Integral Personal Computer

Programmer's Guide

HEWLETT
PACKARD

Edition 1 September 1985

82865-90003

Notice

Portable Computer Division
1000 N.E. Circle Blvd.
Corvallis, OR 97330 U.S.A.

Printing History

Edition 1 September 1985 Mfg. No. 82865-90003

Typesetting Information

With the exception of the covers and title page, this guide was typeset using:

- An HP-UX 2.1 Series 200 computer system.
- An HP 2686A Laserjet Printer with a "B" font cartridge.
- The System V Documentor's Workbench, including the 'troff' text formatter, the 'mm' macros, and the 'tbl' program to format tables.
- The 'dlj' laserjet driver from the ELAN Computer Group.

Contents

Chapter 6: Graphics Windows

Chapter 7: Alpha/Graphics Windows

List of Code Samples

Chapter 1
Introduction

1.1 Hello!

The *Integral Personal Computer Programmer's Guide* shows how to use the C programming language to access the system resources of your computer.

The computer is programmable in a number of high-level languages, including C, BASIC, FORTRAN, Pascal, COBOL, and LISP. Generally, however, the high-level languages don't allow complete control of the system. For example, using HP-UX Technical BASIC by itself, you can't change the location of a window on the display.

The C programming language *does* offer you programmatic access to almost all system resources, including windows on the display, the mouse, the file system, and external devices. In addition, the C compiler produces efficient and fast code.

This guide, then, focuses on the C language as a powerful programming tool.

If you prefer programming in other languages, keep in mind that you may be able to *call* (or access) prewritten C routines from the language of your choice, thereby extending that language. For example, you can use the CALLBIN statement from HP-UX Technical BASIC to call a prewritten C routine to move a window on the display.

1.2 Your Operating System

The Integral Personal Computer has an HP-UX operating system (*executing from ROM*) that runs on an 8 MHz MC68000 CPU and a custom graphics processor unit.

1.2.1 Versions

There are two versions of the Integral Personal Computer operating system:

- Release 1.0.0
- Release 5.0

The primary difference between the two is that Release 1.0.0 is based on AT&T UNIX System III and Release 5.0 is based on UNIX System V.2.[TM]

Essentially, Release 5.0 extends the services offered by Release 1.0.0 by providing shared memory, semaphores, message passing, real-time capabilities, and a new alpha/graphics window type. Release 5.0 removes nothing from the original Release 1.0.0.

Another difference is that Release 5.0 has been tuned to perform better during disc accesses, display-intensive activities, and instrument I/O.

Throughout this guide, we'll mention differences between the two versions when they're important, which isn't often.

1.2.2 Features

The Integral Personal Computer operating system offers:

- A friendly single-user environment with quick, disc-less boots (less than 30 seconds), easy automounting of discs, a windows-and-mouse user interface, and a tightly integrated package.
- Windows for maintaining clean separation of user processes. Windowing features include:
    - A powerful programmatic interface to the window manager routines in ROM.
    - *Alpha windows* that emulate terminals and *graphics windows* that emulate high-speed plotters.
    - Programmatic support for downloading character fonts from disc and painting windows with them.
    - Support for pointing devices, such as mouse and graphics tablet.
- A window-smart shell named "PAM" (for *Personal Applications Manager*). You can rely on PAM in three ways:

[TM] UNIX is a trademark of AT&T.

  - As a friendly, runtime environment for your application.
  - As a capable programming tool that supports I/O redirection, background/foreground processes, and shell scripts.
  - As a system resource that you can access programmatically.
- Support for a wide variety of HP-IB discs, printers, and plotters that makes it easy to connect and start using a full system.
- Extensive I/O capabilities that include a Device Independent Library, HP-UX real-time extensions, and drivers for a half-dozen interfaces.
- Support for dynamically loading new device drivers, such as the BCD interface driver.
- Support for up to 7M bytes of RAM.
- Multi-process control that includes shared memory, semaphores, and message control.[1]

1.2.3 You Can't Have Everything

The Integral Personal Computer does not support the following:

- Multi-user sessions. The Integral PC is a single-user computer.
- Virtual memory. Implementing a page fault mechanism in the software requires an expensive memory manager in the hardware.
- LAN. The computer is not a member of the HP Local Area Networking community. However, the System V tools discs provide the **uucp** and **mail** utilities, which work well for hard-disc users.
- The **tar** and **cpio** utilities for tape drives. You can use **tar** and **cpio** to make backups on *discs*, but not on tapes.

1.3 Who Should Use This Guide?

This guide is intended for programmers who want extended performance or capabilities from their computers. Here are some assumptions:

1. Available on Release 5.0 of the computer.

- The guide assumes most importantly that you're interested in HP-UX programming, especially as it relates to the Integral Personal Computer operating system.
- The guide assumes that you're well acquainted with many non-programmatic aspects of the computer, especially the user interface--the PAM shell and windows.
- The guide assumes that you're somewhat familiar with standard UNIX programming tools for software development, such as the **vi** screen-editor (or any decent text editor), the **cc** compiler, and the Bourne shell.
- The guide assumes that you have a working knowledge of C or at least have a C language manual handy.

1.4 What's Covered in This Guide?

This guide discusses the use of the programs, header files, and library files on the C compiler discs.

The guide shows how to write software routines that you can *call* from your C programs to control the system.

This guide does *not* cover programmatic access to the machine *hardware*. Such access requires an extensive knowledge of machine internals (as well as "hacking" expertise).

For example, the guide doesn't show how to write directly to the registers in the speaker chip to emit a tone. (Instead, you'll see how to do the same thing by using the speaker *driver*.)

This guide does *not* cover most of the utilities found on the HP-UX tools discs. The HP-UX tools discs are shipped with their own documentation set (all 27 lbs!)

Notably under-documented are:

- The real-time capabilities of the operating system.
- System V extensions, such as shared memory and semaphore operations.
- User interface conventions for software applications.

A word of caution: Many of the low-level intrinsics described in this guide are specific to the Integral Personal Computer and will not port directly to other

HP-UX computers. If your interest is in writing portable HP-UX code, refer to the *HP-UX Reference Manual* for standard programming features.

1.5 Using This Guide

This guide presents Integral Personal Computer programming information as clearly and directly as possible.

We recommend that you:

1. Read the rest of this introductory chapter and all of Chapter 2, *Your Software Development Environment.*
2. Review the table of contents.
3. Pick out those chapters that interest you and read them from the beginning.

1.6 Organization

Each chapter focuses on one aspect of the computer--the keyboard and mouse, for example, or the file system, or graphics windows.

Chapters are organized as follow:

- **Concepts.** These are introductions and overviews of the matter at hand.
- **Descriptions of Programming Tasks.** Typical tasks are:
  - "opening a window",
  - "writing characters to an external printer," and
  - "reading the status of the mouse."

  When possible, tasks are arranged chronologically; for example, in Chapter 6:
  - Opening a Graphics Window
  - Reading Keyboard Input
  - Drawing Lines in a Graphics Window
  - Closing a Graphics Window

- **Code Samples.** These short, byte-size routines demonstrate the how-to-do-it details for programming the computer.[2]
- **Further Information.** This section has pointers to more complete online and hardcopy documentation.

Much of the information in this guide is *machine specific.* It applies only to the Integral Personal Computer. We'll remind you when code samples rely on machine dependencies.

2. We assume that you have the gumption and wherewithal to modify these routines to fit your individual needs.

1.7 Typesetting Conventions

Following are the guide's typesetting conventions:

Typeface	Description	Examples
boldface	Boldface type is used for:	
	path names	**/usr/include/curses.h**
	file names	**fcc**
	device names	**/dev/plotter**
	function names	**signal()**
	window names	**echo**
	library names	**libc.a**
`courier`	Courier type is used for:	
	variable names	`windo_fd`
	structure names	`m_event`
	fields of structures	`e_xloc`
	commands that you type	`print foobar`
	code samples	`#define YES 1`
	special symbols	# (number sign)
	escape sequences	`ESC & d B`
italics	Italics type is used for:	
	new words	Hewlett-Packard Graphics Language (*HP-GL*)
	emphasis	Do *not* play with matches.

The *faces* of keys on the keyboard are bracketed, as [Shift] and [f1].

The *labels* of function keys as they appear at the bottom of the display are boxed, as $\boxed{\text{Start}}$ (PAM function key [f1]).

1.8 A Member of the Family

The Integral Personal Computer is a member of the Hewlett-Packard 9000 Family of computers, which include Series 200, Series 300, and Series 500

HP-UX systems.

The Integral Personal Computer is C *source code compatible* with other HP-UX machines. In addition, a number of libraries are available which make moving code from machine to machine easier. Common libraries include the *fast alpha* library and the *Device Independent Library* (DIL).

Object code compatibility does *not* run across Series 200/300/500 machines[3]

There are also differences in file systems among Series 200/300/500 machines. However, most file system calls are compatible at the application level, and utilities exist to support media transfers.

1.9 Further Information

In addition to the programmer's documentation you're looking at now, there are five major sources of information regarding the Integral Personal Computer.

- The owner's manuals for the computer, which describe PAM, the windowing user interface, and the "friendly" utilities for your "run-only users."
- The *HP-UX Reference Manual* (sometimes called the "brick" or "briquettes"-- small sized), which describes all utilities, system calls, file formats, etc. in alphabetical order.
- The *Tutorials and Concepts* binders that, like the *HP-UX Reference Manual*, are shipped with the HP-UX tools discs. These tutorials have in-depth discussions of the Bourne shell and other powerful tools like **make**, **sed**, and **nroff**. The tutorials also cover system I/O, process control, and the HP-UX assembler.
- The C language documentation disc that is shipped with the C compiler discs. The documentation disc includes the HP-UX manual pages for the dozens of system routines and devices supported by the operating system.
- If you're curious, there are the header files (such as **scrn/wmcom.h**) that define the constants and data structures used throughout the system.

  Header files are shipped on the C language preprocessor disc.

3. However, Release 1.0.0 and Release 5.0 are object code compatible.

Printing out and studying selected header files are excellent ways to get into the system internals of the computer.

Remember: "When all else fails, read the documentation. When that fails too, *blame* the documentation."

Chapter 2

Your Software Development Environment

This chapter shows how to use the Integral Personal Computer C compiler and related utilities to develop software for the computer.

This information is useful if you're interested in:

- Developing software specifically for this computer.
- *Porting* (or bringing in) software from another computer in order to compile it on this computer.
- Porting your software *to* another computer system.

The windowing system of the computer makes software development easy. For example, you can edit your source file in one window and *leave the editor running all the time*, even during compilations.[1] With adequate system memory, you can quickly and easily switch among the editor window, the shell window, and your application window, as you write, compile, and debug your application.

2.1 System Configuration

2.1.1 Pieces of the System

The ideal system for software development consists of the mainframe itself (with keyboard and display) and the following items:

- A printer (either internal or external) for listing files.
- One megabyte (or more) of Random Access Memory (RAM). You may wish to add the HP 82904A Bus Expander and one or more RAM cards to increase the

1. In fact, you can run two or more instances of a shell or editor at a time, each in its own window.

amount of memory in your system.[2]

- An HP-IB hard disc having 20 megabytes (or more) of storage. The computer supports two disc protocols:
  - The *Amigo* protocol, as in the HP 9133B/V/XV and HP 9134B/XV disc drives.
  - The *Subset-80* (SS-80) protocol, as in the HP 9133H/L and HP 9134H/L disc drives.

  To get the best performance during disc accesses, use a hard disc that supports the SS-80 protocol.[3]
- The C compiler discs. We assume that you've already gotten the compiler.
- The HP-UX tools disc set. The HP-UX discs consist of compiler support utilities, such as **make** and **ranlib** and a host of other software development utilities.
- A serial interface or modem for the computer and the Datacomm software for uploading and downloading files.

  You can move source code and object code to and from other computers efficiently when you use your computer as a "smart" terminal with file transfer capability.[4]

2.1.2 Setting Up the File System

Due to naming conventions and common usage, you will probably wish to set up the following directories in your system:[5]

/usr This is typically the volume name of your hard disc. It will hold the lion's share of your files.

2. There's no such thing as too much memory!
3. Of course, you can use any HP-IB flexible disc drive or hard disc drive that supports either disc protocol.
4. The Datacomm program for the computer uses the Christensen umodem file transfer protocol, available on many host computers.
5. Note that at power-on the / root directory is always installed in the electronic disc. Consequently, each directory name beginning with a slash (such as "/basic") must correspond to 1) an external disc drive, to 2) a partition of a configurable hard disc, or to 3) a directory created out of a portion of the electronic disc.

/**bin**	Having a /**bin** directory around is very handy for storing typical HP-UX utilities. In a system with a limited number of disc drives, you can create a /**bin** directory in the electronic disc.
/**etc**	The /**etc** directory is used for miscellaneous files. Typical files needed in /**etc** are **passwd** and **termcap**.

At power on, the computer creates the following useful directories for you in the electronic disc: /**dev** (for device assignments) and /**tmp** (for temporary files).

As "system manager," you'll quickly learn how to put together the pieces of a small but efficient system.

2.1.3 Autost Programs

To configure your system at power-on, you may wish to create an **Autost** file that the PAM shell will execute when PAM first starts running.

The name **Autost** typically refers either to an executable code file (such as **vi**) or to a PAM shell script.

Autost shell scripts can include PAM commands:

- To create one or more directories in the electronic disc.
- To copy heavily used files to the electronic disc.
- To begin the execution of one or more programs of your choice.
- To initialize an external printer with the **printer_is** utility.*
- To initialize an external plotter with the **plotter_is** utility.*
- To get a new shell environment with the **getenv** command.*
- To change to a different working directory with the **cd** command.*

For your information, Release 1.0.0 of PAM executes the **Autost** file as a separate application program. Consequently, whatever commands the **Autost** file contains (such as **printer_is**) apply *only* to the **Autost** process itself.

In contrast, Release 5.0 of PAM *sources* the **Autost** file so that PAM itself (and not a separate process) executes the commands in the **Autost** file one by one.

* Not applicable to Release 1.0.0 of the operating system because the operation affects only the Release 1.0.0 autostart process and not the system at large.

2.2 Safeguarding Your Files

The code that you develop on the computer takes time and energy. To safeguard your work, you need to *back up your discs and files.*

You can perform backups in several ways:

a. By using the **copy_disc** utility shipped with the computer. **copy_disc** is a user-friendly program for replicating discs. Note that **copy_disc** expects source and destination discs to be the same size and type.

b. By copying files and directories one by one to microflexible discs. Typically, you direct the PAM **copy** command or the **cp** utility to write one or more files to an external backup disc.

c. By using the Datacomm program or the **uucp** HP-UX utility to upload individual files to a host computer. You leave the file copies on the host computer for safekeeping.

You can automate the file copying business by using the **tar** and **cpio** HP-UX utilities that can copy entire directories and subdirectories to external discs.[6] Refer to Chapter 11, *The File System*, for a short **copydir** shell script that uses **cpio** to copy directories.

After writing out your important files to an external disc, the idea is to store the disc in a *safe place.* This is obvious, right?

A little known feature of the Datacomm program is that you can automate your Datacomm file transfers:

1. Batch your Datacomm commands into a text file using the file format specified in the Reference Topics section of the Datacomm documentation.
2. Specify the file name of your command file as part of the `Connect` string.

Datacomm will execute the file transfers one by one, as specified in your command file. Furthermore, if you use the **runat** utility shipped on the Datacomm disc, you can arrange for running your command file at odd hours, when the telephone rates are low.

6. The Integral Personal Computer supports neither tar nor cpio for making tape backups.

Use any combination of the above methods to protect your work against accidental damage or erasure. Whatever method you choose, do it *often* and *regularly*. For example, every afternoon you may wish to make copies of the individual files that you've edited and then perform a more complete backup once each week.

Note that the HP-UX discs include the useful Source Code Control System (*SCCS*) for tracking incremental changes that you make to text files.

2.3 The PAM, Bourne, and C Shells

All UNIX systems have a *shell*, or command interpreter, that enables the user to enter system commands from the keyboard.

The main purposes of a shell are:

- To start the execution of other programs (sometimes called *exec-ing a file*).
- To manipulate files and maintain the file system hierarchy.

