HP Windows/9000 User's Manual

HP 9000 Series 300 Computers

HP Part Number 97069-90002

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

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

WARRANTY

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Copyright © Hewlett-Packard Company 1988

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written premission is prohibited, except as allowed under the copyright laws.

Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government Department of Defense is subject to restrictions as set forth in paragraph (b)(3)(ii) of the Rights in Tec.inical Data and Software clause in FAR 52.227-7013.

Use of this manual and flexible disc(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs can be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Copyright © AT&T, Inc. 1980, 1984

Copyright © The Regents of the University of California 1979, 1980, 1983

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

April 1988...Edition 1

Table of Contents

Getting Started 1

Welcome to the *HP Windows/9000 User's Manual.* This manual is intended for any new user of the window system, but can also be used as a reference by experienced users. This manual:

- explains window system concepts
- shows how to use the system interactively via the keyboard, mouse, and graphics tablet
- describes window system commands
- shows different methods for starting the window system
- shows how to customize HP Windows/9000 via environment variables
- defines window system terminology

Other Windows Documentation

In addition to this manual, other HP Windows/9000 documentation exists:

- *HP Windows/9000 Programmer's Manual*—describes how to use window system library routines from your C programs. If you are developing applications that make use of windows, you should use this manual.
- *HP Windows/9000 Reference*—contains HP-UX reference pages for window system commands (section 1) and library routines (section 3W).
- *Term0 Reference Manual*—describes various escape sequences that can be used with term0 windows (i.e., textual windows, described in more detail later). You would use this manual mainly if you are developing applications that run in term0 windows.

Conventions

The following conventions are used throughout this manual:

- *Italics* indicate the names of files and HP-UX commands, system calls, subroutines, etc. found in the *HP-UX Reference* (e.g., *wsh(1)*).
- **Boldface** is used when a word is first defined (as **term0**) and for general emphasis (**never do this**).
- `Computer font` indicates a literal, either typed by the user or displayed by the system. Keys are shown capitalized and enclosed in a rounded envelope. For example:

  `wmstart` Return

- Environment variables, such as WMDIR and WMIATIMEOUT, are represented in upper-case letters.

User's Manual Contents

Chapter 1: Getting Started

This chapter describes the scope, goals, conventions, organization, and content of the *User's Manual.* Other HP Windows/9000 manuals are described also.

Chapter 2: Concepts

This chapter explains window system concepts used throughout this manual. You should be sure to read this chapter before using the system.

Chapter 3: Interactive Use

This chapter shows how to start and stop the window system. However, the main purpose of this chapter is to show you how to interactively use the window system—i.e., how to use the keyboard, mouse, and/or graphics tablet to manipulate windows.

Chapter 4: Using Commands

This task-oriented chapter illustrates the use of window system commands. You can use commands to start and stop the system, and you can use commands to manipulate windows or display information about windows.

Chapter 5: Environment Variables

The window system has a number of environment variables that control the way the system performs. These variables are set to default values—values most users should find sufficient. However, some users may wish to alter system characteristics. This chapter describes how to customize your system by altering the value of window system environment variables.

Appendix A: Window Limitations

This appendix describes resrouce usage limitations inherent with windows, and discusses how to get around these obstacles where possible.

Appendix B: Glossary

Terms found throughout HP Windows/9000 documentation are defined in this glossary.

Notes

Concepts 2

This chapter discusses concepts essential to understanding the window system and how to use it. Specifically the following topics are presented:

- the desk top analogy
- rationale for using windows
- window types
- window format
- selected window
- the pointer (echo)
- the keyboard, mouse, and graphics tablet
- pop-up menus
- icons

HP Windows/9000 architecture and data flow is not covered in great detail in this chapter. If you require more detail than is presented here, read the "Concepts" chapter of the *HP Windows/9000 Programmer's Manual*, which contains much more detailed information on the intrinsic structure of the window system.

The Desk Top Analogy

Most of us are familiar with the picture of a desk scattered with papers. Some papers may be memos, others reference materials, and others current projects. As you place these papers on the desk, the most current are placed on top of others. During the course of the work day, papers are shuffled, bringing some on the bottom of the stack to the top. You may even work on several papers at the same time, thus performing several tasks simultaneously.

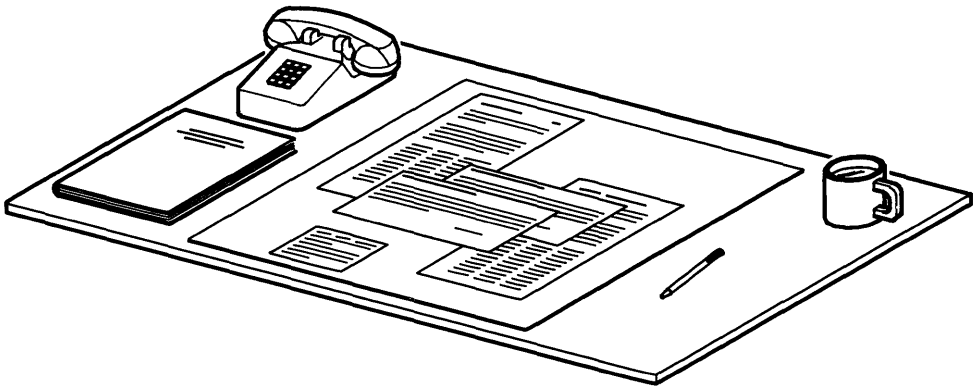

Figure 2-1. A Typical Desk Top

Now lets take this scenario and apply it to computers: In the past, before window systems existed, you basically performed one task at a time per computer terminal. Performing several tasks at once—i.e., running several programs at once—at a single terminal was often inconvenient or infeasible.

With window systems, however, this problem is eliminated. HP Windows/9000 allows you to have several **windows** on a single display screen. You can execute a different application in each window, and all applications can execute simultaneously.

Figure 2-2 shows a typical display screen with windows. Note how windows can be organized like papers on a desk top. Each window can be thought of as a terminal in which you perform a task. You can execute commands and run programs in each window, and you interact with each program within the border of the window. It's like having several terminals on the same screen.

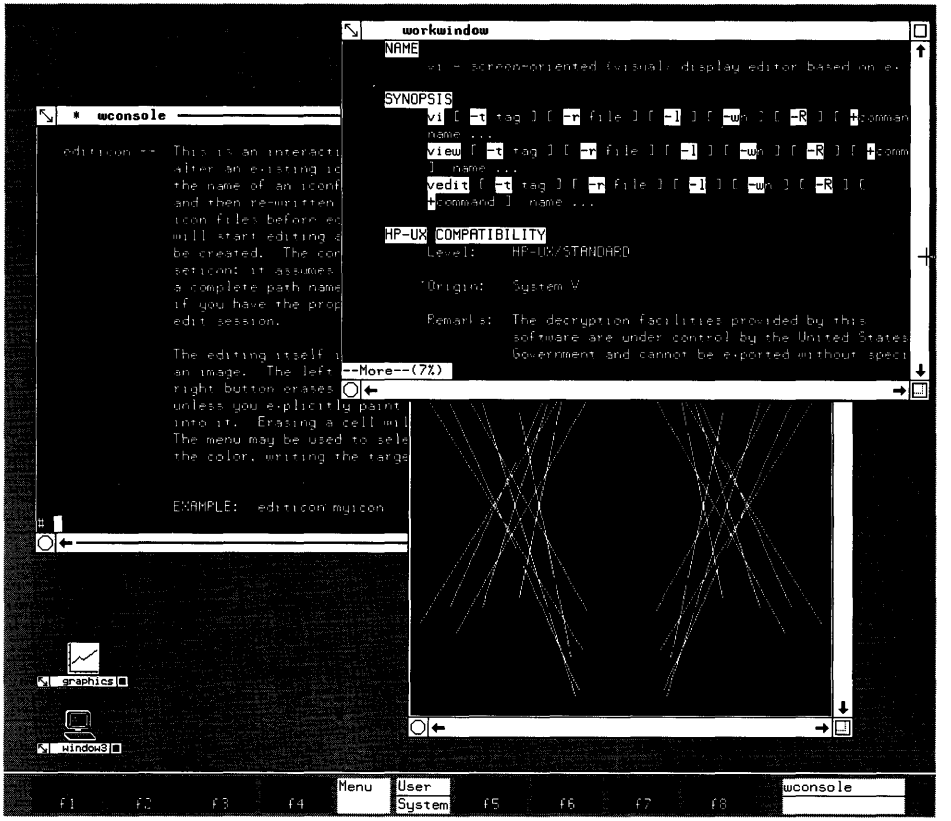

Figure 2-2. A Typical Window System Screen

Why Use Windows?

As mentioned above, Windows/9000 provides a break from the traditional user-computer interface. Traditionally, you would interact with a single terminal, entering commands to perform one task at a time at that terminal. Using windows, you can visually separate tasks on the display screen. The confusion of running multiple tasks from one terminal is reduced. In addition, you can observe the interaction between various programs on the same screen.

Another advantage of Windows is the ability to organize your tasks in the same manner you have done for years with paper. Like papers on your desk top, windows can be moved and shuffled. In fact, almost any task that is practicable with papers can also be performed with windows. For example:

- papers can be moved on the desk top; windows can be moved on the display screen
- you can throw away a paper; windows can be destroyed
- you can place papers in desk drawers; windows can be concealed
- you can set papers off in a corner until they're needed later; windows can be changed to icons (small, pictoral representations of windows, discussed later)
- papers can be folded (for example, to make them smaller); a window's size can be changed.

Window Types

There are basically two types of terminals: graphics displays and non-graphics (text) terminals. Graphics applications run on graphics displays; applications that do not require graphics run in text terminals.

Windows/9000 supports two window types: **terminal** and **graphics** windows. Some graphics hardware configurations additionally support the **see_thru** window type (for details on this window type, see the appendix "HP 98720 Graphics Display Station"). Graphics applications run in graphics windows, and non-graphics programs usually run in terminal windows.

The following discussion describes the terminal and graphics window types in more detail. If you are satisfied with the description above, then move on to the next section, "Window Structure"; otherwise you should read the next two sub-sections.

The Terminal Window Type

The terminal window type (also known as **term0**, pronounced "term-zero") emulates an HP 2622 terminal without block or format mode. In addition, term0 windows support HP 2627 color escape sequences.

Most non-graphics applications will run in term0 windows. For example, all HP-UX commands can be executed from term0 windows. For greater detail on term0 windows, see the *HP Windows/9000 Programmer's Manual* and the *Term0 Reference Manual.*

The Graphics Window Type

Graphics windows emulate the bit-mapped displays supported by HP Windows/9000. Starbase Graphics Library routines can be used to perform graphics in graphics windows. Applications that perform Starbase graphics can be run in graphics windows.

For more details on graphics windows, see the *HP Windows/9000 Programmer's Manual* and the *Starbase Device Drivers Library.*

Window Structure

Although there are two window types, all windows have basically the same structure (format). Figure 2-3 shows a terminal window. Every window, whether it's a terminal or graphics window, is comprised of two main parts: the **user area** and the **border**.

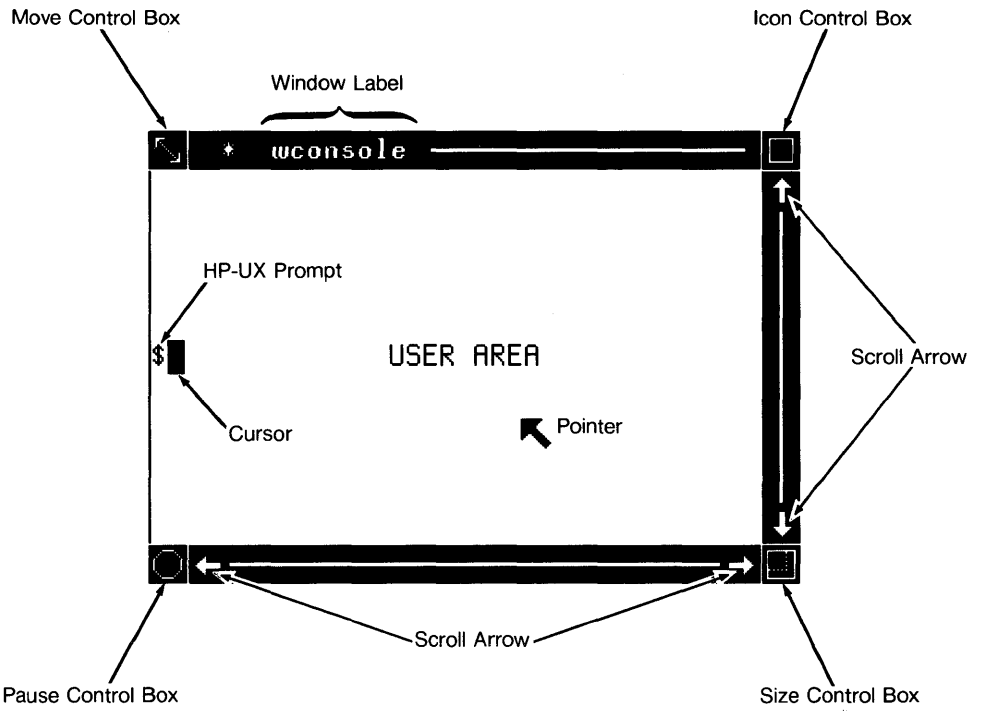

Figure 2-3. Window Structure

The User Area

Interaction with programs take place through the user area (also known as the **contents** area). In other words, when a program executes in a window, you see the program's output through the user area. The user area is analogous to a terminal screen.

For example, suppose you have a shell running in a window. Any commands you type at the keyboard will appear in the window's user area; in addition, any output from the commands goes to the user area also. Its just as if you're typing commands at a regular terminal, except that the output appears within the window, instead of the whole screen.

The Border

The border surrounds the user area. Notice in the previous Figure 2-3 that the left edge of a window has a thin border while the other edges are wider. Within the border are the window's name, symbols representing the status of the window, and areas that allow you to control the window and its relation to other windows. Note the names and locations of the various parts of the label—they will become important when you start manipulating windows interactively.

Each window, regardless of window type, can have a **normal** or a **thin** border. Figure 2-3 shows a window with a normal border; figure 2-4 shows a window with a thin border. Notice that none of the normal border areas—window name and interactive control areas—exist in a thin border.

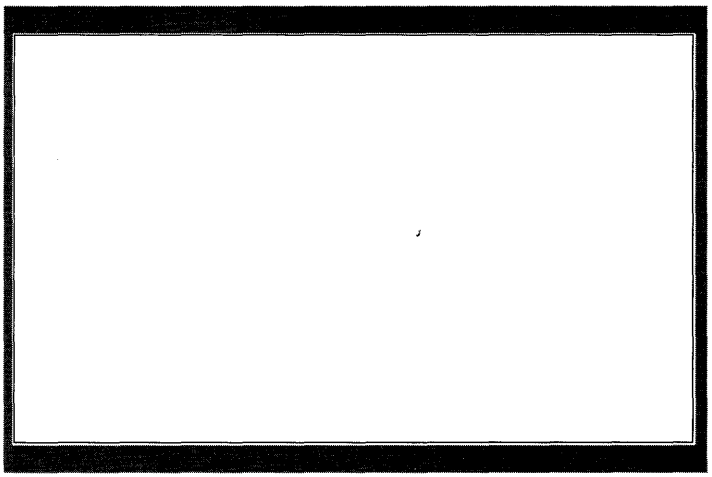

Figure 2-4. A Thin Border

The graphics window type also supports a **null** border—that is, graphics windows without any border at all. Figure 2-5 shows a graphics window with a null border.

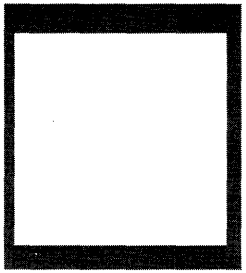

Figure 2-5. A Graphics Window with a Null Border

Selected Window

In Figure 2-3, note the asterisk (*) preceding the window name, and the presence of a line extending throughout the middle of the entire border. This indicates that this window is **selected**.

The selected window is important because anything you type at the keyboard is sent *only* to the selected window. The keyboard is associated with one window at a time, therefore it is important to know which window is currently being used. You can tell which window is selected by looking for an asterisk in front of the window name and a line in the border (as in Figure 2-3).

Applications running in a non-selected window *cannot* take input from the keyboard. Therefore, it is important to know how to select a window. Selecting a window is discussed in detail in the "Interactive Use" and "Using Commands" chapters.

The Pointer (Echo)

When the window system is running, there is a display **pointer**. You can move this pointer to different locations on the display screen by using keys on the keyboard, or the optional mouse or graphics tablet. (The subsequent section, "Moving the Pointer," describes how to move the pointer via either the keyboard, mouse, or graphics tablet.)

The pointer is also called an **echo**. It gets this name because it echoes the screen location specified through the keyboard, mouse, or graphics tablet.

As the pointer moves on the screen, it changes shape over different areas. Table 2-1 shows the various pointer shapes and describes which screen area causes the pointer to take the shape.

Table 2-1. Standard Pointer Shapes

Pointer	Screen Area
	When the pointer is *not* located over any windows or softkeys, that is, when it is located over the screen background area, the pointer is a box with a dot in the middle.
	When positioned over a window's border, the pointer is a small cross-hairs.
	When located over a window's user area or a shifted softkey, the pointer is an arrow pointing up and left.
	When positioned over an unshifted softkey, the pointer is an arrow pointing down and left.

Display Screen Coordinates

Some window system commands, such as *wmove(1)*, *wcreate(1)*, *wsh(1)*, and *wsize(1)*, require an understanding of display screen coordinates. This section discusses display screen coordinates in detail.

The Origin

The upper-leftmost pixel on the display screen (known as the **origin**) has coordinates *0,0*. The *x* coordinates increase as you move to the right; *y* coordinates increase downward. Figure 2-6 illustrates this concept.

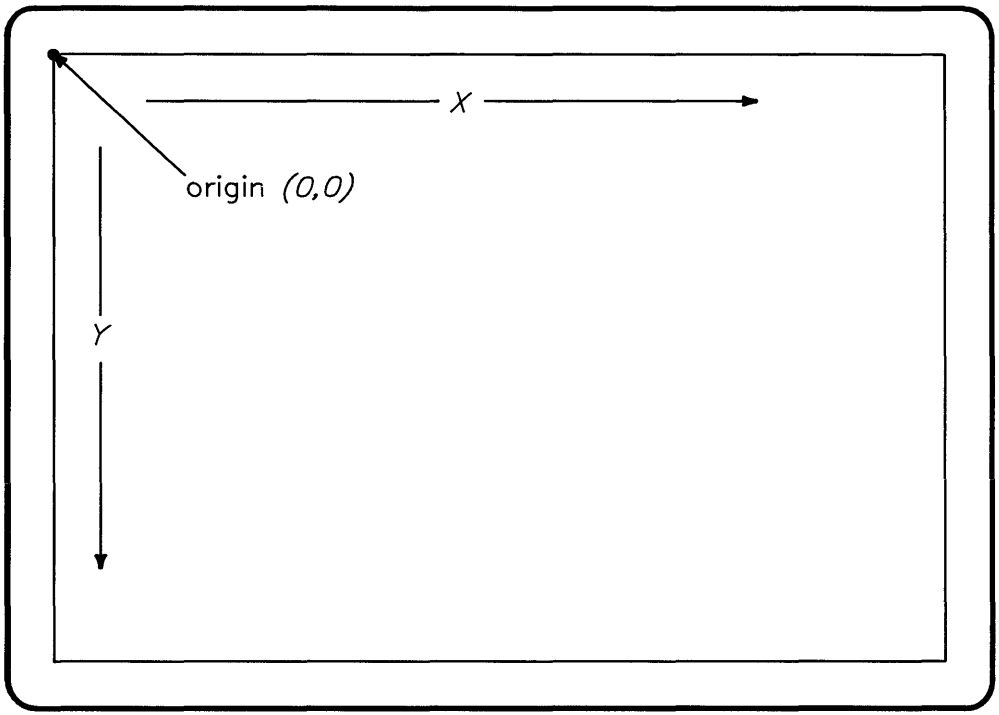

Figure 2-6. Display Screen Coordinates

Maximum Screen Coordinates

The coordinates of the lower-rightmost pixel on the display screen depend on the resolution of your display screen. HP Windows/9000 supports two resolutions: high-resolution and low-resolution.

Low-Resolution Displays

Low-resolution displays are 512 pixels wide by 400 pixels high. This means the range of displayable coordinates is from *0,0* to *511,399*. Figure 2-7 illustrates this concept and shows some example coordinates.

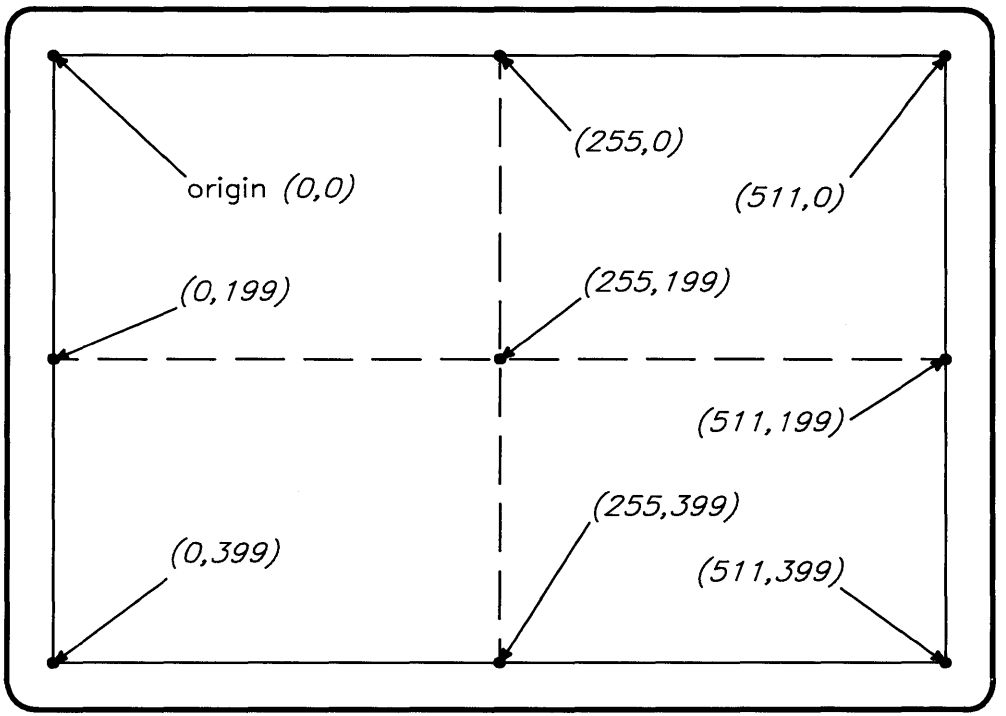

Figure 2-7. Low-Resolution Display Coordinates

High-Resolution Displays

High-resolution displays are either 1024×768 or 1280×1024 pixels. Therefore, the valid coordinate ranges are *0,0* to *1023,767* and *0,0* to *1279,1023*, respectively. Figure 2-8 shows a 1024×768 screen with some sample coordinates. Figure 2-9 shows a 1280×1024 display with some sample coordinates.

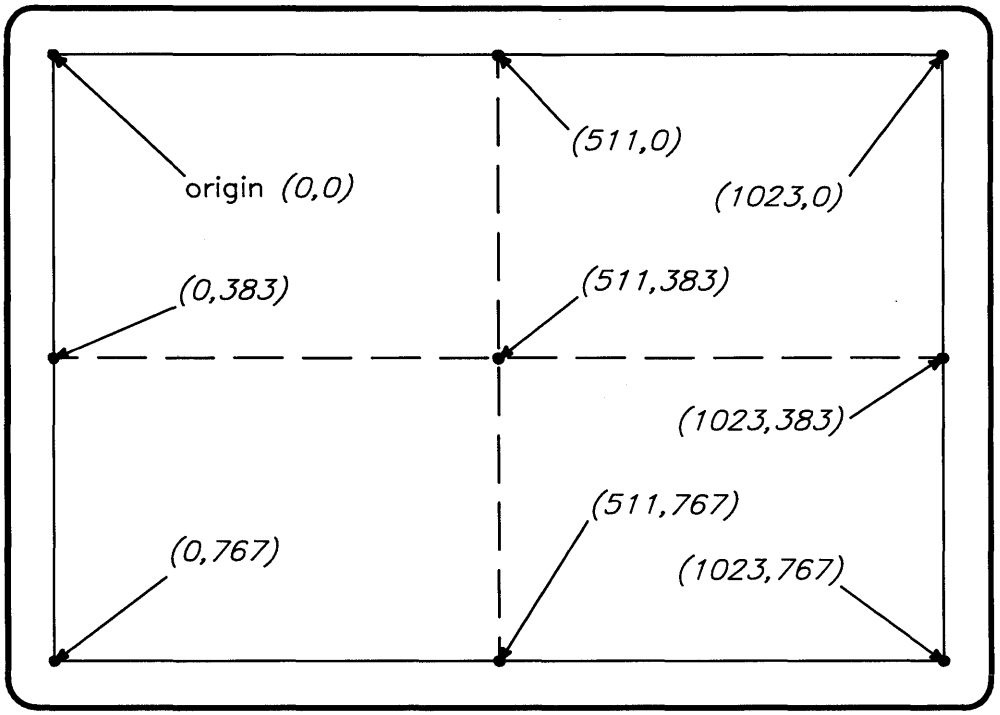

Figure 2-8. High-Resolution Display Coordinates

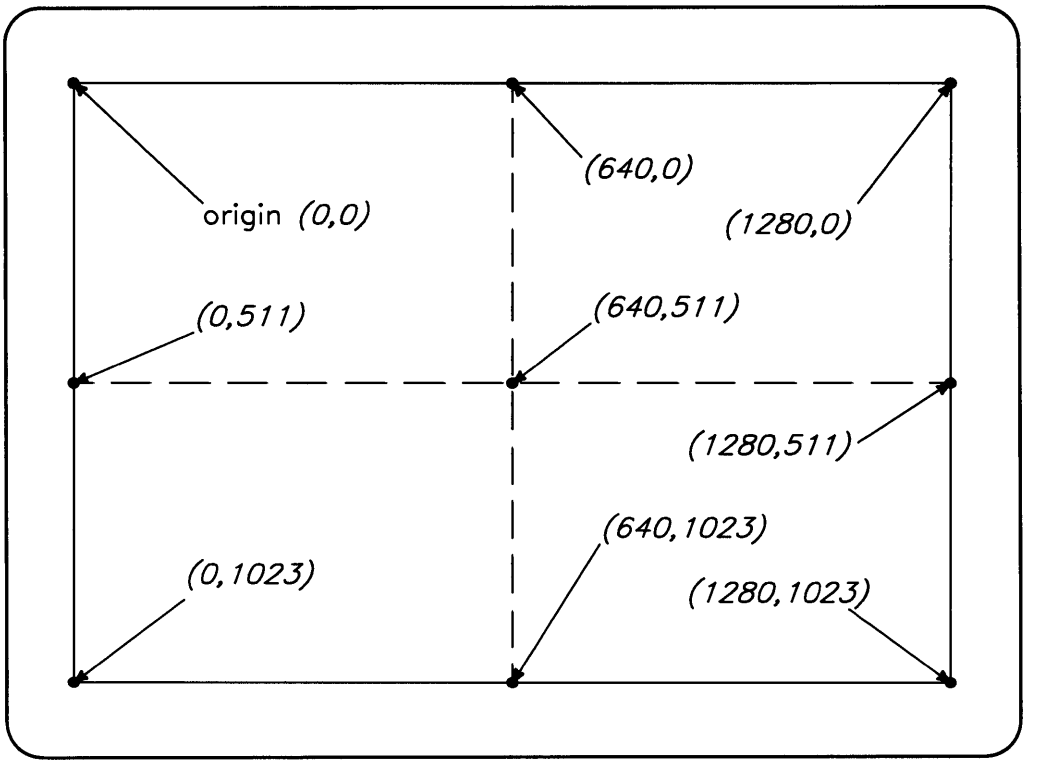

Figure 2-9. Very High Resolution Display Coordinates

Cursor

The pointer is *not* the same as a **cursor**. A cursor is a special character which is used to mark where the next character typed at the keyboard will appear in a terminal window.

For example, a cursor is present when an HP-UX shell is running (as in Figure 2-3). A shell can run in a window, and thus the window contains a cursor. Remember that a cursor is confined to a window, but the pointer is not.

Note that it *is* possible to turn the cursor off, and some applications may do this. (The *Term0 Reference Manual* and the *HP Windows/9000 Programmer's Manual* describe how to turn the cursor off in a window.)

Moving the Pointer

The window system accepts instructions from various devices. While it is possible to use only a keyboard to interact with Windows/9000, a graphic input device (such as a mouse or graphics tablet) is much more effective. These devices allow fast and easy interaction with the window system.

The Mouse

The mouse is one of the most commonly used HP-HIL input devices (see Figure 2-10). As you move the mouse along a flat surface, such as your desk top, the pointer moves correspondingly on the display screen.

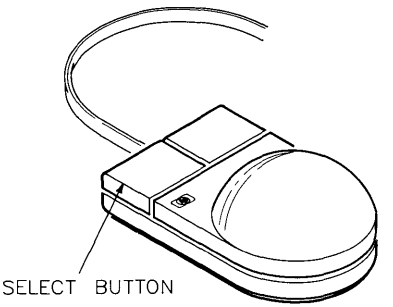

Figure 2-10. The Mouse

The left button on the mouse (as you face the cord, see Figure 2-10) is the **select button**. You perform interactive operations with the mouse by moving the pointer to some location on the screen, such as a window's border, and **clicking** the select button.

For example the box in the upper left corner of the border on a window represents the *move* operation. By moving the pointer to that area and clicking the select button, the *move* operation is activated. You can *select* a window in a similar manner. (Performing interactive operations with a mouse is described in the "Interactive Use" chapter.)

Graphics Tablet

An HP-HIL graphics tablet can also be used as an interactive device with the window system. You can use a **puck** (flat device with selection buttons) or a **stylus** (pen device) with the graphics tablet. Figure 2-11 shows a graphics tablet puck (on the left) and stylus (on the right).

The tablet stylus or puck moves the pointer in nearly the same manner as the mouse. However, the graphics tablet is unique in that every point on the screen corresponds to a point on the graphics tablet.

You can take the mouse off the table surface, place it in another location, and the pointer will respond only when the mouse actually moves on the table. With the graphics tablet, lifting the stylus or puck off the tablet and placing it in another location affects the pointer's location on the screen immediately.

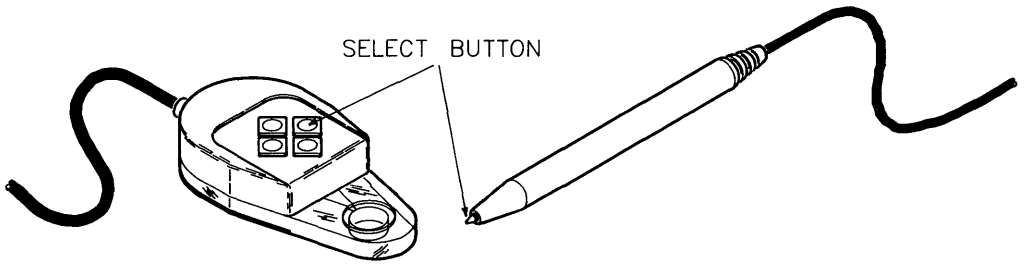

Figure 2-11. The Puck and Stylus

Like the mouse, the puck and stylus also have a select button. The select button on the puck is the leftmost button facing the cross-hairs. The stylus select button is activated by pressing the point of the stylus onto the graphics tablet.

As with the mouse, you can perform interactive operations by moving the pointer to a position on the screen and pressing the select button.

Keyboard

If you have neither the mouse nor the graphics tablet, you can still use windows. Using keys on the keyboard, you can move the pointer and perform the same operations as with the mouse or graphics tablet.

Moving the Pointer

To move the pointer via the keyboard, press an arrow key while holding down the CTRL key. To keep the pointer moving, hold the keys down. Releasing the keys stops the pointer movement. Table 2-2 shows the key combinations required to move the pointer.

If you fail to press the CTRL key along with the arrow keys, the pointer will not move.

Table 2-2. Moving the Pointer using Keys.

Key	Operation
CTRL-◄	Moves the pointer left.
CTRL-►	Moves the pointer right.
CTRL-▲	Moves the pointer up.
CTRL-▼	Moves the pointer down.

The Keyboard Select Button

Like the mouse and graphics tablet, the keyboard too has a select button—the Select key. Pressing the Select key has the same effect as pressing the other devices' select buttons.

Special Keys

In addition to the pointer keys and the Select key, other keys perform special functions when pressed within the window system. Table 2-3 shows these keys and describes their function.

Table 2-3. Special Keys.

