitions */
#include <window.h>     /* window library definitions           */
term_gr(wfd)
int     wfd;            /* window type descriptor               */
{
/*
 * STOP GRAPHICS WINDOW COMMUNICATION:
 *
 *      return  0 if successful
 *      return -1 if error occurs
 *
 * STEP 1:  Call wterminate:
 */
    if (wterminate(wfd) < 0) return(-1);
/*
 * STEP 2:  Close the window type device interface.
 */
    if (gclose(wfd) < 0)
        return(-1);
    else
        return(0);
}
```

The next function is similar to the previous, except that it stops communication with any **term0** window. The source is named *term_t0.c* and is found in the *man_examples* directory.

```
#include <fcntl.h>      /* system call i/o definitions */
#include <window.h>     /* window library definitions  */
term_t0(wfd)
int     wfd;            /* window type descriptor      */
{
/*
 * STOP TERMO WINDOW COMMUNICATION:
 *
 *      return  0 if successful
 *      return -1 if error occurs
 *
 * STEP 1:  Call wterminate:
 */
    if (wterminate(wfd) < 0) return(-1);
/*
 * STEP 2:  Close the window type device interface.
 */
    if (close(wfd) < 0)
        return(-1);
    else
        return(0);
}
```

Selecting a Window

The keyboard and mouse or graphics tablet can be attached to one window at a time. Once these input devices are attached to a window, a process can read keyboard information and detect activity (events) in the mouse and/or graphics tablet.

If you wish to read keyboard data from or perform event detection with a window (as described in the "Event Detection" chapter), attach the keyboard and other input devices to the window. The *wselect(3W)* routine attaches and detaches the keyboard and optional mouse or graphics tablet.

Procedure

To attach input devices to a window, simply call *wselect*; its syntax is:

```
wselect(fd, value)
```

The *value* parameter determines whether to attach the window (*value* = SETSELECT), detach the window (*value* = SETNOSELECT), or inquire on select status (*value* = GETSELECT).

Related Routines

The *wautoselect(3W)* routine can also be used to attach input devices to a window. However *wautoselect* works differently than *wselect*: *wautoselect* automatically attaches the input devices to a window when output is sent to the window. The syntax for this routine is:

```
wautoselect(fd, value)
```

If *value* is SETAUTOSELECT, then the window will automatically become the selected window when output is sent to its device interface; if *value* is SETNOAUTOSELECT (the default when the window is created), then the window won't be selected. If *value* is GETAUTOSELECT, then the current auto-selection status is returned.

NOTE: A window's auto-selection status is automatically set to SETNOAUTOSELECT whenever output is sent to its device interface. Therefore, whenever a window becomes deselected, you must re-call this routine if you want the window to again become selected when output is sent to its device interace.

Precautions

Detaching the input devices from a window causes them to be attached to the topmost window in the display stack, unless the detached window was already topmost. In that case, the keyboard is attached to the next window down in the display stack.

If only one window exists, the keyboard cannot be detached from it; attempting to do so won't work.

Example

The following function, named *toggle_sel.c*, inquires on whether the input devices are attached to a specified window. If the window is already selected, then the input devices are detached from the window; otherwise, the window is made the selected window.

```
#include <window.h>     /* window library definitions */
toggle_sel(wfd)
int wfd;                /* window's file descriptor   */
{
    int select_state;   /* current select state variable */


/*
 * TOGGLE A WINDOW'S SELECT STATE:
 *
 *      return -1 if an error occurs
 *      return resulting select state, if successful
 *
 * Check the current state:
 */
    if ((select_state = wselect(wfd, GETSELECT)) < 0) return(-1);
/*
 * Toggle the state:
 */
    if (select_state == SETSELECT)
    {
        if (wselect(wfd, SETNOSELECT) < 0)
           return(-1);
        else
           return(SETNOSELECT);
    }
    else
    {
        if (wselect(wfd, SETSELECT) < 0)
           return(-1);
        else
           return(SETSELECT);
    }
}
```

Displaying and Concealing a Window

At any time after a window is created, it can be normal or iconic. In either of these representations, a window can either be **displayed**—capable of being seen on the screen—or **concealed**—i.e., made invisible. This section discusses how to display and conceal windows in a normal state.

Changing a window from a visible to an iconic representation (and vice versa) is discussed in the chapter "Icons." Keep in mind that the routines discussed here can also be used to display or conceal icons.

Procedure

Following are separate discussions for displaying a window and concealing a window.

Displaying a Window

By default when a window is created via window library routines, it is concealed. Three routines can be used to make windows visible:

- *wtop(3W)*—displays a window as the top window in the display stack. No portion of the window will be occluded by any others. However, depending on the window's location and size, all or part of the window may be off screen. The syntax for this routine is:

  `wtop(`*fd, value*`)`

  The *value* parameter determines the action of the routine: if *value* is `SETTOP`, then the window (specified by *fd*) is made the top window in the display stack; if *value* is `GETTOP`, then the routine returns a value of `SETTOP` if the window is the top window in the stack, `SETNOTOP` otherwise.

- *wbottom(3W)*—displays a window as the bottom window in the display stack. All or part of the window may be occluded by other windows. In addition, all or part of the window may be off screen. This routine's syntax is:

  `wbottom(`*fd, value*`)`

  If *value* is `SETBOTTOM`, then the window is made visible as the bottom window in the display stack; if *value* is `GETBOTTOM`, then *wbottom* returns a value of `SETBOTTOM` if the window is the bottom one, `SETNOBOTTOM` otherwise.

- *wautotop(3W)*—causes a term0 window to be displayed as the top window in the stack when output is sent to the window's device interface. This routine is useful to applications that display urgent information in a term0 window and want the window to be visible when the urgent information is displayed.

For example, suppose that you've written an application for a nuclear power plant, and you have a meltdown emergency window. You probably want this window to be displayed should a meltdown occur. You can use *wautotop* to have the window automatically come to the top if the meltdown message is ever displayed.

The syntax for this routine is:

```
wautotop(fd, value)
```

If *value* is SETAUTOTOP, then the window specified by *fd* will automatically come to the top when output is written to its device interface. If *value* is SETNOAUTOTOP then the window won't automatically come to the top. If *value* is GETAUTOTOP, then the routine returns the current autotop state.

NOTE: The window's autotop state will be automatically set to SETNOAUTOTOP whenever output is sent to the window. Therefore if you must re-call *wautotop* after any data is written to the window, if you want the window to come to the top when the next message is written.

Concealing a Window

After a window is made normal, it can be concealed. You might conceal a window if you don't want to destroy it (or its associated programs) but do want to remove it temporarily from the display screen. The window can then be made normal again (via *wtop* or *wbottom*) when needed. Windows are concealed via *wconceal(3W)*; its syntax is:

```
wconceal(fd, value)
```

If *value* is SETCONCEAL, then the window is concealed; if *value* is GETCONCEAL, then *wconceal* returns a value of SETCONCEAL if the window is concealed, NOSETCONCEAL otherwise.

Precautions

Remember that displaying a window via *wtop* or *wbottom* does **not** ensure that the window will be visible on the screen: the window may be displayed totally or partly off the screen. In addition, if the window is made displayable by *wbottom* the window may be occluded by other windows.

Example

The following program requires that a window named *t0win* exist. You can create this window (if it doesn't already exist) by typing:

```
wsh t0win [Return]
```

The following program toggles the window from displayed to concealed representation, waits about five seconds, and makes the window visible again. The program is found in the *man_examples* directory and is named *conceal_t0.c*.

```
#include        <window.h>
#include        <stdio.h>
main()
{
        int     wfd;                    /* window file descriptor */
        char    wt_path[WINNAMEMAX];    /* window path name       */


/*
 * Build the window's path name and start window communication by
 *      using the est_t0 routine defined in the previous chapter.
 */
        wmpathmake("WMDIR", "t0win", wt_path);
        if ((wfd = est_t0(wt_path)) < 0)
        {
                printf("est_t0 failed - path is %s\n", wt_path);
                exit(1);
        }

/*
 * CONCEAL the window:
 */
        wconceal(wfd, SETCONCEAL);

/*
 * Wait for approximately five seconds:
 */
        sleep( 5 );
/*
 * DISPLAY the window as the top window in the stack:
 */
        wtop(wfd, SETTOP);

/*
 * Stop communication with the window:
 */
        if (term_t0(wfd) < 0)
        {
                printf("term_t0 failed - wfd is %d\n", wfd);
                exit(1);
        }
        exit(0);        /* NORMAL TERMINATION */
}
```

Moving a Window

A window's **location** attribute determines the position (in *x,y* pixel coordinates) of the window's anchor point on the physical display. (The anchor point is the upper-left corner of the window's user area.) Location *0,0* is the upper-left corner of the display—i.e., the origin. The *x* coordinates increase to the right; *y* coordinates increase downward. The *wmove(3W)* routine allows you to change a window's location.

Note that moving a window does **not** change its position in the display stack.

Procedure

To move a window, do the following tasks:

Compute the New Coordinates

First, compute the new coordinates for the window. The new coordinates can be either absolute, relative, or default:

- To compute **absolute** coordinates, simply determine the exact *x,y* coordinates to which you wish the window to be moved. Then supply these as parameters to the *wmove* routine.

  To determine if moving the window will cause part of it to appear off screen, call the *wgetscreen(3W)* routine, which returns the maximum *x,y* coordinates on the physical display—i.e., the coordinates of the lower-rightmost pixel on the display. Coordinates less than *0,0* or greater than the screen size will cause part of the window to appear off screen.

- To compute new coordinates **relative** to a window (which could be the window itself), use the *wgetcoords* routine. Then compute the relative coordinates as offsets of those returned by *wgetcoords(3W)*.

- You can also specify **default** coordinates returned from *wdfltpos(3W)* which returns coordinates for the next window to be created by the window manager. However, this routine is normally used only when creating windows.

Call wmove

Call the *wmove* routine with the desired coordinates; its syntax is:

```
wmove(fd, x,y)
```

The *x,y* parameters specify the new location for the window denoted by *fd*.

Precautions

Keep in mind that when you move a window, all or part of the window may be off screen and, therefore, not visible on the display.

Example

The following program requires that a window named *my_win* exists on the display screen. To create this window, simply type:

```
wsh my_win Return
```

On each invocation of the following program, *my_win* will move 50 pixels down and 60 pixels to the right on the display screen. Before moving the window, the program checks, via *wgetscreen* and *wgetcoords*, to see if moving the window will cause part of it to appear off screen. If so, the window will instead be moved to absolute location *0,0* so that the stair-step movement can proceed again from the upper-left corner of the display. This program is stored in the *man_examples* directory and is named *stair_step.c*.

```
#include <fcntl.h>                  /* system i/O call definitions */
#include <window.h>                 /* window library definitions */

#define stepx 60                    /* pixel step in x direction */
#define stepy 50                    /* pixel step in y direction */

main()
{
    int wmfd;                       /* window manager file descriptor */
    int screenw, screenh;           /* screen width and height */
    int bytepp, cmapent, sfkh;      /* bytesper pixel, color map entries,
                                       and softkey height for the screen */
    int wfd;                        /* window file descriptor */
    char wt_path[WINNAMEMAX];       /* path name for window type */
    int bx, by, bw, bh;             /* dimensions of the border */
    int x,y, w,h, dx,dy, rw,rh;     /* dimensions of the window */
    int est_wm_com();               /* routine to start wm communication */
    int term_wm_com();              /* routine to stop wm communication */
```

```
/*
 * Start window manager communication.
 */
        if ((wmfd = est_wm_com()) == -1)
        {
                perror("est_wm_com failed");
                exit(1);
        }

/*
 * Create a path name for the window.
 */
        wmpathmake("WMDIR", "my_win", wt_path);
/*
 * Open the window and initialize it.
 */
        wfd = open(wt_path, O_RDWR);
        if (wfd <0)
        {
                perror("open of window failed");
                exit(1);
        }
        if (winit(wfd) < 0)
        {
                perror("winit of window failed");
                exit(1);
        }

/*
 * Get the screen size.
 */
        if (wgetscreen(wmfd, &screenw, &screenh, &bytepp,
                           &cmapent, &sfkh) < 0)
        {
                perror("wgetscreen to wmfd failed");
                exit(1);
        }
```

```
/*
 * Get the border size and the contents size of the window.
 */
        if (wgetbcoords(wfd, &bx, &by, &bw, &bh) < 0)
        {
                perror("wgetbcoords of window failed");
                exit(1);
        }
        if (wgetcoords(wfd, &x,&y, &w,&h, &dx,&dy, &rw,&rh) < 0)
        {
                perror("wgetcoords of window failed");
                exit(1);
        }

/*
 * Compute the new position of the window by adding stepx pixels to the
 * x value and stepy pixels to the y value.  If any part of the window
 * will move off the screen, then reposition the window to 0,0.
 */
        if (((bx + bw + stepx) >= screenw) || ((by + bh + stepy) >= screenh))
        {
                x -= bx;
                y -= by;
        } else
        {
                x += stepx;
                y += stepy;
        }
        if (wmove(wfd, x, y) < 0)
        {
                perror("wmove of window failed");
                exit(1);
        }

/*
 * Close the window.
 */
        if (close(wfd) < 0)
        {
                perror("close of window failed");
                exit(1);
        }
/*
 * Stop window manager communication.
 */
        term_wm_com(wmfd);
}
```

Changing a Graphics Window's Size

Each graphics window has a size attribute that represents the pixel width and height of the window; *wsize(3W)* changes a window's size.

Maximum Window Size

The maximum size for a graphics window is its raster size (determined by the *rasterw* and *rasterh* parameters to *wcreate_graphics*) and pan position (set by the *wpan* routine). The window cannot be wider than the raster width minus the *x*-coordinate of the pan position; it can be no taller than the raster height minus the *y*-coordinate pan position.

Minimum Window Size

The window's minimum possible width and height depend on whether the window has a normal border (window border displayed) or thin border (border not displayed):

- With a normal border, the window has a minimum size determined by factors such as the current border font size and the location of the interactive manipulation areas (which must all remain visible).
- With a thin border, the minimum width and height is one pixel by one pixel.

Note: Attempting to set a window's size less than the minimum will cause the window to be redrawn to the minimum size; setting the size larger than the maximum will cause the window to be redrawn to its maximum size.

Procedure

Changing a window's size involves the following tasks:

Optionally Get Window and Screen Size Information

Before changing a window's size, you may want to make sure that changing its size won't make it appear off screen. The following routines can be used with *wsize*:

- *wgetcoords(3W)*—gets information about the window's user unit. This information includes the window's anchor point, current width and height, offset into the virtual raster, and maximum width and height (as specified when the window was created).
- *wgetbcoords(3W)*—gets information about the window's border unit. Specifically, it returns the *x,y* location of the upper-left corner of the window's border. It also returns the width and height of the window's border.
- *wgetscreen(3W)*—returns the pixel width and height of the display screen.

Call wsize

Call *wsize* with the new width and height parameters; its syntax is:

```
wsize(fd, w,h)
```

The window's new width and height are given by the *w,h* parameters.

Precautions

Keep in mind that making a window larger may cause parts of it to be occluded by other windows or the edge of the screen.

Example

The following program requires that a graphics window named *grwin* exist on the display screen. To create this window (if it doesn't already exist), simply type:

```
wcreate -wgraphics -l400,100 -s100,150 -r800,150 grwin [Return]
```

The window is created at location 400,100; its initial size is 100 pixels wide by 150 pixels high; its virtual raster is 800 pixels wide by 150 pixels high. **If you want the following program to work as stated, then do not move or manipulate the window created above.**

On invoking the following program, the window will stretch to the right edge of the display screen. This program is stored in the *man_examples* directory and is named *stretch_gr.c.*

```
#include <fcntl.h>              /* system i/O call definitions */
#include <window.h>             /* window library definitions */
main()
{
    int wmfd;                   /* window manager file descriptor */
    int screenw, screenh;       /* screen width and height */
    int bytepp, cmapent, sfkh;  /* bytesper pixel, color map entries,
                                   and softkey height for the screen */
    int wfd;                    /* window file descriptor */
    char wt_path[WINNAMEMAX];   /* path name for window type */
    int bx, by, bw, bh;         /* dimensions of the border */
    int x,y, w,h, dx,dy, rw,rh; /* dimensions of the window */
    int est_wm_com();           /* routine to start wm communication */
    int term_wm_com();          /* routine to stop wm communication */
```

```
/*
 * Start window manager communication.
 */
        if ((wmfd = est_wm_com()) == -1)
        {
                perror("est_wm_com failed");
                exit(1);
        }

/*
 * Create a path name for the window.
 */
        wmpathmake("WMDIR", "grwin", wt_path);
/*
 * Open the window and initialize it.
 */
        wfd = open(wt_path, O_RDWR);
        if (wfd <0)
        {
                perror("open of window failed");
                exit(1);
        }
        if (winit(wfd) < 0)
        {
                perror("winit of window failed");
                exit(1);
        }

/*
 * Get the screen size.
 */
        if (wgetscreen(wmfd, &screenw, &screenh, &bytepp,
                             &cmapent, &sfkh) < 0)
        {
                perror("wgetscreen to wmfd failed");
                exit(1);
        }
```

```
/*
 * Get the border size and the contents size of the window.
 */
        if (wgetbcoords(wfd, &bx, &by, &bw, &bh) < 0)
        {
                perror("wgetbcoords of window failed");
                exit(1);
        }
        if (wgetcoords(wfd, &x,&y, &w,&h, &dx,&dy, &rw,&rh) < 0)
        {
                perror("wgetcoords of window failed");
                exit(1);
        }

/*
 * Compute the new size of the window so that the right edge of the border
 * is flush with the right edge of the display screen and then change the
 * size of the window.
 */
        w += screenw - bx - bw;
        if (wsize(wfd, w, h) < 0)
        {
                perror("wsize of window failed");
                exit(1);
        }

/*
 * Close the window.
 */
        if (close(wfd) < 0)
        {
                perror("close of window failed");
                exit(1);
        }
/*
 * Stop window manager communication.
 */
        term_wm_com(wmfd);
}
```

Changing a Term0 Window's Size

Like graphics windows, term0 windows have a current size (pixel width and height) attribute. The *wsize(3W)* routine—the same one used to change the size of graphics windows—changes the window's size to the specified pixel width and height. However, it works slightly differently with term0 windows.

If you specify a new pixel width/height that doesn't fall on a character boundary, then the window will be redrawn to the outermost edge of the character boundary. For example, if you specified a width and height that sliced through the middle of column 15 and row 12, then the window would actually be redrawn to touch the right edge of column 15 and the bottom edge of row 12—i.e., slightly larger than the width and height that you specified.

You can see this phenomenon when you interactively change the size of a term0 window. Try to change the size of a term0 window to the middle of some column in the window; the window is always drawn to the right edge of the column.

Maximum Window Size

The maximum window size for a term0 window depends on two factors:

1. The maximum number of columns and rows in the window. The *scrncols* and *scrnrows* parameters to *wcreate_term0* specify the maximum columns and rows when the window is created.
2. The second factor is the size of fonts being used in the window's user (contents) area. All fonts displayed in the window's user area at a given time have the same pixel width and height. The current font size can be determined via the *fontsize_term0* routine.

With these two factors in mind, the maximum pixel width and height of a window are computed as:

$$max\_width = scrncols \times fontsize\_width$$
$$max\_height = scrnrows \times fontsize\_height$$

Minimum Size

The window's minimum size depends on whether the window has a normal or thin border:

- With a normal border, the window has a minimum size determined by factors such as the current border font size and the location of the interactive manipulation areas (which must all remain visible).
- With a **thin** border, the minimum pixel width and height is that of the current font size. That is, the window can be shrunk so that only one character is displayed in the user area.

Attempting to set a window's size less than the minimum will cause the window to be drawn to the minimum size; setting its size larger than the maximum will cause the window to be redrawn to its maximum size.

Procedure

The following tasks should be done to change a window's size:

Calculate the New Pixel Width and Height

The first thing you must do is calculate the new pixel width and height of the window. The *wsize* routine requires that width and height be specified in pixels. The formulas for converting from columns and rows to pixel width and height are:

$$pixel\_x = (col \times current\_font\_width) - 1$$
$$pixel\_y = (row \times current\_font\_height) - 1$$

Fortunately, there is a routine you can use instead of calculating these values every time you want to change a window's size. The *toxy_term0(3W)* routine converts *column* and *row* coordinates to *x,y* coordinates. Note that the returned coordinates are the zero-based coordinates of the upper-left corner of the character positioned at *column,row.*

Optionally Determine Window and Screen Size Information

Before changing a window's size, you may want to make sure that changing its size won't make it appear off screen. You may also want to ensure that you don't make the window larger than its maximum size. The following routines can be used with *wsize*:

- *wgetcoords(3W)*—gets the window's location and current size in pixel units.
- *wgetbcoords(3W)*—gets information about the window's border unit. Specifically, it returns the *x,y* location of the upper-left corner of the window's border. It also returns the width and height of the window's border.

- *wgetscreen(3W)*—returns the pixel width and height (in pixels) of the display screen.

Call wsize

Call *wsize* with the new width and height parameters; its syntax is:

```
wsize(fd, w,h)
```

The window's new width and height are given by the *w, h* parameters.

Precautions

Keep in mind that making a window larger may cause parts of it to be occluded by other windows or the edge of the screen.

Example

The following program requires that a term0 window name *t0win* exist on the display screen. To create this window (if it doesn't already exist), simply type:

```
wsh t0win Return
```

The initial width and height of the window is 80 columns by 24 rows. The following program will shrink the window to one fourth its original size (40 columns by 12 rows) and will move the window to be flush with the lower-right corner of the display screen. The program is stored in the *man_examples* directory and is named *shrink_t0.c*.

```
#include <fcntl.h>              /* system i/O call definitions */
#include <window.h>             /* window library definitions */
main()
{
    int wmfd;                   /* window manager file descriptor */
    int screenw, screenh;       /* screen width and height */
    int bytepp, cmapent, sfkh;  /* bytesper pixel, color map entries,
                                   and softkey height for the screen */
    int wfd;                    /* window file descriptor */
    char wt_path[WINNAMEMAX];   /* path name for window type */
    int bx, by, bw, bh;         /* dimensions of the border */
    int x,y, w,h, dx,dy, rw,rh; /* dimensions of the window */
    int est_wm_com();           /* routine to start wm communication */
    int term_wm_com();          /* routine to stop wm communication */
```

```
/*
 * Start window manager communication.
 */
        if ((wmfd = est_wm_com()) == -1)
        {
                perror("est_wm_com failed");
                exit(1);
        }

/*
 * Create a path name for the window.
 */
        wmpathmake("WMDIR", "t0win", wt_path);
/*
 * Open the window and initialize it.
 */
        wfd = open(wt_path, O_RDWR);
        if (wfd <0)
        {
                perror("open of window failed");
                exit(1);
        }
        if (winit(wfd) < 0)
        {
                perror("winit of window failed");
                exit(1);
        }

/*
 * Shrink the window to one fourth of its size.
 */
        if (toxy_term0(wfd, &w, &h, 40, 12) < 0)
        {
                perror("toxy_term0 to wfd failed");
                exit(1);
        }
        if (wsize(wfd, w, h) < 0)
        {
                perror("wsize of window failed");
                exit(1);
        }
```

```
/*
 * Get the screen size.
 */
        if (wgetscreen(wmfd, &screenw, &screenh, &bytepp,
                             &cmapent, &sfkh) < 0)
        {
                perror("wgetscreen to wmfd failed");
                exit(1);
        }

/*
 * Get the border size and the contents size of the window.
 */
        if (wgetbcoords(wfd, &bx, &by, &bw, &bh) < 0)
        {
                perror("wgetbcoords of window failed");
                exit(1);
        }
        if (wgetcoords(wfd, &x,&y, &w,&h, &dx,&dy, &rw,&rh) < 0)
        {
                perror("wgetcoords of window failed");
                exit(1);
        }

/*
 * Compute the new position of the window so that it is flush with the lower
 * right hand corner of the display screen.
 */
        x += screenw - bx - bw;
        y += screenh - by - bh;
        if (wmove(wfd, x, y) < 0)
        {
                perror("wmove of window failed");
                exit(1);
        }

/*
 * Close the window.
 */
        if (close(wfd) < 0)
        {
                perror("close of window failed");
                exit(1);
        }
/*
 * Stop window manager communication.
 */
        term_wm_com(wmfd);
}
```

Changing a Window's Border

By calling the *wbanner(3W)* routine, a program can change a window's border type to *normal*, *thin*, or *null*, depending on the window's type. Table 4-1 describes the valid border types.

Table 4-1. Window Border Types

Border Type	Description
Normal	Window label and interactive areas present in the window's border; border is "thick".
Thin	No label or manipulation areas are present; border is a thin line surrounding the window's contents area; user can get a pop-up menu by clicking the locator on the thin border.
Null	No border exists whatsoever; the user **cannot** click on the border to get a system menu because there is no border.

Procedure

To change a window's border, call the *wbanner* routine; its syntax is:

```
wbanner(fd, value)
```

For term0 and graphics windows: If *value* is **SETBANNER**, then the window's border is changed to a normal border; if *value* is **SETNOBANNER**, the window's border is changed to a thin border; if *value* is **SETNULLBANNER**, then the window's border is removed.

If *value* is **GETBANNER**, then *wbanner* returns the window's current border type: that is, **SETBANNER** for a normal border, **SETNOBANNER** for a thin border, and **SETNULLBANNER** if the window has no border.

Example

The following function determines a window's border representation. If the window has a normal border, the routine changes it to a thin border; if the window has no border or if the window's border is thin, the routine changes the border to normal.

For example, if this routine is called on a graphics window with no border, its border will then be changed to normal. The source is stored in the *man_examples* directory and is named *wbanner_sub.c*.

```
#include <window.h>      /* window library definitions */
wbanner_sub(wfd)
int wfd;                 /* window's file descriptor   */
{
    int border_state;    /* current border state variable */


/*
 * TOGGLE A WINDOW'S BANNER STATE:
 *
 *     return -1 if an error occurs
 *     return resulting border state, if successful
 *
 * Check the current state:
 */
    if ((border_state = wbanner(wfd, GETBANNER)) < 0) return(-1);


/*
 * Toggle the state:
 */
    if (border_state == SETBANNER)
    {
        if (wbanner(wfd, SETNOBANNER) < 0)
           return(-1);
        else
           return(SETNOBANNER);
    }
    else
    {
        if (wbanner(wfd, SETBANNER) < 0)
           return(-1);
        else
           return(SETBANNER);
    }
}
```

Changing a Window's Label

A window's label is displayed in the window border area, the title of the pop-up menu, the title for the softkey labels, and the label in the icon for the window. By default, a window's label is identical to its name (i.e., the base name of its full path name), specified when the window is created. The *wsetlabel(3W)* routine changes the label to a string other than the window's name.

The maximum number of bytes allowed in a window's label depends on the window's type. A term0 window's label can contain `LABELMAX` bytes (defined in *window.h*). A graphics window's label can contain 128 bytes. The final byte in a window's label must be a `'\0'` (terminating NULL).

Procedure

To set a window's label, call *wsetlabel*; its syntax is:

`wsetlabel(`*fd, label*`)`

The *label* parameter points to a null-terminated string containing the new label to use.

Example

This sample program requires that a graphics window named *grwin* exist. To create the window, simply type:

```
wcreate -wgraphics grwin [Return]
```

Executing the following program will change the window's label from the default `grwin` to `>>NEWLABEL<<`, wait five seconds, and change its label back to `grwin`. The program is found in the *man_examples* directory and is named *setlabel_gr.c*.

```
#include <fcntl.h>              /* system i/O call definitions */
#include <window.h>             /* window library definitions */
main()
{
    int wfd;                    /* window file descriptor */
    char wt_path[WINNAMEMAX];   /* path name for window type */
/*
 * Open the window and initialize it.
 */
        wmpathmake("WMDIR", "grwin", wt_path);
        wfd = open(wt_path, O_RDWR);
        if (wfd <0) {
                perror("open of window failed");
                exit(1);
        }
        if (winit(wfd) < 0) {
                perror("winit of window failed");
                exit(1);
        }

/*
 * Change the window's label and wait 5 seconds.
 */
        if (wsetlabel(wfd, ">>NEWLABEL<<") < 0) {
                perror("wsetlabel of window failed");
                exit(1);
        }
        sleep(5);

/*
 * Change the window's label back.
 */
        if (wsetlabel(wfd, "grwin") < 0) {
                perror("wsetlabel of window failed");
                exit(1);
        }

/*
 * Close the window.
 */
        if (close(wfd) < 0) {
                perror("close of window failed");
                exit(1);
        }
}
```

Setting a Window's Border Colors

Each window has foreground and background colors for its border. By default when a window is created, its foreground border color is black, and its background border color is white (0 is black and 1 is white). (These default values are defined by the window system environment variables WMBDRFGCLR and WMBDRBGCLR.)

Table 4-2 defines the default Starbase color map entries.

Table 4-2. Default Starbase Color Map Entries

Color Index	Default Starbase Color
0	Black
1	White
2	Red
3	Yellow
4	Green
5	Cyan
6	Blue
7	Magenta

Procedure

Determining Border Colors

The *wgetbcolor(3W)* routine returns the current foreground and background colors for a window's border; its syntax is:

`wgetbcolor(`*fd, fgbanner, bgbanner*`)`

The *fgbanner* and *bgbanner* parameters are pointers to integers that will contain the current border foreground and background colors, respectively.

Setting Border Colors

The *wsetbcolor* routine sets the foreground and background colors for a window's border; its syntax is:

`wsetbcolor(`*fd, fgbanner, bgbanner*`)`

The *fgbanner* and *bgbanner* parameters are indices into the system color map that specify the new border foreground and background colors to use.

Example

The following program swaps the border foreground and background colors for any specified window. To use this program, you would type:

```
invert_bc wname [Return]
```

Wname is the name of the window whose border is to be inverted. The above syntax assumes that you have compiled the following source program (named *invert_bc.c*) and have named it *invert_bc*. The program is found in the *man_examples* directory.

```
#include <fcntl.h>                /* system i/O call definitions */
#include <window.h>               /* window library definitions */
main(argc, argv)
int argc;                         /* number of arguments */
char *argv[];                     /* array of arguments */
{
    int wfd;                      /* window file descriptor */
    char *wt_name;                /* name of window */
    char wt_path[WINNAMEMAX];     /* path name for window type */
    int fg_color, bg_color;       /* border colors */

/*
 * INVERT THE BORDER COLORS OF A WINDOW
 *
 * First, check for a window name argument.
 */
        if (argc < 2)
        {
                printf("usage:  invert_bc window_name\n");
                exit(1);
        }

        wt_name = argv[1];
/*
 * Create a path name for the window.
 */
        wmpathmake("WMDIR", wt_name, wt_path);
/*
 * Open the window and initialize it.
 */
        wfd = open(wt_path, O_RDWR);
        if (wfd <0)
        {
                perror("open of window failed");
                exit(1);
        }
        if (winit(wfd) < 0)
        {
                perror("winit of window failed");
                exit(1);
        }

/*
 * Inquire the border foreground and background colors of the window.
 */
        if (wgetbcolor(wfd, &fg_color, &bg_color) < 0)
        {
                perror("wgetbcolor of window failed");
                exit(1);
        }

/*
 * Swap the foreground and background colors of the border.
 */
        if (wsetbcolor(wfd, bg_color, fg_color) < 0)
        {
                perror("wsetbcolor of window failed");
                exit(1);
        }

/*
 * Close the window.
 */
        if (close(wfd) < 0)
        {
                perror("close of window failed");
                exit(1);
        }
}
```

Pausing and Resuming Output to a Term0 Window

Output to a term0 window can be halted and restarted by the *wpauseoutput(3W)* routine. On term0 windows, pausing output via this routine is equivalent to typing an XOFF character on an HP 2622 terminal; resuming output is equivalent to typing an XON character.

Note that pausing a window's output does not halt the process that is creating the output. The output is simply "frozen." However, if a window is paused long enough, the window's buffers will become full and the writing process will be forced to wait until window output is resumed.

Procedure

To pause output to a term0 window, call *wpauseoutput*; its syntax is:

```
wpauseoutput(fd, value)
```

If *value* is SETPAUSE, then the window will be paused; if *value* is GETPAUSE, then a value of SETPAUSE will be returned if the window is currently paused, otherwise SETNOPAUSE is returned.

Example

The following function determines whether output to a window is paused or resumed and toggles the window to the opposite state. For example, if this function is called on a window that is paused, output to the window will resume. The source is stored in the *man_examples* directory and is named *pause_resume.c*.

```
#include <window.h>     /* window library definitions */
pause_resume(wfd)
int wfd;                /* window's file descriptor   */
{
    int pause_state;    /* current pause state variable */
/*
 * TOGGLE A TERMO WINDOW'S OUTPUT PAUSE STATE:
 *
 *      return -1 if an error occurs
 *      return resulting pause state, if successful
 *
 * Check the current state:
 */
    if ((pause_state = wpauseoutput(wfd, GETPAUSE)) < 0) return(-1);


/*
 * Toggle the state:
 */
    if (pause_state == SETPAUSE)
    {
        if (wpauseoutput(wfd, SETNOPAUSE) < 0)
           return(-1);
        else
           return(SETNOPAUSE);
    }
    else
    {
        if (wpauseoutput(wfd, SETPAUSE) < 0)
           return(-1);
        else
           return(SETPAUSE);
    }
}
```

Panning a Graphics Window

In graphics terminology, *panning* changes the view into a graphics window's virtual raster. When the window's size is smaller than the raster, panning a window can be thought of as moving the window over the raster to a specific position in the raster. It has the effect of looking as though the raster moves under the window while the window remains motionless on the display. The *wpan(3W)* routine is used to pan graphics windows.

Window panning is limited by both the current pan position within the virtual raster and the size of the raster. Attempting to pan outside the virtual raster will do nothing.

Note

Term0 windows cannot be panned. Attempting to use *wpan* with term0 windows will cause an error. Note, however, that term0 escape sequences for "rolling" window contents can be used instead. For details, see the "Term0 Windows" chapter in this manual, and the *Term0 Reference Manual.*

Procedure

To pan a graphics window, call the *wpan* routine; its syntax is:

```
wpan(fd, x,y)
```

The *x,y* parameters specify the offset (panning position) into the virtual raster. The upper-left corner of the virtual raster has coordinates *0,0*; the lower-right corner has coordinates one less than those specified by the *rasterw* and *rasterh* parameters used when the window was created via *wcreate_graphics(3W).*

Example

The following function pans any graphics window to be flush with the lower-right corner of its virtual raster. The source is found in the *man_examples* directory and is named *pan_gr.c.*

```
#include <window.h>              /* window system definitions */
pan_gr(wfd)
int wfd;                         /* file descriptor for a window */
{
    int x,y, w,h, dx,dy, rw,rh; /* dimensions of the window */

/*
 * PAN A GRAPHICS WINDOW TO BE FLUSH WITH THE LOWER-RIGHT CORNER OF ITS
 * VIRTUAL RASTER
 *
 * Inquire the size of the window and raster:
 */
        if (wgetcoords(wfd, &x,&y, &w,&h, &dx,&dy, &rw,&rh) < 0)
        {
                perror("wgetcoords of window failed");
                exit(1);
        }

/*
 * Compute the new offset into the raster so that the window is flush with
 * the lower-right corner of the virtual raster and call wpan.
 */
        dx = rw - w;
        dy = rh - h;
        if (wpan(wfd, dx, dy) < 0)
        {
                perror("wpan of window failed");
                exit(1);
        }
        return(0);
}
```

Setting Autodestroy Status

The *wrecover(3W)* and *wautodestroy(3W)* routines control whether a window is automatically destroyed when all processes that have opened the window stop executing.

Procedure

The use of the *wrecover* and *wautodestroy* routines is discussed here.

The wrecover(3W) Routine

The *wrecover* routine controls whether a window is automatically destroyed; its syntax is:

```
wrecover(fd, value)
```

The *fd* parameter is the file descriptor returned from starting communication with the window. The *value* parameter controls the window's **recover** state.

If *value* is SETNORECOVER, then windows are **not** automatically deleted from the system; they must explicitly be removed using the *wdestroy(1)* command, the *wdestroy(3W)* window management routine, or the *Destroy* option of the pop-up menu.

If *value* is SETRECOVER, then windows are automatically destroyed after all process are disassociated with the window. The exact time at which the window is destroyed is determined by the *wautodestroy* routine, described next.

If *value* is GETRECOVER, then the current recover state is returned.

The wautodestroy(3W) Routine

The *wautodestroy* routine works only when the recover state is set to SETRECOVER. When a window has been set to SETRECOVER, the *wautodestroy* routine will control **when** the window is automatically removed from the system. The syntax for this routine is:

```
wautodestroy(fd, value)
```

If *value* is SETAUTODESTROY, then the window is destroyed immediately when all processes have closed the window's device interface.

If *value* is SETNOAUTODESTROY, then the window is destroyed when all processes have closed the window's device interface **followed by** a new window being created.

If *value* is GETAUTODESTROY, then the current auto-destroy status is returned.

Icons 5

By calling icon routines, a window program can:

- change a window's iconic state
- move an icon
- customize icons.

Concepts

This section explains concepts you should understand before using icon routines.

At any time, a window is in one of three states: **concealed**, **normal**, or **iconic**. When in an iconic state, a window is represented by a graphic picture known as an **icon**.

An icon can be thought of as the shrunken form of a window. However, it has no user (contents) area. Instead, it is comprised of two components: the top portion is known as the **picture**, the bottom part is the **label**. Figure 5-1 defines the layout of an icon.

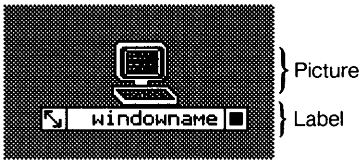


Figure 5-1. Icon Format.

Normally when a window is changed to an icon, both parts—picture and label—are displayed. By using icon routines, a program can suppress either the picture or label. (For details, see "Customizing Icons.")

Additionally, two interactive manipulation symbols appear within the label area:

⤡	moves the icon
■	returns the window to normal representation

See the section "Moving an Icon" for details on moving icons. For information on changing a window to an icon and vice versa, see the section "Changing a Window's Iconic State"

Term0 and graphics windows use default, predefined pictures when an icon is displayed. However, you can create your own iconic pictures. These pictures are stored in files and can be recalled to replace the default icon picture for any term0 or graphics window. See "Customizing Icons" for details on creating user-defined pictures for icons. ,ind picture, icon default

Changing a Window's Iconic State

The *wiconic(3W)* routine changes a window's iconic state. It performs three different functions:

- change a window to an icon
- change an icon to a window
- return the window's iconic state.

Procedure

To change a window to an icon, or vice versa, call *wiconic* with the appropriate parameters; its syntax is:

```
wiconic(fd, value)
```

The *fd* parameter is the file descriptor of the window type device interface for the window whose iconic state you wish to change.

The *value* parameter determines the action taken by *wiconic*:

value	action of wiconic
`SETICONIC`	The window is changed to iconic representation.
`SETNOICONIC`	The window is changed to normal representation.
`GETICONIC`	Return the window's iconic state (`SETNOICONIC` = normal; `SETICONIC` = iconic).

When a window is changed to an icon, the icon appears at the position specified by its **icon location** attribute; likewise, changing an icon to a window causes the window to appear at the screen coordinates specified by the **window location** attribute.

By default when a window is changed to an icon, the icon is placed at the left edge of the screen. Each new icon is placed above the previous one, starting from the bottom of the screen. Therefore, when changing a window to an icon, if you want the icon to appear at some position other than the default, you must change the icon's location before changing it to an icon. (Moving icons is covered in the section "Moving an Icon.")

Similarly, if you want to use a custom picture with an icon instead of the default, customize the icon before changing the window to an icon. This way, only your custom picture will appear when the window is changed to an icon—the default icon won't. (This is covered in the section "Customizing Icons.")

Example

The following function, named *toggle_icon.c*, checks the iconic state of a specified window. It then toggles the window to the opposite state. Because the routine requires the file descriptor of the window's device interface, communication with the window should be started before calling this routine.