Typical shells offer additional features:

- A search path capability that directs the shell to search through specified directories to locate program files.
- I/O redirection and piping so that user can easily change the input stream *into* a program as well the output stream *from* the program.
- Batch processing (that is, executing commands from a *script file*).
- Parameter substitution, by which command line arguments are passed to script files and other programs.
- Pattern matching so that the user can specify *wildcards* in path names.
- The ability to run programs in the *background* and let the user do other work without waiting for them to run to completion.

You can think of a shell as simply an *application program* that serves as the user interface to the operating system and file system.

The Integral Personal Computer operating system supports three different shells:

- The *Personal Applications Manager* (**PAM**) is the default shell for the computer. At power on, PAM is one of the first processes to start up.

PAM is easy-to-use and visually oriented. It enables users to enter command lines from the keyboard *and* to enter commands with a pointing device, such as a mouse.

- The *Bourne shell* (or **sh**) is a small but capable command interpreter.
- The *C shell* (or **csh**) is the biggest of the three and offers the most extensive command language.

PAM is built into the Integral Personal Computer operating system (as **/rom/PAM**) and is always available. The Bourne and C shells are included on one of the discs shipped with the mainframe (as **/hp-ux2/sh** and **/hp-ux2/csh**, respectively, on the Release 5.0 discs).

To start a shell, simply enter its path name in the PAM command line. For example, with the HP-UX commands disc inserted in the internal disc drive, enter the following in the PAM command line:

```
/hp-ux2/sh
```

--or just:

```
sh
```

--and PAM will create a window for the Bourne shell to run in. The current working directory will be the directory from which you invoked the shell.

Note: You can run *two* or more shells at a time. For example, it's often efficient to run PAM in one window and **sh** in another and to switch between the two windows. To start another instance of PAM, simply enter:

```
PAM
```

--in the command line.

The following sections briefly describe the features and limitations of each of the three shells.

2.4 The PAM Shell

PAM is a visual shell--it always displays the current working directory (called the *Open Folder*) and the file entries in that directory.[7]

2.4.1 Advantages of PAM

- The visual display in PAM is convenient for "browsing" through your file system to see what's there. PAM offers a clean separation of file entries into *Programs*, *Data Files*, *Folders*, and *Devices*.
- PAM is "window-smart." Some of the benefits are:
  - PAM gives each program its own window to run in. The window inherits the current PAM character font, the PAM window color, and the status of the window border.[8]
  - You can use PAM to send strings and redirect output to specified windows with the window redirection symbol (a number sign, #).
  - PAM makes effective use of a pointing device to allow you to select file entries, to build command lines (with the $\boxed{\text{Echo}}$ key label), and to execute command lines easily.
  - PAM creates view windows according to the current font size.
- PAM has a number of friendly, built-in file commands, including:

move	Similar to the **mv** command.
copy	Similar to the **cp** command.
view	Similar to the **more** command.
delete	Similar to the **rm** command.
makefolder	Similar to the **mkdir** command.

- PAM maintains a stack of the last 20 commands you've entered. You can scroll through the command stack and edit individual commands with the

7. Exceptions are files and directories whose names begin with a period (.)--these names are not displayed.
8. Not available on Release 1.0.0 of the operating system.

editing keys, such as [Insert char]. It's easy then to re-execute earlier or modified commands.

- PAM is built into computer Read-Only Memory *(ROM)* so that PAM is always available.

2.4.2 Limitations of PAM

- PAM displays only the *names* of file entries. To examine the sizes, times, or read/write/execute permissions of files, you need to execute the **ls** program.
- PAM does not support variable assignments and provides only limited support for parameter substitution.
- PAM does not allow conditional testing or looping in script files.
- PAM's syntax is somewhat inconsistent with the syntax of **sh** and **csh**. For example, Release 1.0.0 of PAM recognizes the exclamation mark (!) rather than the number sign (#) as the comment character. (However, Release 5.0 of PAM recognizes that # in *column 1* of a script file indicates a comment.)
- Release 1.0.0 of PAM does not support pattern matching of the asterisk (*) and question mark (?).

2.4.3 Using PAM Programmatically

You may wish to give your application the ability to fork a shell to perform a specific task. The way to invoke PAM from a program is to use the use the execl() system call, as follows:

```
execl ("/rom/PAM", "PAM", "-c", string, 0);
```

--where `string` is a valid command string, such as:

```
"MyProg File1 File2"
```

The `"-c"` flag instructs PAM to run the command `string` and to otherwise suppress display output.

After the **execl()**, the invoked instance of PAM will perform an `exit(1)` if no error was encountered in running the command; an `exit(0)` otherwise.[9]

9. Release 1.0.0 of PAM always performs an exit(1) after an execl().

2.4.4 Other PAM Information

When a **getenv** command is executed (either from the keyboard or in an **Autost** file), the **LANG** language variable is set. The default language is `american`.

2.5 The Bourne Shell

2.5.1 Advantages of the Bourne Shell

- The Bourne shell is relatively small (not much larger than PAM) and requires less than 40K bytes of user memory.
- The Bourne shell provides programming constructs, such as **if**, **case**, **for**, **while**, and **until**.
- The Bourne shell is standard across all HP-UX systems.

2.5.2 Limitations of the Bourne Shell

- The Bourne shell knows nothing about windows. All applications started from the Bourne shell will run in the **sh** window.
- The Bourne shell has no command stack or history mechanism.
- The Bourne shell has no in-line editing capabilities.

2.5.3 Placement

There are a number of utilities, such as **make** and **sh** that expect to find a Bourne shell in the /**bin** directory (named /**bin**/**sh**) in order to do their work.

For greatest convenience, keep a copy of the Bourne shell in the /**bin** directory.

2.5.4 Stopping the Bourne Shell

Use a [CTRL][D] to stop the Bourne shell. Then press the [Stop] key to eliminate the **sh** window.

Alternately, press [Shift][Stop] to do both at once.

2.6 The C Shell

2.6.1 Advantages of the C Shell

- The C shell has a flexible and useful "history" mechanism.
- The C shell offers a large set of programming constructs and intrinsic commands.

2.6.2 Limitations of the C Shell

- The C Shell is *not* memory efficient.
  - The shell itself requires about 90K of user memory.
  - You can assume that you will lose about 1K of memory with each command that you execute and *won't get it back* until you terminate **csh**.
- Little of the programming language is consistent with the Bourne shell.

2.7 Shell Scripts

It's often convenient to automate keyboard commands by batching them all into one file, called a *shell script*, and executing the shell script--as a single batch file--from the command line.

PAM supports shell scripts by:

1. Creating an alpha window for the shell script to run in.
2. Executing the individual command lines in the shell script one by one.

Autost files, discussed earlier, are typically shell scripts.

Because of the richer command language of the Bourne and C shells, you may wish to have your script files executed by the Bourne or C shell rather than by PAM. The easiest way to execute a shell script from the Bourne or C shell is to create a **sh** or **csh** window and to run the shell script from that window. Release 5.0 of PAM provides two additional ways to run a shell script from the Bourne or C shell:

1. Set the SCRSHELL environment variable of PAM to identify the shell that you want PAM to use in running shell scripts. For example:

```
SCRSHELL=/usr/bin/sh
```

--causes PAM *always* to invoke **/usr/bin/sh** when asked to run a shell script. Another example:

```
SCRSHELL=sh
```

--causes PAM to look for a copy of **sh** anywhere on your current search path and **exec()** the specified shell.

2. Include a number sign (#!) as the first two characters of your shell script. For example:

```
#!/usr/bin/sh
echo 'Baby, we were born to run...'
```

Shell scripts beginning with #! and an executable path name cause PAM to look for the specified shell and **exec()** that shell.[10]

The second method takes precedence over the first. In other words, you can override the SCRSHELL assignment by including a #! command line in your shell script. Refer to the owner's documentation for more information about PAM scripts.

We recommend that you write your shell scripts as "portable" as possible so that the scripts will run OK whether started from the PAM, Bourne, or C shell. You never know what shell you'll be using when you want to fire off a script file!

2.8 Shell Environments

The shell that you're using makes available an *environment* to your application. Typical items (or shell variables) in an environment include:

- The home directory.
- The default shell.
- The terminal type.

10. Release 1.0.0 of PAM runs all shell scripts simply as PAM scripts.

- The **termcap** entries for the terminal type.

The information in the current environment is kept in a global variable `environ` that is readily accessible to your application. The following **showenv** program prints out all, or a specified number of, environment values:

```
/*******************************************************/

#include <stdio.h>

main (argc, argv)
int argc;
char *argv[];
{
    extern char *getenv(); /* global function */
    extern char **environ; /* pointer to global info */

    char *envar; /* local vars */
    char **envptr;

   if (argc > 1) {
       while (--argc > 0)  {
            envar = getenv(*++argv);
         printf("%s\n",envar);
        }
    }
    else {
        envptr = environ;
        while (*envptr) {
            printf("%s\n",*envptr++);
        }
    }
}
/*******************************************************/
```

Code Sample 2-1. Program To Print Out the Environment

Running this `showenv` program is easy. For example, to check the value of the time zone variable (TZ), you can execute:

```
showenv TZ
```

To check the value of all environment variables, execute simply:

```
showenv
```

At power on, PAM reads the information contained in a file named **.environ** (the default file is **/rom/.environ**).

Whenever you start another shell, the information contained in the current environment is exported to the new shell.

The only way to alter the PAM environment is to execute the built-in **getenv** command on an entire file. For example:

```
getenv /usr/lib/NewEnviron
```

It's possible to alter individual *environment variables* in the Bourne and C shells from the command line by setting the variables to new values and then *exporting* them. For example:

```
PATH=$PATH:/NewPlace ; export PATH
```

An important reason for wanting to alter an environment is to change the search path used by the shell to locate program files.

It's possible to alter the environment of an application from *within the application.* Such know-how, however, is beyond the scope of this guide.

2.9 The C Compiler (cc and fcc)

The C compiler is your main tool for developing software for the computer. The compiler disc set offers you a choice of *two* compilers--**cc** (for compiling from a hard disc) and **fcc** (for compiling from flexible discs).

The **cc** and **fcc** programs are very similar--both use the same "back end" in the compilation process, and both produce identical object code. The difference is that **fcc** can handle flexible disc manipulations and will prompt you for the right disc at the right time.

The convention in this guide is to assume you'll be using **cc** most of the time.

The C compiler software product includes:

- A short "getting started" guide.
- A preprocessor disc named /**cpp**.

- A translator disc named /**c**.
- A documentation disc named /**man**.

On the compiler discs themselves are:

- The C compiler drivers (**cc** and **fcc**).
- The C preprocessor (**cpp**).
- The compiler itself (**ccom**).
- An assembly-language optimizer (**c2**).
- An MC68000 assembler (**as**).
- A capable linker/loader (**ld**).
- All relevant *header* files (***.h** files).
- All relevant *libraries* (**lib*.a** files).
- Extra utilities.
- Reference documentation (**man** pages) from sections 1, 2, 3, and 4 of the *HP-UX Reference Manual.*

2.9.1 Hard Disc Installation

We assume that you want to install your compiler and supporting files on a hard disc. This is a one-shot operation and requires about an hour of your time and about 2M bytes of disc space.

The convention is to name this disc /**usr**, either by using the **format_disc** utility or the **rename_disc** utility that is shipped with the computer.

To avoid wasting time, carefully follow the instructions in the getting started guide for the compiler in order to install the compiler complex on your hard disc.

A utility shipped with the compiler, called either **copyhard** or **sysload**, does the work in a somewhat friendly fashion. There is a useful **−v** ("verbose") option that causes the utility to report on its progress as it copies one file after another.

You will probably want to use the **−f** option as well to overwrite existing files when necessary.

The getting started guide includes a diagram for the hard disc to show the resultant file system created on the disc. In particular, the following directories on the /**usr** disc are important:

/usr/bin	Stores most of your useful utilities, such as **ls** and **cat**.
/usr/include	Stores the *header files* that contain the window, keyboard, and system definitions for Integral Personal Computer data types.
/usr/lib	Stores the C support libraries (including the runtime C library, the math library, and the fast alpha library) and the major pieces of the C compiler (including the preprocessor, the **ccom** compiler, and the optimizer).

2.9.2 Notes About Flexible Disc Compilations

There may be times when you wish to compile code using only flexible disc drives. At such times, use the **fcc** compiler as explained in the getting started guide.

If you have enough system memory, you can use the following procedure to speed up compilations:

1. Copy all your source files (***.c** files) to /**tmp** (in the electronic disc).
2. Copy all *local* header files (those with simple names that are enclosed within quotation marks, **""**) to /**tmp** as well.
3. Change your working directory to /**tmp**.
4. Insert the preprocessor disc (/**cpp**) in the flexible disc drive.
5. Enter the following command line:

```
/cpp/bin/fcc <flags> <file1> <file2> ...
```

The **fcc** compiler will start the compilation and will prompt you whenever a disc exchange is necessary. The compiler will leave the resulting **a.out** file in /**tmp**.

As an alternative, you can copy all your source files and local header files to the preprocessor disc itself in the /**cpp** directory.

The advantage of this second method is that it is more memory efficient. The disadvantage of this method is that the preprocessor disc is almost full (less than 100K bytes remain) and therefore can't hold much source text.

Note that it's not possible to put your source on a separate flexible disc when doing single disc drive compilations with **fcc**.

A final difference between using **fcc** and **cc** is that in **fcc** all options except the `-l` library option must be specified *before* any file names are given. Thus:

```
fcc -S myfile.c
```

--is legal, and

```
cc myfile.c -S
```

--is legal, but

```
fcc myfile.c -S
```

--is not.

2.9.3 Header Files

Header files are text files that consist of prewritten declarations for your programs. For example, in the `stdio.h` file shipped on the C preprocessor disc, the string EOF (for *end of file*) is defined to be *-1*.

Header files are so named because they are typically referenced at the "heads" of source files. Names of header files usually end in `.h` (dot-"h"), as in `terminal.h`.

By including selected header files in the compilation process (by means of `#include` statements), you can access prewritten constants and data types. For example, including the following statement at the beginning of a source file:

```
#include "ThisFile"
```

--causes the preprocessor to insert all the text within `ThisFile` (in the current directory) into your source file before passing the source on to the compiler.

Enclosing the header file name within angle brackets (<>) as in:

```
#include <stdio.h>
```

--causes the preprocessor to look for the file in one of two directories:

- **/usr/include** (if you're using the **cc** compiler).
- **/cpp/include** (if you're using the **cpp** compiler).

Note that the header file name may consist of a *relative* path name, as in `sys/errno.h`.

From now on, when a header file name appears in a code sample, you should know where the preprocessor expects to find the physical file. For example, when you see the following in a code sample,

```
#include <scrn/wmcom.h>
```

--you should know that the preprocessor will expect to find

```
/usr/include/scrn/wmcom.h
```

(if you're running **cc**) or

```
/cpp/include/scrn/wmcom.h
```

(if you're running **fcc**).

Note that the *order* in which header file names are included in your source file may be important. For example, header file `<scrn/plotem.h>` depends on declarations contained in `<scrn/disp.h>`.

Therefore, to reference the two files in the proper order, you would use:

```
#include <scrn/disp.h>
#include <scrn/plotem.h>
```

The same is true of the following pair--their order is important:

```
#include <sys/types.h>
#include <sys/stat.h>
```

2.9.4 File Suffixes

The C compiler distinguishes files based upon the final two letters of the file name. File suffix conventions include:

.c	C source files.
.i	C intermediate files, the result of the preprocessor pass.
.s	Assembly language files.
.o	Object files.
.a	Archive and library files.

2.9.5 Compiler Flags

Compiler *flags* are options that you may specify in a command line when invoking the compiler. Both **cc** and **fcc** recognize standard compiler flags and either use them to direct the compilation or pass them on "as is" to the loader. Particularly useful compiler flags include:

-o	Use the following string as the name of the resulting file, rather than **a.out**.
-v	Report on the progress of the compilation, as each pass is completed.