Key	Description
[Shift]-[Select]	Shuffles windows on the display screen. The resulting topmost window is automatically selected.
[Break]	Stops execution of the program in the selected window.
[Stop]	Pauses output to the selected terminal window. To resume output after it has been paused, simply press this key again. This key works only with terminal windows.
[Menu]	Controls whether or not a softkey menu is displayed at the bottom of the display. If no softkey menu is currently displayed, then pressing [Menu] will cause a softkey menu to be displayed for the selected window. If a softkey menu *is* displayed, then pressing [Menu] turns off the softkey menu.
[Shift]-[User]	If no softkey menu is displayed for the selected window, then pressing [Shift]-[User] has the same effect as the [Menu] key—the softkey menu for the selected window is displayed at the bottom of the screen. However, [Shift]-[User] does **not** turn softkey labels off. Note that this key works only with terminal windows.
[System]	Causes the selected window's terminal configuration menu to appear at the bottom of the screen. Note that this key works only with terminal windows.

Pop-Up Menus

Operations on windows can be performed three ways:

- by executing commands
- interactively with a pointer in a window's border
- interactively via the pointer and **pop-up menus.**

Pop-up menus are useful when you don't wish to use commands and when you want to perform an operation on a window whose border is inaccessible (e.g., covered by other windows or off screen).

Activating a Pop-Up Menu

Pop-up menus are activated by moving the pointer to a special screen location and pressing the select button. The location of the pointer when the select button is pressed determines which window the pop-up menu is invoked for:

- If the pointer is not over any window, that is, if it is over the background pattern on the screen, then a pop-up menu is displayed for the *selected* window.
- To get a pop-up menu for a window other than the selected window, you must move the pointer over the desired window's border—not over any of the control boxes in the window's border.

Pop-Up Menu Format

Activating a menu causes it to pop up at the screen location specified by the pointer. Once the menu is displayed, you can make selections from the menu. Figure 2-12 shows a typical menu for a window.

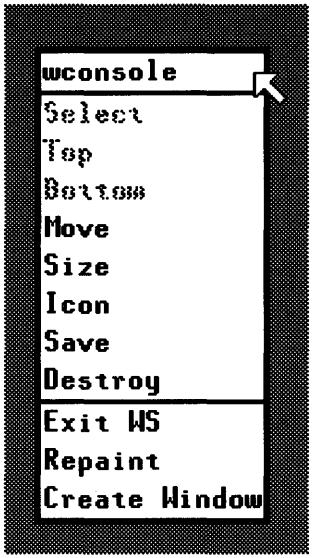

Figure 2-12. A Pop-Up Menu

The name of the window for which the menu was invoked appears at the top of the menu, and the various menu selections appear underneath the name.

Some items in the menu may not be selectable. Items that are not selectable appear in grey letters, while selectable items appear in black letters.

Note also that the last three items in the menu—*Exit WS*, *Repaint*, and *Create Window*—are separated from the other items by a horizontal bar. All items above this bar apply only to the window for which the menu was invoked; these items are called **local** items because they are local to the window. Selections below the bar have no relation to the window and are known as **global** items. Global items are normally always selectable, whereas some local items may not be selectable.

Using the Pop-Up Menu

To select an item, simply move the pointer to the item and press the select button. You will notice that selectable items will be highlighted (inverted) as the pointer moves over them; non-selectable items are not highlighted. Figure 2-13 shows a menu for the window named *wconsole*; the *Move* item is highlighted.

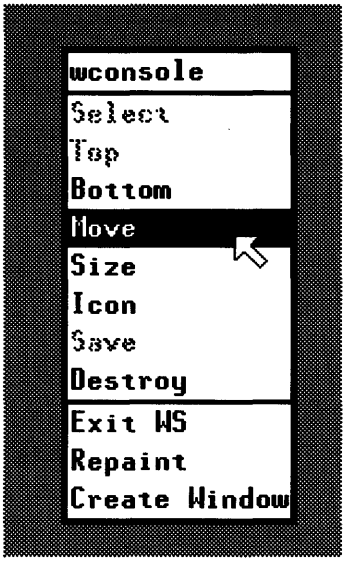

Figure 2-13. Selecting an Item from a Pop-Up Menu

After you press the select button, the operation specified by the menu item will be performed.

Exiting a Pop-Up Menu

Sometimes you may want to exit a pop-up menu without making a selection. To abort a pop-up menu, you can do one of the following:

- move the pointer in a quick motion out of the menu area (this is the default, but can be changed; see WMIUICONFIG section in "Environment Variables" chapter)
- press any disabled button (by default, the rightmost mouse button and puck switch buttons other than the leftmost button abort the menu)
- press any key on the keyboard (other than Select)
- wait for a sufficiently long time (by default, 60 seconds) and the menu will disappear
- move the pointer to a non-selectable item and press the select button.

If you do any of these, the menu will disappear from the screen, and you'll hear a beep to indicate that the menu was aborted.

Icons

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.

Why Use Icons?

To understand the usefulness of icons, let's return to the desk top analogy. Suppose your desk top is becoming covered with papers—becoming less managable as you have more tasks to maintain. To fix this problem you might set the less-important papers—i.e., papers that don't require your *immediate* attention—off in a corner of your desk top until they are needed later.

You can do the same with windows. For example, if you have applications running in several windows at once, you can turn less-important windows into icons. Then when you need to use the application later, simply change the icon back to a window.

Note that changing a window to an icon does not stop any application running in the window: the application will still continue to run, and any output sent to the window (when in the iconic state) will be lost.

Icon Format

Although an icon is referred to as the "shrunken form of a window," its format is somewhat different. Unlike a window, 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**. Only the first 12 characters of the window's label are displayed in the icon label. Figure 2-14 shows the format of an icon.

Figure 2-14. Icon Format

Clicking the locator over the icon's *picture* or *window label* will invoke a pop-up menu for the icon.

Two interactive manipulation symbols appear within the label area:

 moves the icon

 returns the window to normal representation

Icon Types

Term0 and graphics windows each use default, predefined pictures when an icon is displayed. This is so that you can distinguish between the icon for a terminal window and a graphics window. Figure 2-15 shows a terminal window icon (on the left) and a graphics window icon (on the right).

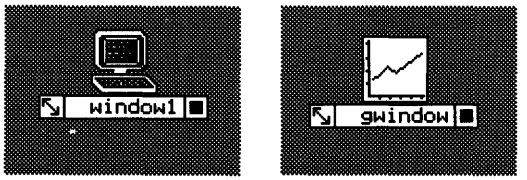

Figure 2-15. A Terminal Icon and Graphics Icon

Interactive Use 3

HP Windows/9000 allows you to perform many interactive window operations via the keyboard and optional mouse or graphics tablet. For example, you can move windows, change their size, and change them to icons. This chapter discusses how to perform interactive operations with windows; specifically, the following topics are covered:

- starting the window system
- leaving the window system
- creating a terminal window
- destroying a window (or icon)
- moving a window
- changing a window's size
- selecting a window
- bringing a window to the top of the stack
- putting a window on the bottom of the stack
- changing a window to an icon
- moving an icon
- changing an icon to a window
- pausing terminal window output
- scrolling window information
- saving a window
- repainting the screen.

Starting HP Windows/9000

Before discussing how to interactively manipulate windows, the window system must be running. This section discusses how to start up the window system.

The wconsole Window

By default when the window system starts up, a terminal window named *wconsole* is created and displayed in the upper-left corner of the display (see Figure 3-1). This window contains an HP-UX shell (either a Bourne shell or C-shell, depending on the value of the SHELL environment variable; for details, see the "Concepts" section of the "Using Commands" chapter).

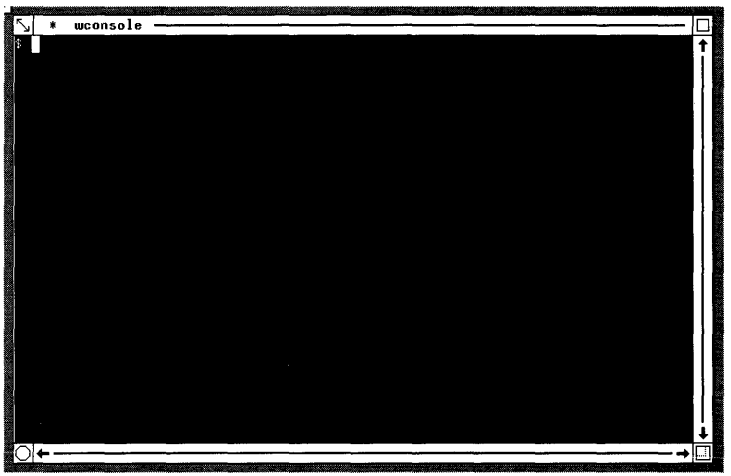

Figure 3-1. The wconsole Window

The window is selected when it is created; therefore, anything you type at the keyboard will be sent to this window. To execute HP-UX commands or non-graphics applications in this window, simply enter the command or program name as you would from a regular terminal.

Automatic Startup

Depending on how your HP-UX operating system is configured, HP Windows/9000 may or may not automatically start up on its own when you log in. If your system is configured to automatically start windows, then the *wconsole* window should be displayed shortly after you log in. If this is the case, then you needn't worry about starting the system and can move onto subsequent sections in this chapter.

Note

The "Starting Windows/9000" section of the "Using Commands" chapter describes the various methods for automatically starting the system.

Executing the wmstart Command

If your window system does not automatically start running when you log in, then you must start the system via the *wmstart(1)* command. Simply enter the command to the HP-UX prompt:

```
wmstart [Return]
```

Shortly thereafter, the *wconsole* window will appear, and you can start performing interactive operations described in the remainder of this chapter.

Leaving HP Windows/9000

When you are through using the window system, you should exit from it. When you leave the window system, most processes associated with the system are killed (exceptions are *nohup*ed process; see *nohup(1)* in the *HP-UX reference*, for details).

This means that not only does the window system itself die, but also any programs that are running when you kill it. You should, therefore, be absolutely sure you are ready to leave the system before performing this operation.

When you exit the window system, the screen is cleared; and depending on how your system is configured, you'll either:

- be returned to the HP-UX command prompt, or
- be logged off of the HP-UX system.

Action

There are two ways to exit the window system: via the *wmstop(1)* command or the pop-up menu.

The wmstop Command

To leave the window system, simply enter the *wmstop* command from a selected terminal window (such as the *wconsole* window):

```
wmstop Return
```

The Pop-Up Menu

To leave via the pop-up menu:

1. **Invoke a pop-up menu.** You can do this by moving the pointer over the background pattern and clicking the select button. (See the "Pop-Up Menus" section of the "Concepts" chapter for details on getting a pop-up menu.)
2. **Highlight the *Exit WS* option of the pop-up menu.** This is done by moving the pointer to this option in the menu. Figure 3-2 shows a pop-up menu with the *Exit WS* option highlighted.

Figure 3-2. Highlighting the *Exit WS* Option

3. **Select the *Exit WS* option.** This is done by clicking the select button when this option is highlighted. After selecting this option, a *verification* menu will appear (as shown in Figure 3-3).

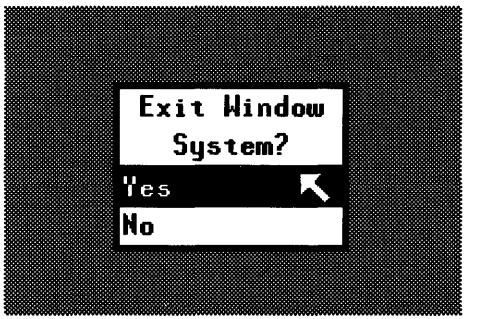

Figure 3-3. The Verification Menu

4. **Select *Yes* or *No*.** If you *do* want to leave the window system, then highlight and select *Yes* from the verification menu; if you wish to remain in the window system, then highlight and select *No*.

Things That Can Go Wrong

It is possible to accidentally activate the *Exit WS* item of a pop-up menu. If you've accidentally activated this option and would like to abort, you can easily cancel the menu using any of the following methods:

- choose the *No* option of the verification menu
- press a disabled button—a button other than the select button
- press a key other than Select
- quickly move the pointer out of the verification menu
- wait sufficiently long (by default, 60 seconds) for the pop-up menu to time-out (automatically abort after 60 seconds).

In any case, you'll be returned to the window system.

Creating a Terminal Window

New terminal windows can be created via the pop-up menu. (Note that *only* terminal windows, and not graphics windows, can be created interactively via the pop-up menu; you must use commands to create graphics windows.)

New windows are created in a stair-step fashion: the first window (*wconsole*) is created at the upper-left corner of the display, and subsequent windows are created down and to the right of the previous window.

When a new window is created, it automatically becomes the selected window. In addition, each new window is given a default name by the window system:

```
windown
```

where *n* is a sequential number starting at one. For example, the first window created after *wconsole* is named *window1*; the second, *window2*; and so on. (Note that when you create a window via commands, you have the option of assigning a name other than the default.)

Figure 3-4 shows *wconsole* and three more windows, created via the pop-up menu. Note how the windows stair-step down from the upper-left corner of the screen, and how the last window created, *window3*, is selected. The stair-step pattern is repeated after every fifth window.

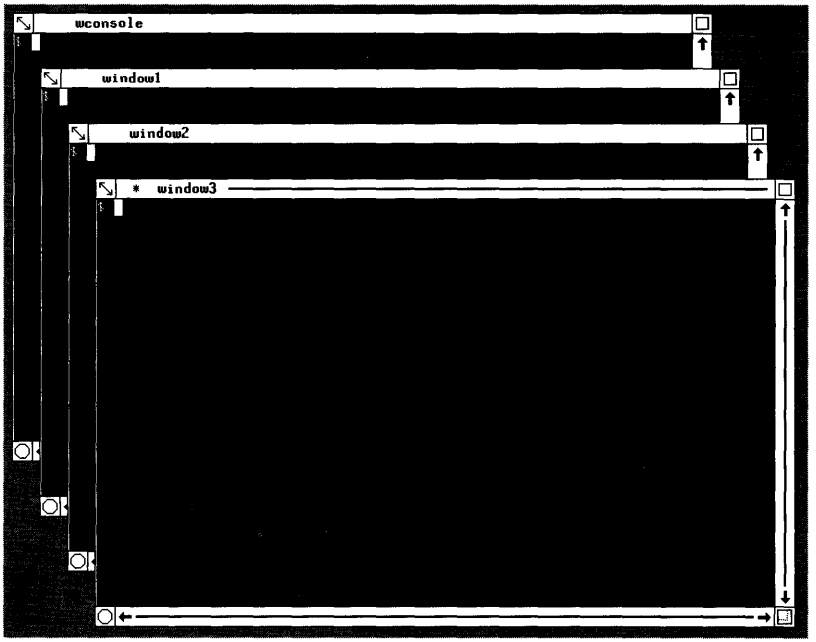

Figure 3-4. Stair-Stepping Windows

Action

1. **Invoke a pop-up menu.** You can do this by moving the pointer over the background pattern and clicking the select button. (See the "Pop-Up Menus" section of the "Concepts" chapter for details on getting a pop-up menu.)
2. **Highlight the *Create Window* option of the pop-up menu.** This is done by moving the pointer to this option in the menu. Figure 3-5 shows a pop-up menu with the *Create Window* option highlighted.

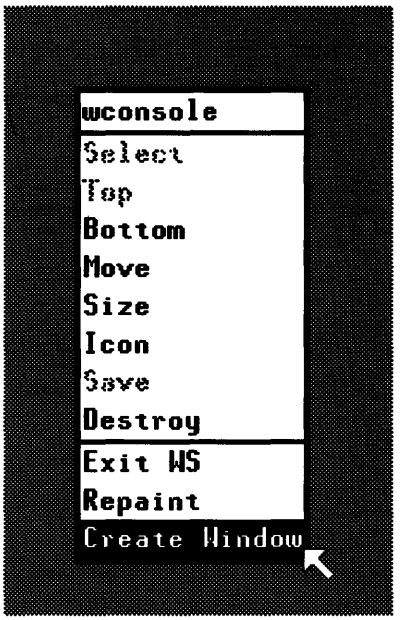

Figure 3-5. Highlighting the *Create Window* Option

3. **Select the *Create Window* option** by clicking the select button when this option is highlighted. After selecting this option, a new terminal window will appear.

Things That Can Go Wrong

- If you select the wrong item in the pop-up menu (such as *Repaint*) you will have to bring up the menu again.
- There is a limit on the number of windows you can create. The maximum number of windows that you can create is somewhere between four and twenty-seven, depending on the amount of memory in your computer system, kernel configuration for such things as maximum number of user processes, and the value of certain environment variables. (For details on the maximum number of windows, see the chapter "Environment Variables.")

Destroying a Window or Icon

When you are through using a window or an icon, you can destroy it—i.e., remove it totally from the system. The pop-up menu is used to interactively destroy a window.

All processes (programs) in the destroyed window (or icon) are killed (except *nohup*ed processes; see *nohup(1)* in the *HP-UX Reference*). Therefore, **make sure you really wish to destroy a window or an icon before you perform this task.**

If you destroy the selected window, the resulting topmost window in the stack becomes the selected window.

Action

1. **Bring up a pop-up menu for the window you wish to destroy.**
   - To get the pop-up menu for a window, move the pointer over the window's border and click the select button.
   - To get the pop-up menu for an icon, move the pointer over the icon's picture and click the select button.

   (See the "Pop-Up Menus" section of the "Concepts" chapter for more information.)
2. **Verify the menu name.** Compare the name at the top of the pop-up menu with the name of the window you want destroyed. If the names are not the same, you have selected the wrong window; exit the pop-up menu and try again.
3. **Highlight the *Destroy* option.**
4. **Click the select button** to activate the *Destroy* item. The window, and programs running in it, will disappear from the screen.

Things That Can Go Wrong

If you accidentally destroy a window, you cannot retrieve the window. Therefore, be prudent when using this option. Remember that you don't have to make a menu selection; you can abort the menu if you wish.

If you accidentally destroy all windows on the screen, you can still access a *System Menu* pop-up menu. You can perform only global options from this menu: *Exit WS*, *Repaint*, and *Create Window* (see Figure 3-6). Therefore you can either leave the system, repaint the screen, or create a new window. You will not, however, be able to retrieve the destroyed windows.

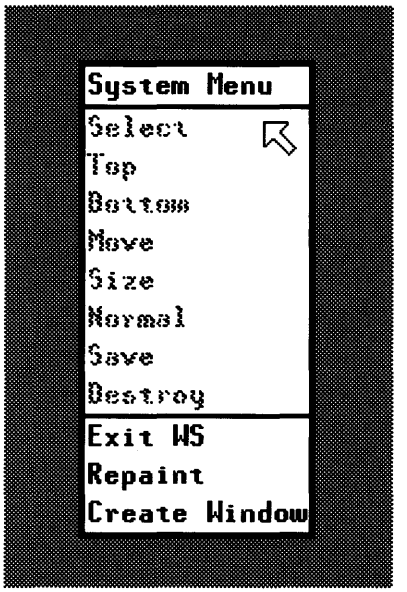

Figure 3-6. The System Menu

Moving a Window

This operation allows you to move a window to different locations on the screen. The window can be moved anywhere on the screen and can be moved partially off of the screen.

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

Action

You can interactively move a window either by the **move control box** or a pop-up menu.

Using the Move Control Box

1. **Move the pointer to the upper-left box in the border of the window you wish to move.** This box is known as the **move control box**. The pointer changes to cross-hairs when in this box. Make sure the pointer is in the box as shown in Figure 3-7.

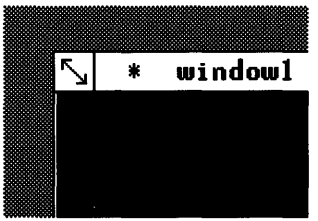

Figure 3-7. The Move Control Box

2. **Click the select button** to activate the *move* operation. A dotted rectangle will appear, surrounding the user area. Notice that you can move this rectangle much the same way that you move the pointer. This rectangle is important because you use it to designate the new location for the window.

   Figure 3-8 shows a window for which the *move* operation has been activated. In this case, the window will be moved down and to the right, as specified by the dotted rectangle.

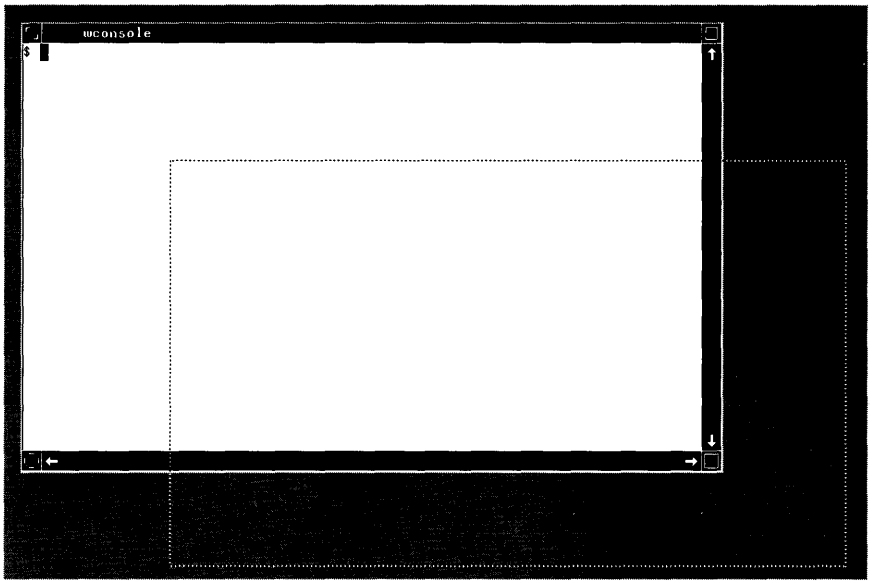

Figure 3-8. Moving a Window

3. **Move the dotted rectangle to the desired new location** for the window; click the select button when the rectangle is at the location. The window will move to the new area.

   Note the dotted rectangle corresponds to the user area and not the border, and you may want to compensate for the border (if you get close to the screen edge).

Using a Pop-Up Menu

If the move control box is inaccessible, you may wish to use the pop-up menu for the *move* operation:

1. **Invoke a pop-up menu for the desired window.**
2. **Highlight the *Move* item.**
3. **Click the select button** and follow step 3 above.

Changing a Window's Size

You can interactively change the size of any window. The largest a window can be is the size it was when you created the window. There is also a minimum size for a window. You can find these sizes by experimenting with this operation.

You can change the size of a window from its size when created to a smaller size if desired. But what happens to the information in a terminal window when you shrink it? The information is not lost, the viewing area simply becomes smaller.

You will see later (in the section "Scrolling Terminal Window Information") that it is possible to scroll the information in the viewing area of a terminal window up, down, left, and right to view the information in this smaller window.

Action

You can change a window's size by using either the **size control box** or the pop-up menu.

Using the Size Control Box

1. **Move the pointer to the box in the lower right corner of the border.** This is known as the **size control box** (see Figure 3-9).

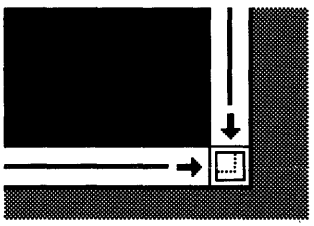

Figure 3-9. Size Control Box

2. **Click the select button** to activate the *size* operation. As with the *move* operation, a dotted rectangle appears around the user area of the chosen window. Note that you can change the size of this rectangle by moving the pointer device. This rectangle is important because you specify the window's new size with it.
3. **Change the dotted rectangle to the desired new size** for the window. As an example, the *wconsole* window shown in Figure 3-10 will be changed to approximately one-fourth of its original size, as specified by the size rectangle.

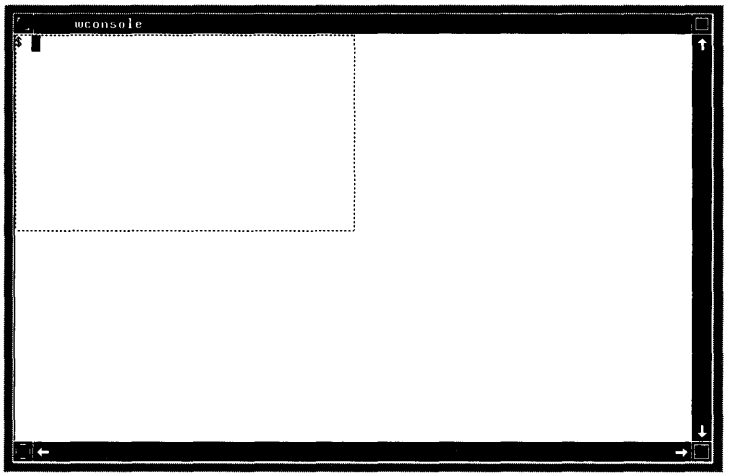

Figure 3-10. The Size Rectangle

4. When you've changed the rectangle to the desired window size, **Click the select button**. The window will change to the size of the rectangle.

Using a Pop-Up Menu

1. **Bring up a pop-up menu for the desired window.**
2. **Highlight the *Size* option.**
3. **Click the select button** to perform the *size* operation.
4. **Follow steps 3 & 4** from the "Using the Size Box" section above.

Selecting Windows

As mentioned in the "Concepts" chapter, keyboard input is sent only to the selected window, and only *one* window can be selected at a time. In order to communicate with an application in a particular window, you must first *select* the window. Once the window is selected, all keystrokes are sent to the window.

It is possible to select an icon. In this case the icon name is preceded by an asterisk. Anything typed while the icon is selected cannot be seen; you must first change the icon to a window before seeing output.

Actions

There are three methods for interactively selecting a window:

- you can select it and automatically bring it to the top of the window stack as it is selected
- you can select it but leave it at its position within the window stack
- you can shuffle the bottom window to the top and have it automatically selected.

Selecting and Topping

This method of selection automatically brings the window to the top of the stack of windows:

1. **Move the pointer to the user area of the window you wish to select.**
2. **Click the select button.** The window becomes the selected window and moves to the top of the stack.

Selecting a Window without Topping It

If you wish to select a window without moving it in the stack, this method keeps the selected window in place:

1. **Bring up a pop-up menu for the desired window.**
2. **Highlight the *Select* option.**
3. **Click the select button** to activate the *select* operation. The window becomes the selected window, but does *not* move within the display stack.

Shuffling Windows

Shuffling windows brings the window on the bottom of the stack to the top and selects that window. The other windows in the stack remain in the same position with respect to each other.

To shuffle windows in this manner, simply press the [Shift] and [Select] keys simultaneously. The window on the bottom of the stack moves to the top and automatically becomes the selected window.

Things That Can Go Wrong

- If you do not position the pointer in the user area when selecting a window, you may activate the wrong operation or a pop-up menu. Remember that the pointer is in the shape of an arrow while in the user area.
- You may inadvertently select the wrong window. In this case, simply perform the select operation again to select the correct window.
- If you do everything correctly and nothing happens, you may have selected a window which is already selected. If you use a pop-up menu, make sure the *Select* item is highlighted. If your pointer is over the window name, nothing will happen. If the *Select* item is greyed, you have chosen a window already selected.

Bringing a Window to the Top of the Stack

If you have more than one window on the screen, and some overlap, you may find it useful to bring a window to the top where its information can be viewed easily. This operation is performed via the pop-up menu.

Note that bringing a window to the top does not select the window. See "Selecting a Window" for details on selecting a window when bringing it to the top.

Action

1. **Invoke a pop-up menu** for the window that you wish to bring to the top of the window stack.
2. **Highlight the *Top* option.**
3. **Click the select button.** The window is then displayed as the top window in the stack.

Putting a Window on the Bottom of the Stack

This operation is useful when you have overlapping windows, and you want to move one window underneath the others. The pop-up menu is used to place a window on the bottom of the display stack.

Action

1. **Bring up a pop-up menu** for the window to place on the bottom of the stack.
2. **Highlight the *Bottom* option.**
3. **Click the select button.** The window will move to the bottom of the stack.

Note that moving a window to the bottom of the display stack does not affect its selection status. See the "Selecting a Window" section for details on selecting a window.

Changing a Window to an Icon

As mentioned in the "Concepts" chapter, changing a window to an icon is useful when you temporarily want to move a window out of the way. When the window is needed later, it can be changed back to a window.

Action

You can change a window to an icon using either the **icon control box** or a pop-up menu.

Using the Icon Control Box

1. **Move the pointer to the box in the upper right corner of the window's border.** This is known as the **icon control box** (see Figure 3-11).

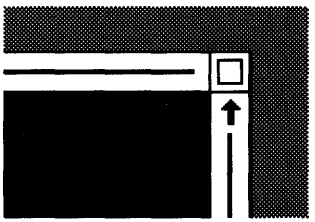

Figure 3-11. Icon Control Box

2. **Press the select button** to change the window to an icon. The window will disappear from the screen, and an icon will appear on the lower left portion of the screen. The icon will display the name of the window. If the icon covers part of another window, applications in the covered window will still execute properly.

Using a Pop-Up Menu

1. **Bring up a pop-up menu** for the window you wish to change to an icon.
2. **Highlight the *Icon* option.**
3. **Click the select button** to change the window to an icon.

Moving an Icon

Like windows, icons can be moved on the display screen. You can use either the **icon move box** or a pop-up menu.

Action

Using the Icon Move Box

1. **Move the pointer to the leftmost box in the icon's label.** This is known as the **icon move box**. Figure 3-12 shows a terminal window's icon with the icon move box labelled; be sure to center the pointer within this box.

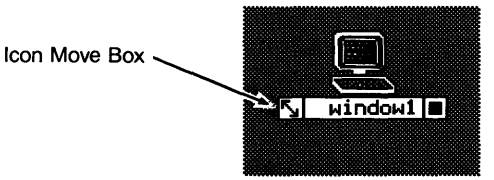

Figure 3-12. The Icon Move Box

2. **Click the select button.** A small dotted rectangle will appear around the icon. You specify the icon's new location by moving the rectangle, similar to the way you move a window.
3. **Move the rectangle to the desired new location.**
4. **Click the select button.** The icon moves to the new location.

Using a Pop-Up Menu

1. **Invoke a pop-up menu for the icon** that you wish to move. You can get a pop-up menu for an icon by clicking the select button when the pointer is over the icon's picture.
2. **Highlight the *Move* option** of the pop-up menu.
3. **Perform steps 3 and 4** from the section above.

Changing an Icon to a Window

When you need a window that is currently iconic, you can change it back to a window using either the **icon control box** or the pop-up menu.

Action

Using the Icon Control Box

1. **Move the pointer to the rightmost box in label of the icon that you wish to change back to a window.** This box is known as the icon's **icon control box**. Figure 3-13 shows a terminal window's icon with the icon control box labelled; be sure to get the pointer directly over this box when performing this task.

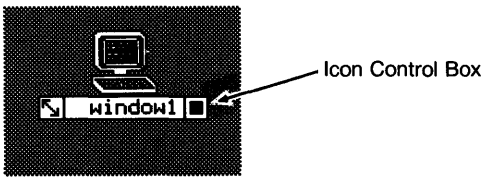

Figure 3-13. The Icon Control Box

2. When the pointer is over the icon control box, **click the select button.** The icon is changed back to a window.

Using a Pop-Up Menu

1. **Invoke a pop-up menu for the icon** by moving the pointer over the icon's picture area and clicking the select button.
2. **Highlight the *Normal* option** of the pop-up menu, as shown in Figure 3-14.

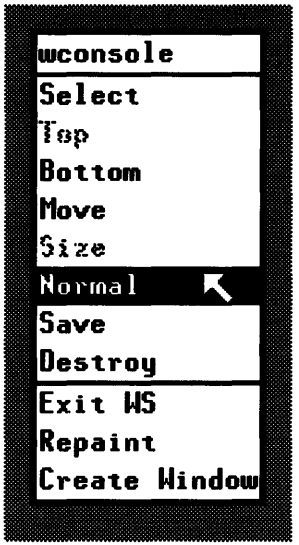

Figure 3-14. Icon-to-Window Pop-Up Menu

3. **Click the select button.** The icon will then change back to a window. It will be in the same location and will contain the same information that it had when it was changed to an icon.

Pausing Terminal Window Output

This operation allows you to halt and restart output in a terminal window. For example you may have window output (resulting from a command or program) which is scrolling too quickly for you to read. You can stop the scrolling with this operation and restart it when ready.

Note: This operation works only with terminal windows. You cannot pause graphics window output via this operation.

Action

You can use either the **pause control box** or the keyboard to pause output in a terminal window.

Using the Pause Control Box

1. **Move the pointer to the box in the lower left corner of the desired window**, known as the **pause control box**. Figure 3-15 shows the pause control box.

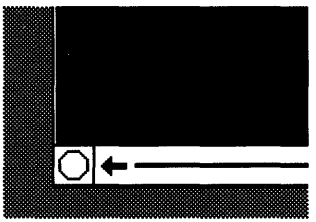

Figure 3-15. The Pause Control Box

2. When you wish to pause the window output, **click the select button**. Note that the octagon becomes highlighted. This indicates that the *pause* operation is activated.
3. **To restart the output, click the select button again over the pause control box.** The area in the shape of a stop sign will return to its original form.

Using the Keyboard

1. **Press the Stop key to pause window output.**
2. **To restart output, press Stop again.**

Scrolling Information in a Window

This operation allows you to scroll the information in the user area of a window. This operation is especially useful when a window contains more information than can be shown in its user area. You can scroll a window's contents up, down, right, or left.

Inside the right and lower border of a window you will see small arrows near the control boxes (see Figure 3-16). The scroll arrows scroll the screen in the indicated directions.