```
#include <window.h>      /* window library definitions */
toggle_icon(wfd)
int     wfd;             /* window's file descriptor   */
{
    int iconic_state;    /* current iconic state variable */

/*
 * TOGGLE A WINDOW'S ICONIC STATE:
 *
 *      return -1 if an error occurs
 *      return resulting iconic state, if successful
 *
 * Check the current state:
 */
    if ((iconic_state = wiconic(wfd, GETICONIC)) < 0) return(-1);

/*
 * Toggle the state:
 */
    if (iconic_state == SETICONIC)
    {
        if (wiconic(wfd, SETNOICONIC) < 0)
            return(-1);
        else
            return(SETNOICONIC);
    }
    else
    {
        if (wiconic(wfd, SETICONIC) < 0)
            return(-1);
        else
            return(SETICONIC);
    }
}
```

Moving an Icon

An icon's location attribute determines (in *x,y* pixel coordinates) the position on the screen of the upper-left corner of the icon's picture rectangle. Location *0,0* is the origin—i.e., the upper-leftmost pixel on the screen. *X* coordinates increase to the right; *y* coordinates increase downward. The *wseticonpos(3W)* routine changes an icon's location.

Procedure

Compute New Icon Location

First, compute the new icon coordinates. The new coordinates can be either absolute, relative, or default:

- To compute **absolute** coordinates, simply determine the exact *x,y* coordinates at which you want the icon to appear. Supplying negative coordinates or coordinates greater than the screen size has the same effect as when moving a window. The *wgetscreen(3W)* routine returns information about the size of the screen. (See "Moving a Window" of Chapter 4, "Window Manipulation.")
- To compute coordinates **relative** to an existing icon, use the *wgeticonpos(3W)* routine, which returns the location of an icon. Then compute the relative coordinates as offsets of those returned by *wgeticonpos.*
- The *wdfltpos(3W)* routine returns **default** icon coordinates. These are the default coordinates used when a window is changed to an icon. These coordinates start at the lower-left corner of the display and move upward as more windows are changed to icons.

Call wseticonpos

Call *wseticonpos* with the coordinates computed above; its syntax is:

```
wseticonpos(fd, x,y)
```

The *x,y* parameters specify the location at which the icon will be displayed when the window is in an iconic state.

Example

The following code segment gets an icon's location and moves the icon 30 pixels to the right and 40 pixels down from its previous position.

```
        .
        .
        .
{
    int  wfd;        /* window file descriptor */
    int  ix, iy;     /* icon's x,y location    */

        .
        .
        .
/*
 * Get icon's current location:
 */
    if (wgeticonpos(wfd, &ix, &iy) < 0)
    {
        perror("wgeticonpos wfd");
        exit(1);
    }

/*
 * Set new location relative to previous:
 */
    ix = ix + 30;
    iy = iy + 40;
 /*
  * Set new icon location:
  */
    if (wseticonpos(wfd, ix, iy) < 0)
    {
        perror("wseticonpos wfd");
        exit(1);
    }

        .
        .
        .
```

Customizing Icons

HP Windows/9000 provides you with the capability to define your own custom icons. You can suppress the display of the icon's label area, its picture, or both. In addition, custom pictures can be defined and stored in **icon files**. These files can then be recalled as necessary to replace the standard icon for a particular window. The *wseticon(3W)* routine provides these capabilities.

Procedure

To customize an icon for a given window, call *wseticon*; its syntax is:

`wseticon(`*fd, imode, lmode, iconfile*`)`

Brief descriptions of each parameter follow:

- *fd*—the file descriptor for the window's device interface
- *imode*—controls the icon's picture:
  - if *imode*=`IMODE_NONE`, then the picture will not be displayed with the icon;
  - there are default term0 and graphics pictures for term0 and graphics windows, respectively; if *imode*=`IMODE_TYPE`, then the window's type-dependent picture is displayed with its icon;
  - if *imode*=`IMODE_FILE`, then a user-defined icon picture will be used in place of the default, type-dependent picture.
- *lmode*—controls the icon's label:
  - if *lmode*=`LMODE_NONE`, then do not display the icon's label;
  - if *lmode*=`LMODE_DISP`, then do display the icon's label.
- *iconfile*—is a pointer to the *full* path name of a file containing a custom icon definition.

  Note that unless *imode*=`IMODE_FILE`, this parameter should be null, because it doesn't make sense to specify an icon file unless you want to use a custom icon picture. If *imode* is not set to `IMODE_FILE`, then the *iconfile* parameter is ignored.

Controlling the Display of Picture/Label

Controlling the display of the icon's picture and/or label is easy once you know what values to use for the *imode* and *lmode* parameters. Table 5-1 shows the result of using each possible combination of *imode* and *lmode.*

Table 5-1. imode and lmode combinations

imode	lmode	What Is Displayed
`IMODE_NONE`	`LMODE_NONE`	NOT ALLOWED[1].
`IMODE_NONE`	`LMODE_DISP`	Only the label.
`IMODE_TYPE`	`LMODE_NONE`	Type-dependent picture only.
`IMODE_TYPE`	`LMODE_DISP`	Standard icon.
`IMODE_FILE`	`LMODE_NONE`	Only the customized picture.
`IMODE_FILE`	`LMODE_DISP`	Custom picture with label.

Defining and Using a Non-Standard Picture

As mentioned above, you can design your own icon pictures to be used in place of the type-dependent pictures used by default. Each custom picture is stored in its own file; the file consists of one variable-length record; and the record structure is of the type *iconstruct*, defined in */usr/include/fonticon.h.*

Before going into detail on the format of this file, a discussion of **picture size**, **masks**, and **images** is required.

When the window system displays an icon, the picture is drawn from three entities: *picture size*, *mask*, and *image.* The picture size is simply the pixel width and height of the rectangle in which the picture is drawn; the mask defines the shape of the picture; and the image defines the colors to be used within the shape outlined by the mask.

The following analogy helps in understanding picture size, mask, and image: Suppose you are an artist, and you're commissioned to draw a picture of a computer terminal. The picture is to be drawn on a polka-dotted piece of paper that matches the person's desk top. The paper is 50 centimeters wide by 30 centimeters high—this is the picture size.

[1] This combination of parameters is not allowed; to conceal an icon, you should use the *wconceal(3W)* routine.

After obtaining the paper, your first task is to define the terminal's shape—this is the mask.

Finally, you fill in the mask with the appropriate lines and colors to make it look like a terminal—this is the image. (Note that the person commissioning the drawing is rather eccentric and doesn't want the polka dots outside the image to be covered by the picture. The window system is the same, any area of the icon's picture not defined by the mask is displayed in the desk top pattern.) Figure 5-2 illustrates this process.

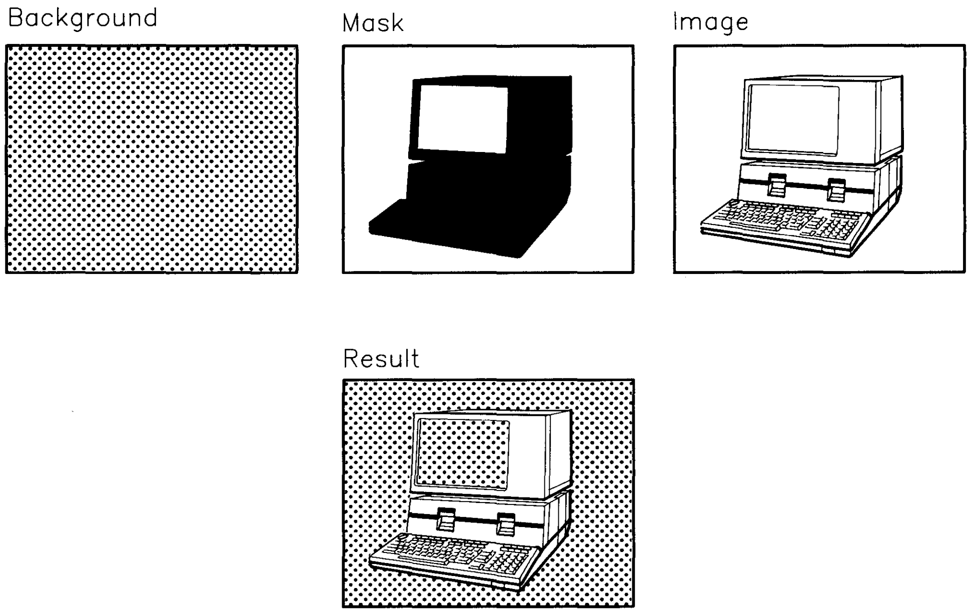


Figure 5-2. Creating a Picture from Mask and Image.

Defining an icon's picture is similar to drawing a picture as described above. The only difference is that you define the picture size, mask, and image in a record structure in a data file, called an **icon file**. Table 5-1 provides a detailed discussion of each field of the *iconstruct* structure defined in *fonticon.h.*

Table 5-2. The *iconstruct* structure.

Item	Description	Range
`magic`	This number flags the file as being an icon file. By default, this value should be set to `FMAGICNUM`, as defined in *fonticon.h*.	`FMAGICNUM`
`header_length`	Defines the offset to the mask and image arrays from the start of the file. This field was included only to allow upward compatibility of HP Windows/9000 if more fields are added to the structure in later versions. It should always be set to the size of the structure. (The C-language compile-time operator **`sizeof`** gives the size in bytes of any data structure).	**`sizeof(struct iconstruct)`**
`filetype`	Defines whether the file is for a font (**`filetype`**=*0*), sprite (**`filetype`**=*1*), or icon (**`filetype`**=*2*).	2
`width`	Image width in pixels.	1 to 127
`height`	Image height in pixels.	1 to 127
`mbytes`	Number of bytes in the pixel mask array that immediately follows this structure.	1 (for a one-pixel picture) to 2032 (for a 127-by-127-pixel picture)
`ibytes`	Number of bytes in the image array that follows the pixel mask array.	**`ibytes`** ≥ 1
`idepth`	Number of bytes per pixel in the image. If you only wish to design a black-and-white picture, then this value should be zero: a zero value specifies that each bit in the image corresponds to a specific pixel in the picture. However, if you wish to design color pictures, then you need one or more bytes for each pixel on the screen. Each byte will hold the color for a corresponding pixel in the picture.	0 $\Rightarrow$ one bit per pixel 1 $\Rightarrow$ one byte per pixel 2 $\Rightarrow$ two bytes per pixel
`hotx` and `hoty`	These fields are used with sprites only.	set to 0 for future expansion
`unused`	Not currently used by the system.	nothing

The Pixel Mask Array
The pixel mask array, `mask[mbytes]`, immediately follows the *iconstruct* record in the picture file. Each bit in the mask can correspond to a pixel in the picture.

Note that each row of pixels is masked, starting at a byte boundary; in addition, masking proceeds left to right through the bits of each mask byte. For example, if you designed a picture with 14 bits per row and 10 rows, the pixel mask would consist of 20 bytes: ten rows at two bytes per row; the most-significant bit of the first byte of each row maps to the leftmost bit of the corresponding row; and the two least-significant bits of the second byte of each row are ignored because only 14 bits are needed for each row. (The example below should help clarify this.)

The Image Array
The image array, `image[ibytes]`, immediately follows the pixel mask array in the picture file. Depending on the value of the `idepth` parameter, each bit, byte, or group of bytes, corresponds to a bit in the pixel mask array. If the bit in the pixel mask is set, then the value in the image array is used to plot the corresponding pixel on the screen; otherwise the background is drawn as the desk top dither pattern.

Remember that if `idepth`=0, then there is a one-to-one relation between the bits in the mask and image arrays. If `idepth`=1, there is a byte-to-bit relation between the image and mask arrays; that is, the first byte in the image array corresponds to the most-significant bit in the first byte of the mask array.

Example
The following example program, *build_icon.c*, defines an icon that is simply a square with a white line through its border; the middle of the square is not masked, so the dither pattern can be seen through it if the icon is displayed.

```
   #include <stdio.h>              /* standard I/O definitions */
   #include <fonticon.h>           /* icon definitions */
   main()
   {
       int icon_file;                      /* file descriptor for the icon file
*/
       struct iconstruct square_icon;      /* icon structure to be defined */
```

```
/*
 * Define the mask for the icon:
 */
   static unsigned char mask[32*32/8] = {
   255,255,255,255, 255,255,255,255, 255,255,255,255, 255,255,255,255,
   255,255,255,255, 248,0,0,31, 248,0,0,31, 248,0,0,31,
   248,0,0,31, 248,0,0,31, 248,0,0,31, 248,0,0,31,
   248,0,0,31, 248,0,0,31, 248,0,0,31, 248,0,0,31,
   248,0,0,31, 248,0,0,31, 248,0,0,31, 248,0,0,31,
   248,0,0,31, 248,0,0,31, 248,0,0,31, 248,0,0,31,
   248,0,0,31, 248,0,0,31, 248,0,0,31, 255,255,255,255,
   255,255,255,255, 255,255,255,255, 255,255,255,255, 255,255,255,255 };
```

```
static unsigned char image[32*32] = {
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,
0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
```

```
/*
 * DEFINE AN ICON CONSISTING OF A SQUARE WITH A WHITE LINE THROUGH ITS
 * BORDER; THE MIDDLE OF THE SQUARE IS NOT MASKED, SO THE BACKGROUND
 * WILL SHOW THROUGH WHEN THE ICON IS DISPLAYED.
 *
 * Initialize the icon structure variables:
 *             magic is FMAGICNUM as defined in fonticon.h.
 *             The length of the header is sizeof(iconstruct).
 *             The file version number is 1.
 *             The file type is icon (2).
 *             The icon size is 32 pixels wide by 32 pixels high.
 */
    square_icon.magic = FMAGICNUM;
    square_icon.header_length = sizeof(square_icon);
    square_icon.version = 1;
    square_icon.filetype = 2;
    square_icon.width = 32;
    square_icon.height = 32;


/*
 * Initialize the image and mask specific variables of the icon structure:
 *             The number of bytes in the mask are 128.
 *             The number of bytes in the image is 1024.
 *             The image contains one byte per pixel.
 */
    square_icon.mbytes = square_icon.width * square_icon.height / 8;
    square_icon.ibytes = square_icon.width * square_icon.height;
    square_icon.idepth = 1;


/*
 * Create and open a file to hold the icon definition, and then
 * write the iconstruct, mask, and image to the file.
 */
    if ((icon_file =
        creat("/tmp/squareicon",0600)) < 0)
    {
        perror("cerat of square_icon failed");
        exit(1);
    }
    write(icon_file, &square_icon, sizeof(square_icon));
    write(icon_file, mask, square_icon.mbytes);
    write(icon_file, image, square_icon.ibytes);
    close(icon_file);
}
```

The next program, named *replace_icon.c*, replaces the icon for any window, specified on the command line, with the icon created above; in addition, the icon's label is not displayed.

```
#include <fcntl.h>               /* system I/O call definitions        */
#include <window.h>              /* window library definitions         */
main(argc, argv)
int   argc;                      /* number of arguments on command line */
char *argv[];                    /* command line argument list         */
{
    int  wmfd;                   /* window manager file descriptor     */
    int  wfd;                    /* window type file descriptor        */
    char wt_path[WINNAMEMAX];    /* path name for window type          */
    int  est_wm_com();           /* routine to start wm communication */
    int  term_wm_com();          /* routine to stop wm communication */

/*
 * START WINDOW MANAGER COMMUNICATION:
 */
    if ((wmfd = est_wm_com()) == -1) {
           perror("est_wm_com");
           exit(1);
    }

/*
 * REPLACE THE WINDOW'S CURRENT ICON WITH THE SQUARE ICON.
 *
 * STEP 1:  Build the path name of the window:
 */
    wmpathmake("WMDIR", argv[1], wt_path);

/*
 * STEP 2:  Open the window and call winit:
 */
    if ((wfd = open(wt_path, O_RDWR)) < 0) {
           perror("open wfd");
           exit(1);
    }
    if (winit(wfd) < 0) {
           perror("winit wfd");
           exit(1);
    }
/*
 * STEP 3:  Set up the new icon and don't display the label:
 */
    if (wseticon(wfd, IMODE_FILE, LMODE_NONE,
        "/tmp/squareicon") < 0) {
            perror("wseticon wfd");
            exit(1);
    }

/*
 * STEP 4:  Stop communication with the window:
 */
    if (wterminate(wfd) < 0) {
            perror("wterminate wfd");
            exit(1);
    }
    if (close(wfd) < 0) {
            perror("close wfd");
            exit(1);
    }

/*
 * STOP WINDOW MANAGER COMMUNICATIONS:
 */
    if (term_wm_com(wmfd) == -1) {
            perror("term_wm_com wmfd");
            exit(1);
    }
    exit(0);
}
```

Event Detection 6

Using HP-UX **signal** capabilities, a process can be signalled (interrupted) when window system **events** occur. Moving a window, changing a window's size, and moving the pointer over a particular area of a window are examples of events. On receiving a signal, a process can execute **event detection** routines to determine what event caused the signal.

This chapter contains many complex concepts, some of which you may already understand, depending on your level of Windows/9000 or HP-UX expertise. The following table briefly describes the three sections in this chapter and how you should read them.

Section	**Contents**	**How to Read**
HP-UX Signals	An overview of HP-UX signals. Describes what signals are and how to use them in programs.	Read this section thoroughly if you know nothing about signals. If you already know how to use signals, then you should at least skim this section.
Events	An overview of events. Describes what events are and lists all the events currently supported.	Read the introduction to this section thoroughly so you can understand basically what events are. Then skim the list of events just to get a feel for the kind of events you can detect.
Event Detection	Describes how to detect events from your programs using event detection routines. This is the main information you will need to write programs that interact with users through the locator devices.	Read this section thoroughly since it contains the most important information in the chapter. As you read this section, you may find it helpful to refer back to the "HP-UX Signals" and "Events" sections.

HP-UX Signals

A **signal** is an interrupt sent to a process. For example, if a program is set up properly, it can be signalled (interrupted) when the Break key is pressed on the keyboard.

On receiving a signal, a process can do one of three things:

- Do a default action, depending on the signal. For example, if the Break key is pressed, terminate the process.
- Ignore the signal, as if it never occurred. For example, if the Break key is pressed, continue executing as if the Break key was never pressed.
- Call a **signal handler**, a routine that determines what caused the signal and does an appropriate action. For example, if the Break key is pressed, the process is interrupted from whatever it was doing and calls the signal hander; the signal handler then asks the user whether he or she really wants to exit the program.

The signal(2) System Call

The *signal(2)* system call allows the calling process to choose which way—default, ignore, or signal handler—to react to the receipt of a specific signal. The *signal* system call has the following syntax:

```
signal(sig, func)
```

The *sig* parameter is an integer code specifying the signal the process may recieve. (Valid signals are defined on the *signal(2) HP-UX Reference* page and in the */usr/include/sys/signal.h* header file.)

The *func* parameter specifies the action the process should take on receiving the signal. *Func* should be assigned one of three values: `SIG_DFL`, `SIG_IGN`, or the address of a signal handler.

If the *signal* call is unsuccessful, a negative value is returned.

SIG_DFL

If *func* is set to `SIG_DFL`, then the signal will cause the process to take the default action as defined in the *signal(2)* page of the *HP-UX Reference.* For example, if a program calls *signal* as

```
signal(SIGINT, SIG_DFL)
```

then the program will terminate if the process receives the SIGINT signal (usually caused by the user pressing the Break key).

SIG_IGN

If *func* is set to `SIG_IGN`, the signal will be ignored by the process. For example, if a program calls *signal* as

```
signal(SIGINT, SIG_IGN)
```

then the program will **not** terminate as usual if it receives the SIGINT signal. Instead, it will continue executing as if nothing happened.

Signal Handler

Finally, the *func* parameter can be set to the address of a *signal handler.* On receiving the signal, the receiving process is to execute the signal-handling routine pointed to by *func.*

To pass the address of a signal handler, simply use the name of the signal handler as the *func* parameter. For example,

```
signal(SIGINT, catch_break)
```

will cause the calling process to execute the user-defined routine `catch_break` if the process receives the SIGINT signal.

Example Signal Handler

The following program sets up a signal handler named **`catch_sigint`** for the SIGINT signal (lines 8–11). The SIGINT signal is defined in the *signal.h* header file (line 1). The *signal* system call returns negative one (`-1`) if it fails, thus the **`if`** statement on line 8.

After calling the *signal* system call, the program waits via the *pause(2)* system call (line 12) till it receives the SIGINT signal, at which time the **`catch_sigint`** signal handler (lines 16–29) is executed.

The **`catch_sigint`** routine simply sleeps for one second (line 21), displays a message asking whether the user wants to exit (lines 22, 23), and reads the user's answer (line 24). If the user answers by entering 'y', then the program exits via the *exit(2)* system call. Otherwise, the routine simply returns to the **`while`** loop (lines 7–13) to do everything again.

```
 1:  #include        <signal.h>
 2:  #include        <stdio.h>
 3:  main()
 4:  {
 5:  int     catch_sigint();
 6:
 7:        while (1) {
 8:                if (signal(SIGINT, catch_sigint) = -1) {
 9:                        fprintf(stderr, "\nsignal(2) failed\n");
10:                        exit(1);
11:                }
12:                pause();
13:        }
14:  } /* END of main() */
15:
16:  catch_sigint( signal )      /* the signal handler */
17:  int     signal;
18:  {
19:  int     ans;
20:
21:        sleep(1);
22:        printf("SIGINT received. Do you wish to exit (y = yes)?");
23:        fflush(stdout);
24:        fflush(stdin); ans = getchar();
25:        if (ans == 'y')
26:                exit(0);
27:        else
28:                return(ans);
29:  }
```

Events

An event is an action that changes the state of the window system. For example, moving a window is an event, as is attaching the keyboard to a different window. The window manager always knows the current state of the window system and, therefore, knows when an event occurs.

Events occur on a per-window basis; that is, each event is sent to one window. Event detection routines, described in the next section "Event Detection," allow a process to be signalled when an event occurs in a specific window.

Button Press Events

The window system supports devices (e.g., mouse, puck) which have **buttons**. A button is simply a switch on an input device. HP Windows/9000 recognizes eight buttons, defined in Table 6-1.

A button press event occurs when a button is pressed (pushed down) or released (raised up from the down position). Thus, there are 16 distinct button press events: button 1 up, button 1 down, button 2 up, button 2 down, ..., button 8 up, button 8 down.

If a button is released, then the button up event goes to the same window to which the button down event was sent. When a button is pressed (button down event), the window manager steps through the following rules to determine where the button down event is sent:

1. If full-screen sprite mode is enabled for a window, then all button press events are sent to that window.
2. If the locator is over a window's user area, then the button press goes to that window.
3. If the button is a system button (e.g., the Select button over the desk top, or over a window's border), then the button press invokes a system pop-up menu.
4. If user-defined menus are activated on the button press, then a user-defined menu pops up.
5. Otherwise, the button press goes to the selected window.

Table 6-1. Supported Buttons

Button	Corresponding Locator Device Button
1	The following are all equivalent to button one: • The Select key on the ITF keyboard • The left mouse button • The leftmost button on the graphics tablet puck • The graphics tablet stylus point.
2	The following are both equivalent to button two: • The right mouse button • The rightmost button on the puck.
3	The topmost button (closest to the cross-hairs) on the graphics tablet puck.
4	The bottom button (furthest from the cross-hairs) on the graphics tablet puck.
5,6,7	Not currently used on locator devices.
8	Indicates stylus or puck **proximity**. That is, the stylus or puck may be touching the graphics tablet. The stylus or puck touching the tablet is defined as button 8 being pressed down. The stylus or puck not touching the tablet is defined as button 8 being up.

Locator Moved

This event is occurs if the locator moves. The event is sent to the selected window only.

Note that enabling this event may significantly degrade system performance because the window manager must constantly track the locator and signal processes that have enabled event detection in the selected window.

Window Moved

This event occurs when a window is moved. The event is sent to the window that moved.

Window's Size Changed

This event occurs when a window's size is changed. The event is sent to the window that changed size.

Window's Selection Status Changed

This event occurs when a window's select status changes. If a different window is selected, two events actually occur:

1. One event occurs for the window that becomes unselected;
2. The other event occurs for the window that becomes selected.

Window Needs Repainting

This event occurs if any part of a non-retained graphics window needs repainting. For example, if a non-retained graphics window is partially occluded by another window, and the other window is moved completely away from the non-retained window, then more of the non-retained window becomes visible; thus, the window repaint event occurs for the non-retained window.

This event also occurs when the user selects the *Repaint* option of the system pop-up menu, or when the *wmrepaint(3W)* routine is called.

This event is helpful for letting an application know when it should repaint a non-retained graphics window.

A Selection Made from User-Defined Menu

This event occurs when the user makes a selection from a user-defined menu for a window. The event is sent to the window from which a pop-up menu selection was made. (See the "Pop-Up Menus" chapter for details on using user-defined pop-up menus in your programs.)

A Hot Spot Was Activated

This event is generated for graphics windows only. **Hot spots** are sensitive rectangles that you can define in a graphics window's raster. Using hot spot routines described in the "Graphics Window Hot Spots" chapter, a program can define the location and size of hot spots, and the sensitivity of the hot spot—e.g., did the pointer move in or out of the hot spot. An event occurs when the user does some action as defined by the hot spot routines.

Window Destroyed

This event is generated when a graphics window is destroyed. Term0 windows do not support this event.

BREAK Key Pressed

This event is generated when the Break key is pressed in a selected graphics window. Term0 windows do not support this event.

Window's Iconic State Changed

This event is generated for graphics window is changed to an icon, or vice versa. Term0 windows do not support this event.

Elevator Moved

This event is generated when a graphics window's elevator bar moves. See the "Graphics Window Scroll Bars and Elevators" chapter for details. Term0 windows do not support this event.

Arrow Activated

This event is generated when an arrow is activated in a graphics window's border. See the "Arrows and Elevators" chapter for details. Term0 windows do not support this event.

Full-Screen Sprite Mode Aborted

This event is generated when the user aborts full-screen sprite mode. See the "Locator and Echo Routines" chapter for details. Term0 windows do not support this event.

Event Detection

At this point, you have the conceptual information you need to perform event detection. That is, you understand signals, and you should know what events can be detected. This section pulls everything together by showing you how to write programs that detect events.

A program that performs event detection consists of the following steps:

1. Start communication with the window in which you wish to detect events.
2. Set up a signal handler to receive SIGWINDOW.
3. Define an **event mask**.
4. Call *wsetsigmask(3W)*.
5. Wait for an event to happen.
6. Receive the signal.
7. Execute the signal handler.
8. Discover which event(s) caused the signal and do the appropriate action.
9. Before the program terminates, stop communication with the window.

Figure 6-1 illustrates this procedure. Each of the steps is described in detail following the diagram.

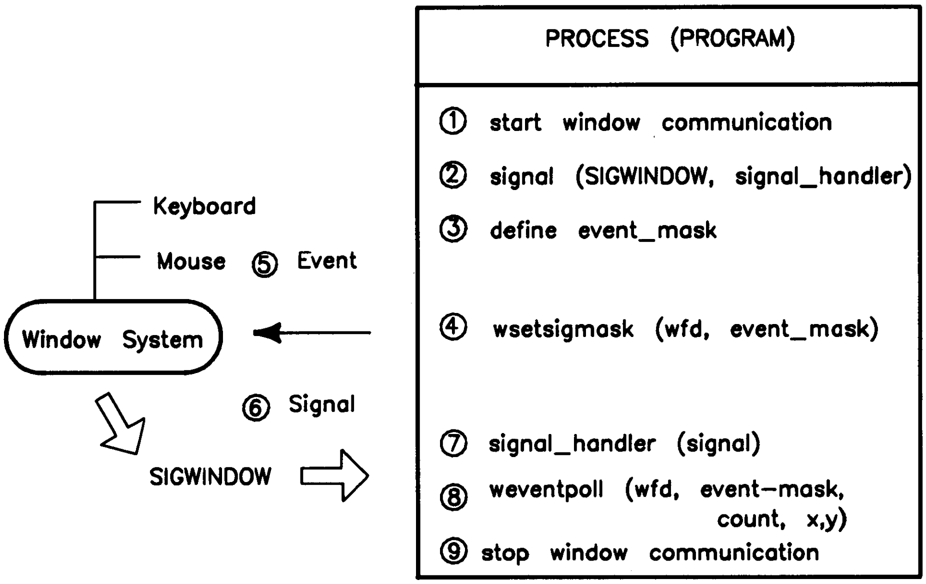

PROCESS (PROGRAM)
① start window communication
② signal (SIGWINDOW, signal_handler)
③ define event_mask
④ wsetsigmask (wfd, event_mask)
⑦ signal_handler (signal)
⑧ weventpoll (wfd, event−mask, count, x,y)
⑨ stop window communication
Keyboard
Mouse
⑤ Event
Window System
⑥ Signal
SIGWINDOW

Figure 6-1. Event Detection.

Step 1: Start Communication with the Window

As mentioned in the "Events" section, events occur per window. To determine which window to detect events in, event detection routines require the file descriptor returned from starting communication with the window. (If you don't remember how to start communication with a window, review the "Concepts" section of the "Window Manipulation" chapter.)

Step 2: Set Up a Signal Handler to Receive SIGWINDOW

The SIGWINDOW signal is sent to a process when an event occurs in the window in which the process has enabled event detection.

The *signal* system call allows a program to receive the SIGWINDOW signal and, upon receiving the signal, execute a signal handler. The signal handler can then determine (via event detection routines) which event(s) caused the signal and take the appropriate action for the event.

When calling *signal*, the *sig* parameter should be set to SIGWINDOW, and the *func* parameter should be the name of the signal handler. The following code example, taken from the example program at the end of this chapter, illustrates how to do this:

```
#include <signal.h>      /* signal definitions */
   :
main(argc, argv)
   :
int     which_event();   /* signal handling procedure */
   :
/*
 * STEP 2:  Set up the signal handler to catch SIGWINDOW:
 */
    if (signal(SIGWINDOW, which_event) < 0)
    {
        perror("signal SIGWINDOW");
        exit(1);
    }
   :
```

Step 3: Define the Event Mask

As mentioned in Step 2, a process receives the SIGWINDOW signal when an event occurs in the window with which event detection has been enabled. The **event mask** allows a program to select which events should cause SIGWINDOW to be sent to a process.

The event mask is a 32-bit word; each bit represents an event. For example, the least-significant bit represents button 1 being pressed down.

For a process to be signalled when an event occurs, the corresponding bit in the event mask must be set. In other words, if you want to check for an event, set its bit in the event mask.

Constant definitions for each bit are found in the */usr/include/window.h* header file. Table 6-2 shows what event each constant represents. You can set the bit for an event you wish to detect by OR-ing its constant with the event mask.

The following code segment, taken from the final example in this chapter, sets the event mask to check for button 1 being pressed, or button 2 being released, or the window being sized:

```
int     event_mask;  /* mask of events to be detected */
   :
/*
 * STEP 3:  Define the Event Mask:
 */
    event_mask = EVENT_B1_DOWN | EVENT_B2_UP | EVENT_SIZE;
```

Table 6-2. Event Constant Definitions

Constant	Event Description
EVENT_B*n*_DOWN	Button number *n* has been pressed down. Note that this does **not** indicate that it has been released after being pressed down. Valid values for *n* are 1 through 8. Table 6-1, found in the "Events" section of this chapter, maps the various buttons to the corresponding buttons on HP-HIL devices.
EVENT_B*n*_UP	Button number *n* has been released. Valid values for *n* correspond to those for EVENT_B*n*_DOWN.
EVENT_ECHO	Means that the pointer (locator position) changed on the screen while the window was selected.
EVENT_MOVE	The window moved.
EVENT_SIZE	The window's size was changed.
EVENT_SELECT	The keyboard attach state changed—the window was selected or unselected.
EVENT_REPAINT	The window needs to be repainted. This event is generated when a non-retained graphics window needs to be repainted by the application controlling the window.
EVENT_MENU	A selection was made from the window's user-defined pop-up menu(s).
EVENT_HOTSPOT	A hotspot was activated. This works with graphics windows only. (See the "Graphics Window Hot Spots" chapter for details on using window hot spots.)
EVENT_DESTROY	The window was destroyed. This works with graphics windows only.
EVENT_BREAK	The Break key was pressed while the window was selected. This works in graphics windows only.
EVENT_ICON	The window's iconic state changed. This works with graphics windows only.
EVENT_ELEVATOR	The window's border elevator moved. This works with graphics windows only. See the "Graphics Window Scroll Bars and Elevators" chapter for details.
EVENT_SB_ARROW	The window's scroll bar arrow was activated. This works with graphics windows only. See the "Graphics Window Scroll Bars and Elevators" chapter for details.
EVENT_ABORT	A full-screen sprite operation was aborted. This works with graphics windows only. See the "Graphics Window Scroll Bars and Elevators" chapter for details.

Step 4: Call wsetsigmask(3W)

The *wsetsigmask(3W)* shows a process's event mask to the window manager so the window manager knows which process to signal if one of the events (specified in the event mask) occurs in a particular window.

The syntax for *wsetsigmask* is:

`wsetsigmask(`*fd, mask*`)`

The *fd* parameter is a file descriptor returned from starting communication with the window in which you wish to detect events.

The *mask* parameter is the event mask defined in Step 3.

Note

A maximum of three processes can call *wsetsigmask* for a given window. In other words, a window can send signals to only three processes.

The following code segment, taken from the example at the end of this chapter, calls *wsetsigmask* using the event mask from Step 3. The `if` statement checks whether *wsetsigmask* returns a negative value; if so, an error occurred. The `wfd` parameter is the file descriptor returned from starting window communication (Step 1).

```
/*
 * STEP 4:  Call wsetsigmask(3W):
 */
    if (wsetsigmask(wfd, event_mask) < 0)
    {
           perror("wsetsigmask wfd");
           exit(1);
    }
```

Step 5: Wait for the Event(s)

After calling *wsetsigmask*, the process must wait for an event to occur. Typically, a program will wait via the *pause(2)* system call, which suspends execution of a process until a signal is received.

The following code segment, from the example program, calls *pause* to wait until the process receives the SIGWINDOW signal.

```
/*
 * STEP 5:  Wait for an event to occur:
 */
    pause();
```

Step 6: Receive the Signal

If the window manager detects an event in a window, it sends the SIGWINDOW signal to all processes that requested the signal via *signal(2)*.

Step 7: Execute the Signal Handler

On receiving the SIGWINDOW signal, the process will execute the signal handler set up in Step 2.

Step 8: Call weventpoll(3W)

To determine which event(s) caused the SIGWINDOW interrupt, use the *weventpoll(3W)* routine. Typically, you would call *weventpoll* in the signal handler. The syntax for this routine is:

`weventpoll(`*fd, mask, count, x,y*`)`

Weventpoll has input and output parameters, discussed next.

Inputs

- *fd*—the integer file descriptor returned from starting communication with the window.
- *mask*—an integer point to an event mask. Note that *mask* is also an output parameter. The output value in *mask* depends on what is sent as input in *mask*.

  If *mask* is set to 0 before calling *weventpoll*, then event information is returned for the most recent event only.

  If *mask* is set to a valid event mask, then information is returned for the event(s) that caused the SIGWINDOW interrupt.

Outputs

- *mask*—a pointer to the resulting event mask, depending on the input value of *mask*.

  If *mask* was passed in as 0, then the bit corresponding to the most recent event is set in *mask*. (Note that `EVENT_ECHO` is never returned as the last event.)

  Weventpoll does not queue events; so you can't use *weventpoll* to determine the order of events. You can, however, determine the order of events that have taken place with graphics windows; the "Graphics Window Input" chapter describes how to do this.

  If *mask* is passed with event bits set, the resulting *mask* is the logical AND of *mask* and the bits corresponding to any events that have occurred since the last call to *weventpoll*. In other words, *weventpoll* checks only for events whose bits are set in *mask*. On return from *weventpoll*, *mask* contains an event mask representing which of the requested events occurred.

  Typically, a signal handler will call *weventpoll* in a loop until *mask* is 0. This way, the signal handler will not miss any events that occur between the time SIGWINDOW is actually generated and the time the signal handler is executed.

- *count*—the number of times the event represented by the most-significant bit set in *mask* has occurred since the last call to *weventpoll*.

  Note that the window system keeps an internal *count* for the number of times each event has occurred. Whenever *count* is returned for a particular event, the internal *count* is reset to zero. This way, the next time *weventpoll* is called for the event, *count* will accurately reflect the number of times the event has occurred since the last time it was polled.

- *x*,*y*—the values are also returned for the event represented by the most-significant bit that is set in *mask*; their values depend on the type of event. Table 6-3 defines the values returned for each event.

Table 6-3. Weventpoll(3W) x,y Values

Event(s)	x,y Return Values
EVENT_B*n*_UP EVENT_B*n*_DOWN EVENT_ECHO	The locator's position when this event occurred.
EVENT_MOVE	The new window location.
EVENT_SIZE	The new width and height of the window.
EVENT_SELECT EVENT_REPAINT EVENT_DESTROY EVENT_BREAK EVENT_ICON EVENT_ABORT	Both are set to zero.
EVENT_MENU	*X* contains the menu id of the menu from which a selection was made; *y* contains the item id of the item selected from the menu. The "User-Defined Menus" chapter describes these topics in detail. (A program must use *winput_read(3W)* to get the cause of a menu item selection; see the "Graphics Window Input" chapter.)
EVENT_HOTSPOT	*X* contains the *event_byte* specified for the hot spot via the *whotspot_create(3W)* or *whotspot_set(3W)* routines; *y* contains the cause of the hot spot activation. (See the "Graphics Window Hot Spots" chapter for details.)
EVENT_ELEVATOR	*X* contains a horizontal or vertical scroll bar indicator; *y* contains the requested value for the elevator. (See the "Graphics Window Scroll Bars and Elevators" chapter for details.)
EVENT_SB_ARROW	*X* contains the sum of all horizontal scroll bar arrow movements: a right arrow movement adds 1 to the sum, and a left arrow movement subtracts 1. Likewise, *y* contains the sum of all vertical scroll bar arrow movements: a down arrow adds 1 to the sum, and an up arrow subtracts 1. (See the "Graphics Window Scroll Bars and Elevators" chapter for details.)

Weventpoll(3W) Example

The following code segment, taken from the example at the end, uses *weventpoll* to determine which event generated the signal. It then prints a message to standard output, describing which event caused the signal.

```
/*
 * FIND OUT WHICH EVENT CAUSED THE SIGWINDOW.
 *
 * STEP 9:  Set the event mask and call weventpoll:
 */
    event_mask = EVENT_B1_DOWN | EVENT_B2_UP | EVENT_SIZE;
    if (weventpoll(wfd, &event_mask, &count, &x, &y) < 0)
    {
        perror("weventpoll wfd");
        exit(1);
    }

/*
 * Check for the individual events and print an appropriate message:
 */
   if (event_mask & EVENT_B1_DOWN)
        printf("Button 1 was depressed\n");
   if (event_mask & EVENT_B2_UP)
        printf("Button 2 was released\n");
   if (event_mask & EVENT_SIZE)
        printf("The window's has been resized.\n");
   return(0);
}
```

Step 9: Stop Window Communication

Before the event detection program terminates, it must stop communication with the window. If you don't remember how to do this, review the "Concepts" section of the "Window Manipulation" chapter.

Related Routines

Two other window input routines are applicable to event detection:

- *wgetsigmask* returns the event mask that was set by the calling process for a given window. This is useful if you want to alter the event mask for a window.
- *weventclear* clears events' internal counters (specified by an event mask parameter) so that an immediate call to *weventpoll* will return a 0 count for the cleared events.

Performance Considerations

Enabling the SIGWINDOW interrupt on every movement of the locator, via the EVENT_ECHO bit in the event mask, may significantly degrade system performance. This is because the system must constantly track and signal any locator movements. However for most applications, tracking the locator is not necessary anyway, because *weventpoll* returns the locator's coordinates when a button press event occurs: Most programs can wait to get the locator's position until the user presses a button, or when a hotspot is activated.

Example

The following program, named *poll_events.c*, polls for three events: EVENT_SIZE, EVENT_B2_UP, and EVENT_B1_DOWN. If any of the events occur, a message describing the event is written to standard output, and the program exits. You must provide the name of an existing window's device interface to the program; for example:

```
poll_events wconsole
```

will check for the events for the window named *wconsole.* The program source is found in the *man_examples* directory.

(**Note**: Programs which give excellent examples of event detection can also be found in the chapters "Arrows and Elevators" and "Graphics Window Hotspots.")