-c	Don't link the resulting object module with any other module--leave it as a **.o** (dot-"o") file.
-S	Leave the assembly language output in files denoted with **.s** suffixes. Viewing a **.s** file is a good way to find out how efficient the compiled code is.
-d	Send the following string to the preprocessor. Used in conjunction with the `#ifdef`, `#elseif`, and `#endif` directives to cause *conditional compilation* of selected portions of your source code.
-O	Optimize the assembly language output before generating object code (see below).
-n	Pass this option to the linker to create a shared text program (see below).
-s	Pass this option to the linker to strip out the symbol table from the object code. This option makes the resulting object code smaller and is useful for producing production quality code files of minimum size.
-R	Set the load address for the resulting object code (default is 2000 hexadecimal). Refer to the memory management chapter of this guide for more information.

2.9.6 Invoking the C Optimizer

The C compiler features an optimizer on the **/c** disc (**/c/bin/c2**). The optimizer can reduce the size of resulting object code by as much as 10%, depending on the type of control structures you're using, and increase execution speed by a corresponding amount.

Typically, you won't want to bother with the optimizer during early code development. The optimizer requires an additional pass during the compilation and causes the compiler to take longer. When you near completion of your project, however, use the optimizer to cut down the space and execution time of your program.

To invoke the C optimizer, use the **-O** flag (uppercase "Oh") in the command line; for example:

```
cc -o outfile -O infile.c
```

What Happens: The above command line directs the C compiler to output an assembly language file named **infile.s** and then to invoke the assembler (**as**) to finish the translation to object code on file **outfile**. Including the **-O** flag causes the **c2** optimizer to process the assembly language file *before* the assembler does its work.

Note: If the **-o** output option isn't specified, the resulting object file will be named **a.out** and will be placed in the working directory from which you invoked the compiler.

It's possible and often convenient to mix the file types sent to **cc**, as in:

```
cc foo.c boo.o hoo.s
```

The compiler is smart enough to know what it needs to do with each file before invoking the linker.

For your information: You can check the resulting size of object files by means of the **size** utility, included on the translator disc (**/c/bin/size**).

2.10 Shared Text Programs--What Are They?

The operating system allows programs to run in *shared text.* If you are running multiple instances of the same program at the same time, there's no need for multiple copies of the same code to reside in memory at the same time.

By specifying the **-n** flag in the **cc** command line, you direct the linker to create a shared text file whose executable code (not data space) can be shared among two or more programs.

Unlike other programs, a shared text program is *not* limited to 8K bytes of user *stack space.*

Refer to Chapter 9, *Memory Management*, for more information on shared text programs.

2.11 Calling C Routines From Other Languages

It's sometimes desirable to do most of your software development in one language (such as BASIC or Pascal) but to call C language subroutines from your application at strategic spots.

Most languages for the Integral Personal Computer operating system allow you to call C subroutines.

2.11.1 Calling C Subroutines From BASIC

HP-UX Technical BASIC allows you to call C subroutines fairly easily. You can pass parameters into the C routine either by address or by value. When dealing with pointers or when you want a return value from the C subroutine, then you should pass parameters *by address.*

Here are the steps:

1. Write and compile your C routine using the **-c** option to produce an object file with a **.o** suffix. For example:

   ```
   cc -c dummy.c
   ```

   --leaves **dummy.o** as the resultant object file, ready to be linked into a binary file for BASIC.

2. Use the linker to link the object code with the necessary libraries by including the **-rdo** linker flags. For example:

   ```
   ld -rdo NewBinary dummy.o -lc
   ```

   --links your BASIC routine with the standard C runtime support library, **/usr/lib/libc.a**. (If you need additional libraries, list their names in front of the **-lc**.) The **-o** flag causes the linker to write the output to a **NewBinary** file in the current directory.

3. Write your BASIC program using the Technical BASIC interpreter.

   a. Dimension all arrays using `OPTION BASE 0`. This dimensioning ensures that both BASIC and C begin their subscripting at the zeroeth element.

   b. Use `LOADBIN "file_name"` to load the executable file that contains the C routine(s).

4. Finally, use the following syntax to call a C routine:

   ```
   CALLBIN "subroutine name" (param1, param2, (param3))
   ```

Additional notes:

- You may skip the `LOADBIN` command by putting the name of your binary file in a text file named **.bconfig** in the current directory. This file contains a list of binary programs to be loaded whenever BASIC is fired up.
- Variables will be passed by to the C routine by address unless they are enclosed within parentheses `()`, as in `(param3)`. When passed by value, the value of the parameter cannot be changed by the C routine.

- You should append an ASCII NULL character[11] at the end of a string before passing that string to a C routine. You can use the `CHR$()` function and the string concatenation operator (&) to set up your string variables. For example:

  ```
  3000 LET MyString$ = MyString$ & CHR$(0)

  3010 CALLBIN "PassString" (MyString$)
  ```

  Even when passing string *constants*, you should append a null character, as in:

  ```
  3020 CALLBIN "FunnyGuy" ("Everett~0")
  ```

  Your C routine can handle the string parameter as a standard character array. Be careful, however, when adding characters to the string in your C routine--BASIC won't know that you've changed the length of the string parameter unless you explicitly keep track of the null character that C uses to mark the end of the string.

- BASIC `INTEGER` variables are represented in C as 32-bit signed integers of type `int`; BASIC `SHORT` variables are 32-bit floating point numbers of type `float` (pass by address) or of type `double` (pass by value); BASIC `REAL` variables are 64-bit floating point numbers of type `double`.

- You cannot expect to use standard output within your C routine (for example, by means of **printf()** statements) or to access any files already opened by BASIC. Unexpected side effects may result.

2.11.2 A BASIC Example

The following is a toy BASIC program that calls a C subroutine to add two numbers:

11. Decimal code 0.

```
/*******************************************************/

10 INTEGER x,y,z
20 DISP "Please enter two numbers";
30 INPUT x,y
40 LOADBIN "binfile"
50 CALLBIN "add_numbers" ((x),(y),z)
60 ! x and y passed by value; z passed by address
70 DISP "Answer is"; z
80 SCRATCHBIN "binfile" ! Get rid of it
90 END

/*******************************************************/
```

Code Sample 2-2. BASIC Program To Call a C Subroutine

The following **add_numbers()** routine does the work:

```
/*******************************************************/

add_numbers (a,b,c)
int a,b,*c;
{
     *c = a + b;
}
/*******************************************************/
```

Code Sample 2-3. Routine To Add Two Numbers

We declare variable `c` a pointer so that the C routine can access and change the storage location of the pass parameter from BASIC.

2.11.3 Calling C Subroutines From Pascal

Assume that you are programming in HP-UX Pascal and want to pass a string from Pascal to a C language subroutine that will **exec()** that string from the shell.

First, you would include declarations such as the following at the beginning of your Pascal source:

```
CONST
   LINE_LENGTH = 132; (* an arbitrary size *)
   ...

TYPE
   E_STRING = packed array [1..LINE_LENGTH] of char;
   ...

PROCEDURE $alias '_ExecShell'$
    call_os(var foo: E_STRING) ; external;
```

Note in particular the `external` procedure declaration that makes the linkage possible.

The `alias` is the name of the entry point in your C source (minus the prepended underscore (_). The procedure name (`call_os`) is the name of the procedure as you will reference it in your Pascal source.

The fact that you are passing the string by reference (as specified by the `var` in the parameter list) means that the routine will pass a *pointer* to the string.

The subroutine itself is written in your C source, similar to the following:

```
/******************************************************/

#include <stdio.h>

ExecShell(bstring)
char *bstring;
{
     write(1,"Execute this string:\015\012",22);
     write(1,bstring,strlen(bstring));
     write(1,"\015\012",2);

#ifdef IPC
if (vfork() == 0) { /* child process -- to use PAM */
     close(0);
     if (execl ("/rom/PAM","PAM","-c",bstring,0) == -1)
          {
          write(1,"\t\t\tNo go...\015\012",13);
          }
     write (1,"\t\t\tSorry--didn't work...\015\012",26);
     _exit (1);
     }
#else /* not IPC */
if (fork() == 0) { /* child process -- to use sh */
     close(0);
     execl ("/bin/sh","sh","-c",bstring,0);
     write (1,"\t\t\tSorry--didn't work...\015\012",26);
     exit (1);
     }
#endif IPC
write(1,"Thanks for calling...\015\012",23);
}
/******************************************************/
```

Code Sample 2-4. Routine to Fork and Exec a Shell Command Line

Finally, the function reference appears as follows in your Pascal source:

```
call_os(YourString);
```

For example, `call_os("Happiness")` would call the PAM or Bourne shell to execute the string "Happiness".

2.11.4 Linking Modules With the Linker

After you've compiled your individual source files using the appropriate language compilers, it's up to the linker to link all the ***.o** files and the appropriate library files into a single executable code file.

The easiest way to invoke the linker is to let the **cc** compiler do it for you! For example:

```
cc -o MyProg file1.o file2.o -lpc
```

--causes the C compiler to notice that no compiling needs to be done--only linking--and to invoke the linker. The linker in turn resolves external function references--the result is a single executable file named **MyProg**.

In this example, the `-lpc` option causes the linker to search the **/usr/lib/libpc.a** library file for the Pascal runtime support routines. (The **libc.a** C runtime library is automatically searched.)

2.12 Performance Considerations

When you use external memory as provided by the external expansion box, the CPU is forced to wait for external RAM accesses. The delay may slow performance by as much as 30 per cent.

If performance is an issue, install any extra memory cards in the computer mainframe and forego the expansion box.

2.13 Further Information

To learn more about the PAM shell, refer to the owner's documentation for your computer.

To learn more about **make** and the Source Code Control System (SCCS), refer to the *Software Development Tools* volume of the *HP-UX Concepts and Tutorials* manual set.

To learn more about the C compiler and the HP-UX tools discs, refer to the getting started guides for the compiler and the tools.

To learn more about the Bourne and C shells, refer to the *Shells and Miscellaneous Tools* volume of the *HP-UX Concepts and Tutorials* manual set.

To learn how to call C language subroutines from another language, refer to the appropriate language reference manual.

Chapter 3

Windows

3.1 Introduction

Windows are an important part of the Integral Personal Computer, enabling separate applications to run cleanly in a multi-tasking environment. For example, a user can quickly switch between a spreadsheet program running in one window and a calculator program running in another. You as a software developer can be running vi in one window, compiling in another, and running your application in a third.

The Integral Personal Computer offers your application three window types:

alpha windows	For displaying text. Alpha windows (sometimes called *Term0* windows) emulate HP 2622 terminals without block or format mode. You control alpha windows using HP and ANSI escape sequences, fast alpha library routines, TERMCAP entries, or any combination.
graphics windows	For drawing lines and performing raster operations. You control graphics windows using the HP-GL plotter language, low-level system calls, or both.
alpha/graphics windows	For displaying text and performing graphics operations at high speeds. You control alpha/graphics windows using low-level system calls.

Each window type has its own *window driver* built into ROM that supports different capabilities. However, *all* window types can be manipulated with the same *window manager routines.*

The window manager is an extensive body of code within the operating system that handles windows in the display--their size, location, color, status--their very existence.

Between the time your user presses a key or moves a mouse and the time your application finds out about it, the window manager plays an important role. Typically:

1. The HP-HIL driver (or "keyboard driver") detects keyboard, mouse, or other user activity and passes control to the window manager.
2. The window manager decides what to do with the user input. If necessary, the window manager calls the display driver to update the display immediately, as when the user selects a different window. From a few pixels to the entire screen, the window manager controls the look (or user interface) of the display.
3. The window manager passes keyboard information to the window driver for the appropriate window type--that is, the window manager calls the alpha, graphics, or alpha/graphics window driver.[1]
4. The window driver decides what it's going to do with any keyboard information before it returns control to your application.
5. Finally, if your application is reading from **stdin**, your application will receive the keyboard input, but only *after* the input has been handled by the HP-HIL driver, the window manager, and the window driver.

As you can see, the window manager is the "ring leader" of many system activities. The window manager features a programmatic interface that allows your application program to take control of its window, including its size and placement in the display.

Note right now that your application can probably run *as is* on the computer without having to call *any* window manager routines. In particular, applications that are written for 24-line by 80-column HP 2622 alpha terminals and don't require block mode or format mode will port easily to the computer. Windows in general, and alpha windows in particular, are designed to emulate standard UNIX tty device types.

On the other hand, if you're interested in writing a "window-smart" application, then read on. This chapter teaches how to create and manipulate windows from your application.

1. If the application has so requested, the window manager can interrupt application execution to jump to a user-defined signal handler, as when the user moves a mouse.

Using either prewritten *window library* routines or low-level *system calls*, your application can:

- Create one or more windows of a given type, size, and location on the display.
- Change the characteristics of an existing window.
- Define and control the display pointer associated with a given window.

3.2 Windows and PAM

It's reasonable to assume that users of your window-smart application will be invoking your application from PAM--the window-smart shell.

Starting an application from PAM:

1. Ensures that the application runs in its own window.
2. Ensures that the application inherits the current font, color, and window border[2] of the PAM window itself. (As this chapter shows, however, your application has the final say in how its window looks.)

Neither the Bourne shell nor the C shell is window-smart. All processes started from the Bourne or C shell will run in the same window as the shell. Thus, whatever your application does to *its* window (such as changing window size) will affect the look of the Bourne or C shell window.

If, for example, your application changes its default window into a tiny window and if the user invokes your application from the Bourne shell, then when your application exits, the user will be left with a tiny Bourne shell window.

3.3 Programmatically Speaking

You can program window manipulations in two ways:

- By using the window *library routines* included in the Windows/9000 library for the Integral Personal Computer.

2. Window borders are not part of Release 1.0.0 of the window manager.

- By using low-level *system calls* to the window manager to carry out your work.

We recommend the first way--using library routines--because your code will end up more "portable." That is, you will be able to port your application easily to other HP-UX systems.

The second way--using **ioctl()** system calls--means that your code will rely on hardware dependencies that will make porting more difficult.

Regardless of your technique, your "handle" to a given window is its *file descriptor*. The operating system regards each window in the display as an HP-UX character device. When your application opens a window, the operating system returns a unique file descriptor for that window.

The file descriptor is a small integer that can specify either a window (as in our case), a disc file, or a physical device. The Integral Personal Computer allows up to 20 file descriptors to be open at a time for a given application.

Calls to window library routines and window manager routines always involve the file descriptor for the window. For example, if you want to define the appearance of the *display pointer* (or mouse cursor) for a window, you have to specify the file descriptor for the desired window.

By default, the name of the window as the user sees it is installed in the **/dev/screen** directory. For example, if you look now, you'll see the name **/dev/screen/PAM** as the complete path name of the PAM window device.

For Release 5.0 of the operating system, the `WMDIR` variable in the shell environment specifies the default directory for installing window device files.[3] Included in the **/rom/.environ** file is the following assignment:

```
WMDIR=/dev/screen
```

We recommend that you leave the `WMDIR` setting alone, unless there's a specific need to alter it.

3.4 Manipulating Windows With Library Routines

The Windows/9000 library for the Integral Personal Computer provides a

3. The `WMDIR` variable is not part of Release 1.0.0 of the operating system.

number of high level windowing routines that you can use to manipulate windows.

For example, to find the name of a given window, you can use the **wgetname()** library routine:

```
int wgetname (fd, name)

int fd;
char *name;
```

Pass parameter `fd` is the file descriptor of the desired window and `*name` is the character string that will be filled out by the **wgetname()** routine.

Using the window library ensures that the software you write to run on the Integral Personal Computer will port easily to other HP-UX 9000 computers.

Contact your Hewlett-Packard Sales Representative for more information about the Integral Personal Computer Windows/9000 library.

3.5 Manipulating Windows With Low-Level Routines

You can access the power of the Integral Personal Computer window manager directly by using low-level **ioctl()** system calls. The following pages show how to create windows and change window characteristics using low-level calls.

Note

The following routines are specific to the Integral Personal Computer operating system and will not port to other HP-UX systems.

Be sure that you isolate these system-dependent routines in a separate module of your software.

In addition to its unique file descriptor, each window is associated with a unique *window structure* of type `windio`.