Figure 3-16. Scroll Arrows

Action

1. **Move the pointer to a scroll arrow that points in the direction that you wish to scroll.**
2. **Click the select button.** The information is scrolled one character for each click of the select button.

Note: If you wish to scroll rapidly, hold down the Select key on the keyboard while the pointer is over the scroll arrow.

Graphics Window Elevators

In addition to arrows, a graphics window may contain **elevators** in its border. A graphics window's border will contain elevators only if an application has enabled them in the window's border. Like arrows, elevators pan a window's contents. Figure 3-17 shows a window with elevators enabled.

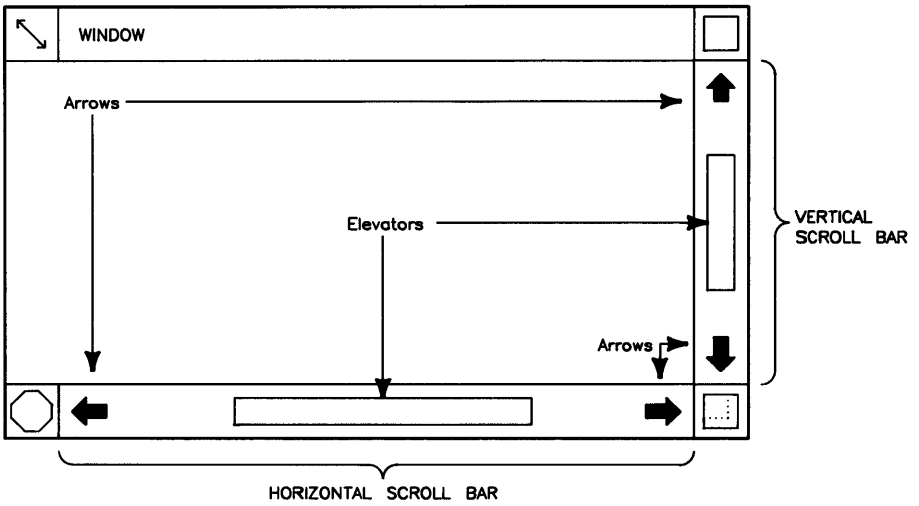

Figure 3-17. Graphics Window with Elevators

To pan a window via elevators, click the select button when the pointer is over an elevator; then move the elevator. As you move it, a dotted box representing the elevator will move within the window's border. To complete the elevator pan operation, click the select button. The window will then pan to the position represented by the new elevator location.

For example, if you move the vertical elevator as high as possible within the window's border, the window will pan to the topmost position within the window's raster.

Elevators may also have an application-dependent function other than scrolling. For example, an application may set up elevators to do some special function, such as making a menu selection.

Elevators are described more thoroughly in the "Arrows and Elevators" chapter in the *HP Windows/9000 Programmer's Manual.*

Things That Can Go Wrong

If the information does not move it could be due to not having the pointer correctly over a scroll arrow. Also, the information will not scroll if there is no more information to scroll in the window's scroll buffer. For example, a graphics window created with its raster the same size as its view is not scrollable in any direction because all the information (the entire graphics picture) is already completely viewable.

The Save Option

You have the option to specify a window to be **saved** or **not saved.**

If you have an application running in one or several windows there are two options that can occur when the application has stopped: the windows created can either stay on the screen or they can be automatically destroyed. Using the *Save* option keeps a window on the screen when all processes of that window are terminated.

You can have windows which are not saved. Windows that are not saved will be automatically destroyed when all processes running in them are terminated.

By default, windows created via the pop-up menu are **not** saved. To change a window's state to *saved*, you should use the *Save* item in the pop-up menu for the window. Therefore, for all windows created via the pop-up menu, the only option is to *save* the window.

Once you have *saved* a window, you cannot make it un-saved via the pop-up menu. That is, changing a window to un-saved is not possible via the pop-up menu; however, it can be un-saved via commands (discussed in the "Using Commands" chapter).

Action

1. **Bring up a pop-up menu for the window you wish to save.**
2. **Highlight the *Save* item.**
3. **Click the select button** to change the window's status to saved.

Things That Can Go Wrong

If the save item is not highlighted when you move the pointer over the item area, the save option is already turned on for this window.

Repainting the Screen

At some point you may be running a program that prints outside of or over a window. The *repaint* operation redraws the screen and restores most windows to their original format. Graphics windows with a retained raster may not be repainted properly.

Action

1. **Bring up a pop-up menu.**
2. **Highlight the *Repaint* option.**
3. **Click the select button** to execute the *Repaint* item. The screen will be repainted.

Things That Can Go Wrong

You may try the Repaint item and notice that nothing changed. This is probably due to having nothing to repaint. If there has been no change of the screen format (over the windows, for example), repaint duplicates the window format (since it is already in correct format).

Notes

Using Commands 4

In addition to HP Windows/9000's interactive capabilities, you can also use window system commands to accomplish windowing tasks. This chapter discusses how to use window system commands; specifically, the following topics are discussed:

- starting the window system
- stopping the window system
- creating a window
- creating a window with a shell
- destroying a window
- changing a window's autodestroy attributes
- selecting a window
- moving a window or icon
- changing a window's size
- shuffling windows
- changing a window's representation (iconic, concealed, or normal)
- controlling a window's border
- managing terminal window fonts
- listing window status information.

Starting the Window System

The *wmstart(1)* command starts the window system running on a bit-mapped display. To start up the window system, simply type the following to the HP-UX shell prompt:

```
wmstart [Return]
```

The window system will then begin running, and a terminal window named *wconsole* will appear in the upper-left corner of the display screen.

The remainder of this section discusses the *wmstart* shell script in detail. If you simply want to start the system as shown above, then the remaining information in this section will be of little use to you. But if you are an application developer, the information in this section may be quite useful.

Concepts

Wmstart is actually a Bourne-shell script which resides in the `/usr/bin` directory. The following concepts are helpful in understanding what *wmstart* does when invoked.

Window System Environment Variables

The window system has a default configuration which determines how it works. For most users, this configuration is quite acceptable and doesn't require change. Nevertheless, some users *do* have special needs that the window system in its default configuration cannot handle.

Window system environment variables provide a way to alter the default window system configuration. These variables, which are initially set in the *wmstart* shell script, define the *environment* in which the window system executes. By altering these variables, you can change how the window system operates. Changing window system environment variables is discussed in the chapter "Environment Variables."

The Window Manager

When the *wmstart* command is executed, it invokes a special server process (`/usr/lib/wm`) known as the **window manager**. The window manager controls the window system. For information on the window manager, see the "Concepts" chapter of the *HP Windows/9000 Programmer's Manual.*

Input Devices

The window system must run on a bit-mapped graphics display, and must use an HP-HIL keyboard for input. You can also use an optional mouse or graphics tablet with the system. For more information on input devices, see the "Concepts" chapter of the *HP Windows/9000 Programmer's Manual.*

The Internal Terminal Emulator (ITE)

When the window system is not running, programs interact with the bit-mapped display and its keyboard through an **internal terminal emulator** (**ITE**). The emulator makes the hardware look like a simple terminal. **`/dev/console`** is a typical path name of the special file (*tty(7)*) for this terminal. The ITE ignores the optional mouse or graphics tablet if they are present; it accepts input only from the keyboard.

Window System Architecture

When the window system starts executing, the window manager takes control of the keyboard from the ITE, thus blocking input to the ITE (but not output from it). In addition, the window manager starts listening to the optional mouse or graphics tablet through their special files.

The window manager determines which device special files to use for input and output by looking at window system environment variables. The variables used and their default values are defined in Table 4-1.

Table 4-1. Environment Variables and Special Files.

Variable	Description	Default
WMDIR	Directory where window special files are maintained by the window manager.	**`/dev/screen`**
WMSCRN	Special file of the display device where windows will appear.	**`/dev/crt`**
WMKBD	Special file for the HP-HIL keyboard.	**`/dev/hilkbd`**
WMINPUTCTLR	Special file of the input controller which handles HP-HIL input devices.	**`/dev/rhil`**
WMLOCATOR	Special file for the optional locator device—either a mouse or graphics tablet, but *not both* at the same time.	**`/dev/locator`**

Executing wmstart(1)

Depending on how your system is configured, the window system may automatically start up when you log in. If the system does not automatically start up, then you must start it yourself by executing the *wmstart* command from the HP-UX shell.

Default Action

Before discussing the syntax of *wmstart*, you must understand the default actions taken when it is executed. The action of *wmstart* is summarized as follows; you may wish to refer to the *wmstart* shell script (found in `/usr/bin`) when reading this:

1. **Set environment variables to their default values.** If a variable is undefined, then it defaults to a value predetermined by *wmstart* or the window manager itself.
2. **Check if the window manager is running.** If the window manager is already running, then terminate with exit status 1; otherwise, continue execution.
3. **Remove any leftover special files in the WMDIR directory.** The window system uses a number of special files to communicate with windows. These special files are kept in the directory specified by the WMDIR environment variable. If the window system terminates abnormally for some reason, window special files might be left over in this directory. *wmstart* ensures the proper execution of the window system by removing all character special files in the WMDIR directory before starting the window manager.

IMPORTANT

The window manager removes **all character special files** in the directory specified by WMDIR. Therefore you should never change WMDIR to the path name of a directory containing non-window system special files (such as the `/dev` directory). For details on changing WMDIR, see the chapter "Environment Variables."

---

4. **Start the window manager.** The window manager is executed via the *sh(1)* special command *exec* as follows:

   `exec /usr/lib/wm` *command_line*

   This way, the caller of *wmstart* can wait for the *wmstart* process to terminate as the window manager.

5. **Execute a window command.** The window manager process (`/usr/lib/wm`) can receive a command line as an argument. By default, the argument supplied to *wm* when it is invoked in step 4 is:

   ```
   /usr/bin/wsh -ak wconsole
   ```

   which creates a terminal window named *wconsole* as the first window in the system. However, you can execute a different command, perhaps a window application of your own, as described in the "Syntax" section below.

When these steps are successfully completed, the window manager will take control of window system input/output devices, the screen desk top pattern will be displayed, a window named *wconsole* will appear in the upper-left corner of the display screen, and a shell from which you can execute commands is spawned in the window.

The SHELL Environment Variable

The type of shell used in the initial *wconsole* window, and in any windows created via the pop-up menu or *wsh(1)* command, depends on the value of the SHELL environment variable when *wmstart* is called. If $SHELL is `/bin/sh`, then a Bourne shell is used; if $SHELL is `/bin/csh`, then the C-shell is used.

Syntax

To start HP Windows/9000, execute the *wmstart* command which has the following syntax:

`wmstart` [*optional_args*]

If you want *wmstart* to execute in the default manner, as defined in steps 1 through 5 above, then simply enter the command to the HP-UX prompt. For example, if you simply wish to start the window system, enter the following to the HP-UX prompt:

`wmstart` Return

You can alter the default action of *wmstart* by specifing the *optional_args* with the command. If you specify *optional_args*, they are passed to *wm* instead of the default command described in step 5. For example, suppose you have a customized window application that you want to execute, without having the *wconsole* window come up first; the name of your application is *window_sys*, and it is found in the `/usr/contrib/bin` directory; then you would enter the following:

`wmstart /usr/contrib/bin/window_sys` Return

The wmready Command

Occasionally, you may wish to determine if the window manager is running before you attempt to execute the *wmstart* command. The *wmready(1)* command is used for this purpose.

As an example of how you might use this command, suppose you have a multi-user system with one graphics display devoted to HP Windows/9000. You're seated away from the graphics display and cannot see if anyone is using the system. You can use the *wmready* command to determine if the window system is already in use.

The *wmready* command determines if the window manager is running by looking at the value of the WMDIR environment variable. For example, if WMDIR is set to `/dev/screen`, *wmready* will look in this directory for the window manager's device interface (`/dev/screen/wm`). If the device interface exists, then *wmready* verifies that there is an active window manager associated with the special file. If there is, then the window manager is running; otherwise the window manager is not running.

Note

Because *wmready* is typically executed outside the window system (e.g., from a non-window system terminal), and because *wmready* requires the value of the WMDIR environment variable, you may want to set WMDIR to the appropriate value before executing this command, for example:

```
env WMDIR=/dev/screen wmready
```

See the chapter "Environment Variables" for details on setting environment variables.

Syntax

wmready has the following syntax:

```
wmready [ -v ]
```

Return Value

When *wmready* is executed, it returns a value indicating whether or not the window manager is running. If **1** is returned, the window manager is not currently running; if **0** is returned, the window system is in use. The method for getting this return value depends on which shell you use.

If you use the Bourne shell, you can interrogate the **$?** shell parameter, which contains the value returned by the last synchronously executed command.

If you use the C-shell, interrogate the **$status** environment variable, which contains the status returned by the last command.

Examples

The following Bourne shell script displays a message indicative of whether or not the window system is currently running:

```
env WMDIR=/dev/screen wmready
if [ $? -eq 1 ]
then
     echo "window system is free to use"
else
     echo "window system is already in use"
fi
```

The next C-shell script performs the same function as the above Bourne shell script:

```
#
# determine if window manager is already in use
#
env WMDIR=/dev/screen wmready
if ($status == 1) then
     echo "window system is free for use"
else
     echo "window system is already in use"
endif
```

The -v Option

Rather than having to interrogate the status returned from *wmready*, you can use the verbose (**-v**) command option. If you specify **-v** on the command line:

```
env WMDIR=/dev/screen wmready -v
```

one of two messages will be displayed.

If the window manager is *not* running, the following message is displayed:

```
Window manager (/dev/screen/wm) is not ready.
```

This simply means that the window manager process is not running and, therefore, cannot accept any requests that would be made if it were running.

If the window manager is running, this message is displayed:

```
Window manager (/dev/screen/wm) is ready.
```

This means that the window manager is running, and the window manager process is named `/dev/screen/wm`. The basename of the window manager process will always be *wm*, but the rest of the path name (`/dev/screen` in this case) will depend on the value of the WMDIR environment variable.

Automatically Starting Windows/9000 from Login

Depending on your needs, you may wish for the window system to automatically start up whenever you log in. This section describes various methods for automatically starting HP Windows/9000 when logging in to your system.

Note

The topics discussed here are probably more advanced than most users require. You should consult your system administrator for help in performing any of the described tasks.

There are two primary methods for starting the window system on login:

- you can execute *wmstart* from your *.profile* or *.login* initialization script
- you can make *wmstart* your login shell.

Other more-obscure methods can be used, but they are not discussed here. Ask your system administrator for more information on automatically starting the window system.

From .profile or .login

Two initialization shell scripts are associated with HP-UX: *.profile* and *.login*. Both are kept in your home directory. When you log in to your computer system, commands from these files are executed, depending on which shell you use as your login shell.

If you log in to the Bourne shell, commands in the *.profile* initialization script are executed. If you use the C-shell, commands in the *.login* script are executed. Therefore, if you want to immediately start up the window system after logging in, you should add the *wmstart* command to your *.profile* or *.login* file (depending on which is your login shell). Ask your system administrator if you are unsure of which login shell you use.

Running as a Subprocess vs. Executing Directly

Two methods can be used to execute *wmstart* from your *.profile* or *.login* shell script; the results of each method differ:

- You can run the command as a subprocess:

  ```
  /usr/bin/wmstart
  ```

  The *.profile* or *.login* script waits for window manager termination. When the window system terminates, you are returned to an HP-UX shell from which you can manually start the window system again.

- You can execute the command directly via *exec*:

  ```
  exec /usr/bin/wmstart
  ```

  In this case, the *.profile* or *.login* script is replaced by the window manager process. When the window system terminates, you'll be logged out of the system.

Executing wmstart as Your Login Shell

To put *wmstart* in `/etc/passwd` as your login shell, the following must be done:

1. Make a custom version of *wmstart*, and name it something that does **not** contain the letter **r**, for example, *my_wmgo*. This must be done because if the login program sees a login shell containing the letter **r**, it assumes that it is a restricted shell.
2. In the custom version, explicitly set SHELL to the appropriate value. If you want to use the Bourne shell:

   ```
   SHELL="/bin/sh" ;export SHELL
   ```

   If you wish to use the C-shell:

   ```
   SHELL="/bin/csh" ;export SHELL
   ```

3. Have your system administrator put the name of the custom *wmstart* script in `/etc/passwd`. Or you can use the *chsh(1)* command to change it yourself.

Stopping the Window System

The *wmstop(1)* command stops the window system. To stop the window system, simply type the following to the HP-UX prompt:

```
wmstop Return
```

The window system will then stop running. And all processes started while in windows will also stop running (with a few exceptions, discussed below). Therefore, be absolutely sure that you want to exit the window system before executing this command.

If you are a novice user and don't really want to know intricate detail about what happens when the window system stops running, then you needn't read any more of this section. If, however, you do require detailed knowledge about the window system—e.g., for developing applications—then you should read the remainder of this section.

Concepts

Wmstop stops the window system for one display, normally the display from which it was invoked. It may be called by any process.

Wmstop looks at the environment variable WMDIR to find the window manager process's special file ($WMDIR/`wm`). It then uses the *wmkill(3W)* window library routine to kill the window manager. This causes the window manager to terminate gracefully, destroying all windows and clearing the screen.

When the window manager terminates, control of keyboard input is returned to the ITE.

Precautions

Executing *wmstop* normally causes **all** processes in the window group to terminate, gracefully or not. (For details on the *window group*, see the "Concepts" chapter of the *HP Windows/9000 Programmers' Manual.*)

In some cases, processes started from the window system will *not* terminate when the window system exits (for example, processes started with *nohup(1)*, and background processes). Output from these processes may be lost or may overwrite portions of the screen asynchronously, unless it was redirected away from a window.

Creating a Window

Once the window system is running, you can create new windows via the *wcreate(1)* command. This section discusses the use of the *wcreate* command and its various parameters.

To simply create a term0 window, type the following to the HP-UX prompt from a term0 window:

wcreate *window_name* Return

where *window_name* is the name of the window to create. *Wcreate* will then create and display a default term0 window named *window_name.*

To create a graphics window, type the following:

wcreate -w graphics *window_name* Return

where *window_name* is the name of the graphics window to create. The "**-w graphics**" option tells *wcreate* to create a graphics window.

The *wcreate* command has many other options for creating windows. You should read the remainder of this section if you wish to use the special features of *wcreate.*

Concepts

Before discussing *wcreate*, you should understand some basic concepts concerning windows.

Window Shell

Windows created via *wcreate* do **not** automatically contain an HP-UX shell. Creating a window that contains an HP-UX shell is discussed in the next section "Creating a Window with a Shell."

Window Location

The *wcreate* command allows you to specify the screen location for each new window's **anchor point**. A window's anchor point is the upper-leftmost pixel in the window's user (contents) area (see Figure 4-2).

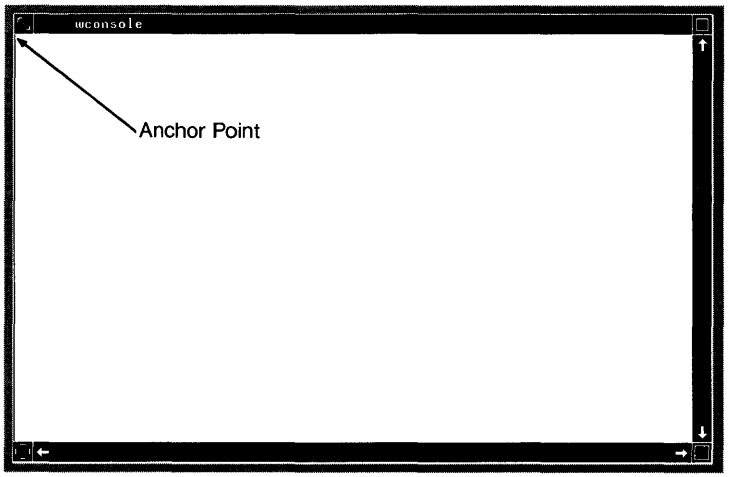

Figure 4-2. The Anchor Point

Coordinates are specified in *x,y* pixels. The upper-leftmost pixel on the display screen is location *0,0*; *x* coordinates increase to the right; *y* coordinates increase downward (see Figure 4-3).

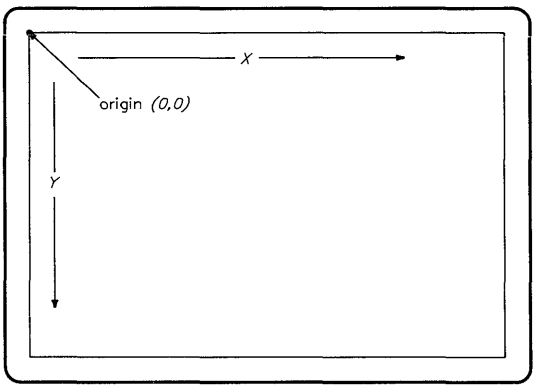

Figure 4-3. Display Screen Pixel Coordinates

Maximum *x,y* coordinates depend on the type of display screen used with the system. For example, if the resolution of your display screen is 1024 by 768 pixels, then maximum *x,y* coordinates are *1023,767*. See the "Display Screen Coordinates" section of the "Concepts" chapter for details on display screen coordinates. If you do not specify a new window location, the window is placed at default, stair-step coordinates returned by the window manager.

Window Size

You can also specify a window's **size** when it is created. Terminal window size is specified in *columns* and *rows* (known as the **logical screen size**); graphics window size, in pixel *width* and *height*. A window is initially displayed at the specified size.

If you do not specify a window size, then a default window size is assigned to the window. Terminal windows default to 80 columns by 24 rows; graphics windows default to 200 by 200 pixels.

Raster/Buffer Size

Closely related to window size is **raster** and **buffer** size. Raster size refers to graphics windows, and buffer size refers to terminal windows.

A graphics window's raster size is the size of the virtual graphics display being emulated by the window. In other words, it is the pixel width and height of the graphics display the window emulates. A window's size must always be less than or equal to its raster size. Any graphics performed in a graphics window will be performed in the entire raster, not just the visible portion given by the window's size. If you do not specify the raster size, it defaults to the window size.

Each terminal window has a scroll buffer. This buffer holds information that scrolls out of the user area. Buffer size specifies the size of this screen buffer for the given window. If you do not specify a buffer size, it defaults to 80 columns by 48 rows (two full window user areas of text) or to the window size, whichever is larger.

Figure 4-4 illustrates the relationship between a graphics window's raster and window size, and a terminal window's buffer and logical screen size.

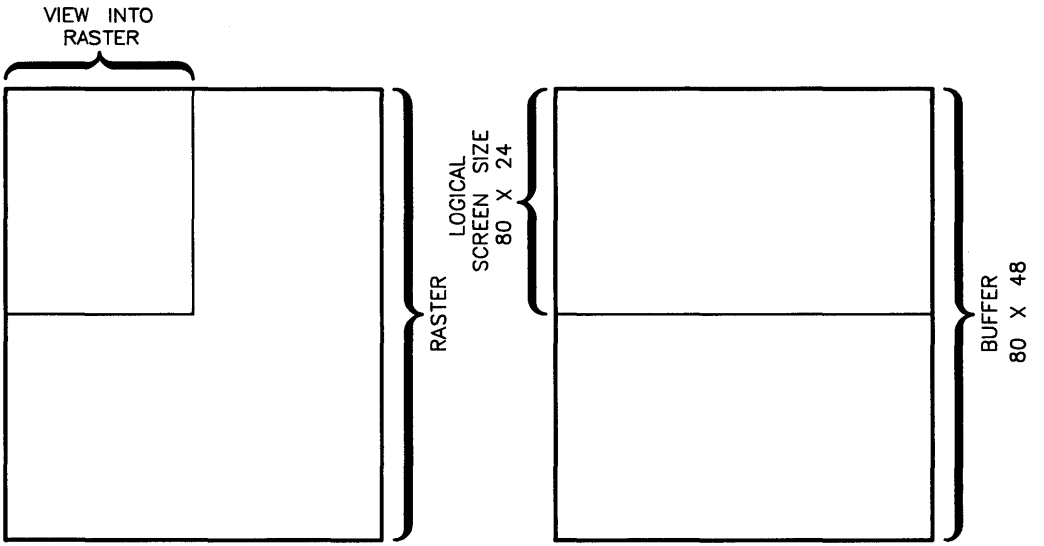

Figure 4-4. Relationship between Raster/Buffer and Window Size

Maximum Window Size

Once a window is created, you can shrink and increase its size. The maximum size for a graphics window is its raster size; the maximum size for a terminal window is its logical screen size (the size at which it was created).

Retained Graphics Window Raster

By default when a graphics window is created, it has a **retained raster**. This means that memory is allocated for the window. The memory may be allocated **one byte-per-pixel** (the default for the *wcreate* command) or **one bit-per-pixel** depending on the parameter value passed to `wcreate_graphics`, or the options used on the *wcreate (1)* or *wsh (1)* commands. The **byte-per-pixel** retained memory driver is called the "byte driver" and the **bit-per-pixel** retained memory driver is called the "bit driver". The benefit of a retained raster is that any graphics performed to the window, when it is non-viewable, are preserved in the retained memory; the window manager takes care of maintaining the window's contents when the screen is updated.

The disadvantage of retained rasters is that they consume shared memory. For example, a 1024- by 512-pixel **byte-per-pixel** retained-raster window uses half a megabyte of shared memory. The same retained-raster window that is allocating memory in a **bit-per-pixel** format uses 65 536 bytes of shared memory. (See the appendix "Window Limitations" for details on shared memory.)

Important

Use the bit driver instead of the byte driver when using windows on a monochrome display where memory is constrained.

The 3001 Device Driver does not support **bit-per-pixel** retained rasters.

Both the bit driver and the byte driver provide retained raster support for graphics windows. However, the byte driver allocates one byte per pixel while the bit driver allocates one bit per pixel. Thus, the byte driver can use up to eight bits of memory to contain the color information for each pixel. On monochrome displays only one bit of each byte is used.

Because the bit driver uses a bit per pixel format, eight times less memory is used when monochrome images are stored using this driver. Only monochrome images are stored using the bit driver.

An example of the above information is:

The byte driver will allocate 199 680 bytes of memory to support a raster that is 512 by 390 pixels on a monochrome display.

The bit driver will only allocate 24 960 bytes to support the retained raster for the same display.

The *wcreate* command also allows you to create windows with non-retained rasters. The user area of non-retained windows cannot be redrawn from memory; therefore, windows with non-retained rasters have the potential to become mussed. *You* must ensure that the window's user area remains accurate.

This page intentionally left blank.

For example, if the window system screen is repainted, the window manager cannot repaint non-retained graphics windows from memory; you must catch the repaint signal and repaint the window yourself.

The advantage of non-retained rasters is that they don't consume shared memory. They are also especially useful for graphics programs that mantain a vector list from which the window can be easily redrawn when the screen needs to be updated.

Window Type Device Interface

Each window created has a corresponding special file through which communication with the window is possible. These special files are known as **window type device interfaces** and are stored in the directory specified by the WMDIR environment variable. The path name for each special file is $WMDIR/`window_name`, where *window_name* is the name used when the window is created.

Executing wcreate(1)

To create a terminal or graphics window, execute the *wcreate* command which has the following syntax:

`wcreate` [`-w` *type*] [`-kboitTMmNnv`] [`-l` *x,y*] [`-s` *w,h*] [`-r` *w,h*] [*window_spec...*]

Optional parameters are shown in brackets []. Descriptions of each parameter follow.

Specifying Window Name (window_spec...)

If you do not specify a window name, a default window name will be taken from the window manager. Default window names are assigned sequentially, and have the format:

`window`*n*

where *n* is a sequential number starting at one. For example, the first window created after *wconsole* is *window1*; the second, *window2*; and so on.

To specify a window name other than the default, give the window name as the last parameter (*window_spec*). You can create more than one window by giving more than one name. For example:

```
wcreate win1 win2 my_win
```

creates three windows: *win1*, *win2*, and *my_win*.

For details on different ways of specifying the *window_spec*, see the *windows(1)* page in the *HP Windows/9000 Reference*.

Specifying Window Type (-w)

The `-w` parameter is used to specify the type of the window to create. Recognized values for the type are: **`graphics`**, **`term0`** (for terminal windows), and **`see_thru`**. When this parameter is omitted, a terminal window is created by default.

The space between `-w` and the window type is optional.

The following creates a graphics window named *my_grwin*:

```
wcreate -wgraphics my_grwin
```

The next example creates a terminal window with a default name:

```
wcreate -w term0
```

You can leave off the `-w` parameter since term0 is the default window type.

Selecting the Window (-k)

To automatically select a window upon creation, use the `-k` option, which attaches the keyboard to the newly created window. If you create more than one window, the keyboard is attached to the last window specified on the command line.

The following creates a graphics window named *wqix* and selects it:

```
wcreate -wgraphics -k wqix
```

The next example creates three terminal windows—*win1*, *win2*, and *win3*—and attaches the keyboard to *win3*:

```
wcreate -k win2 win1 win3
```

Placing the Window on Bottom (-b)

By default, new windows are placed on the top of the displayed stack of windows. If you want a window to be placed on the bottom, use the `-b` option.

The following creates a terminal window named *bottom_win* on the bottom of the stack and attaches the keyboard to it:

```
wcreate -kb bottom_win
```

Note that only one of the `-b` and `-o` options can be specified at a time; attempting to give both will result in an error.

Concealing a Window (-o)

By default, new windows are displayed in their normal form. You can cause a window to be concealed (not displayed on the screen) by using the `-o` option.

Once a window is concealed, you can make it visible using the *wdisp(1)* command. (See the section "Changing a Window's Representation" for details on using *wdisp*.)

The following creates a default-named graphics window, but conceals it:

```
wcreate -wgraphics -o
```

Note that only one of the `-b` and `-o` options can be specified at a time; attempting to give both will result in an error.

Making the Window Iconic (-i)

Normally a new window is displayed in its normal form. The `-i` option is used to make the window an icon initially.

Note that this option *can* be used with the `-o` option. Using them together causes the window to be a concealed icon. Then when the window is displayed, using *wdisp(1)*, it is displayed as an icon.

The following creates an iconic terminal window named *splork*:

```
wcreate -i splork
```

Thin Border (-t)

To give a window a thin border, as described in the "Concepts" above, use the `-t` option. If `-t` is not specified, the window will have a normal border.

The following creates a terminal window named *thin_border*; the keyboard is attached to it:

```
wcreate -kt thin_border
```

No Border (-T)

To create a window with no border, use the `-T` option:

```
wcreate -T no_border
```

Retained Graphics Window Raster (-M or -m)

If neither `-M`, `-m`, `-n` or `-N` are given, the graphics window raster defaults to retained byte/pixel (`-M`). The `-m` option specifies retained bit/pixel.

```
wcreate -wgraphics -m bit_pixel
wcreate -wgraphics byte_pixel
```

IMAGE Graphics Window (-N)

This is a non-retained graphics window with the user area displayed (mapped) in the image planes and the border area corresponding to the user area cleared to the see-thru color index. This allows accelerated 3D graphics in a window, since the functionality of the image planes is required to perform 3D graphics.

```
wcreate -wgraphics -N image_window
```

Non-Retained Graphics Window Raster (-n)

All graphics windows, by default, have a retained raster, as described in "Concepts" above. To create a graphics window with a non-retained raster, use the `-n` option.

The following creates a non-retained graphics window named *no_retain*; the window is created to the default size (i.e., no window or raster size is specified):

```
wcreate -n no_retain
```

Verbose Mode (-v)

If you would like *wcreate* to display the path name of the window's device interface when the window is created, use the `-v` option.

Verbose mode is useful when you don't give *window_spec*—when you allow the window manager to create a name for you. You can capture new window names in a shell variable. For example, if you are a Bourne shell user:

```
win_path=`wcreate -vw graphics`
win_name=`basename "$win_path"`
```

creates a graphics window with the default window manager name; the path name of the window's device interface is stored in `win_path`; and the window's name is stored in `win_name`.

Specifying Location (-l)

The window's new location is specified using the `-l` option. Coordinates are specified in *x,y* pixels. If no coordinates are given, the window is placed at default coordinates taken from the window manager. (For details on screen coordinates, see the "Display Screen Coordinates" section of the "Concepts" chapter in this manual.)

The following creates a graphics window, with default size and a thin border, at location 100,150:

```
wcreate -w graphics -t -l 100,150
```

The space between `-l` and the *x,y* coordinates is optional.

If you specify coordinates **and** create more than one window, all windows will be placed at the same location.

Specifying Size (-s)

A window's size is specified with the `-s` option. If this option is omitted, terminal windows default to 80 columns by 24 rows, and graphics windows default to 200 by 200 pixels.

The following creates a non-retained graphics window named *my_gr*; the window is created 400 pixels wide by 200 pixels high:

```
wcreate -wgraphics -n -s 400,200 my_gr
```

The space between the `-s` and the width and height can be omitted. The next example creates a terminal window that is 80 columns by 48 rows:

```
wcreate -s80,48
```

This page intentionally left blank.

Specifying Raster/Buffer Size (-r)

A window's raster or buffer size is specified via the `-r` option. If no raster size is specified (for graphics windows), the raster defaults to the window size. If no buffer size is specified (for terminal windows), the scroll buffer defaults to 80 columns by 48 rows of characters (two default window screens of information) or to the window size, whichever is larger.