```
#include <fcntl.h>        /* system I/O call definitions              */
#include <signal.h>       /* signal definitions including SIGWINDOW   */
#include <window.h>       /* window library definitions               */

int  wfd;                 /* window type file descriptor              */

main(argc, argv)
int   argc;               /* number of arguments on command line      */
char *argv[];             /* command line argument list               */
{
    int  wmfd;                 /* window manager file descriptor       */
    char wt_path[WINNAMEMAX];  /* path name for window type            */
    int  event_mask;           /* mask of events to be trapped         */
    int  est_wm_com();         /* routine to establish wm communication */
    int  term_wm_com();        /* routine to terminate wm communication */
    int  which_event();        /* signal handling procedure            */
```

```
/*
 * START WINDOW MANAGER COMMUNICATION:
 */
    if ((wmfd = est_wm_com()) == -1)
    {
           perror("est_wm_com");
           exit(1);
    }

/*
 * POLL FOR EVENT_B1_DOWN, EVENT_B2_UP, AND EVENT_SIZE.
 *
 * STEP 1:  Start communication with the window in which
 *          event detection is to be done.
 */
    wmpathmake("WMDIR", argv[1], wt_path);
    if ((wfd = open(wt_path, O_RDWR)) < 0)
    {
           perror("open wfd");
           exit(1);
    }
    if (winit(wfd) < 0)
    {
           perror("winit wfd");
           exit(1);
    }

/*
 * STEP 2:  Set up the signal handler to catch SIGWINDOW:
 */
    if (signal(SIGWINDOW, which_event) == -1)
    {
           perror("signal SIGWINDOW");
           exit(1);
    }

/*
 * STEP 3:  Define the Event Mask:
 */
    event_mask = EVENT_B1_DOWN | EVENT_B2_UP | EVENT_SIZE;
```

```
/*
 * STEP 4:  Call wsetsigmask(3W):
 */
    if (wsetsigmask(wfd, event_mask) < 0)
    {
           perror("wsetsigmask wfd");
           exit(1);
    }

/*
 * STEP 5:  Wait for an event to occur:
 */
    pause();

/*
 * STEP 9:  Terminate communication with the window:
 */
    if (wterminate(wfd) < 0)
    {
           perror("wterminate wfd");
           exit(1);
    }
    if (close(wfd) < 0)
    {
           perror("close wfd");
           exit(1);
    }

/*
 * STOP WINDOW MANAGER COMMUNICATIONS:
 */
    if (term_wm_com(wmfd) == -1)
    {
           perror("term_wm_com wmfd");
           exit(1);
    }
    exit(0);
}
```

```
/*
 * STEPS 6,7:  Receive the SIGWINDOW signal and execute the signal handler:
 *
 * NOTE:  When the signal is generated, the signal handler is called with
 *        its parameter set to the signal value that caused the interrupt.
 */
which_event(signal)
int signal;                     /* signal value set when routine is called */
{
    int  event_mask;            /* mask of events to be polled for         */
    int  count;                 /* number of times the event has occurred  */
    int  x, y;                  /* locator position at time of the event   */

/*
 * FIND OUT WHICH EVENT CAUSED THE SIGWINDOW.
 *
 * STEP 8:  Set the event mask and call weventpoll:
 */
    do {
            event_mask = EVENT_B1_DOWN | EVENT_B2_UP | EVENT_SIZE;
            if (weventpoll(wfd, &event_mask, &count, &x, &y) < 0)
            {
                perror("weventpoll wfd");
                exit(1);
            }

/*
 * Check for the individual events and print an appropriate message:
 */
           if (event_mask & EVENT_B1_DOWN)
                printf("Button 1 was depressed\n");
           if (event_mask & EVENT_B2_UP)
                printf("Button 2 was released\n");
           if (event_mask & EVENT_SIZE)
                printf("The window's has been resized.\n");

   } while (event_mask != 0);     /* LOOP to ensure you don't miss
                                     any events */
   return(0);
}
```

Locator and Echo Routines 7

A **locator** is any HP-HIL input device that provides *x,y* location information. The mouse, and graphics tablet stylus or puck switch are locator devices. The window manager uses locator information to move the **echo** (also known as a **pointer** or **sprite**) on the display screen; that is, when the user moves a locator device, the window manager moves the echo on the display screen. Using locator and echo routines, a program can:

- get locator information—i.e., the locator's *x,y* position and which buttons are pressed
- set the locator's *x,y* position
- change the echo's representation
- customize the echo
- enable full-screen sprite control.

Concepts

This section presents concepts you should understand before using locator and echo routines.

Absolute Locator Device

The graphics tablet stylus and puck switch provide **absolute** location information. That is, the user specifies exact coordinates by pointing at an *x,y* location on the graphics tablet. Every *x,y* location on the graphics tablet corresponds to an *x,y* location on the display screen.

Note, however, that the WMLOCSCALE environment variable can be used to **scale** the graphics tablet so that only a sub-portion of the graphics tablet corresponds to the display screen. See the "Environment Variablesx" chapter of the *HP Windows/9000 User's Manual* for details on scaling the graphics tablet via the WMLOCSCALE environment variable.

Relative Locator Device

The keyboard cursor keys and the mouse provide **relative** location information. These devices don't specify exact coordinates; they tell the window system the direction of movement so that the system can track the exact location.

The Echo

The echo is a Starbase graphics **sprite** that shows the locator's current location. The echo gets its name because it echoes the locator's position on the display screen.

Using echo manipulation and customization routines, a program can change the echo to different representations, such as a full-screen cross hair, small tracking cursor, or user-defined picture.

The Locator's Hot Spot

The locator and echo are directly related. Moving the locator causes the echo to move similarly on the screen. The exact relation between the two is given by the echo's **hot spot**.

The hot spot is a special pixel in the echo. This pixel is denoted in *x,y* pixel coordinates relative to the upper-left corner of the echo. The hot spot tells the window system how the echo should align over the coordinates specified by the locator. Figure 7-1 illustrates this concept.

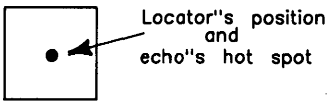


Figure 7-1. The Echo's Hot Spot

The echo's hot spot always covers the screen pixel whose coordinates are returned by the locator. However, the hot spot's location within the echo may vary, depending on which echo is displayed, and the displayed echo depends on which interactive manipulation area the locator is positioned over. Table 7-1 shows where the hot spot is for the various echoes displayed by the window system.

Table 7-1. Standard Hot Spots

Echo	**Where Its Hot Spot Is**
⊡	When the echo is located over the dither pattern, the hot spot is located directly in the middle of the box.
+	When the echo is located in a window's border and manipulation areas, the hot spot is in the middle of the cross hairs.
⇖	When the echo is located over a shifted soft key or window contents area, the hot spot is in the upper-leftmost pixel, at the arrow's tip.
⇙	When located over an unshifted soft key, the hot spot is the lower-leftmost pixel, at the arrow's tip.

Getting Locator Information

You can determine the current location of the locator via the *wgetlocator(3W)* routine. In addition to returning the locator's position, this routine returns the state of the locator buttons **at the time the routine is called**.

Procedure

To get locator information, call *wgetlocator*. The syntax for this routine is:

```
wgetlocator(fd, x, y, buttons)
```

Fd is an integer file descriptor returned from starting communication with a window. Locator coordinates are returned **relative** to the window's anchor point (the upper-leftmost pixel in the window's contents area).

The *x,y* parameters point to integers containing the pixel coordinates of the locator (and echo's hot spot) with respect to the window's anchor point. The window's anchor point is at location *0,0*.

Buttons is a pointer to an integer specifying the state of the locator buttons when *wgetlocator* is called. The least-significant bit of this integer represents button one; the next bit, button two; and so on. If a bit is set, then the button is down.

Precautions

This precaution applies only to graphics windows. If you use a graphics escape (*gesc(3G)*) to lock the screen, then do not call *wgetlocator* until after the screen is unlocked. Calling *wgetlocator* with a locked graphics window will cause the system to hang; you must do a hard reset to get out of this state.

Example

The following function, named *loc_in_user.c*, determines whether the locator is within a window's user (contents) area. If an error occurs when inquiring, then **-1** is returned; if the locator is within the window's user area, then **1** is returned; if the locator is outside the window's contents area, then **0** is returned. The source for this routine is found in *man_examples*.

```
#include <window.h>      /* window routine definitions */
loc_in_user( wfd )
int     wfd;             /* window's file descriptor   */
{
    int w, h;            /* window's width and height (of user area) */
    int wx, wy;          /* window's x,y-coordinates                 */
    int dx, dy, rw, rh;  /* other parameters to wgetcoords           */
    int lx, ly;          /* the locator's x,y position               */
    int buttons;         /* locator button mask                      */

/*
 * First, get the pixel width and height of the window:
 */
    if (wgetcoords(wfd, &wx,&wy, &w,&h, &dx,&dy, &rw,&rh) < 0) return(-1);
/*
 * Now get the locator's position:
 */
    if (wgetlocator(wfd, &lx,&ly, &buttons) < 0) return(-1);
/*
 * Check if the locator's position is within the window's user area:
 */
    if ((lx >= 0 && lx < w) && (ly >= 0 && ly < h))
        return(1);
    else
        return(0);
}
```

Moving the Locator

The *wsetlocator(3W)* routine sets the locator's and echo's *x,y* location. Moving the locator to an *x,y* location causes the echo to move to the same *x,y* pixel on the display screen. If the specified location is off screen, then *x,y* are adjusted to keep the echo within the screen boundaries.

Procedure

To move the locator call *wsetlocator*; its syntax is:

```
wsetlocator (fd, x,y)
```

Fd is the file descriptor of a window with which communication is started. The locator will be moved relative to this window.

The *x,y* parameters are integers specifying the new location. These coordinates are interpreted as being relative to the window's anchor point. The window's anchor point is at *0,0.*

Example

The following function, named *reset_loc.c*, moves the locator to the anchor point of the window specified by the file descriptor parameter `wfd`. If the function is successful, it returns 0; otherwise, it returns $^{-}1$. The source for this routine is found in *man_examples.*

```
#include <window.h>  /* window constant definitions */
reset_loc( wfd )
int  wfd;            /* window's file descriptor    */
{
     if (wsetlocator(wfd, 0, 0) < 0)
         return(-1);
     else
         return(0);
}
```

Changing the Echo

When the locator is moved over a window's user area, a standard echo (an arrow pointing up and left) is displayed by default. With the *wsetecho(3W)* routine, a program can replace the standard echo with a Starbase-compatible echo or a user-defined raster image.

This capability is powerful because it allows you to use special echoes in your own applications.

Procedure

To manipulate and change the echo's representation, call *wsetecho*; its syntax is:

`wsetecho(`*fd, echo_value, x2,y2, optimized*`)`

where:

- *fd*—is an integer file descriptor returned from starting communication with a window. Whenever the locator is moved over the visible part of the window's contents area, the echo will be displayed as defined by the remaining parameters.
- *echo_value*—is the type of echo to use. If its value is 7, then a user-defined echo is displayed instead of the standard echo. If its value is 1-6 or 8, then a Starbase echo types is displayed. If the value is 9 or greater, then a device-dependent representation is displayed.

  See the *wsetecho(3W)* reference page for details on the values supplied for this parameter.
- *x2,y2*—are the echo's anchor position or box width and height, depending on the value of *echo_value*. For example, if *echo_value* is 5 (rubber band rectangle), then *x2,y2* represent the anchor point for the rubber band, and the lower-right corner is moved by the locator. (This representation is used when you interactively change the size of a window.)
- *optimized*—is a boolean that is either 0 or 1. If 0, then the echo is displayed and moves exactly as defined. If 1, then the echo representation may be modified to make it track the best way possible for the display hardware.

  A value of 1 is recommended since it takes advantage of specialized hardware and is more efficient.

Related Routines

- *wgetecho(3W)* retrieves the above parameters for a specified window. This is useful if you wish to determine the what the current echo is.
- *wsetrasterecho(3W)* allows you to define a custom echo. By calling *wsetecho* with the *echo_value* parameter set to 7, the window's default echo will be replaced with the echo defined by *wsetrasterecho*. Defining a custom echo with *wsetrasterecho* is discussed in the next section, "Customizing the Echo."
- *wset_hw_sprite_color(3W)* allows the colors used for displaying the sprite, when using hardware support for sprites on the HP 98730, to be set. This must be called after every *wsetecho(3W)* unless the defaults (or the colors specified by the last *wset_hw_sprite_color(3W)*) are acceptable.
- *wget_hw_sprite_color(3W)* inquires the colors being used for displaying the hardware sprite on the HP 98730.

Example

The following function, named *shrink_it.c*, allows the user to interactively shrink a specified window. It uses a rubber band rectangle echo to make the window size visible as it is changed.

```
#include <fcntl.h>       /* system I/O call definitions                  */
#include <signal.h>      /* signal definitions, including SIGWINDOW      */
#include <window.h>      /* window library definitions                   */

int  global_wfd;         /* window type file descriptor                  */

shrink_it(wfd)
int wfd;                 /* window's file descriptor                     */
{
    int  event_mask;     /* mask of events to be trapped                 */
    int  size_window();  /* signal handling procedure                    */
    int  echo_type;      /* original echo type                           */
    int  x, y;           /* original echo anchor point                   */
    int  optimized;      /* original echo optimized state                */

/* INTERACTIVELY SHRINK A WINDOW.
 * STEP 1:  Get the current echo values:
 */
   global_wfd = wfd;
   if (wgetecho(wfd, &echo_type, &x, &y, &optimized) < 0)
    {
           perror("wgetecho wfd");
           exit(1);
    }
/*
 * STEP 2:  Set up the signal handler to catch SIGWINDOW:
 */
   if (signal(SIGWINDOW, size_window) < 0)
   {
          perror("signal SIGWINDOW");
          exit(1);
   }


/*
 * STEP 3:  Change the sprite to a rubber band rectangle:
 */
   if (wsetecho(wfd, ECHO_RUBRECT, 0, 0, ECHO_NOOPT) < 0)
   {
          perror("wsetecho wfd");
          exit(1);
   }


/*
 * STEP 4:  Set up to size the window when button 1 goes up:
 */
   event_mask = EVENT_B1_UP;
   if (wsetsigmask(wfd, event_mask) < 0)
   {
          perror("wsetsigmask wfd");
          exit(1);
   }
   pause();


/*
 * STEP 5:  Change the sprite back to its original representation:
 */
   if (wsetecho(wfd, echo_type, x, y, optimized) < 0)
   {
          perror("wsetecho wfd");
          exit(1);
   }

}
```

```
/*
 * The signal handler:
 */
size_window(signal)
int signal;                     /* signal value set when routine is called */
{
    int  x, y;                  /* locator position    */
    int  buttons;               /* locator button mask */


/*
 * RESIZE THE WINDOW WHEN BUTTON 1 IS RELEASED.
 *
 * Find out the current locator position:
 */
   if (wgetlocator(global_wfd, &x, &y, &buttons) < 0)
   {
        perror("wgetlocator wfd");
        exit(1);
   }


/*
 * Change the size of the window:
 */
   if (wsize(global_wfd, x, y) < 0)
   {
        perror("wsize wfd");
   }
   return(0);
}
```

Customizing the Echo

As described in the previous section, each window can have a customized echo—i.e., a user-defined picture. Customized echoes are defined via the *wsetrasterecho(3W)* routine. Once defined, the customized echo can then be displayed via *wsetecho(3W)*.

Procedure

To define and display a customized echo:

1. Define the echo via *wsetrasterecho(3W)*.
2. Display the echo via *wsetecho(3W)*.

Define the Echo Via wsetrasterecho(3W)

To define the echo, call *wsetrasterecho*; its syntax is:

```
wsetrasterecho(fd, dx,dy, w,h, rule,mask_rule, mask, image)
```

where:

- *fd*—is the file descriptor of the window for which the custom echo will be defined.
- *dx,dy*—are the offset of the echo hot spot to the upper-left corner of the echo. For example, if the hot spot is the upper-leftmost pixel of the echo, then *dx,dy* are *0,0*.
- *w,h*—are the echo's pixel width and height.
- *rule,mask_rule*—are the rules to be used when displaying the echo against the screen background. For example, with these rules, you can specify that the echo's image is to be exclusive-OR'd with the background, thus making it the complement of anything it appears over.

  Valid values for these parameters are defined in Table 7-2.
- *mask*—is a pointer to an array of up to 128 characters; each bit of the array represents one pixel in the echo. This array contains a mask that defines the shape of the echo. This mask is defined like an icon mask, except that a pointer to the mask is passed as a parameter—the mask is not defined in a file as with icons. See the "Icons" chapter for details on defining the mask.
- *image*—is a pointer to a **byte**-per-pixel array of up to 1024 characters. This array defines the image to be used with *mask*. Each byte represents the color to use for the corresponding bit in *mask*. The rules for defining the image are the same as for icons; however, the image is supplied as a parameter, not a file. See the "Icons" chapter for details on defining the image.

Table 7-2. Valid rule and mask_rule Values[1]

Value	Resulting Destination Pixel
0	ZERO
1	*source* AND *destination*
2	*source* AND NOT *destination*
3	*source* (the default rule)
4	NOT *source* AND *destination*
5	*destination*
6	*source* EXCLUSIVE OR *destination*
7	*source* OR *destination*
8	NOT *source* AND NOT *destination*
9	*source* EXCLUSIVE NOR *destination*
10	NOT *destination*
11	*source* OR NOT *destination*
12	NOT *source*
13	NOT *source* OR *destination*
14	NOT *source* OR NOT *destination*
15	ONE

Display the Echo Via wsetecho(3W)

After defining the custom echo via *wsetrasterecho*, a program must call *wsetecho* to display the new echo in place of the old one. To replace the current echo with the customized echo, call *wsetecho* as:

```
echo_value = 7;       /* user-defined raster echo */
x2 = y2 = -1;         /* these params are no-ops for echo_value == 7 */
optimized = 1;        /* optimize the echo */
wsetecho(fd, echo_value, x2,y2, optimized);
```

[1] The *source* is the pixel to be written to the screen; *destination* is the current value of that pixel on the screen where *source* is to be written. For example, if *value* is 1, then the resulting pixel will be turned on if both *source* AND *destination* are asserted.

Related Routines

- *wgetrasterecho(3W)* returns the above parameters for the pointer in a given window. This is useful if you wish to change the custom echo for a window.
- *wset_hw_sprite_color(3W)* allows the colors used for displaying the sprite, when using hardware support for sprites on the HP 98730, to be set. This must be called after every *wsetecho(3W)* unless the defaults (or the colors specified by the last *wset_hw_sprite_color(3W)*) are acceptable.
- *wget_hw_sprite_color(3W)* inquires the colors being used for displaying the hardware sprite on the HP 98730.

Example

The following function, *echo_hand.c*, defines the custom echo to be a pointing hand with the hot spot being the finger tip. It then calls *wsetecho* to ensure that the hand is displayed whenever the locator moves over the window's user area.

```
#include <stdio.h>              /* standard I/O definitions   */
#include <fcntl.h>              /* file definitions           */
#include <window.h>             /* window library definitions */

/*
 * Define the echo mask:
 */
char mask[72] =        {        0x0,    0x0,    0x0,    0x0,
                                0x0,    0x1E,   0x0,    0x0,
                                0x0,    0x1F,   0xFF,   0xC0,
                                0x0,    0x0F,   0xFF,   0xE0,
                                0x0,    0x07,   0xFF,   0xF8,
                                0x7F,   0xFF,   0xFF,   0xFC,
                                0xFF,   0xFF,   0xFF,   0xFE,
                                0xFF,   0xFF,   0xFF,   0xFE,
                                0x7F,   0xFF,   0xFF,   0xFF,
                                0x0,    0x07,   0xFF,   0xFF,
                                0x0,    0x07,   0xFF,   0xFF,
                                0x0,    0x07,   0xFF,   0xFF,
                                0x0,    0x01,   0xFF,   0xFF,
                                0x0,    0x01,   0xFF,   0xFF,
                                0x0,    0x01,   0xFF,   0xFF,
                                0x0,    0x0,    0x7F,   0xFF,
                                0x0,    0x0,    0x7F,   0xFF,
                                0x0,    0x0,    0x7F,   0xF0,
                       };
```

```
/*
 * Define the echo image:
 */
char image[576] = {
        0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0, 0,0,0,1,1,1,1,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0, 0,0,0,1,0,0,0,1, 1,1,1,1,1,1,1,1, 1,1,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0, 0,0,0,0,1,0,0,0, 0,0,0,0,0,0,0,0, 0,0,1,0,0,0,0,0,
        0,0,0,0,0,0,0,0, 0,0,0,0,0,1,0,0, 0,0,0,0,0,0,0,0, 0,0,0,1,1,0,0,0,
        0,1,1,1,1,1,1,1, 1,1,1,1,1,1,1,1, 1,1,1,0,0,0,0,0, 0,0,0,0,0,1,0,0,
        1,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,1,0,
        1,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,1,
        0,1,1,1,1,1,1,1, 1,1,1,1,1,1,1,1, 1,1,1,1,1,0,0,0, 0,0,0,0,0,0,0,1,
        0,0,0,0,0,0,0,0, 0,0,0,0,0,1,0,0, 0,0,0,0,1,1,0,0, 0,0,0,0,0,0,0,1,
        0,0,0,0,0,0,0,0, 0,0,0,0,0,1,0,0, 0,0,0,0,1,1,0,0, 0,0,0,0,0,0,0,1,
        0,0,0,0,0,0,0,0, 0,0,0,0,0,1,1,1, 1,1,1,1,1,0,0,0, 0,0,0,0,0,0,0,1,
        0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,1, 0,0,0,0,0,1,1,0, 0,0,0,0,0,0,0,1,
        0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,1, 0,0,0,0,0,1,1,0, 0,0,0,0,0,0,0,1,
        0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,1, 1,1,1,1,1,1,0,0, 0,0,0,0,0,0,0,1,
        0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,1,0,0,0,0,1,1, 0,0,0,0,0,0,0,1,
        0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,1,0,0,0,0,1,1, 0,0,0,0,1,1,1,0,
        0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0, 0,1,1,1,1,1,1,1, 1,1,1,1,0,0,0,0,
};


/*
 * Define some program constants:
 */
#define DONT_CARE       -1
#define ECHO_WIDTH      32              /* width of the echo in pixels    */
#define ECHO_HEIGHT     18              /* height of the echo in pixels   */
#define ECHO_X2         DONT_CARE       /* there is no anchor point for   */
#define ECHO_Y2         DONT_CARE       /*      raster echoes              */
#define ECHO_DX         0               /* hot spot for the echo          */
#define ECHO_DY         -7
#define ECHO_OPT        1               /* optimize the echo              */
#define ECHO_RULE       7               /* replacement rule for the image */
#define ECHO_MASKRULE   4               /* replacement rule for the mask  */
#define ECHO_TYPE       7               /* define a raster echo           */
```

```
echo_hand(wfd)
int wfd;                /* window's file descriptor */
{
/*
 * SET THE WINDOW'S ECHO TO A POINTING HAND.
 *
 * STEP 1:  Define a new rasterecho:
 */
    if (wsetrasterecho(wfd,ECHO_DX,ECHO_DY,ECHO_WIDTH,ECHO_HEIGHT,
                       ECHO_RULE,ECHO_MASKRULE,mask,image) < 0) return(-1);


/*
 * STEP 2:  Change the echo for the current window to a raster echo:
 */
    if (wsetecho(wfd,ECHO_TYPE,ECHO_X2,ECHO_Y2,ECHO_OPT) <0) return(-1);

/*
 * STEP 3:  Move the echo to the upper left hand corner of the window:
 */
    if (wsetlocator(wfd,0,0) < 0) return(-1);
    return(0);
}
```

Enabling Full-Screen Sprite Control

By default, the echo type displayed by the window manager depends on what area of the screen the locator is position over. Similarly, when the user presses a button, the action taken by the window manager for the particular button press depends on the echo's location. For example, when located over a window's border, the echo is displayed as cross-hairs; and when the user presses the select button, the window manager will display a system pop-up menu for the window.

This default mode can be changed to **full-screen sprite mode** via the *wscrn_sprite_mode(3W)* routine. Full-screen sprite mode causes the echo to be set to that of a given window, regardless of the echo's location. When the echo is moved over other windows, the desk top, or window borders, its representation will not change. In addition, a locator button press or [Select] key press while in full-screen sprite mode will not cause the default action; rather, the button press will be transmitted as a button press event to the full-screen sprite mode window.

If the user presses any key other than the [Select] key or buttons while in full-screen sprite mode, then this mode is aborted—the window manager returns to default mode. Then the SIGWINDOW signal is sent to all processes that have enabled the EVENT_ABORT event for this window.

All button press events are sent to the process that enables full-screen sprite mode, **if** the process has enabled event detection for th se button presses. As described above, these button presses would normally cause a default action by the window manager; however, in full-screen sprite mode, the window manager does **not** do anything for the button press. Therefore, a program must check for button presses and do default window manager actions itself **if** the program wishes to emulate the window manager.

Programs can determine the default window manager configuration from the value of the WMIUICONFIG environment variable (see the "Environment Variables" chapter of the *HP Windows/9000 User's Manual*). To get this value, programs will typically call *wminquire(3W)*, which returns the value of a window manager environment variable.

Procedure

Programs that use full-screen sprite mode will typically:

1. Set up a signal handler for full-screen sprite mode.
2. Call wscrn_sprite_mode(3W).

Set Up a Signal Handler for Full-Screen Sprite Mode

Before enabling full-screen sprite mode, a program should set up a signal handler to catch button presses that occur during full-screen sprite mode. Depending on your needs, the program may also need to catch the `EVENT_ABORT` event, which is generated when full-screen sprite mode is aborted.

Call wscrn_sprite_mode(3W)

After the signal handler is set up, the program can call *wscrn_sprite_mode*; its syntax is:

```
wscrn_sprite_mode(fd, value)
```

The integer file descriptor returned from starting communication with the window should be passed as the *fd* parameter.

The *value* parameter determines the action of *wscrn_sprite_mode*. If *value* is `SETFULLSPRITE`, then full-screen sprite mode is enabled. If *value* is `SETNOFULLSPRITE`, then full-screen sprite mode is turned off, which causes the `EVENT_ABORT` event. If *value* is `GETFULLSPRITE`, then the current mode is returned: `SETNOFULLSPRITE` if disabled for the window, `SETFULLSPRITE` if enabled for the window.

Precautions

- The process that enables full-screen sprite mode for a window is the only process that can disable full-screen sprite mode for that window.
- If a process other than the enabling process attempts to disable full-screen sprite mode, then a system error occurs, and *errno(2)* is set to `EACCESS`.
- If a process attempts to enable full-screen sprite mode for a window, but full-screen sprite mode is already enabled for a window, then a system error occurs, and *errno(2)* is set to `EBUSY`.
- If a process attempts to disable full-screen sprite mode in a window in which full-screen sprite mode is *not* already enabled, a system error occurs, and *errno(2)* is set to `EBADF`.

Example

The following code sets up a signal handler named **`full_screen`**, which will handle any button press or `EVENT_ABORT` events occurring during full-screen sprite mode for the window identified by `wfd` (the window interface file descriptor). It then turns on full-screen sprite mode for the window.

```
#include <window.h>
#include <signal.h>
#include <stdio.h>
main()
{
    int     wfd;             /* file descriptor for the window */
    int     event_mask;      /* event mask for the window */
    int     full_screen();   /* signal handler for full-screen sprite mode */
       :
       :

/*
 * At this point, the program has already started communication with
 *   the window that full-screen sprite mode will be used with.
 * "wfd" is the file descriptor returned from starting communication.
 *
 * STEP 1:  Set up the signal handler:
 */
    if (signal(SIGWINDOW, full_screen) < 0)
    {
        perror("signal SIGWINDOW");
        exit(1);
    }
 /*
  * Define the event mask for button 1 & 2 events and EVENT_ABORT:
  */
    event_mask = EVENT_B1_DOWN | EVENT_B1_UP |  /* button 1 events */
                 EVENT_B2_DOWN | EVENT_B2_UP |  /* button 2 events */
                 EVENT_ABORT;                   /* abort full-screen mode */
    if (wsetsigmask(wfd, event_mask) < 0)
    {
        perror("wsetsigmask wfd");
        exit(1);
    }
 /*
  * STEP 2: Call wscrn_sprite_mode to enable full-screen sprite mode:
  */
    if (wscrn_sprite_mode(wfd, 1) < 0)
    {
        perror("wscrn_sprite_mode wfd");
        exit(1);
    }
       .
       .
       .
}   /* END of main() */

       .
       .
       .
int     full_screen(signal)
int     signal;
{
        .
        .
        .
/*
 * The body of the signal handler goes here.  This is where the program
 *     handles button presses and aborts during full-screen sprite mode.
 */
        .
        .
        .
 }   /* END of full_screen() */
```

Notes

Arrows and Elevators 8

This chapter describes how to use **scroll bar** routines with graphics windows. By calling scroll bar routines, a program can:

- Enable pan mode (the default for all newly created graphics windows), in which clicking the locator over arrows or elevators causes the window to pan.
- Enable user mode, in which an event occurs when the user clicks the locator over either an arrow or elevator.

Concepts

This section discusses concepts essential to using scroll bar routines with graphics windows. Be sure to read this section before using these routines.

Scroll Bars

Each graphics window has a **vertical** and a **horizontal scroll bar**. A scroll bar is simply an area in a window's border in which the window's arrows and elevators are displayed. The vertical scroll bar contains the vertical arrows and elevator; the horizontal scroll bar contains the horizontal arrows and elevator. Figure 8-1 shows the scroll bar, arrows, and elevators in a typical graphics window.

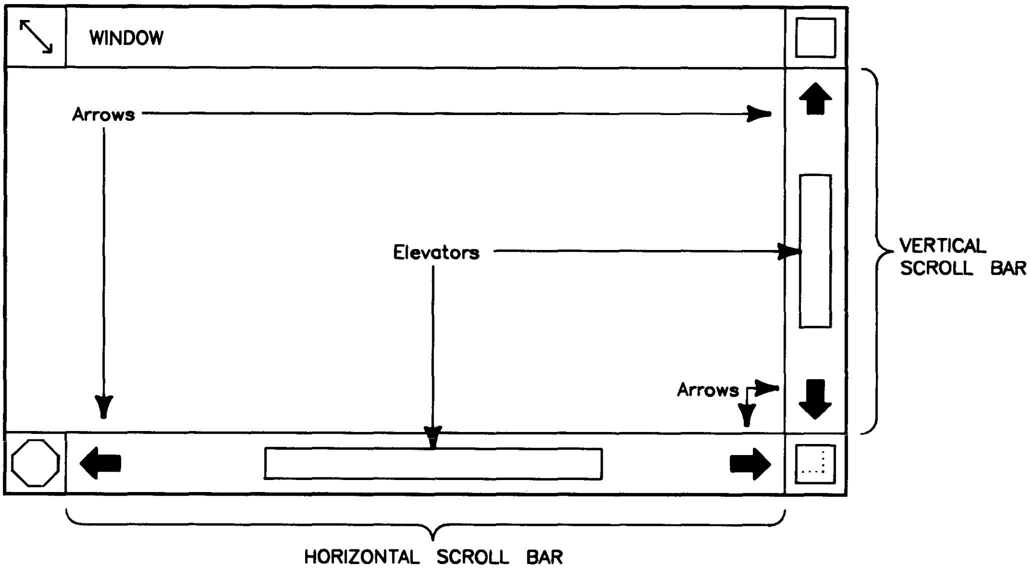


Figure 8-1. Scroll Bars, Arrows, and Elevators.

Arrows and elevators are either **enabled** or **disabled**. When an arrow or elevator is enabled, it is visible in the window's border. In addition, when the user clicks the locator over an arrow or elevator, the window manager does an action appropriate to the **scroll bar mode**. Two scroll bar modes are supported: **pan** mode and **user** mode.

Pan and user modes are discussed in the next two sections. If an elevator or arrows are disabled in a scroll bar, then they are not visible, and they have no function.

In the example window in Figure 8-1, the arrows and elevators are enabled in both the vertical and horizontal scroll bars.

By default, when a graphics window is created, only its arrows are enabled in the vertical and horizontal scroll bars. Also, the window is in pan mode.

Pan Mode

In pan mode, the arrows and elevators cause a window to pan; that is, they change the window's pan position. However, arrows and elevators pan differently, as discussed next.

Panning via Arrows

To pan a window via arrows, move the pointer over an arrow and click the Select button. The picture will move in the direction of the arrow; the window's pan position will change opposite the direction of the arrow. The magnitude of the pan depends on the size of the window's view: pans are always one-fifth the size of the view into the raster.

Recall that the origin of the raster (*0,0*) is the upper-left corner of the raster, and a window's pan position (*dx,dy*) is the location of the view with respect to the raster. Figure 8-2b (later in this section) shows the relationship between pan position and a window's view into the raster. Table 8-1 defines the relationship between arrows, the direction the picture moves, and the effect on the window's pan position.

Table 8-1. Relationship between Arrows, Pan Direction, and Pan Position

Arrow Direction	Picture Moves	Effect on Pan Position
Up	Up	Pan position moves down: *dy* = *dy* + (vertical view ÷ 5) *dx* does not change
Down	Down	Pan position moves up: *dy* = *dy* - (vertical view ÷ 5) *dx* does not change
Right	Right	Pan position moves left: *dx* = *dx* - (horizontal view ÷ 5) *dy* does not change
Left	Left	Pan position moves right: *dx* = *dx* + (horizontal view ÷ 5) *dy* does not change

Note, however, that the direction of the pan can be reversed by setting the 0x8 000 000 bit in the WMIUICONFIG environment variable. (See the "Environment Variables" chapter of the *HP Windows/9000 User's Manual* for details on setting this variable.)

Panning via Elevators

Elevators provide another intuitive way to pan windows. The advantage of elevators over arrows is that elevators can pan much faster: arrows pan a fixed amount, while elevators pan directly to a location in the window's raster.

To pan a window via elevators, position the locator over the elevator and press the Select button. The elevator will then appear as a dotted box, which can then be moved within the scroll bar (up and down in the vertical scroll bar; left and right in the horizontal scroll bar). The interactive elevator pan is completed by moving the elevator to the desired location and pressing the Select button again. When the interactive elevator operation is finished, the window will be panned to the position represented by the vertical and horizontal elevators.

The notion of an **elevator shaft** is useful. An elevator shaft is simply the length in which the elevator can move within the scroll bar. It is slightly shorter than the distance between the arrows. In the vertical scroll bar, the elevator moves up and down only; in the horizontal scroll bar, the elevator moves only left and right. The length of the elevator shaft is determined from the size of the window. For example, if the window grows wider, the horizontal elevator shaft will get longer, too. And if the window grows taller, the vertical elevator shaft will get longer, also.

In pan mode, the length of an elevator is determined from the window's current size and the size of the raster. The elevator's length is directly proportional to the ratio of the window's size to its raster size. The following formulas define the length of the vertical and horizontal elevators:

```
vert_elev_length = vert_shaft_length × (window_height ÷ raster_height)
horz_elev_length = horz_shaft_length × (window_width ÷ raster_width)
```

The position of an elevator within the shaft is determined from the window's pan position. For example, if the window's pan position is *0,0*, then the vertical elevator will be flush with the top of the vertical elevator shaft, and the horizontal elevator will be flush with the left end of the horizontal elevator shaft. If the window's view is flush with the lower-right corner of the window's raster, then the vertical elevator will be flush with the bottom of the vertical elevator shaft, and the horizontal elevator will be flush with the right end of the horizontal elevator shaft.

Figures 8-2a and 8-2b show a window with its elevators enabled, illustrating the relationship between elevator size/position and the window's pan position, window size, and raster size. In 8-2a the window is flush with the upper-left corner of its raster; in 8-2b the window is centered within its raster. Note the location and size of the elevators as the pan position changes in each part of the figure.

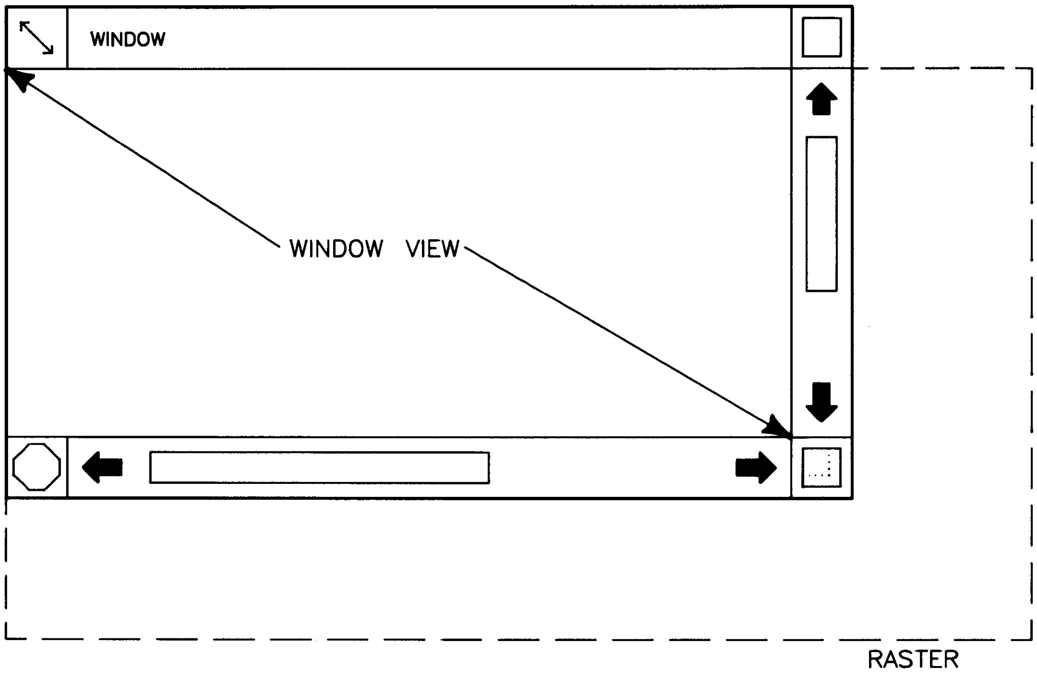


Figure 8-2a. View Flush with Upper-Left Corner of Raster

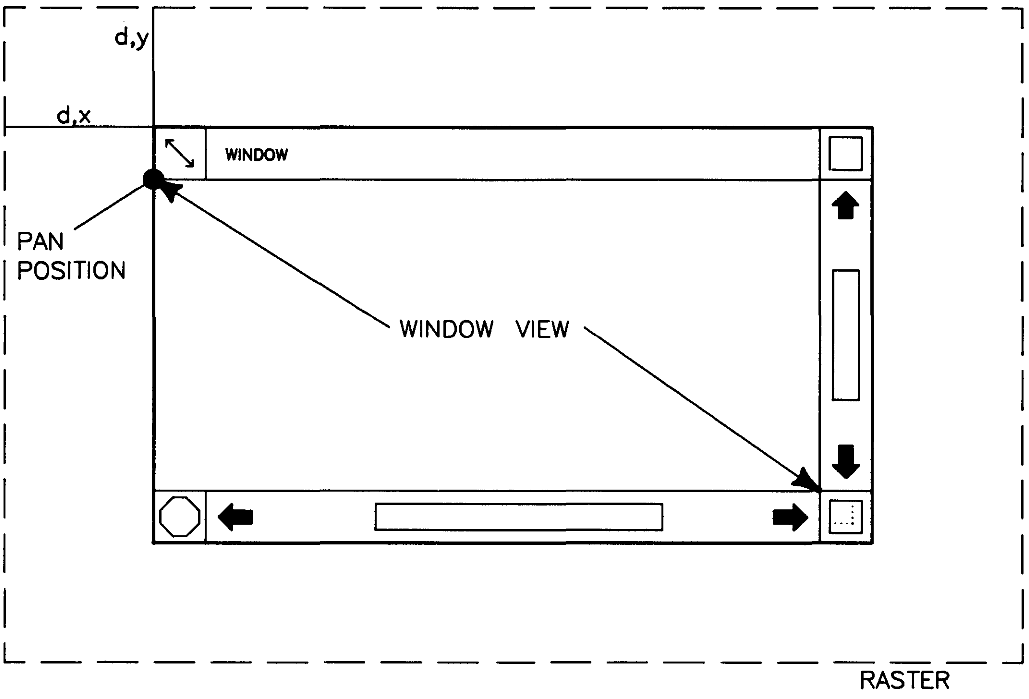


Figure 8-2b. View Centered within Raster

User Mode

In user mode, the arrows and elevators do not pan windows. Instead, when the user clicks the locator over an arrow or elevator, the window manager interprets it as an event. Clicking the locator over an arrow causes an `EVENT_SB_ARROW` event; completing an elevator move operation causes `EVENT_ELEVATOR`.

A program can detect these events by enabling SIGWINDOW for the desired event via *wsetsigmask(3W)*; then, on receiving SIGWINDOW, use *weventpoll(3W)* to determine which event occurred. This method is discussed in the section "Getting Scroll Bar Events in User Mode."

A program can also detect these events via **graphics window input** routines, discussed fully in the "Graphics Window Input" chapter. You should see that chapter for details on using this method.

Enabling Arrows, Elevators in Pan Mode

The *wscroll_set(3W)* routine enables arrows, elevators, or both for a graphics window in pan mode.

Procedure

To enable arrows or elevators in pan mode, call *wscroll_set*; its syntax is:

```
wscroll_set(fd, which, mode[, value, min, max, size])
```

The *value*, *min*, *max*, and *size* parameters are **not** used in pan mode; they are used only in user mode. When enabling pan mode, a program need only supply the *fd*, *which*, and *mode* parameters, described below. The other parameters, if supplied, will be ignored in pan mode.

fd

The *fd* parameter is an integer file descriptor returned from starting communication with the graphics window.

which

The *which* parameter identifies whether the vertical scroll bar, horizontal scroll bar, or both will be affected by this function. Set this parameter by ORing the following values, defined in *window.h*:

SCROLLBAR_V	Affect the vertical scroll bar.
SCROLLBAR_H	Affect the horizontal scroll bar.

To affect both the vertical and horizontal scroll bars, set the *which* parameter to the OR of both SCROLLBAR_V and SCROLLBAR_H; e.g.:

```
which = SCROLLBAR_V | SCROLLBAR_H;
```

To affect only the vertical arrows or elevator, set *which* to SCROLLBAR_V. To affect only the horizontal arrows or elevator, set *which* to SCROLLBAR_H.

mode

The *mode* parameter controls whether the arrows, elevator, or both are enabled within the affected scroll bar(s). Valid values for *mode* are defined in *window.h* as:

`SCROLLBAR_ELEVATOR`	Enable the elevator in the scroll bar(s).
`SCROLLBAR_ARROWS`	Enable the arrows in the scroll bar(s).

To enable only the arrows within the affected scroll bar(s), set *mode* to `SCROLLBAR_ARROWS`. To enable only the elevator, set *mode* to `SCROLLBAR_ELEVATOR`. To enable arrows **and** the elevator within the affected scroll bar(s), OR the two together, as:

```
mode = SCROLLBAR_ARROWS | SCROLLBAR_ELEVATOR.
```

Examples

Given the file descriptor returned from starting communication with a graphics window (`gwfd`), the following function:

1. Enables arrows **and** elevators in the window's vertical and horizontal scroll bars;
2. Sets the window's border to a normal border, via *wbanner(3W)*; and
3. Displays the window as the top window in the display stack, via *wtop(3W)*.