The important `windio` structure is declared as follows in **<scrn/wmcom.h>**:

```
struct windio {

    char w_path[40];    /* path name for node */

    long w_minor;       /* window number */

    short w_type;       /* type of window */

    unsigned short w_stat;  /* window status */

    short w_xloc;       /* left edge of desired window */

    short w_yloc;       /* top of desired window */

    short w_height;     /* height of desired window */

    short w_width;      /* width of desired window */

    short w_gen1;       /* general purpose var */
    short w_gen2;       /* general purpose var */
    short w_gen3;       /* general purpose var */
    short w_gen4;       /* general purpose var */
};
```

The `windio` structure contains such information as the *name*, *type*, and *dimensions* of the window.

The code samples that follow in this chapter rely on a few header files and two global variables:

```
/*******************************************************/

#include <stdio.h>      /* the usual */

#include <scrn/wmcom.h> /* very important! */

#include <fcntl.h>  /* for opening file descriptors */

#include <errno.h>  /* for reporting error conditions */

extern int errno;   /* global variable--optional
                    ** on Rel. 5.0.  However,
                    ** on Rel. 1.0.0 you must
                    ** declare 'errno' explicitly.
                    */
int w_fd;      /* for your window file descriptor */

/*******************************************************/
```

Code Sample 3-1. Some Window Declarations

Most window data types and constants are declared in the **<scrn/wmcom.h>** header file.

In this and following chapters, the appropriate file descriptor (in this case `w_fd`) is always defined at the outset as a global variable so that subsequent routines can easily use it.

The `errno` system variable is set by the operating system whenever an error occurs. Possible error values are defined in **<errno.h>**.[4] Refer to the **errno** file on the C language documentation disc for more information.

3.5.1 Creating a Window With Low-Level Routines

You have a great deal of flexibility in creating a window. This section shows a few options.

There are four steps involved in creating a window using low-level window manager routines:

4. For Release 1.0.0 of the C compiler, it's necessary to declare the `errno` variable externally in each file that needs to reference the variable.

1. Get an existing `windio` structure for your application using an **ioctl()** system call.
2. Define the desired attributes for your window.
3. Create the window using a second **ioctl()** call.
4. *Open* the newly created window with an **open()** call, which will return the file descriptor for the newly created window.

After these steps, you can start *using* the new window by means of the window's file descriptor.

The following **NewWindow()** routine creates an alpha window and returns the file descriptor of that window.

A typical invocation looks like:

```
w_fd = NewWindow("barbara", 20, 300, 0, 200);
```

--which creates a 20 by 300-dot alpha window named **barbara** at the lower left edge of the display.

```
/******************************************************/

NewWindow(newname, height, width, xloc, yloc)
char *newname;
int height, width, xloc, yloc;
{
    int fd;   /* the file descriptor we'll return */

    extern char *strrchr(); /* for the name */
    char *dummy;

    struct windio w;
    /* local var for the window structure */

    /* get the default window attributes */
    ioctl (0, WMGET, &w);

    /* now set the fields -- first the name */

    /* find the last slash in the path name */
    dummy = strrchr(w.w_path, '/');
    /* zero out the end of the current name */
    *(++dummy) = 0;

    /* now add the name our calling routine wants */
    strcat (w.w_path, newname);

    w.w_type   = TOTYPE; /* create an alpha window */
    w.w_stat  |= AUTO_ACT | AUTO_DEST | AUTO_FRONT;
    w.w_xloc   = xloc;
    w.w_yloc   = yloc;
    w.w_height = height;
    w.w_width  = width;

    /* use default buffer size */
    w.w_gen1   = 0;
    w.w_gen2   = 0;

    /* actually create the new window */
    if (ioctl (0, WMCREATE, &w) < 0 )
     {
     printf("Can't create window %s.\n", newname);
     printf("Error is %d.\n", errno);
     return (-1);
     }

    /* open the named window for
    ** reading and writing
    */
    fd = open (w.w_path, O_RDWR);

    /* return the all-important file descriptor */
    return (fd);
}
/******************************************************/
```

Code Sample 3-2. Routine To Create an Alpha Window

The input parameters to **NewWindow()** are as follows:

`newname`	The simple name of the window, as it will appear in the window banner.

`height`	The physical size of the window in the vertical direction, measured in pixels.
`width`	The physical size of the window in the horizontal direction, measured in pixels.
`xloc`	The absolute horizontal position of the window on the display, measured in pixels.
`yloc`	The absolute vertical position of the window on the display, measured in pixels.

Here are the details:

- The window is created as an *alpha window* (of type TOTYPE). Other possibilities include *graphics window* (of type PETYPE and *alpha/graphics window* (of type AGTYPE).[5]
- The location and size fields are in terms of pixels, or dots. Use as dimensions 512 by 255 pixels (width by height). The absolute origin of the entire display, *(0,0)*, is anchored at the *upper left corner* of the display.
- There is a field in the `windio` structure--`w_path`--that is filled out by the **WMGET ioctl()** call with the absolute path name of the window. The default name will be **/dev/screen** plus the simple name of your application, such as **/dev/screen/MyProg.**
- The path name of the window is logically tied to the *file descriptor* of the window by means of the **open()** call.

  In the code sample, the **NewWindow()** routine returns the file descriptor for the new window, which will be used in all subsequent communications between your application and the window.
- You may choose *not* to name the window from your application. A problem can arise--what happens if the user wants to run *two* instances of your application? If you use the same name for both windows, the second window create will fail. If instead you let PAM do the naming for you, then everything will be OK. For example, if the file name of your program is **mickey**, then PAM will give the first instance of your program the window name **mickey**, the second instance **mickey!**, the third instance **mickey!!**, and so

5. The window types are declared in <scrn/wmcom.h>. The AGTYPE is available only on Release 5.0 of the operating system.

on, avoiding window name collisions.

- Your application begins its career running in an *alpha window.* Although your application can create any number of windows, it always has available its original alpha window. This alpha window, as a **tty** device, has associated with it the following special file descriptors:

  - Standard input (**stdin**), or *0.*

  - Standard output (**stdout**), or *1.*

  - Standard error (**stderr**), or *2.*

  For example, if your application writes to **stdout** using a **printf()** statement, the output will go to the startup alpha window.

- Fields `w_gen1` and `w_gen2` of the `windio` structure set the *buffer size* for the window (which may be different than the *physical size* of the window). For terminal (or alpha) windows, `w_gen1` specifies *rows* and `w_gen2` specifies *columns* of buffer size. When both parameters are `0` ("zero"), then the buffer size defaults to 48 rows by 80 columns (two "pages" of screen memory), the standard size for HP 2622 terminals.

- It's a good idea to check for errors when you create and open a window. The return value of the **ioctl()** system call will be *-1* if something goes wrong. Similarly, you can check the return value of **NewWindow()** itself--if the **open()** fails, the routine will return a *-1.*

The `w_stat` field of the `windio` structure exerts a great deal of control over the appearance and the behavior of the associated window.

Following are the possible window attributes for the `w_stat` field, as defined in **<scrn/wmcom.h>**:

Attribute	Description
INVERT	Set window background to black.
ACTIVE	Make the window active (connect it to the keyboard).
AUTO_ACT	Make the window active when it is written to (the default). When set and the window is written to, the window is connected to the keyboard. The bit is cleared when used.
FRONT	Put the window in front on the display.
AUTO_FRONT	Put the window in front when a write occurs (the default). The bit is cleared with used.
CORK	Pause/block/suspend all output to the window.
ON_SCREEN	Put the window on the screen.
NO_DESTROY	Make the window "indestructible" (so that [Stop] and [Shift][Stop] have no effect).
AUTO_DEST	Destroy the window immediately when closed or the program exits. (The default is ~AUTO_DEST.)
IGNORE_KEY	Don't activate automatically on hide, etc.

Attribute	Description
NO_BANNER	Turn off the banner.
RECOVER	Destroy the window when no longer in use (the default). The destroy is delayed until PAM executes another command or updates the PAM window.
SHRUNK	Shrink the window to the banner only.
KEY_SCAN	Put the keyboard in key scan mode.
NO_MOVESTRETCH	Disallow move and stretch of the window. (Not available on Release 1.0.0 of the operating system.)
ICON_BORDER	Turn on the window border. (Not available on Release 1.0.0 of the operating system.)

The previous **NewWindow()** code sample sets a few of these "bits" in the 16-bit `w_stat` field structure by means of the bitwise OR operators, |= and |. The effect in this case takes place upon execution of the WMCREATE **ioctl()** call.

You may instead wish to *clear* a bit in the `w_stat` mask. For example, to cause a window to remain in the display long after program termination, you need to clear the RECOVER bit. The corresponding C code uses the the bitwise AND operator and the one's complement operator, ~:

```
w.w_stat &= ~RECOVER
```

The effect is the same as if the user presses the $\boxed{\text{Save}}$ key on the System menu.

A multitude of combinations is possible. As a final example, you can force your window to be offscreen and invisible with the following:

```
w.w_stat &= ~(ON_SCREEN | AUTO_FRONT)
```

In contrast to setting or clearing the `w_stat` bits, you can *check* the bits after an WMGET **ioctl()** call whenever you want to examine window status. Typically, WMGET and WMCREATE **ioctl()** calls work in pairs, as do WMGET and WMSET **ioctl()** calls.

The window manager has a set of "sanity rules" that it enforces on windows. In other words, you can set the `w_stat` bits only in ways that make sense. For example, a window cannot be both offscreen (~ON_SCREEN) and in front (FRONT).

Following is a small driver program that calls the **NewWindow()** routine:

```
/*******************************************************/

#define MESSAGE "Is everybody happy?\n\n"

extern int w_fd;
extern int NewWindow();

main()
{
    char buf[50], ch;
    int HE, WI, XL, YL;
    int i;

    for (i = 1; i < 10; i++)
    {
        printf("Please enter the name, ");
        printf("height, width, xloc, and yloc, ");
        printf("separated by spaces: \n");

        scanf("%s%d%d%d%d", buf, &HE, &WI, &XL, &YL);

        w_fd = NewWindow (buf, HE, WI, XL, YL);
        /* w_fd is our global var */

        if (w_fd < 0)
            printf("Can't do it--Sorry!\n");
        else
        {
        write (w_fd, MESSAGE, sizeof(MESSAGE));
        }
    }
    pause();    /* be patient */
}
/*******************************************************/
```

Code Sample 3-3. A Window Program

The program uses the file descriptor returned from the **NewWindow()** routine for its window input/output.

Additional notes:

- Window names, like simple file names, may consists of 1 to 14 letters, digits, and the following characters: . (period), _ (underscore), ! (exclamation mark), and - (hyphen).
- The default name for a window is simply the name of the program as invoked from the PAM shell.
- Each window must be given a *unique* window name; otherwise, the **ioctl()** WMCREATE call will return *-1* and errno will be set to 17, defined in **<errno.h>** as EEXIST.
- When the window is eliminated, the name will be removed from the WMDIR directory, typically **/dev/screen**.

3.5.2 Changing Window Characteristics With Low-Level Routines

It's easy to change the characteristics of an existing window using the following sequence:

1. You "get" an existing windio window structure using an WMGET **ioctl()** system call.
2. You assign a new value to one or more of the window structure fields.
3. You "set" the new structure with a WMSET **ioctl()** system call.

The following **ChangeWindow()** routine changes the characteristics of an existing window. The assumption is that you've already opened the window and are using its file descriptor.

```
/******************************************************/

ChangeWindow()
{
      struct windio w;
      /* local window structure */

      /* get the default window params */
      ioctl (w_fd, WMGET, &w);

      /* now alter the chosen parameters */
      /* first, give it a new name */
      strcat (w.w_path, ".new");

      /* now invert the color of the window */
      w.w_stat   |= INVERT;

      /* set a new location */
      w.w_xloc    = 25;
      w.w_yloc    = 100;

      /* set a new size */
      w.w_height = 75;
      w.w_width  = 250;

      /* OK--make the changes--NOW */
      ioctl (w_fd, WMSET, &w);
}
/******************************************************/
```

Code Sample 3-4. Routine To Change the Characteristics of an Existing Window

The changes will go into effect as soon as the system executes the WMSET **ioctl()** call.

3.6 Handling Multiple Windows

Your application can create one or more windows to handle as it pleases. For example, you can create an alpha window to display text and handle keyboard interaction and create a graphics window to handle plotting routines.

Managing multiple windows is fairly straightforward:

- You create multiple windows by creating and opening one window at a time:
  1. Issue a WMGET request of the form:

     ```
     ioctl (0, WMGET, &w)
     ```

     --to get a new, "ready-made" window structure.
  2. Fill out the fields of the `windio` structure as desired.
  3. Issue a WMCREATE request of the form:

     ```
     ioctl (0, WMCREATE, &w)
     ```
  4. Open the new window with an **open()** call.
- After creating your windows, manipulate the individual windows by their unique file descriptors, returned from the **open()** call.
- You can set up a **signal()** routine to handle mouse inputs, as shown in Chapter 4, *Keyboard, Mouse, and Display Pointer*. When an interrupt occurs, you poll each window (as with a series of WMEVENTPOLL **ioctl()** calls) to find out whether the window has something to say to you.

3.6.1 Rules of Etiquette

In a multi-tasking, windowed environment, the user expects well-behaved applications.

Your user will probably run other applications while running your own and may in fact try running two or more instances of your application at the same time. To be a good citizen, don't let your application be a display, keyboard, or system hog!

Try putting yourself in your user's shoes--see what it's like to run your application in a busy system. If you can comfortably run two instances of your application at the same time, you know you've done well in your coding practices.

In closing, the idea is to keep the user *in control* of the computer. OK?

3.6.2 Sending Strings to Other Windows

There may be times when you'd like to send a character string to another window in the display. For example, the HP-UX Calculator program offers a $\boxed{\text{Send}}$ function key. Pressing $\boxed{\text{Send}}$ starts the calculator cycling through the current window "stack" and enables the user to send calculator results to selected windows.

The following **SendString()** routine sends a given character string to a named window (as the simple name appears in the window banner).

```
/*******************************************************/

#include <scrn/wmcom.h>
#include <fcntl.h>

SendString(str, wname)

char *str;   /* string to send */
char *wname; /* name of target window */
{
    int fd;  /* temporary file descriptor */

    struct windio wparms;
    register char *p, *q, *name;

    char ostr[256];  /* an arbitrary length
                     ** output string
                     */
    /**********************************************
    ** first open the window manager "device node"
    ** and get information on the desired window
    */
    fd = open("/dev/screen/wm", O_RDWR);

    /* form path name of window */
    strcpy(wparms.w_path, "/dev/screen/");
    strcat(wparms.w_path, wname);

    /* get window parameters of target window */
    ioctl(fd, WMGET, &wparms);

    /* make sure it's active */
    wparms.w_stat |= ACTIVE | ON_SCREEN;

    /* un-hide it, too. */
    wparms.w_stat &= ~SHRUNK;

    /* now set new window parms */
    ioctl(fd, WMSET, &wparms);

    /* close the window manager */
    close(fd);

    /*************************************************
    ** now open the keyboard
    ** and write the string to it
    */
    fd = open("/dev/screen/keyboard", O_RDWR);

    /* set pointer to start of input string */
    p = str;

    /* set pointer to start of output string */
    q = &ostr[0];

    while (*p)
    /* translate the string from ascii chars to
    ** 16-bit IPC-specific keycode values.
    **
    ** (Each keycode has a zero high-byte
    ** indicating a  normal typing key.)
    */
     {
     *q++ = '\0';
     *q++ = *p++;
     }
    /* write the string to the keyboard */
    write(fd, ostr, q - ostr);

    /* finally, close the keyboard */
    close(fd);
}
/*****************************************************/
```

Code Sample 3-5. Routine To Send a String to a Specified Window

This routine relies on a simple trick: It selects the desired window programmatically and then writes to the keyboard (the HP-HIL driver). In effect, the routine is "pressing the keys" of another application.