The following creates a graphics window named *gr_win*; its size is 200 by 200 pixels, but its raster size is 800 by 400 pixels; the raster is retained; and the window has a thin border:

```
wcreate -w graphics -t -l100,100 -s200,200 -r 800,400
```

The space between the `-r` and the raster width and height is optional. The next example creates a terminal window named *four_screens*; it is created to the default columns and rows (80 by 24); but its scroll buffer can hold up to four screens (80 columns by 96 rows) of information:

```
wcreate -r80,96 four_screens
```

Creating a Window with a Shell

The *wsh(1)* command creates a terminal window containing an HP-UX shell; it can also put a shell in an existing terminal window. To simply create a terminal window with a shell, type the following to the HP-UX prompt from a terminal window:

wsh *window_name* Return

where *window_name* is the name of the window to create. *Wsh* will then create a default terminal window named *window_name.*

The remainder of this section discusses *wsh* and its parameters in detail. Essential concepts are borrowed from the previous "Creating a Window" section. You should be sure to read that section before continuing with this one.

Concepts

Before discussing how to create a window containing a shell, you should understand the following essential concepts.

The SHELL Environment Variable

The window system uses the SHELL environment variable to determine which shell to put in a window. The SHELL variable is, by default, set to the path name of the shell you use.

The SHELL variable, by default, is set to the path name of your login shell as defined in **/etc/passwd**. For example, if you use the Bourne shell, SHELL is set to **/bin/sh**; if you use the C-shell, SHELL is set to **/bin/csh**.

You can determine the value of SHELL by using the *echo(1)* command. Type the following from HP-UX, and HP-UX will display the value of SHELL:

```
echo $SHELL
```

When a new window is created with a shell, the window system looks at SHELL to determine which shell to put in the window.

Setting SHELL

For most users, SHELL is automatically set when they log in or power up their system. However, some users may wish to use a different shell than the default. To change SHELL, you should set it in your personal `.profile` or `.login` initialization script.

For example, if you want the Bourne shell in your windows, you should put the following in your *.profile* shell script:

```
SHELL=/bin/sh ; export SHELL
```

and the following in your *.login* script:

```
setenv SHELL /bin/sh
```

Inherited Environment

All windows created via the pop-up menu inherit their run-time environment from the existing environment when *wmstart(1)* is invoked. For example, if *wmstart* is executed from the directory `/usr/lib/hpwindows/demo`, all windows created will have their current working directory initially set to the same.

Unlike windows created using the pop-up menu, windows created via *wsh(1)* inherit the environment that existed when *wsh* was executed, which may be different from the environment that existed when *wmstart* was executed.

Terminating a Window Shell

A shell in a window can be terminated in the same manner as a shell at a terminal. Simply execute the appropriate command (e.g., *exit* for the Bourne shell; *logout* for the C-shell).

Once you've terminated a shell in a window, you cannot execute any more commands from the window. Depending on the options used when the window was created, it may automatically disappear when the shell is terminated (discussed next in "Automatic Window Destruction").

A window that is *not* automatically destroyed when its shell is terminated is in the same state as a terminal window created via *wcreate(1)*—it is simply a terminal window with no programs running in it.

Automatic Window Destruction

By default, when all the processes in a window (including the shell) terminate and the window's device interface (special file) is closed by all processes, the window remains intact in the system until you *explicitly* destroy it via the pop-up menu's *Destroy* option or the *wdestroy(1)* command.

By using special command options with *wsh*, you can cause the window to be automatically destroyed when its device interface is closed by all processes. For example, you can cause the window to be destroyed when its shell terminates.

A window that is marked to be automatically destroyed is said to be **recoverable**.

You can also control *when* the window is destroyed:

- It can be destroyed immediately when its device interface is closed by all processes that had it open. In window terminology, a window in this state is recoverable and **autodestroyable**.
- It can be destroyed subsequently when a new window is created, either via the pop-up menu or commands. In window terminolgy, the window is recoverable, but **not** autodestroyable.

The `-a` and `-d` options are used for this purpose; they are described below.

Executing wsh(1) to Create a Window

As mentioned earlier, *wsh* can be used to create a window containing a shell, or it can be used to attach a shell to an existing window. Using *wsh* to create a window with a shell is discussed here.

Syntax

When used to create a window containing a shell, the *wsh* command has the following syntax:

`wsh` [`-w` *type*] [`-kboitTMmNnv`] [`-l` *x,y*] [`-s` *w,h*] [`-r` *w,h*] [`-gad`] [`-c` *cmd*] [*window_spec...*]

wcreate(1) Options

All of the options avaible for the *wcreate* command are also available for *wsh*. The meaning of these options also remains the same for *wsh*. Table 4-2 summarizes the common options between *wsh* and *wcreate*.

For details on these options, see "Executing wcreate(1)" in the previous "Creating a Window" section.

Table 4-2. Common Options between wsh(1) and wcreate(1).

Option	Summary
window_spec	Specifies the name(s) of the window(s) to create. If you do not give a window specification, a default name is assigned by the system to the new window. For details on specifying *window_spec*, see the *windows(1)* reference page in the *HP Windows/9000 Reference.*
`-w` *type*	Gives the type—**graphics** or **term0** for the window to create. Normally you would just omit this parameter, as it defaults to terminal type (term0). However, if you need to execute a graphics application from a shell, set the window type to **graphics** (see the example in "Destroy Upon Close (-a)" below).
`-k`	If present, it means to select (attach the keyboard to) the window after it is created.
`-b`	Says to make the window the bottom window in the display stack. You cannot specify both `-b` and `-o` at the same time.
`-o`	Conceal the window. The window is invisible. Only one of `-b` and `-o` can be specified at a time.
`-i`	Make the window iconic.
`-t`	Gives the window a thin border.
`-T`	Give the window a null border—no border.
`-M`	Creates a byte-per-pixel retained memory raster. (Default)
`-m`	Creates a bit-per-pixel retained memory raster.
`-N`	Creates an IMAGE graphics window.
`-n`	If the window is a graphics window, give it a non-retained raster.
`-v`	Verbose mode. Display the path name of the window's device interface when the window is created.
`-l` *x,y*	Gives the window's *x,y*-pixel location. If not specified, it defaults to a system-determined stair-step location.
`-s` *w,h*	For a terminal window, this gives the number of *columns* and *rows* of characters for the window; if omitted, window size defaults to 80 columns by 24 rows. For graphics windows, this option gives the pixel width and height of the window; if not specified, it defaults to 200 by 200 pixels.
`-r` *w,h*	For a terminal window, this gives the size of the scroll buffer; if omitted, the scroll buffer default to 80 columns by 48 rows (enough for two default-sized window screens of information). For graphics windows, this gives the width and height (in pixels) of the virtual raster; if not specified, it defaults to the window's size.

Passing a Command (-c)

Occasionally, you may wish to create a window for nothing but the purpose of executing a command or application in the window. The `-c` option allows you to start a command or application in a window, **without ever getting an interactive HP-UX shell in the window.**

For example, the following creates a terminal window named *vi_window* for nothing but the purpose of editing a file named *flebnee*:

```
wsh -c'vi flebnee' vi_window
```

In this example, when you exit from *vi(1)*, you'll have a dead window, a window containing no shell or application. You must explicitly destroy the window. However, it is possible to have the window automatically destroyed when you're finished with it; this is described later in "Destroying upon Close (-a)" and "Destroying upon Next Create (-d)."

Making a Login Shell (-g)

When you create a new window with a shell, you may want the shell initialization scripts to be executed, just as if you had logged into the window. For example, if you're creating a Bourne-shell window, you may want `/etc/profile` and $HOME/`.profile` executed when the shell is created in the window. And if you create a C-shell window, you may want the `/etc/csh.login`, $HOME/`.cshrc`, and $HOME/`.login` scripts executed. The `-g` option causes the login initialization sequence to be performed when the shell is created in the window.

Destroying upon Close (-a)

If you want a window to be automatically destroyed when its shell (or application) terminates, use the `-a` option. If `-a` is specified, the window is immediately destroyed when all commands or applications executing in the window close the window's device interface.

For example, suppose you create a window for the sole purpose of editing a file named `stuff.dat`. When you are through editing the file, you want the window to disappear. The following performs this task:

```
wsh -kga -c'vi stuff.dat' vi_window
```

Note that the keyboard is attached to the window (`-k`), and login scripts are read (`-g`).

For the next example, suppose you have a graphics application named *graph_master* that requires a graphics screen that is 512 pixels wide by 512 pixels high. If you wish to create a graphics window that simply executes *graph_master* and terminates when it is finished, use:

```
wsh -w graphics -ka -r 512,512 -c graph_master
```

Destroying upon Next Create (-d)

The `-d` option is similar to the `-a` option, except that the window is not destroyed until a new window is created. When a new window is created, the window that is marked with the `-d` option will be automatically destroyed.

The following example lists the contents of the current directory in a window named *ls_window*. The window will continue to exist until you create another window or destroy the window explicitly (via the pop-up menu or *wdestroy(1)* command:

```
wsh -d -cls ls_window
```

Executing wsh(1) to Start a Shell

A **dead** window is a window whose shell or application has terminated, but which has not yet been destroyed. The window is basically inactive: anything you type at the window is ignored. This section discusses how to start a shell in a dead window.

Syntax

When used to start a shell in a dead window, *wsh* has the following syntax:

`wsh -e [-gad]` [`-c` *commandline*] window_spec...

Descriptions of each parameter follow.

Specifying the Window (window_spec...)

Whereas giving the window specification is optional when creating a window, the window specification *must* be given when using *wsh* to start a shell in a dead window. Otherwise, *wsh* does not know which window to start the shell in.

Start a Shell (-e)

The `-e` option tells *wsh* to start a shell in the specified window. The `-e` option should not be used when creating a window.

Suppose you have a dead window named *dead-un*. To start a shell in the window, you would use:

```
wsh -e dead-un
```

The -gadc Options

The `-g`, `-a`, `-d`, and `-c` options can also be used with *wsh* in this case. They work the same as described previously.

Destroying a Window

When you are finished using a window, you can destroy it using either the *Destroy* option of the pop-up menu, or the *wdestroy(1)* command.

Wdestroy can also be used to set a window's auto-destruction status, as described in the next section, "Setting a Window's Autodestroy Attributes."

Executing wdestroy(1)

When *wdestroy* is used simply to destroy a window, its syntax is:

```
wdestroy window_spec...
```

You can destroy more than one window by giving the window name of each window to destroy. To destroy the window attached to standard input (i.e., the window from which *wdestroy* is executed), use - for the *window_spec* (see the third examle below).

Examples

The following destroys the window named *mywindow*:

```
wdestroy mywindow
```

The next example destroyes three windows—*win*, *mywin*, and *gerschwin*:

```
wdestroy win mywin gerschwin
```

To destroy the window attached to standard input, use:

```
wdestroy -
```

Precautions

- Destroying a window completely removes it from the window system. Any programs executing in the window cannot be retrieved. You should be certain you want to destroy a window before using this command.
- When a window is destroyed, all of its *pty* special files are removed from the $WMDIR directory. They then become available in the pool of *ptys* to create new windows.

Setting a Window's Autodestroy Attributes

In addition to destroying windows, the *wdestroy(1)* command can be used to mark an existing window to be automatically destroyed when the window's shell (or application) terminates.

Concepts

By default, when all the processes in a window (including the shell) terminate and the window's device interface (special file) is closed by all processes, the window remains intact in the system until you *explicitly* destroy it via the pop-up menu's *Destroy* option or the *wdestroy(1)* command.

By using special command options with *wdestroy*, you can cause the window to be automatically destroyed when its device interface is closed by all processes. For example, you can cause the window to be destroyed when its shell terminates.

A window that is marked to be automatically destroyed is said to be **recoverable**.

You can also control *when* the window is destroyed:

- It can be destroyed immediately when its device interface is closed by all processes that had it open. In window terminology, a window in this state is recoverable and **autodestroyable**.
- It can be destroyed subsequently when a new window is created, either via the pop-up menu or commands. In window terminology, the window is recoverable, but **not** autodestroyable.

The `-a` and `-d` options are used for this purpose; they are described below.

Executing wdestroy(1)

When used to set a window's autodestroy attributes, *wdestroy* has the following syntax:

```
wdestroy -adn [window_spec...]
```

Only one of the `-a`, `-d`, or `-n` options may be used at a time, and the *window_spec* is optional. You can set the autodestroy status for more than one window by giving the name of each window for *window_spec*. If no *window_spec* is given, *wdestroy* destroys the window connected to standard input (typically, the window from which *wdestroy* was executed).

Destroy Upon Close (-a)

If you want a window to be automatically destroyed when its shell terminates, use the `-a` option. If `-a` is specified, the window is immediately destroyed when all commands or applications executing in the window close the window's device interface.

For example, suppose you have terminal window named *term0win* which contains a shell; the window was created in the following manner, using *wsh(1)*:

```
wsh -k term0win
```

Because the `-a` option was not used when the window was created, the window will not be automatically destroyed when the shell terminates. To change this—i.e., to automatically destroy the window immediately when its shell terminates—use *wdestroy* as follows:

```
wdestroy -a term0win
```

Destroy upon Next Create (-d)

The `-d` option is similar to the `-a` option, except that the window is not destroyed until a new window is created. When a new window is created, the window that is marked by the `-d` option will be automatically destroyed.

Suppose that in the previous example, you want the window to be automatically destroyed *when* a new window is created. You would use:

```
wdestroy -d term0win
```

Turn off Autodestroy (-n)

You can turn auto-destruction off via the `-n` option. Using `-n` tells the window system to *not* automatically destroy the window when its shell (or application) terminates.

For this example, suppose you've created a terminal window named *flebnee*, and it was created with auto-destruction turned on (`-a`):

```
wsh -a flebnee
```

To turn auto-destruction off for the window, you would use:

```
wsh -n flebnee
```

Selecting a Window

In addition to using the pop-up menu to select a window, you can use the *wselect(1)* command.

Concepts

As mentioned in the "Concepts" chapter, keyboard input can be read from a window only when the window is selected. That is, only when a window is selected can processes read keyboard (and locator information) from the window's device interface.

For example, if you have a shell running in a particular window, you cannot enter HP-UX commands in the window until the window is selected.

Executing wselect(1)

To select a window, use *wselect(1)* which has the following syntax:

wselect [*window_spec*]

The window specified by *window_spec* will become selected. If no *window_spec* is given, then the window attached to standard input (typically, the window from which the command was executed) is selected.

Executing *wselect* without the *window_spec* parameter would typically be used from a script. The window in which the script is running might not be the selected window, but when the *wselect* command is executed in the script, it becomes the selected window.

Example

Suppose you create a new window, but you forget to attach the keyboard to the window when it is created:

```
wsh vi_window
```

To select the window via the *wselect* command, you would use:

```
wselect vi_window
```

After which you can begin using *vi(1)* within the window.

Precautions

Remember that keyboard input can be taken only from the selected window. You can only use the keyboard with one window at a time.

Moving a Window or Icon

Every window and icon has a location on the display screen. The *wmove(1)* command is used to change a window's location.

Concepts

Each window's location on the display screen is given in *x,y* pixel coordinates. When you move a window, you should specify coordinates which are valid for your display device—they should be within the resolution of the display screen to guarantee that you can see the window after the move operation is finished. (For details on display screen coordinates, refer to the "Display Screen Coordinates" section of the "Concepts" chapter in this manual.)

Executing wmove(1)

To move a window or icon, execute *wmove*; its syntax is:

```
wmove [-i] [-l x,y] [window_spec...]
```

Each parameter is optional. Descriptions of each follow.

Specifying the Window (window_spec...)

The *window_spec* parameter is a list of one or more windows to move. All specified windows are moved to the desired location. If *window_spec* is not given, *wmove* moves the window connected to standard input (typically, the window from which the command was executed).

Specifying Location (-l)

The `-l` option is used to specify the new window location. The new window location is given in *x,y*-pixel coordinates. If no window location is specified, the window(s) will be moved to the next default stair-step location, returned by the window manager.

For example, to move a window named *mywin* to pixel location 100,150, you would use:

```
wmove -l 100,150 mywin
```

The space between the `-l` and *x,y* is optional.

The next example moves the window *wconsole* to the next default stair-step location:

```
wmove wconsole
```

Execute this command several times with your *wconsole* window to see how the window stair-steps down the display screen.

Moving an Icon (-i)

Each window's icon has a location attribute also, distinct from the window's location. To move an icon's location, use the `-i` option.

The following example moves the icon for a window named *xx317* to *x,y*-pixel coordinates 123,456:

```
wmove -i -l123,456 xx317
```

Precautions

The results of attempting to move a window via *wmove* may not be immediately visible, if the window is:

- concealed
- located off-screen
- occluded by other windows
- normal, but its icon is moved
- iconic, but its normal form is moved.

Changing a Window's Size

You can use the *wsize(1)* command to change the size of one or more windows; its syntax is:

wsize [**-s** *w,h*] [*window_spec...*]

Each parameter is optional. Descriptions of each follow.

Specifying the Window (window_spec...)

The *window_spec* parameter specifies the name(s) of the window(s) for which the size will be changed. All specified windows are changed to the same size. If no window is specified, *wsize* changes the size of the window attached to standard input (typically, the window from which it was executed).

Specifying Size (-s)

The **-s** option is used to specify the new window size. *w,h* are in units appropriate to the window type: for terminal windows, *w,h* are *columns* and *rows* of characters; for graphics windows, *w* and *h* are pixels.

Attempting to change a window to a size larger than its maximum results in the window being changed to its maximum size.

Attempting to change a window to a size smaller than its minimum results in the window being changed to its minimum size. For thin-bordered windows, a terminal window's minimum size is one character cell; a graphics window's minimum size, one pixel. For normal-bordered windows, the minimum size is such that all manipulation areas in the border (i.e., control boxes, scroll arrows, and the first character of the window's label) can be seen. For null-bordered graphics windows, the minimum size is one pixel.

If no size is specified, and the window is a terminal window, then the window is changed to its maximum size. If the window is a graphics window, then the window is changed to a size such that its lower-right corner is flush with the lower-right corner of its raster.

Examples

The following changes a graphics window named *grwin* to 100 pixels wide by 200 pixels high:

```
wsize -s 100,200 grwin
```

The space following `-s` is optional.

The next example changes the window connected to standard input (typically, the window from which *wsize* is called) to its maximum size:

```
wsize
```

Shuffling Windows

As you accumulate more than one window on the display screen, they may become overlapped. When windows are piled in this manner, they are thought of as being in a display stack. Windows can be shuffled up or down through the displayed stack of windows using the *wdisp(1)* command.

Note

wdisp is also used to control the representation—normal, iconic, or concealed—of windows. This is discussed in the next section, "Changing a Window's Representation."

Shuffling the Top Window Down (-d)

To move the top window in the display stack to the bottom, and move the remaining windows up one position, use *wdisp* as follows:

```
wdisp -d
```

The resulting topmost window in the display stack automatically becomes the selected window.

Shuffling the Bottom Window Up (-u)

To move the bottom window in the display stack to the top, and move the remaining windows down one position, use *wdisp* as follows:

```
wdisp -u
```

The new top window automatically becomes the selected window when *wdisp* is used in this manner.

Changing a Window's Representation

Each window has two possible representations: *normal* or *iconic*. In addition, windows can be concealed, that is, made invisible. The *wdisp(1)* command is used to change a window's representation or concealment.

Concepts

Before using *wdisp*, you should understand the following basic concepts.

Normal vs. Iconic Representation

As mentioned previously, each window can be either *normal* or *iconic*. Figure 4-9 shows the *wconsole* window in its normal form.

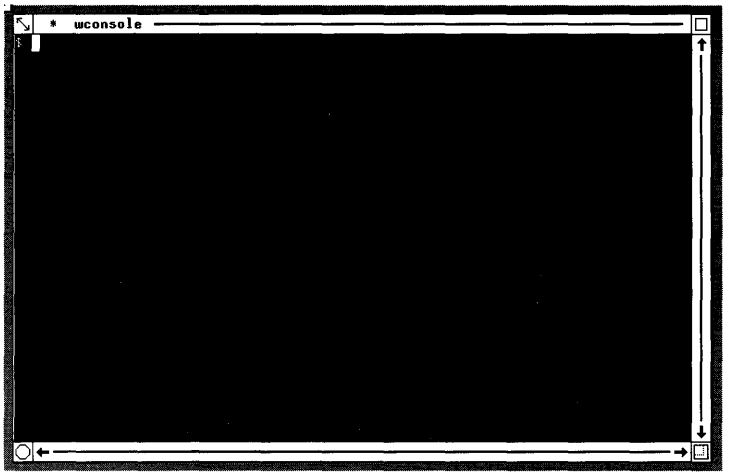

Figure 4-9. Normal Representation

Terminal and graphics windows each use different default pictures for their iconic representations. This is so that you can distinguish between the icon for a terminal window and a graphics window. Figure 4-10 shows a terminal window icon (on the left) and a graphics window icon (on the right).

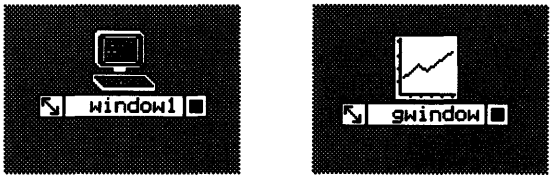

Figure 4-10. A Terminal Icon and Graphics Icon

Concealed vs. Displayed

Each window, regardless of whether it is normal or iconic, is also either *concealed* or *displayed.* It is not possible to see concealed windows or icons on the display screen: they are not displayed.

You might want to conceal a window or icon when you temporarily want to remove it from the display screen. For example, if you are playing a video game in a graphics window (but you're supposed to be generating quarterly reports), and your boss is coming over, you can conceal the graphics window until she's gone.

Note, however, that programs will still execute in concealed windows. You should temporarily stop any applications running in a window before concealing the window, if you don't wish to lose the application's output. (For example, if you don't temporarily stop the video game in the above example, you might get eaten by 4,927 ganglion invaders.)

Note also: even though a window or icon is displayed does not ensure that it will be visible on the display screen. It may be off-screen or occluded (covered) by other windows or icons.

Top vs. Bottom Window

When you have more than one window on the display screen, they tend to overlap. The "pile" of overlapped windows is known as the **display stack**. The *wdisp* command also allows you to move a window to the top or bottom of the display stack.

Executing wdisp(1)

To change a window to an icon, or vice versa, and/or to control the concealment or displayability of a window or icon, use *wdisp* with the following syntax:

```
wdisp [-tbo] [-ni] [window_spec...]
```

Each parameter is optional. The `-n` and `-i` options are mutually exclusive, that is, you cannot use both of them at the same time. The `-t`, `-b`, and `-o` options are also mutually exclusive and cannot be combined.

If no options are specified, then `-tn` is used as the default. For example,

```
wdisp my_window
```

produces the same effect as:

```
wdisp -tn my_window
```

Detailed descriptions of each parameter follow.

Specifying the Window (window_spec...)

The *window_spec* parameter specifies the name(s) of the window(s) for which to change representation, displayability, and/or position in the display stack. All specified windows are affected. If no window is specified, only the window attached to standard input (typically, the window from which *wdisp* is executed) will be affected.

Changing from Normal to Iconic Representation (-i)

To change a window from normal to iconic, use the `-i` option. This option assumes, of course, that the specified window is currently in normal representation.

Note: This option cannot be used with the `-n` option.

The following example changes the *wconsole* window to an icon:

```
wdisp -i wconsole
```

Changing from Iconic to Normal Reresentation (-n)

To change from iconic to normal representation, use the `-n` option. Likewise, this option assumes the specified window is currently iconic.

Note: This option cannot be used with the `-i` option.

This example changes the *wconsole* window from its iconic state back to normal representation:

```
wdisp -n wconsole
```

Note

Neither the `-n` or `-i` options, by themselves, affect a window's concealment or position in the display stack. They merely control the window's representation.

Displaying a Window as the Top Window (-t)

To display a window as the top window in the display stack, use the `-t` option. Note, this option works, regardless of whether a window is normal or iconic; in other words, an iconic window can be the top window in the stack, even though it is an icon.

The following example displays the *wconsole* window as the top window in the display stack:

```
wdisp -t wconsole
```

The `-n` and `-i` options can be used in combination with `-t`. For example, the following changes *wconsole* to an icon and makes it the top window in the display stack:

```
wdisp -ti wconsole
```

Displaying a Window as the Bottom Window (-b)

To display a window as the bottom window in the display stack, use the `-b` option. Like the `-t` option, this option works regardless of whether a window is normal or iconic.

The following example move the window named *wconsole* to the bottom of the display stack:

```
wdisp -b wconsole
```

As with `-t` option, `-b` can be combined with `-n` or `-i`. **However**, the `-b` and `-t` options cannot be combined. The following example changes *wconsole* to an icon and displays it as the bottom window in the stack:

```
wdisp -bi wconsole
```

Concealing a Window (-o)

To conceal a window, use the `-o` option. This option can be combined with the `-n` or `-i` options, but *cannot* be used with `-t` or `-b`.

The following changes *wconsole* to normal representation and conceals it:

```
wdisp -on wconsole
```

The next example conceals a graphics window, *ganglion_game*:

```
wdisp -o ganglion_game
```

Controlling a Window's Border

Via the *wborder(1)* command, you can control certain attributes of a window's border. This section discusses the use of *wborder* and its options.

Concepts

Before discussing *wborder*, you should understand some rudimentary concepts about window borders.

Normal, Thin, or Null Border

As mentioned in the "Concepts" chapter, a window's border can be either **normal** or **thin** for either terminal or graphics windows. In addition, the graphics window type supports the **null** border type (no border on the window).

Foreground and Background Border Colors

Each window has a foreground and background border color. By default, the background color is white and the foreground color is black. The *wborder* command allows you to specify new foreground and background colors for a window's border.

A **color** is actually an index into the graphics device's color map. Table 4-3 shows the default colors used when you power up your system. Note that the mapping in this table is valid only as long as you don't change the default color map for your system. In addition, although you may not explicitly change the color map, some other Starbase/DGL/AGP graphics application running to the screen or a window may change the color map; you should be aware of this fact.

Note

On monochromatic (black-and-white) systems, only black and white (0 and 1) colors are valid. Colors other than black or white default to the color map entry for white (1).

Table 4-3. Default System Color Map.

Color	Value
black	0
white	1
red	2
yellow	3
green	4
cyan	5
blue	6
magenta	7

Window Label

The *wborder* command also allows you to change a window's **label**. The window label is the name displayed in the window's border and icon. Normally, the window label is the same as the window name.

Note: Changing a window's label does **not** affect the window name. The window name remains the same. All commands still require you to use the window's name, if the label is different from the name.

Only the first 12 characters of the label are displayed in a terminal window's border. A graphics window's label can contain up to 128 characters.

Executing wborder(1)

The *wborder* command has the following syntax:

`wborder` [`-ntT`] [`-c` *fcolor,bcolor*] [`-l` *label*] [*window_spec*]

All parameters are optional. Descriptions of each follow.

Specifying the Window (window_spec...)

The *window_spec* parameter specifies the name(s) of the window(s) whose border is to be changed. All specified windows are affected. If no window is specified, only the window attached to standard input (usually, the window from which *wborder* is executed) will be affected.

Making the Border Thin (-t)

The `-t` option is used to make a window's border thin. The following example changes the *wconsole* window's border to thin:

```
wborder -t wconsole
```

Making the Border Normal (-n)

The `-n` option returns a window's border to normal representation. The following example changes the *wconsole* window's border back to normal:

```
wborder -n wconsole
```

Note: Using the `-n` option with *wborder* may fail if the window is too small to leave room for manipulation areas in the border.

Removing the Border of a Graphics Window (-T)

The `-T` option causes a graphics window to become borderless—i.e., the null border type. The following example changes a graphics window's border to null:

```
wborder -T graphwin
```

The `-n`, `-t`, and `-T` options cannot be used together, nor would it make any sense to do so.

Specifying Foreground and Background Colors (-c)

The `-c` option is used to set a window's foreground and background border colors. Colors can be specified either as color indexes (i.e., values from Table 4-3) or as abbreviations of color names (also from Table 4-3).

IMPORTANT

Foreground and background colors must be distinct (that is, they must be different from each other); otherwise the *wborder* command will fail.

For example, the following sets *wconsole*'s foreground and background colors to yellow and green, respectively:

```
wborder -c 3,4 wconsole
```

So does the following:

```
wborder -c yellow,green wconsole
```

And so does the following:

```
wborder -c y,4 wconsole
```

Note

On monochromatic (black-and-white) systems, only black and white (0 and 1) colors are valid. Colors other than black or white default to the color map entry for white (1).

Keeping in mind that foreground and background colors must be distinct, if follows that black should always be one of the specified colors on monochromatic systems. For example,

```
wborder -c yellow,green
```

will fail because both colors default to white. However,

```
wborder -c black,red
```

will work because the colors will default to black and white.

Note that you can also set foreground and background colors for characters displayed in terminal windows. These colors are set via term0 escape sequences. For details on setting terminal window foreground and background colors, see the *Term0 Reference Manual* and the "Term0 Windows" chapter of the *HP Windows/9000 Programmer's Manual.*

Setting the Window Label (-l)

To change a window's label to something other than the window's name, use the `-l` option. Remember: changing a window's label does not change its name; you must still use the window's name as the *window_spec* for any command.

For terminal windows, the maximum label length is 12 characters; for graphics windows, the window's label can contain up to 128 characters.

The following changes the label of the *wconsole* window to HELLO:

```
wborder -l HELLO wconsole
```

If the new label contains imbedded spaces, then you must enclose it within single (') or double (") quotes. The following changes the name of a graphics window named *grwin* to "this is my window":

```
wborder -l'this is my window' grwin
```

Managing Terminal Window Fonts

HP Windows/9000 allows you to use different fonts in each terminal window. The *wfont(1)* command is used to manage fonts in terminal windows.

Concepts

Before proceeding with the discussion on *wfont*, you should understand the following essential concepts for font management.

What is a Font?

In computer terminology, a font is a complete set of character representations, all of the same style and typically of the same cell size. This definition itself produces two new terms: **style** and **cell size**.

Font Style

Perhaps the best way to define style is to give some examples. The text you are now reading is all of the same style, *this text is of a different style—italic*, `and this text is of yet a different style--computer style`.

Cell Size

Each character displayed in a terminal window is displayed in a rectangle known as a cell. The cell is not normally visible; only the character is displayed.

All the fonts displayed in a given terminal window at a single time must all be of the same cell size. You can intermix different *styles* of fonts, but all the styles must be the same size.

Cell size is simply the pixel width and height of the cell in which characters of a font are displayed. Figure 4-11 should help clarify the idea of cell size.

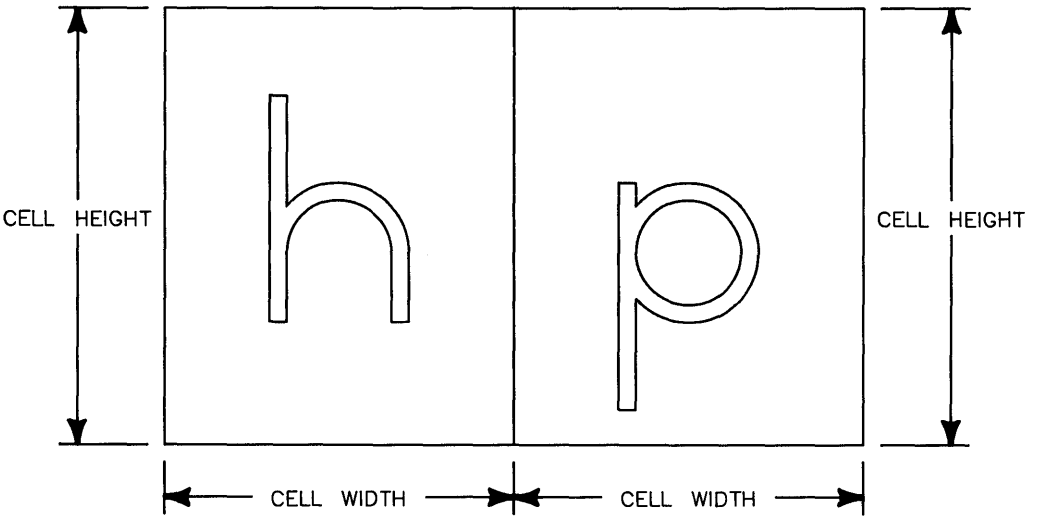

Figure 4-11. Font Cell Size

HP-15 (Two-Byte) Fonts

Most fonts require one byte to represent one character. HP-15 (two-byte) fonts require two bytes to represent some characters. Some of these two-byte characters may require two character cells to be displayed. This doesn't mean that the font size is different for the two-cell characters; it simply means that they require two cells instead of one to be displayed.

Base and Alternate Fonts

At any one time, each terminal window has a **base** and **alternate** font. By default, all text in the window's user area is displayed in the base font. To display text in the alternate font, a special character that switches any following text to the alternate font must be sent to the window. Typically, the base and alternate fonts have different styles.

Maximum Number of Fonts

Any window can have up to eight fonts loaded (i.e., available for use) simultaneously. The *wfont* command controls which fonts are loaded and which loaded fonts are designated as the base and alternate fonts. Consequently, this means you could see characters displayed in up to eight fonts at the same time in a window.