```
#include        <window.h>
int     pan_mode_top(gwfd)
int     gwfd;           /* fildes returned from starting comm with window */
{
        int     which;
        int     mode;

/*
 * Enable both the vertical and horizontal scroll bar:
 */
        which = SCROLLBAR_V | SCROLLBAR_H;
/*
 * Enable both the elevator and the arrows within each scroll bar:
 */
        mode = SCROLLBAR_ELEVATOR | SCROLLBAR_ARROWS;
/*
 * Call wscroll_set(3W):
 */
        if (wscroll_set(gwfd, which, mode) < 0)
                return(-1);
/*
 * Make the window's border normal and display the window:
 */
        wbanner(gwfd, SETBANNER);
        wtop(gwfd, SETTOP);
        return(0);
}
```

Given the file descriptor returned from starting communication with a graphics window (`gwfd`) the next function enables only the elevator in the vertical scroll bar:

```
#include        <window.h>
int     v_elev_only(gwfd)
int     gwfd;           /* fildes returned from starting comm with window */
{
        int     which;
        int     mode;

/*
 * Call wscroll_set(3W):
 */
        if (wscroll_set(gwfd, SCROLLBAR_V, SCROLLBAR_ELEVATOR) < 0)
                return(-1);
/*
 * Make the window's border normal and display the window:
 */
        wbanner(gwfd, SETBANNER);
        wtop(gwfd, SETTOP);
        return(0);
}
```

User Mode

The *wscroll_set(3W)* routine is also used to set a window to user mode. However, the parameters are set differently from those used in pan mode.

Procedure

To enable arrows or elevators in user mode, call *wscroll_set* as:

```
wscroll_set(fd, which, mode, value [,min, max, size])
```

The *fd*, *which*, *mode*, and *value* parameters must always be supplied. The *min*, *max*, and *size* parameters are optional; they specify elevator scale information.

fd

The *fd* parameter is an integer file descriptor returned from starting communication with the window.

which

The *which* parameter defines which scroll bars will be affected. The rules for setting *which* in user mode are the same as those for pan mode. For example, to affect both the vertical and horizontal scroll bars, set *which* to the OR of SCROLLBAR_V and SCROLLBAR_H:

```
which = SCROLLBAR_V | SCROLLBAR_H;
```

mode

As with pan mode, this parameter defines whether to enable arrows, elevators, or both in the affected scroll bar(s). But in addition to enabling arrows or scroll bars, *mode* has other functions, described next.

The first additional function of *mode* is to enable user mode. To enable user mode, OR *mode* with SCROLLBAR_USERMODE. For example, to enable user mode and the arrows and elevators, set *mode* as:

```
mode = SCROLLBAR_USERMODE | SCROLLBAR_ARROWS | SCROLLBAR_ELEVATOR;
```

The second additional function of *mode* is to indicate whether optional scale information is provided, i.e., whether the *min*, *max*, and *size* parameters are passed. This is useful only if elevators are enabled.

To specify elevator scale information, OR *mode* with SCROLLBAR_SCALE. If this bit is set in *mode*, then the *min*, *max*, and *size* parameters must be supplied also. If this bit is *not* set, but elevators are enabled, then *wscroll_set* assumes default scale values. Elevator scale parameters are discussed next.

value

When in user mode, the *value* parameter must be supplied. *Value* specifies the location to put the elevator within the elevator shaft. *Value* must be within the range specified by the *min* and *max* parameters, described next.

min, max, and size

If the SCROLLBAR_SCALE and SCROLLBAR_ELEVATOR bits are set in the *mode* parameter, then these parameters must be supplied. These parameters allow the programmer to define scale information for the elevators.

Table 8-2 defines each parameter. The **Default** column shows the values *wscroll_set* uses when elevators are enabled **but** the SCROLLBAR_SCALE bit is not set.

Table 8-2. Min, Max, and Size Parameters

Param	Description	Default
min	The value corresponding to the upper (if SCROLLBAR_V is set) or left (if SCROLLBAR_H is set) end of the elevator shafts. If both scroll bars are enabled, then *min* is the same for both the vertical and horizontal elevator shafts.	0
max	The value corresponding to the lower (if SCROLLBAR_V is set) or right (if SCROLLBAR_H is set) end of the elevator shafts. If both scroll bars are enabled, then *max* is the same for both the vertical and horizontal elevator shafts.	100
size	The length (on a scale of *min* to *max*) of the elevator. For example, if *min* and *max* are $^{-}100$ and +100, respectively, then setting *size* to 50 will cause the elevator to always be one-fourth the size of the elevator shaft ($50 \div (100 - {}^{-}100) = 1/4$).	10

Reusing Scroll Bar Scale Information

If *scrollbar_set* is called for the first time, and the SCROLLBAR_SCALE bit is not set in the *mode* parameter, then the defaults shown in Table 8-2 are used for scroll bar scaling. On subsequent calls to *scrollbar_set*, if the SCROLLBAR_SCALE bit is *not* set, *wscroll_set* uses the scroll bar scale values set by the last call.

This allows a program to set scroll bar scaling information once, when *wscroll_set* is first called for a window. Thereafter, the program doesn't have to re-supply scroll bar scale information (doesn't have to supply the *min max*, and *size* parameters).

Precautions

In user mode, the EVENT_ELEVATOR event, per se, does not cause the elevator to move. The application must move the elevator by calling *wscroll_set* again with the *value* parameter set "appropriately" for the attempted move.

The "appropriate" value is determined via event detection. This is discussed in detail in the "Getting Scroll Bar Events in User Mode" section, later.

Example

The following code segment enables user mode in a graphics window. The program has already started communication with the window; gwfd is the integer file descriptor returned from doing so. Both the vertical and horizontal elevators are enabled, but not the arrows. In addition, optional elevator scale information is provided.

```
#include   <window.h>
   ⋮
int     gwfd;   /* graphics window file descriptor */
int     which;  /* identifies which scroll bars to enable */
int     mode;   /* identifies:
                 *    - whether to enable arrows, elevators, or both
                 *    - user or pan mode
                 *    - optional elevator scale information
                 */
int     value;  /* where to put the elevator in the shaft */
int     min,
        max;    /* coordinates for ends of elevator shaft */
int     size;   /* size of elevator, based on min and max */
   ⋮
/*
 * Start communication with the window, obtaining "gwfd":
 */
          ⋮
/*
```

```
 * Enable both the vertical and horizontal scroll bars:
 */
        which = SCROLLBAR_V | SCROLLBAR_H;
/*
 * Enable the elevators only:
 */
        mode = SCROLLBAR_ELEVATOR;
/*
 * Enable user mode:
 */
        mode |= SCROLLBAR_USERMODE;
/*
 * Optional elevator scale information to be passed also:
 */
        mode |= SCROLLBAR_SCALE;
/*
 * Define scale information:
 */
        min = -100; max = 100;   /* from -100 to +100 */
        size = 25;               /* elevator will be 1/8th length of shaft */
        value = 0;               /* position the elevator in the middle */
/*
 * Call wscroll_set:
 */
        if (wscroll_set(gwfd, which, mode, value, min, max, size) < 0)
        {
                fprintf(stderr, "Error calling wscroll_set...\n");
                exit(1);
        }
           :
```

Getting Scroll Bar Events in User Mode

Once user mode is enabled for a window, a program can get **EVENT_SB_ARROW** and **EVENT_ELEVATOR** event information via either event detection routines, or graphics window input routines. This section discusses how to use event detection routines to get this information; graphics window input routines are discussed in the "Graphics Window Input Routines" chapter.

Procedure

To detect scroll bar events in a window, a program would typically do the following:

1. Call *signal(2)* to set up a signal handler for SIGWINDOW.
2. Call *wsetsigmask(3W)* to enable scroll bar events; then wait for events to occur.
3. Enable user mode for the window.
4. On receiving SIGWINDOW, call *weventpoll(3W)* to get scroll bar event information.

Each step is discussed in more detail next.

Call signal(2)

Setting up a signal handler is discussed in detail in the "Event Detection" chapter. You should understand the concepts in that chapter before proceeding.

```
#include        <stdio.h>
#include        <signal.h>
    :
void    sb_events();            /* signal handler for elevator & arrow events */
void    pollevents();           /* routine to poll elevator & arrow events */
    :
/*
 * Call signal(2) to set up the signal handler; then set the event mask:
 */
        if (signal(SIGWINDOW, sb_events) == -1)
        {
                fprintf(stderr, "Error setting up signal handler.\n");
                exit(1);
        }
    :
```

Call wsetsigmask(3W)

After setting up the signal handler, the program should call *wsetsigmask(3W)* to ensure that SIGWINDOW is sent when the desired scroll bar events occur. To catch arrow events, a program should set the **EVENT_SB_ARROW** bit in the *event mask* passed to *wsetsigmask*; to catch elevator events, the program should set the **EVENT_ELEVATOR** bit. The following code segment sets a window's event mask so the window will send SIGWINDOW when arrow and elevator events occur.

```
#include        <stdio.h>
#include        <window.h>
    :
int     gwfd;                       /* graphics window file descriptor */
int     events;                     /* event mask */
    :
/*
 * Set the window's event mask to catch arrow and elevator events:
 */
        events = EVENT_SB_ARROW | EVENT_ELEVATOR;
        if (wsetsigmask(gwfd, events) == -1)
        {
                fprintf(stderr, "Error setting event mask.\n");
                exit(1);
        }
    :
```

Enable User Mode

After the signal handler is set up and the window's event mask is set appropriately, the program should call *wscroll_set* to enable user mode for the arrows and/or elevators. The following code segment enables user mode and defines scroll bar scale information.

```
#include        <stdio.h>
#include        <window.h>
    :
int     value = 0;
int     min = -100;
int     max =  100;
int     size = 20;
int     which = SCROLLBAR_V | SCROLLBAR_H;
    :
int     gwfd;                       /* graphics window file descriptor */
int     mode;                       /* parameters to wscroll_set */
    :
```

```
/*
 * Enable vertical and horizontal scroll bars and arrows.
 */
        mode = SCROLLBAR_ELEVATOR | SCROLLBAR_ARROWS;
        mode |= SCROLLBAR_USERMODE | SCROLLBAR_SCALE;
        if (wscroll_set(gwfd, which, mode, value, min, max, size) == -1)
        {
                fprintf(stderr, "Error setting scroll bar info.\n");
                exit(1);
        }
    .
    .
    .
```

Wait for SIGWINDOW

Once all the setup is finished, the program can wait for the SIGWINDOW signal. The following code segment waits for SIGWINDOW in a `while` loop. The variable `got_sigwindow` is a global boolean variable which is set by the signal handler `sb_events` when SIGWINDOW is received. If `sb_events` == 1, then the program executes `pollevents`, which determines what event occurred and does the appropriate action. This is shown more thoroughly in the final code example in this section.

```
int     got_sigwindow = 0;  /* global boolean, set by signal handler when
                               SIGWINDOW is received */
    .
    .
    .
main(argc, argv)
    .
    .
    .
/*
 * Now wait for SIGWINDOW.
 */
        while (1)
        {
                if (got_sigwindow)
                {
                        got_sigwindow = 0;
                        pollevents(gwfd);
                        continue;       /* go back in case signals occurred
                                           before "continue" was reached */
                }
                pause();
        }
    .
    .
    .
```

Example

The following program enables both the vertical and horizontal elevators and arrows in a graphics window's border. The program assumes the name of the graphics window is passed as the first positional parameter; i.e., the program is called as:

```
program grwindow
```

where *grwindow* is the name of the graphics window in which to enable elevators and arrows, and **program** is the name of the program after it is compiled.

When the user activates an arrow or scroll bar in *grwindow*'s border, the program displays a message telling what type of event (arrow or elevator) occurred along with descriptive information about the event. If the user moves an elevator, the program re-positions the elevator accordingly.

```
#include        <stdio.h>
#include        <window.h>
#include        <signal.h>
int     events = EVENT_ELEVATOR | EVENT_SB_ARROW;
int     got_sigwindow = 0;      /* boolean, set if SIGWINDOW received */

/*
 * Define elevator location (value) and scale information globally:
 */
int     value = 0;
int     min = -100;
int     max =  100;
int     size = 20;
```

```
main(argc, argv)
int     argc;
char    *argv[];
{
int     wmfd;                   /* window manager file descriptor */
int     gwfd;                   /* graphics window file descriptor */
char    wname[WINNAMEMAX];      /* path name of window device interface */
int     mode;                   /* parameters to wscroll_set */
int     which;                  /* determines which scroll bars to affect */
int     est_wm_com();           /* starts wm communication */
int     term_wm_com();          /* stops wm communication */
int     est_gr();               /* starts graphics window communication */
int     term_gr();              /* stops graphics window communication */
void    sb_events();            /* sig handler for elevator & arrow events */
void    pollevents();           /* routine to poll elevator & arrow events */

/*
 * Start communication with the window manager to get "wmfd":
 */
        if ((wmfd = est_wm_com()) == -1) {
                fprintf(stderr, "Error starting wm communication.\n");
                exit(1);
        }

/*
 * Build the window's path name and start communication:
 */
        if (wmpathmake("WMDIR", argv[1], wname) == -1) {
                fprintf(stderr, "Error building window path name.\n");
                exit(1);
        }
        if ((gwfd = est_gr(wmfd, wname)) == -1) {
                fprintf(stderr, "Error starting window communication.\n");
                exit(1);
        }

/*
 * Call signal(2) to set up the signal handler; then set the event mask:
 */
        if (signal(SIGWINDOW, sb_events) == -1) {
                fprintf(stderr, "Error setting up signal handler.\n");
                exit(1);
        }
        if (wsetsigmask(gwfd, events) == -1) {
                fprintf(stderr, "Error setting event mask.\n");
                exit(1);
        }
/*
 * Enable vertical and horizontal scroll bars and arrows.
 */
        mode = SCROLLBAR_ELEVATOR | SCROLLBAR_ARROWS;
        mode |= SCROLLBAR_USERMODE | SCROLLBAR_SCALE;
        which = SCROLLBAR_V | SCROLLBAR_H;
        if (wscroll_set(gwfd, which, mode, value, min, max, size) == -1) {
                fprintf(stderr, "Error setting scroll bar info.\n");
                exit(1);
        }

/*
 * Now wait for SIGWINDOW.
 */
        while (1) {
                if (got_sigwindow) {
                        got_sigwindow = 0;
                        pollevents(gwfd);
                        continue;       /* go back in case signals occurred
                                           before "continue" was reached */
                }
                pause();
        }
}

/*
 * Signal catcher for scroll bar events.
 *      Sets "got_sigwindow" to 1 and re-enables itself to catch SIGWINDOW.
 */
void    sb_events(signum)
int     signum;
{

        if (signum != SIGWINDOW) {
                fprintf(stderr, "Wrong signal number received.\n");
                exit(1);
        }

        printf("\nSIGWINDOW received!\n");
        got_sigwindow = 1;
        /* Re-enable SIGWINDOW to catch events should they occur again. */
        if (signal(SIGWINDOW, sb_events) == -1) {
                fprintf(stderr, "signal failed in sb_events.\n");
                exit(1);
        }
}
```

```
/*
 * Routine to poll for elevator and arrow events;
 *      called when global "got_sigwindow" is set to 1.
 */
void    pollevents(gwfd)
int     gwfd;
{
int     event_mask;
int     count, x,y;
int     mode;
char    *elev_str;

        do {
                event_mask = events;
                if (weventpoll(gwfd, &event_mask, &count, &x,&y) == -1) {
                        fprintf(stderr, "Weventpoll failed.\n");
                        exit(1);
                }

                if (event_mask & EVENT_ELEVATOR)
                {

                        switch (x)
                        {
                                case SCROLLBAR_V :
                                        elev_str = "VERTICAL";
                                        break;
                                case SCROLLBAR_H :
                                        elev_str = "HORIZONTAL";
                                        break;
                                default:
                                        elev_str = "INVALID";
                        }

                        printf("%s elevator activated.\n", elev_str);
                        printf("Is now at location %d.\n\n", y);

        /*
         * Call wscroll_set again to display elevator at its new location:
         */
                        mode = SCROLLBAR_ELEVATOR | SCROLLBAR_ARROWS;
                        mode |= SCROLLBAR_USERMODE;
                        if (wscroll_set(gwfd, x, mode, y) == -1) {
                                fprintf(stderr, "Wscroll_set failed.\n");
                                exit(1);
                        }
                        if (y == min) {
                                printf("Elevator at 'min' position.\n");
                                printf("\n\n***FINISHED***\n");
                                exit(0);
                        }
                }

                else if (event_mask & EVENT_SB_ARROW) {
                        printf("Arrow clicked.\n");
                        printf("Sum of horizontal arrows is %d.\n", x);
                        printf("Sum of vertical arrows is %d.\n\n", y);
                }
        } while (event_mask);
}
```

Graphics Window Hotspots 9

Graphics window **hotspots** allow a program to be signalled when the locator enters or exits a rectangular area within the graphics window's raster, or when the user presses a button within the rectangle. This chapter describes how to use hotspot routines.

By calling hotspot routines, a program can:

- create a hotspot
- change a hotspot's characteristics
- delete a hotspot
- get hotspot event information.

Concepts

This section discusses hotspot concepts. You should read this section before using hotspot routines.

What Is a Hotspot?

A hotspot is an invisible rectangular area on a graphics window's raster. The location and size of the hotspot are defined via hotspot routines.

A hotspot event (`EVENT_HOTSPOT`) occurs when the user **activates** a hotspot. Any of the following actions can activate a hotspot:

- moving the locator into or out of a hotspot
- pressing the [Select] key while the locator is within a hotspot
- pressing a button (1 through 8) while the locator is within a hotspot.

Using hotspot routines, a program can control *which* action or actions activate the hotspot. For example, a hotspot could be activated only when the locator enters or exits the hotspot, or when the user presses button 2.

A window can have up to 128 hotspots. Each hotspot has a unique **hotspot id**, returned when the hotspot is created. Hotspot routines use the hotspot id to identify which hotspot to change, inquire, or delete.

Hotspot Event Detection

Hotspot events can be detected two ways: via graphics window input routines or event detection routines. The "Graphics Window Input" chapter discusses the use of graphics window input routines; event detection is covered in the "Event Detection" chapter and later in this chapter.

Creating a Hotspot

The *whotspot_create(3W)* routine creates a hotspot for a graphics window.

Procedure

To create a hotspot, call *whotspot_create*; its syntax is:

```
whotspot_create(fd, bmask, x,y,w,h, event_byte)
```

return value (hotspot id)

If *whotspot_create* successfully creates the hotspot, then the hotspot's *hotspot id* is returned. The hotspot id is an integer which identifies the hotspot for the window. Other hotspot routines require the hotspot id as a parameter.

If *whotspot_create* cannot create the hotspot, then $^{-}1$ is returned, and *errno(2)* may be set.

fd

The *fd* parameter is an integer file descriptor returned from starting communication with the graphics window.

bmask

Bmask is an integer bit mask defining which activity or activities should activate the hotspot. Each bit in *bmask* represents an activity, such as moving the locator into or out of a hotspot. To activate the hotspot when a particular activity occurs, simply set the corresponding bit in *bmask*.

Table 9-1 defines the valid activities and the constant values for the bits representing the activities. The constants are defined in *window.h*.

Table 9-1. Activity Bit Definitions

Constant (Bit)	Activity
HS_MASK_BUTTON*n*	Locator button *n* pressed while the locator is within the hotspot. *n* must be between 1 and 8. For example, the constant for button 2 is HS_MASK_BUTTON2.
HS_MASK_SELECT	The Select key pressed while the locator is within the hotspot.
HS_MASK_ENTEREXIT	The locator entered or exited the hotspot.

x,y,w,h

The *x* and *y* parameters define *where* to put the hotspot. The upper-left corner of the hotspot is placed at the specified coordinates. The origin (*0,0*) is the upper-leftmost pixel of the raster. The *w* and *h* parameters give the hotspot's pixel width and height, respectively.

event_byte

The *event_byte* parameter specifies what should happen when the user activates the hotspot. Table 9-2 describes the acceptable values for *event_byte*, as defined in *window.h.*

Table 9-2. Valid Values for event_byte (from window.h)

Value	Description
K_MOVE_ST	Start an interactive window move operation on the window. When the window move is finished, the EVENT_MOVE event will occur for the window.
K_SIZE_LR_ST	Start an interactive window size operation on the window. When the size operation is finished, the EVENT_SIZE event will occur for the window.
K_POPUP_ST	Activate a system or user-defined pop-up menu for the window. The user can then abort the menu, or make a selection.
K_USER_HS to K_USER_HS + 127	If you do *not* want to move or size the window or generate a system pop-up menu when the hotspot is activated, then set *event_byte* to one of these values. When set to one of these values, *event_byte* identifies the hotspot to event detection and graphics window input routines. Typically, *event_byte* is set to a value corresponding to the hotspot. For example, for the first hotspot, set *event_byte* to K_USER_HS; for the second hotspot, set *event_byte* to K_USER_HS + 1; for the third, K_USER_HS + 2; and so on. Then, when the program calls event detection or graphics window input routines to get hotspot information, the *event_byte* parameter is returned, identifying which hotspot was activated.

Overlapping Hotspots

Hotspot rectangles can overlap. When they overlap, they form a stack within the window. When an overlapping hotspot is created, it is placed on top of the stack. The only way to change a hotspot's location within the stack is to delete it and create it again, forcing it to the top.

When hotspots overlap, the window manager uses the following rules for hotspot enter-exit semantics:

- If the locator moves from a hotspot that is lower in the hotspot stack to a hotspot that is higher in the hotspot stack, then the lower hotspot is exited and the higher hotspot is entered.
- If the locator moves from a higher into a lower hotspot, then the higher hotspot is exited and the lower hotspot is entered.

For example, suppose two hotspots (**A** and **B**) are set up so that **B** is on top of **A** (see Figure 9-1). When the locator moves into hotspot **A**, then hotspot **A** is entered. When the locator moves into hotspot **B**, then hotspot **A** is exited and hotspot **B** is entered. Then when the locator moves out of **B** and into **A**, hotspot **B** is exited and hotspot **A** is entered.

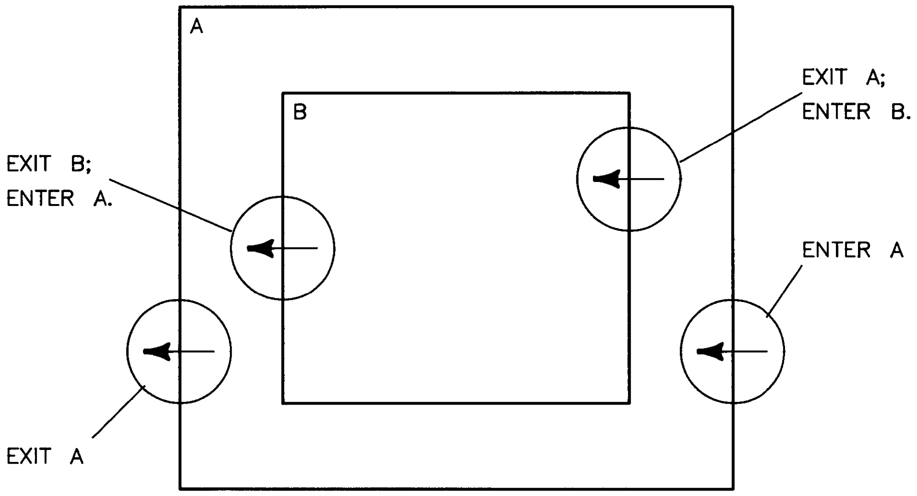


Figure 9-1. Overlapping Hotspot Example

Precautions

- A hotspot can be activated, regardless of whether or not the hotspot's window is selected.
- Pressing an enabled button over a hotspot will activate the hotspot but will *not* select the window. For example, if a hotspot's *bmask* is set to **HS_MASK_BUTTON1** and the window is not currently selected, then pressing button 1 over the hotspot will activate the hotspot, but will *not* select the window, as would normally happen.
- Only one hotspot at a time can be activated. When the locator is over an area which is overlapped by two or more hotspots, only the topmost hotspot in the stack can be activated.
- When a hotspot is activated via a button press, the activation occurs when the button is pressed down—*not* when the button is released. The upstroke of the button cannot be detected.
- A button press over a hotspot whose *bmask* includes that button will activate the hotspot, but will not select the window.
- If a hotspot is completely obscured (covered) by another hotspot or hotspots, then it **cannot** be activated.

Example

The following code segment sets up two hotspots. The hotspot id for each hotspot is stored in the array named **hotspots**.

The first hotspot is located at *0,0*, is *50* pixels wide by *100* pixels high, is activated by button 1 or 2 being pressed within the hotspot, and on being activated, a window move operation is generated. The second hotspot is activated by the locator entering or exiting the hotspot, is located at *50,0*, is *20* by *20* pixels, and on being activated, its *event_byte* parameter is set to **K_USER_HS**.

```
#include  <window.h>
   :
main()
{
int     hotspots[128];     /* holds up to 128 hotspot id's per window */
int     gwfd;              /* graphics window file descriptor */
int     bmask,             /* activation bit mask */
        x,y,w,h;           /* hotspot location and size information */

int     event_byte; /* what-to-do on hotspot activation */
              :
/*
```

```
 * Start communication with the window, obtaining "gwfd".
 */
          :
/*
 * Create the first hotspot:
 */
        bmask = HS_MASK_BUTTON1 | HS_MASK_BUTTON2;
        x = y = 0;
        w = 50; h = 100;
        event_byte = K_MOVE_ST;
        hotspots[0] = whotspot_create(gwfd, bmask, x,y,w,h, event_byte);
        if (hotspots[0] < 0)
        {
                perror("whotspot_create gwfd");
                exit(1);
        }
/*
 * Create the second hotspot:
 */
        bmask = HS_MASK_ENTEREXIT;
        x = 50; y = 0;
        w = h = 20;
        event_byte = K_USER_HS;
        hotspots[1] = whotspot_create(gwfd, bmask, x,y,w,h, event_byte);
        if (hotspots[1] < 0)
        {
                perror("whotspot_create gwfd");
                exit(1);
        }
          :
}
```

Changing a Hotspot's Characteristics

Once a hotspot is created, it can be modified. For example, it may be desirable to change the width and height, or the location of a hotspot after it is activated. Two routines are used to change a hotspot's characteristics: *whotspot_get(3W)* and *whotspot_set(3W).*

Procedure

To modify a hotspot, call *whotspot_set* to change the hotspot's parameters to new values. A program can also optionally call *whotspot_get* to get information about a hotspot.

whotspot_set

The *whotspot_set* routine changes a hotspot's parameters; its syntax is:

```
whotspot_set(fd, hotspot_id, bmask, x,y,w,h, event_byte)
```

The *hotspot_id* parameter tells *whotspot_set* which hotspot to modify. Set this parameter to the hotspot id returned when the hotspot was created via *whotspot_create.*

The *fd*, *bmask*, *x,y,w,h*, and *event_byte* parameters are identical in function to those of *whotspot_create.* That is, *fd* is the file descriptor of the window, and *bmask*, *x,y,w,h*, and *event_byte* define the hotspot's activity bit mask, location, size, and event byte, respectively.

whotspot_get

The *whotspot_get* routine's syntax is:

```
whotspot_get(fd, hotspot_id, bmask, x,y,w,h, event_byte)
```

Fd is the file descriptor returned from starting communication with the window.

The *hotspot_id* parameter is an integer id number, returned when the hotspot was created via *whotspot_create.* This parameter lets *whotspot_get* know which hotspot to get information for.

On return from *whotspot_get*, *bmask*, *x*, *y*, *w*, *h*, and *event_byte* point to integers containing the hotspot's activation bit mask, location, size, and event_byte respectively.

Precautions

When a hotspot is changed to be activated on entering or exiting the hotspot—i.e., when its *bmask* parameter is changed to HS_ENTEREXIT—the hotspot will be activated according to the rules in Table 9-3.

Table 9-3. Activating a Hotspot When ENTEREXIT Is Changed

ENTEREXIT Set Previously	Pointer in Hotspot Before Change	Pointer in Hotspot After Change	Hotspot Entered or Exited
No	No	Yes	Enter
No	Yes	Yes	Enter
Yes	No	Yes	Enter
Yes	Yes	No	Exit

Example

The following code segment creates a hotspot, modifies the hotspot, and finally, gets and displays the hotspot's parameters.

```
#include <stdio.h>
#include <window.h>
main()
{
int     gwfd;           /* graphics window file descriptor */
int     hotspot_id;
int     bmask, x,y,w,h, event_byte;

                  :
/*
 * Create the hotspot:
 */
    bmask = HS_MASK_BUTTON1 | HS_MASK_SELECT;
    x = y = 0; w = h = 50;
    event_byte = K_POPUP_ST;
    hotspot_id = whotspot_create(gwfd, bmask, x,y,w,h, event_byte);
    if (hotspot_id < 0) {
        perror("whotspot_create gwfd");
        exit(1);
    }
/* Modify the hotspot's width and height: */
    w = 100; h = 75;
    if (whotspot_set(gwfd, hotspot_id, bmask, x,y,w,h, event_byte) < 0) {
        perror("whotspot_set gwfd");
        exit(1);
    }

/* Get and display the hotspot's parameters: */
  if (whotspot_get(gwfd, hotspot_id, &bmask, &x,&y,&w,&h, &event_byte) < 0) {
        perror("whotspot_get gwfd"); exit(1);
  }
  printf("HOTSPOT parameters for hotspot id = %d\n\n", hotspot_id);
  printf("  bmask      = %d\n", bmask);
  printf("  x,y, w,h   = %d,%d, %d,%d\n", x,y, w,h);
  printf("  event_byte = %d\n", event_byte);
         :
}
```

Deleting a Hotspot

The *whotspot_delete(3W)* routine deletes a hotspot from a window.

Procedure

To delete a hotspot, simply call *whotspot_delete*; its syntax is:

```
whotspot_delete(fd, hotspot_id)
```

Fd is the file descriptor returned from starting communication with the graphics window. The *hotspot_id* parameter is the hotspot id returned when the hotspot was created via *whotspot_create*. After calling this routine, the hotspot will no longer exist.

Precautions

Hotspots are *not* activated when they are deleted. For example, a program doesn't get a hotspot exit if the locator is over a hotspot when the hotspot is deleted. **However**, if the deleted hotspot is over another hotspot, then the hotspot underneath the deleted one *will* get a hotspot enter activation (if enabled in the hotspot underneath).

Detecting Hotspot Events

Once a hotspot is created, a program can get hotspot event information via either event detection routines or graphics window input routines. This section discusses the use of event detection routines; graphics window input routines are discussed in the "Graphics Window Input Routines" chapter.

Procedure

To detect hotspot events, a program would typically do the following:

1. Create the hotspot(s) via *whotspot_create(3W)*.
3. Call *signal(2)* to set up a signal handler for SIGWINDOW, and wait for a hotspot event to generate SIGWINDOW.
3. Call *wsetsigmask(3W)* to enable hotspot events.
4. On receiving SIGWINDOW, call *weventpoll(3W)* to get hotspot event information.

Each step is discussed in more detail below.

Call whotspot_create

To create hotspots in a window, call *whotspot_create*, as described in the previous section "Creating a Hotspot." Once a hotspot is created, it can be activated and can send SIGWINDOW to any process that has enabled it, as discussed next.

Call signal(2)

To catch SIGWINDOW for hotspot (or any other) window events, a program must set up a signal handler via *signal(2)*. The following code segment sets up a signal handler named `hs_events`.

```
#include        <stdio.h>
#include        <signal.h>
#include        <window.h>
     :
int     gwfd;   /* graphics window file descriptor */
void    hs_events();            /* signal handler for hotspot events */
```

```
/*
 * Call signal(2) to set up the signal handler; then set the event mask:
 */
        if (signal(SIGWINDOW, hs_events) == -1)
        {
                fprintf(stderr, "signal failed.\n");
                exit(1);
        }

    :
/*
 * This signal handler set a global variable "got_sigwindow" to 1
 * upon recieving SIGWINDOW.  When this variable is "1", the program
 * knows it has received SIGWINDOW and can act accordingly.
 */
void    hs_events(signum)
int     signum;

{
        if (signum != SIGWINDOW)
        {
                fprintf(stderr, "Wrong signal number received.\n");
                exit(1);
        }
        printf("\nSIGWINDOW received!\n");
        got_sigwindow = 1;

        /* Re-enable SIGWINDOW to catch events should they occur again. */
        if (signal(SIGWINDOW, hs_events) == -1)
        {
                fprintf(stderr, "signal failed in hs_events.\n");
                exit(1);
        }
}
```

Call wsetsigmask(3W)

After setting up the signal handler, the program should call *wsetsigmask* to ensure that the SIGWINDOW signal is sent when hotspot events occur. *Wsetsigmask* sets the event mask for a window. Every bit in the event mask corresponds to a window system event. To catch hotspot events, set the `EVENT_HOTSPOT` bit in the event mask, and call *wsetsigmask*. For example, the following code enables hotspot events the window whose file descriptor is `gwfd`:

```
#include        <stdio.h>
#include        <window.h>
    :
        if (wsetsigmask(gwfd, EVENT_HOTSPOT) == -1)
        {
                fprintf(stderr, "wsetsigmask failed.\n");
                exit(1);
        }
```

Call weventpoll(3W)

On receiving SIGWINDOW, the program can determine which event caused SIGWINDOW by calling *weventpoll(3W)*.

For hotspot events, *weventpoll* sets its *x* parameter to the *event_byte* parameter of the activated hotspot. For example, suppose a hotspot was the second hotspot created in a window, and its *event_byte* was set to `K_USER_HS + 1` when the hotspot was created. Then if the hotspot is activated, *wevent_poll* will return the value `K_USER_HS + 1` in the *x* parameter.

On return from *weventpoll*, the *y* parameter is set to the cause that activated the hotspot. Table 9-4 defines the valid return values for *y*; these values are defined in *window.h.*

Table 9-4

Cause	Description
EC_BUTTON*n*	Button *n* was pressed while the locator was within the hotspot; $1 \leq n \leq 8$. *For example,* EC_BUTTON1 means that button 1 activated the hotspot.
EC_SELECT	The Select key was pressed while the locator was within the hotspot.
EC_ENTER	The locator entered the hotspot. That is, the locator crossed over the hotspot's border into the hotspot.
EC_EXIT	The locator exited the hotspot. That is, the locator crossed over the hotspot's border out of the hotspot.

Example

The following program creates two hotspots in a graphics window and draws the hotspots' borders using Starbase graphics routines. The first hotspot is underneath the second hotspot. The first hotspot is activated by button one, the Select button, and entering or exiting the hotspot. The second hotspot is activated by button two and entering or exiting the hotspot. Whenever a hotspot is activated, the program displays a message to standard output describing what which hotspot was activated and what caused the activation.