Additional notes:

- The routine forces the target window to the front and makes it active. These steps ensure that the target window is connected to the keyboard.
- The routine allows transmission only of normal typing keys (including shifted and [CTRL] characters).
- A major side effect of the routine is to leave the target window *active*--that is, in front of the display and with the keyboard connected to it. Probably this end result is exactly what you want.[6]
- This routine does *no* error-checking. If, for example, your application specifies a nonexistent window name, then this routine will have a net effect of nothing.
- This routine is not sophisticated enough to guarantee absolutely that the right window will receive the specified string. Although the chance is slight, in a "window active" system, the user could select a window at the wrong time and *that* window would be sent the character information.
- The idea of "opening the window manager" may seem strange, but opening **/dev/screen/wm** is an easy way to access the file descriptor of your window and of other windows in the display. Refer to the *Window Manager Reference Manual* for more information.
- You may not wish to hardcode **/dev/screen/keyboard** and **/dev/screen/wm** into your source (although these path names should be OK in most instances). A safer approach is to read the `WMDIR` environment variable and take *that* value to form the path name of the **keyboard** and **wm** device nodes.

3.7 Calling It Quits

In the exit routine of your application, you should clean up a bit and close the file descriptor for your window. The following **CloseWindow()** routine does just that:

6. In HP-UX Calculator, for example, the assumption is that the user chose the calculator for temporary work, wants to fire off the results to another window, and then wants to switch immediately to the target window.

```
/*******************************************************/

CloseWindow()
{
    /* home up and clear the alpha window */
    write (w_fd, "\033h\033J", 4);

    /* close the window */
    close (w_fd);
}
/*******************************************************/
```

Code Sample 3-6. Routine To Close an Alpha Window

If you've created an *autodestroy* window, as in the beginning **NewWindow()** code sample, then executing the **close()** in this routine will cause the window to be eliminated from the display.

3.8 Further Information

For learning how to control window characteristics from the *keyboard* (using the [Move], [Stretch], [Invert], [Save], and [Hide] function keys, refer to the owner's documentation for the computer.

This chapter contains an example of creating an alpha window. For instructions on creating graphics windows and alpha/graphics windows, refer to the chapters for those window types.

For learning more about the fields in the `windio` structure, refer to the *Window Manager Reference Manual.*

For learning more about the keyboard and mouse, refer to the next chapter of this guide.

Chapter 4
Keyboard, Mouse, and Display Pointer

This chapter shows how to control the keyboard, mouse, and display pointer from your application program.

The keyboard and mouse are *connected* to one window at a time--the active window. If that happens to be your application window, then you have complete access to what the user is doing with the keyboard and mouse.

The display pointer (sometimes called the *sprite* or *mouse cursor*) is logically tied to the mouse. As the user moves the mouse around his/her desktop, the display pointer tracks the movement around the display. When your application window is active, your application has control over the appearance and position of the display pointer.

It's *not* necessary to read this chapter if your application relies on the standard HP-UX device model of a **tty** terminal device and doesn't need to use the mouse or display pointer. That is, you can issue **read()** statements of the form:

```
read(0, Char_Pointer, Number_Bytes);
```

--and receive character information from the keyboard. The 0 ("zero") specifies *standard input.* Each alpha window is built on the model of a terminal device and provides your application with standard input from the keyboard.

Refer to the individual chapters on the various window types to determine:

- How to throw a particular window type into *raw mode* in order to capture special, non-typing keys.
- How to read keystrokes as they are typed, without echoing characters to the display and without requiring carriage returns.
- How to interpret keyboard reads for a particular window type.

The following pages show how to handle inputs from the mouse and how to handle *mixed* inputs from the keyboard and mouse.

Note

The following routines are specific to the Integral Personal Computer operating system and will not port to other HP-UX systems.

Be sure that you isolate these system-dependent routines in a separate module of your software.

4.1 The HP-HIL Driver

The keyboard and mouse are Hewlett-Packard Human Interface Loop (*HP-HIL*) devices that are controlled by the built-in HP-HIL driver.[1] The HP-HIL driver supports up to seven devices connected in "daisy-chain" fashion to the computer.

When an application program reads from the *keyboard*, the window driver for the particular window type (for example, the alpha window driver) issues a call to the HP-HIL driver to handle the details of the keyboard read.

When an application program wishes to access the *mouse*, the application issues a low level call to the window manager, which in turn calls the HP-HIL driver to handle the details of the mouse interaction.

The HP-HIL driver handles communication with the keyboard and mouse at a sufficiently low-level so that your application shouldn't have to deal directly with the HP-HIL driver at all.

4.2 Keyboard Mappings

4.2.1 Alpha Window Mappings

If your application is running in an *alpha* window, then the keyboard-to-character-code mappings are defined to be either:

1. Sometimes referred to as the "keyboard driver" or the "caravan loop driver."

- HP 2622 terminal mappings, or
- ANSI console mappings

--depending on whether the alpha window is set to HP or ANSI mode.

Refer to the *Term0 Reference Manual* for a description of alpha window keyboard mappings.

4.2.2 Graphics Window and Alpha/Graphics Window Mappings

If your application is running in "raw mode" in a *graphics* window or an *alpha/graphics* window, then the keyboard-to-character-code mappings are defined in the `<scrn/keycode.h>` header file.

Refer to Chapters 6 and 7 in this guide for more information on handling keyboard inputs for graphics windows and alpha/graphics windows.

4.3 Localized Keyboards

There are more than *18* localized keyboards available for the computer. At power on, the operating system calls the HP-HIL driver to determine the identity of the keyboard connected to the HP-HIL loop.

Based on the information returned from the HP-HIL driver, the operating system reads the appropriate keymappings from a large table in ROM.

From then on, the character codes transmitted by the typing keys will be correctly matched to the key faces on the keyboard.

If your application itself needs to determine the type of keyboard (German, French, etc.) connected to the computer, then the application must open the HP-HIL driver (**/dev/keyboard**) and check the *identifier byte* from the `device_info` structure.

For more information, refer to the HP-HIL file on the C language documentation disc.

4.4 A Word About Keyscan Mode

A highly specialized application may require absolute control over the keyboard, more than provided by the *raw mode* of an individual window.

The window manager and HP-HIL driver provide such a facility, known as *keyscan mode*. When in keyscan mode, your application intercepts *all* keystrokes as they are pressed. The application can detect presses of the [CTRL] key, the [Caps] key, the [Menu] key, the [User/System] key, and other system keys that are normally processed by the operating system instead of being passed on to your application. Your application can even distinguish between the left and right [Shift] keys!

Because such a high degree of keyboard control is rarely needed, *we advise against using keyscan mode*--your application has plenty of control using the keyboard routines provided by the individual window types.

To enter keyscan mode, an application simply sets the `KEY_SCAN` bit in the `windio` structure for the application window and then reads directly from the HP-HIL driver. Keyscan mode **read()**'s consist of special two-byte keycodes for each key press.[2]

However, if you throw your application into keyscan mode, your application has to assume a great deal of responsibility. For example, when the user presses the [Shift][Select] keystroke, normally the window manager intercepts the keystroke and causes the windows in the display to "shuffle." But in keyscan mode, *your application* has to know about system keystrokes and specifically be ready to handle the [Shift][Select] keystroke on its own. In effect, the application will take over the *whole system*.

For more information on keyscan mode, refer to the **hp-hil** and the **wm** files on the C language documentation disc.

4.5 Introduction to the Mouse

The HP-HIL mouse is a two-button input device. The movement of the mouse is tied to the *display pointer* on the display.

When an application window is the *active window*, that application can take advantage of the mouse and the display pointer. The application can:

2. Keycodes for system keys are listed in `<scrn/keycode.h>`.

- Detect the presence or absence of a mouse.
- Detect any physical movement of the mouse.
- Detect the press (or release) of either the left or the right button.
- Control the placement of the display pointer on the display as well as its shape.
- Ignore the mouse altogether (the default).

Using mouse routines in combination with the display pointer, an application can perform a large number of functions to enhance "friendliness" and usability.

The mechanisms that enable mouse control are the **ioctl()** and **signal()** system calls and the built-in window manager routines.

In this chapter, the term *mouse event* means any change in mouse status--whether just rolling the mouse or pressing a mouse button.

The term *mouse signal* means an interrupt in program execution that occurs due to a prespecified mouse event--that is, the jump to a user-defined routine caused by a desired mouse activity.

Controlling the mouse involves:

1. Defining the mouse event or events that you want to look for.
2. Enabling a mouse signal so that the desired mouse events will cause an interrupt.
3. Saving, processing, or ignoring mouse events as they occur.

4.6 Mouse Declarations

You can find the declarations for the relevant mouse structures in `<scrn/wmcom.h>`:

```
/*******************************************************/

struct m_stat {
   double s_xloc;  /* x coord. of sprite */
   double s_yloc;  /* y coord. of sprite */
   unsigned short s_buttons;  /* button status */
   unsigned long s_mask;      /* interrupt mask */
   unsigned long s_event;     /* interrupt reason */
};

struct m_event {
   unsigned long e_event; /* desired event */
   short e_count;  /* no. of times event has occurred */
   double e_xloc;  /* sprite x last time event occurred */
   double e_yloc;  /* sprite y last time event occurred */
};

struct msrel_stat {
   int m_exist;  /* mouse exists (number of buttons) */
   int m_xshift; /* horizontal multiplier */
   int m_yshift; /* vertical multiplier */
};
/*******************************************************/
```

Code Sample 4-1. Predefined Mouse Structures

Note that the **m_stat** and **m_event** structures include 32-bit fields for the x-y location of the display pointer:

- For alpha windows, **s_xloc**, **s_yloc**, **e_xloc**, and **e_yloc** are measured in columns and rows.
- For graphics and alpha/graphics windows, **s_xloc**, **s_yloc**, **e_xloc**, and **e_yloc** are measured in pixels.

Note that your application can read and set the x-y coordinates of the display pointer anywhere on the display, as long as your application is the active window. Coordinates are measured *relative to your window*. The upper left corner of the window is defined as *0, 0*.

A negative coordinate means that the display pointer is to the *left* or *above* your window area. A "large" positive coordinate means that the display pointer is to the *right* or *below* your window area.

The difference between the **m_stat** and **m_event** structures above is that you can use **m_stat** to set and check the *current* status of the mouse/display pointer while you can use **m_event** during an interrupt routine to find out what has happened.

4.7 First, Is There a Mouse?

Unless your application demands it, it's a good idea to assume that a number of your users *won't* have a mouse. The following **mouse_there()** routine returns "1" if there is a mouse available and returns "0" otherwise.

```
/****************************************************/

#define TRUE 1
#define FALSE 0

#include <scrn/wmcom.h>

extern int windo_fd;
    /* file descriptor of your window */

mouse_there()
{
     struct msrel_stat mouse_params;

     ioctl (windo_fd, WMGETMSREL, &mouse_params);

     return ((mouse_params.m_exist < 0) ? FALSE : TRUE);
}
/****************************************************/
```

Code Sample 4-2. Routine To Detect the Presence of a Mouse

Note: This **mouse_there()** routine and the following routines in this chapter assume that you are addressing the appropriate window by means of the *file descriptor* for that window. (The code samples all use variable `windo_fd`.) Refer to the Chapter 3, *Windows*, to learn how to use the **open()** statement to get a file descriptor for your window.

Note the keyboard equivalents of certain mouse events:

- Pressing the [CTRL]-arrow keys is equivalent to rolling the mouse on a desktop.
- Pressing the [Select] key is equivalent to pressing the left mouse button.
- Pressing the [Menu] key is equivalent to pressing the right mouse button.
- Pressing one of the *shifted* or *unshifted* function keys is equivalent to "mousing" the function key *labels* at the bottom of the display.
- Pressing the [User/System] key is equivalent to "mousing" the status block at the bottom center of the display.

Note also that Release 1.0.0 of the operating system *always* indicates that a mouse is present.

4.8 An Extended Mouse-Handling Example

The following sections include three routines that work together to detect and save mouse events:

mouse_set()	Defines the mouse events that you want to examine.
mouse_event()	Saves the x-y coordinates of the display pointer and the specific mouse event whenever a mouse signal occurs.
mouse_inquire()	Returns information about the oldest mouse event.

The example is designed to give you a clear enough understanding to enable you to tailor a similar set of routines to your applications needs.

An additional routine, **mouse_poll()**, enables you to monitor mouse and display pointer activity *constantly* for specialized applications.

4.9 Defining Mouse Events

To specify what mouse events you want to detect, you can use the following **mouse_set()** routine. Each call of **mouse_set()** will define one mouse event.

For example, if you are interested in the lifting of the right mouse button, then invoke this routine with:

```
mouse_set (EVENT_B2_UP);
```

Following are possible parameters for the function, as defined in `scrn/wmcom.h`:

```
/*****************************************************/

EVENT_B1_DOWN  /* button 1 pressed */
EVENT_B1_UP    /* button 1 released */
EVENT_B2_DOWN  /* button 2 pressed */
EVENT_B2_UP    /* button 2 released */
/*****************************************************/
```

Code Sample 4-3. Predefined Mouse Events

There are other predefined mouse events in the header file, but the above four are the most useful.

At the beginning of your program, you can define as many mouse events as you'd like by calling **mouse_set()** repeatedly, each time with a different parameter.

```
/****************************************************/

#include <signal.h>
#include <scrn/wmcom.h>

extern int windo_fd;
    /* file descriptor of your window */

int mouse_event();
    /* function called upon mouse events */

mouse_set (mask)
unsigned long mask;
{
     struct m_stat mouse;

     ioctl (windo_fd, WMGETMOUSE, &mouse);
     mouse.s_mask |= mask;  /* Logically OR the value */

     ioctl (windo_fd, WMSETMOUSE, &mouse);
     signal (SIGMOUSE, mouse_event);
}
/****************************************************/
```

Code Sample 4-4. Routine To Define a Mouse Event

The code is typical of many **ioctl()** calling sequences:

1. Get the current structure of the mouse.
2. Define one (or more) of the fields of that structure.
3. Set the mouse structure to the new values.

The **signal()** at the end of the routine causes subsequent mouse events (those that you've defined) to activate the **mouse_event()** routine.

An "interesting" mouse event at one point in your program may not be interesting at another point. For example, at times you may be interested in the *press* of the left mouse button and at other times in its *release*.

Dynamically changing your application's concept of "interesting" events is easy:

1. Write a slightly different **mouse_set()** routine that simply *assigns* the value of the mask to the `s_mask` field, rather than OR-ing it into the field. The assignment statement will thus overwrite all previous settings.

2. Call the original **mouse_set()** routine above to add on other interesting mouse events.

4.10 Saving Mouse Events

Chances are that you don't *want* to deal with a mouse event until you're ready for it. The following technique enables you to save current mouse events until you're ready to process them one by one.

The example uses a *circular queue* to save the x-y coordinates of the display pointer and the most recent event that caused the interrupt.

```
/*****************************************************/

#define QUEUESIZE 20
    /* keep track of the last 20 mouse events */

struct queue_element
{
     int x;
     int y;
     unsigned long event;
};

static struct queue_element queue[QUEUESIZE];
    /* the queue is an array */

static queue_element *queue_head = queue;
static queue_element *queue_tail = queue;
    /* 'queue_head' and 'queue_tail'
    ** initially point to beginning of queue
    */
/*****************************************************/
```

Code Sample 4-5. Declaration of the Queue Structure

The queue array and pointers are declared *static* so that they will be known to all the routines in this source file and *only* to these routines.

The next routine, **mouse_event()**, is your interrupt handler. The **signal()** call from the previous section defined not only which event we are looking for but the routine that will be invoked when that event occurs.

mouse_event() pushes the event and the x-y coordinates onto a circular queue and then reenables the original signal.

```
/****************************************************/

mouse_event()
{
     struct m_event mouse_status;

     mouse_status.e_event = 0;
     ioctl (windo_fd, WMEVENTPOLL, &mouse_status);

     queue_head -> x = mouse_status.e_xloc;
     queue_head -> y = mouse_status.e_yloc;
     queue_head -> event = mouse_status.e_event;
     if (++queue_head >= queue + QUEUESIZE)
          queue_head = queue;

     signal (SIGMOUSE, mouse_event);
}
/****************************************************/
```

Code Sample 4-6. Routine To Save Individual Mouse Events

The key information here is that we've set `e_event` to 0 in the `m_event` structure before using the WMEVENTPOLL **ioctl()** call. In this case, the *last predefined mouse event to occur* will be returned in the `m_event` structure.

If instead we set `e_event` to a non-zero value (such as EVENT_B1_UP or a combination of events), then we are requesting information about the specified event or events. That is, the returned value in `e_event` will be 0 if the specified event or events *haven't* occurred.