Selecting the Alternate Font

As mentioned above, text is not normally displayed in the alternate font. To cause text to be displayed in the alternate font, you must send a special ASCII control character to the window, the SO[1] character (which stands for **S**hift **O**ut of the base font).

You can send this character to a window by pressing the CTRL and letter N keys at the same time. Try typing the following to the HP-UX prompt in a terminal window (press the CTRL and N keys together at the same time):

```
echo "The base font. CTRL-N The alternate font." Return
```

HP-UX will respond by displaying the first phrase in the base font and the second phrase in the alternate font.

Selecting the Base Font

The base font is reselected (returned to) when either of the following conditions is met:

- an ASCII SI[1] character (which stands for **S**hift **I**n to the base font) is sent
- you leave the current line by any means (e.g., by an ASCII line-feed {LF[2]}, escape sequence, etc.)

In the previous example, the Return key forced a new line (LF), thus causing the base font to be reactivated. To activate the base font using the SI character, press the CTRL and letter O keys at the same time.

Font Files

Fonts are defined in **font files**. Font files contain such information as font cell size, font style, and raster definitions for each character. In order for a font to be used as the base or alternate font, it must be **loaded** from a font file and **activated**.

Font file names are descriptive and indicate font style and character set size. A typical example of a font file name is *lp.b.8U*. This means the font is a line printer font (`lp`), is bold (`b`), and is a Roman-8 font (`8U`).

---

[1] Decimal 14; octal 016.
[1] Decimal 15; octal 017.
[2] Decimal 10; octal 012.

Font Directories

Font files are stored in font directories; all font directories are located under the directory specified by the WMFONTDIR environment variable, typically `/usr/lib/raster`. All fonts of the same cell size are stored in a single directory; the name of the font directory indicates the size of fonts contained in that directory. For example, the directory `/usr/lib/raster/12x20` contains font files for all 12-by-20-pixel fonts.

Figure 4-12 illustrates the font directory structure.

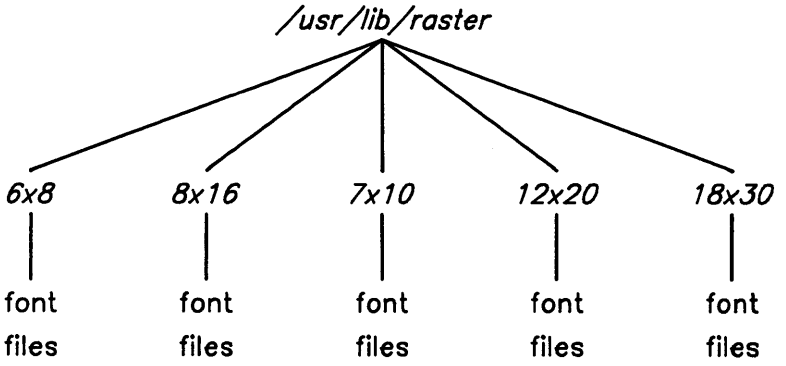

Figure 4-12. Font Directory Structure

Default Base and Alternate Fonts

As mentioned above, the window system uses default base and alternate fonts. The defaults used depend on (1) the resolution of the display screen, and (2) the value of the HP-UX environment variable $LANG, which defines the language to use for Native Language Support. (For details on the $LANG environment variable, see the *Native Language Support* tutorial in *HP-UX Concepts and Tutorials: Device I/O and User Interfacing.*)

By default, $LANG is not set to any value. In this case, terminal windows on high-resolution displays (1024x768 or 1280x1024) use `/usr/lib/raster/8x16/lp.8U` as the base font, and `/usr/lib/raster/8x16/lp.b.8U` as the alternate font. On low-resolution displays (512x400), terminal windows use `/usr/lib/raster/6x8/lp.8U` as the base font, and `/usr/lib/raster/6x8/lp.b.8I` as the alternate font.

If $LANG *is* set, then defaults are found under the `/usr/lib/raster/dflt` directory. Under this directory are two directories: `b` for base fonts and `a` for alternate fonts. Under those directories are two more directories: `h` for high-resolution displays (1024x768 or 1280x1024), and `l` for low-resolution displays (512x400). Under each of these directories

is a file named after the current language, as defined by $LANG; this file is linked to the default font file to use.

For example, suppose your system supports Japanese fonts. Then if the $LANG environment variable is set to "japanese", then the default base font for a high-resolution display would be `/usr/lib/raster/dflt/b/h/japanese`, which is linked to the font file `/usr/lib/raster/8x18/kanji.16K`. The default alternate font would be `/usr/lib/raster/dflt/a/h/japanese`, linked to `/usr/lib/raster/8x18/kana.8K`.

Font Management Escape Sequences

In addition to using the *wfont* command to manage fonts, you can use terminal window **escape sequences**. Escape sequences are special sequences of characters starting with the ASCII ESC[1] character. When sent to a terminal window, escape sequences tell the window to perform some task, for example, to activate a font. For details on using escape sequences in terminal windows, see the *Term0 Reference Manual* and the "Term0 Windows" chapter of the *HP Windows/9000 Programmer's Manual.*

Executing wfont(1)

Depending on how *wfont* is used, it has three different syntaxes:

`wfont` [`-F` *base_font_path alt_font_path* [*window_spec...*]]

`wfont -f` *font_path* [*window_spec...*]

or

`wfont` [`-ar`] *font_path* [*window_spec...*]

In all cases, if no *window_spec* parameter is given, *wfont* affects the window attached to standard input (typically, the window from which it was executed).

A Word About Font Paths

Font paths are common to each syntax of *wfont* (*font_path*, *base_font_path*, and *alt_font_path*). The font path is simply the path name of the font file to load.

To specify a font path, you can either give the whole path name from the root (for example, `/usr/lib/raster/8x16/lp.b.8U`), or you can specify the font path relative to the WMFONTDIR environment variable (`8x16/lp.b.8U` for the previous example). You can also specify a relative path name (begining with `./` or `../`).

Decimal 27; octal 033.

Replacing Both the Base and Alternate Fonts (-F)

The `-F` option is used when you wish to replace both the base and alternate font. It can also be used to switch to a pair of different-sized fonts.

When used with `-F`, *wfont* repaints the window's contents area so all characters written in the old alternate font are changed to the new alternate font; all others are changed to the new base font.

The following changes the *wconsole* window's base and alternate fonts to 8-by-16-pixel bold and italic fonts respectively:

```
wfont -F 8x16/lp.b.8U 8x16/lp.i.8U wconsole
```

The following changes *wconsole*'s base and alternate fonts to 12-by-20-pixel courier and bold courier fonts. Note that the window will change size accordingly when you switch to a different-sized font:

```
wfont -F 12x20/cour.0U 12x20/cour.b.0U wconsole
```

Note: If you attempt to change to a smaller font, and doing so would cause the window to be sized smaller than its minimum size, *wfont* will fail. A thin-bordered terminal window's minimum size is one character cell, so this case will always work. However, a normal-bordered terminal window must be big enough so all manipulation symbols and part of the window's label can be seen—so this case could fail.

Returning to Default Base and Alternate Fonts (no parameters)

If *wfont* is invoked with no parameters, then the current base and alternate fonts are returned to default values. The window is repainted so all characters written in the previous alternate font are changed to the default alternate font; all others are changed to the default base font.

Executing *wfont* with no parameters is analogous to using *wfont* with the `-F` option as follows:

`wfont -F` *base_font_path alt_font_path*

where *base_font_path* and *alt_font_path* are the path names for the default base and alternate fonts.

Replacing All Fonts with One Base Font (-f)

When you want to replace both the current base and alternate fonts with a new base font, use the `-f` option. Using this option causes the window's user (contents) area to be repainted; *all* characters are redisplayed in the new base font, even those that were displayed in the alternate font.

The following flushes the current base and alternate font from the *wconsole* window and replaces them with 8-by-16-pixel line printer font:

```
wfont -f 8x16/lp.8U wconsole
```

Activating a New Alternate Font (-a)

To load and activate a new alternate font, use the `-a` option. This option by itself does *not* cause the window's user area to be repainted. Only subsequent alternate-font characters are displayed in the new alternate font; old alternate-font characters remain unchanged.

The following activates 8-by-16-pixel italic as the new alternate font in the *wconsole* window:

```
wfont -a 8x16/lp.i.8U wconsole
```

Note

The `-a` option can be used only with same-size fonts. In other words, if the current font cell size is 8-by-16 pixels, then replace the alternate font *only* with an 8-by-16-pixel font.

Replacing the Base Font and Repainting (-r)

The `-r` option, when specified alone, causes the current base font to be replaced, and all characters in the old base font are repainted in the new base font.

The following replaces *wconsole*'s current base font to 8-by-16-pixel bold line printer font:

```
wfont -r 8x16/lp.b.8U wconsole
```

Note

The `-r` option can only be used with same-size fonts. In other words, if the current font cell size is 8-by-16 pixels, then replace the base font *only* with an 8-by-16-pixel font.

Replacing the Alternate Font and Repainting (-ar)

The `-a` and `-r` options, when used together, replace the alternate font and repaint all characters displayed in the old alternate font with the new alternate font.

For example, the following replaces the current alternate font with the 8-by-16-pixel math font:

```
wfont -ar 8x16/math.OM wconsole
```

Note

This usage of *wfont* is valid only with same-size fonts. In other words, if the current font cell size is 8-by-16 pixels, then replace the base font *only* with an 8-by-16-pixel font.

Activating a New Base Font (neither -a nor -r)

To load and activate a new base font, invoke *wfont* with only the path name of the new base font to use. This option by itself does *not* cause the window's user area to be repainted. Only subsequent base-font characters are displayed in the new base font; old base-font characters remain unchanged.

The following activates 8-by-16-pixel math font as the new base font in the *wconsole* window:

```
wfont 8x16/math.OM wconsole
```

Note

This usage of *wfont* is valid only with same-size fonts. In other words, if the current font cell size is 8-by-16 pixels, then replace the base font *only* with an 8-by-16-pixel font.

Listing Window Status

Using the *wlist(1)* command, you can list status information for windows. For example, you can discover which fonts are currently in use in a window, or a window's type, location, and select status.

Note

This section borrows many essential concepts from previous sections in this chapter. For example, it is assumed you understand the ideas of window location and size, and terminal window fonts.

Executing wlist(1)

To list window information, execute *wlist* which has the following syntax:

```
wlist [-fl] [window_spec...]
```

All parameters are optional. Descriptions of each follow.

Default Action (no options)

When neither `-f` nor `-l` is given, *wlist* simply displays the full path name(s) of the window type device interface(s) for the specified window(s). When no window specification is given, the path name for the window attached to standard input (typically, the window from which *wlist* is executed) will be displayed.

For example, the following displays the path name of the device interface for the *wconsole* window:

```
wlist wconsole
```

The next example lists device interface path names for all existing windows:

```
wlist '*'
```

This could also by typed as:

```
wlist \*
```

Typically, this example might list something like:

```
/dev/screen/wconsole
/dev/screen/window1
/dev/screen/graphwin
```

In the above example, three windows exist: *wconsole*, *window1*, and *graphwin*. The device interface for each window is found in the WMDIR directory, `/dev/screen`.

Listing Brief Font Information (-f)

When used alone, the `-f` option lists the device interface path name of each specified window (as above), followed by the full path names of all fonts loaded in the window. Note that this option works only with terminal windows.

The following example lists all the loaded fonts in the *wconsole* terminal window:

```
wlist -f wconsole
```

Assuming that *wconsole* contains only the default fonts loaded when it was created, the above example will typically produce a report like:

```
/dev/screen/wconsole:
/usr/lib/raster/8x16/lp.8U
/usr/lib/raster/8x16/lp.b.8U
```

Listing Extended Font Information (-fl)

The `-f` and `-l` options, when used together, produce a report similiar to the `-f` option alone, except that it contains additional font status information. Specifically, each font's size is displayed, along with an activation indicator:

size activation_indicator font_path

The activation indicator can have one of the four values described in Table 4-4.

Table 4-4. Activation Indicators.

Indicator	Description
b/a	The font is both base and alternate font.
base	The font is the base font.
alt	The font is the alternate font.
-	The font is currently neither the base nor the alternate font.

The following example lists extended font information for a window named *many_fonts*:

```
wlist -fl many_fonts
```

Let's assume the hypothetical window *many_fonts* has three fonts loaded, all 8-by-16-pixel fonts. The base font is the Roman-8 line printer font; the alternate font is the Roman-8 italic line printer font; and an additional, inactive font, Roman-8 bold line printer, is also loaded. The report produced by the above command would look like:

```
/dev/screen/many_fonts:
 8x16  base   /usr/lib/raster/8x16/lp.8U
 8x16  -      /usr/lib/raster/8x16/lp.b.8U
 8x16  alt    /usr/lib/raster/8x16/lp.i.8U
```

Listing Window Status Information (-l)

When invoked with only the `-l` option, *wlist* generates a columnar report giving status information for all specified windows. Descriptive headers are printed at the top of each column. Figure 4-13 shows a sample report generated by using this option.

```
WT KDTIA LOCX LOCY WIDE HIGH PANX PANY RASW RASH ILCX ILCY FGC BGC WINDOW
t0 kt---   10   28   80   24    ?    ?   80   48   10  560   0   1 wconsole
t0 -b-id  114  236   80   24    ?    ?   80   48   10  505   0   1 icon_win
gr --t-a  619  510  300  300    0    0  300  300   10  450   1   0 ganglions
```

Figure 4-13. A Sample wlist(1) Report

Table 4-5 describes the columnar data displayed by this report.

Table 4-5. Descriptions of wlist(1) Report Columns.

Column	Description
WINDOW	Each window's name is listed in the last column.
WT	This columns contains a two-character code for the window's type: **t0** for terminal windows, **gr** for graphics windows, and **gi** for IMAGE graphics windows.
K	If the window is selected, "k" appears in this column; otherwise, "-".
D	Gives the window's display status. If the window is the top window in the display stack, a **t** is displayed; if the window is the bottom window, a **b** is shown; if the window is concealed, **c**; otherwise, if the window is neither top nor bottom but is displayed, a **-** is shown.
T	Indicates the window's border style: **-** indicates a normal border, **t** means a thin border, and **T** means a null border.
I	Iconic status: **-** indicates that the window is in normal form, **i** means that the window is iconic.
A	Indicates the window's current autodestroy state: **-** means that the window is *not* recoverable; that is, it will not be automatically destroyed. An **a** in this columns indicates the window is recoverable and autodestroyable; that is, it will be automatically destroyed when its shell or application terminates. A **d** in this column means the window is recoverable but not autodestroyable; in other words, it will be automatically destroyed when a new window is created, after its application closes it (usually at the termination of the application).
LOCX, LOCY	These columns give the *x,y* location (in pixels) for the window when it is in normal form.
WIDE, HIGH	The columns display the window's width and height: *columns* and *rows* for terminal windows, and pixel *width* and *height* for graphics windows.
PANX, PANY	Give the current pan position into a graphics window. Pan position is the *x,y* offset (in pixels) of the graphics window's view into its raster. These columns have no meaning for terminal windows, and are filled with a question mark (?).
RASW, RASH	The width and height of the window's raster (for graphics windows) or scroll buffer (for terminal windows). For graphics windows, units are in pixels; for terminal windows, units are columns and rows of characters.
ILCX, ILCY	The location of the window's icon is given by these columns. Coordinates are in *x,y* pixels.
FGC, BGC	These columns give the window's foreground and background border colors, respectively. Colors are given as indices into the system color map.

Environment Variables 5

This chapter discusses the use of window system environment variables. Most users will *not* require the information provided here. However, if you need to "fine tune" your system, that is, if you need to alter default window system characteristics, then this is the chapter to read.

The following topics are discussed in this chapter:

- concepts essential to understanding the use of window system environment variables
- setting window system environment variables
- input/output special files
- the bit-mapped display driver
- graphics tablet scaling
- configuring the interactive user interface
- default fonts
- default colors
- desk top pattern
- interactive timeout and locator tracking
- pseudo-tty (*pty*) special files
- setting window manager real-time priority
- window manager memory locking
- shared memory.

Concepts

What Are Window System Environment Variables?

A number of **window system environment variables** define the window system's default run-time environment. Only a few of these variables are set when the window system starts executing; they are set to default values in the *wmstart(1)* shell script which starts the window system. (For details on how *wmstart* works, see the section "Starting the Window System" in the "Using Commands" chapter).

The remaining variables needn't be set for the window system to work properly. If a variable is not set before the window manager starts executing, the window manager assumes a reasonable default value. Note, however, you still can set them if you wish to alter certain default characteristics of the window system.

Why Set Environment Variables?

For most users, the default window system configuration is quite acceptable and no changes need be made. However, if you have specialized needs/applications, you may need to alter the system configuration.

IMPORTANT

Window system environment variables are a resource. And like any resource, using them can be costly. You don't get an unlimited number of environment variables—there is a limited amount of space in a process's environment to hold variables and their values. We recommend that if you decide to alter environment variables, change only those which *absolutely* need to be set.

By setting window system environment variables to values other than the defaults, you can alter the way the window system runs. Following are examples of the types of things configurable via environment variables:

- the interactive user interface
- the mapping the graphics tablet to the display screen
- the path names of window system input and output special (device) files
- default fonts to use in window borders, icon labels, pop-up menus, terminal windows, and softkey labels
- the location and size of window system shared memory
- default window border foreground and background colors.

A Summary of Environment Variables

Table 5-1 lists window system environment variables along with a brief description of each and the variable's default value. More-detailed descriptions of the variables are found later in this chapter.

Table 5-1. Window System Environment Variables.

Variable	Description	Default
TERM	This HP-UX shell variable is usually set to **hp9836** for the window system. This variable is set by *wmstart*.	**hp9836**
WMDIR	Directory where the window manager's device interface (special file) and window device interfaces are put by the window manager. (For details, see the "Special Files" section in this chapter.) This variable is set by *wmstart*.	**/dev/screen**
WMKBD	Special file for the keyboard. (See the "Special Files" section.) This variable is set by *wmstart*.	**/dev/hilkbd**
WMINPUTCTLR	Special file for the HP-HIL input controller. (See the "Special Files" section.) This variable is set by *wmstart*.	**/dev/rhil**
WMLOCATOR	Special file for the HP-HIL locator device (e.g., mouse or graphics tablet) used with your system. (See the "Special Files" section.) This variable is set by *wmstart*.	**/dev/locator**
WMSHMSPC	The maximum size of window system shared memory used by the window manager and window processes. (See the "Shared Memory" section in this chapter.) This variable is set by *wmstart*.	**0x200000** (2Mb)
WMFONTDIR	Gives the path name of the directory under which font directories and font files are stored. (See the "Default Fonts" section in this chapter.) This variable is set by *wmstart*.	**/usr/lib/raster**
WMSCRN	Special file of the physical display where the window system executes.	**/dev/crt** or **/dev/ocrt** (HP 98730)
WMDRIVER	The Starbase device driver used when writing to your display.	Depends on your display model.
SB_DISPLAY_ADDR	Memory address in the user address space of the Starbase shared memory. Used to configure window system shared memory.	**0xb00000** (11Mb)

Table 5-1. Window System Environment Variables, Con't

Variable	Description	Default
WMLOCSCALE	Maps a sub-portion of the graphics tablet (if used on your system) to the window system physical display screen.	The entire screen maps to the entire graphics tablet.
WMPTYMDIR	Path name of the directory where master psuedo-tty (*pty*) special files are located.	`/dev/ptym`
WMPTYSDIR	Path name of the directory where slave pseudo-tty (*pty*) special files are located.	`/dev/pty`
WMPTYNAME	Starting name of the set of pseudo-tty (*pty*) special files used for windows.	`ttyp8`
WMPTYCNT	Number of contiguous pseudo-ttys (*ptys*) used by the window manager.	31
WMPTYCACHECNT	Number of *ptys* from the *pty* pool to pre-open. Pre-opened *ptys* will not be consumed by processes other than the window manager.	10
WMIATIMEOUT	Gives: (1) the timeout period (in seconds) for interactive operations, and (2) the number of milliseconds relinquished by the window manager during tracking.	`0x1c003c` See "Interactive Time-out and Tracking — WMIATIMEOUT."
WMIUICONFIG	Allows you to reconfigure the window system's interactive configuration.	`0x80781` See "Configuring the Interactive User Interface — WMIUICONFIG."
WMCONFIG	Controls window manager memory locking, clearing of graphics window retained rasters on window create, double buffering and software versus hardware sprites.	`0x0` See "Changing the Window Manager Configuration — WMCONFIG."
WMRTPRIORITY	Real-time priority for the window manager and window servers. Ranges from 0 (highest) to 127 (lowest). Zero, and numbers near 0, are dangerous and should not be used.	`0x787c` i.e. 120 for window manager, 124 for servers.

Table 5-1. Window System Environment Variables, Con't

Variable	Description	Default
WMDESKPTRN	Dither pattern for desk top. Current valid values are 0, 25, 50, 75, 100.	50
WMDESKFGCLR	Foreground color for the desk top.	0 — black, unless the system color map has changed from default.
WMDESKBGCLR	Background color for the desk top.	1 — white, unless the system color map has changed from default.
WMBDRFGCLR	Default foreground color used for new window borders.	$WMDESKFGCLR
WMBDRBGCLR	Default background color used for new window borders.	$WMDESKBGCLR
WMMENUFONT	Font used for pop-up menu text.	Depends on screen size and $LANG HP-UX environment variable.
WMSFKFONT	Font used for softkey labels.	Depends on screen size and $LANG HP-UX environment variable.
ICONFONT	Font used for icon labels.	Depends on screen size and $LANG HP-UX environment variable.
BANNERFONT	Font used in window borders.	$WMMENUFONT
WMBASEFONT	Default font to load as the base font in newly created terminal windows. We recommend that you never change this variable unless absolutely necessary.	Depends on screen size and $LANG HP-UX environment variable.
WMALTFONT	Default font to load as the alternate font in newly created terminal windows. We recommend that you never change this variable unless absolutely necessary.	Depends on screen size and $LANG HP-UX environment variable.

Setting Environment Variables

Window system environment variables can be set primarily through the following four methods; detailed descriptions of each method, along with examples, are described below:

- on the command line when *wmstart* is invoked
- from the system-wide login initialization scripts (`/etc/profile` for Bourne shell users, `/etc/csh.login` for C-shell users)
- from your personal login initialization script ($HOME/`.profile` for Bourne shell users, $HOME/`.login` for C-shell users)
- in your personal copy of *wmstart.*

Descriptions of each method are discussed next.

Note

Attempting to set window system environment variables when the window system is already running **will not work**. You must set environment variables before the window manager starts executing, as discussed in the following methods.

Setting Variables on the Command Line

Perhaps the simplest, and safest, method for setting environment variables is setting them on the command line when *wmstart* is invoked. To set variables on the command line, use the *env(1)* command.

The *env* command, when used to start the window system, has the following syntax:

```
env VARIABLE=value ... wmstart
```

where `VARIABLE` is the name of the environment variable you wish to set, and *value* is its value. The "..." simply means that you can set more than one variable on the command line.

For example, suppose you wish to change the desk top dither pattern from the default (50) to a brighter pattern (25); you also wish to change the default menu font to a larger, 12-by-20-pixel courier font. You would use *env* as follows:

```
env WMDESKPTRN=25 WMMENUFONT=/usr/lib/raster/12x20/cour.OU wmstart Return
```

Setting from System-Wide Initialization Scripts

A more permanent method for setting environment variables is to set them in the system-wide login initialization scripts: `/etc/profile` for Bourne shell users, `/etc/csh.login` for C-shell users. Commands in these scripts are executed for every user who logs in (on multi-user systems) or when you power up (on single-user systems). Therefore, window system environment variables that are set and exported in these scripts will be used by the window system.

Note

Only the super-user can set window system environment variables using this method. This is because the `/etc/profile` and `/etc/csh.login` shell scripts must be edited, and only the super-user has write permission for these files.

The advantage of using this method is that window system environment variables are automatically set for all users of the system when they log in. This way the window system has the same configuration for all users. In addition, users can still alter the values of environment variables, if so desired, by setting them on the command line.

Setting Variables from /etc/profile

To automatically set window system environment variables when users log in to a Bourne shell, you must set and export the desired variables in `/etc/profile`. Variables are exported from `/etc/profile` to users via the *export* command.

Adding the following to `/etc/profile` sets the WMDESKPTRN and BANNERFONT variables to new values for all users of the Bourne shell:

```
WMDESKPTRN=75 ; export WMDESKPTRN     # change the desk top dither pattern
BANNERFONT=/usr/lib/raster/7x10/lp.8U
export BANNERFONT                     # change the default border font
```

Setting Variables from /etc/csh.login

To automatically set window system environment variables when users log in to a C-shell, you must set and export the desired variables in `/etc/csh.login`. The *setenv* C-shell command sets variables and exports them to users.

The following, when added to `/etc/csh.login`, performs the same tasks as the Bourne shell example above, except that it sets environment variables for all C-shell users:

```
setenv WMDESKPTRN 75                          # change the dither pattern
setenv BANNERFONT /usr/lib/raster/7x10/lp.8U # change the border font
```

Setting from Your Login Shell Script

You can also set window system environment variables from your login initialization script: $HOME/`.profile` for Bourne shell users, $HOME/`.login` for C-shell users. When you log in (on multi-user systems) or power up (on single-user systems), commands in these scripts are automatically executed. Therefore, any window system environment variables you set and export from these scripts will be used by the window system.

The advantage of using this method is that you can customize window system environment variables without affecting other users of the window system. The variables set in your login script will not affect other users of the window system.

Setting Variables from $HOME/.profile

If you are a Bourne shell user, then you should set the environment variables in your `.profile` login script. As with the `/etc/profile` script, you must set and export the variables that you wish to change.

For example, if you want the timeout period for interactive operations to be 15 seconds instead of the default 60 seconds, you would enter the following in your $HOME/`.profile` file:

```
WMIATIMEOUT=15 ;export WMIATIMEOUT   # set interactive timeout to 15s
```

Setting Variables from $HOME/.login

If you are a C-shell user, then you should set the environment variables in your `.login` initialization script. The variables must be set and exported, as with the `/etc/csh.login` script.

The following example sets the default softkey font to 8-by-16-pixel bold line printer font:

```
setenv WMSFKFONT /usr/lib/raster/8x16/lp.b.8U # reset default softkey font
```

Changing Your Copy of wmstart(1)

The final method for setting environment variables is to set them in a personal copy of the *wmstart* shell script—not the actual *wmstart* script itself. Rather than actually changing the *wmstart* shell script, we strongly suggest that you make a personal copy of *wmstart* (from **/usr/bin/wmstart**), rename it to something other than *wmstart*, and set the environment variables in your personal copy. Thereafter, when you want to execute windows, run your personalized copy instead of *wmstart*.

The reason we suggest making a personal copy of *wmstart* is so that the original *wmstart* script will always remain intact. In addition, making a personal copy ensures that your copy of *wmstart* won't get destroyed if you update your system.

Wmstart is a Bourne shell script, so Bourne shell syntax should be used when setting environment variables in *wmstart*:

```
VARIABLE=value ; export VARIABLE
```

Special Files

Because HP Windows/9000 is interactive, it makes intensive use of input and output devices. For output, Windows/9000 requires a bit-mapped display; for input, HP-HIL devices such as a keyboard and optional mouse or graphics tablet are used.

Each input or output device has an associated **special file** (also known as a **device file**) through which communication with the device is facilitated. In addition, HP Windows/9000 makes extensive use of pseudo-tty (*pty*) special files to communicate with windows. This section discusses HP Windows/9000 input/output devices and their special files, *pty* special files, and how they relate to window system environment variables.

Note

This section does **not** give a detailed discussion of the input/output architecture of HP Windows/9000. If you require this information, you should read the "Concepts" chapter of the *HP Windows/9000 Programmer's Manual.*

Note

It is assumed the reader knows how to use HP-HIL devices with HP-UX. For details on HP-HIL devices, you should read the tutorial *Using HP-HIL Devices with HP-UX* in *HP-UX Concepts and Tutorials: Facilities for Series 200, 300, 500.*

Pseudo-Terminal (pty) Special Files

To perform input from and output to windows, Windows/9000 uses *pty(7)* special files. Each terminal window uses three *ptys*, and each graphics window uses one *pty*. Each *pty* is comprised of a slave *pty* and master *pty*. The window manager itself also uses a *pty*.

For details on *pty* special files, see your *System Administrator Manual* and the *pty(7)* page in the *HP-UX Reference*. For details on how *ptys* are used with the window system, see the "Concepts" chapter of the *HP Windows/9000 Programmer's Manual*.

You may also want to read the "Pseudo-Terminal (pty) Limitation" section of the appendix "Window Limitations" in this manual: it provides details on why you might want to change the *pty*-related environment variables discussed here.

The $WMDIR Directory

All window system *pty* special files are stored in the directory specified by the WMDIR environment variable. The *wmstart* shell script sets WMDIR to `/dev/screen` by default.

You can see all the window system *ptys* by listing the $WMDIR directory from an HP-UX shell in a window:

```
ll -a $WMDIR
```

If you change this variable, be sure to set it to the path name for a valid directory; otherwise the window system will fail.

IMPORTANT

The *wmstart* shell script **removes all character-type special files** in the $WMDIR directory before starting the window manager. Therefore, be sure not to set WMDIR to the path name of a directory containing character special files you want to keep. **Never set WMDIR to `/dev` because all character special files for devices in your system will get destroyed when *wmstart* is executed.**

Master and Slave ptys - WMPTYMDIR and WMPTYSDIR

As mentioned at the start of this section, each *pty* is comprised of master and slave *ptys*. The WMPTYMDIR variable gives the path name of a directory containing master *ptys*; the WMPTYSDIR variable, the path name of a slave *pty* directory.

The *wmstart* shell script does *not* set WMPTYMDIR or WMPTYSDIR. When these variables are not set or null, the window manager uses `/dev/ptym` for the directory containing master *ptys* and `/dev/pty` for the directory containing slave *ptys*.

Defining the Starting Name for ptys - WMPTYNAME

When the window manager creates a window, it must use *pty* special files from the master and slave directories defined by WMPTYMDIR and WMPTYSDIR, respectively. The WMPTYNAME tells the window manager the name of the first *pty* in a contiguous "pool" of *pty* special files from the master and slave directories. The window manager will then consume *ptys*, starting at the *pty* name given by WMPTYNAME.

This variable should follow the following *pty* naming convention:

```
tty[p-v][0-9a-f]
```

By default, *wmstart* does not set this variable, and the window manager assumes the starting *pty* to be *ttyp8*.

Defining pty Pool's Size - WMPTYCNT

In addition to knowing the first *pty* in the *pty* pool, the window manager must know the size of pool that can be used from the master and slave directories. The WMPTYCNT variable defines the size of the *pty* set.

By default, the *wmstart* script does not set WMPTYCNT, and the window manager assumes the *pty* pool's size is 31.

Note

This number directly affects the maximum number of windows attainable with the window sytem. You should refer to the "Pseudo-Terminal (pty) Limitations" section of the "Window Limitations" appendix in this manual for details on the implications of setting this and other *pty* environment variables.

Pre-Opening pty's in the Pool - WMPTYCACHECNT

WMPTYCACHECNT tells the window manager how many *ptys* to pre-open for its own use. The window manager will open the specified number of *ptys* from the pool. These opened *ptys* will then be used to create new windows. When a new window is created, the window manager uses *ptys* from the cache until they run out, at which point the window manager uses *ptys* from the rest of the pool. WMPTYCACHECNT simply defines the size of the cache.

By default, *wmstart* does not set this variable. The window manager assumes a *pty* cache size of 10.

Note: Increasing WMPTYCACHECNT from the default value of 10 will cause the window system to take longer to start running (because it must pre-open more *ptys*). But once it starts running, more windows can be created faster because the *ptys* to use for new windows will already be opened.

Example

The following example sets the maximum number of *ptys* to 21; in addition, the starting set of *ptys* is changed from the default, *ttyp8*, to *ttyq0*:

```
WMPTYNAME=ttyq0 ; export WMPTYNAME
WMPTYCNT=21 ; export WMPTYCNT
```

The Display Screen Device - WMSCRN

To visually interact with users, HP Windows/9000 requires a bit-mapped display. The special file for your display screen is created when the window system is installed. Typically, the special file is named *crt* and is found in the `/dev` directory (`/dev/crt`).

The WMSCRN Variable

The WMSCRN environment variable specifies the path name of the bit-mapped display used with your window system. When the window manager starts executing, it looks at WMSCRN to determine which special file to open as the display screen.

Setting WMSCRN

If WMSCRN is not set or is null (as is the default case), the window manager assumes the display screen's special file path name is `/dev/crt`, and opens it for output. Otherwise, if WMSCRN is set, the window manager attempts to open the specified path name as the window system output device. One exception to this is the HP 98730 display, where the default is `/dev/ocrt`, not `/dev/crt`.

Possible Errors

If WMSCRN is set to an invalid value (for example, if it is set to the path name of a non-bit-mapped terminal, or if it is set to an invalid path name), the window manager will fail and terminate.