```
#include        <stdio.h>
#include        <signal.h>
#include        <window.h>
#include        <starbase.c.h>

/*
 * Global variables:
 */
int     got_sigwindow = 0;      /* boolean, set if SIGWINDOW received */
struct  hs_info {               /* hotspot ids and event bytes structure */
                int     id;
                int     eb;
} hs[2];

main(argc, argv)
int     argc;
char    *argv[];
{

int     wmfd;   /* window manager file descriptor */
int     gwfd;   /* graphics window file descriptor */
char    wname[WINNAMEMAX];
int     x,y,w,h,dx,dy,rw,rh;    /* window size information variables */
int     bmask;                  /* hotspot bmask parameter */
int     hsx, hsy;               /* hotspot location */
int     hsw, hsh;               /* hotspot width and height */
void    hs_events();            /* signal handler for hotspot events */
void    pollevents();           /* routine to poll hotspot events */

/*
 * Start window manager communication
 */
        if ((wmfd = est_wm_com()) == -1)
        {
                fprintf(stderr, "est_wm_com failed.\n");
                exit(1);
        }
```

```
/*
 * Start window communication:
 */
        if (wmpathmake("WMDIR", argv[1], wname) == -1)
        {
                fprintf(stderr, "wmpathmake failed.\n");
                exit(1);
        }
        if ((gwfd = est_gr(wmfd, wname)) == -1)
        {
                fprintf(stderr, "est_gr failed.\n");
                exit(1);
        }

/*
 * Get the size of the window so that hotspots will be one-fourth the
 *      size of the window's raster:
 */
        if (wgetcoords(gwfd, &x,&y,&w,&h, &dx,&dy,&rw,&rh) == -1)
        {
                fprintf(stderr, "wgetcoords failed.\n");
                exit(1);
        }
        if (wsize(gwfd, rw, rh) == -1)   /* make window full size */
        {
                fprintf(stderr, "wsize failed.\n");
                exit(1);
        }

/*
 * Draw the outline for the hotspots:
 */
        write_enable(gwfd, -1);
        drawing_mode(gwfd, 3);
        background_color(gwfd, 0);
        clear_view_surface(gwfd);
        perimeter_color_index(gwfd, 1);
        interior_style(gwfd, INT_HOLLOW, TRUE);
        dcrectangle(gwfd, 0,0, rw /2 - 1 , rh / 2 - 1);  /* hs 0 */
        hsx = rw / 4; hsy = rh / 4;
        hsw = rw / 2; hsh = rw / 2;
        dcrectangle(gwfd, hsx,hsy, hsx + hsw - 1, hsy + hsh - 1); /* hs 1 */
        make_picture_current(gwfd);
```

```
/*
 * Set hotspot values for hotspot 0:
 */
        bmask = HS_MASK_BUTTON1 | HS_MASK_SELECT | HS_MASK_ENTEREXIT;
        hsx = hsy = 0;
        hsw = rw / 2; hsh = rh / 2;
        hs[0].eb = K_USER_HS;
        if ((hs[0].id = whotspot_create(gwfd,
                        bmask, hsx,hsy, hsw,hsh, hs[0].eb)) == -1)
        {
                fprintf("whotspot_create failed on hotspot 0.\n");
                exit(1);
        }

/*
 * Set hotspot values for hotspot 1:
 */
        bmask = HS_MASK_BUTTON2 | HS_MASK_ENTEREXIT;
        hsx = rw / 4; hsy = rh / 4;
        hsw = rw / 2; hsh = rw / 2;
        hs[1].eb = K_USER_HS;
        if ((hs[1].id = whotspot_create(gwfd,
                        bmask, hsx,hsy, hsw,hsh, hs[1].eb)) == -1)
        {
                fprintf("whotspot_create failed on hotspot 1.\n");
                exit(1);
        }

/*
 * Call signal(2) to set up the signal handler; then set the event mask:
 */
        if (signal(SIGWINDOW, hs_events) == -1)
        {
                fprintf(stderr, "signal failed.\n");
                exit(1);
        }
        if (wsetsigmask(gwfd, EVENT_HOTSPOT) == -1)
        {
                fprintf(stderr, "wsetsigmask failed.\n");
                exit(1);
        }
```

```
/*
 * Now wait for SIGWINDOW.
 */
        while (1)
        {
                if (got_sigwindow)
                {
                        got_sigwindow = 0;
                        pollevents(gwfd);
                        continue;       /* go back in case signals occurred
                                           before "continue" was reached */
                }
                pause();
        }
}

/*
 * Signal handler for hotspot events.  Sets global got_sigwindow to 1.
 */
void    hs_events(signum)
int     signum;
{
        if (signum != SIGWINDOW)
        {
                fprintf(stderr, "Wrong signal number received.\n");
                exit(1);
        }
        printf("\nSIGWINDOW received!\n");
        got_sigwindow = 1;

        /* Re-enable SIGWINDOW to catch events should they occur again. */
        if (signal(SIGWINDOW, hs_events) == -1)
        {
                fprintf(stderr, "signal failed in hs_events.\n");
                exit(1);
        }
}

void    pollevents(gwfd)
int     gwfd;
{
int     event_mask;
int     count, x,y;
char    *cause;
int     hs_num;
```

```
        do {
                event_mask = EVENT_HOTSPOT;
                if (weventpoll(gwfd, &event_mask, &count, &x,&y) == -1)
                {
                        fprintf(stderr, "Weventpoll failed.\n");
                        exit(1);
                }

                if (event_mask == EVENT_HOTSPOT)
                {
                        if (x == hs[0].eb)
                                hs_num = 0;
                        else if (x == hs[1].eb)
                                hs_num = 1;
                        else
                                hs_num = -1;

                        switch (y)
                        {
                                case EC_BUTTON1 :
                                        cause = "button one";
                                        break;
                                case EC_BUTTON2 :
                                        cause = "button two";
                                        break;
                                case EC_SELECT :
                                        cause = "Select button";
                                        break;
                                case EC_ENTER :
                                        cause = "entering hotspot";
                                        break;
                                case EC_EXIT :
                                        cause = "exiting hotspot";
                                        break;
                                default:
                                        cause = "unknown cause";
                                        break;
                        }

                        printf("Hostpot %d (id: %d) activated by %s.\n",
                                        hs_num, hs[hs_num].id, cause);
                }
        } while (event_mask);
}
```

Notes

User-Definable Menus 10

You can create your own pop-up menus by using window menu routines. These menus are much like the system pop-up menu: by using the locator, users of your applications can select items from these menus; your programs can then determine which item(s) the user selected and do some appropriate action. This chapter describes the use of window menu routines; the following topics are covered:

- concepts essential to using menu routines
- creating a menu
- activating a menu
- adding menu items
- getting menu selections
- deleting a menu

Concepts

Each window can have up to 24 user-defined menus. Each menu has a unique **menu id**, returned when the menu is created.

Although a window can support several menus, only one menu at a time can be **displayed** per window. Only when a menu is displayed can the user select items from the menu.

To be displayed, a menu must first be **activated**. Several menus can be activated at the same time, and window routines allow a program to specify which button press(es) should cause the menu to be displayed.

Each menu is comprised of **menu items**. Menu items are added to menus via menu routines. Each menu item is identified by an **item id**, returned when the item is added to the menu.

Menu items can be either **selectable** or **non-selectable**. If event detection is enabled for a window, then selectable items are capable of generating a menu event; that is, they can be selected from a pop-up menu by clicking the locator over the menu item.

If the locator is clicked over a selectable menu item of the active displayed menu, the SIGWINDOW interrupt will be sent to all processes which requested window signals via *signal(2)* and *weventpoll(3W). The weventpoll* routine can be used to determine that a menu item selection was made (the `EVENT_MENU` event mask indicates that a menu selection was made); menu routines can be used to get the menu and item id's for the selected item.

If the locator is clicked over a non-selectable item, the menu is aborted, and a value of -1 is returned as the selected item id. A menu item should be made non-selectable if you don't wish the menu item to be selectable from the pop-up menu. For example, on the system pop-up menu, the window label appears on the menu but is not selectable; invalid options, such as the *Top* option is already the top window in the display stack, should also be made non-selectable.

You can define your own **menu button mask** which tells the window manager which locator button(s) should invoke a pop-up menu in a window. This mask is specified when the menu is created.

The window manager determines whether to invoke a user-defined or system pop-up menu by looking at the WMIUICONFIG window system environment variable and the menu button masks for the active menus in each window.

When a locator button is pressed, the window manager looks at the lower eight bits of the WMIUICONFIG environment variable. If the bit corresponding to the button is set, then a system pop-up menu will be invoked under either of the following conditions:

- The locator is positioned over the desk top **and** the pop-up menu is enabled over the desk top (i.e., WMIUICONFIG is **not** logically ORed with 0x040000). By default the pop-up menu is enabled over the desk top. In this case, the system pop-up menu is invoked for the selected window.
- The locator is positioned over a window's border **and** the pop-up menu is enabled over window borders (i.e., WMIUICONFIG is **not** logically ORed with 0x020000). By default the pop-up menu is enabled over window borders. The system pop-up menu is invoked for the window whose label the locator was clicked over.

If none of the bits in WMIUICONFIG corresponds to the button that was pressed, then the window manager looks at the menu button mask for the active menus in the various windows. Whether or not a user-defined menu is invoked depends on the position of the locator when the button is pressed.

- If the locator is positioned over the desk top, then a pop-up menu will be invoked for the selected window if all the following are true:
    a. The selected window has an activated pop-up menu,
    b. A bit corresponding to the pressed button is set in the active menu's button mask, and
    c. The menu is "marked" to pop up automatically when the button is pressed. (Whether or not a menu is marked to pop up is specified when the menu is activated.)
- If the locator is positioned over a window's border area (not over manipulation areas) or user area, then a pop-up menu will be invoked for the window if all the following are true:
    a. The window has an activated pop-up menu,
    b. A bit corresponding to the pressed button is set in the window's menu button mask, and
    c. The menu is "marked" to pop up automatically when the button is pressed.

Creating a Menu

The *wmenu_create(3W)* routine allocates the resources necessary for a user-defined pop-up menu. A menu's button mask is also specified when the menu is created. The button mask determines which locator buttons invoke the pop-up menu.

Procedure

To create a pop-up menu, call *wmenu_create*. It will create the menu and return the menu's menu id. The syntax for this routine is:

`wmenu_create(`*wfd, cbits, button_mask, parent_menuid, parent_item*`)`

You specify which window to create the menu for by passing the file descriptor of the window's device interface, *wfd*.

The *cbits* parameter defines the menu's type. Currently, only the pop-up menu type is supported, so *cbits* should always be set to `MENU_POPUP`.

The *button_mask* parameter defines which button(s) invoke this pop-up menu and which button(s) select an item when clicked over a menu item. The lower eight bits (least-significant byte) define the buttons that invoke the pop-up menu; the next eight bits define the buttons that select a menu item.

The least-significant bit of each button mask corresponds to button one; the second bit corresponds to button two; and so on. The most-significant bit of each mask (bit eight) corresponds to the select key. For example, if you want button two to invoke a menu and both buttons one and two to make item selections, then set this mask to `0x0302`.

You should always set the *parent_menuid* and *parent_item* parameters to `MENU_NOPARENT`.

Precautions

Note that calling this routine does not activate the menu, nor does it cause the menu to have any items. For information on activating the menu, see the section "Activating a Menu." For details on adding menu items, see the section "Adding Menu Items."

Example

The following call to *wmenu_create* creates a menu; the menu will be invoked when button number two (for example, the rightmost mouse button) is pressed; either button one, two, or the [Select] key will select a menu item. The menu's id is stored in the `menu_id` variable.

```
menu_id = wmenu_create(wfd, MENU_POPUP, 0x8202,
                        MENU_NOPARENT, MENU_NOPARENT);
```

Adding Menu Items

The *wmenu_item(3W)* routine adds menu items to a menu; it also can be used to change items.

Procedure

To add items to a menu, call *wmenu_item*; its syntax is:

```
wmenu_item(wfd, menuid, itemno, type, disp_sel, type_struct)
```

The *wfd* and *menuid* parameters identify the window and menu for which menu items are added (or changed).

If you're adding a new item to a menu, then *itemno* should be set to `MENU_NEWITEM`. Items are added to the menu sequentially; i.e., the first item added is the topmost menu item displayed in the menu. If you're changing an item, you supply the item id that was returned when the item was added.

The *type* parameter defines the type of the menu item. Set this parameter to `MENU_STRING` if you want a text string to appear as a menu item. Currently, the only other valid value is `MENU_SEPARATOR` which causes a horizontal bar to appear; you can use this type to separate different sections of the menu.

The *disp_sel* parameter defines whether the item is selectable, how the item is displayed within the menu, and how the item will appear when the locator is positioned over it. This parameter is set by logically ORing the bits defined in *window.h.*

If the item is selectable, then the *disp_sel* should be logically ORed with `MENU_SELECTABLE`, i.e., the least-significant bit of this parameter should be set; if the parameter is not be selectable, then OR it with `MENU_NOTSELECTABLE`.

Two different modes can be used to display a menu item: normal or grey. On the system menu, valid options are displayed in normal mode (i.e., dark letters); invalid options are displayed in grey mode (i.e., grey letters). To display an item in normal mode, logically OR the *disp_sel* parameter with `MENU_DISPNORM`; to display it in grey mode, logically OR it with `MENU_DISPGREY`. If neither bit is set, the default is `MENU_DISPNORM`.

Two different modes can be used for tracking an item: inverse tracking and no tracking. With inverse tracking, a menu item is displayed in inverse video whenever the locator moves over it; with no tracking, the menu item does not invert when the locator moves over it. To set inverse tracking for an item, set the `MENU_TRACKINV` bit in *disp_sel*; for no tracking, set the `MENU_TRACKNOCHNG` bit. If neither bit is set, the default is `MENU_TRACKNOCHNG`.

The *type_struct* parameter depends on the type of the menu item. If the type is `MENU_STRING`, then this parameter should point to a null-terminated character string to display as the menu item. If the type is `MENU_SEPARATOR`, then this parameter should point to a single-character integer containing the pixel thickness (from 0 to 255) of the line (0 defaults to 2).

Example

The following code sample defines a menu titled *fruits*. Below the title is a horizontal separator bar. Below this are three options defined as follows:

Menu Item	**Description**
kumquat	A selectable menu item displayed in normal mode and inverted when the locator tracks over it.
carrot	A non-selectable menu item displayed in grey mode and not tracked when the locator moves over it.
kiwi	A selectable menu item displayed in normal mode and inverted when the locator tracks over it.

```
   :

if ((menu_id = wmenu_create(wfd, MENU_POPUP, 0x0202,
                       MENU_NOPARENT, MENU_NOPARENTITEM)) < 0)
{
     perror("wmenu_create failed");
     exit(1);
}
/*
 * Now add the menu items:
 *
 * First, assign the menu title.
 */
if ((title_id = wmenu_item(wfd, menu_id, MENU_NEWITEM, 0, MENU_STRING,
                        (MENU_DISPNORM | MENU_TRACKNOCHNG), "fruits")) < 0)
{
     perror("wmenu_item failed on fruits");
     exit(1);
}

/*
 * Next, put in the horizontal separator bar.
 */
if ((bar_id = wmenu_item(wfd, menu_id, MENU_NEWITEM, 0, MENU_SEPARATOR,
                        (MENU_DISPNORM | MENU_TRACKNOCHNG), NULL)) < 0)
{
     perror("wmenu_item failed on bar");
     exit(1);
}

/*
 * Assign the "kumquat" menu item.
 */
if ((item_id[0] = wmenu_item(wfd, menu_id, MENU_NEWITEM, 0, MENU_STRING,
                        (MENU_SELECTABLE | MENU_DISPNORM | MENU_TRACKINV),
                         "kumquat")) < 0)
{
     perror("wmenu_item didn't like the kumquat");
     exit(1);
}
```

```
/*
 * Assign the "carrot" menu item.
 */
if ((item_id[1] = wmenu_item(wfd, menu_id, MENU_NEWITEM, 0, MENU_STRING,
                        (MENU_DISPNORM | MENU_TRACKNOCHNG), "carrot")) < 0)
{
     perror("wmenu_item choked on the carrot");
     exit(1);
}

/*
 * Assign the "kiwi" menu item.
 */
if ((item_id[2] = wmenu_item(wfd, menu_id, MENU_NEWITEM, 0, MENU_STRING,
                        (MENU_SELECTABLE | MENU_DISPNORM | MENU_TRACKINV),
                         "kiwi")) < 0)
{
     perror("the kiwi gave wmenu_item indigestion");
     exit(1);
}
/*
 * Now activate the menu so that items can be selected from it:
 *   (This is discussed in detail in the next section.)
 */

   ⋮
```

Activating a Menu

In order for selections to be made from a pop-up menu, the menu must be activated, as defined in the concepts section. Activating a menu causes it to be displayable; only after a menu is activated and displayed can menu selection be made from it. The *wmenu_activate(3W)* routine activates a user-defined menu.

Procedure

To activate a menu, call *wmenu_activate*; its syntax is:

```
wmenu_activate(wfd, menuid, value)
```

The *wfd* parameter is the file descriptor for the window whose menu is to be activated.

The *menuid* parameter is the menu id of the menu to activate. This is the id returned when the menu was created.

The *value* parameter defines when and if to display the menu:

- If *value* is `MENU_ACT_DIS`, then the pop-up menu will be disabled; that is, it won't pop up when the user presses the appropriate locator button. You would use this if you wanted to disable the menu.
- If *value* is `MENU_ACT_AUTO`, then the pop-up menu will automatically pop up when the user presses the appropriate locator button, as defined in the menu button mask that was used when the menu was created. The menu pops up at the current locator position.
- If *value* is `MENU_ACT_INQ`, then the routine returns the current pop-up state of the menu, either `MENU_ACT_DIS` or `MENU_ACT_AUTO`.
- If *value* is `MENU_ACT_IM`, then the current pop-up state is ignored and the menu is automatically popped up at the current locator position.

Precautions

Be sure to add items to a menu before activating it. Otherwise, you'll have a null menu from which no item selections can be made.

Example

The following code segment creates a pop-up menu and activates it. The menu is activated on button one being clicked; menu selections are made with either button one or two; the menu can be exited by moving the locator outside its boundary; and the menu will automatically pop up when the locator button is pressed.

```
if ((menu_id = wmenu_create(wfd, MENU_POPUP, 0x0301,
                       MENU_NOPARENT, MENU_NOPARENTITEM)) < 0)
{
     perror("wmenu_create failed");
     exit(1);
}
/*
 * Now add items to the menu...
 */
    .
    .
    .
/*
 * Now activate the menu to pop up automatically:
 */
if ((wmenu_activate(wfd, menu_id, MENU_ACT_AUTO)) < 0)
{
     perror("wmenu_activate wfd failed");
     exit(1);
}
```

Getting Menu Information

There are two methods for getting menu selection information:

- you can set up event detection so that you application is notified when a menu selection is made
- or you can constantly poll for menu information using the *wmenu_eventread(3W)* routine.

Setting Up Event Detection

As mentioned in the "Concepts" section, if you want your programs to be notified when a menu selection is made, you must establish a signal handler and event detection with the window in which the menu is activated. After you've established event detection, your program will be signaled with the SIGWINDOW interrupt that a window event has occurred.

After receiving the signal, you must be sure that it was a menu item selection that caused the event. To do this, you must call *weventpoll(3W)* with the EVENT_MENU bit set in the event mask.

When you've determined that a menu item was selected, you can use the *wmenu_eventread(3W)* routine to determine which menu and what item was selected.

Polling for Event Information

To poll for event information, simply poll for menu information using *wmenu_eventread.* This means to keep calling the routine as often as needed—until you get the menu information that you want. Note that although this method is less efficient than setting up event detection, it is easer.

Procedure

To get the id of the menu from which an item was selected, and to get the item's id, call *wmenu_eventread(3W)*; its syntax is:

```
wmenu_eventread(wfd, menuid, itemno)
```

Menu selections are kept in a queue until requested with this routine. If the queue is empty, then the routine returns -1. Otherwise, it returns the number of items (menu selection data items) remaining in the queue.

Because selection items are queued, you receive menu selection information for the oldest menu selection made. The queue holds information for up to 32 item selections.

The *wfd* parameter is the file descriptor of the window for which menu selection information is to be read.

The *menuid* parameter points to an integer which will contain the id of the menu in which a selection was made.

The *itemno* parameter points to an integer which will contain the item id of the menu item that was selected from the menu. A -1 is returned if the menu was aborted. The menu is aborted any of the following ways:

- the user tries to select a non-selectable item
- the interactive timeout period (as defined by the WMIATIMEOUT environment variable) is exceeded
- an invalid button is pressed.

Deleting a Menu

When you are finished with a menu, you should delete it from the system. The *wmenu_delete(3W)* routine removes a user-defined pop-up menu from the system, releasing the resources that were allocated for that menu.

Procedure

To delete a menu, call *wmenu_delete*; its syntax is:

```
wmenu_delete(wfd, menuid)
```

The *wfd* parameter is the file descriptor of the window whose menu will be removed.

The *menuid* parameter is the menu id of the menu to delete from the window.

Graphics Window Input Routines 11

Depending on your application development needs, you may find **graphics window input** capabilities useful. Graphics window input routines provide different ways to read input from graphics windows. By calling graphics window input routines, a program can:

- enable different input modes (ASCII, two-byte, and packetized input modes)
- read characters in ASCII mode
- read two-byte keycodes in two-byte mode
- change a window's input configuration
- reroute a window's input to another window
- read event packets in packetized input mode.

Concepts

This section discusses concepts essential to using graphics window input routines. Be sure to read this section before using these routines.

Input Modes

A window's **input mode** determines how the window handles input. That is, a window's input mode determines what kind of data the window sends to processes which read from the window. There are three input modes:

- Mode 0 (ASCII Mode)
- Mode 1 (Two-Byte Mode)
- Mode 2 (Packetized Input Mode)

Mode 0 (ASCII Mode)

By default, when a graphics window is created, its input mode is **Mode 0**, also known as **ASCII mode**. If a window is in ASCII mode *and* the window is selected, then all keystrokes are sent to the window's device interface as ASCII characters. See the section "Reading Data in ASCII Mode" for details on using ASCII mode.

Mode 1 (Two-Byte Mode)

In **Mode 1** (also known as **two-byte mode**), each keypress on the keyboard sends a two-byte packet identifying which key (or combination of keys) was pressed. Two-byte mode is useful to applications that require complete keyboard control.

For example, when reading data from a window in ASCII mode, a program cannot tell the difference between a '2' from the typewriter keys and a '2' from the numeric pad keys. However, by reading data from a window in two-byte mode, a process *can* differentiate the $\boxed{2}$ key on the typewriter keys and the $\boxed{2}$ key on the numeric pad. See the section "Reading Data in Two-Byte Mode" for details on using two-byte mode.

Mode 2 (Packetized Input Mode)

In **Mode 2** (also known as **packetized input mode**), a process reads **event code packets** from a window via the *winput_read(3W)* routine. Event code packets contain information for a single key press or window event. Event code packets are time-stamped and usually time-ordered from the input queue, so a program can determine the order of and time between events and key presses.

When a window is in packetized input mode, button presses over the window will not select or top the window as usual. Instead, such button presses are sent to the window in an event code packet.

Packetized input mode provides an alternative to using event detection routines with graphics windows. Some application developers may find packetized input mode more useful than event detection because the order of events can be determined from the time-stamp. Also, event code packets generally provide more detailed information than can be obtained with event detection routines. The use of packetized input mode is discussed in the section "Reading Data in Packetized Input Mode."

Input Re-Routing

Windows/9000 allows the input from one graphics window to be routed to a different graphics window. The window from which input is re-routed is called the **source**; the window to which the source window's input is sent is called the **destination**. A program can re-route the input from several source windows to one destination window. The section "Re-Routing Window Input" discusses how to re-route window input.

Input Configuration

Each graphics window has a set of **input configuration parameters**, which determines how the window handles keyboard input and locator tracking information. For example, a window's input configuration defines the "nationality" of the keyboard when the window is selected; it also determines how the window handles shifted characters; and so on.

Each window has a default configuration, determined from the window's input mode. For modes 0 and 1 (ASCII and two-byte modes), the input configuration cannot be changed. However, the input configuration *can* be changed for windows in Mode 2 (packetized input mode). The section "Changing Input Configuration" describes how to change the input configuration for a window in packetized input mode. It also describes the default configurations for ASCII and two-byte modes.

Changing Input Mode

When a window is created, its input mode is ASCII mode by default. The *wgskbd(3W)* routine changes a graphics window's input mode.

Effect on Line Discipline

Some application developers may need to know how *wgskbd* affects the window's **line discipline**. The line discipline is a group of attributes which determine how the window type device interface handles input. *Wgskbd* calls *ioctl(2)* to change the line discipline for the window as follows (see *termio(7)* and *tty(7)* for details):

- `BRKINT` is enabled—any process affiliated with a window in ASCII or two-byte modes will be signaled (via SIGINT) when the Break key is pressed and the window is selected.
- `ICANON` is disabled—canonical processing is turned off. That is, the window will not process editing keys; read requests are satisfied directly from the input queue. Read requests will not be satisfied until at least `VMIN` bytes have been received, or the timeout value `VTIME` has expired between bytes.
- `VTIME` is 0—there is no time-out. Read requests are satisfied only when `VMIN` bytes have been received.
- `VMIN` is set to the size of the packets read from the window: 1, 2, or `sizeof(struct event_code)`, depending on whether *mode* is 0, 1, or 2, respectively.

Procedure

To change a graphics window's input mode, call *wgskbd(3W)*; its syntax is:

```
wgskbd(fd, mode)
```

fd

The *fd* parameter is the integer file descriptor returned from starting communication with the window.

mode

Set *mode* to 0, 1, or 2 to enable *ASCII*, *two-byte*, or *packetized input* mode, respectively. Set *mode* to $^{-}1$, and *wgskbd* will return the current mode (0, 1, or 2).

Precautions

When the input mode is changed, any unread data from the window will be flushed (lost).

Example

The following function enables *packetized input* mode for a graphics window, given the window's file descriptor (**gwfd**).

```
#include <window.h>
int     enable_packetized(gwfd)
int     gwfd;
{
        if (wgskbd(gwfd, 2) < 0)
            return(-1);
        else
            return(0);
}
```

Reading Data in ASCII Mode

When in ASCII mode, a graphics window processes input data the same as a term0 window in **transmit functions mode**. That is, the function keys (f1...f8), editing keys (e.g., Back space, Delete line), and cursor/screen control keys (e.g., Clear display, or ▼) don't do their usual function; instead, they transmit a special character or an escape sequence (string of special characters) when pressed.

Because the editing keys do not function in ASCII mode, programs must interpret and do the appropriate actions for editing keys, such as Back space, Return, Delete line, etc. The "Graphics Softkeys" chapter defines the special characters and escape sequences returned by editing keys.

Procedure

A program can read from a window's device interface using the *read(2)* system call. The syntax of *read* is:

```
read(fd, buf, nbyte)
```

Fd is the file descriptor returned from starting communication with the window. The *buf* parameter is a pointer to an area of memory, such as an array, in which to put the input characters. *Nbyte* is the number of bytes to read into the *buf* area.

Precautions

- Even though a graphics window is, by default, in ASCII mode when created, it is still good programming practice to call *wgskbd* to set the mode before reading data. This ensures that the window is properly set to ASCII mode.
- Characters typed at the keyboard will not be sent to a window unless the window is *selected*—i.e., the keyboard is attached to the window—or unless the selected window's input is re-routed to it. The *wselect(3W)* routine, described in the "Window Manipulation" chapter, attaches the keyboard to a window.
- Characters typed in the selected graphics window are not displayed. The program which reads the characters must display the characters. Fast alpha and font manager routines display characters in graphics windows; see the "Fast Alpha Library" and "Font Manager Library" chapters for details.

Example

The following program sets a graphics window's input mode to ASCII mode, selects the window, and displays it as the top window in the stack. All characters typed at the keyboard are echoed to standard output. When the user presses the ESC key, the program terminates.

To use this program on a window, you would type:

`program` *window-name*

where `program` is the name of the program after it is compiled and linked, and *window-name* is the name of the graphics window from which `program` reads keyboard input data.

```
#include        <stdio.h>
#include        <window.h>
main(argc, argv)
int     argc;
char    *argv[];
{
int     wmfd;
int     gwfd;
char    wname[WINNAMEMAX];
char    ch;

/*
 * Start communication with the window, select it, and display as top:
 */
        wmfd = est_wm_com();
        wmpathmake("WMDIR", argv[1], wname);
        if ((gwfd = est_gr(wmfd, wname)) == -1) {
                fprintf(stderr, "est_gr failed.\n");
                exit(1);
        }
        wselect(gwfd, SETSELECT);
        wtop(gwfd, SETTOP);

/*
 * Set the input mode to 0 and echo input characters from the window
 *      until the ESC character is read.
 */
        if (wgskbd(gwfd, 0) == -1) {
                fprintf(stderr, "wgskbd failed.\n");
                exit(1);
        }
        while (ch != '\033') {
                read(gwfd, &ch, 1);
                if (ch != '\033') {
                        putchar(ch);
                        fflush(stdout);
                }
        }
        printf("\n\nInput terminated by ESC character.\n");

/*
 * Stop communication with the window:
 */
        if (term_gr(gwfd) == -1) {
                fprintf("term_gr failed.\n");
                exit(1);
        }
        term_wm_com(wmfd);
}
```

Reading Data in Two-Byte Mode

When the selected window is in two-byte mode, each key or combination of keys pressed sends a two-byte **keycode packet** to the window's device interface. The keycode packet identifies which key or combination of keys was pressed on the keyboard.

Key Types

To understand what keycode packets are, one must first know what the different **key types** are. Table 11-1 defines the key types.

Table 11-1. Key Types.

Type	Definition
modifier	The $\boxed{\text{CTRL}}$, $\boxed{\text{Shift}}$, and $\boxed{\text{Extend char}}$ keys are known as *modifier* keys. These are the *only* modifier keys.
normal	Any key that represents a single ASCII character is a *normal* key. This includes the $\boxed{\text{ESC}}$/$\boxed{\text{DEL}}$, alphabetical, numeric, punctuation, and math symbol keys.
special	Any key that is neither a *modifier* nor a *normal* key. This includes keys labelled with words (e.g., $\boxed{\text{Clear display}}$, $\boxed{\text{Print}}$/$\boxed{\text{Enter}}$, $\boxed{\text{Next}}$), function keys ($\boxed{\text{f1}}$...$\boxed{\text{f8}}$), cursor keys ($\boxed{⇱}$, $\boxed{◄}$, $\boxed{►}$, $\boxed{▲}$, and $\boxed{▼}$), and blank keys (like those above the numeric keypad).
npad	Any key which is part of the 18 keys grouped together on the right side of the keyboard and the four unlabeled keys above it.
roman8	Any normal key which is not an *npad* key and not the $\boxed{\text{ESC}}$/$\boxed{\text{DEL}}$ key.

Some keys are members of more than one set of key types. For example, the $\boxed{3}$ key in the numeric pad is both a *normal* and an *npad* key.

Reading Keycode Packets

Reading data in two-byte mode is similar to reading data in ASCII mode: The graphics window device interface must first be opened via *gopen(3G)*. Once the interface is open, the program can read data from the device interface using the *read(2)* system call.

However, in two-byte mode, a program reads two-byte keycode packets instead of ASCII characters. The format of a keycode packet is:

```
struct  keycode {
                unsigned char   control_byte;
                unsigned char   data_byte;
};
```

This structure is *not* defined in any `#include` files. It is shown here only to clarify the structure of keycode packets.

When reading keycode packets, a program should always read in two-byte multiples to ensure that packets don't get "split up." For example, the following code segment requests five keycode packets from the graphics window whose file descriptor is `gwfd`. The keycode packets will be placed in an array of `keycode` structures named `keycode_array`.

```
int     gwfd;
struct  keycode {
                unsigned char   control_byte;
                unsigned char   data_byte;
} keycode_array[5];
                :
        if (read(gwfd, keycode_array, 5 * sizeof(struct keycode)) < 0)
        {
                perror("read gwfd");
                exit(1);
        }
```

Control Byte

The first byte of a keycode packet is the **control byte**. The control byte is a bit mask. The bits of this mask are defined as constants in *window.h*; Table 11-2 shows the constants and briefly describes them.

Table 11-2. Control Byte Constants

Constant (Bit)	Definition
`K_SPECIAL`	Special Key
`K_NPAD`	Numeric Pad Key
`K_SHIFT_B`	Shift Key Pressed Also
`K_CONTROL_B`	CTRL Key Pressed Also
`K_META_B`	Meta (Left Extend char Key) Pressed Also
`K_EXTEND_B`	Extend (Right Extend char Key) Pressed Also

K_SPECIAL

If a *special* key is pressed, then the `K_SPECIAL` bit is set. For example, if the user presses the Enter key on the numeric pad, then the `K_SPECIAL` bit will be set, as well as the `K_NPAD` bit.

K_NPAD

If an *npad* key is pressed, then the `K_NPAD` bit is set. For example, if the user presses the + key on the numeric pad, then this bit will be set in the control byte.

Modifier Keys (K_SHIFT_B, K_CONTROL_B, K_META_B, K_EXTEND_B)

If the user holds down a *modifier* key while pressing a *normal*, *npad*, or *special* key, then the appropriate *modifier* key bit will be set. For example, if the user holds down the left Extend char key and the CTRL key while pressing the Back space key, then the `K_META_B`, `K_CONTROL_B` bits will be set along with the `K_SPECIAL` bit.

Data Byte

The second byte of the keycode packet is the **data byte**. The value of the data byte depends on the key type.

Data Byte Values for Special Keys

If the `K_SPECIAL` bit is set in the control byte, then the data byte is set to a special key value, not an ASCII value. Data byte values for special keys are defined in */usr/include/window.h* under the "Special key defines" section. For example, if the user presses the rightmost key above the numeric pad, the data byte will be `K_NP_K3`, as defined in *window.h.*

A program may not be able to read some of the special keys defined in *window.h.* The special keys a program *can* receive from a window depend on the keyboard's nationality. For example, a program cannot read a `K_GO_KANJI` key from a USASCII keyboard.

Data Byte Values for Normal Keys

The data byte for normal keys is determined from the following rules:

1. **If no modifier keys are pressed**, then the data byte is set to the shifted or unshifted value of the pressed key, determined from the current **capslock** state.

   Capslock is initially OFF, meaning that *roman8* characters will be lower case when typed. When the [Caps] key is pressed, the current *capslock* state is toggled. If *capslock* is ON, then *roman8* characters will be mapped to upper case. For example, if *capslock* is OFF, then the [H] key will send a keycode packet whose control byte is zero and whose data byte is set to the ASCII value for the 'h' character. And if *capslock* is ON, then pressing the [H] key by itself will send a packet whose control byte is zero and whose data byte is set to the ASCII value for the 'H' character.

   The [Caps] key itself sends a special key value when pressed. If *capslock* is currently OFF, then pressing [Caps] will send a keycode packet with the `K_SPECIAL` bit set in the control byte, and the data byte set to `K_CAPS_ON`. If *capslock* is ON, then the [Caps] will send a keycode packet with the `K_SPECIAL` bit set and the data byte set to `K_CAPS_OFF`.

2. **If the user holds down the [Shift] modifier key** while pressing another key, then the `K_SHIFT_B` bit is set in the control byte, and the data byte is mapped to the appropriate upper- or lower-case ASCII character, depending on the current *capslock* state.

   For example, if *capslock* is ON, and the user holds down [Shift] while pressing the [A] key, then the `K_SHIFT_B` bit will be set in the control byte and the data byte will be set to the ASCII value for the 'a' character.

3. **If the user holds down the CTRL modifier key** while pressing another key, then the `K_CONTROL_B` bit is set in the control byte, and the data byte is set to the ASCII value of the key. For example, if *capslock* is OFF and the user holds down CTRL while pressing the C key, then the control byte will be set to `K_CONTROL_B` and the data byte will be set to the ASCII value for the 'c' character.
4. **If the user holds down the left Extend char modifier key** while pressing another key, then `K_META_B` is set in the control byte, and the data byte is set to the ASCII value of the key. For example, if *capslock* is ON and the user holds down the left Extend char key while pressing the M key, then the control byte will be set to `K_META_B` and the data byte will be set to the ASCII value for the 'M' character.
5. **If the user holds down the right Extend char modifier key** while pressing a *roman8* key, then `K_EXTEND_B` is set in the control byte, and the data byte is set to a value corresponding to the appropriate Roman-8 character. If the user does *not* press a *roman8* key, then data byte is set to the ASCII value for the pressed key.

   For details on the ITF keyboard layout, see the article "Series 300 System Console" in *HP-UX Concepts and Tutorials: Facilities for Series 200, 300, and 500.* For information on the Roman-8 character set, see *roman8(4)* in the *HP-UX Reference.*

Numeric Pad Keys

When an *npad* key is pressed, the `K_NPAD` bit is set in the control byte. The numeric pad contains both *normal* and *special* keys. If a *normal* key is pressed, then the data byte is set to the ASCII value of the key. If a *special* key is pressed, then the `K_SPECIAL` bit is also set in the control byte, and the data byte is set to the value of the special key, as defined in *window.h.*

Note that the *modifier* keys do *not* cause *npad* keys to be mapped to different values: data byte is always set to the value of the pressed key.

Special Cases

There are special keystrokes which a program should know about.

BREAK Key

If the user presses the Break key by itself, or with any *modifier* key except Shift, then a keycode packet will be generated with both the control byte and data byte set to zero. In addition, a `TCIOBREAK` *ioctl(2)* call will be issued, which will send the SIGINT signal to the user process if it has done a *setpgrp(2)* properly; otherwise, the user process will only receive the keycode packet.

SHIFT-SELECT

A process cannot read the Shift-Select key combination. This key combination is caught by the window system (to shuffle windows) and cannot be used by user programs.

CONTROL-ARROWS

The combination of the CTRL and arrow (◄, ►, ▲, ▼) keys will be sent to processes reading from the window. However, these key presses will also move the pointer on the window system desktop.

Example

The following program reads keys from a graphics window in two-byte mode. The program echoes a descriptive message for the control byte and data byte for each key pressed. When the user holds down the CTRL key while pressing the Enter key on the numeric pad, the program terminates.

The program would be executed as:

`program` *window-name*

where `program` is the name of the program after it is compiled, and *window-name* is the name of the graphics window from which to read two-byte keycode data.

```
#include        <stdio.h>
#include        <window.h>
main(argc, argv)
int     argc;
char    *argv[];
{
int     wmfd;
int     gwfd;
char    wname[WINNAMEMAX];
short   done = 0;
unsigned        char    control_byte;
unsigned        char    data_byte;

/*
 * Start communication with the window, top it , and select it:
 */
        wmfd = est_wm_com();
        wmpathmake("WMDIR", argv[1], wname);
        if ((gwfd = est_gr(wmfd, wname)) == -1) {
                fprintf(stderr, "est_gr failed.\n");
                exit(1);
        }
        wselect(gwfd, SETSELECT);
        wtop(gwfd, SETTOP);

/*
 * Set the input mode to 1 and echo keycode structures
 *      until the Enter key on the numeric pad is pressed:
 */
        if (wgskbd(gwfd, 1) == -1) {
                fprintf(stderr, "wgskbd failed.\n");
                exit(1);
        }

        while (done == 0) {
                read(gwfd, &control_byte, 1);
                read(gwfd, &data_byte, 1);
                printf("----------\n");
                if (control_byte & K_SPECIAL)
                        printf("Special Key\n");
                else
                        printf("Normal Key\n");
                if (control_byte & K_NPAD)
                        printf("Numeric Pad Key\n");
                if (control_byte & K_SHIFT_B)
                        printf("SHIFT also pressed\n");
                if (control_byte & K_CONTROL_B)
                        printf("CTRL also pressed\n");
                if (control_byte & K_META_B)
                        printf("Meta also pressed\n");
                if (control_byte & K_EXTEND_B)
                        printf("Extend also pressed\n");
                if (control_byte == K_SPECIAL | K_NPAD | K_CONTROL_B)
                        done = (data_byte == K_NP_ENTER);
                if (control_byte & K_SPECIAL)
                        printf("Data byte value: %d\n", data_byte);
                else
                        printf("Data byte value: %c\n", data_byte);

        }
        printf("\n\nInput terminated by CTRL + Enter key on numeric pad.\n");

/*
 * Stop communication with the window:
 */
        if (term_gr(gwfd) == -1) {
                fprintf("term_gr failed.\n");
                exit(1);
        }
        term_wm_com(wmfd);
}
```

Changing Input Configuration

As mentioned in the "Concepts" section, each graphics window has a default *input configuration*, a set of parameters which determine how the window handles keyboard input and locator tracking. For ASCII and two-byte modes this configuration cannot be changed. However, for packetized input mode, it can be changed via the *winput_conf(3W)* routine.

Input Configuration Parameters

Before you can change a window's input configuration parameters, you must know what they are. Listed below are the input configuration parameters and their default values for each input mode. The parameter names are defined in *window.h*.

K_TRACK

If set, causes the window to report all locator movements for the window when in packetized input mode; that is, allows a program to read locator movements from the window via *winput_read(3W)*. Locator moves are reported only when the keyboard is attached to the window. Locator moves during an interactive size or move or pop-up menu operation are not reported.

Note: Enabling `K_TRACK` will degrade window system performance because the window manager must constantly report locator movements. Typically, a program can use hotspots to eliminate the need for continuous locator tracking.

By default, `K_TRACK` is cleared for all input modes.

K_LANGUAGE

Language nationality of the keyboard when attached to the window. See the *winput_conf(3W)* page for supported keyboard nationalities. The constant values shown on the *winput_conf(3W)* page are defined in *window.h*.

The default language is that of the keyboard attached to the computer. For example, if the system uses a United States ITF keyboard, then the default language is `K_I_USASCII`.

K_CAPSMODE

Determines whether *capslock* processing is done. If this parameter is set, then *capslock* processing is enabled. If this parameter is not set, then *capslock* processing is disabled.

When *capslock* processing is enabled, the [Caps] key toggles the current *capslock* state and causes either the `K_CAPS_ON` or `K_CAPS_OFF` key to be sent. When *capsmode* is disabled, the [Caps] key simply causes the `K_CAPS_LOCK` key to be sent; all *capslock* processing is disabled (the `K_CAPS_ON` and `K_CAPS_OFF` keys are not sent).

K_CAPSLOCK

Affect the *capslock* state. If set, then *capslock* is turned ON; if cleared, *capslock* is turned off. This is effective only if `K_CAPSMODE` is set.

When *capslock* is OFF, *roman8* keys are mapped to lower case characters, unless the user also holds down the [Shift] key, in which case the key is mapped to an upper case character. For example, if *capslock* is OFF and the user holds down the [Shift] key while pressing the [T] key, then a 'T' character is sent. The converse is true when *capslock* is ON: For example, if *capslock* is ON and the user presses the [N] key, then a 'N' character is sent; but if the user also holds down the [Shift] key, then a 'n' character is sent.

This parameter is cleared by default (*capslock* is initially OFF).

K_EXTEND

Alternate keyboards. For some languages, the [Extend char] key to the right of the space bar toggles between normal and alternate keyboards. For other languages, the right [Extend char] key is a *modifier* key to get additional keycodes. This parameter controls whether this key does the language-dependent function.

If this parameter is set, the language-dependent function will be done when this key is pressed; if not set, this function won't be done. Depending on your application development needs, this may or may not be important.

By default, this parameter is set for all input modes.

K_CONTROL
Control collapsing of printable characters. If this parameter is set, the CTRL key causes characters from 64 to 127 decimal to be collapsed to their control values before being sent. If not set, then collapsing is not done. (In either case, the CTRL key still causes the `K_CONTROL_B` bit to be set in the control byte when the keycode is sent.) For example, if `K_CONTROL` is set and the user holds down CTRL while pressing D, then ASCII EOT character (decimal value 4) is sent (`'D' & 037`).

For mode 0, this parameter is set by default; for modes 1 and 2, this parameter is cleared.

K_SHIFT
Shift collapsing of capitals. If set, the Shift key toggles the case of keys that are affected by *capslock*. If not set, then the Shift key does not affect the case of keys. (In either case, the `K_SHIFT_B` bit is set in the control byte if Shift is pressed.)

This parameter is set by default.

K_META
Enable Meta modifiers. If set, the presence of Meta keys will be recognized by setting the appropriate Meta bits when key codes are sent. If cleared, this capability is disabled.

This parameter is set by default.

K_META_EXTEND
Enable the Extend char key to the left of the space bar as the Meta key. This is effective only when `K_META` is set. If this parameter is set, the left Extend char key becomes the Meta key. For the Katakana keyboard, it will also switch the keyboard to the Roman keyboard at the same time.

If this parameter is cleared, the left Extend char key is simply treated as an Extend char key, and not a Meta key.

For mode 0, this parameter is cleared by default; for modes 1 and 2, this parameter is set.

K_KANAKBD
Katakana keyboard. If set, the alternate Katakana keyboard is currently active. If cleared, the Katakana keyboard is not active. This parameter is effective only with Katakana-language keyboards.

This parameter is cleared by default.

K_KANJI
Enable KANJI mode. This parameter is effective only when the keyboard language is japanese. If this is set, the left Extend char key toggles the state of `K_KANJIKBD`, described next.

This parameter is cleared by default.

K_KANJIKBD
KANJI input mode. If set, the left Meta key will be used as a key only. If cleared, the left Meta key will be used as a Meta key. (The left and right Meta keys, when present, are directly under the Shift keys.)

This parameter is cleared by default.

Procedure
Input configuration parameters cannot be changed for windows in ASCII or two-byte modes. However, a program *can* change input configuration parameters for a window in *packetized input mode.*

The *winput_conf(3W)* routine can be used to change or determine an input configuration parameter; its syntax is:

```
winput_conf(fd, param, value)
```

The *fd* parameter is the integer file descriptor returned from starting communication with the window. *Param* should be set to the parameter, from the above list, to change or inquire. The *value* parameter defines the value to set the parameter to.

To set a parameter other than `K_LANGUAGE`, set *value* to 1; to clear a parameter other than `K_LANGUAGE`, set *value* to 0. (For the `K_LANGUAGE` parameter, set *value* to a supported value shown on the *winput_conf(3W)* reference page.)

To inquire a parameter's value, set *value* to -1; *winput_conf* will then return the current value (0 if the parameter is not set, 1 if set).

Example

The following code segment sets the window's input mode to packetized input (Mode 2). Then it determines the current value for K_TRACK and displays whether it is set. Finally, it sets the K_CONTROL parameter.