Note also that by setting the `e_event` to 0 and then using WMEVENTPOLL, we clear all remembrance of past mouse events except for the most recent mouse event.

There is a very small chance that a mouse event could occur *during* our routine and not be caught; however, note that the mouse signal is re-enabled almost immediately (at the end of the routine).

4.11 Recalling Mouse Events

When you're ready for a mouse input, you can call the following routine. **mouse_inquire()** removes the oldest mouse event from the queue. The function parameters are set to the x-y location of the display pointer when the mouse signal occurred and to the event type.

The function returns "0" at the outset if the queue is empty and "1" if the queue is not empty.

```
/******************************************************/

#define TRUE  1  /* These definitions may be redundant */
#define FALSE 0

mouse_inquire (x, y, status)
int *x, *y;
unsigned long *status;
{
     if (queue_tail == queue_head)
           return (FALSE);

     *x = queue_tail -> x;
     *y = queue_tail -> y;
     *status = queue_tail -> event;

     if (++queue_tail >= queue + QUEUESIZE)
           queue_tail = queue;

     return (TRUE);
}
/******************************************************/
```

Code Sample 4-7. Routine To Return Information About Past Mouse Events

This routine allows you to handle mouse events at your convenience. The routine returns in parameters `x` and `y` the x-y coordinates of the display pointer when one of your predefined events occurred. The routine returns in parameter `status` the 32-bit quantity that represents exactly which one of those events occurred.

There's a possible "gotcha" in our queuing scheme: If the queue ever fills up--that is, if the head ever catches up to the tail--then all current information in the queue *will be lost.*

What happens is that when the head comes all the way around the queue, it will start overwriting elements, and the "old" elements will become completely unavailable. For this reason, be sure to declare your `queue` structure adequately large. (The example uses a size of 20 elements, which should be enough in most cases.)

4.12 Polling the Mouse

The preceding routines rely on an interrupt method of mouse signals. That is, once the application enables the mouse signal with the **signal()** call, the application will be interrupted every time the specified mouse event occurs.

An alternate form of mouse handling uses a *polling* technique, in which you check the current status of the mouse/display pointer at your convenience.

The following **mouse_poll** routine is used simply to get the current display pointer position.

```
/*******************************************************/

mouse_poll (x, y)
int * x, * y;

{
     struct m_stat mouse;

     ioctl (windo_fd, WMGETMOUSE, &mouse);
     *x = mouse.e_xloc;
     *y = mouse.e_yloc;
}
/*******************************************************/
```

Code Sample 4-8. Routine To Poll the Mouse

The advantage of this routine is that it tells you where the display pointer is *right now.* It doesn't depend on interrupts or mouse events to do its work.

If, for example, you wanted to track the display pointer (by drawing an onscreen line to it) while the user holds the left button down, then you could:

1. Jump to the line drawing routine when you sensed the EVENT_B1_DOWN mouse event and there enable an EVENT_B1_UP signal.
2. Use the **mouse_poll()** routine above to follow the movement of the display pointer and to do your line drawing work.
3. When the user releases the mouse button, your EVENT_B1_UP interrupt handling routine would finish the line-drawing activity and probably re-enable the EVENT_B1_DOWN mouse signal.

The disadvantage of this technique--as with any polling scheme--is that it consumes system resources heavily, even when the window is no longer active. A good idea is to use mouse polling only during short, specialized tasks within your application.

4.13 Mouse and Keyboard Interactions

Typically, you will want the user to be able to intermix mouse and keyboard inputs. For example, you may offer a menu from which the user can make a selection either by pressing the [Return] key or by pressing the left mouse button.

The crux of the following **getkey()** routine is the **read()** statement. The **read()** statement will do one of two things:

a. Return one byte of keyboard information if a keyboard key is pressed, or
b. *Fail* (that is, return a *-1*) and fall through if a mouse button is pressed.

```
/*****************************************************/

#define NOCHAR '\0'

char getkey()
{
     char ch = NOCHAR;

        /* enable the mouse interrupt routine */
     signal(SIGMOUSE, mouse_interrupt);

        /* now wait for a key press or a mouse hit */
        read(0, &ch, 1);   /* requesting one byte */

       /* if the read fails--meaning the user hit the
       ** mouse--we jump from the read()
       ** to the mouse interrupt routine.
       */

    /* disable mouse signals */
    signal(SIGMOUSE, SIG_IGN);

    /* return one byte of character info */
    return(ch);
}
/*****************************************************/
```

Code Sample 4-9. Routine To Handle Keyboard and Mouse Inputs

The **getkey()** routine depends on the existence of a **mouse_interrupt()** routine to handle the press of a mouse button.

The **getkey()** routine will return either a keyboard character or else '\0' (the ASCII NUL character), meaning that a mouse press has occurred instead of a key press.

4.14 Defining a Display Pointer

Each window has associated with it a *display pointer* that you can customize to any 16 by 16 bit-pattern.

The mouse and the display pointer can work well together in providing natural human/computer interactions for your application.

Whenever the display pointer moves in front of your window or window banner, the pointer will assume the pattern that you've set. (If you've set no shape, the display pointer resembles a small arrow.)

Typically, you define the display pointer during program initialization. Additionally, you can define different shapes according to different program states or different window locations (for example, an hour glass during a disc access, or an unhappy face after an invalid menu selection, or a double-headed arrow at a certain x-y location within your window.)

Following is the display pointer structure, as declared in <scrn/wmcom.h>:

```
/*****************************************************/

struct  sprt_stat {
   char sprite[32];          /* sprite image */
   short hot_x;              /* hot spot x */
   short hot_y;              /* hot spot y */
};
/*****************************************************/
```

Code Sample 4-10. The Structure of the Display Pointer

The 32-byte sprite[] structure is what defines the shape of the display pointer.

The hot_x, hot_y values are what determine the *single pixel* within the display pointer that will sense the pointer's location on the display. That is, any time your application examines the position of the display pointer, your application will be using the coordinates of the hot_x, hot_y spot.

The following display pointer template defines the shape of a circle:

```
/*******************************************************/

char a_circle[] =
    {
    0x03, 0xc0,     /*    ......1111......     */
    0x0e, 0x70,     /*    ....111..111....     */
    0x18, 0x18,     /*    ...11......11...     */
    0x30, 0x0c,     /*    ..11........11..     */
    0x60, 0x06,     /*    .11..........11.     */
    0x40, 0x02,     /*    .1............1.     */
    0xc0, 0x03,     /*    11............11     */
    0x80, 0x01,     /*    1..............1     */
    0x80, 0x01,     /*    1..............1     */
    0xc0, 0x03,     /*    11............11     */
    0x40, 0x02,     /*    .1............1.     */
    0x60, 0x06,     /*    .11..........11.     */
    0x30, 0x0c,     /*    ..11........11..     */
    0x18, 0x18,     /*    ...11......11...     */
    0x0e, 0x70,     /*    ....111..111....     */
    0x03, 0xc0,     /*    ......1111......     */
    };
/*******************************************************/
```

Code Sample 4-11. Structure for a Custom Display Pointer

The above pattern may not *look* like a circle, but the shape will appear differently on the computer display, where the aspect ratio of the pixels is 1:1. **Hint:** The **fedit** font editor shipped with the computer makes it easy to lay out and evaluate new display pointer patterns.

Note that the initialization values above are expressed in hexadecimal (four digits per 16-pixel row). A somewhat mindless task is hand translating your dot pattern into numeric codes.

The following **set_pointer()** routine will set the display pointer to a given pattern and will set the "hot spot" to the given *x* and *y* values location within the pattern.

```
/*****************************************************/

set_pointer (my_pointer, x, y)
char *my_pointer;
short x, y;
{
    struct sprt_stat pointer_parms;

    pointer_parms.hot_x = x;
    pointer_parms.hot_y = y;
    blt (&pointer_parms.sprite[0], my_pointer, 32);

    ioctl (windo_fd, WMSETSPRITE, &pointer_parms);
}
/*****************************************************/
```

Code Sample 4-12. Routine To Define the Display Pointer

Invoke the **set_pointer()** routine whenever you want to define a new shape for the display pointer. For example, use:

```
set_pointer(a_circle, 8, 8);
```

--to make the display pointer into a circle pattern (from the 32-byte array defined earlier) and to set the hot spot of the pointer to the center of the pattern.

Note that the numbering for the hot spot *x,y* coordinates begins at the upper left corner of the image, at location *0,0*. For example, specifying *0,15* would set the hot spot to the lower left corner of the display pointer image.

By the way, the **blt()** function in our **set_pointer** routine is part of the C runtime support library (**libc.a**) and allows fast byte-copies from source to destination character arrays.

Note that this **set_pointer()** routine does *not* change the pointer *location.* A common mistake is to use a routine like this to try to move the display pointer on the display--the effort will fail.

Instead, use the the following **move_pointer()** routine to move the display pointer anywhere on the display:

```
/*****************************************************/

move_pointer(x, y)
int x, y;
{
    struct m_stat mouse;

    ioctl (windo_fd, WMGETMOUSE, &mouse);

    mouse.s_xloc = x;
    mouse.s_yloc = y;

    ioctl (windo_fd, WMSETMOUSE, &mouse);
}
/*****************************************************/
```

Code Sample 4-13. Routine To Move the Display Pointer

Although the above routine seems more interested in the mouse than the display pointer, it actually enables you to position the display pointer whenever your window is active. It does *not* move the mouse around your desktop!

The horizontal `s_xloc` and vertical `s_yloc` values are appropriate for the current window type--alpha windows use column and row values; graphics and alpha/graphics windows use pixel values.

Note that the `x` and `y` values used in **move_pointer()** are locations relative to your window and are *not* absolute locations on the display.

For example, because the origin of an alpha window *(0,0)* is in the upper left corner, passing in *-1* as the `x` value to **move_pointer()** would move the hot spot (and display pointer) to one row just *above* the alpha window.

4.15 Polling on a Multi-Tasking System

If the user runs your application with one or more other applications at the same time, be aware of a few implications:

- The computer allows only one window to be active at a time. If that is your window, you want to take full advantage of system resources.
- If, on the other hand, your window is *deselected*, then you want to relinquish any system-intensive resources until you are once again made active.

- An example is using a mouse polling scheme any time during your application--knowing that the user is no longer working with your application, you can stop polling the mouse and not be a drain on system resources.
- You *can* jump to another routine, however, and continue on with other business, such as straight computation.

The following routines work together to detect deselection and selection of your window. A typical scenario is as follows:

- While your window is active, you want to *poll the mouse* for certain events--this is a system-intensive activity.
- As soon as the user selects another window, either with the mouse or the [Shift][Select] keystroke, you want to pause your application.
- When the user again selects your window, you want to return to your mouse polling activity.

Note: The following material is for experienced users only.

In the following code sample, the **MouseInt()** routine handles a mouse interrupt. You enable the **MouseInt()** routine in your main program as follows:

```
signal (SIGMOUSE, MouseInt);
```

The **Poll()** routine is a dummy routine that would typically include the system intensive code.

```
/*******************************************************/

#define TRUE 1
#define FALSE 0

int MouseInt()

/* global flags and data */
int WinPause = FALSE;  /* Stop polling if flag is TRUE */
int MEventMask = (EVENT_B1_DOWN | EVENT_B2_DOWN);
int button_1 = FALSE;  /* button 1 pressed if TRUE */
int button_2 = FALSE;  /* button 2 pressed if TRUE */

MouseInt() /* handle the mouse interrupt */
{
    static struct m_event mouse_event;
    static struct windio  win;
    mouse_event.e_event = 0;
    ioctl (0, WMEVENTPOLL, &mouse_event);

    /* if the window is going inactive,
    ** then set pause flag to stop polling.
    */
    if (mouse_event.e_event & EVENT_ACTIVE) {
    ioctl (0, WMGET, &win);

    /* Has window gone inactive? */
    if (!(win.w_stat & ACTIVE)){
       /* then pause on next mouse poll */
        WinPause = TRUE;
        }
    }
    else if (mouse_event.e_event & EVENT_B1_DOWN) {
    button_1 = TRUE;
    }
    else if (mouse_event.e_event & EVENT_B2_DOWN) {
    button_2 = TRUE;
    }
   signal (SIGMOUSE, MouseInt);
}
Poll() /* generic polling routine */
```

```
{
    unsigned long save_event_mask;

    if (WinPause)  {
    /* window became inactive,
    ** so pause to stop polling
    */
    /* disable  button 1 */
    save_event_mask = MEventMask;
    SetMouseEvents( MEventMask & ~EVENT_B1_DOWN );

    /* take a break! */
    pause();
    /* enable button 1 */
    SetMouseEvents (save_event_mask);
    WinPause = FALSE;
    return;
    }
    /* dummy routine--it only sleeps */
    sleep(1);
}
/* Tell the window manager what events
** we want to look for
*/
SetMouseEvents (eventmask)
unsigned long eventmask;
{
    struct m_stat mouse_stat;
    MEventMask = eventmask;
    ioctl (0, WMGETMOUSE, &mouse_stat);
    mouse_stat.s_mask = (EVENT_ACTIVE | eventmask);
    ioctl (0, WMSETMOUSE, &mouse_stat);
    signal (SIGMOUSE, MouseInt);
}
/*****************************************************/
```

Code Sample 4-14. Routines To Detect Selection and Deselection

Your application receives mouse signals *only* when it is active. That is, while the user is working with another application, his/her mouse activity will *not* trigger any mouse signals in your application.

When the user again selects your application window with the left mouse button, your application will receive that event immediately. However, you may wish to *throw away* that initial press of the left button. Chances are that the user pointed to a random spot in the window to pick your application and that you don't want to process that event.

The above routines ensure that your application won't process the activation of your window as a button event at some random spot in the window. **Explanation:** A **SetMouseEvents()** call is issued prior to the **pause()**, which disables the left-button-down event. Consequently, when the window is reactivated with the left button, the event seen in the **MouseInt()** routine is assured of being EVENT_ACTIVE and not EVENT_B1_DOWN.

4.16 Further Information

For more information on the mouse and display pointer user interface of the computer, refer to the owner's documentation for the computer.

For more information on the keyboard driver, refer to the **HP-HIL** file on the C language documentation disc.

For more information on defining the display pointer, refer to the *Window Manager Reference Manual.*

For more information on defining mouse events and polling the mouse, refer to the *Window Manager Reference Manual.*

Chapter 5

Alpha Windows

5.1 Introduction

Most applications run in *alpha windows*--that is, windows that display alphanumeric characters.

Alpha windows (or *Term0* windows) have many of the capabilities of Hewlett-Packard terminals--specifically, HP 2622 terminals. If your application is written for an HP 2622 terminal, then it should run with few or no changes in an alpha window.

If your application requires only limited terminal support--simply displaying characters and reading characters typed from the keyboard--then you needn't bother with this chapter.

If, on the other hand, you're interested in more sophisticated terminal capabilities--for example, underlining characters, positioning the cursor at x-y locations, and accepting keyboard input without echoing characters--then this chapter will be useful.

5.2 Alpha Window Features

Each alpha window on the computer offers the following features:

- A minimum of two pages of display memory (one page equals 24 lines by 80 characters). You may request a greater or lesser amount of display memory.
- Complete screen editing functions: insert/delete character, insert/delete line, and clear line/display.
- Absolute and relative cursor positioning.
- Vertical *and* horizontal scrolling.
- Tab and margin settings.

- Keyboard locking and unlocking.
- One set of eight function keys, or one set of sixteen function keys (the eight unshifted keys plus the eight *shifted* keys). You may program both the *key labels* (the characters composing the menu at the bottom of the display) as well as the *key definitions* (the string of characters that are output when the function key is pressed).
- User-selectable character sets, with each character set consisting of 256 (8-bit) characters. The default system font (**/rom/font**) is the ROMAN8 character set.
- Two command languages--HP and ANSI.
- Underlining and inverse video display enhancements. Display enhancements may be field-oriented (HP mode) or character-oriented (ANSI mode).
- Support for TERMCAP entries and the `curses` library routines.
- Support for *fast alpha* library routines.
- A standard HP-UX **tty** programmatic interface.