Keyboard Input - WMKBD

HP Windows/9000 allows you to have an HP-HIL keyboard. The special file for your HP-HIL keyboard is created when the window system is installed. By default, the window system looks for the keyboard special file named `/dev/hilkbd`.

If WMKBD is null, then the window manager assumes there is no keyboard, and keyboard input to windows is disabled.

The Cooked Keyboard Driver

The HP-HIL keyboard has two special files:

- a standard HP-HIL special file which uses the standard HP-HIL input driver—the **raw keyboard driver**—and has an HP-HIL address, based on its position in the HP-HIL device loop
- a special file which uses a **cooked keyboard driver**—the `/dev/hilkbd` special file.

HP Windows/9000 uses the cooked keyboard special file for all keyboard input, instead of the standard (raw) HP-HIL special file.

The WMKBD Variable

The WMKBD environment variable specifies the path name of the cooked-driver special file for your system's HP-HIL keyboard. When the window manager starts executing, it looks at WMKBD to determine which special file to open for keyboard input.

Setting WMKBD

By default, WMKBD is set to `/dev/hilkbd` in the *wmstart* shell script. If you change WMKBD to null (as `WMKBD=""`), keyboard input will be disabled in the window system.

Possible Errors

If WMKBD is set to an invalid value (for example, if it is set to the path name of a raw HP-HIL special file, or if it is set to an invalid path name), unpredictable results will occur. If you change this value, you should be absolutely sure it is set to a valid value.

The Locator Device – WMLOCATOR

HP Windows/9000 allows you to have a single locator device such as a mouse or graphics tablet stylus or puck. The device file for the locator is named `/dev/locator` by default, and is automatically linked to `/dev/hil2` when Windows/9000 is installed. If you add a locator device later, or change the locator's address in the HP-HIL device loop, you must update `/dev/locator` accordingly.

The WMLOCATOR Variable

The WMLOCATOR environment variable specifies the path name of the locator device's special file. When the window manager starts up, it looks at WMLOCATOR to determine which special file to open for locator device input.

If WMLOCATOR is set to null, then locator input will be disabled.

Setting WMLOCATOR

By default, WMLOCATOR is set to `/dev/locator` in the *wmstart* shell script; `/dev/locator` is linked to the HP-HIL special file corresponding to the locator device used with your system.

For example, suppose you use a mouse device with your system, and the mouse is the second device in the HP-HIL loop. If you list the `/dev` directory, you'll see a standard HP-HIL special file for the mouse device, typically `/dev/hil2`. In addition, you would see the locator special file `/dev/locator` linked to `/dev/hil2`.

Possible Errors

If WMLOCATOR is set to an invalid value (for example, if it is set to the path name of an invalid locator device such as a button box), unpredictable results will occur. If you change this value, you should be absolutely sure it is set to a valid value.

Also, if your computer system has more than two HP-HIL devices (e.g., more than just the keyboard and mouse or graphics tablet puck), then the default configuration (`/dev/locator` linked to `/dev/hil2`) may be wrong. `/dev/locator` should be linked to the approriate window system locator device. For example, if you want to use the puck switch for input, and the graphics tablet is the third device in the HP-HIL loop, then `/dev/locator` should be linked to `/dev/hil3`.

The HP-HIL Input Controller - WMINPUTCTLR

HP Windows/9000 also must be able to communicate with the HP-HIL input controller. The special file for this processor is typically named `/dev/rhil`.

The WMINPUTCTLR Variable

The WMINPUTCTLR environment variable specifies the path name for the input controller's special file.

Setting WMINPUTCTLR

WMINPUTCTLR is set, by default, to `/dev/rhil` in the *wmstart* shell script. If set to null (as `WMINPUTCTLR=""`), keyboard input and the beeper will be disabled.

As always, if you change WMINPUTCTLR, you should be sure change it to a valid value. Otherwise, unpredictable results may occur.

The Bit-Mapped Display Driver – WMDRIVER

As mentioned in the previous section, the WMSCRN variable specifies the path name of the special file for the graphics display used with your window system. Each display special file has an associated device driver which allows you to write information to the bit-mapped display. The WMDRIVER environment variable specifies which driver to use.

The WMDRIVER environment variable is normally not set by *wmstart*. When it is not set (or is null), the window manager uses a default driver appropriate for the attached display.

Normally you should never need to set this variable because the window manager automatically selects the correct driver. However, if you should need to change it from the default, be sure to set it to a valid Starbase driver for the display hardware.

Graphics Tablet Scaling - WMLOCSCALE

Windows/9000 allows you to map a specific rectangular portion of the graphics tablet to the entire display screen of your window system. In other words, you can define a sub-area of the graphics tablet to map to the entire display screen. The window system environment variable WMLOCSCALE is used to map the graphics tablet to the display screen.

Why Use Graphics Tablet Scaling?

Graphics tablet scaling is useful when you have a window system application that uses the graphics tablet as a locator device. By using scaling, only part of the graphics tablet is required by the window system, and the remaining areas can be used by your application.

For example, suppose you have a template that overlays the graphics tablet; different areas of the template correspond to different functions that can be performed by the application. Graphics tablet scaling allows a specific area of the graphics tablet to be devoted to window system use; the remaining areas can be used by your application.

Default Value

When WMLOCSCALE is not set (or is null), then the entire graphics tablet maps to the entire display screen. This is the default case.

Setting WMLOCSCALE

The WMLOCSCALE variable requires four values, all specified in a string:

```
WMLOCSCALE="x1 y1 x2 y2"
```

These values specify the coordinates of a sub-rectangle on the graphics tablet that will map to the screen. The *x1* and *y1* coordinates correspond to the lower-left corner of the display screen; *x2* and *y2* correspond to the upper-right corner of the screen. Figure 5-1 illustrates the relation between graphics tablet coordinates (*x1,y1,x2,y2*) and the display screen.

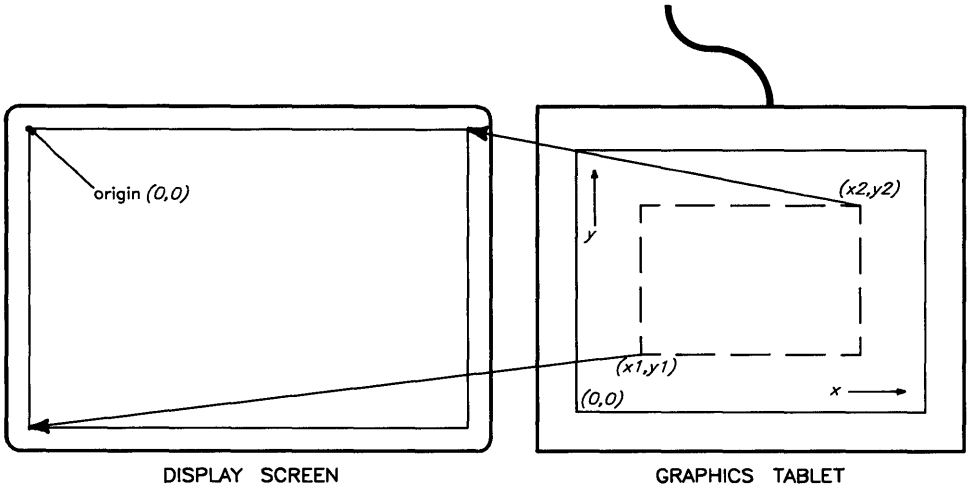

Figure 5-1. Graphics Tablet Scaling

Coordinates can be specified as either absolute or percentage. Absolute coordinates correspond to actual graphics tablet coordinates and are specified by whole numbers; percentage coordinates correspond to a percentage location of the tablet and are specified as whole numbers with a trailing percent sign.

Absolute Coordinates

Absolute coordinates require knowledge of the resolution of the graphics tablet. The lower-left corner of the graphics tablet is usually the origin (location *0,0*). The *x* coordinates increase to the right; *y* coordinates increase upward. Figure 5-2 illustrates this concept.

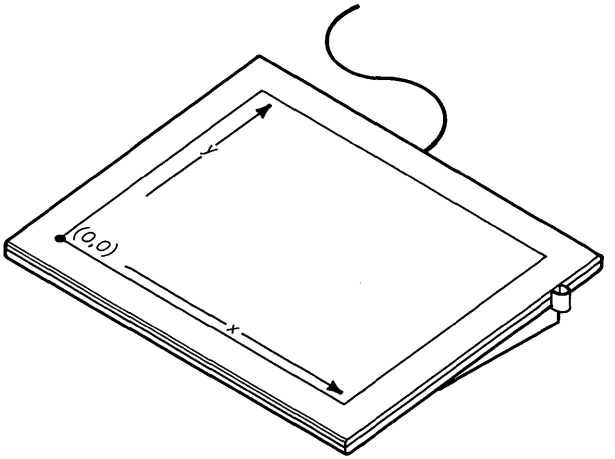

Figure 5-2. Graphics Tablet Absolute Coordinates

Percentage Coordinates

Percentage coordinates are handy because you don't need to know any specifics on the size of the graphics tablet; they just estimate a portion (location) of the tablet. The lower-left corner of the graphics tablet has percentage coordinates *0%, 0%*; the upper-right corner (regardless of graphics tablet size) has coordinates *100%, 100%*. Figure 5-3 illustrates percentage coordinates.

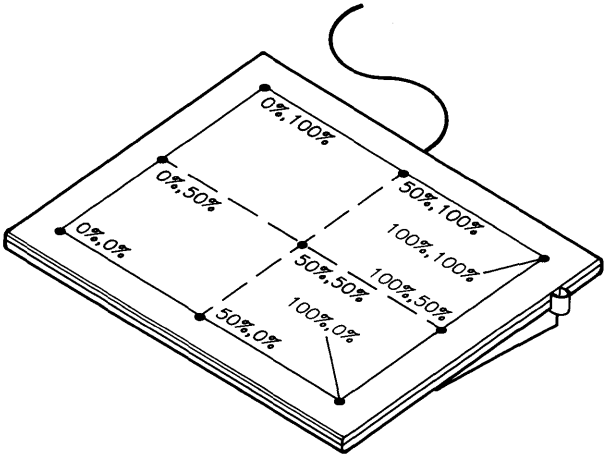

Figure 5-3. Graphics Tablet Percentage Coordinates

Combining Absolute and Percentage Coordinates

Absolute and percentage coordinates *can* be combined. For example, the following is valid:

```
WMLOCSCALE="0 0 50% 75%"
```

Examples

In the following example, the screen is mapped to a rectangle that is one-fourth the graphics tablet area, and which is centered in the graphics tablet (as shown in Figure 5-4):

```
WMLOCSCALE="25% 25% 75% 75%"
```

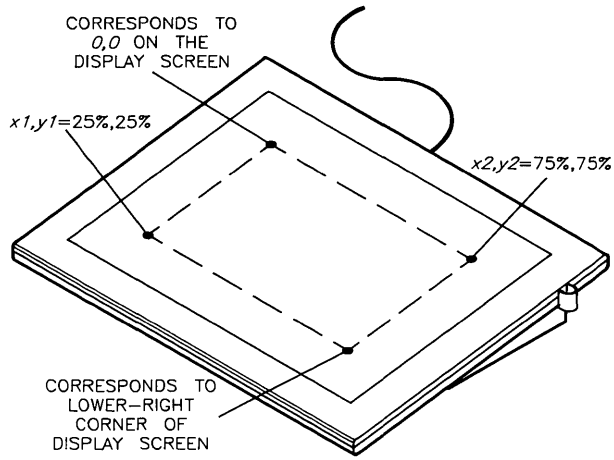

Figure 5-4. Scaling Example 1

The next example maps the lower-left corner of the tablet to the lower left-corner of the screen, using absolute coordinates; the upper-right corner of the display maps to the center of the graphics tablet. In other words, the lower-left quadrant of the graphics tablet maps to the entire screen (as shown in Figure 5-5):

```
WMLOCSCALE="0 0 50% 50%"
```

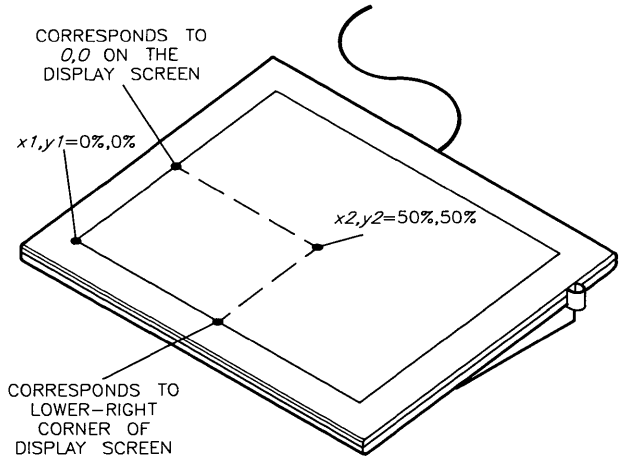

Figure 5-5. Scaling Example 2

Precautions

- Mapping the screen to a tablet rectangle that has lower resolution than the screen will cause there to be less than a one-to-one correspondence between screen pixels and tablet coordinates.
- If you use this feature, the locator and echo can move to coordinates that are off the screen. Normally this is not possible. However, using this feature, you can inquire locator position on areas outside the screen rectangle and can define these areas to be special to your application (as in the template example above).

Configuring the Interactive User Interface - WMIUICONFIG

HP Windows/9000 allows you to reconfigure default interactive characteristics of the window system. For example, you can disable any of the window control boxes, you can redefine the select button, and you can disable pop-up menus over different screen areas. The WMIUICONFIG variable is used to reconfigure the interactive user interface.

The WMIUICONFIG Variable

The interactive user interace is defined by a string of bits. Each bit represents a particular characteristic of the window system. By setting or clearing a bit, you can alter the interactive user interface.

Normally, the WMIUICONFIG environment variable is not set. When this variable is not set, the window manager uses a default configuration (defined later). If you want to alter the default configuration, you must specify a new bit string via the WMIUICONFIG environment variable.

Setting WMIUICONFIG

WMIUICONFIG is set to a number which represents the new bit string to use for the interactive user interface. This number can be decimal, octal, or hexadecimal; however, **we suggest you use hexadecimal numbers** since they are more convenient for setting bits.

Table 5-3 briefly defines each bit in the interactive user interface definition. Bit positions are given in hexadecimal values; the least-significant bit is referred to as **bit 0**.

Table 5-3. Interactive User Interface.

Bit(s)	Description
0x000007f	The seven least-significant bits specify which buttons on the HP-HIL locator device are used by the window manager for interactive window operations. If a bit is set, the corresponding button on the HP-HIL input device is enabled; if the bit is not set, the button is disabled. Bit 0 corresponds to button one (i.e., the leftmost mouse button, the graphics stylus point, and the leftmost puck button); bit 1 corresponds to button two (the rightmost mouse button and the rightmost puck button); and so on. Independent of whether or not a button is enabled or disabled for interactive window operations, a button press over a window or non-sensitive area of the border is sent to processes sensitive to button input.
0x0000080	If this bit is set, the [Select] key on the keyboard is enabled; if not set, it is disabled.
0x0000100	This bit, if set, causes a window to automatically come to the top of the display stack if the window is selected.
0x0000200	If this bit is set, then whenever a window is changed from normal to iconic representation (or vice versa), the window automatically becomes the top window in the display stack. If not set, the window's position is unchanged in the display stack.
0x0000400	When this bit is set, a window automatically becomes the top window, if when moving or changing the window's size, the window is unobscured by any other windows (not occluded).
0x0000800	This bit performs the opposite function of the previous bit (0x0000400): a window is automatically topped if moving it or changing its size causes the window to be obscured (occluded) by another window.
0x0001000	Setting this bit disables the move control box for every window and icon.
0x0002000	This bit, when set, disables the icon/normal control box for every window and icon.
0x0004000	Setting this bit disables the size control box for every window.
0x0008000	This bit, when set, disables the pause control box for every window.
0x0010000	Setting this bit disables the scroll arrows on windows.
0x0020000	If this bit is set, window system pop-up menus are disabled over window borders—you can't get a pop-up menu by clicking the pointer over a window's border. The effect of setting this bit is that you can get a system pop-up menu *only* for the selected window; you can't get a menu for an unselected window.

Table 5-3. Interactive User Interface, Con't

Bit(s)	Description
0x0040000	If this bit is set, window system pop-up menus are disabled over the screen desk top—you can't get a pop-up menu by clicking the pointer over the desk top dither pattern. When this bit is set, you can get pop-up menus *only* in a window's border.
0x0080000	Setting this bit causes windows to be unselected when changed to iconic representation.
0x0100000	By default, the iconic representations of windows are placed in the lower-left quadrant of the display screen. By setting this bit, you can cause icons to be placed in the upper-right corner of the display by default.
0x0200000	Some display hardware, such as the 98700H Display Station, has the capability to do faster screen updates when the updated information is aligned along so-called **tile boundaries**. For example, a window move operation may occur more quickly to an even pixel address than an odd pixel address. By default, interactive move operations are optimized in this manner. When you interactively move or size a window, the window might not be moved or sized exactly as you specified, although it will be extremely close—the difference, if any, is practically unnoticeable. If you wish to turn off tile boundary alignment on your system, set the bit described here. Doing this will cause interactive move and size operations to be placed exactly as you specify, but performance may decrease.
0x0400000	If set, this bit allows the pointer to move outside a pop-up menu's boundaries without aborting the menu.
0x0800000	If this bit is set, then changing a window from iconic to normal representation causes the window to be selected.
0x1000000	To speed up terminal window creation via the pop-up menu, Windows/9000 keeps a terminal window cache. This cache contains the next terminal window to create via the pop-up menu. The window is not displayed, and you cannot use it until you officially create it via the *Create Window* option of the pop-up menu. The disadvantage of the terminal window cache is it consumes window resources. For example, if you run applications that don't even require terminal windows, this option is somewhat wasteful. To disable the terminal window cache, set this bit.

Table 5-3. Interactive User Interface, Con't

Bit(s)	Description
0x2000000	By default, the window system allows you to create windows simultaneously. For example, you can create a window via the pop-up menu, and possibly before the window appears, you can create another. By setting this bit, you disable simultaneous window creation. When this bit is set, the *Create Window* item of the pop-up menu will be "greyed" whenever the system is busy creating a window.
0x4000000	If this bit is set, audio feedback from interactive operation errors is disabled. In other words, set this bit, and you won't get the window system beep.
0x8000000	If set, then the direction of panning via the arrows in the scroll bars is reversed. This is effective for both graphics and term0 windows.

The Default Value

As mentioned previously, this value is not set by *wmstart*. In this case, the window manager uses the default value `0x080781`. This gives the default characteristics:

- button one is enabled—i.e., is the select button
- the [Select] key is enabled on the ITF keyboard
- window's are automatically made the top window in the display stack under any of the following circumstances:
  - when the window is selected
  - when the window is changed to an icon (or vice versa)
  - when the window is moved or its size changed such that it is unobscured by any other window/icon
- all control boxes are enabled
- pop-up menus are enabled over the desk top and window borders
- the window is unselected when changed to an icon
- icons are placed at the lower-left corner of the display screen
- tiler alignment is enabled for optimum performance
- moving the pointer outside a menu's border aborts the menu

- the cache of pop-up terminal windows is enabled
- windows can be created simultaneously
- the system beeps if you make an error when using windows interactively
- windows scroll in the direction of the arrows.

Examples

The following sets WMIUICONFIG to the default value, except that the pointer can be moved outside a menu's border without aborting the menu (0x0400000), and the beeper is disabled (0x4000000):

```
WMIUICONFIG="0x4480781"
```

The next example sets WMIUICONFIG to the default value, except that pop-up menus are disabled over a window's border, and the control boxes are also disabled; in other words, only the scroll arrows work in the window's border:

```
WMIUICONFIG="0x00af781"
```

Default Fonts

By setting the value of certain environment variables, you can alter the default fonts used for the following:

- items in pop-up menus
- the label in window borders
- the icon label area
- softkey labels
- base and alternate fonts for terminal window text.

Note

You should read the "Managing Terminal Window Fonts" section of the "Using Commands" chapter before proceeding with this section.

The Font Directory - WMFONTDIR

The WMFONTDIR variable specifies the path name of the main font directory, the directory under which all font directories and font files are stored. By default, this variable is set to `/usr/lib/raster` by the *wmstart* shell script.

If you set this variable to an invalid value—i.e., a path name under which no fonts can be found—then some commands, such as *wfont(1)* may not work properly.

Base and Alternate Fonts - WMBASEFONT and WMALTFONT

The WMBASEFONT and WMALTFONT variables specify the path names of the default base and altenate fonts to use in newly created terminal windows. These variables are not set by *wmstart*. The default base and alternate fonts used depend on the display screen's resolution and the value of the $LANG environment variable.

$LANG Not Set

By default, $LANG is not set to any value. In this case, the default base and alternate fonts depend solely on the display screen resolution. High-resolution displays use $WMFONTDIR/`8x16/lp.8U` as the base font, and $WMFONTDIR/`8x16/lp.b.8U` as the alternate font. Low-resolution displays use $WMFONTDIR/`6x8/lp.8U` as the base font, and $WMFONTDIR/`6x8/lp.b.8I` as the alternate font.

$LANG Set

When using Windows/9000 on a system other than USASCII nationality, the $LANG environment variable will typically be set to the language of the attached keyboard. For example, when using Windows/9000 in Japan, $LANG would most likely be set to "japanese".

When $LANG is set, default fonts are found under the `/usr/lib/raster/dflt` directory. Under this directory are two directories: `b` for base fonts, `a` for alternate fonts. Under each of these are two more directories: `h` for high-resolution displays, `l` for low-resolution displays. Under each of these directories is a file named after the current language, as defined by the $LANG environment variable.

For example, suppose your system supports Japanese fonts and the $LANG environment variable is set to "japanese". Then the default base font on a high-resolution display is `/usr/lib/raster/dflt/b/h/japanese`, which is linked to the font file `/usr/lib/raster/8x18/kanji.16K`. The default alternate font is `/usr/lib/raster/8x18/kana.8K`.

Pop-Up Menu Font - WMMENUFONT

The WMMENUFONT variable defines the font to use in pop-up menus. This variable is not set by *wmstart.*

If this variable is not set, the window manager uses a default font based on display screen resolution and the value of the $LANG environment variable (see discussion of WMBASEFONT and WMALTFONT).

The Window Border Font - BANNERFONT

The BANNERFONT variable defines the font used for the window label in window borders. This variable is not set by *wmstart.*

If not set, the window manager uses $WMMENUFONT for window labels.

The Icon Label Font - ICONFONT

The ICONFONT environment variable specifies the path name of the font used in an icon's label area. This variable is not set by *wmstart.*

If this variable is not set, the window manager uses a default font based on the display screen's resolution and the value of the $LANG environment variable (see discussion of WMBASEFONT and WMALTFONT).

The Softkey Label Font - WMSFKFONT

The WMSFKFONT variable specifies the path name of the font to use in softkey labels. This variable is not set by *wmstart.*

If this variable is not set, the window manager uses a default font based on the display screen's resolution and the value of the $LANG environment variable (see discussion of WMBASEFONT and WMALTFONT).

Examples

To set the pop-up menu font to 8-by-16-pixel italic line printer font, set WMMENUFONT as follows:

```
WMMENUFONT="/usr/lib/raster/8x16/lp.i.8U"
```

To set the softkey label font to 18-by-30-pixel courier, set WMSFKFONT as:

```
WMSFKFONT="/usr/lib/raster/18x30/cour.OU"
```

Changing the Desk Top Dither Pattern - WMDESKPTRN

You can use one of five different dither patterns for the window system desk top area. These patterns are specified via the WMDESKPTRN environment variable.

What Is a Dither Pattern?

Dither patterns are created by displaying dots in a consistent pattern on the display screen. Different patterns produce different shading effects. By default, the window system displays black dots on a white background, thus producing a desk top pattern that ranges from white to black, depending on the dither pattern used.

Note that you can change the default dither pattern foreground and background colors via window system environment variables discussed in the next section, "Default Colors."

The WMDESKPTRN Variable

The WMDESKPTRN environment variable is used to specify a new dither pattern. Valid values for this variable are: 0, 25, 50, 75, 100. Each number represents a specific dither pattern. Figure 5-6 shows four of the five dither patterns along with the associated value. (Dither pattern zero is not shown because, by default, it is solid white which doesn't show up on paper.)

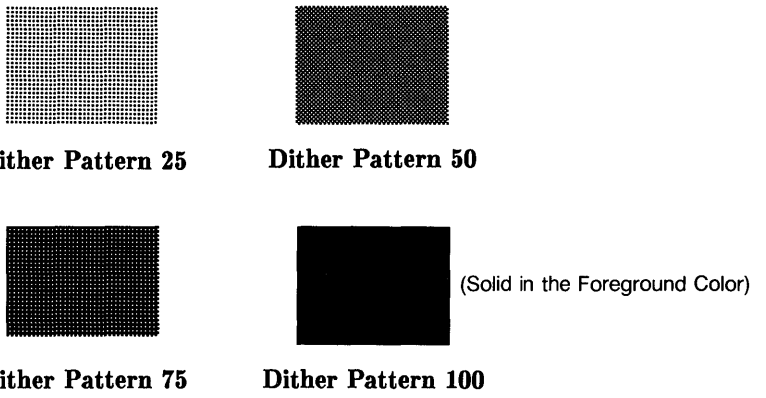

Figure 5-6. Dither Patterns.

Note

People who use color displays will typically set WMDESKPTRN to 100; then pick a pleasing background color. This way the entire desk top is solid in the color. Alternatively, WMDESKPTRN values other than 100 may produce interesting "blends" as well.

Setting WMDESKPTRN

By default, WMDESKPTRN is not defined when the window system starts executing. When WMDESKPTRN is undefined or null, the window manager assumes the dither pattern to be 50.

If you set WMDESKPTRN to a value other than 0, 25, 50, 75, or 100, the window manager rounds it to the nearest significant value.

Default Colors

Through window system environment variables, you can change default colors for the following:

- desk top foreground and background colors
- initial border foreground and background colors used in newly created windows

Table 5-4 shows the default color map entries.

Table 5-4. Default Color Map Entries

Value	Color
0	black
1	white
2	red
3	yellow
4	green
5	cyan
6	blue
7	magenta

Changing Desk Top Colors – WMDESKFGCLR and WMDESKBGCLR

The WMDESKFGCLR and WMDESKBGCLR variables, respectively, are used to change the default desk top dither pattern's foreground and background colors.

The *wmstart* shell script does not set these values. If a variable is not set (or null), the window manager assumes the foreground color to be 0 (black in the default system color map) and the background color to be 1 (white in the default system color map).

Window Border Colors – WMBDRFGCLR and WMBDRBGCLR

The WMBDRFGCLR and WMBDRBGCLR variables, respectively, are used to change the window border foreground and background colors for newly created windows.

wmstart does not set these variables. If a variable is not set (or null), the window manager assumes the foreground color to be the same as $WMDESKFGCLR, and the background color the same as $WMDESKBGCLR.

Interactive Timeout and Tracking - WMIATIMEOUT

The WMIATIMEOUT environment variable serves two purposes:

- It specifies the timeout period (in seconds) for interactive operations.
- It determines the number of milliseconds the window manager will not process locator changes. This allows other processes to run when the window manager is tracking.

Following are detailed descriptions for using WMIATIMEOUT for either purpose.

Specifying Timeout

The two least-significant bytes of this variable (`0xFFFF`) specify the number of seconds of absolute inactivity in the locator that the window manager will allow during an interactive operation. In other words, if you start an interactive operation, such as moving or sizing a window, the operation will be aborted after the specified number of seconds *if* there is no activity in the locator during this time.

By default, WMIATIMEOUT is not set by *wmstart*. If this value is not set, or is less than or equal to zero or null, then the window manager assumes a timeout period of 60 seconds. That is, an interactive operation will be aborted if the locator remains inactive for 60 seconds.

Note that the maximum timeout period that can be specified is `0xffff`—65 535 seconds (approximately 18 hours and 12 minutes).

Tracking

The window manager reads information from the locator device **whenever its position changes**; this is known as **tracking**. If the position changes frequently, the window manager (because it is a high-priority process) may use nearly all the CPU, thereby causing other processes to run slowly.

The Problem

If one or more other processes are also trying to track locator movement, their tracking becomes jerky or stops because they are only able to read locator information when the window manager has stopped tracking. Figure 5-7 illustrates this problem.

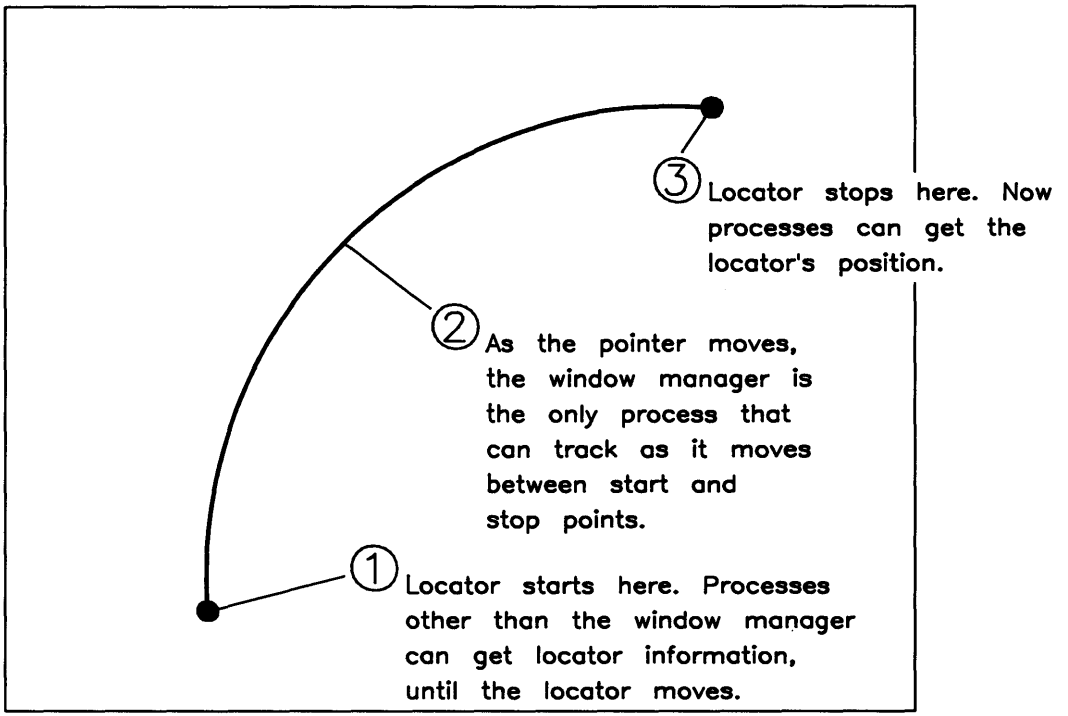

Figure 5-7. The Locator Tracking Problem

The Solution

To more fairly allow other processes to run, the window manager temporarily ignores the locator for a short period of time, thus allowing the other processes to run. Figure 5-7 illustrates this solution to the problem.

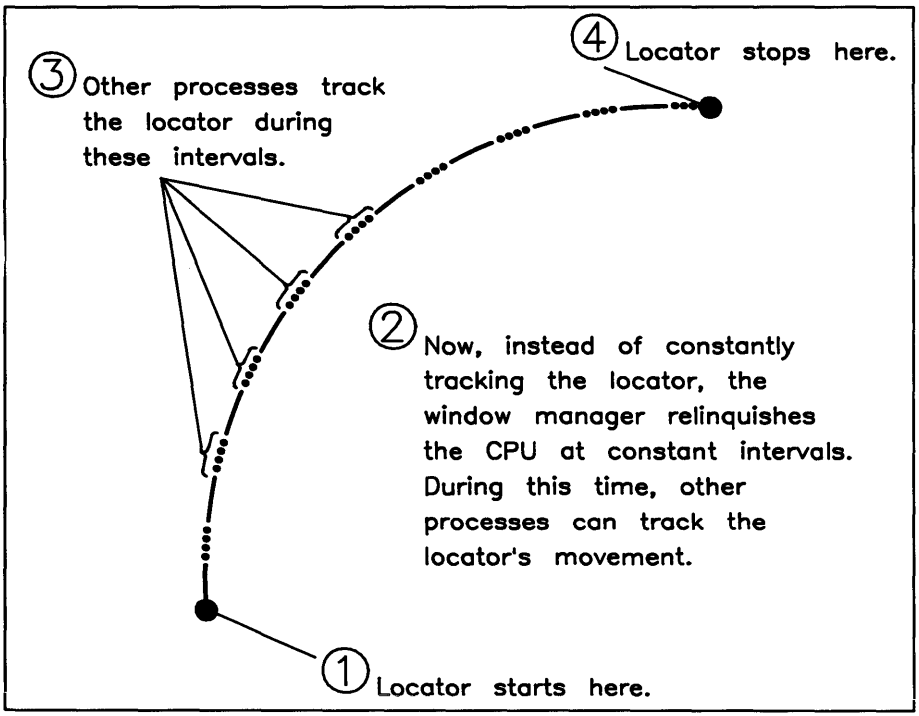

Figure 5-8. Solution to the Tracking Problem

Specifying Tracking Timeout Period

The length of time the window manager ignores locator input is called the **tracking timeout period**. Windows/9000 allows you to change the tracking timeout period via the WMIATIMEOUT variable.