```
#include        <stdio.h>
#include        <window.h>
main()
{
int     gwfd;   /* fildes for the graphics window */
int     param;  /* variable to hold input configuration parameter */
int     pval;   /* variable to hold value of the input config param */
           :

/* Set the window's input mode to packetized input (Mode 2): */
        if (wgskbd(gwfd, 2) == -1) {
                perror("wgskbd gwfd");
                exit(1);
        }
           :

/* Inquire the window's K_TRACK parameter value and display: */
        pval = winput_conf(gwfd, K_TRACK, -1);
        switch (pval) {
                case 0 : printf("Tracking is OFF.\n");
                         break;
                case 1 : printf("Tracking is ON.\n");
                         break;
                default : perror("winput_conf gwfd");
                          exit(1);
        }
           :

/* Set the K_CONTROL parameter: */
        if (winput_conf(gwfd, K_CONTROL, 1) == -1) {
                perror("winput_conf gwfd");
                exit(1);
        }
           :
}
```

Re-Routing Window Input

By calling graphics window input routines, a program can re-route a window's input to another window. That is, the keystrokes and events for one window can be sent to another window. By re-routing several windows' input to one window, a program can receive all events and keystrokes through one window, instead of having to read them separately from each window. The following routines are used to re-route graphics window input: *winput_setroute(3W)*, *winput_getroute(3W)*, *winput_widpath(3W)*.

Concepts

Before using re-routing routines, you should understand some basic concepts.

Source and Destination Window

The window from which input is re-routed is the **source** window. The window that the source window's input is re-routed to is the **destination** window.

Many-to-One Re-Routing

Each source window can be re-routed to only one destination window at a time. However, several source windows can be re-routed to the same destination window (a **many-to-one** group). Figure 11-1 shows some valid many-to-one groups.

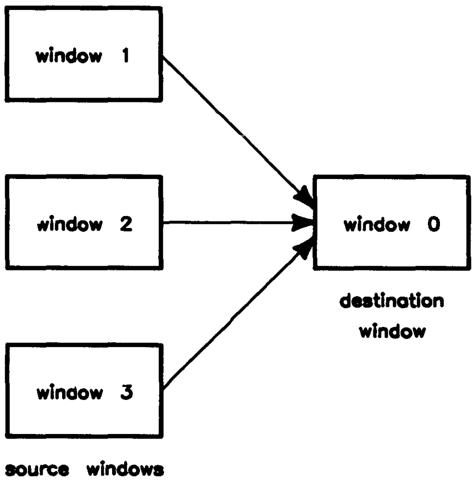


Figure 11-1. Valid Many-to-One Input Re-Routing Groups

Multi-Hop Re-Routing

Windows can also be re-routed in a **multi-hop** formation (i.e., from window to window to window, etc.). Figure 11-2 shows some valid multi-hop routes.

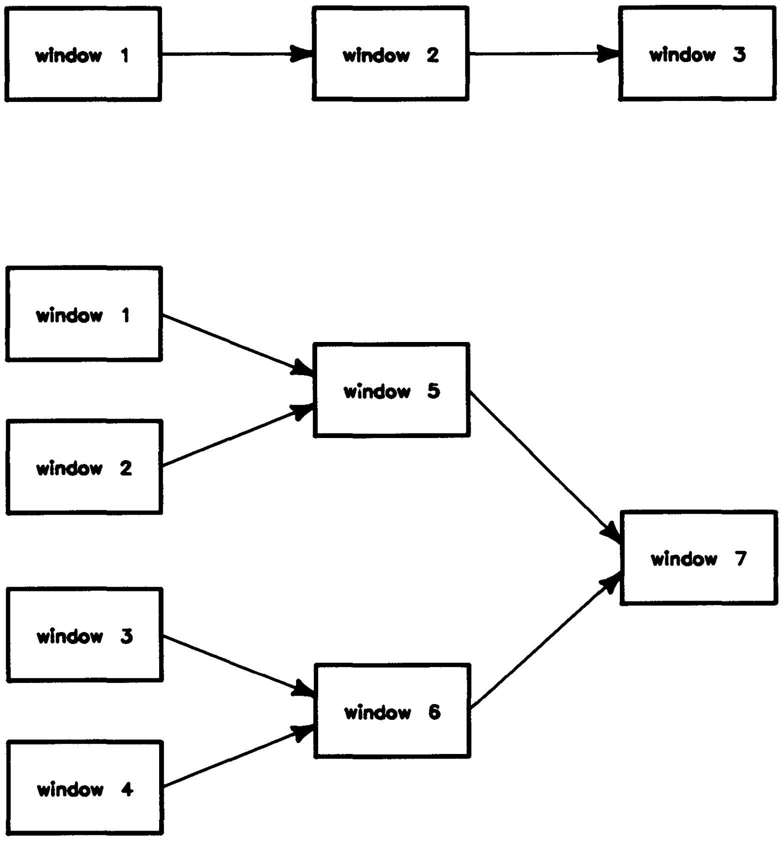


Figure 11-2. Valid Multi-Hop Routes

Routing Loops

A window's input *cannot* be re-routed backward to any preceding window in the input path (known as **routing loops**). All routing loops are illegal. Figure 11-3 shows some routing loops.

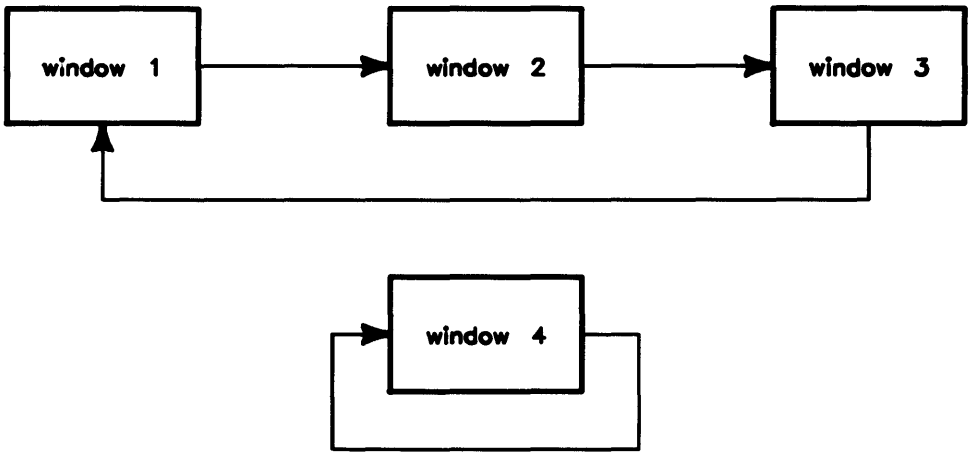


Figure 11-3. Routing Loops—Don't Do Them

Window ID (wid)

Every window in an input path has a unique **window id** (**wid**) that identifies the window. Window id's are useful in packetized input mode: they identify the window from which an event code packet originated.

In packetized input mode, each event code packet has a *wid* field which is set to the window id of the window from which the packet originated. In ASCII and two-byte modes, a program cannot determine the originating window for keystrokes.

Note that a window id is **not the same** as a window's file descriptor. Window id's are maintained globally by the window system, whereas file descriptors are private to processes.

Final Destination Window

The **final destination window** is the last window in an input path. Typically, a program reads from this window to get input from all windows in the input path.

Input Modes and Configuration Parameters

On receiving input from a source window, a destination window handles the data in a way appropriate to its input mode, as set by the *wgskbd(3W)* routine, regardless of the input mode of the source window.

Each window also has its own set of input configuration parameters, as determined by the window's input mode (or set via *winput_conf(3W)*). When using input routing, it is normally desirable to have the same input configuration parameters for all windows. This can be accomplished by calling *wgskbd* to set the mode for each window; or if in packetized input mode, by calling *winput_conf* to set the parameters for each window.

Although it is desirable to have the same input configuration parameters, it is not absolutely necessary. If the source and destination windows have different input configuration parameters, then input from the destination window will conform to input configuration parameters of the source window.

For most applications, the input mode and input configuration parameters for all windows in an input route should be the same. For example, if the final destination window is in ASCII mode, then all windows in the path should be in ASCII mode. If the final destination window is in two-byte mode, then all windows in the path should be in two-byte mode. (By default, if all windows are in ASCII mode or in two-byte mode, their input configuration parameters will be identical, too.)

As another example: If the final destination window is in packetized input mode, then all windows in the path should be in packetized input mode. If a program changes the input configuration parameters of any window in the input path, then all windows should be changed similarly, thus ensuring that the input configuration parameters are identical for all windows.

Procedure

By calling graphics window input routines a program can:

- re-route a window's input
- get re-routing information for a window

Re-Routing

To re-route a window's input to another window, call *winput_setroute(3W)*; its syntax is:

```
winput_setroute(fd, routepath)
```

Fd is the integer file descriptor returned from starting communication with the source window. The *routepath* parameter points to the path name of the destination window. After calling this routine, all input from the window represented by *fd* will be sent to the window whose window type device interface is pointed to by the *routepath* parameter.

Winput_setroute returns the window id of the source window. **Note** that the window id is **not** the same as the *fd* parameter.

If a program reads data in packetized input mode, it should save the window id's of windows when calling *winput_setroute*. This way, it can determine which window an event code packet came from by comparing the window id field of the event code packet with the window id's saved from *winput_setroute*.

Cancelling Re-Routing

To turn off input re-routing for a window, call *winput_setroute* on the window and set the *routepath* parameter to NULL. For example, to turn off input re-routing for the window whose file descriptor is `wfd`, call *winput_setroute* as:

```
winput_setroute(wfd, NULL)
```

where NULL is defined in *stdio.h*.

Determining Whether Input Routing Is in Effect

It may be useful for a program to determine whether a window is part of an input routing path, and if so, what is its destination window. The *winput_getroute(3W)* routine does this; its syntax is:

```
winput_getroute(fd, routepath)
```

The *fd* parameter is the file descriptor returned from starting communication with the window for which input routing is to be determined. The *routepath* parameter is a pointer to a space to be filled with the null-terminated path name of the destination window device interface if routing is in effect. The space must be large enough to hold WINNAMEMAX characters (the maximum path name length for window type device interfaces).

If input routing is not in effect for the source window, then *routepath* will point to a zero-length, null-terminated string.

If *fd* is valid, then *winput_getroute* always returns the window id of the window of the window represented by *fd*. This is useful for getting the window id of the final destination window.

Getting the Path Name for a Window ID

Given a window id, the *winput_widpath(3W)* routine returns the path name of the window's window type device interface. The syntax of *winput_widpath* is:

```
winput_widpath(wmfd, wid, wname)
```

The *wmfd* parameter is an integer file descriptor returned from starting communication with the window manager. The *wid* parameter is the window id to get the path name for. And the *wname* parameter is a pointer to a space to put the path name of the window type device interface whose window id is *wid*. This space must be large enough to hold WINNAMEMAX characters (the maximum length of window type device interface path names).

If *wid* represents a valid window id, then *winput_widpath* returns zero.

Reading Data in Packetized Input Mode

In packetized input mode (Mode 2), each event or keystroke in a window is sent to the window's window type device interface in an **event code packet**. The event code packet contains information which defines whether an event or keystroke occurred. The event code packet also contains information describing what type of event or which keystroke occurred. Programs must read event code packets using the *winput_read(3W)* routine.

Event Code Packets

An event code packet is a structure defined in *window.h*:

```
struct event_code {
        unsigned char control_byte;
        unsigned char data_byte;
        unsigned char event_byte;
        unsigned char event_cause;
        unsigned int timestamp;
        unsigned int wid;
        int x;
        int y;
};
```

A description of each field of this structure is listed next.

control_byte

The *control_byte* is a bit mask, similar to the control byte used in two-byte mode, except that it has two additional bits: `K_EVENT` and `K_UP`.

If the `K_EVENT` bit is set (==1), then the event code packet represents an event; if the `K_EVENT` bit is cleared (==0), then the event code packet represents a keystroke.

If the event code packet represents a keystroke (`K_EVENT` bit == 0), then the *control_byte* field represents the same information as the control byte in two-byte mode. For example, if a window has the default input configuration parameters for packetized input mode, and the user holds down the Shift key while pressing the J key, then an event code packet will be sent with the `K_SHIFT_B` bit set.

The `K_UP` bit is used only with locator button events. If `K_UP` is set (==1), then a button was released from a down position; if `K_UP` is cleared (==0), then a button was pressed. In either case, the `K_EVENT` bit will also be set, indicating that the event code packet represents an event.

data_byte

If the event code packet represents a keystroke (K_EVENT bit == 0), then the *data_byte* field represents the same information as the data byte field in two-byte mode: If the K_SPECIAL bit is set in *control_byte*, then *data_byte* is set to a value (defined in *window.h*) representing the special key. A complete list of supported special keys is given on the *winput_read(3W)* reference page.

If the K_SPECIAL bit is cleared (==0), then data byte is mapped to the appropriate character value, depending on:

1. Which modifier keys are pressed (determined from *control_byte*), and
2. The window's input configuration parameters (changed via *winput_conf(3W)*).

event_byte

If the event code packet represents an event (K_EVENT bit set in *control_byte*), then *event_byte* represents the type of event that occurred. A complete list of event causes is listed on the *winput_read(3W)* reference page.

For hotspot events, *event_byte* is set to the hotspot's event byte value, specified when the hotspot was created (via *whotspot_create(3W)*) or changed (via *whotspot_set(3W)*). Review the "Graphics Window Hotspots" chapter for details on hotspot event bytes.

event_cause

If the event code packet represents an event, **and** if the event was caused by a hotspot or valid menu item selection, then *event_cause* indicates what caused the event. Valid values for *event_cause* are defined on the *winput_read(3W)* reference page.

timestamp

The *timestamp* field is a 32-bit integer specifying when the packet was received by the window. Time is given in milliseconds.

wid

The *wid* field is the window id of the window from which the packet originated. The window id is useful if a program reads packets from a destination window that is part of an input routing path. For details on window id's, see the section "Re-Routing Window Input."

x and y

If the event code packet represents an event other than hotspot events, *x* and *y* are set to the *x,y* values returned by *weventpoll(3W)* during event detection. (See the "Event Detection" chapter for detailed descriptions of *x* and *y*.)

For hotspot events, *x* and *y* give the position of the locator when the hotspot event occurred. *X* and *y* are given relative to location *0,0* of the window's virtual raster.

Procedure

The *winput_read(3W)* routine reads event code packets from a window's device interface; its syntax is:

```
winput_read(fd, bufadr, count)
```

Winput_read attempts to read *count* event code packets into the buffer *bufadr*. Event code packets are read from the window whose file descriptor is *fd*. *Bufadr* is a pointer to a memory space to contain the event code packets read.

To get the best performance when reading event code packets, set *count* to 25. *Winput_read* returns the number of event code packets actually read, which may be less than *count*. See "Input Blocking" below for details.

Event Code Overflow

It is possible that more than 25 events or keypresses may occur between calls to *winput_read*. If this happens, an event code packet is sent with its *event_byte* field set to K_OVERFLOW.

Any hotspot events or keystrokes occurring after overflow will be lost. However, the window manager keeps track of the most recent state for other types of events, and will send an event code packet indicating these states.

For example, suppose overflow occurs and the user moves a window several times before the program calls *winput_read*. All the locator moves after the K_OVERFLOW packet will be lost. But the last window move will *not* be lost; the program will still receive a K_MOVE_CT packet indicating where the window was placed by the final move.

Input Blocking

If a program calls *winput_read* on a window that doesn't yet have any event code packets, then the action of *winput_read* depends on the `O_NDELAY` value, set when the window type device interface was opened. (See *open(2)* and *fcntl(2)* for details on `O_NDELAY`.)

If `O_NDELAY` is set, *winput_read* returns zero, meaning that no event code packets have yet occurred since the last call to *winput_read.*

If `O_NDELAY` is not set, *winput_read* will block until an event occurs in the window or a signal aborts the read. If an event occurs, *winput_read* returns an event code packet for the event; if a signal occurs, *winput_read* returns $^{-}1$ and *errno(2)* is set to `EINTR`.

Example

The following program reads event code packets from a graphics window. It then displays information from each event code packet.

The program would be executed as:

`program` *window-name*

where `program` is the name of the program after it is compiled, and *window-name* is the name of the graphics window from which event code packets will be read.

```
#include        <stdio.h>
#include        <window.h>
extern  int     errno;
main(argc, argv)
int     argc;
char    *argv[];
{
int     wmfd;
int     gwfd;
int     i, count;
char    wname[WINNAMEMAX];
struct event_code buffer[10];
/*
 * Start communication with the window, select it, and display as top:
 */
        wmfd = est_wm_com();
        wmpathmake("WMDIR", argv[1], wname);
        if ((gwfd = est_gr(wmfd, wname)) == -1) {
                fprintf(stderr, "est_gr failed.\n");
                exit(1);
        }
        wselect(gwfd, SETSELECT);
        wtop(gwfd, SETTOP);

/*
 * Set the input mode to 2 and echo input characters from the window
 *      until the ESC character is read.
 */
        if (wgskbd(gwfd, 2) == -1) {
                fprintf(stderr, "wgskbd failed.\n");
                exit(1);
        }

/*
 * Loop and read event code packets no more than 10 at a time:
 */
        while ((count=winput_read(gwfd, buffer, 10)) >= 0) {
            printf("\nwinput_read returned %d\n\n",count);
            for (i = 0; i < count; i++)
                if (buffer[i].control_byte & K_EVENT) {
                    print_event(&buffer[i]);
                }
                else {
                    print_key(&buffer[i]);
                }
            fflush (stdout);
        }
        if (count < 0) fprintf(stderr, "errno = %d \n", errno);

/*
 * Stop communication with the window:
 */
        if (term_gr(gwfd) == -1) {
                fprintf("term_gr failed.\n");
                exit(1);
        }
        term_wm_com(wmfd);
}
```

```
/*
 * Display information for an event:
 */
print_event(ep)
struct event_code *ep;
{
    printf("Event, wid = %2d\n",ep->wid);
    print_control_byte(ep->control_byte);
    print_data_byte(ep->control_byte, ep->data_byte);
    printf("   event_byte = ");
        switch (ep->event_byte) {
        case K_MOVE_CT:       printf("K_MOVE_CT");    break;
        case K_SIZE_LR_CT:    printf("K_SIZE_LR_CT"); break;
        case K_ICON_SHK:      printf("K_ICON_SHK");   break;
        case K_ICON_EXP:      printf("K_ICON_EXP");   break;
        case K_PAUSE:         printf("K_PAUSE");      break;
        case K_DESTROY:       printf("K_DESTROY");    break;
        case K_SELECTED:      printf("K_SELECTED");   break;
        case K_USELECTED:     printf("K_USELECTED");  break;
        case K_REPAINT:       printf("K_REPAINT");    break;
        case K_FSSM_ABORT:    printf("K_FSSM_ABORT"); break;
        case K_MOUSE_MOVE:    printf("K_MOUSE_MOVE"); break;
        case K_BUTTON:        printf("K_BUTTON");     break;
        case K_MENU_ITEM:     printf("K_MENU_ITEM");  break;
        case K_ELEV_CT:       printf("K_ELEV_CT");    break;
        case K_SB_ARROW:      printf("K_SB_ARROW");   break;
        case K_OVERFLOW:      printf("K_OVERFLOW");   break;
        case K_MOVE_ST:       printf("K_MOVE_ST");    break;
        case K_POPUP_ST:      printf("K_POPUP_ST");   break;
        case K_SIZE_LR_ST:    printf("K_SIZE_LR_ST"); break;
        default:              printf("Unknown (%3d)",ep->event_byte); break;
        }
        printf("\n");
        printf("   event_cause = ");
        switch (ep->event_cause) {
        case EC_NONE:          printf("EC_NONE");              break;
        case EC_BUTTON1:       printf("EC_BUTTON1");           break;
        case EC_BUTTON2:       printf("EC_BUTTON2");           break;
        case EC_BUTTON3:       printf("EC_BUTTON3");           break;
        case EC_BUTTON4:       printf("EC_BUTTON4");           break;
        case EC_BUTTON5:       printf("EC_BUTTON5");           break;
        case EC_BUTTON6:       printf("EC_BUTTON6");           break;
        case EC_BUTTON7:       printf("EC_BUTTON7");           break;
        case EC_BUTTON8:       printf("EC_BUTTON8");           break;
        case EC_SELECT:        printf("EC_SELECT");            break;
        case EC_ENTER:         printf("EC_ENTER");             break;
        case EC_EXIT:          printf("EC_EXIT");              break;
        default:               printf("Unknown (%3d)",ep->event_cause); break;
        }

        printf("\n");
    printf("   timestamp = %d\n",ep->timestamp);
    printf("   x,y = %d,%d\n",ep->x,ep->y);
}

/*
 * Print keycode packet information:
 */
print_key(kp)
struct event_code *kp;
{
    printf("Key, wid = %2d\n",kp->wid);
    print_control_byte(kp->control_byte);
    print_data_byte(kp->control_byte, kp->data_byte);
    printf("   timestamp = %d\n",kp->timestamp);
}

/*
 * Print the data byte of the keycode:
 */
print_data_byte(control_byte, data_byte)
unsigned char control_byte, data_byte;
{
        if (control_byte&K_SPECIAL) {
            printf("   data_byte = ");
            switch (data_byte) {
            case K_ILLEGAL:        printf("K_ILLEGAL");            break;
            case K_EXTEND_LEFT:    printf("K_EXTEND_LEFT");        break;
            case K_EXTEND_RIGHT:   printf("K_EXTEND_RIGHT");       break;
            case K_META_LEFT:      printf("K_META_LEFT");          break;
            case K_META_RIGHT:     printf("K_META_RIGHT");         break;
            case K_CAPS_ON:        printf("K_CAPS_ON");            break;
            case K_CAPS_OFF:       printf("K_CAPS_OFF");           break;
case K_GO_ROMAN:        printf("K_GO_ROMAN");         break;
case K_GO_KATAKANA:     printf("K_GO_KATAKANA");      break;
case K_BUTTON1:         printf("K_BUTTON1");          break;
case K_BUTTON2:         printf("K_BUTTON2");          break;
case K_BUTTON3:         printf("K_BUTTON3");          break;
case K_BUTTON4:         printf("K_BUTTON4");          break;
case K_BUTTON5:         printf("K_BUTTON5");          break;
case K_BUTTON6:         printf("K_BUTTON6");          break;
case K_BUTTON7:         printf("K_BUTTON7");          break;
case K_BUTTON8:         printf("K_BUTTON8");          break;
case K_GO_KANJI:        printf("K_GO_KANJI");         break;
case K_GO_NOKANJI:      printf("K_GO_NOKANJI");       break;
case K_BREAK:           printf("K_BREAK");            break;
case K_STOP:            printf("K_STOP");             break;
case K_SELECT:          printf("K_SELECT");           break;
case K_NP_ENTER:        printf("K_NP_ENTER");         break;
case K_NP_K0:           printf("K_NP_K0");            break;
case K_NP_K1:           printf("K_NP_K1");            break;
case K_NP_K2:           printf("K_NP_K2");            break;
case K_NP_K3:           printf("K_NP_K3");            break;
case K_HOME_ARROW:      printf("K_HOME_ARROW");       break;
case K_PREV:            printf("K_PREV");             break;
case K_NEXT:            printf("K_NEXT");             break;
case K_ENTER:           printf("K_ENTER");            break;
case K_SYSTEM:          printf("K_SYSTEM");           break;
case K_MENU:            printf("K_MENU");             break;
case K_CLR_LINE:        printf("K_CLR_LINE");         break;
case K_CLR_DISP:        printf("K_CLR_DISP");         break;
case K_CAPS_LOCK:       printf("K_CAPS_LOCK");        break;
case K_TAB:             printf("K_TAB");              break;
case K_F1:              printf("K_F1");               break;
case K_F2:              printf("K_F2");               break;
case K_F5:              printf("K_F5");               break;
case K_F6:              printf("K_F6");               break;
case K_F7:              printf("K_F7");               break;
case K_F3:              printf("K_F3");               break;
case K_F4:              printf("K_F4");               break;
case K_DOWN_ARROW:      printf("K_DOWN_ARROW");       break;
case K_UP_ARROW:        printf("K_UP_ARROW");         break;
case K_F8:              printf("K_F8");               break;
case K_LEFT_ARROW:      printf("K_LEFT_ARROW");       break;
case K_RIGHT_ARROW:     printf("K_RIGHT_ARROW");      break;
case K_INSERT_LINE:     printf("K_INSERT_LINE");      break;
case K_DELETE_LINE:     printf("K_DELETE_LINE");      break;
case K_INSERT_CHAR:     printf("K_INSERT_CHAR");      break;
case K_DELETE_CHAR:     printf("K_DELETE_CHAR");      break;
case K_BACKSPACE:       printf("K_BACKSPACE");        break;
case K_RETURN:          printf("K_RETURN");           break;
default:                printf("Unknown");            break;
}
```

```
        } else {
            printf("   data_byte = %3d",data_byte);
            if (data_byte >= ' ' && data_byte <= '~')
                printf(" '%c'",data_byte);
        }
        printf("\n");
}

/*
 * Print the control byte of a keycode packet:
 */
print_control_byte(control_byte)
unsigned char control_byte;
{
    printf("   control_byte =");
        if (control_byte&K_SHIFT_B)     printf(" K_SHIFT_B");
        if (control_byte&K_CONTROL_B)   printf(" K_CONTROL_B");
        if (control_byte&K_META_B)      printf(" K_META_B");
        if (control_byte&K_EXTEND_B)    printf(" K_EXTEND_B");
        if (control_byte&K_UP)          printf(" K_UP");
        if (control_byte&K_NPAD)        printf(" K_NPAD");
        if (control_byte&K_EVENT)       printf(" K_EVENT");
        if (control_byte&K_SPECIAL)     printf(" K_SPECIAL");
        if (control_byte == 0)          printf(" 0");
        printf("\n");
}
```

Graphics Softkeys 12

This chapter discusses the use of softkeys with graphics windows. The following topics are covered:

- concepts essential to understanding the use of softkeys with graphics windows
- turning on and off softkey labels
- changing softkey labels

Note

For details on user-defined softkeys with term0 windows, see Chapter 10, "Term0 Windows."

Concepts

Any process that reads input from a selected graphics window can take input from the keyboard function keys ([f1], [f2],..,[f8], [User], [System], and [Menu]). Each of these function keys has a **softkey definition** comprised of a **softkey label** and **return value**.

Softkey Labels

Softkey labels are descriptive names that correspond to the function keys. The selected window's softkey labels are displayed at the base of the display if the display of the window's softkey labels has been enabled via window routines.

Windows/9000 allows you to set your own labels for the function keys [f1] through [f8]. For example, you could set the softkey labels to represent menu options that the user chooses by pressing the corresponding function key on the keyboard.

Changing the selected window changes the softkey labels to those of the newly selected window. If the display of softkey labels is not enabled in the newly selected window, then softkey labels are not displayed.

The labels are displayed on top of any windows that extend into the softkey display area. Windows still have full use of the display; just keep in mind that softkey labels are displayed on top of any window(s) extending into the softkey display area.

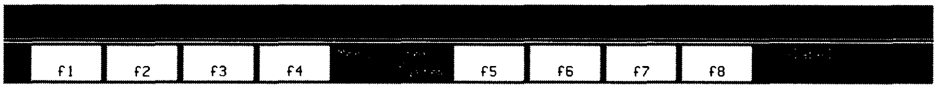


Figure 12-1. Softkey Label Format.

Figure 12-1 illustrates the format of softkey labels as they are displayed on the screen. Key labels 1 through 8 are eight characters wide by two characters high.

The [User]/[System] and [Menu] softkey labels are six by two characters in size. The [User]/[System] label always has a horizontal line that separates its shifted and unshifted modes.

The *wlabel* label is 14 characters wide by two characters high. The window label of the selected window is displayed in the top portion of the label.

The key labels are centered in a pseudo window at the bottom of the display.

The [Menu], [User]/[System], and *wlabel* labels are displayed with their color pair inverted. If the *wlabel* label doesn't fit, then it isn't shown. (This may occur on the HP 300 medium-resolution displays, for instance.)

Return Values

When a function key is pressed, an escape sequence[1] is sent to the selected window's device interface. The sequence returned is always the same, regardless of whether modifier keys such as [Shift] or [CTRL] are held down when the function key is pressed. Table 12-1 defines the escape sequence generated by each softkey.

You can get these softkey values by opening and reading from the window's device interface. Read the "Graphics Window Input" chapter for details on how to read from graphics windows.

Table 12-1. Returned softkey escape sequences.

Key	Escape Sequence Returned
[Menu]	␛ & j @
[System]	␛ & j A
[User]	␛ & j B
[f1]	␛ p
[f2]	␛ q
[f3]	␛ r
[f4]	␛ s
[f5]	␛ t
[f6]	␛ u
[f7]	␛ v
[f8]	␛ w

[1] An escape sequence is a string of characters that starts with the escape character, denoted as ␛ in Table 12-1. The escape character is 033 in octal, 27 in decimal, and 0x1b in hexadecimal.

The escape sequences in Table 12-1 contain spaces only for readability. The actual escape sequences do not contain any embedded spaces.

Keys other than softkeys also return escape sequences. Table 12-2 shows these keys and their return values.

Table 12-2. Other escape keys.

Key	Escape Sequence Returned
Clear line	$^{E}_{C}$ K
Clear display	$^{E}_{C}$ J
Insert line	$^{E}_{C}$ L
Delete line	$^{E}_{C}$ M
Insert char	$^{E}_{C}$ Q
Delete char	$^{E}_{C}$ P
↖	$^{E}_{C}$ h
▲	$^{E}_{C}$ A
▼	$^{E}_{C}$ B
►	$^{E}_{C}$ C
◄	$^{E}_{C}$ D

Turning Softkey Labels On and Off

By default when a graphics window is created, its softkey labels are not displayed. With the *wsfk_mode(3W)* routine, you can enable or disable the display of softkey labels for a given graphics window.

Procedure

To turn a graphics window's softkey labels on or off, call *wsfk_mode*; its syntax is:

```
wsfk_mode(fd, mode)
```

The *mode* parameter detemines whether or not softkey labels are displayed. Following are valid values for *mode*:

- if *mode*=`SFKON`, then the window's softkey labels are displayed when the window is selected;
- if *mode*=`SFKOFF`, then the labels are not displayed.

Note that the example in the next section illustrates the use of this routine.

Changing Softkey Labels

For any graphics window, the *wsfk_prog(3W)* routine sets the softkey label string for a specified function key. It also controls whether or not a horizontal separator bar is placed in the softkey's label area.

Procedure

To change a graphics window's softkey label for a specific function key, simply call *wsfk_prog*; its syntax is:

```
wsfk_prog(fd, key, label, separator)
```

The *key* parameter specifies the function key number; the new softkey label to use is pointed to by *label*; and the *separator* parameter is a horizontal separator flag.

The *key* parameter must range from 1 to 8, corresponding to function keys f1 through f8.

The *label* parameter is a null-terminated character string that replaces the softkey label designated by the *key* parameter. The following rules describe how *label* fills in the softkey label area:

- Each softkey label contains only 16 characters—two rows at eight characters per row. Therefore, if *label* is longer than 16 characters, only the first 16 are used.
- If *label* is less than 16 characters, then blanks are assumed for the remaining characters.
- *label* fills the top row of the label area first, then the second row: the first eight characters of *label* fill the top row; the second eight fill the lower row. Therefore, if you want *label* to appear in the bottom row only, you must pad *label* with eight leading blanks.

The *separator* parameter indicates whether or not to display a horizontal line (separator) between the upper and lower portions of the softkey's label area:

- if *separator*=`SFKSEPON`, then the separator is displayed
- if *separator*=`SFKSEPOFF`, then no separator is displayed.

Presently, shifted softkeys cannot be differentiated from unshifted softkeys. (The returned codes are documented in Table 12-1.)

Example

The following function, *set_gr_labs.c*, resets the softkey labels for a graphics window to those specified in an array parameter, **new_labels**. A separator bar is displayed if specified by the **sep_bar** parameter. Finally, the displaying of softkey labels is enabled for the specified window.

```
#include <window.h>      /* window library definitions      */
set_gr_labs(wfd, new_labels, sep_bar)
int  wfd;                /* window's file descriptor        */
char *new_labels[];      /* new softkey labels              */
int  sep_bar;            /* 0 = no separaotr, 1 = separator */
{
    int  key;            /* softkey number                  */

/*
 * DISPLAY NEW SOFTKEY LABELS FOR A GRAPHICS WINDOW.
 *
 * STEP 1:  Change the softkey labels:
 */
    for (key=1; key<=8; key++)
        if (wsfk_prog(wfd, key, new_labels[key-1], sep_bar) < 0) return(-1);

/*
 * STEP 2:  Display the new softkeys:
 */
    if (wsfk_mode(wfd, SFKON) < 0) return(-1);

    return(0);
}
```

Notes

Term0 Windows 13

Term0 windows (pronounced "term zero") have many of the capabilities of Hewlett-Packard terminals—specifically, HP 2622/2627 terminals. Applications written for these terminals are easily ported to term0 windows.

If your application requires only limited terminal support—simply displaying characters and reading characters typed from the keyboard—then you needn't bother with this chapter.

If, on the other hand, you're interested in more sophisticated terminal capabilities—for example, font management, underlining characters, positioning the cursor at *x,y* locations, and accepting keyboard input without echoing characters—then this chapter will be useful.

The following topics are discussed in this chapter:

- term0 window concepts
- turning the cursor on and off
- turning the softkeys on and off
- defining softkeys
- getting font information
- setting the base/alternate font
- replacing fonts
- converting pixel and character coordinates
- using raw mode

Concepts

This section discusses concepts essential to successfully using and programming term0 windows. Specifically, the following topics are discussed:

- term0 window features
- term0 window escape sequences
- user-definable softkeys
- the term0 font management model
- colors
- raw mode

Term0 Window Features

Each term0 window offers the following features:

- a user-selectable number of lines of display memory
- complete screen editing functions: insert/delete character, insert/delete line, and clear line/display
- absolute and relative cursor positioning
- vertical and horizontal scrolling
- tab and margin settings
- one set of eight (or sixteen) softkeys; you may program both the **soft key labels** (the characters composing the menu at the bottom of the display) as well as the **soft key definitions** (the string of characters that are output when a function key is pressed)
- user-selectable fonts
- underlining and inverse video display enhancements in HP mode (field-oriented)
- color enhancements
- support for TERMCAP entries and the *curses(3X)* library routines
- a standard HP-UX *tty* programmatic interface

Note: term0 windows do **not** offer the following HP 2622 capabilities:

- block mode
- format mode (including protected and unprotected fields)
- memory lock
- programmable time delay
- "local" mode.

Also, term0 windows do not provide HP 2623 or HP 2627 *graphics* capabilities.

Term0 Window Escape Sequences

Almost any term0 window function that can be performed from the keyboard (for example, pressing the [Clear display] key or the [Caps] keys) can be programmed by means of **escape sequences**.

What Is an Escape Sequence?

An escape sequence is a string of characters that begins with an ASCII ESC character[1] (denoted hereafter as $^E{}_C$). Term0 windows interpret escape sequences as *commands* rather than as a sequence of simple display characters.

Escape sequences can be typed from the keyboard (in which case, they affect the window attached to the keyboard), or they can be written (via *write(2)*, *putchar(3)*, etc.) to any term0 window. Note that if you type escape sequences from a shell, they will also be interpreted by the shell and will cause error messages, because the shell won't understand them. (Escape sequences cannot be typed in if you're using the C-shell, *csh(1)*; this is because the [ESC] key has special meaning in this shell.)

Examples

For example, with a term0 window, you can turn off the menu at the bottom of the display by outputting the following escape sequence[2] to the term0 window:

```
Ec & j @
```

The effect is the same as if you had pressed [Menu].

1 Decimal code 27; octal code 33.

2 The escape sequences in this chapter are shown with embedded blanks for readability. When writing an escape sequence to a term0 window, omit the blanks.

The following escape sequence turns on the *user* function keys:

```
␛ & j B
```

The effect is the same as if you had pressed [User].

Commonly Used Escape Sequences

Term0 windows recognize only HP escape sequences. Table 13-1 defines some of the most commonly used escape sequences. The *Term0 Reference Manual* provides more detail on the various supported escape sequences.

Table 13-1. Useful Escape Sequences

Escape Sequence	Its Effect on the Window
`␛ h`	Home the cursor.
`␛ J`	Clear from cursor to bottom of window buffer.
`␛ h ␛ J`	Home up and clear to bottom of window buffer.
`␛ K`	Clear from the cursor to the end of the line.
`␛ & d D`	Turn on underlining.
`␛ & d B`	Turn on inverse video.
`␛ & d @`	Turn off underlining, inverse, or both.
`␛ A`	Move the cursor up one row.
`␛ B`	Move the cursor down one row.
`␛ C`	Move the cursor right one column.
`␛ D`	Move the cursor left one column.
`␛ & a <col> c <row> R`	Position the cursor to <col> and <row> in the window.

User-Definable Softkeys

Windows/9000 supports anywhere from eight up to 16 **user-definable softkeys** per term0 window. A user-definable softkey is comprised of two parts: a **label** and a **definition string**.

Softkey Labels

Softkey labels are descriptive names that correspond to function keys ($\boxed{\text{f1}}$, $\boxed{\text{f2}}$,..,$\boxed{\text{f8}}$). The selected window's softkey labels are displayed at the base of the display if the display of the window's softkey labels has been enabled via the $\boxed{\text{Menu}}$ key or **term0 escape sequences**.

You can define your own softkey labels, for both unshifted **and** shifted softkeys, via term0 escape sequences. This is useful when you want to use the function keys from your application(s), and when you want to assign meaningful names to the function keys. For example, you may want the softkey labels to correspond to menu options that the user can select by pressing a function key.

The following rules apply to softkey labels:

- Changing the selected window changes the softkey labels to those of the newly selected window. If the display of softkey labels is not enabled in the newly selected window, then no softkey labels are displayed.
- The labels are displayed on top of any windows that extend into the softkey display area. Windows still have full use of the display; just keep in mind that softkey labels are displayed on top of any window(s) extending into the softkey display area.
- Figure 13-1 shows a term0 window with softkeys enabled; it illustrates the format of softkey labels as they are displayed on the screen. Key labels 1 through 8 are eight characters wide by two characters high.

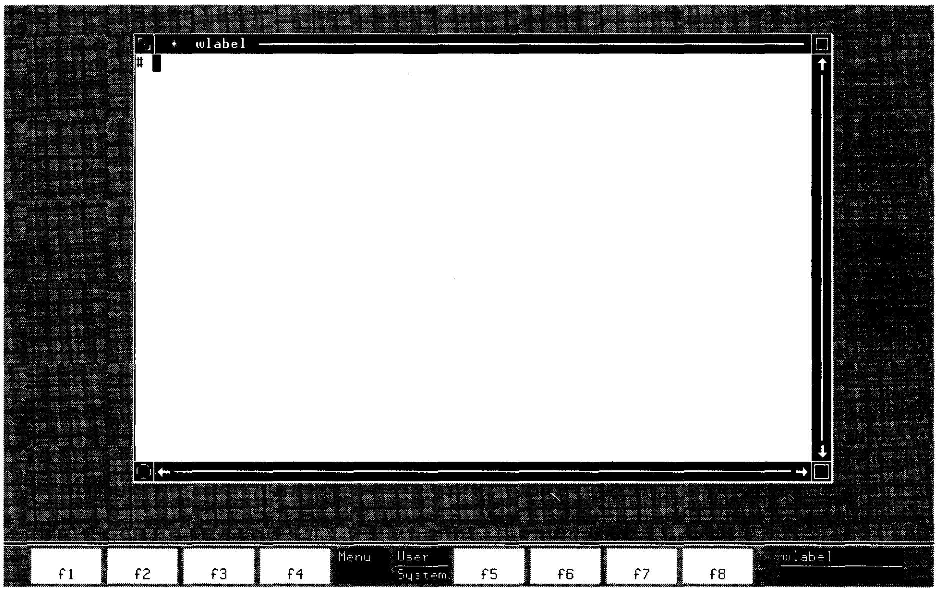


Figure 13-1. Softkey Label Format.

- The *wlabel* label is 14 characters wide by two characters high. The window label of the selected window is displayed in the top portion of the label.
- The key labels are centered at the bottom of the display.
- The [Menu], [User]/[System], and *wlabel* labels are displayed with their color pair inverted. If the *wlabel* label doesn't fit, then it isn't shown. (This may occur on the HP 300 medium-resolution displays, for instance.)

Softkey Definition Strings

A softkey definition is a string of characters, up to 80 characters in length, that is returned when the user presses a function key. Using term0 escape sequences, you can create a softkey definition string for either unshifted or shifted function keys. Then when a function key is pressed, the definition string is sent as standard input from the device interface of the selected window—it's as if the definition string is actually typed at the keyboard, even though only the function key is pressed.

Each function key has a default escape sequence value that is returned if no definition string has been defined. Table 9-1 in chapter 9, "Graphics Softkeys," shows the default values that are returned for each function key.

The Term0 Font Management Model

Each term0 window has a **base** and **alternate** font. By default, when you write characters to a term0 window, the characters are displayed in the base font. You can cause characters to be written in the alternate font by sending an ASCII SO[1] character (CTRL-N); to return to the base font, send an ASCII SI[2] character (CTRL-O) or a carriage-return.

Note, however, that you are not limited to just two fonts in a window at a time—you can have up to eight fonts displayed in a window simultaneously. The fonts must all be of the **same size** (in pixel width and height). However by using commands (e.g., *wfont(1)*) or window library routines, you can switch to a different font size, in which case the term0 window's size is automatically readjusted to accomodate the new font size.

The simultaneous display of eight fonts is possible because of the **font cache**. The font cache is an array that holds font information in memory. Each term0 window has a font cache which holds font definitions for up to eight same-sized fonts. Font definition information is kept permanently in **font files**; when needed, font files are **loaded** into the font cache via term0 font management routines. (The `fontstruct` structure in */usr/include/fonticon.h* defines the format of a font file.) Each font in the cache has a unique **font id** which distinguishes that font from others in the cache.

Font files are stored in **font directories**. Font files for all fonts of the same size—i.e., same pixel width and height—are stored in specific font directory. The size of fonts in a given font directory is denoted by the font directory's name, and all font directories are found in the directory specified by the window system environment variable WMFONTDIR (normally */usr/lib/raster*). To see the various font sizes supported on your system, simply list the $WMFONTDIR directory; for example:

```
ls $WMFONTDIR Return
```

may list the following font directories:

```
12x20      18x30      7x10      8x16      L6x15
```

Directory *12x20* contains font files for all fonts that are 12 pixels wide by 20 pixels high; directory *8x16* contains font files for all 8-by-16-pixel fonts; and so on.

1 Decimal 14; octal 016.
2 Decimal 15; octal 017.

After you've loaded any needed fonts into the font cache via term0 font management routines or escape sequences, you must **activate** one of the fonts as the base font; you may also optionally activate one of the fonts as the alternate font. Term0 font management routines or term0 escape sequences can be used to activate fonts. However, we suggest that you use term0 font management routines to activate fonts, because escape sequences are sometimes too ambiguous to specify exactly which font in the cache to activate.

Once base and alternate fonts are activated, you can cause characters to be written in the alternate font by sending an SO character to the window; you can return to the base font by sending SI or CR (carriage-return).

In addition to loading and activating a font, you can **replace** a font in the cache with a font from a font directory. This is useful if the font cache is full and you wish to use another font—an unused font in the cache can be replaced with the needed font from the font directory.

Colors

Every character displayed in a term0 window has a **color pair attribute**. The color pair attribute determines the foreground and background colors to use when displaying the character.

The color pair attribute is essentially a pointer to a **color pair** in the term0 window's **color pair table**. A color pair defines foreground and background colors to use when displaying a character; the color pair table contains eight color pairs. Table 13-2 shows the default color pair table, i.e., the color pair table for newly created term0 windows.

Table 13-2. Default color pair table.

Color Pair	Foreground Color	Background Color
0	White	Black
1	Red	Black
2	Green	Black
3	Yellow	Black
4	Blue	Black
5	Magenta	Black
6	Cyan	Black
7	Black	Yellow

Color pair 0 (white characters on black background) is the default color pair used when displaying text in term0 windows. Certain color escape sequences allow you to use different color pairs when displaying text. In other words, you're not restricted to using color pair 0; you can display text using any of the color pairs defined in Table 13-2.

Each color in the color pair table is defined by mixing different amounts of red, green, and blue colors (this is known as RGB technology). With term0 windows, the red, green, and blue values can be on or off. The result is that eight different colors can be created. Table 13-3 defines the colors produced by each possible combination of red, green, and blue.

Table 13-3. RGB color definitions.

R	G	B	Resulting Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

Other color escape sequences allow you to redefine the color pairs. For example, if you don't like the default white characters on a black background, you can redefine the default color pair (0) to be black characters on a yellow background.

Note that if you change a color pair, then any visible characters that were displayed using the changed color pair will automatically be redisplayed using the new color pair values.

Note

The "Windows/9000 Escape Sequences" chapter of the *Term0 Reference Manual* provides detailed information on using color escape sequences.

Turning the Cursor On and Off

Each term0 window has its own cursor, to mark where the next display character will appear in the term0 window. You may want to turn the cursor off to avoid distracting the user, especially if your application does a lot of cursor positioning.

As an added benefit, displaying rows of characters with the cursor off is slightly **faster** than with the cursor on.

Procedure

Turning the Cursor On

To turn the cursor back on, use:

```
    ␛ * d Q
```

Turning the Cursor Off

To turn off the cursor, use the following escape sequence:

```
    ␛ * d R
```

Turning Softkeys On and Off

HP escape sequences are also used to enable or disable the display of the softkey menu for a term0 window.

Procedure

Turning the Softkeys On

To ensure that the softkeys are displayed in a term0 window, use the following HP escape sequence:

$^{E}{}_{C}$ & j B

Turning the Softkeys Off

To stop displaying a term0 window's softkey menu, use the following escape sequence:

$^{E}{}_{C}$ & j @

Defining Softkeys

As mentioned in the "Concepts" section, softkey labels and definition strings can be defined via HP escape sequences. By sending (writing) the appropriate escape sequence to a term0 window, you can define the softkeys for that window.

Procedure

An HP escape sequence for a softkey typically specifies three items:

- the function key that you wish to define (1-16)
- the key label to appear in the menu at the bottom of the display (1 to 16 characters)
- the string of characters to be output when the user presses the function key (up to 80 characters).

In general, the escape sequence used to define softkeys conforms to the following format; the embedded blanks are shown only to enhance readability; **do not include blanks in the actual escape sequence**:

$^{E}_{C}$`&f 0a` *i*`k` *j*`d` *k*`l` *label definition*

The `0a`, *i*`k`, *j*`d`, and *k*`l` parameters can appear in any order; just remember that the last parameter must be capitalized. Table 13-4 describes each parameter in detail.

Table 13-4. Parameters to softkey defintion escape sequence.

Parameter	Definition
$^{E}_{C}$&f	Tells the window that the escape sequence redefines a softkey.
0a	Says that the softkey will be a **normal** softkey. This is the only softkey type supported by Windows/9000.
*i*k	The softkey to set the label and definition strings for is denoted by *i*. *i* is an ASCII integer string from 1 to 16.
*j*d	The length of the **label** string is denoted by *j*. *j* is an ASCII integer string.
*k*l	The length of the **definition** string is given by *k*. *k* is an ASCII integer string.
label	The label string. Should contain exactly as many characters as specified by *j*d.
definition	The definition string. Should contain exactly the number of characters specified by *k*l.

Note: You should refer to the *Term0 Reference Manual* as you read this section; the escape sequence is described in more detail in that manual.

If you specify a key greater than eight (8), then the term0 window will display a separator bar between the shifted and unshifted representations for the softkey—the lower row for the *unshifted* function key, the upper row for the *shifted* function key. If the lower half of the key label is empty, and there is no horizontal separator bar between the upper and lower halves, then pressing the shifted function key is no different than pressing the unshifted function key.

Softkey definitions take effect as soon as the escape sequence is written to the window.

Examples

Entering the following escape sequence from a term0 window renames the softkey label for f8 to `DateTime`; the definition string is set to the HP-UX command `date` terminated with a carriage-return. After you have typed in the following sequence, the softkey for f8 will be redefined as described. Whenever you press this key, or boink the locator over its displayed softkey label, the *date* command will be excecuted. The blanks in the escape sequence are included only to enhance the readability of the sequence; **do not include the blanks when typing this sequence**.

```
ESC&f 0a 8k 8d 5L DateTime dateReturn
```

The following *write(2)* statement when called from a program redefines the softkey for function key **f9** (Shift-f1). Its label is set to `INVERSE` and its definition is set to the escape sequence for inverse video.

```
write(1, "\033&f0a9k7d4LINVERSE\033&dB", 21);
```

Getting Font Information

You can use term0 window library routines to obtain information about the fonts being used in a term0 window. For example, you can determine:

- the current font size—the size of the fonts stored in the font cache
- the font ids of fonts stored in the cache, given their names
- font names of fonts stored in the cache
- the font id of the current base font
- the font id of the current alternate font

Procedure

To get font information for a term0 window, call the appropriate term0 library routine. These routines all require a file descriptor (*fd*) for the term0 window's device interface; the remaining parameters vary:

- `fontsize_term0(`*fd, wptr, hptr*`)` — returns the current font size being used in the window. *wptr* and *hptr* are pointers to integers that will contain the pixel width and height of the current font size. For example:

  ```
  int     font_w, font_h;
     :
  fontsize_term0(fd, &font_w, &font_h);
  ```

  gets the current font width and height and puts them in `font_w` and `font_h`, respectively.

- `fontgetid_term0(`*fd, fontpath*`)` — returns the font id of the font specified by the path name pointed to by *fontpath*, **if** the font exists in the font cache. For example:

  ```
  int    font_id;
     :
  font_id = fontgetid_term0(fd, "/usr/lib/raster/8x16/lp.8U");
  ```

  sets `font_id` to the font id of the font specified by the second argument, **if** that font is currently in the font cache.

- `fontgetname_term0(`*fd, id*`)` — this function returns a pointer to a static storage area containing a null-terminated character string that represents the path name of the font denoted by *id*, **if** the font exists in the font cache.

  ```
  char     *font_path;
  int      font_id;
     :
  font_path = fontgetname_term0(fd, font_id);
  ```

  makes `font_path` point to the path name of the font specified by `font_id`, **if** the font exists in the font cache.

- `basefont_term0(`*fd, id*`)` — returns the base font's id when *id* is set to `GETFONTID`. For example:

  ```
  #include <window.h>
     :
  int      base_font_id;
     :
  base_font_id = basefont_term0(fd, GETFONTID);
  ```

  sets `base_font_id` to the font id of the base font.

- `altfont_term0(`*fd, id*`)` — returns the alternate font's id when *id* is `GETFONTID`. For example:

  ```
  #include <window.h>
     :
  int      alt_font_id;
     :
  alt_font_id = altfont_term0(fd, GETFONTID);
  ```

  sets `alt_font_id` to the alternate font's id.

Setting the Base/Alternate Font

The base and alternate fonts for any window can be set via term0 window library routines. This section describes the use of these routines.

Procedure

The procedure for setting the base and/or alternate font is straightforward:

1. A font cannot be made the base or alternate font until it is loaded into the font cache. Therefore, if the desired base/alternate font is not already loaded, you must load it. The *fontload_term0(3W)* routine loads a font into the font cache **and** returns the font's id; its syntax is:

   `fontload_term0(`*fd, fontpath*`)`

   The *fontpath* parameter is the path name of the font file to load into the font cache. You may want to use the *fontgetid_term0* routine to make sure that loaded font doesn't already exist in the font cache before loading it.

   Note that if the font cache is already full, then you must replace, or swap, base and alternate fonts; this is discussed in the section "Replacing Fonts."

2. Once a font file is loaded into the font cache, you can activate it as the base or alternate font. The *basefont_term0(3W)* routine activates a font as the base font; its syntax is:

   `basefont_term0(`*fd, id*`)`

   The *id* parameter is the font id of the font. The font id can be obtained when the font is loaded (as in step 1), **or** it can be obtained via the *fontgetid_term0* routine.

   The *altfont_term0(3W)* routine activates a font as the alternate font; its syntax is:

   `altfont_term0(`*fd, id*`)`

   The *id* parameter is the font id of the font to activate as the alternate.

Example

The following program, *basalt.c*, loads the bold, 8-by-16-pixel font into the font cache; it also loads the italic, 8-by-16-pixel font. It then activates the bold font as the base font and the italic font as the alternate.

Note that this program works on the window from which it is invoked. In other words, if you invoke this program from a term0 window, the base and alternate fonts for that window will be reset as described above.

This is possible because the window's device interface is automatically opened for standard input and output when the program (process) starts execution. That is, the window's device interface is opened as standard output **if** standard output isn't redirected when the program is invoked. Therefore, you **must initialize (via *winit(3W)*) standard input (or output) before calling the term0 window routines with the standard input (or output) file descriptor.**

```
#include <window.h>      /* window constant definitions */
#define  WFD_STDOUT  1  /* standard output filedes     */

#define  BOLD_8x16_FONT "/usr/lib/raster/8x16/lp.b.8U"  /* base font */
#define  ITAL_8x16_FONT "/usr/lib/raster/8x16/lp.i.8U"  /* alt  font */

main()
{
    int   base_id, alt_id;  /* base and alternate font ids */

/*
 * The term0 window is opened as stdout when the window is created.
 * We will be using the stdout file descriptor (WFD_STDOUT) as the
 *      file descriptor for the window's device interface.
 * Since the window's device interface is automatically opened when
 *      the process is spawned from the window, all that's left to
 *      do is initialize the device interface so that you can use
 *      window library routines on the window using the stdout file
 *      descriptor.
 */
    if (winit(WFD_STDOUT) < 0)
    {
        fprintf(WFD_STDOUT, "error initializing stdout\n");
        exit(1);
    }
/*
 * STEP 1:  Load the base font into the font cache.
 *
 *      Before loading the font, check to see if it already
 *      exists in the font cache.  If so, then skip this step;
 *      otherwise, go ahead and load the font.
 */
   if ((base_id = fontgetid_term0(WFD_STDOUT, BOLD_8x16_FONT)) < 0)
       if ((base_id = fontload_term0(WFD_STDOUT, BOLD_8x16_FONT)) < 0)
       {
           fprintf(WFD_STDOUT, "error loading base font\n");
           exit(1);
       }
/*
 * STEP 2:  Activate the font as the base font:
 */
   if (basefont_term0(WFD_STDOUT, base_id) < 0)
   {
       fprintf(WFD_STDOUT, "error activating the base font\n");
       exit(1);
   }

/*
 * STEP 1:  Load the alternate font into the font cache.
 *
 *      Before loading the font, check to see if it already
 *      exists in the font cache.  If so, then skip this step;
 *      otherwise, go ahead and load the font.
 */
   if ((alt_id = fontgetid_term0(WFD_STDOUT, ITAL_8x16_FONT)) < 0)
       if ((alt_id = fontload_term0(WFD_STDOUT, ITAL_8x16_FONT)) < 0)
       {
           fprintf(WFD_STDOUT, "error loading alternate font\n");
           exit(1);
       }
/*
 * STEP 2:  Activate the font as the alternate font:
 */
   if (altfont_term0(WFD_STDOUT, alt_id) < 0)
   {
       fprintf(WFD_STDOUT, "error activating the alternate font\n");
       exit(1);
   }
}
```

Replacing Fonts

The font cache holds a maximum of eight fonts. Trying to load another font when the cache is already full results in an error. With term0 window routines, you can replace a font in the cache with a font from the font directories, thus avoiding the problem of loading too many fonts in the cache.

Procedure

Two kinds of font replacement can be performed using term0 window routines: you can replace any single font in the cache, or you can replace both the base and alternate fonts.

Replacing a Single Font in the Cache

To replace a single font in a term0 window's font cache, use the *fontswap(3W)* routine which has the following syntax:

```
fontswap_term0(fd, newpath, oldid)
```

The cache font represented by *oldid* is replaced with the font file represented by *newpath*, a null-terminated path name. The new font's id is returned by the routine.

Any characters that were displayed in the old font will be redisplayed in the new font.

Replacing the Base and Alternate Fonts

To replace a term0 window's base and alternate fonts, use the *fontreplaceall_term0(3W)* routine. Its syntax is:

```
fontreplaceall_term0(fd, bfpath, afpath)
```

The current base and alternate fonts are removed from the cache and replaced with *bfpath* and *afpath*. *bfpath* and *afpath* are both null-terminated path names of font files.

This routine does not return font ids for the new base and alternate fonts. You must obtain their font ids by using the *fontgetid_term0* routine (or the *basefont_term0* and *altfont_term0* routines).

Examples

The following function, *base_load.c*, is used to load and activate a base font, specified by the `fontname` parameter. It checks to see if the font is already loaded; if so, it doesn't load it. Otherwise, it tries to load the font; if the font cache is full, then it replaces the current base font with the new base font.

The source for this function is found in the *man_examples* directory.

```
#include <window.h>     /* window constant definitions */
base_load(wfd, fontname)
int  wfd;                      /* window's file descriptor    */
char *fontname;                /* path name of new base font  */
{
    int base_id;               /* base font id                */
    int new_base_id;           /* font id of new base font    */


/*
 * Load the new font into the font cache:
 *
 *      Before loading the font, check to see if it already
 *      exists in the font cache.  If so, then skip this step;
 *      otherwise, go ahead and load the font.  If the font
 *      cache is full, swap this font with the current base
 *      font.
 */
    if ((new_base_id = fontgetid_term0(wfd, fontname)) < 0)
        if ((new_base_id = fontload_term0(wfd, fontname)) < 0)
        {
                base_id = basefont_term0(wfd, fontname, GETFONTID);
                if ((new_base_id = fontswap_term0(wfd, base_id)) < 0)
                {
                        fprintf(wfd, "error loading the new base font\n");
                        exit(1);
                }
        }
/*
 * Activate the font as the base font:
 */
    if (basefont_term0(wfd, new_base_id) < 0)
    {
        fprintf(wfd, "error activating the  base font\n");
        exit(1);
    }
}
```

The following code segment replaces the current base and alternate fonts. The new base font is 18-by-30-pixel pica; the new alternate font, 18-by-30-pixel math font. Note that the window size will be changed if this font size is different that the current font size.

```
/*
 * Replace all the fonts in the font cache with a new base and
 *      a new alternate font.
 */
    if ((fontreplaceall_term0(wfd, "/usr/lib/raster/18x30/pica.8U",
                                   "/usr/lib/raster/18x30/math.0M")) < 0)
    {
        perror("fontreplaceall_term0 failed");
        exit(1);
    }
```

Converting Pixel and Character Coordinates

Window routines can be used to convert from pixel *x,y* coordinates to *column, row* coordinates (and vice versa) with term0 windows.

Procedure

Converting Pixels to Characters

The *fromxy_term0(3W)* routine converts pixel coordinates to character coordinates; its syntax is:

```
fromxy_term0(fd, x,y, colptr,rowptr)
```

The *x,y* parameters specify the pixel coordinates to be converted. When *fromxy_term0* is finihsed, *colptr,rowptr* will point to integers containing the equivalent column and row coordinates.

Converting Characters to Pixels

The *toxy_term0(3W)* routine converts character coordinates to pixel coordinates; its syntax is:

```
toxy_term0(fd, xptr,yptr, col,row)
```

The *xptr,yptr* parameters will point to integers containing pixel coordinates as converted from the *col,row* parameters.

Using Raw Mode

The default setting of a term0 window is **cooked mode**, which has the following characteristics:

- The editing keys act **locally**. For example, when the user presses the down-arrow key, the cursor will move one row at a time down the term0 window. An application waiting for keyboard input will have no knowledge of—and no control over—the cursor. In cooked mode, the editing keys **don't transmit**.
- The typing keys **echo** on the display. For example, when the user presses the unshifted [A] key, the term0 window will display the character **a**.

In **raw mode**, your application has much greater control over keyboard input. For example, your application might define the [Tab], [►], and space bar keys to move the cursor to the next selection in a menu, while "locking out" other keyboard responses.

Note that the following **system keystrokes** are not affected by raw mode—they perform their normal system functions:

- [Menu]—toggles the function key labels off and on.
- [User]/[System]—switches between the *user* (application program) key labels and the *system* key labels.
- [Select]—selects the active application window.

The following header files contain the data type declarations to set a term0 window to raw mode and to perform terminal i/o:

```
#include <stdio.h>
#include <signal.h>
#include <termio.h>
```

In addition, the *tty(7)* entry in the *HP-UX Reference* contains more detailed documentation of the `termio` structure and the meaning of its various fields.

Changing to Raw Mode

The following *setraw* routine sets the term0 window to raw, no-echo mode:

```
setraw ()
{
  struct termio t;

  ioctl (0, TCGETA, &t);     /* get the TTY parameters */
  t.c_cc[VMIN] = 1;
  t.c_cc[VEOL] = 1;
  t.c_lflag &= ~(ICANON | XCASE | ECHO);
  t.c_lflag |= ISIG;
  ioctl (0, TCSETA, &t);      /* set the new parameters */
  write (1, "\033&s1A", 5);  /* set XMIT straps        */
  write (1, "\033*dR", 4);   /* turn the cursor off    */
  signal (SIGINT, fixup);    /* enable interrupt       */
  signal (SIGQUIT, SIG_IGN); /* disable quit           */
}
```

The `termio` structure above, defined in */usr/include/termio.h*, is common across all HP-UX terminal devices.

The *setraw* routine uses one *ioctl(2)* call to get the current `termio` values, turns off **canonical** processing, turns off keyboard echo, and then uses a second *ioctl* to set the `termio` structure to the new values.

Notice the two *signal(2)* calls appearing at the end of the routine. The first, with SIGINT, sets up an interrupt—if the user presses the CTRL-C keystroke (or whatever `INT` is set to), execution will jump to the *fixup* routine (see below).

The second *signal* call, with SIGQUIT, tells the program to ignore CTRL-| if it is pressed. You could instead define SIGQUIT to jump to your own interrupt handler.

Handling Interrupt

The SIGINT signal, appearing in the *setraw* routine above, enables an interrupt routine named *fixup*.

fixup will be called if the user presses CTRL-C (or whatever `INT` is set to):

```
fixup()
{
        setcook();
        exit(0);
}
```

The purpose of *fixup* is to "clean up" the term0 window and to cause the application to terminate normally.

The *setcook* routine appears below.

Returning to Cooked Mode

When your application exits, or whenever you want to restore the term0 window to its initial state, you can use the following *setcook* routine:

```
setcook()
{
    struct  termio t;

    ioctl (0, TCGETA, &t);
    t.c_cc[VEOF] = CEOF;
    t.c_cc[VEOL] = CNUL;
    t.c_iflag |= (BRKINT | IGNPAR | ISTRIP | ICRNL | IXON | IXOFF);
    t.c_oflag |= OPOST;
    t.c_cflag &= ~CSIZE;
    t.c_cflag |= CS8;
    t.c_lflag |= ECHO;
    t.c_lflag |= (ISIG | ICANON);
    ioctl (0, TCSETAW, &t);
    write (1, "\033&s0A", 5);  /* clear xmit straps */
    write (1, "\033*dQ", 4);   /* turn the cursor on! */
}
```

A Word About the Transmit Function

The **transmit strap**, a software "switch" that is initially cleared, determines whether an escape sequence is *transmitted* to an application or is handled *locally* by the term0 window (unknown to the application).

One of the purposes of the *setraw* routine above is to enable your application to receive **all** editing keystrokes—hence, that routine includes a *write* statement to set the transmit strap. The following escape sequence does the trick:

```
EC & s 1 A
```

The next escape sequence restores normal (or local) handling of escape sequences:

```
EC & s 0 A
```

We'll send this escape sequence to the term0 window when we want to restore the term0 window to cooked mode.

Running in Raw Mode

With your term0 window in raw mode, certain terminal functions are no longer handled for you automatically. Your application must handle them on its own.

This section mentions some of the more important responsibilities left to your application.

Echoing Characters

The most common need is to echo characters as they're typed. For example:

```
write(1,&ch,1);
```

writes one character to standard output.

Without an explicit *write* statement, the user can't see what is typed at the keyboard.

Destructive Back Space

In raw mode, [Back space] simply moves the cursor to the left one space. To create a *destructive* back space:

1. Move the cursor one space to the left by sending an ASCII BS character.[1]
2. Write a space (to "white out" the next character).[2]
3. Again move the cursor one space to the left.

The net effect is a back space with erasure. But that's not all!

In cooked mode, your application will never *see* the user's input line until the user presses [Return]. Any erasures will be handled for you and automatically eliminated from the key buffer so that your application receives only the corrected line.

In raw mode, however, your application receives characters as they are typed. If the user presses [Back space], your application will detect an ASCII BS character just as it would detect a printable character. In the event of a back space, it's up to your keyboard handling routines to "erase" (or remove) the undesired character from your own input buffer.

New Lines and Carriage Returns

Normally in cooked mode, when a new line character (ASCII LF)[3] is written to a term0 window, a carriage return/linefeed sequence is generated.

In raw mode, you must explicitly write the two characters to the window:

```
write(1,"\015\012",2);
```

The effect is to position the cursor at the left margin of the next line.

[1] Decimal code 8; octal code 10.
[2] Decimal code 32; octal code 40.
[3] Decimal code 10; octal code 12.

Escape Sequence Parsing

Recognizing escape sequences as they are transmitted from the keyboard can involve a lot of code. The simplest way is to set up an input parser of the form:

```
#define ESC '\033'

char parsekey()
{
        char ch;
        read(0, &ch, 1);
        if (ch == ESC)
                {
                read(0, &ch, 1);
                   :
                process the remainder of the
                   escape sequence...
                   :
                }
        else return(ch);
}
```

The *parsekey* routine can include a large *switch* statement, after it has detected an $^{E}_{C}$ character, to pick out the individual keystrokes that the user might have pressed.

Blocked vs. Unblocked Reads

There are two common ways to accept terminal input: using **blocked** reads and **unblocked** reads.

The previous *setraw* routine initialized the term0 window to accept blocked reads. That is, when reading from the keyboard, your program will **block**, or wait, until a key is pressed. (The *read* won't return.)

You may want to use **unblocked** reads instead. In an unblocked read, the *read* statement returns immediately, whether or not there's a key in the key buffer.

The advantage of an unblocked read is that your application doesn't have to be hung up waiting for the user—it can go off and do other things.

It's easy to set up the term0 window for unblocked reads: Simply define both the `VMIN` and the `VTIME` values in your `termio` structure to be *0* (zero) inside the *setraw* routine.

The main disadvantage is that **polling the keyboard**, as happens during unblocked reads, can be a drain on system resources. If a number of applications are all polling the keyboard at the same time, then system response time can become degraded.

A compromise solution is to set up a signal handler that is executed when the window in which your application is executing becomes detached (not selected), etc. You can start off by polling the keyboard, but when your application senses that its window is detached from the keyboard, it can return to blocked reads.

Another alternative is to use the HP-UX system call *select(2)*. Or, you can limit the number of times your program polls the keyboard.

Notes

The Fast Alpha Library 14

The fast alpha display library provides high-performance alpha (textual) capabilities with graphics windows and bit-mapped graphics displays. For example, you can write text and manipulate fonts, you can clear a portion of a window/display, or you can scroll part of a window/display. The following topics are covered in this chapter:

- concepts essential to using fast alpha routines
- initializing and terminating the fast alpha environment
- changing the fast alpha environment
- cursor control
- writing characters
- font manipulation
- clearing part of a window
- scrolling part of a window.

Concepts

This section discusses concepts essential to understanding the use of fast alpha routines. The following topics are discussed:

- the fast alpha programming model
- cursor positioning
- character enhancements
- fast alpha rectangles

Note

Definitions of fast alpha constants and structures are found in the file */usr/include/fa.h.*

Also, programs that call fast alpha routines require that both the fast alpha **and** font manager libraries be linked.

Programming Model

Fast alpha routines can be used with any graphics window **or** any bit-mapped display supported by Windows/9000. In other words, you can call fast alpha routines to work with bit-mapped displays or windows on bit-mapped displays. It's the same concept as using Starbase graphics routines with either a bit-mapped display or graphics windows. Certain tasks must always be performed in programs that call fast alpha routines.

First, Get the File Descriptor

Fast alpha routines require the file descriptor of the window or the display's opened device interface. The following rules should be followed:

- When using fast alpha routines with a graphics window, you must first start communication with the graphics window (as described in the "Concepts" section of the "Window Manipulation" chapter). The file descriptor returned from performing a graphics open (*gopen(3G)*) is required by fast alpha routines.
- When using fast alpha routines with a bit-mapped display, you must obtain a file descriptor for the display by performing a graphics open (*gopen(3G)*) on the display's device interface. The file descriptor returned from *gopen*ing the device interface is the one used by fast alpha routines.

Note

If you are using fast alpha routines to combine text and graphics in the same window, then you should obtain separate file descriptors for fast alpha routines and Starbase routines; that is, you must open the graphics window once for fast alpha routines and once for Starbase routines.

Getting separate file descriptors ensures that fast alpha routines work predictably. You can use the same file descriptor for both fast alpha and Starbase routines, but the results will be unpredictable.

Initializing/Terminating the Fast Alpha Environment

Before calling any other fast alpha routines, you must **initialize** the fast alpha environment for the window/display on which the routines operate. Once the fast alpha environment is initialized, you can call fast alpha routines that manipulate the window/display. When you are finished using the fast alpha routines with a window/display, the fast alpha environment must be **terminated**. (The section "Initializing/Terminating the Fast Alpha Environment" contains more information on how to do this.)

Note that you can use window library routines on the window before, during, and after initializing and terminating the fast alpha environment; however, you can use fast alpha routines only between initializing and terminating the environment.

Don't Forget to Close

The final task that must be performed in fast alpha programs is closing the device interface of the window or display:

- If you're operating on a graphics window, then stop communication with the window (as described in the "Concepts" section of the "Window Manipulation" chapter).
- If you're operating on a bit-mapped display, then perform a graphics close (*gclose(3G)*) on the display's device file.

Cursor Positioning

To provide you with a more intuitive interface, the screen position for placing characters is specified by character column and line, rather than display pixels. The leftmost column of the display or window is column 0; the topmost row is row 0.

The pixel coordinate equivalents of column-row depend on the size of the current font—the smaller the font, the smaller the pixel coordinates; the larger the font, the larger the pixel coordinates. Proportionally spaced fonts can be used, but the results are unpredictable because fast alpha routines will use the height and width of the biggest character for determining spacing.

Character Enhancements

Each character may be enhanced with one or more video enhancements. Inverse video and underlining are currently the only enhancements supported by Windows/9000.

When the fast alpha environment is initialized, font colors default to white foreground and black background—white characters on a black background. Through fast alpha routines, you can redefine the font foreground and background colors.

Fast Alpha Rectangles

Many fast alpha routines reference **rectangles**. Rectangles are your means of specifying a particular subset of the window/display area (in columns and lines) for a fast alpha operation involving more than one line. The rectangle structure is defined in */usr/include/fa.h* as:

```
struct fa_coordinate {
    int x,y;
};

/*
 * A rectangle includes [origin] but does
 * not include [corner]
 */

struct fa_rectangle {
    struct fa_coordinate origin;
    struct fa_coordinate corner;
};
```

As the comment indicates, the lower-right-corner character is not included in the rectangle as is consistent with C-language arrays. For example, if `fa_rect` is defined as:

```
struct fa_rectangle fa_rect;     /* fast alpha rectangle structure */
```

then the following rectangle:

```
                    columns

                    012345
                   0......
                   1..XX..
        rows       2..XX..
                   3..XX..
                   4......
                   5......
```

is denoted by:

```
fa_rect.origin.x = 2;
fa_rect.origin.y = 1;
fa_rect.corner.x = 4;
fa_rect.corner.y = 4;
```

This method of display access is fairly low-level, and you may want to build a "friendlier" interface upon this base. The main purpose of the fast alpha routines is to provide you with a fast and intuitive method for getting alpha information in a graphics window or on the display.

Initializing/Terminating the Fast Alpha Environment

The *fainit(3W)* routine initializes the fast alpha environment for a window or display device; the *faterminate(3W)* terminates a window's (or display's) fast alpha environment—i.e., it releases resources allocated when *fainit* was called.

Procedure

Initializing the Fast Alpha Environment

To initialize a graphics window's (or display's) environment, simply call *fainit*; its syntax is:

```
fainit(gfd, driver)
```

The *driver* parameter should be set to `FAWINDOW` for graphics windows and bit-mapped displays supported by Windows/9000. The *gfd* parameter is the file descriptor for the graphics window or display.

Initializing the fast alpha environment causes environment information to be allocated for the window or display. This information affects how fast alpha routines work with the window or display. You can inquire and change this information via fast alpha routines (discussed in the next section, "Changing the Fast Alpha Environment").

Terminating the Fast Alpha Environment

To terminate a graphics window's (or display's) environment, call *faterminate*; its syntax is:

```
faterminate(gfd)
```

Calling this routine causes fast alpha environment information to be deallocated for the window or display device represented by *gfd*. (In order to use fast alpha routines with the window again, you must call *fainit* again.)

Example

The following code segment exemplifies the structure of programs that call fast alpha routines.

```
   :

/*
 * Establish communication with the window with which
 *        fast alpha routines will be used.
 */
      gfd = gopen(device_path, OUTDEV, device_name, INIT);

   :

/*
 * Initialize the fast alpha environment for the window.
 */
      fainit(gfd, FAWINDOW);


   :

/*
 * Now other fast alpha routines can be called to write
 *          text to the window, scroll it, clear it, etc.
 */

   :

/*
 * Finally, you must terminate the fast alpha environment.
 */
      faterminate( gfd );

   :
```

Changing the Fast Alpha Environment

As described in the previous section, fast alpha environment information is allocated when the environment is initialized. This information affects the manner in which fast alpha routines work with graphics windows and bit-mapped displays.

The exact information maintained in the fast alpha environment is defined by the *fainfo* structure in the header file *fa.h.* Table 14-1 (which follows "Performance Considerations" in this section) briefly describes each of *fainfo*'s fields; for more information on this structure and its values, see *fa.h* and the *HP-UX Reference* pages for *fasetinfo(3W).*

Procedure

Getting Environment Information

To get the current fast alpha environment for a window or display, simply call *fagetinfo*; its syntax is:

```
fagetinfo(gfd, fainfoptr)
```

The *fainfoptr* parameter is a pointer to a *fainfo* structure as defined in *fa.h.* After calling *fagetinfo*, the fields of the structure will the fast alpha environment values for the window (or screen) specified by *gfd.*

Setting Environment Information

To set fast alpha environment parameters for a window or display, call *fasetinfo*; its syntax is:

```
fasetinfo(gfd, fainfoptr)
```

The *fainfoptr* parameter is a pointer to a *fainfo* structure containing the new values for the environment. Note that only the following parameters can be set via this routine:

- `defaultenhancements`
- `clearbeforewrite`
- `colormode`
- `makecurrent`.

Performance Considerations

- You can set `defaultenhancements` to a different value, but remember that it initially contains the value that makes the fast alpha library work most efficiently (i.e., `FAOFF`). Therefore, changing this may degrade system performance.
- The default value for `clearbeforewrite` is `TRUE` which causes the screen background to be cleared before writing any characters. This ensures that the space where characters are to be displayed is properly cleared, so that new characters are readable. However, you can change this value to `FALSE`, in which case the background is not cleared before writing, and you are responsible for controlling the background area.
- Changing the `colormode` parameter to `FACOLOR` will cause the fast alpha routines to run slower. Black and white colors (the default) cause the system to run faster.
- To increase the speed of fast alpha routines, you may wish to suppress the updating of the display until several write operations are queued. Then when updating is desired, simply signal the fast alpha environment to update by setting the `makecurrent` field to `MCALWAYS`. Queued operations will be displayed at that time. Then reset `makecurrent` so that operations will queue up—i.e., set the bits in `makecurrent` that will suppress screen updates (see *fa.h*). By doing this you are making effective use of the Starbase buffering facility.

  The default value is `MCALWAYS`, which updates the screen after every fast alpha call and may degrade system performance (compared to queueing).

Table 14-1. The fainfo structure[1].

Field	Description	Range
`size`	This is a rectangle structure as defined in the "Concepts" section; it defines the screen size.	The limits of the window's (or display's) size.
`capabilities`	The contents of this field may be used to detect what additional capabilities are available on a particular device.	`FAWINDOW`
`enhancements`	This bit-mask defines the default enhancements that are supported on the window or display device.	See *fa.h*
`defaulten-hancements`[2]	Is initially set to a value which optimizes the performance of the window system (`FAOFF`).	See *fa.h*.
`cursor`	Is `TRUE` if the cursor can be physically removed from the window device and is `FALSE` otherwise. `TRUE` in Windows/9000.	`TRUE` or `FALSE`
`fontcellheight` `fontcellwidth`	Indicate the pixel height and width of the active font.	
`clearbefore-write`[2]	Determines whether the background is automatically cleared before writing characters. The default value is `TRUE`, which causes the background to clear before writing.	`TRUE` or `FALSE`
`foregroundplanes` and `backgroundplanes`	Specifies the number of memory planes available for controlling the foreground and background colors, respectively.	0, 1, 4, or 8
`colormode`[2]	Indicates which color option is currently in use. Default is `FAWONB`.	`FAWONB`, `FACOLOR`, `FABONW`
`makecurrent`[2]	This bit-mask controls the updating of fast alpha operations to the screen. Various bits in the mask control *when* information is displayed via fast alpha routines.	See *fa.h*.

1 All fields of the *fainfo* structure are 32-bit integers; this provides compatibility with other languages.
2 Only these fields can be changed via *fasetinfo*.

Example

The following code segment sets `clearbeforewrite` to `TRUE` and sets `makecurrent` so that the screen won't be updated for any *fawrite* operations—i.e., writes will be queued. Later on, `makecurrent` is reset so that all queued *fawrite* operations will be performed.