Note: Alpha windows do *not* offer the following HP 2622 capabilities:

- Block mode.
- Format mode (including protected and unprotected fields).
- Memory lock.
- Programmable time delay.

Also, alpha windows do *not* provide HP 2623 terminal *graphics* capabilities.

The following pages describe the workings of alpha windows as they relate to your application program.

5.3 Using Display Functions

To begin, it's useful to study the behavior of an alpha window as you press keys from the keyboard.

To create and practice on a scratchpad alpha window:

1. Type in the PAM command line simply:

   ```
   echo
   ```

--and then press [Return]. PAM creates an "empty" alpha terminal window named **echo**.

2. This window will act as your scratchpad--try typing in it, scrolling it, etc.
3. Now put the window in *display functions* mode:
   a. Access the Alpha menu for the alpha window and press Display Functions (key [f8]).
   b. Afterwards, the editing and cursor keys will display their corresponding escape sequences. The window will otherwise behave as a line-oriented terminal.

      Try pressing some of the editing keys, such as [Clear line], to examine the character codes that these special keys generate.
4. To switch out of display functions mode, press Display Functions a second time.
5. Finally, try entering some escape sequences manually from the keyboard. For example, press [ESC] and then press [H] to cause the cursor to home up.

With the alpha window in display functions mode, the window displays--rather than executes--the keystrokes that the user presses.

An important concept is that your application can recognize and *handle the character codes as they are transmitted from the keyboard* to achieve exceptional keyboard control.

Note that putting an alpha window in display functions mode can be very useful when you are debugging an application and want to see exactly what your application is writing to the window--control characters, escape sequences, and all.

Refer to the windows chapter for information on creating alpha windows programmatically.

5.4 Alpha Window Escape Sequences

Almost any alpha window function that can be performed from the keyboard (for example, pressing the [Clear display] key or the [Caps] keys) can be programmed by means of *escape sequences.*[1]

An escape sequence is a string of characters that begins with an ASCII ESC character[2] and that the alpha window interprets as a *command* rather than as a sequence of simple display characters.

For example, with an alpha window in HP mode, you can turn off the menu at the bottom of the display by outputting the following escape sequence to the alpha window:[3]

```
ESC & j @
```

Assuming the alpha window is *active* when you write this escape sequence, the effect is the same as pressing the [Menu] key to turn off the menu.

The following escape sequence turns on the *user* function keys:

```
ESC & j B
```

Assuming the alpha window is *active* when you write this escape sequence, the effect is the same as pressing the [Shift][User] keystroke.

Alpha windows recognize both HP and ANSI-defined escape sequences.

5.5 HP Escape Sequences

By default, alpha windows recognize HP escape sequences.

Some useful HP escape sequences include:[4]

- `ESC h` -- Home up.
- `ESC J` -- Clear from cursor to bottom of window buffer.
- `ESC h ESC J` -- Home up and clear to bottom of window buffer.

1. Exceptions are window manager functions that affect the entire window—such as moving a window, inverting window color, and switching on/off window borders.
2. Decimal code 27; octal code 33.
3. The escape sequences in this chapter are shown with embedded blanks for readability. When writing an escape sequence to an alpha window, omit the blanks.
4. The escape sequences in this chapter are shown with embedded blanks for readability. When writing an escape sequence to an alpha window, omit the blanks.

- `ESC K` -- Clear from cursor to end of line.
- `ESC & d D` -- Turn on underlining.
- `ESC & d B` -- Turn on inverse.
- `ESC & d @` -- Turn off underlining, inverse, or both.
- `ESC A`, `ESC B`, `ESC C`, and `ESC D` -- Cursor up, down, right, and left.
- `ESC & a <col> c <row> R` -- Position cursor to <col> (column) and <row> in the window.

You can find a complete listing of HP escape sequences in the *Term0 Reference Manual.*

5.6 ANSI Escape Sequences

To switch an alpha window to ANSI mode, use the following escape sequence:

```
ESC & k 1 \
```

(The `1` is a "one.")

Thereafter, the window will parse and accept ANSI standard escape sequences.

To switch an alpha window back to HP mode, use the following escape sequence:

```
ESC & k 0 \
```

(The `0` is a "zero.")

You can find a complete list of ANSI escape sequences in the *Term0 Reference Manual.*

5.7 Alpha Windows and File Descriptors

When a user starts your application from PAM, your application is automatically given an alpha window with three file descriptors:

- Standard input (0, or **stdin**).
- Standard output (1, or **stdout**).

- Standard error (2, or **stderr**).

The code samples in the following pages use **stdin** for keyboard reads and **stdout** for alpha window writes.

If the user starts your application from the Bourne or C shell, these file descriptors will refer to the alpha window in which the shell "lives."

To achieve better control over an alpha window I/O, as in **read()** and **write()** statements. use the file descriptor for the *window itself*. In Chapter 3, we use global variable `w_fd` to hold the window file descriptor that is returned from the **open()** call. However, using the window file descriptor instead of **stdin**, **stdout**, and **stderr** is necessary only if you want to disallow the shell's ability to redirect input and output to and from your application.

5.8 Raw Mode--A Useful Way To Be

The default setting of an alpha window is *cooked mode*, which has the following characteristics:

- The editing keys act *locally*. For example, when the user presses the down-arrow key, the cursor will move one row at a time down the alpha window. An application waiting for keyboard input will have no knowledge of--and no control over--the cursor. In cooked mode, the editing keys *don't transmit.*
- The typing keys *echo* on the display. For example, when the user presses the unshifted [A] key, the alpha window will display the character **a**.

In *raw mode*, your application has much greater control over keyboard input. For example, your application might define the [Tab] key, right-arrow key, and space bar to move the cursor, say, to the next selection in a menu, while "locking out" other keyboard responses.

Note that the following *system keystrokes* are not affected by raw mode--they perform their normal system functions:

- [Shift][Reset]--reboots the operating system.
- [Shift][Stop]--forcibly stops the application.
- [Menu]--toggles the function key labels off and on.
- [User/System]--switches between the *user* (application program) key labels and the *system* key labels.

- [Shift][Select]--selects the active application window.

The following header files contain the data type declarations to set an alpha window to raw mode and to perform terminal i/o:

```
/*******************************************************/

#include <stdio.h>
#include <signal.h>
#include <termio.h>
/*******************************************************/
```

Code Sample 5-1. Header Files for Terminal Control

5.8.1 Setting an Alpha Window to Raw Mode

The following **setraw()** routine sets the alpha window to raw, no-echo mode:

```
/*******************************************************/

#include <termio.h>

int fixup();  /* routine defined in a couple of pages */

static struct termio savetty; /* to restore later on */

setraw ()
{
  struct termio t;

  /* save the tty parameters */
  ioctl (0, TCGETA, &savetty);

  /* get the tty parameters to modify */
  ioctl (0, TCGETA, &t);

  t.c_cc[VMIN] = 1; /* read at least 1 character */
  t.c_cc[VTIME] = 0; /* don't time out */
  t.c_lflag &= ~(ICANON | XCASE | ECHO);
  t.c_lflag |= ISIG; /* check for ^C and [Stop] */
  ioctl (0, TCSETAW, &t);   /* set the new parameters */
  write (1, "\033&s1A", 5); /* set transmit strap  */

  /* The following are optional settings:
  **                      --------
  */
  write (1, "\033*dR", 4);  /* turn the cursor off */
  signal (SIGINT, fixup);   /* enable ^C */
  signal (SIGQUIT, SIG_IGN); /* disable the [Stop] key */
}
/*******************************************************/
```

Code Sample 5-2. Routine To Set the Alpha Window to Raw Mode

The `termio` structure above, defined in `<termio.h>`, is common across all HP-UX terminal devices.

Right at the outset, the `termio savetty` structure is filled out with the initial values of the alpha window; we'll restore these values in a later **restoretty()** routine.

The **setraw()** routine uses another **ioctl()** call to get the current `termio` values. The routine masks off *canonical* processing and keyboard echo, and then uses a final **ioctl()** to set the `termio` structure to the new values.

Notice the two **signal()** calls appearing at the end of the **setraw()** routine. The first, with SIGINT, sets up an interrupt--if the user presses the [CTRL][C] keystroke, execution will jump to the **fixup()** routine (see below).

The second **signal()**, with SIGQUIT, uses a "dummy" function SIG_IGN to disable the [Stop] key on the keyboard so that it will do nothing if the user presses it. You could instead define SIGQUIT to jump to your own interrupt handler.

5.8.2 Turning the Cursor Off and On

Each alpha window has its own cursor, to mark where the next display character will appear in the alpha window. You may want to turn the cursor off to avoid distracting the user, especially if your application does a lot of cursor positioning.

To turn off the cursor as in the **setraw()** above, write out the following escape sequence:[5]

```
ESC * d R
```

To turn the cursor back on, use the following escape sequence:

```
ESC * d Q
```

As an added benefit, displaying rows of characters with the cursor off is slightly *faster* than with the cursor on.

5.8.3 Handling ^C

The SIGINT **signal()** appearing in the **setraw()** routine above will enable an interrupt routine named **fixup()**.

This **fixup()** routine will be called if the user presses [CTRL][C]:

5. The escape sequences in this chapter are shown with embedded blanks for readability. When writing an escape sequence to an alpha window, omit the blanks.

```
/*****************************************************/

fixup()
{
     write(1, "\033h\033J", 4); /* home and clear */
     restoretty();
     exit(0);
}
/*****************************************************/
```

Code Sample 5-3. Routine Called If a [CTRL][C] Occurs

The purpose of **fixup()** is to "clean up" the alpha window and to cause the application to terminate normally.

The **restoretty()** routine appears later in this chapter.

5.8.4 A Word About the Transmit Strap

Each alpha window has a software "switch" called the *transmit strap*, that determines whether escape sequences are *transmitted* to the application or are handled *locally* by the alpha window (unknown to the application).

The transmit strap is initially cleared so that editing keystrokes and other escape sequences will control the cursor within the alpha window and won't be passed to your application.

One of the key purposes of the **setraw()** routine above is to enable your application to receive *all* editing keystrokes--hence, that routine includes a **write()** statement to set the transmit strap. The following escape sequence does the trick:

```
ESC & s 1 A
```

(The 1 is a "one.")

The next escape sequence restores normal (or local) handling of escape sequences:

```
ESC & s 0 A
```

(The 0 is a "zero.")

You'll send this second escape sequence to the alpha window when you want to restore the alpha window to cooked mode.

5.8.5 Running in Raw Mode

With your alpha window in raw mode, certain terminal functions are no longer handled for you automatically. Your application must handle them on its own.

This section mentions some of the more important responsibilities left to your application.

5.8.5.1 Echoing Characters

The most common need is to echo characters as they're typed. For example:

```
  write(1, &ch, 1);
```

--will write one character to standard output.

(The `1` is a "one.")

Without an explicit **write()** statement, the user won't know what he or she is typing.

5.8.5.2 Destructive Back Space

In raw mode, the [Back space] key simply moves the cursor to the left one space. To create a *destructive* back space:

1. Move the cursor one space to the left by sending an ASCII BS character.[6]
2. Write a space (to "white out" the next character).[7]
3. Again move the cursor one space to the left.

The net effect is a back space with erasure. But that's not all!

In cooked mode, your application will never *see* the user's input line until the user presses [Return]. Any erasures will be handled for you and automatically eliminated from the key buffer so that your application will receive only the corrected line.

In raw mode, however, your application will receive characters as they are typed. If the user presses [Back space], your application will detect an ASCII BS character just as it would detect a printable character. In the event of a back space, it's up to your keyboard handling routines to "erase" (or remove) the

6. Decimal code 8; octal code 10.
7. Decimal code 32; octal code 40.

undesired character from your own input buffer.

5.8.5.3 New Lines and Carriage Returns

In cooked mode, a new line character (ASCII LF)[8] written to the alpha window causes a carriage return/linefeed sequence.

In raw mode, you must explicitly write the two characters to the window:

```
write(1, "\015\012", 2);
```

The effect is to position the cursor at the left margin of the next line.

5.8.5.4 Escape Sequence Parsing

Most HP escape sequences generated from the keyboard consists of *two* characters--the ESC character and one other. For example, the up-arrow key generates:

```
ESC A
```

That is, pressing the up-arrow key causes the keyboard driver to send an **ESC A** to the alpha window driver, which normally interprets the string as an instruction to move the cursor up one row.

However, when your application intercepts keystrokes in raw mode, you can set up customized keyboard routines to handle cursor movement as you please.

Recognizing escape sequences as they are transmitted from the keyboard can involve a lot of code. The simplest way is to set up a **read()** sequence of this form:

8. Decimal code 10; octal code 12.

```
/*******************************************************/

#define ESC '\033'

char parsekey()
{
     char ch;
     read(0, &ch, 1);
     if (ch == ESC)
          {
          read(0, &ch, 1);
          /*
          ** process the remainder of the
          ** escape sequence...
          */
          }
     else return(ch);
}
/*******************************************************/
```

Code Sample 5-4. Routine to Trap Escape Sequences

The **parsekey()** routine can include a large **switch()** statement after it has detected an ESC to pick out the individual keystrokes that the user might have pressed.

In HP mode, the *multiply shifted* arrow keys generate multi-character escape sequences as follows:

Keystroke	HP ESC Sequence
[Shift] up-arrow	ESC S
[Shift] down-arrow	ESC T
[Shift] right-arrow	ESC & r 1 R
[Shift] left-arrow	ESC & r 1 L
[Shift] up-and-right arrows	ESC & r 1 u 1 R
[Shift] up-and-left arrows	ESC & r 1 u 1 L
[Shift] down-and-right arrows	ESC & r 1 d 1 R
[Shift] down-and-left arrows	ESC & r 1 d 1 L

In HP mode, the simplest way to trap these keystrokes in your **parsekey()** routine is to:

1. Look for an ampersand (&) as the second character of the escape sequence routine.
2. Keep reading into your input buffer until you detect an *uppercase letter* (ASCII 'A' through 'Z'). At this point, you know that the escape sequence from the keyboard is complete.
3. Either ignore the keystroke or parse the input buffer and take appropriate action.

For more information, refer to the *Term0 Reference Manual.*

5.8.6 Blocked vs. Non-Blocked Reads

There are two common ways to accept terminal input--using *blocked* reads and *unblocked* reads.

The **setraw()** routine above initialized the alpha window to accept blocked reads. That is, when your program reads from the keyboard, it will *block*, or wait, until a key is pressed. (The **read()** won't return until the user presses .)

You may want to use *unblocked* reads instead. In an unblocked read, the **read()** statement returns immediately, whether or not there's a key in the key buffer.

The advantage of an unblocked read is that your application doesn't have to be hung up waiting for the user--it can go off and do other things.

It's easy to set up the alpha window for unblocked reads: Simply define both the `VMIN` and the `VTIME` values in your `termio` structure to be *0* (zero) during the **setraw()** routine.

The main disadvantage is that *polling the keyboard*, as happens during unblocked reads, can be a drain on system resources.

If a number of applications are all polling the keyboard at the same time, then system response time can become degraded.

Perhaps the best solution is to set up a window manager **signal()** that will inform you when your application goes inactive (that is, your user picked a different application window).

You can start off by polling the keyboard when your window is active, but when your application senses that it's going inactive, you can return to blocked reads or simply stop reading the keyboard altogether.

Refer to Chapter 4, *Keyboard, Mouse, and Display Pointer*, for more information.

5.8.7 Returning to Cooked Mode

Cooked mode is the opposite of raw mode; cooked mode allows normal terminal handling of keystroke and cursor movement. For example, when the user presses the up-arrow key, the cursor will move up a row.

When your application exits, or whenever you want to restore the alpha window to its initial state, you can use the following **restoretty()** routine:

```
/*******************************************************/

restoretty()
{
    ioctl (0, TCSETA, &savetty);
    write (1, "\033&s0A", 5);  /* clear transmit strap */

    /* turn the cursor on (if necessary) */
    write (1, "\033*dQ", 4);
}
/*******************************************************/
```

Code Sample 5-5. Routine for Restoring Cooked Mode

Note that this routine depends on the `savetty` structure, whose values were set before the alpha window was thrown into raw mode in the first place by the **setraw()** routine.