Values are specified in the third byte of this environment variable (`0xFF0000`) and range anywhere from 0 to 255 milliseconds.

If no value is specified or 0 is specified, then the window manager uses 30 milliseconds for the tracking timeout period. If the value is 255, then the window manager does not ignore tracking information (i.e., the tracking timeout period is set to 0).

Changing the tracking timeout period has the following effects:

1. As this number becomes lower, the echo tracks better on the screen, but user processes won't track as well;
2. As this number becomes higher, the echo tracks worse on the screen, but user processes track the locator at least as well as the window manager.

Note

Although time can be specified in one-millisecond increments, the Series 300 clock "clicks" every twenty milliseconds. Therefore, you may want to specify time in 20-millisecond increments on Series 300.

Examples

The following example changes the interactive timeout period to 15 seconds, but the tracking timeout period is left as the default value (the value is specified in hexadecimal):

```
WMIATIMEOUT=0x0f
```

The value shown here was specified in hexadecimal; however, you can specify values in decimal. For example, the above would be specified as:

```
WMIATIMEOUT=15
```

The next example sets the tracking timeout period to 40 milliseconds (`0x28`) but leaves the interactive timeout period unchanged. The net effect is that user processes track better, but the window manager tracks worse than normal:

```
WMIATIMEOUT=0x280000
```

This last example sets the interactive timeout period to 30 seconds (`0x1e`) and the tracking timeout period to 32 milliseconds (`0x20`):

```
WMIATIMEOUT=0x20001e
```

Changing Window Server Priority – WMRTPRIORITY

Windows/9000 makes extensive use of servers, special processes that facilitate communication between *ptys* and device special files. The window manager is a server, and each window has an associated server. (The server for terminal windows is named `/usr/lib/tOserver`; the server for graphics windows, `/usr/lib/gserver`.)

Because a server is a process, you can change its real-time priority. The WMRTPRIORITY environment variable is used to reassign the real-time priority for the window manager and servers.

Setting WMRTPRIORITY

The least-significant byte of WMRTPRIORITY (`0x00ff`) gives the real-time priority for all window servers; the next byte (`0xff00`), the real-time priority for the window manager.

Valid values range from 0 (the highest priority) to 127 (lowest priority). If a value is specified out of range, real-time priority is disabled.

Default Value

By default, *wmstart* does not set WMRTPRIORITY, and the window manager assumes a value of `0x787c`: window manager server priority is 120; window servers' priority is 124.

IMPORTANT

Only the system administrator should modify this variable. Do *not* change this variable unless you absolutely understand the consequences. You might change this variable if you're writing real-time applications with windows.

If you do change this variable, use the highest number (lowest priority) that will achieve the results you need. The use of numbers which are too low can interfere with the proper execution of the HP-UX operating system.

Windows/9000 Shared Memory

All processes associated with Windows/9000 share a contiguous section of memory through which interprocess communication is facilitated. In addition, this shared memory contains retained rasters for graphics windows. Two window system environment variables—SB_DISPLAY_ADDR and WMSHMSPC—can be used to move the location and change the size of shared memory.

IMPORTANT

The purpose of this section is simply to give a brief overview of these variables. You should read the "Shared Memory Usage" section of the "Window Limitations" appendix for details on how these variables are used.

Controlling Shared Memory Location - SB_DISPLAY_ADDR

The SB_DISPLAY_ADDR variable controls the location of windows shared memory. You should refer to the "Shared Memory Usage" section of the "Window Limitations" appendix for information on setting this variable.

By default, *wmstart* does not set SB_DISPLAY_ADDR, and the window manager assumes an address of `0xb00000` (at 11Mb).

Setting the Size of Shared Memory - WMSHMSPC

The WMSHMSPC variable controls the size of the window shared memory space. You should refer to the "Shared Memory Usage" section of the "Window Limitations" appendix for details on setting this variable.

By default, WMSHMSPC is set to `0x200000` (2Mb) by the *wmstart* shell script.

Changing Window Manager Configuration – WMCONFIG

The WMCONFIG variable controls window system memory locking, determines how graphics windows are cleared when created, and enables/disables window system double buffering color mode. This variable greatly affects window system performance. You should understand the shared memory concepts presented in the "Window Limitations" appendix before proceeding with this section.

Window Manager Process Locking

The three least-significant bits of WMCONFIG (0x7) control window manager process locking. If the 0x1 bit is set, the window manager's *text* (*code*) area is locked into memory. If the 0x2 bit is set, the window manager's *data* area is locked into memory. If the 0x4 bit is set, then window system shared memory is locked into memory. Bits can be set in any combination.

When memory is locked, it cannot be swapped out to accomodate other processes' memory needs: the text, data, or shared memory stays in physical memory until the window manager terminates. Therefore, locking window manager memory will improve window system performance.

This capability is typically useful on systems with large amounts of physical memory when the window system competes with very large processes for memory. Setting these bits will improve window system performance.

IMPORTANT

If shared memory locking is enabled, then setting WMSHMSPC to the minimum possible value is advisable.

Clear of Retained Rasters

Set the `0x8` bit in WMCONFIG to unconditionally clear a retained graphics window's raster when the window is created. The default (not set) is much faster because the clear is delayed until the window is made visible or graphics is done in the window.

This WMCONFIG bit provides compatibility for old programs having the following attributes:

- The program was linked prior to HP-UX 5.2 (has not been re-compiled since a pre-5.2 release).
- The program draws in a concealed window that it did not create.
- The program depends on a window being cleared prior to the window being opened by *gopen(3G)*.

Double Buffering Color Mode

If the 0x10 bit in WMCONFIG is set, the window system will use double buffering color mode. Double buffering color mode will cause all colors used in window borders, window icons, softkeys, desktop, popup menus, and term0 windows, to be modified so that the visible color for these will not change whenever the display enabled planes are modified by a Starbase program using double buffering. All colors written to the display are first converted to (C << (N/2) + C), where C is the color being written and N is the number of planes on the display. The window manager will modify the color map so that color indices (C << (N/2)) and (C << (N/2) + C) have the same RGB values as color index C.

Enabling double buffering color mode reduces the effective number of colors from 2^N to 2^(N/2). The window system will force all colors set via environment variables to be within the allowed range.

Since the window manager sets up the color map so that it is appropriate for use with double buffering, Starbase programs should never modify the color map. This means that INIT should not be specified in the **mode** argument of **gopen**.

If a Starbase program is not going to use double buffering, but expects to be able to run in a window when some other program is doing double buffering, it should modify the colors it uses in the same way the window system modifies its colors.

Note that only one Starbase program at a time can make effective use of double buffering inside the window system. If more than one program tries to double buffer, the planes that will be visible for **all** of the programs will be the planes enabled by the last program to do a `dbuffer_switch`. By convention, a program should do a `dbuffer_switch` only when the window it is displaying to is the selected window.

Double buffering color mode may only be enabled on displays with six or more planes.

Software Sprites Mode

If the `0x20` bit in WMCONFIG is set, the window system forces the use of software sprites (pointers), which implies that, on displays which provide hardware support for sprites, the hardware will not be used for window system sprites. So, when the `0x20` bit in WMCONFIG is set, all sprites are rendered using software. The default of 0 provides better performance, but at the cost of not being able to represent the full range of raster echo sprites. The ability to use software sprites is for compatibility purposes in the case of programs that use sprites that have more than two color index values, and require that those sprites be displayed exactly as defined, or in the case of an application which requires exclusive use of the hardware sprite.

On the HP 98730 display, the hardware is capable of displaying only two colors for sprites. However, raster echoes may contain color index values up to eight bits in depth (0–255). When the window system is using the hardware for sprites (the default), a conversion must take place to convert the raster echo from *n* colors to two colors. This conversion causes all background-color index values in the raster echo to be displayed as the background color, and all non-background-color index values are displayed as the foreground color. When the window system is using software sprites, the raster echo is displayed as specified.

The window's border sprite colors are set, respectively, to the border's foreground and background colors. For all other window sprites, if no call has been made to `wset_hw_sprite_color`, the defaults are a foreground color of 1 and a background color of 0 for all other sprites.

Default Value

By default, *wmstart* does not set WMCONFIG. When WMCONFIG is not set (or null), the window manager uses a default value of `0x0`. That is, window manager process locking is disabled, higher-performance delayed clear of graphics windows is used when graphics windows are created, and double buffering color mode is disabled.

Window Limitations A

The following resource restrictions may cause problems when using the window system:

- process limits
- pseudo-tty (*pty(4)*) limits
- maximum number of open files
- window shared memory
- configuring swap space
- good-citizen processes.

Process Limits

One of the first limits encountered when creating windows is the **process limit**. By default, the kernel allows up to 25 processes per each user logged in to the system. As windows are created, and as the user runs more applications in the windows, processes are consumed. If 25 processes are already consumed, and the user attempts to create a new window or to run another application, then the window system will fail.

Process Usage

The window manager is a process, charged to the user who starts the window system. In addition, whenever a window is created, the window manager spawns a temporary process, which exists until the window is created. Finally, the window system is started from the user's shell, another process. So before any windows are even created, three processes are consumed, leaving 22 to be consumed by windows and applications:

```
   25 (maximum processes to start with)
-   1 (temporary process used to create a window)
-   1 (for the login shell)
-   1 (for the window manager)
=====
   22
```

Each term0 window has a server (`/usr/lib/t0server`), which is a process charged to the user of the window system. Graphics windows do not use any servers.

Also consider that any program or shell consumes a process. For example, a term0 window with a shell which is running the *ls* command consumes three processes: one for the server, one for the shell, and one for the *ls* command when it is running.

The *ps* command can be used to determine how many process the user of the window system is consuming. See the *ps(1)* page of the *HP-UX Reference* for details on using this command.

Examples

How many term0 windows can be created if each window contains a shell and an application? Each window consumes three processes (server, shell, application), and 22 processes are available, so seven windows can be created ($22 \div 3 = 7.333 = 7$).

On the other hand, a graphics program that creates two graphics windows uses only one process, because graphics windows do not consume processes, per se. So, theoretically, 22 such applications could run, handling 44 windows (22 processes $\times$ 2 windows apiece = 44 windows). However, 44 windows is not attainable with the default configuration for the window system; other limitations, discussed in subsequent sections, will be reached before 44 windows can be created.

Getting Around the Limit

For some users, this limitation may not pose a problem. If it does for you, then you must reconfigure the kernel to allow more than 25 processes per user. Reconfiguring the kernel should be performed **only by your system administrator**. The process limit can be reconfigured via the `maxuprc` kernel configuration parameter. Refer to your *System Administrator Manual* for details on reconfiguring this limit.

Pseudo-Terminal (pty) Limitations

The window system uses pseudo-terminal (*pty*) special files extensively. As more and more windows are created, *ptys* from the `/dev/ptym` and `/dev/pty` directories may be used up to the point where no more windows can be created.

Pty Usage

The window system consumes *ptys* as follows:

- the window manager itself uses one *pty*
- each term0 window requires three *ptys*
- each graphics window requires one *pty*.

The window manager consumes *ptys* from a contiguous set (pool) of *ptys*. This size and location of the pool of *ptys* is defined by *pty* environment variables, listed below.

WMPTYMDIR

Each *pty* is composed of a master and slave side; together they make a single *pty*. This variable gives the path name of the directory containing the master sides for each *pty*.

By default, WMPTYMDIR is not set, and the window manager assumes the master *pty* directory is `/dev/ptym`.

WMPTYSDIR

This variable specifies the path name of the directory containing the slave sides for each *pty*.

By default, WMPTYSDIR is not set, and the window manager assumes the slave *pty* directory is `/dev/pty`.

WMPTYNAME

The window manager must know where the pool of *pty* starts in the $WMPTYMDIR and $WMPTYSDIR *pty* directories. The WMPTYNAME variable specifies the name of the first *pty* in the contiguous set (pool) to use from these directories. In other words, WMPTYNAME points to the start of the pool of *ptys* used by the window manager.

The *pty* name must follows the naming convention: tty[p-v][0-f]. For example, the following is a contiguous set of 10 *ptys* starting at `ttyp8`:

```
ttyp8
ttyp9
ttypa
ttypb
ttypc
ttypd
ttype
ttypf
ttyq0
ttyq1
```

By default, WMPTYNAME is not set, and the window manager assumes the starting *pty* is `ttyp8`.

WMPTYCNT

The WMPTYCNT variable specifies the size of the *pty* pool, that is, the number of contiguous *ptys* that the window manager is allowed to use. The window manager cannot use more than this number of *ptys*. If WMPTYCNT is set to a value exceeding 128, the window manager uses 128. The minimum allowable value is 4.

By default, WMPTYCNT is not set, and the window manager assumes the pool size is **31**.

WMPTYCACHECNT

The WMPTYCACHECNT variable specifies the number of *ptys* that the window manager can keep pre-opened in a *pty* cache. Pre-opened *ptys* improve window system performance when creating new windows. This is because the *ptys* for new windows are already allocated and opened, so the window manager doesn't have to do this when the window is created. The *ptys* in the cache are a subset of those in the contiguous pool, described above.

When creating a graphics window, the window manager will attempt to use one *pty* from the cache; if the cache is empty, it then looks to remaining un-opened *ptys* in the pool defined by WMPTYMDIR, WMPTYSDIR, WMPTYNAME, and WMPTYCNT.

The same applies for term0 windows, except that term0 windows require *three ptys*. When a term0 window is created, the window manager attempts to use two *ptys* from the cache. It gets the remaining *pty*(s) from the pool.

By default, WMPTYCACHECNT is not set, and the window manager assumes the size of the cache is **10**.

The Limit

Based on the default limit of 31 usable *ptys*, as determined by the WMPTYCNT variable, the maximum number of *ptys* available for windows is 30: (31 *ptys* − 1 *pty* for the window manager = 30 remaining *ptys*).

Getting Around the Limit

To get around the limit established by *pty* environment variables, change them. You should refer to the "Environment Variables" chapter for details on how to change these variables.

Examples

Assuming that the process limit has been raised, then the maximum number of term0 windows that can be created is 10: 30 *ptys* ÷ 3 *ptys* per term0 window = 10 windows. The maximum number of graphics windows that can be created is 30: 30 *ptys* ÷ 1 *pty* per graphics window = 30 windows.

The above examples assume that you are either creating only term0 or only graphics windows. More realistically, you will probably have a mix of term0 and graphics windows. The number of term0 and graphics windows must conform to the following relation:

$$\langle \text{graphics windows} \rangle + 3 \times \langle \text{term0 windows} \rangle \leq \text{WMPTYCNT} - 1 \text{ (for } wm\text{)}$$

IMPORTANT

Nothing ensures that the *ptys* in the window manager's pool won't be used by some other application unrelated to the window manager. If window manager *ptys* are used by another application, then even fewer windows can be created. This is important only if some other application uses *ptys* from the same pool as the window manager.

Kernel pty Limitations

The default kernel limits the number of *ptys* in the system to 82. The `npty` kernel configuration parameter is used to change this default value. Only the system administrator should do this; see the *HP-UX System Administrator Manual* for details.

The set of *ptys* defined by `npty` starts at `ttyp0`. Therefore, `npty` should be large enough to accomodate the *pty* requirements of the window system and any other applications that use *ptys*. `npty` must be large enough to include all the *ptys* starting at `ttyp0` and ending at the last *pty* in the set of *ptys* defined by WMPTYNAME and WMPTYCNT.

Maximum Number of Open Files

Increasing the number of *ptys* by changing the *pty* environment variables still does *not* ensure that you can create as many windows as desired. This is because the HP-UX kernel imposes an unalterable limit of 60 open files (file descriptors) per process. This translates into an upper bound for the number of windows allowable per instance of the window manager.

File Usage

Three of the 60 available file descriptors are always used by the window manager: one for the window manager device interface, two more for the display screen special file. In addition, the window manager by default uses three more file descriptors for input devices. Thus, the maximum number of file descriptors available for windows is 54:

```
   60 (maximum allowable open files)
-   1 (for the window manager device interface)
-   2 (for the display screen special file)
-   3 (for opened input devices)
-----
   54
```

Each term0 window requires two open file descriptors inside the window manager; each graphics window requires one.

Examples

Given open file constraints, the maximum number of term0 windows that can be created is 27: 54 free file descriptors ÷ 2 file descriptors per term0 window = 27 term0 windows. The maximum number of graphics windows is 54: 54 free file descriptors ÷ 1 file descriptor per graphics window = 54 graphics windows.

More likely, graphics and term0 windows will be mixed. The following equation defines the maximum number of term0 and graphics windows, based on the file descriptor constraint.

$$\langle \text{graphics windows} \rangle + 2 \times \langle \text{term0 windows} \rangle \leq 54$$

Getting Around the Limit

You cannot circumvent this limit—the window manager cannot support more than 60 open files at a time. Note, however, that you will encounter process and *pty* limitations before this limitation.

Open File Limitations per User Process

Just as the window manager process has a hard limit of 60 open files, so does each of the user's processes. It is much rarer for this limit to impede a user's process, since a user process rarely needs to open sixty files, devices, and windows at the same time. Nevertheless, the possibility exists, especially if a process uses many graphics windows at once, or opens windows more than once.

User Process File Usage

The rules for determining user process file descriptor usage follow:

- Each *open(2)* costs 1 file descriptor (e.g., when opening a term0 window).
- Each *gopen(3G)* costs 1 file descriptor (e.g., when using *gopen* to open a graphics window).
- The first *gopen* call by a process costs 2 file descriptors, not just 1.
- Each *gopen* call on a graphics window temporarily costs 1 extra file descriptor, which is then freed shortly after the *gopen* call. This is very rare and will only present a problem if a process has 59 open files and attempts to call *gopen* on another window.
- Each file, device file, pipe, etc. uses 1 file descriptor per open.
- Standard input (stdin), standard output (stdout), and standard error output (stderr) typically use 3 file descriptors.

Shared Memory Usage

HP Windows/9000 on Series 300 has a virtual memory address space as shown in Figure A-1. Stack space starts near the top of virtual memory, and the stack grows down from the top. Code and data space start at the bottom (address 0) of virtual memory and grow up from the bottom. Located between these two areas is Windows/9000 **shared memory**. The "Memory Management" section of Chapter 3 of your *HP-UX System Administrator Manual* describes the organization of shared memory in detail.

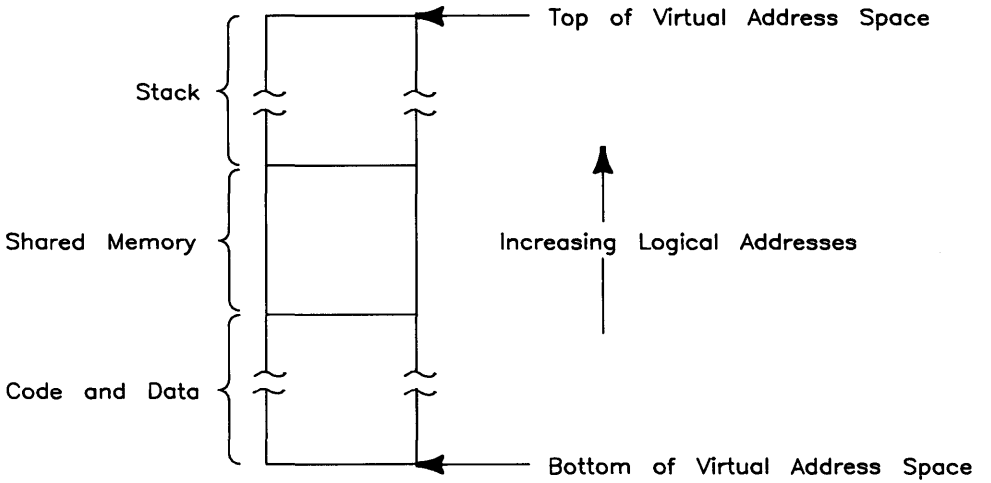

Figure A-1. Virtual memory map with Windows/9000

Window Processes

Before discussing why and how Windows/9000 shared memory is used, you must understand what a **window process** is. In general, a **window process** is any process that meets both of the following conditions:

1. The process is linked with the window library (*/usr/lib/libwindow.a*), and
2. The process performs Starbase graphics in a window **or** uses either of the following window library routines:
   - *wsetrasterecho(3W)*
   - *wgetrasterecho(3W).*

A process must strictly meet these requirements to be a window process; processes not meeting these requirements are not window processes and do not use windows shared memory.

The window manager, term0 window servers, and graphics applications that run in graphics windows are examples of window processes. In addition, any non-graphics application that uses *wsetrasterecho* or *wgetrasterecho* is also a window process.

How Do Window Processes Use Shared Memory?

All window processes related to a given instance of the window manager use this shared memory to access global data pertinent to their window, and to maintain other global data associated with the window system. For example, a graphics window with a retained raster keeps its raster in this shared memory so both the window manager and any process(es) doing output to the window can access the window's raster.

Shared Memory Problems

Two main problems are encountered with Windows/9000 shared memory; each problem is discussed in detail below:

- shared memory is too small
- code and data space is too small (shared memory is positioned too low in the virtual address space)

Shared Memory Size

Usually, the contiguous block of shared memory used by Windows/9000 is two megabytes (2Mb) in size. One problem encountered with shared memory is that it just isn't large enough for some applications.

When an application attempts to use more shared memory (e.g., for retained rasters for newly created graphics windows) than is available, the shared memory "get" fails and the application terminates. This does *not* affect the window system, except that you run into a limit for the number of retained-raster graphics windows you can create.

For example, suppose you have an application that uses many graphics windows with retained rasters. Retained rasters are costly in terms of memory usage. Each pixel of the raster consumes a byte of memory. Therefore, a window that is 1024 by 512 pixels consumes a half-megabyte (0.5Mb) of memory. At this rate, windows shared memory will be completely consumed by four such graphics windows (4 windows $\times$ 0.5Mb = 2.0Mb), and you may not be able to create the fourth (and any subsequent) window(s).

You can circumvent this problem via window system environment variables which control the size and location of shared memory. The use of these variables is discussed in the remaining sections of this appendix.

Code and Data Space

All window processes (code and data space) reside in the contiguous block of memory immediately below shared memory—the code and data space. By default the code/data space is 8.75Mb in size.

Because this shared memory space resides at the same address for all window processes, all window processes must have a code/data space that will fit into the area below shared memory. In other words, the shared memory must start at some address such that the largest code/data space of all window processes will not collide (interfere) with the shared memory space.

For example, suppose you've written a C program to perform graphics in a graphics window. The program does lots of program-controlled heap manipulation, so it must allocate (via *malloc(3)*) enormous amounts of dynamic data space. The program's code is 2Mb in length, but as it runs, it consumes up to 8Mb of dynamic data space (heap)—a total of 10Mb for both the code and its data. This will exceed the default maximum code/data space size by approximately 1.25Mb (10Mb − 8.75Mb max size = 1.25Mb over max).

This problem can be surmounted via window system environment variables. By moving shared memory upward, you create more room for the code/data space.

A Close-Up of Shared Memory

Before discussing how to change shared memory to circumvent shared memory problems, you should understand how the shared memory is organized. Figure A-2 shows the structure of this shared memory.

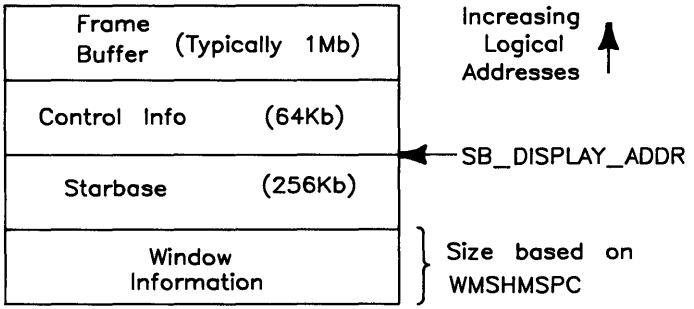

Figure A-2. Windows/9000 Shared Memory

Table A-1 briefly describes the four components of shared memory.

Table A-1. Windows Shared Memory Close-Up.

Component	Description
Frame Buffer	This is the area of memory that corresponds to the screen of your display. Each byte represents one pixel. Typically, this area is one to two megabyte in size.
Control Information	This area contains I/O device control information for the display hardware.
Starbase	This is used by the Starbase Graphics system.
Window Information	This area stores information pertinent to particular windows, such as retained rasters for graphics windows and fonts.

Shared Memory Environment Variables

Two environment variables—SB_DISPLAY_ADDR and WMSHMSPC—control the location and size of Windows/9000 shared memory. To configure the shared memory, you must change the value of the appropriate variable(s) **before** starting (or restarting) the window manager.

The SB_DISPLAY_ADDR Variable

SB_DISPLAY_ADDR points to the address immediately above the Starbase area of shared memory. Since the shared region is positioned relative to this address, you can change the position of the region by changing the value of SB_DISPLAY_ADDR. If you do not set the value of SB_DISPLAY_ADDR, it defaults to `0xB00000` (11Mb), which leaves approximately 8.75Mb for code/data space.

The WMSHMSPC Variable

The other environment variable for configuring shared memory is WMSHMSPC. This determines the size of the window information area of the shared region. The *wmstart(1)* shell script sets WMSHMSPC to `0x200000` (2Mb). However, if WMSHMSPC is undefined or null, the window manager assumes a shared memory space of `0x20000` (128K).

If WMSHMSPC is set to values less than `0x200000` (2Mb), then only one segment of the size specified by WMSHMSPC (rounded to the next page boundary) is allocated. If WMSHMSPC is set to values greater than 2Mb, then the window manager gets 2Mb initially, but waits until the window space becomes full before allocating additional memory up to the value of WMSHMSPC. Table A-2 shows how segments are allocated for various values of WMSHMSPC.

Table A-2. Example WMSHMSPC Values.

WMSHMSPC	Resulting Segments
0x20000 (128Kb)	one 128Kb segment
0x100000 (1Mb)	one 1Mb segment
0x200000 (2Mb)	one 2Mb segment
0x300000 (3Mb)	one 2Mb segment and one 1Mb segment
0x400000 (4Mb)	two 2Mb segments
0x500000 (5Mb)	two 2Mb segments and one 1Mb segment

Note: The window manager is limited to allocating shared memory segments no more than 2Mb in size. Therefore, you cannot create a window with a retained raster larger than 2Mb.

Changing Shared Memory

If you decide that the default configuration for shared memory, as defined by the environment variables SB_DISPLAY_ADDR and WMSHMSPC, is inadequate for your system, then you should set these variables accordingly. In general, you should use the following rules when reconfiguring windows shared memory:

- If you have processes requiring more shared window space than the default 2Mb, then you should increase the value of WMSHMSPC to accomodate the amount of shared memory required. (Note, however, that a single window requiring more than 2Mb of memory cannot be accomodated even by increasing the value of WMSHMSPC.)
- If you have processes that require more code and data space than the default 8.75Mb, then increase the value of SB_DISPLAY_ADDR so the processes will fit in the code/data space.

Side Effects from Changing Variables

The WMSHMSPC and SB_DISPLAY_ADDR environment variables are closely related; changing one may affect the other. Before changing a variable, you should understand the possible side effects.

WMSHMSPC

If you increase the value of WMSHMSPC but do *not* change the value of SB_DISPLAY_ADDR, you effectively decrease the size of the code/data space by the increased size of the shared area. This is because the window information area (defined by WMSHMSPC) resides below SB_DISPLAY_ADDR (see Figure A-2); therefore, increasing the size of this area while holding SB_DISPLAY_ADDR constant shrinks the code/data space (see Figure A-3).

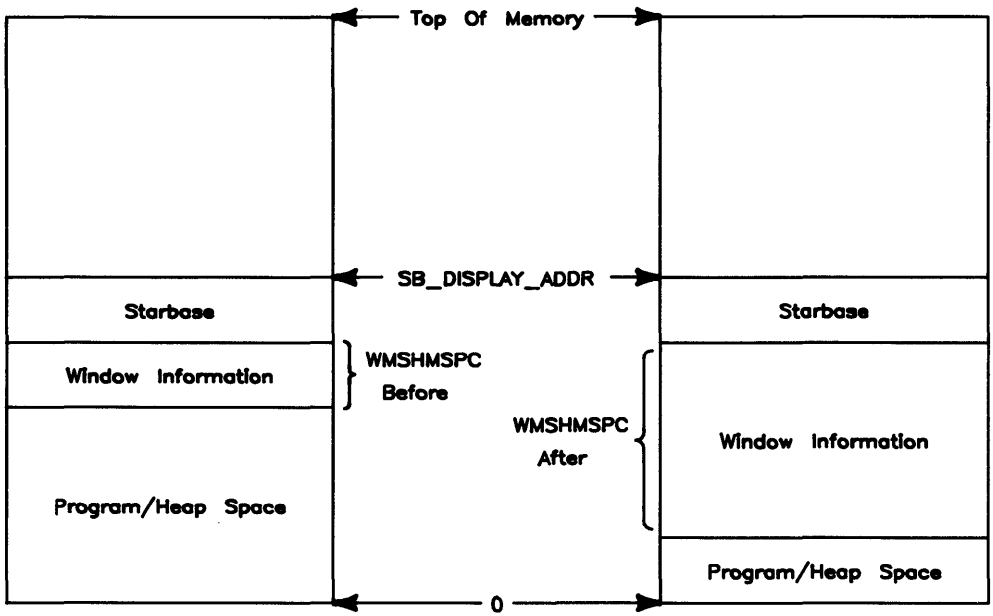

Figure A-3. Squeezing Code/Data Space via WMSHMSPC

Conversely, if you decrease the size of shared memory while holding the SB_DISPLAY_ADDR variable at a constant value, the code/data space will increase by the change in WMSHMSPC.

Equation A-1 shows the relationship between code/data size and the variables WMSHMSPC and SB_DISPLAY_ADDR.

```
code/data space = SB_DISPLAY_ADDR - 256Kb (for Starbase) - WMSHMSPC
```

Equation A-1. Computing Code/Data Space.

If you do not want the Code/Data space to change when you change the value of WMSHMSPC, you must change SB_DISPLAY_ADDR likewise. For example, if you increase WMSHMSPC from the default 2Mb to 3Mb (an increase of 1Mb), you must also increase SB_DISPLAY_ADDR from the default 11Mb to 12Mb.

SB_DISPLAY_ADDR

Whenever you increase the value of SB_DISPLAY_ADDR, you decrease the stack space available to window processes. Stack space size is given by equation A-2.

```
stack space = highest memory address - SB_DISPLAY_ADDR
              - frame buffer size - 64Kb (control info)
```

Equation A-2. Computing Stack Space.

Most of the time, you needn't worry about consuming too much stack space. However, keep in mind that the Series 300/Model 310 has a 16Mb address space; other Series 300 Models have theoretically a four-gigabyte address space. Just be sure that if you change SB_DISPLAY_ADDR, you leave enough room for the process with the largest possible stack.

Kernel Configuration Limitations

Even though Windows/9000 allows you to move and change the size of shared memory, there are still some limitations to its location and size. These limitations are defined by kernel configuration parameters. Consult your *HP-UX System Administrator Manual* for details on setting and changing these parameters.

The shmmaxaddr Variable

The maximum (highest) address allowable for any shared memory is defined by the `shmmaxaddr` kernel configuration variable. By default `shmmaxaddr` is defined as `0xFFFFFF` (16Mb). This means the topmost address of Windows/9000 shared memory cannot exceed the value of `shmmaxaddr` (`0xFFFFFFF`), as shown in Figure A-4.

Equation A-3 gives the relationship between `shmmaxaddr` and the SB_DISPLAY_ADDR variable:

```
SB_DISPLAY_ADDR ≤ shmmaxaddr - stack space under shmmaxaddr
                  - frame buffer size - 64Kb (control info)
```

Equation A-3. Relationship between SB_DISPLAY_ADDR and shmmaxaddr.

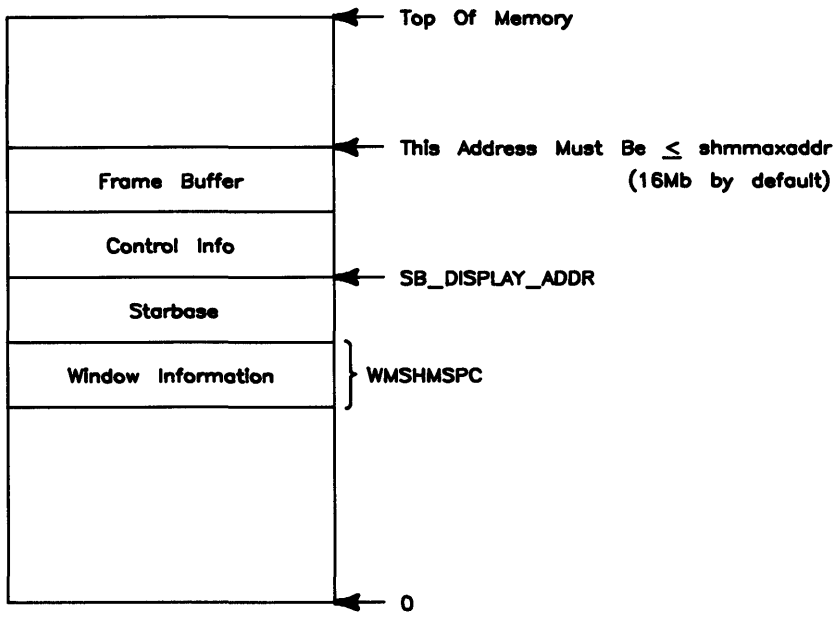

Figure A-4. The shmmaxaddr Variable and Shared Memory

The Series 300/Model 310 address space cannot exceed 16Mb anyway, so shared memory on these systems can be configured anywhere within the address space. However, on other Series 300 Models, which potentially have four gigabytes of address space, **shmmaxaddr** does pose a limitation; if you wish to move shared memory higher than 16Mb on Series 300 machines other than the Model 310, reconfigure the **shmmaxaddr** variable.