```
#include <fa.h> /* fast alpha constant/structure definitions */
    :
        struct fainfo fa_env;       /* fast alpha environment structure */
        int gfd;                    /* graphics window file descriptor  */
    :
        gfd = gopen(device_path, OUTDEV, device_name, INIT);

/*
 * First, get the current environment:
 */
        if (fagetinfo(gfd, &fa_env) < 0) {
                perror("fagetinfo gfd");
                exit(1);
        }

/*
 * Next set the appropriate values in the structure and call fasetinfo:
 */
        fa_env.clearbeforewrite = TRUE;
        fa_env.makecurrent = (NOMCONFAWRITE | NOMCONFARECTWRITE);
        if (fasetinfo(gfd, &fa_env) < 0) {
                perror("fasetinfo gfd NOMCONFAWRITE(S)");
                exit(1);
        }

    :
/*
 * Now any fawrite or clear operations that occur will not be
 *      updated on the display until the makecurrent field is
 *      reset to MCALWAYS.  (This is done next.)
 */
```

```
        :
    /*
     * Now set makecurrent so that queued writes will be displayed:
     */
            fa_env.makecurrent = MCALWAYS;
            if (fasetinfo(fgd, &fa_env) < 0) {
                    perror("fasetinfo gfd MCALWAYS");
                    exit(1);
            }
        :
```

Cursor Control

With fast alpha routines, you can display and move a cursor in graphics windows and bit-mapped displays. The *facursor(3W)* routine performs cursor control operations.

Procedure

To move and/or turn the cursor on or off, call *facursor*; its syntax is:

`facursor(`*gfd, column, line, cflag*`)`

The *column* and *line* parameters specify the column and line at which to position the cursor; the top line of the window or display is line **0**, and the leftmost column is column **0**.

The *cflag* parameter determines whether or not the cursor is displayed. If *cflag* is **TRUE**, the cursor is displayed; if **FALSE**, the cursor is turned off. Note that turning the cursor on or off doesn't in itself change the cursor position as fast alpha remembers it.

If you specify invalid coordinates for *column* and *line*—specifically, **FACURSORNOMOVE** as defined in *fa.h*—then the cursor won't move, but **cflag** is still effective. This is useful if you simply wish to turn on/off the cursor at its current position.

Note that if part of the window is off screen, then the desired cursor position may also be off screen. For example, if the upper-left corner of the window is off screen, then *0,0* are valid cursor coordinates, but the cursor will not be visible; it will be off screen.

Also, you can specify a cursor position that might be occluded by windows higher up in the display stack.

Precautions

The cursor is actually a displayable character and is taken from the currently active font. Therefore if no font is activated, the cursor is automatically turned off. Attempting to turn the cursor on when no font is active will result in an error.

Examples

The following code segment displays the cursor at column 27 and line 12:

```
#include <fa.h> /* fast alpha constant and structure definitions */
     :
facursor(gfd, 27, 12, TRUE);
```

The next example turns the cursor off at its current position:

```
#include <fa.h> /* fast alpha constant and structure defintions */
     :
facursor(gfd, FACURSORNOMOVE, FACURSORNOMOVE, FALSE);
```

Note that if either *column* or *line* is invalid (or equals `FACURSORNOMOVE`), then the cursor's position will not be updated.

Writing Characters

Fast alpha routines provide two kinds of writing operations: you can write a string of characters, or you can fill a rectangle with a specific character.

Procedure

Following are separate discussions for writing strings and filling rectangles.

Writing Character Strings

To write character strings, use the *fawrite(3W)* routine; its syntax is:

```
fawrite(gfd, column, line, charbuf, ebuf, nchars)
```

The *column* and *line* parameters specify the character location where the string should start in the window (or bit-mapped display).

The *charbuf* parameter is a pointer to the buffer of characters that is to be written; this buffer contains *nchars* characters. That is, *fawrite* will write *nchars* characters, taking characters from the address specified by *charbuf*.

The *ebuf* parameter is a pointer to a buffer of enhancements that are to be applied to each character in *charbuf*; *ebuf* can be either `NULL` (no characters at all) or can contain *nchars* characters:

- If *ebuf* is `NULL`, then the enhancements specified in **`defaultenhancements`** are made to each character in the output string.
- Otherwise, each character in *ebuf* defines the enhancement(s) (such as inverse or underlining) to use when displaying the corresponding character in *charbuf*. For example, the fifth character in *ebuf* defines the enhancement(s) to use when displaying the fifth character in *charbuf*.

  Valid enhancements are defined in *fa.h*. To combine enhancements, you should OR the different enhancements.

Filling a Rectangle

The *farectwrite(3W)* routine fills a rectangular area in the window (or screen) that is specified by a rectangle structure. (See the "Concepts" section for details on rectangles.) This routine has the following syntax:

```
farectwrite(gfd, character, enhancement, rp)
```

The *rp* parameter is a pointer to a rectangle structure that defines the area to be filled. The area is filled with the character specified by the *character* parameter, and the *enhancement* parameter describes which enhancement(s) to use when displaying the character. (If *enhancement* is `NULL`, then **`defaultenhancements`** are used when displaying *character.*)

Example

The following code segment fills a screen rectangle with inverse video **X**'s; the rectangle's upper-left corner is at the origin (column 0, line 0). It then writes the message:

```
 What an exciting
rectangle this is!
```

and underlines the word "exciting."

```
#include <fa.h>
   :
struct fa_rectangle rp;
   :

/*
 * First, write the inverse-X rectangle to the screen;
 *      it must be large enough to surround the message
 *      that will be written inside of it.
 */
     rp.origin.x = 0;
     rp.origin.y = 0;
     rp.corner.x = 21;
     rp.corner.y = 4;
     farectwrite(gfd, "X", FAINVERSE, &rp);
```

```
/*
 * Now  write the first line into the rectangle:
 *    Note that in the enhancements string;
 *          @ --> FAOFF (no enhancements)
 *          D --> FAUNDERLINE (underline the text)
 */
    fawrite(gfd, 2, 1, "What an exciting", "@@@@@@@@DDDDDDDD", 16);
/*
 * Now write the second line with no enhancements:
 */
    fawrite(gfd, 1, 2, "rectangle this is!", NULL, 18);
   :
```

Font Manipulation

The fast alpha library contains font manipulation routines which you can use to display different fonts in graphics windows or bit-mapped displays.

Concepts

At fast alpha initialization time, a default font is established. If a font has already been established via font manager routines (discussed in the next chapter), then that font is used. If there is no active font at initialization, then a system default font is activated (the font specified by the WMBASEFONT environment variable). The current font can be changed via fast alpha routines or font manager routines; **however**, it is recommended that once you've started using the fast alpha environment, you should make font changes using only the fast alpha routines; this ensures that the fast alpha environment is always aware of the current font attributes (such as height, width, and colors).

Unlike the term0 font management model, there is no notion of base and alternate fonts; there is only the **active** font. Any text written is always displayed in the active font.

Fonts are **loaded** into the fast alpha **font cache** from the font directories described in the term0 font management model; loading a font causes it to be the active font. Note that the fast alpha font cache is not the same one used by term0 font management routines. However, the fast alpha font cache is the same as the font manager's. (In fact, to perform font management, the fast alpha routines call font manager routines.)

When you are through using a font, you can **remove** it from the font cache.

Note that fast alpha fonts are often denoted by **font ids**; these are **not the same as term0 font ids** but are the same as font manager font ids. Attempting to intermix term0 and fast alpha font ids may result in unpredictable system behavior.

Procedure

Following are brief discussions of how to perform the various fast alpha font management tasks:

Loading a Font

To load a font into your font cache, call the *fafontload(3W)* routine; its syntax is:

```
fafontload(gfd, path)
```

The *path* parameter is the path name of the font file to load. When the font is loaded, it is automatically activated, and *fafontload* returns a unique[1], system-wide font id that identifies the font. This font id is required as a parameter to some other font routines.

Activating a Font

To activate a loaded font that isn't currently active, call *fafontactivate(3W)*; its syntax is:

```
fafontactivate(gfd, fontid)
```

This routine activates as the current font the font specified by *fontid*—the system-wide font id returned when the font was loaded into the cache. After calling this routine, any text written subsequently will be displayed in the new font.

Removing a Font

When you are finished using a font, you can remove it from the font cache. The *fafontremove(3W)* routine removes a font from the cache; its syntax is:

```
fafontremove(gfd, fontid)
```

After calling this routine, the font specified by *fontid* will no longer exist in the font cache; to use this font again, it must be reloaded and reactivated.

[1] Unique only if *gfd* is the file descriptor for an open window device interface.

Setting Font Colors

The foreground and background colors to be used when displaying fonts can be set via the *facolors(3W)* routine; its syntax is:

```
facolors(gfd, foreground, background)
```

After calling this routine, the active font's foreground and background colors will be set to those specified by the *foreground* and *background* parameters. These colors are indices into the system color map.

Supported values are determined by the display device: 0 or 1 for monochromatic displays, 0 to 15 for 4-plane color, and 0 to 255 for 8-plane color.

Note that if the `colormode` field of the *fainfo* structure is not set to `FACOLOR`, the system ignores any color changes—it assumes everything is black and white.

Note that calling this routine also causes the fast alpha environment to take note of the current font attributes (i.e., width, height, color, etc.).

Precautions

The cursor is actually a displayable character and is taken from the currently active font. Therefore if no font is activated, the cursor is automatically turned off. Attempting to turn the cursor on when no font is active will result in an error.

Example

The following code segment loads an 8-by-16-pixel bold font into the font cache, activates the font, changes its colors to black on white, writes the word "HELLO," and removes the font from the cache.

```
#include <fa.h> /* fast alpha definitions */
#define BLACK 0
#define WHITE 1
   ⋮
int      gfd;   /* graphics window file descriptor */
int      fid;   /* font id for the bold font       */
   ⋮
```

```
/*
 * Load the bold font into the font cache:
 */
   if ((fid = fafontload(gfd, "/usr/lib/raster/8x16/lp.b.8U")) < 0)
   {
           perror("fafontload gfd");
           exit(1);
   }

/*
 * Activate the font:
 */
   if (fafontactivate(gfd, fid) < 0)
   {
           perror("fafontactivate gfd");
           exit(1);
   }

/*
 * Change the foreground and background to black on white:
 */
   if (facolors(gfd, BLACK, WHITE) < 0)
   {
           perror("facolors gfd");
           exit(1);
   }

/*
 * Now write the "HELLO" message:
 */
   fawrite(gfd, 0,0, "HELLO", NULL, 5);
/*
 * Finally, remove the bold font from the cache:
 */
   if (fafontremove(gfd, fid) < 0)
   {
           perror("fafontremove gfd");
           exit(1);
   }

  :
```

Clearing a Rectangle

Any rectangular area of characters in a graphics window or bit-mapped display can be cleared (erased). For example, you could clear an entire window. The *faclear(3W)* routine is used for this purpose.

Procedure

To clear a rectangle, simply call the *faclear(3W)* routine; its syntax is:

`faclear(`*gfd, enhancements, rp*`)`

The *rp* parameter is a pointer to a rectangle structure that defines the rectangle to clear.

The *enhancements* parameter is currently ignored by the system and is reserved for future expansion. For now, just leave this parameter set to `FAOFF`.

Example

The following subroutine clears a rectangular portion of a window, given the window's file descriptor. You must specify the rectangle's location and size. Note that the alpha environment must be initialized before calling this routine.

This function is named *clear_gr.c* and is found in the *man_examples* directory.

```
#include <fa.h>          /* fast alpha definitions                     */
clear_gr(gfd, row, col, x_chars, y_chars)
int gfd;                 /* gopened file descriptor                    */
int row, col;            /* starting row and column to be cleared      */
int x_chars, y_chars;    /* number of characters to be cleared         */
{
    struct fa_rectangle rect;  /* rectangle to be cleared              */
/*
 * CLEAR A PORTION OF A GRAPHICS WINDOW, APPLYING THE SPECIFIED ENHANCEMENT
 *
 * Determine the size of the area to be cleared:
 */
    rect.origin.x = row;
    rect.origin.y = col;
    rect.corner.x = row + x_chars;
    rect.corner.y = col + y_chars;


/*
 * Call fast alpha to clear the area:
 */
    if (faclear(gfd, FAOFF, &rect) < 0) return(-1);
    return(0);
}
```

Scrolling a Rectangle

Any screen area defined by a rectangle structure can be scrolled. The *faroll(3W)* routine performs this task.

Procedure

To scroll a portion of a window (or screen), simply call *faroll(3W)*; its syntax is:

```
faroll(gfd, how, howfar, rp)
```

The *rp* parameter points to a rectangle structure that defines the portion of the window to scroll. The *how* parameter defines the direction to scroll, and the *howfar* parameter defines how many character units to scroll in the direction indicated by *how*.

The following are valid values for *how*:

- `FAROLLUP` ('u') — says to roll the rectangle's contents up.
- `FAROLLDOWN` ('d') — roll the rectangle's contents down.
- `FAROLLLEFT` ('l') — roll the rectangle's contents left.
- `FAROLLRIGHT` ('r') — roll the rectangle's contents to the right.

Note that the area uncovered by scrolling is cleared.

Example

The following code segment rolls a graphics window's contents in all four directions: up, right, down, left.

```
      :
struct fa_rectangle rp;
      :
/*
 * Roll the contents of the window specified by gfd up 5 lines:
 */
    if (faroll(gfd, FAROLLUP, 5, &rp) < 0) {
        perror("faroll up");
        exit(1);
    }
```

```
/*
 * Roll the contents of the window specified by gfd right 11 characters:
 */
    if (faroll(gfd, FAROLLRIGHT, 11, &rp) < 0) {
        perror("faroll right");
        exit(1);
    }

/*
 * Roll the contents of the window specified by gfd down 1 line:
 */
    if (faroll(gfd, FAROLLDOWN, 1, &rp) < 0) {
        perror("faroll down");
        exit(1);
    }

/*
 * Roll the contents of the window specified by gfd left 7 characters:
 */
    if (faroll(gfd, FAROLLLEFT, 7, &rp) < 0) {
        perror("faroll left");
        exit(1);
    }
      .
      .
      .
```

Notes

The Font Manager Library 15

The font manager library provides a high-performance, low-level textual interface to graphics windows and bit-mapped displays. This library's functionality overlaps with the fast alpha library, and in fact, some fast alpha routines call font manager routines. However, the font manager does provide some powerful capabilities not provided by the fast alpha library. The following topics are discussed in this chapter:

- concepts essential to using font management routines
- font management
- font information routines
- writing characters
- character clipping.

Concepts

Font manager routines can be used with either graphics windows or bit-mapped displays supported by Windows/9000; the routines require the file descriptor returned from performing a graphics open (*gopen(3G)*) on the device interface for the window or bit-mapped display.

When used with the window system, the font manager is a distributed library that is controlled via a set of data structures kept in shared memory common to all users of the font manager. However, when using the font manager with a supported bit-mapped display, the concepts of "distributed" and "global" are lost; the font manager becomes isolated to the calling process and does not cooperate with other applications trying to use the font manager to the same device.

Font manager routines allow you to load, activate, and remove fonts, and change attributes that affect how a font is displayed. The font management model is identical to that used by fast alpha routines. In fact, fast alpha routines simply call font manager routines to perform font management tasks.

All fonts in the system, regardless of process association, have a unique **font id**; that is, font ids are global. Routines are provided to obtain the path name of the font represented by a font id.

Note that there are two main differences between the font manager and fast alpha libraries:

1. Font manager uses pixel units to specify character coordinates; fast alpha uses character column and line addressing.
2. Font manager can operate with proportionally spaced fonts; fast alpha cannot.

Note

Definitions from the */usr/include/fonticon.h* header file are used throughout this chapter.

Font Management

The font manager library contains font management routines which you can use to display different fonts in graphics windows or bit-mapped displays.

Concepts

Like the term0 and fast alpha routines, the font manager maintains a **font cache** (**font table**). The font table is an area of memory shared by all users of the font manager; it can hold information for up to 32 different fonts.

Font information is **loaded** into the font table from **font files**. Font files are stored in font directories—sub-directories located under the directory specified by the WMFONTDIR environment variable.

When a font is loaded via font manager routines, it automatically becomes the **active** font. Text is always displayed in the active font. Font manager routines can be used to **activate** any font that you have loaded.

In addition when a font is loaded, a unique, system-wide **font id** is returned. This font id is used to identify the font to certain font management routines. Font manager font ids are different than those used by term0 font management routines; these ids should not be mixed.

Loaded fonts can be different sizes; by using font manager routines, different-sized fonts can be mixed in the same graphics window (or bit-mapped display).

When you are finished using a font, you must always **remove** it from the font table. Loading and removing a font is analogous to opening and closing a file—after you open a file, you must eventually close it.

As mentioned previously, the font table is shared by all users of the font manager library—that is, all users of the computer system on which Windows/9000 is used. Therefore, several users may be using the same font in the font table. The font manager takes care not to duplicate fonts in the font table; when a user attempts to load an already-loaded font, the font manager simply takes note that another user is using the font; the font isn't reloaded into the table.

The same goes for removing fonts. If more than one user is using the same font, then the font isn't actually removed from the table; the font manager simply takes note that one less user is using the font. If only one user is using a font, then removing the font **will** cause it to be removed from the font table.

Note: Never assume that a font exists in the font table unless you've loaded it **and** haven't yet removed it. If you remove a font, there's no guarantee that it still exists in the table, because others using the font might remove it. Just remember: don't make any assumptions about the shared memory; load and remove fonts as if you're the only user.

The foreground and background colors of the active font default to white on black. On color systems, you can redefine the foreground and background colors to any from the system color map.

Procedure

Following are brief discussions of how to perform the various font manager tasks:

Loading a Font

To load a font into the font table, call *fm_load(3W)* which has the following syntax:

`fm_load(`*gfd, path, fontid*`)`

The *path* parameter points to the path name of the font file to load into the font table. The font's id is returned in the integer pointed to by the *fontid* parameter.

In addition to being loaded, the font automatically becomes the active font. So if you want a font other than the loaded font to be the active font, you must activate the other font.

Note: When you are finished using a loaded font, you must remove it.

Activating a Font

To activate a previously loaded font, call *fm_activate(3W)*; its syntax is:

`fm_activate(`*gfd, fontid*`)`

The font represented by the *fontid* parameter is made the active font. All text will be displayed in the new font until the next call to *fm_activate*.

Removing a Font

To remove a loaded font, call *fm_remove(3W)* which has the following syntax:

```
fm_remove(gfd, fontid)
```

After calling this routine, the font specified by *fontid* will be removed from the font table (as far as **your** application is concerned).

Note: If you've accidentally loaded a font twice, the font must be removed twice also.

Setting Colors

To change the active font's foreground and background colors, call *fm_colors(3W)*; its syntax is:

```
fm_colors(gfd, foreground, background)
```

The *foreground* and *background* parameters specify the new colors to use; they are indices into the system color map.

Supported values are determined by the display device: 0 or 1 for monochromatic displays, 0 to 15 for 4-plane color, and 0 to 255 for 8-plane color.

Note: The foreground and background colors return to the defaults (black and white) whenever a font is activated.

Example

The following code segment loads an 8-by-16-pixel bold font into the font cache, activates the font, changes its colors to black on white, writes the word "ERROR," and removes the font from the cache.

```
#define BLACK 0
#define WHITE 1
      :
int      gfd;   /* graphics window file descriptor */
int      fid;   /* font id for the bold font       */
      :
```

```
/*
 * Load the bold font into the font cache; the font is
 *       automatically activated:
 */
    if (fm_load(gfd, "/usr/lib/raster/8x16/lp.b.8U", &fid) < 0)
    {
             perror("fm_load gfd");
             exit(1);
     }

/*
 * Change the foreground and background to black on white:
 */
    if (fm_colors(gfd, BLACK, WHITE) < 0)
    {
            perror("fm_colors gfd");
            exit(1);
    }

/*
 * Now write the "ERROR" message (see the section
 *   "Writing Characters" for more details on writing):
 */
    fm_write(gfd, 10,10, "ERROR", 5, TRUE, TRUE);
/*
 * Finally, remove the bold font from the cache:
 */
    if (fm_remove(gfd, fid) < 0)
    {
            perror("fm_remove gfd");
            exit(1);
    }

      :
```

Font Information Routines

The font manager library provides routines that obtain information about fonts. In particular you can discover:

- font size information
- a font's path name
- information on a font's style.

Concepts

Before discussing how to obtain font information, a discussion of font sizes and font styles is needed.

Font Size

Font size is actually comprised of three different attributes: **width**, **height**, and **baseline height**.

Font width and height are straightforward. Each character in a font is displayed in a **font cell**. The font cell is the same size for all characters. The font's width and height represent the pixel width and height of the font cell.

All the characters of a given font "sit" on an invisible line called the **baseline**. The "bottom" of each character is flush with this line. Note however that parts of characters can extend below the baseline—for example, the "stem" that extends below the circle on the letter **p**. Figure 15-1 illustrates each of these size attributes.

The baseline attribute is important because it allows you to align different-sized fonts on the same line. For example, suppose you are writing a story that starts with "In the beginning," and you want the first letter **I** to be in a large font and the rest of the characters to be in a normal-sized font. To make the text look more natural, you should align the baseline of the big **I** with the baseline of the normal-sized font.

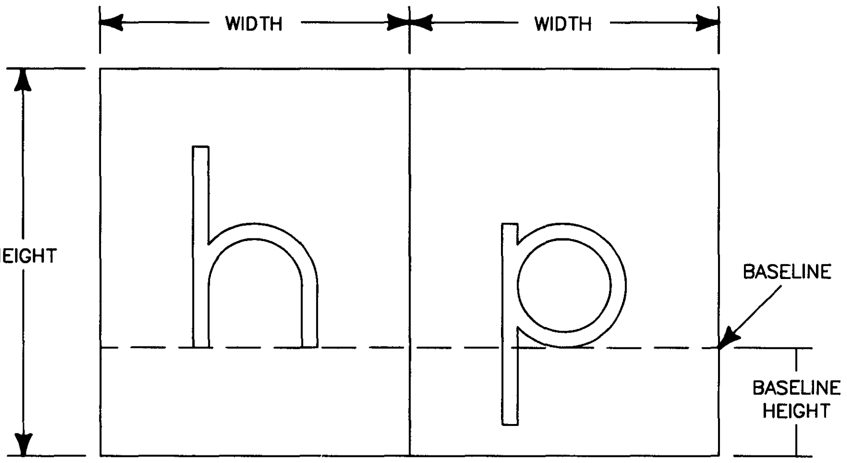


Figure 15-1. Font Size Attributes.

Font Style

Each font has certain attributes that define its style. These attributes are defined by the **escapecodes** structure in the *fonticon.h* header file; Table 15-1 briefly defines each of the fields in this structure.

Table 15-1. Quality attributes defined by escapecodes structure.

Item	Description	Range
`symbol_int`	Gives the numerical part of the font's identification string; e.g., 8 for 8-bit Roman-8 (8U); 0 for 7-bit math font (0M). The value for this field indicates whether the font is 8-bit (=8) or 7-bit (=0).	0, 7, or 8
`typeface`	Specifies the kind of typeface, e.g. *pica*=1, *prestige*=8, etc.	0 to 10
`proportional`	Tells whether the font is uniform width (=0) or proportional (=1).	0 or 1
`hpitch`	Approximates horizontal characters per inch.	Depends on font width.
`vheight`	Approximates vertical characters per inch.	Depends on font height.
`boldness`	Indicates the boldness of the font. -7 is the lightest, 7 is the boldest.	-7 to 7
`quality`	Describes the quality of the font: data processing (=0), near letter quality (=1), or correspondence quality (=2).	0 to 2

Procedure

Getting Font Size Information

Two font manager routines obtain font size information: *fm_fileinfo(3W)* and *fm_rasterinfo(3W)*. The *fm_rasterinfo* routine gets size information for fonts in the font table; its syntax is:

fm_rasterinfo(*gfd, fontid, width,height, baseline*)

This routine returns the font cell width and height (in pixels) and the baseline height (also in pixels) for the font specified by *fontid.*

The *fm_fileinfo* routine gets size information for a font file; its syntax is:

fm_fileinfo(*gfd, path, width,height, baseline*)

This routine returns font size information for the font file whose path name is pointed to by *path.*

Getting Font Style Information

To get font style information (as described in Table 15-1), call the *fm_styleinfo(3W)* routine; its syntax is:

fm_styleinfo(*gfd, fontid, symbol_char, escapecodes*)

The *symbol_char* parameter should point to a character which will contain a character describing the font (e.g., 'U' for Roman-8 fonts, 'K' for Katakana fonts).

The *escapecodes* parameter should point to an **escapecodes** structure as defined in *fonticon.h.* Upon returning, this structure will contain style information for the font indicated by *fontid.*

Getting a Font's Name

The *fm_getfontid(3W)* routine translates a font id into its corresponding font name; its syntax is:

fm_getname(*gfd, fontid, filename*)

The *filename* parameter points to a character string that will contain the path name that was used to load the font represented by *fontid.*

Example

The following function gets font size, style, and name information for the font specified by the `fid` parameter; it returns this information to the calling program.

```
#include <fonticon.h>     /* font manager definitions              */
get_font_info(gfd, fontid, width,height, baseline, symbol_char,
              escapecodes, filename)
int  gfd;                 /* window's file descriptor              */
int  fontid;              /* font id for the desired font          */
int  *width, *height;     /* character cell size                   */
int  *baseline;           /* baseline of the character cells       */
char *symbol_char;        /* character describing the font         */
struct escapecodes *escapecodes;  /* points to font-specific
                                     style information */
char *filename;           /* the path name of the font             */
```

```
{
/*
 * Get the size information for the specified font id:
 */
    if (fm_rasterinfo(gfd, fontid, width, height, baseline) < 0) return(-1);

/*
 * Get the style information for the specified font id:
 */
    if (fm_styleinfo(gfd, fontid, symbol_char, escapecodes) < 0) return(-1);


/*
 * Get the path name for the font specified by font id:
 */
    if (fm_getname(gfd, fontid, filename) < 0) return(-1);

    return(0);
}
```

Writing Characters

With font manager routines, you can write text in the active font to any graphics window or the bit-mapped display. By default, characters are written from left to right; however, characters can be written in any direction—up, down, to the right, or to the left. In addition, you can optimize the generation of characters on your particular display hardware.

Procedure

Controlling the Write Direction

By default characters are written to the right. By using the *fm_fontdir(3W)* routine, you can write characters any horizontal or vertical direction. The syntax of this routine is:

```
fm_fontdir(gfd, direction)
```

After calling this routine, any characters written to the window via the *fm_write* routine will be written in the direction specified by the *direction* parameter. Valid values for *direction* are:

- 'u' — write upward
- 'd' — write downward
- 'l' — write to the left
- 'r' — write to the right (this is the default).

Note: This routine affects **only** the direction of the write and **not** the characters themselves—they are still displayed normally within each character cell. Also, the write direction stays in effect until a different font is activated; at that point, the direction returns to the default ('r').

Writing Characters

The *fm_write(3W)* routine displays character strings to a graphics window or bit-mapped display; its syntax is:

```
fm_write(gfd, x,y, str, numchars, dump, colormode)
```

The string to write is pointed to by the *str* parameter; the number of characters in the string is given by *numchars.*

The string is written at the pixel coordinates specified by *x,y*. Note that in graphics windows, the upper-leftmost pixel in the contents area has coordinates *0,0*; in bit-mapped displays, the upper-leftmost pixel is *0,0*.

How the characters are positioned with respect to the *x,y* coordinates depends on the current write direction. Figure 15-2 illustrates character positioning.

The *dump* parameter indicates whether or not to immediately update the display after the write: `TRUE` means to update; `FALSE` means to let the system-imposed buffering take care of the visual update.

The *colormode* parameter determines whether or not to use colors from the previous call to *fm_colors*. If `TRUE`, then the area where the characters will be written is cleared to the current background color, and the characters are written in the foreground color. Setting this parameter to `TRUE` is analogous to setting `clearbeforewrite` to `TRUE` in the fast alpha environment. This mode has the side effect of leaving the colormode and write-enable masks set as needed.

If *colormode* is `FALSE`, then the characters are displayed using the current Starbase graphics replacement rule and write mask. For example, if the current replacement rule is to OR the image onto the background, then the characters will simply be placed over the background image without erasing it. Setting colormode to `FALSE` is somewhat analogous to setting `clearbeforewrite` to `FALSE` in the fast alpha environment.

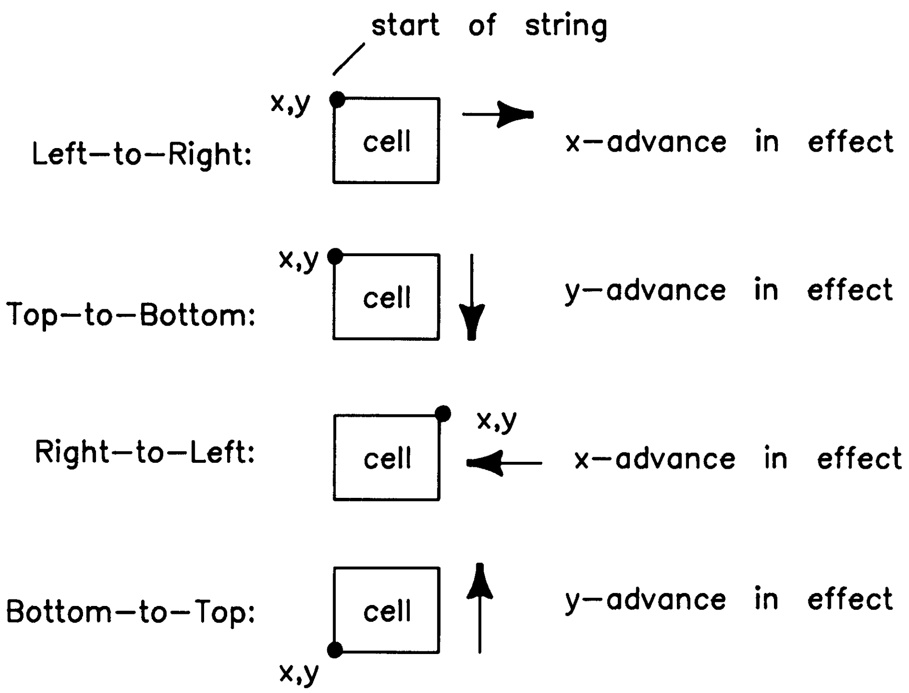


Figure 15-2. Character Positioning at x,y Coordinates.

Optimizing Character Generation

Some systems have specialized hardware for writing to bit-mapped displays. This special hardware allows characters to be written faster to the display. The *fm_opt(3w)* routine allows you to take advantage of specialized display hardware; its syntax is:

```
fm_opt(gfd, optmode)
```

If *optmode* is 1, then optimization is turned on; if *optmode* is 0, then optmization is turned off.

Note: This routine will fail if optimization hardware doesn't exist on the system or if too many fonts have been optimized already. This should *not* be considered a fatal error. Therefore, you should *not* abnormally terminate your program (via *exit(2)* or whatever) if this routine fails.

Determining String Length

Often times you may wish to determine the length of a character string. For example, if you want to determine if a character string is so long that it will "run off" the edge of a window or display. The *fm_str_len(3W)* routine determines the pixel length of any character string along the current direction of the active font; its syntax is:

```
fm_str_len(gfd, str, numchars)
```

The *str* parameter points to the character string which contains *numchars* characters.

Note: This routine is especially useful if character clipping is disabled (see the "Character Clipping" section for details).

Example

The following function, *write_dn.c*, displays a null-terminated character string downward from the pixel location specified by the `xpos,ypos` parameters. It then sets the write direction back to the right.

The source for this routine is found in the *man_examples* directory.

```
#include <fonticon.h>    /* font manager definitions               */
write_dn(gfd, str, str_len, xpos, ypos)
int  gfd;                /* window's file descriptor               */
char *str;               /* the string to be written               */
int  str_len;            /* length of the string to be written     */
int xpos, ypos;          /* starting position for the string       */
```

```
{
/*
 * Set the write direction to down:
 */
    if (fm_fontdir(gfd, 'd') < 0) return(-1);

/*
 * Write the string at the specified position:
 *      update the screen immediately
 *      don't use the colors from fm_colors
 */
    if (fm_write(gfd, xpos, ypos, str, str_len, 1, 0) < 0) return(-1);

/*
 * Set the write direction to right:
 */
    if (fm_fontdir(gfd, 'r') < 0) return(-1);

    return(0);
}
```

Character Clipping

Character clipping controls the area in which characters are written in a graphics window. When character clipping is enabled in a window, the window system will **not** allow characters to be written outside established **clip limits**. You can enable or disable clipping and redefine clip limits via font manager routines.

By default when a graphics window is created, clipping is enabled and the clip limits are always set to the current window size.

Note

Memory can become corrupted if clipping is not enabled. This is because characters could inadvertently be written outside the display memory established by the clip limits. Conceivably, you could write spurious data into your data structures and your program.

If you do not use clipping, be sure to check the length of every character string (via *fm_str_len(3W)*) to ensure that displaying the string will not cause it to extend outside the display screen boundaries.

IMPORTANT

If you are using font manager routines in color mode (`colormode = FACOLOR`), then be careful when writing characters with clipping enabled: **clipping may not work when color mode is in effect.** Therefore, **you** must ensure that writing characters does not cause them to appear outside a window's boundary whenever color mode is in effect.

Procedure

Enabling/Disabling Clipping

To enable or disable clipping for a given window or bit-mapped display, use the *fm_clipflag(3W)* routine; its syntax is:

```
fm_clipflag(gfd, flag)
```

The *flag* parameter indicates whether to enable or disable clipping: if *flag*=**1**, then clipping is enabled; if *flag*=**0**, then clipping is disabled.

Setting Clip Limits

To set clip limits for a graphics window or bit-mapped display, use the *fm_cliplim(3W)* routine; its syntax is:

```
fm_cliplim(gfd, x,y, width,height)
```

The *x,y* parameters indicate the *x,y* location of the upper-left corner of the clipping rectangle (with respect to the upper-leftmost pixel of the window or display); the *width,height* parameters define the pixel width and height of the clipping rectangle. After calling this routine, characters can be written only within the defined rectangle.

Example

The following function, *quarter_clip.c*, redefines the clipping area to be one-fourth the size of the current window, and it centers the clipping area in the window. (The *wgetcoords(3W)* routine is used to get information about the window's size.)

The source for this function is found in the *man_examples* directory.

```
#include <fonticon.h>     /* font manager definitions                    */
quarter_clip(gfd)
int  gfd;                 /* window's file descriptor                    */
{
    int x,y, w,h, dx,dy, rw,rh; /* dimensions of the window */
/*
 * Get the size of the user portion of the window:
 */
    if (wgetcoords(gfd, &x,&y, &w,&h, &dx,&dy, &rw,&rh) < 0) return(-1);
/*
 * Compute the bounds of the clipping area:
 *      The clipping area is one-fourth the size of the window,
 *      and is centered in the window.
 */
    w /= 2;
    h /= 2;
    x = w / 2;
    y = h / 2;


/*
 * Set the new font clip area:
 */
    if (fm_cliplim(gfd, x, y, w, h) < 0) return(-1);

/*
 * Turn on font clipping:
 */
    if (fm_clipflag(gfd, 1) < 0) return(-1);

    return(0);
}
```

Notes

Compiling Window Programs A

This appendix discusses:

- the various libraries supported by Windows/9000
- how to compile programs that call window library routines.

Linking Window Libraries

When compiling a program that uses window library routines, link the libraries in the order shown in the list below. Remember that you need link only the libraries that the program uses.

1. `libfa.a`—if the program calls any fast alpha routines, link this library first.
2. `libfontm.a`—if the program calls any fast alpha or font management routines, then link this library.
3. `libdd`*`driver`*`.a`—always link the device *driver* (or drivers) of the CRT(s) on which the program will run.
4. `libddbyte.a`—if the program performs graphics to windows with retained **byte-per-pixel** rasters, **and** you want the raster to be maintained in memory, then load this driver, which writes to the retained memory.

   `libddbit.a`—if the program performs graphics to windows with retained **bit-per-pixel** rasters, and you want the raster maintained in memory, then load this driver which writes to the retained memory.

Note

The 3001 Device Driver does not support `bit-per-pixel` retained rasters.

5. `libwindow.a`—this library should be linked if the program calls any window library routines or performs Starbase graphics, fast alpha, font manager output to windows.
6. `libsb1.a`—link this library if the program calls any Starbase graphics, fast alpha, or font manager routines.
7. `libsb2.a`—link this library immediately after `libsb1.a` if `libsb1.a` was loaded.

Examples

The following examples should help clarify how the libraries are linked with the main program.

A C program named *winprog.c* that calls only window manipulation routines and only creates term0 windows on a S300 low-resolution display would be compiled as:

```
cc winprog.c -lwindow
```

A C program named *grprog.c* that creates, manipulates, and performs graphics to non-retained graphics windows (on an HP9837 display) would be compiled as:

```
cc grprog.c -ldd9837 -lwindow -lsb1 -lsb2
```

A C program named *grt0.c* that creates and manipulates both term0 and graphics windows, calls fast alpha and font management routines, and does graphics to retained windows (on an HP98700 display) would be compiled as:

```
cc grt0.c -lfa -lfontm -ldd98700 -lddbyte -lwindow -lsb1 -lsb2
```

OR

```
cc grt0.c -lfa -lfontm -ldd300h -lddbit -lwindow -lsb1 -lsb2
```

Windows/9000 Files B

This appendix describes files associated with HP Windows/9000.

Device Files

/dev/crt, */dev/ocrt*	bit-mapped display special files
/dev/hilkbd	ITF keyboard
/dev/locator	locator (mouse or tablet)
/dev/rhil	raw input controller
/dev/screen	window directory—holds window type and unit device files and the window manager's device interface (*wm*)

Manual Examples and Demos

/usr/lib/hpwindows/man_examples	source for all code examples found in this manual
/usr/lib/hpwindows/demo	all demo programs and the *README* file
/usr/lib/hpwindows/demosrc	all demo source

Font and Icon Files

*/usr/lib/raster/**	contains all font directories from fast alpha and font manager filesets
/usr/lib/raster/icons	icon definition files are stored here
/usr/lib/raster/dflt/b/h/$LANG	If this file is present, it is the default (`/usr/lib/raster/dflt`) base font (`/b`) for high-resolution displays (`/h`) for the language defined by the LANG environment variable.
/usr/lib/raster/dflt/b/l/$LANG	If this file is present, it is the default (`/usr/lib/raster/dflt`) base font (`/b`) for low-resolution displays (`/l`) for the language defined by the LANG environment variable.
/usr/lib/raster/dflt/b/v/$LANG	If this file is present, it is the default (`/usr/lib/raster/dflt`) base font (`/b`) for very high resolution displays (`/v`) for the language defined by the LANG environment variable.
/usr/lib/raster/dflt/a/h/$LANG	If this file is present, it is the default (`/usr/lib/raster/dflt`) alternate font (`/a`) for high-resolution displays (`/h`) for the language defined by the LANG environment variable.
/usr/lib/raster/dflt/a/l/$LANG	If this file is present, it is the default (`/usr/lib/raster/dflt`) alternate font (`/a`) for low-resolution displays (`/l`) for the language defined by the LANG environment variable.
/usr/lib/raster/dflt/a/v/$LANG	If this file is present, it is the default (`/usr/lib/raster/dflt`) alternate font (`/a`) for very high resolution displays (`/v`) for the language defined by the LANG environment variable.

Header Files

/usr/include/window.h	window library constants and structure definitions

Windows/9000-Specific Files

*/usr/bin/w**	window system commands
/usr/lib/t0server	a term0 window server
/usr/lib/gserver	a dummy graphics window server process for object compatibility with pre-HP-UX 5.2 releases of HP Windows/9000
/usr/lib/stserver	server for the see_thru window type (for special hardware only)
/usr/lib/wm	the window manager process
/usr/lib/libwindow.a	window management, manipulation, icon, input, term0 font management, menu, and softkey library routines

Other Files

/usr/include/fa.h	fast alpha constant and structure definitions
/usr/include/fonticon.h	font constant and structure definitions
/usr/include/starbase..h*	Starbase constant and structure definitions
/usr/lib/libfa.a	fast alpha library
/usr/lib/libfontm.a	font manager library
/usr/lib/libsb1.a	Starbase graphics routines
/usr/lib/libsb2.a	stubs to Starbase routines—i.e., routines that make direct calls to the kernel
/usr/lib/libdd.a*	Starbase drivers
/usr/lib/libddbyte.a	driver that allows Starbase routines to write retained **byte-per-pixel** raster memory
/usr/lib/libddbit.a	driver that allows Starbase routines to write retained **bit-per-pixel** raster memory

Subject Index

Win an HP Calculator!

Your comments and suggestions help us determine how well we meet your needs. **Returning this card with your name and address enters you into a quarterly drawing for an HP calculator***.

HP Windows/9000 Documentation

	Agree				**Disagree**
The manual is well organized.	○	○	○	○	○
It is easy to find information in the manual.	○	○	○	○	○
The manual explains features well.	○	○	○	○	○
The manual contains enough examples.	○	○	○	○	○
The examples are appropriate for my needs.	○	○	○	○	○
The manual covers enough topics.	○	○	○	○	○
Overall, the manual meets my expectations.	○	○	○	○	○

You have used this product:

__ Less than 1 week	__ Less than 1 year	__ More than 2 years
__ Less than 1 month	__ 1 to 2 years	

fold ——

Please write additional comments, particularly if you disagree with a statement above. Use additional pages if you wish. The more specific your comments, the more useful they are to us.

Comments: ______________________________

*Offer expires June 1990. (97069-90002 E0488)

Please Tape Here

Please print or type your name and address.

Name:	
Company:	
Address:	
City, State, Zip:	
Telephone:	

Additional Comments:

HP Windows/9000 Documentation
HP Part Number 97069-90002
E0488

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Attn: Learning Products Center
3404 East Harmony Road
Fort Collins, Colorado 80525-9988

HEWLETT
PACKARD

HP Part Number
97069-90002

Microfiche No. 97069-99002
Printed in U.S.A. E0488

97069-90608
For Internal Use Only