The shmmax Variable

The window manager *cannot* work with memory segments other than 2Mb in size. The **shmmax** kernel configuration variable defines the maximum allowable size of shared memory segments. By default, this value is set to 0x400000 (4Mb). This does *not* mean shared memory is limited to 4Mb maximum; it merely means each *segment* is limited to 4Mb.

You can set **shmmax** to values other than 4Mb, but the window manager will still work only with 2Mb segments. If you set **shmmax** to a value less than 2Mb, the window manager will fail when it attempts to allocate a 2Mb segment. Therefore, you should **never set shmmax to values less than 2Mb.**

Example

The following example should assist you in understanding how to use these variables: Suppose you've written two C-language programs to perform Starbase graphics with graphics windows. Program one (*prog1.c*) will require large amounts of code/data space when it executes—greater than the default amount. Program two (*prog2.c*) won't use as much code/data space as *prog1.c*, but makes heavy use of retained-raster graphics windows, and requires more window information area than the default. Detailed descriptions of each program follow.

Program One (*prog1.c*)

When compiled, *prog1.c*'s executable code size is just under 2Mb (say, `0x1e0f3b`). For simplicity, round the code size to 2Mb (`0x400000`).

When the program executes, it allocates large amounts of dynamic data space (via *malloc(3)*). Through diligent calculations you've determined that the program could theoretically allocate up to 6.5Mb of memory from the dynamic data space (heap). Again, for simplicity, round this figure to 7Mb (`0x700000`).

The maximum code/data space consumed by this application is 9Mb, computed as follows:

```
  0x200000 (2Mb for executable code)
+ 0x700000 (7Mb for the program's data)
==========
  0x900000 (9Mb total program and data space)
```

Program Two (*prog2.c*)

When running, *prog2.c* will create a maximum of four graphics windows, each with a retained raster with dimensions 1024 by 512 pixels. The retained raster for each window consumes 0.5Mb; therefore the maximum amount of memory required by the retained rasters alone is 2Mb (4 windows at 0.5Mb each = 2Mb).

When determining the size of the window information area, you should also consider that other windows will use the area also. For example, fonts for all windows are loaded into the window information area. For safe measure, you might typically add 1Mb to the size of the window information area to compensate for other windows' needs.

For this example, it gives a maximum size of 3Mb for the window information area—1Mb larger than the default value for WMSHMSPC.

Required Stack Space

None of the processes in this hypothetical system will require more than 0.5Mb of stack space.

Determining the Correct Value for SB_DISPLAY_ADDR

Now you know the maximum code/data space size (9Mb) and the maximum size of the window information area of shared memory (WMSHMSPC = 3MB), so you can determine the correct value for SB_DISPLAY_ADDR.

SB_DISPLAY_ADDR should be computed as follows:

```
SB_DISPLAY_ADDR = code/data size + WMSHMSPC
                        + 256Kb (for the Starbase area)
```

Equation A-4. Determining SB_DISPLAY_ADDR.

Using this equation, SB_DISPLAY_ADDR is computed to be 14.25Mb:

```
   0x900000 (9Mb code/data space for prog1.c)
+  0x300000 (3Mb window information area — WMSHMSPC)
+  0x040000 (256Kb for the Starbase area)
===========
   0xc40000 (12.25Mb = SB_DISPLAY_ADDR)
```

Ensuring Correct Values

Before actually setting these values, you should be sure the values will not cause detrimental side effects; mainly, there must be enough stack space for all window processes to execute, and shared memory must be within the **shmmaxaddr** value.

First, using Equation A-2, compute the stack space based on the current SB_DISPLAY_ADDR value; assume you have a 16Mb address space:

```
   0xffffff (highest address with 16Mb)
-  0xc40000 (SB_DISPLAY_ADDR)
-  0x200000 (size of frame buffer area)
-  0x010000 (control info area)
===========
   0x0a0000 = 0.625Mb of stack area
```

SB_DISPLAY_ADDR passes the first test: 0.625 is greater than the 0.5Mb of stack space requirement.

Next, using equation A-3, ensure that the window shared memory area is within the bounds set by the **shmmaxaddr** kernel configuration variable:

```
   0xffffff (default value for shmmaxaddr)
 - 0x200000 (maximum frame buffer size)
 - 0x010000 (control info size)
 ===========
   0xdeffff ≥ SB_DISPLAY_AREA (0xc40000)
```

SB_DISPLAY_ADDR also passes on this test: the window system shared memory is located below **shmmaxaddr**.

Increasing Performance by Decreasing Memory

The previous discussion has centered mainly on increasing memory size to ensure that all window processes can execute without running into shared memory problems. At the other end of the spectrum, you may be able to reduce memory usage, thus increasing window system performance.

Decreasing memory requirements is practical when most of your window processes are small and when your window shared memory requirements are minimal. For example, if none of your window processes use more than 4Mb of code/data space (4.75Mb less than the default amount available), and only term0 windows are used (i.e., you don't need shared memory for retained rasters), you can set SB_DISPLAY_ADDR and WMSHMSPC to values less than their defaults, thus increasing the performance of your applications.

IMPORTANT

When computing the amount of code/data space required, keep in mind that the window manager is also a window process in the same process group as other window processes. Therefore, you must leave enough code/data space for the window manager to execute.

An Example

The following example should help clarify how to decrease window process memory requirements and improve performance: Suppose you've written a number of window applications that use only term0 windows. You've calculated that none of the window processes will ever consume more than 3.5Mb of code/data space when executing. For simplicity, round this figure to 4Mb.

Since only term0 windows are used by the applications, you've calculated that no more than 0.75Mb will be required for the window information area of shared memory. You round this figure up to 1Mb just to be safe. This means WMSHMSPC should be set to `0x100000` (1Mb) which is 1Mb less than the usual value of 2Mb (`0x200000`).

Determining the Correct Value for SB_DISPLAY_ADDR

Now you know the maximum code/data space required (4Mb) and the maximum size of the window information area of shared memory (WMSHMSPC = 1Mb), so you can determine the correct value for SB_DISPLAY_ADDR.

Using equation A-4, SB_DISPLAY_ADDR is computed to be 5.25Mb:

```
   0x400000 (4Mb maximum code/data space required)
+  0x100000 (1Mb window information area — WMSHMSPC)
+  0x040000 (256Kb for the Starbase area )
===========
   0x540000 (5.25Mb = SB_DISPLAY_ADDR)
```

Configuring Swap Space

When configuring file system swap space, be sure to have enough swap space to handle the largest configuration of window system processes. This section describes the various window system memory requirements with respect to swap space, and shows how to determine how much swap space to use in your file system.

Only the system administrator can change swap space on your system. Consult the *HP-UX System Administrator Manual* for details on configuring swap space.

Window System Swap Space Requirements

The swap space required can be estimated by summing the space required by the following components of the window system:

- window system base requirement (540Kb)
- shared memory requirement (the value of the WMSHMSPC environment variable)
- window manager data (described below)
- term0 data (described below).
- pop-up menu save area (described below).

This estimate accounts only for the swap space required by the window system, per se, and does not account for swap space needed by shells in term0 windows or any other applications being run in the window system.

Window Manager Data

To estimate the amount of swap space required by the window manager, multiply the requirement of each component listed in Table 2-4 by the number of times it is used in the window system; then sum all the multiplied components. Finally, double this amount. Equation A-5 shows how to compute window manager data requirements.

$$\langle \text{wm data} \rangle = 2 \times \Sigma \left(\langle \text{component req} \rangle \times \langle \text{occurrences of component} \rangle \right)$$

Equation A-5. Window Manager Data Requirements

Table A-4. Window Manager Data Swap Space Requirements

Component	Requirement per Instance of Component
user-defined icon	18Kb per icon
user pop-up menu	0.0012Kb × rows × columns, per menu
hotspot	0.03Kb per hotspot in a window
term0 or graphics window	2Kb per window

Term0 Data

To estimate the data required by all the term0 windows, simply sum the amount required for each window. Each term0 window requires one of three possible swap space amounts, depending on the size of the window's scroll buffer (memory used to store characters displayed in the window). Table A-5 defines the amount of memory required for each scroll buffer size.

Table A-5. Term0 Window Scroll Buffer Swap Space Requirements

Scroll Buffer Memory (= 4 × columns × rows)	Space Required
scroll buffer size $\leq$ 48Kb	128Kb
48Kb $<$ scroll buffer size $\leq$ 176Kb	256Kb
scroll buffer size $>$ 176Kb	512Kb

The default scroll buffer size for term0 windows is 80 columns by 48 rows: it requires 15 320 bytes (4 × 80 × 48) of scroll buffer memory—approximately 15Kb. Based on this amount and the data in Table A-5, a default term0 window requires 128Kb of swap space.

Note that a window's scroll buffer size is set when created. To get a scroll buffer different than the default size, use the `-r` option when creating the window via *wcreate(1)* or *wsh(1)*.

Pop-Up Menu Save Area

This memory is required only for user-defined pop-up menus. It stores the portion of the window system's image that is temporarily covered while the user-defined pop-up menu is displayed. When the menu disappears, the window system redraws the portion of the screen that was covered by the menu.

The memory required is one byte per each pixel in the largest possible menu that would be displayed on your system. This is largely dependent on your needs. For example, if you use large menus (many items with long item names) with the large fonts, then you will require more memory for this; if you just use average sized menus with default fonts, your needs will be less.

To compute the amount of memory required, multiply the following values:

- the maximum number of items in any user menu
- the maximum number of characters in any menu item
- the character width of $WMMENUFONT, in pixels
- the character height of $WMMENUFONT, in pixels
- 1.10 (add 10% overhead to be safe)

Then, to convert this value to kilobytes, divide the product of the above values by 1024. Finally, if your system uses a medium-resolution (512x400) display, double this number.

Example

The following example should help clarify how to configure swap space for Windows/9000. Suppose you've determined that the window system will have the following characteristics:

- There will never be more than 10 windows—four term0 and six graphics windows. Four of the graphics windows will be non-retained; the other two will each have retained rasters of 600x400 pixels.
- WMSHMSPC will be set to 1Mb (1024Kb), because the retained rasters will require 480 000 ($2 \times 600 \times 400$) bytes, i.e., approximately 0.5Mb. To allow for all other uses of this window information area, round up to 1Mb.
- Two of the windows will have user-defined icons.

- A maximum of six user-defined pop-up menus will be in use at any one time; each menu is 12 characters wide and has 20 rows. The WMMENUFONT will be `/usr/lib/raster/18x30/cour.OU` (i.e., 18 pixels wide by 30 pixels high).
- There will be a maximum of 64 hotspots in graphics windows at any one time.
- The window system will run on a high-resolution display.

Window Manager Data Requirements

Window manager data requirements are 120Kb, computed using Equation A-5, as shown below:

```
  36Kb (18Kb per user-defined icon × 2 user-defined icons)
+  2Kb (0.0012Kb × 12 characters × 20 rows × 6 user-defined menus)
+  2Kb (0.03Kb per hotspot × 64 hotspots)
+ 20Kb (2Kb per window × 10 windows)
======
  60Kb
×  2 (double the sum)
======
 120Kb required for window manager data
```

Term0 Data Requirements

The term0 windows used in this system will all have 80 columns by 24 rows of characters, thus requiring 1920 characters of scroll buffer memory (80 cols × 24 rows = 1920). This is less than 12Kb, so each term0 window will require 128Kb of swap space (determined from Table A-5).

Therefore, the total term0 data swap space requirement is 512Kb (4 term0 windows × 128Kb per window = 512Kb).

User-Define Menu Area Requirements

Since user-defined menus will be used, this value must be computed. The tallest menu has 20 items, and the widest item is 12 characters. The menu font (`/usr/lib/raster/18x30/cour.OU`) is 18 by 30 pixels. The product of all these (20 × 12 × 18 × 30 × 1.10) is 142 560. Divided by 1024, it equals 139.22; round this to **140Kb**. Since a high-resolution display is used, the amount needn't be doubled.

Final Swap Space Computation

Given the above estimates, the window system will require 2.5Mb of swap space, computed as follows:

```
    540Kb (window system base requirement)
+  1024Kb (shared memory size, as determined by WMSHMSPC)
+   120Kb (window manager data requirements)
+   512Kb (term0 data requirements)
+   140Kb (pop-up menu save area)
========
   2336Kb (estimated swap space required for window system)
```

Round this number up to 2.5Mb, just to be safe, and you have the swap space requirements for the window system.

Good-Citizen Processes

Unlike a graphics process running to a raw device, a process that performs graphics in a graphics window must be a "good citizen" in the graphics world. Since the window manager, term0 windows, and graphics windows are all built upon the Starbase Graphics Library, they will cooperate with other graphics processes that are also "good citizens."

A "bad citizen" is a process that changes global resources which are not process-dependent. Examples of these resources are:

- color map values
- planes displayed
- planes blinking

  For example, if a process had configured the window system to use one set of colors, and a second process changes the color map to a different set of colors, the color map could end up with duplicate colors—e.g., all entries could conceivably be black.

A good-citizen process will generally:

- use gescapes sparingly and in a manner considerate to other processes
- does not change the values of the color map (There may be an agreement to leave the first 4 or 8 color map entries constant and allow all users to modify the remaining entries in the color map, knowing that the others may also be changing these color entries.)
- does not double-buffer undisplayed planes
- does not blink planes
- does not turn Starbase clipping off, since this will allow access outside the window boundaries. (The exception here is when the user is **absolutely certain** that the vdc_extent/device-bounds will **never be exceeded**.
- uses separate file descriptors for Starbase graphics and fast alpha routines. (Violating this principle does not make a process a "bad citizen," but may cause unexpected results.)

Accelerated 3D Graphics Display Stations

B

This appendix describes HP Windows/9000 features specific to the HP 98720 and HP 98730 Graphics Display Station. The following topics are covered:

- Graphics Display Station concepts
- the **see_thru** window type.

Concepts

An accelerated 3D Graphics Display Station consists of

- a high-resolution (1280×1024 pixels), 16- or 19-inch color display;
- a display controller;
- and an interface card, which plugs into an I/O slot of a Series 300 SPU.

Please consult the appropriate chapter of the *Starbase Device Drivers Library Manual* for configuration information on the HP 98720 and HP 98730 display stations.

HP Windows/9000 runs on any configuration, although it does not use all the graphics hardware features.

HP Windows/9000, with one exception, always runs in the overlay planes. On the HP 98720, if no overlay planes are present, Windows/9000 runs in four image planes.

On the HP 98720, the Console ITE (Internal Terminal Emulator) and Windows/9000 use the first three overlay planes. The fourth overlay plane is reserved for the `hp98720` and `hp98721` drivers for their graphics overlay cursors.

On the HP 98730, Windows/9000 defaults to using the first three overlay planes, but can also be run to four overlay planes if WMSCRN is set to the appropriate device file (see the *Starbase Device Drivers Library Manual*).

See the *Starbase Device Drivers Library Manual* for a description of how the color map works on the HP 98720 and HP 98730 Graphics Display Stations.

The SB_OV_SEE_THRU_INDEX environment variable can be used to control which color is used for *see-through.* This environment variable only has an effect when the window system is powered up. For the HP 98720, it must be between 0 and 7 inclusive; for the HP 98730, it must be between 0 and 7 if using three planes, or between 0 and 15 if using four planes. The default see-through color index, if SB_OV_SEE_THRU_INDEX is not set, is 3.

This environment variable also affects the term0 server, graphics server and the window manager. For example, the following scenario will cause your term0 text seem to disappear:

1. A program sends color escape sequences to term0 to get yellow-on-black characters.
2. The *see-through* index corresponds to what used to be yellow (3).
3. The image planes are cleared to black so that *see-through* shows black.

In another example, suppose your window system environment variables are set up to have your window borders cyan-on-black. The borders will appear invisible if:

1. The *see-through* index corresponds to what used to be cyan (5). This is done by setting SB_OV_SEE_THRU_INDEX to "5" before powering up the window system.
2. The image planes are cleared to black so that *see-through* shows black.

If an application wishes to change the see-through color index, *wset_see_thru(3w)* can be called. There is also *wget_see_thru(3w)* for inquiring the current see-through color index that the window system is using.

The IMAGE Subtype of the Graphics Window Type

The IMAGE graphics window is a special case for the HP 98730 display system. This allows a user to create a graphics window which has the user area mapped to the image planes while the border is displayed in the overlay planes. This window type is useful for using the accelerated driver in windows where the user has access to the 3D graphics functionality that is available to the raw (non-window) device with the obvious exception of managing shared resources like the color map and the hardware support for sprites. The IMAGE graphics window can also be used with the non-accelerated driver (**hp98730**). Use of the IMAGE graphics window type only requires the specification of the proper option or parameter at creation time. Thereafter, it behaves exactly the same as any normal graphics window.

To create an IMAGE graphics window, either use the `-N` option with **-wgraphics** on the *wcreate(1)* or *wsh(1)* commands, or specify SETIMAGE for the attributes parameter of the *wcreate_graphics(3w)* library routine.

The see_thru Window Type

While HP Windows/9000 runs in the overlay planes, an application can be running in the image planes, taking advantage of the graphics accelerator. To see applications running in the image planes while running windows in the overlay planes, a **see_thru** window type is supported for *wcreate(1)*. This *see_thru* window is a non-retained graphics window whose background color is set to *see_thru*, as defined by either the SB_OV_SEE_THRU_INDEX environment variable, or *wset_see_thru(3w)*.

To see an application running in the image planes while HP Windows/9000 runs in the overlay planes, do the following:

1. Start the window system:

   ```
   wmstart
   ```

2. Create a *see_thru* window via the *wcreate* command; for example, the following creates a *see_thru* window named **Image_planes**:

   ```
   wcreate -w see_thru Image_planes
   ```

3. Move and size window as appropriate.

4. Run some program to the image planes that uses an accelerated driver (i.e., **hp98721** or **hp98731**). For example:

   ```
   graph_appl /dev/crt hp98721
   ```

5. Top the *see_thru* window; for example, the following moves the **Image_planes** window created in step 2 to the top of the window stack.

   ```
   wdisp -t Image_planes
   ```

   After this step, the application running in the image planes should be visible in the user area of the *see_thru* window.

6. You can then manipulate the window interactively or with commands to suit your needs.

Glossary

C

The following terms are used frequently when discussing the window system:

activate font — Make an already loaded character font the base or alternate font for a term0. At the Font Manager or Fast Alpha libary level, make a font the one to use next for printing characters. See also *load font.*

active font — The font in which characters are currently being written.

affiliation — Special relationship between a terminal or window and a process. User-generated signals (such as SIGINT due to hitting Break) from the terminal or window are sent to all processes affiliated with the window. See also *process group.*

alternate font — For term0 windows, the alternate font is a secondary font (other than the base font) for writing characters. Characters can be written in the alternate font by sending an ASCII SO[1] character. Characters continue to be printed in the alternate font until either an ASCII LF[2] or SI[3] character is encountered, after which characters return to the base font. The *wfont(1)* command (or term0 escape sequences) can be used to change the alternate font.

anchor point — Location of the upper left corner (coordinates *0,0*) of the window's user (contents) area. Also called *window location.* (Also the stationary point of a rubber-band echo.)

attached — See *selected.*

banner — Synonymous with window border (see *border*).

base font — The base font is the default font used for displaying characters in a term0 window. To switch from the alternate font to the base font, use the ASCII SI[1] or LF[2] characters. The *wfont(1)* command (or term0 escape sequences) can be used to change the base font for a term0 window.

[1] Decimal 14; octal 016.
[2] Decimal 10; octal 012.
[3] Decimal 15; octal 017.

bit-mapped display
: Display device which has one or more bits of memory for each pixel on the screen. Images may be written to this memory by user processes. Then the display is updated directly from memory, with minimal calculations. HP Windows/9000 works only with bit-mapped displays.

border
: The portion of a window surrounding the user area and containing the label and manipulation areas (control boxes and scroll arrows). Actually formed from a second window unit which lies underneath the user unit.

border colors
: The foreground and background colors of a window border. The window's label (name) and manipulation areas are displayed in the foreground color on top of the background color. See *color.*

border style
: Status of a window border, either *normal* (label and manipulation areas present) or *thin* (no maniupulation areas or label are present; only the pop-up menu can be used to interactively manipulate windows). In addition, graphics windows can have no border, known as the *null* border type.

bottom window
: Window which is lowest in the display stack; therefore it is occuled by any window which overlaps it.

buffer size
: Width and height in columns and rows of the scroll buffer for a term0 window. By default, term0 windows have a buffer size of 80 columns by 48 rows (two default screens of information). See also *window size*, *logical screen size*, and *raster size.*

buttons
: Switches on a mouse or graphics tablet stylus or puck switch, used to cause an event—i.e., send input to a window.

cell size
: Width and height in pixels of the character cells for a given font. In uniformly sized fonts, cell size will be the same for all characters in the font. Only uniformly sized fonts are currently supplied with your system.

character font
: See *font.*

color
: An index into the color map for the device. For black-and-white displays, color is either 0 or 1. For color displays, typical ranges are 0 to 15, and 0 to 255 (inclusive).

color map — A table which maps index numbers (colors) into colors (intensities for each primary color) on the display. There is usually one color map per physical display, shared by all processes.

concealed — One of three ways of representing a window on the display; the other two ways are *normal* and *iconic*. When a window is concealed, it is not visible. As a consequence of being concealed, a window loses its position in the display stack. It may still be selected (connected to the keyboard), receive input and output, and be otherwise manipulated.

contents area — Synonymous with *user area*. This is the area of a window surrounded by the border. It is the area of the window in which your applications execute (e.g., write information to the window's contents area).

control box — Any one of the four boxes located in the corners of a normal window border. Each box can be used to interactively perform a windowing function. See also *icon control box*, *move control box*, *pause control box*, *size control box*.

desk top — Any portion of the display screen not occluded by any window, pop-up menu, or other displayed object.

display stack — When more than one window appear on the display screen, they form a display stack. Each window has a position in the display stack, e.g., there is a top window and a bottom window. The display stack is implemented as an ordered list of displayable windows which contains information determining which windows occlude others if they overlap.

displayable — Opposite of *concealed*; a window which is in normal or iconic form. Part of the window is visible if it is not occluded or entirely off screen.

echo — Synonymous with *pointer*. The pointer (sprite) on the display screen which corresponds to the locator's position. The echo takes different forms when it appears over different screen and window areas. It can be redefined, via window library routines, to suit your applications needs.

elevator — A rectangular box which can be displayed in a window's border. The box can be used to control graphics window panning, or can do an application-dependent function. See the "Arrows and Elevators" chapter of the *Programmer's Manual*.

event — Action of pressing [Select] on the keyboard, a mouse button, or the tablet stylus switch. Also, the receiving of a signal by a window-smart process, usually due to a user's interactively changing a window's attributes.

Fast Alpha — Set of library routines which let you display character information to graphics windows at high data rates. Can also be used to display information to the display device when the window system isn't running.

font — Collection of bit patterns and associated information which tell the window system how to display characters on the screen. Each font has a characteristic cell size—that is, the number of pixels each character is wide and high. See also *font file.*

font cache — An array (in Windows/9000 shared memory) that contains font information loaded from font files. There is a term0 font cache, used only with term0 font management routines, and a fast alpha-font manager font cache, used only with the fast alpha and font manager libraries.

font file — Ordinary HP-UX file, containing font description information, from which a font is loaded into memory as needed. See also *font path.*

Font Manager — Set of library routines which let you use and control character fonts. Like the Fast Alpha library, Font Manager routines can be used to display characters to graphics windows or the bit-mapped display.

font path — Full or relative pathname (filename) for a font file. If a font path does not begin with "/", "./", or "../", the value of the environment variable WMFONTDIR is prepended to the font path.

generic name — This is the name of the original link to a window's *pty* special file, e.g., `/dev/pty/pty03`. Generic names are managed by the window manager and are allocated to new windows (when the windows are created) by linking them to window names you supply, or those you allow the window manager to automatically choose.

graphics — Graphics display window type. See *window type.*

icon — A symbolic representation for a window, often referred to as the "shrunken form of a window." Icons use very little space on the display. An icon normally includes the window's label and both a move and icon control box.

iconic form — One of the three ways of representing a window on the display; the other to ways are *normal* and *concealed.* See *icon.*

internal terminal emulator — Software which allows you to use the keyboard and bit-mapped display as a terminal-type device when the window system is not active.

ITE — The acronym for internal terminal emulator.

label — A string displayed with a window's normal form, icon, typing aids, and pop-up menu. It defaults to the true name of the window, but may be set to any value up to 12 characters long.

load font — Bring a character font into memory from a font file. This makes the font available for use in a term0 window, or graphics window using Fast Alpha or Font Manager routines, but it does not mean the font will necessarily be used immediately after loading. See also *activate font.*

location — See *window location.*

logical screen size — The maximum size of the view into a terminal window's buffer. This is the same size as given by the *terminfo(5)* `cols` and `lines` values. A terminal window can be sized no larger than its logical screen size. By default, term0 windows have a logical screen size of 80 columns by 24 rows. See also *buffer size.*

locator — Any input device—such as cursor keys, mouse, or a tablet's stylus or puck—which provides *x,y* location information (or changes in location).

manipulation areas — Areas in a window's border which help you to interactively control the window. These consist of *control boxes* and *scroll arrows.* For example, using them, you can move the window, change its size, convert it to an icon, or call up a pop-up menu.

move control box — This control box, located in the upper-left corner of a window's border, allows you to interactively move the window using a locator device such as a mouse or graphics tablet stylus.

mouse — **1:** A small, furry rodent with a hairless tail, often used in scientific experiments; considered vulgar and disgusting by most members of the human species. **2:** Also, a simple input device which glides around on the desktop and has one or more buttons (switches) on it. The mouse provides location (locator) and event (button) input. Considered friendly by most humans.

name — See *window name.*

normal form — One of three ways of representing a window on the display; the other ways are *iconic* and *concealed.* In normal form, the user unit (text, graphics, etc.) is displayable, so it is visible if not off-screen or occluded.

normal border — See *border style.*

null border — See *border style.*

occluded — Window is normal or iconic (and not concealed), but it (or a portion of it) is hidden by parts of other window(s). Occluded windows or parts are not visible. (See also *display stack.*)

off-screen — Window or portion of window is located outside the display screen area, so it is not visible, even though it is displayable. Concealed windows may have locations on-screen or off-screen, but they are not visible at all because they are not displayable.

pan position — The *x,y* pixel location of the view into the virtual raster of a graphics window, i.e., the position where a window's image is taken from, within the virtual raster. A graphics window can grow no larger than its raster size; at this maximum size, no panning is possible.

pixel — One picture element; the smallest displayable area of a display (one point on the display). Each pixel may have multiple bits associated with it in memory (i.e., to control its color).

pointer — The graphics pointer (sprite) on the display screen which corresponds to the locator's position. The pointer takes different forms when it appears over different screen and window areas. It can be redefined, via window library routines, to suit your applications needs. Also known as *echo.*

pop-up menu

A list of interactive choices displayed in a rectangular box. The pop-up menu pops up when [Select], the left mouse button, or tablet stylus is pressed. Most of the choices affect one of the window's attributes. The menu disappears after: a choice is made, selection is done outside the menu, or an interactive timeout period has elapsed (specified by the WMIATIMEOUT variable).

You can also define your own pop-up menus via window library routines.

process group

One or more processes which have a process group number in common. Calling *setgrp(2)* makes a process a group leader by setting its number to its process id; this is inhereted by its descendent processes. The first unaffiliated process group leader that opens an unaffiliated window becomes affiliated to that window. See also *window group.*

pty

Pseudo-terminal (tty) special file. Each has two "sides"; the slave side looks like a terminal-type device, while the master side allows capturing and manipulation of data. See *pty(4)* for more details.

raster

Pattern of lines which makes up the physical display; memory underlying the image displayed on a graphics window. See also *scroll buffer* and *retention.*

raster size

Width and height, in pixels, of the memory which records the data in a retained graphics window. See also *window size*, *screen size*, and *buffer size.*

repaint

Redisplay one window or the whole screen from memory. Only term0 or retained-buffer windows, typing aids, and the desk surface can be successfully repainted.

representation

See *normal form*, *iconic form*, and *concealed.*

retention

Memory (a *raster*) is allocated to "back up" parts of a graphics window that may become occluded. This is done at window creation and permits the window to be repainted (redrawn) from memory if necessary. Non-retained windows cannot be repainted if, say, a portion which was occluded becomes visible. Term0 windows are not retained, but the windows can be repainted from character-level information which is always kept in the window's scroll buffer.

scroll arrows — Arrows which appear in a window's border. You can scroll information in a window's contents area by clicking the select button when the pointer is located over an arrow. Information will scroll in the direction indicated by the scroll arrow.

scroll buffer — Memory of characters and their attributes, underlying the image displayed in a term0 window. The scroll buffer is typically larger than the window in vertical direction, so scrolling is possible. The default buffer size is 80 columns by 24 rows of characters, providing two default screens of information scrollable per term0 window. See also, *logical screen size*, *buffer size*, and *raster*.

select button — Button(s) on the locator device which activate interactive operations. Typically, the select button is set to button one (the leftmost mouse button, the stylus point, and the leftmost puck switch button).

selected — Window is attached to the keyboard and optional mouse buttons, table stylus, or puck switch. Processes which read from the selected window receive input from these devices.

server — Program (process), invoked for one window, which manages the window. The server acts as a go-between for the window manager and user processes.

sfk — An abbreviation for softkey.

shuffle — Rotate the display stack either upwards (top window becomes bottom window) or downwards (bottom window becomes top window). There is no visible change unless at least a portion of the top (bottom) window occludes another window or portion thereof.

size — See *cell size*, *window size*, *logical screen size*, *raster size*, *size control box*, or *buffer size*.

softkeys — Consisting of a label and definition, softkeys correspond to the function keys on the ITF keyboard. Each window can have its own softkey labels and definitions. The labels can be displayed optionally at the bottom of the screen for each selected window. When a function key is pressed, the definition string is returned through the selected window's device interface. Definitions strings can be redefined for term0 windows, but are constant for graphics windows.

stack — See *display stack*.

stylus — Pointing device on a tablet, which usually has a built-in switch.

tablet	A graphics input device from which locator and stylus switch information is read.
term0	Alphanumeric terminal emulator type of window. See *window type.*
thin border	See *border style.*
top window	Window which is highest in the display stack. Therefore, it cannot be occluded by any other window. (It may still be invisible if moved completely off the screen).
unit	See *user unit* and *window unit.*
user unit	Window unit which contains user data—alphanumeric text in term0 windows, and graphical output in graphics windows.
viewing position	See *pan position.*
virtual device	Each window is one of these, because each acts like an independent physical device of some type—that is, an alphanumeric terminal or graphics display.
visible	A window or portion thereof which is actually shown on the screen. A window must be normal and displayed in order to be visible.
window	Collection of associated window units which act as a single entity. Windows may be attached to the keyboard and receive input, receive output from one or more processes, be moved, concealed, expanded or shrunk, etc.
window-dumb	Process doesn't "know" that is is running in the window system; it thinks that it is running at a terminal or raw raster-graphics device, when it is actually running in a window. In other words, the window system is invisible to window dumb-programs.
window group	All processes which are associated with a single instance of the window manager, that is, all processes running on one physical display. This includes the window manager, all server processes for windows on the display, and all user processes affiliated to any window on the display. It includes more than one process group.
window location	Screen location of the upper-left corner of the user unit, in pixels. Also called *anchor point.* See also *pan position.*
window manager	Program (process) invoked once per physical display. The window manager controls the aspects of the window system common to all window processes. Its process name is *wm.*

window name — True name of a window, specified when it is created (or chosen by the window manager). Each window has an associated *pty* special file with the same name which may be more than a basename (e.g., it may include directories). See also *generic name*, and *label.*

window size — The user area's width and height in pixels, or for term0 window commands, rows and columns. See also *logical screen size*, *raster size*, and *buffer size.*

window-smart — Process "knows" it is running in the window system. It calls window library routines and/or recognizes special window system signals.

window spec — A list of zero or more window names, supplied as parameters to window system commands. Shell-like wildcards and ranges are allowed for the window spec, including the special symbol "-". See *windows(1).*

window type — Kind of window, determined when at window creation, and reflected in the choice of the window server program (process). See also *graphics* and *term0.*

window unit — Rectangular area of the display; smallest element of a window. One window is made up of one or more associated window units—the border unit and user (contents) unit.

wm — The command name for the *window manager.*

Subject Index

a

b

C

d

e

f

g

h

i

k

l

m

n

o

p

r

S

t

u

V

W

Notes