5.9 Defining Function Keys

The computer supports either eight or sixteen function keys per alpha window. An HP escape sequence for a function key typically specifies three items:

- The function key you wish to define (1-16).
- The key label to appear in the menu at the bottom of the display (1 to 16 characters).
- The string of characters that will be output when the user presses the function key (up to 80 characters).

Refer to the *Term0 Reference Manual* for the complete function key syntax.

The rule is that if you specify a key greater than eight (8), then the alpha window will provide two rows of function key labels in the display menu--the lower row for the *unshifted* function keys (1-8) and the upper row for the *shifted* function keys (9-16).

If only function keys 1-8 are defined, then pressing either the unshifted or the shifted key will result in the same mapping.

The definition for a particular function key will take effect as soon as the escape sequence is written.

The menu for your application's function keys will be displayed:

- When your application is in the currently active window, *and*
- The menu is turned on, *and*
- The current menu is the *user* menu (as opposed to the *system* menu or the *alpha* menu).

Note that if your application turns *off* the key labels, then the menu will remain invisible until your own application, some other application, or the user turns it back on.

To ensure that your application menu is showing, use the following HP escape sequence:

```
ESC & j B
```

5.10 C Programming Notes

You may intermix **write()** and **printf()** statements to standard output. Use whichever is more convenient, with the following restriction:

To ensure that escape sequences are sent as single units to the alpha window, you may want to write out escape sequences using **write()** statements rather than **printf()** statements. For example, you can use:

```
write(1, "\033&f1k16d5L-Howdy!-********Hello", 31);
```

--to guarantee the integrity of the function key escape sequence. The reason is that the **printf()** statement relies on *buffered* output. There is a reasonable chance that **printf()** will not ship all the characters at once to the alpha window.

Alternately, you can use:

```
fflush(stdout);
```

--to "flush out" the standard output buffer that is filled by a **printf()** statement.

If code size is an issue, use **write()** statements exclusively. The reason is that the **printf()** statement brings with it a rather large set of C runtime library routines at link time. One **printf()** statement can add 12K bytes or more to your program.

5.11 Disabling the [Enter] Key

By default, the [Enter] key reads the current line of the alpha window and sends all the characters in that line to the application. You may choose to disable the [Enter] key so that pressing the [Enter] key will have no effect on your application.

To disable the [Enter] key, use the following escape sequence (with no embedded blanks):

```
  ESC & f -1 k -1 L
```

(The *-1* is a "minus one.")

The result of writing this escape sequence is that when the user presses the [Enter] key, exactly nothing will happen.

5.12 Alternate Character Sets

Alpha windows support any number of character sets (or fonts), depending on available memory.

At power on, the computer loads one font into system memory. Your application can itself load one or more additional fonts from disc into system memory.

Afterwards, you can write characters to the alpha window in a variety of fonts by using HP escape sequences to select among the fonts in system memory.

Refer to the Chapter 8, *User-Definable Fonts*, for more information.

5.13 TERMCAP Entries

Yet another way to control the alpha window is through standard TERMCAP entries.

For a listing of the TERMCAP entries recognized by the alpha window, view the environment file built into Read-Only Memory (**/rom/.environ**).

Operating system support of the TERMCAP environment variable and the *curses* library (**libcurses.a**) enables a variety of HP-UX applications to port quickly to alpha windows.

For more information, refer to the *CURSES(3X)* pages in the *HP-UX Reference Manual.*

5.14 Introduction to Fast Alpha

These pages describe the routines that are available to you as part of the *fast alpha library*, included on the C Language Translator disc (**/c/lib/libfa.a**).

Note that this discussion applies neither to graphics windows nor to alpha/graphics windows, but *only* to alpha windows.

To review, there are four ways to control the contents of alpha windows:

- Using HP escape sequences.
- Using ANSI escape sequences.
- Using the routines in the curses library (**libcurses.a**).
- Using the routines in the fast alpha library.

The fast alpha library enables you to place and move characters in alpha windows at high speeds.

5.14.1 Advantages of Fast Alpha

Here are the advantages of using fast alpha routines:

High performance. Fast alpha routines typically execute three to five times faster than normal alpha window routines. The equivalent display rate is approximately 9600 baud (fast alpha) compared to approximately 2400 baud (normal alpha window).

Flexibility. The fast alpha calls can be used in a variety of combinations to set up fonts, control the cursor, write strings, and "blast" alpha blocks to the window. In addition, you can intermix fast alpha writes, **fawrite()** and **farectwrite()**, with standard C **write()** and **printf()** statements.[9]

9. The implementation of Windows/9000 on HP-UX Series 300 computers does not allow you to mix fast alpha calls with normal writes to an alpha window.

5.14.2 Disadvantages of Fast Alpha

There are a number of reasons why you might not want to use fast alpha:

- The fast alpha calls work at a fairly low level. For example, in order to write a simple character string (with **fawrite()**), you must specify six parameters, including the row and column coordinates.
- None of the fast alpha write routines knows about *control characters* or *escape sequences.* You need to handle special characters (for example, tabs and carriage returns) on your own. Also, you need to handle display enhancements (underline and inverse) on your own--you must add *attributes* to individual characters as you write them to the window.
- The fast alpha routines are not generally known to other UNIX[10] machines. Some HP-UX systems support the fast alpha library, but you may need to rewrite portions of your code in order to port it to other machines.

Our recommendation is to use the fast alpha library only when display performance is an issue. One strategy is to:

1. Begin by using **printf()** or **write()** statements for display out.
2. Add conditional compilation flags (`ifdef#`'s) around fast alpha calls when you add the calls in performance-critical areas of your code.

5.15 The Fast Alpha Routines

Following is a brief summary of the fast alpha routines that are available to you, as part of the **libfa.a** library:

Routine	Description
fainit()	Set up an existing alpha window for fast alpha calls.
fagetinfo()	Get information about the alpha window.

10. UNIX is a trademark of AT&T.

Routine	Description
fasetinfo()	Set information about the alpha window.
fawrite()	Write a group of characters and enhancements to the fast alpha window.
farectwrite()	Fill a block of the window with optional display enhancements.
faroll()	Roll a block of an alpha window up, down, left or right.
facursor()	Control the displayed cursor.
facolors()	Change foreground and background colors.
fafontload()	Load the specified font for the alpha window.
fafontactivate()	Change the current font for the current window.
fafontremove()	Remove the specified font from system memory.
faterminate()	Terminate the fast alpha routines.

Note that the term *block* means a rectangular area within the window that is specified by four coordinates--the *x,y* coordinates of the upper left corner of the block (relative to the upper left corner of the logical screen) and the *x,y* coordinates of the lower right corner of the block.

The following pages provide examples of the more commonly used fast alpha calls. Not all the calls will be described. For more information, refer to the libraries section of C language documentation disc.

5.16 Accessing the Fast Alpha Library

To use the fast alpha library, include the **<fa.h.>** header file in your source:

```
#include <fa.h>
```

Your command line to link in the fast alpha library should be as follows:

```
cc [flags] file ... -lfa
```

5.17 Controlling an Alpha Window Using Fast Alpha Calls

The following pages include an example to show how to perform some common alpha window operations using fast alpha routines.

A typical sequence of using fast alpha is to:

1. Open a regular alpha window.
2. Enable the alpha window to use the fast alpha routines.
3. Download a character font into system memory and activate the newly loaded character font (Chapter 8).
4. Write some characters and enhancements to the alpha window.
5. Remove the character font from system memory.
6. Terminate the fast alpha routines.
7. Close the alpha window.

5.17.1 Initializing a Fast Alpha Window

The following **alpha_open()** routine initializes the fast alpha driver for standard output, normally a 24 by 80 column alpha window, and returns the file descriptor in `alpha_fd`:

```
/*******************************************************/

#include <fa.h> /* structures in fast alpha calls */

int alpha_fd;  /* file descriptor for the alpha window */

alpha_open()
{
     char *getenv();

     if ((alpha_fd = fainit(1,getenv("TERM")) < 0)
          {
          printf("Can't do it!\n");
          exit(1);
          }
}
/*******************************************************/
```

Code Sample 5-6. Routine To Open an Alpha Window for Fast Alpha Calls

Note the following:

- You should use the **fainit()** call only *once*--sometime during program initialization after you've already created the alpha window.
- The fast alpha driver will inherit the current settings of the window--for example, the size of the logical window (default is 80 columns by 24 lines).
- The **fainit()** call immediately throws the alpha window into *ANSI* mode. To restore the alpha window for HP escape sequence processing, write the following string to the window:

  ```
  ESC & k 0 \
  ```

5.17.2 Downloading a Font

Chapter 8 of this guide, *User-Definable Fonts*, covers custom-made fonts, or character sets, that you can load from a disc and select for an alpha window.

You *don't* need to be using fast alpha writes in order to use a customized font. It just so happens that the easiest way to download a font is to use a prewritten font routine and that this font routine happens to exist in the fast alpha library.

After you've downloaded a font, you can select and deselect that font using HP alpha escape sequences.

We assume in this chapter that you're satisfied with the default ROMAN8 system font (in **/rom/font**), having a 6 by 8 cell size.

5.17.3 Writing Characters With Fast Alpha Writes

The following **write_there()** routine writes a string at a given location in the alpha window:

```
/*******************************************************/

write_there(str, x, y)
char *str;
int x, y;
{
     fawrite(alpha_fd, x, y, str, NULL, strlen(str));
}
/*******************************************************/
```

Code Sample 5-7. Routine To Write a Fast Alpha String

Note the following:

- Display coordinates (*x*, *y*) are measured in *character* and *row* positions, relative to the alpha window origin (*0*, *0* at the upper left corner).
- The **fawrite()** has no effect on the position of the cursor in the window. To move the cursor, you can either use **facursor()** fast alpha calls or else write cursor positioning escape sequences to the window.
- The NULL value in the **fawrite()** parameter list indicates that *no* display enhancements (such as underlining or inverse) are to be included with the string.

To include display enhancements in a fast alpha string, you must build up two parallel strings, one consisting of the characters themselves and one consisting of the display attributes of each character.

Legal enhancements are defined in `<fa.h>` and include:

- FAINVERSE
- FAUNDERLINE

For example, to write an underlined 'a' to the display, you would put the character 'a' in the first string parameter to **fawrite()** and then put FAUNDERLINE in the corresponding byte of the second string parameter.

5.17.4 Other Fast Alpha Capabilities

There are fast alpha routines that enable you to position the cursor, to scroll text within a fast alpha window, and to write a large block of character information at one time.

For more information, refer to the libraries section of C language documentation disc.

5.17.5 Closing a Fast Alpha Window

When done with fast alpha capabilities, use the following **alpha_close()** routine, perhaps as part of your application's exit procedure:

```
/*******************************************************/

#define   ON   1

alpha_close()
{
     /* make sure cursor is on */
     facursor(alpha_fd, -1, -1, ON);
     faterminate(alpha_fd);
}
/*******************************************************/
```

Code Sample 5-8. Routine To Close a Fast Alpha Window

Note that the **faterminate()** call has no effect on the normal terminal capabilities of the alpha window.

5.18 Further Information

To learn more about alpha window escape sequences, refer to the *Term0 Reference Manual.*

To learn more about alpha windows as terminal devices, refer to the **tty** file on the C language documentation disc.

To learn more about downloading a different character font into an alpha window, refer to Chapter 8, *User-Definable Fonts*, in this guide.

To learn more about fast alpha library calls, refer to the **fa*** files on the C language documentation disc.

Chapter 6
Graphics Windows

6.1 Concepts

Graphics windows support both vector and raster graphics. They can act as though they are plotter beds for you to draw lines in or as bit-maps that can read blocks and write blocks of data.

Graphics windows are best used for non-alpha portions of applications. A typical setup is to use *two* windows for a graphics-intensive application--one to do the plotting (your graphics window), and one to display text and to handle keyboard input (an alpha window).

If you need to use both alpha and graphics intensively in one window, refer to the next chapter on the third window type, *alpha/graphics windows*.

6.1.1 Low-Level Graphics Routines and HP-GL

You can control the *contents* of graphics windows in two ways:

- By means of low-level graphics routines that make **ioctl()** calls to the *Plotter Emulator Window Driver.* The PE driver resides in ROM and translates the data you pass to it into instructions for the display driver.
- By means of the *Hewlett-Packard Graphics Language* (HP-GL). HP-GL is a plotter command language consisting of strings of ASCII characters. You send HP-GL commands to an *HP-GL interpreter*, which translates the strings into calls to the PE driver. For example, the two-character string "PG" (for *page*) causes the HP-GL interpreter to call the PE driver with instructions to clear the current contents of the graphics window.

6.1.2 Relative Comparison

Here are the advantages of writing software that outputs HP-GL strings:

- HP-GL is the language of Hewlett-Packard plotters. It is a well-defined and highly portable language. You can use the same routines to generate HP-GL commands for both graphics windows and external plotters.

- HP-GL offers a variety of high-level capabilities--including seven line types, variable character sizes and slants, scaling, and tick marks--to save you time in writing your own graphics routines.
- The HP-GL language provides a standard ROMAN8 character set, as well as the capabilities to change the *size* and the *slant* of characters. Without HP-GL, the plotter emulator window does *not* provide a character set--to draw characters in a graphics window requires that you build your own character set.
- HP-GL consists of simple strings of ASCII characters so that HP-GL files are easy to transfer from device to device. Conversely, the low-level graphics routines calls are highly machine-specific. There is *no* assurance that a low-level graphics routine that you write for this computer will work on any other computer!

Here are the advantages of using low-level graphics routines:

- The low-level graphics routines allow *raster operations* in a graphics window, such as block fill and block copy. HP-GL supports only *vector operations* (line drawing).
- Because they are more direct, low-level graphics routines execute faster. They do not rely on any intermediate parsing by the HP-GL interpreter--an extra layer of software.

The actual display facilities used by both are the same--ROM-based, display driver routines directly to the Graphics Processor Unit (GPU).

6.2 Low-Level Graphics Requests

Following is a brief summary of the low-level graphics requests that are available to you, as defined in **<scrn/plotem.h>**:

Request	Description
PEBLKCP	Copy a block of data from one position to another.
PEBLKFILL	Fill a bit-map block of data with a given pattern.
PEBLKRD	Read a bit-map block of data.

Request	Description
PEBLKWR	Write a bit-map block of data.
PECURSOR	Turn plotter cursor on or off.
PELINETYPE	Set the line type used in plots.
PEPLOTABS	Plot or move pen absolute (to coordinates).
PEPRINTPLOT	Print full-size raster images.
PESCROLL	Define relation of plotter window to plotter bed.
PESTRAPA	Put the window into raw/cooked mode.
PEWRRR	Set the replacement rule (OR, EXOR, etc.)

These requests are made as part of **ioctl()** calls to the Plotter Emulator Window Driver. For example, suppose that you've opened a graphics window with file descriptor `graph_fd` and that you've declared a `plot_info` structure as included in **<scrn/plotem.h>**:

```
struct   plot_info {
   int   x;       /* x coord of point to plot to      */
   int   y;       /* y coord of point to plot to      */
   char   pen_down;   /* TRUE if pen should be down   */
};
```

Then you can use the following call to draw a line in the graphics window:

```
ioctl (graph_fd, PEPLOTABS, &my_plot_info);
```

The **plotem** file (for "plotter emulator") on the C language documentation disc describes each of these low-level plotter emulator window requests in detail.

6.3 Controlling a Graphics Window With Low-Level Routines

The following pages contain four control routines for graphics windows:

open_graph()	For creating and initializing a graphics window.
raw_graph()	For putting a graphics window in *raw mode* to handle keyboard input.
draw_vector()	For drawing individual vectors in a graphics window.
fill_raster()	For filling a block of data in a graphics window.
write_raster()	For writing a block of data to a graphics window.
read_raster()	For reading a block of data from a graphics window.
close_graph()	For severing the connection between application and graphics window.

Note

The following routines are specific to the Integral Personal Computer operating system and will not port to other HP-UX systems.

Be sure that you isolate these system-dependent routines in a separate module of your software.

6.3.1 The Header Files and Global Variable

Include the following header files and global variable in order to use the low-level graphics routines:

```
/*******************************************************/

#include <termio.h>      /* for putting a graphics window
                         ** in raw mode
                         */
#include <scrn/disp.h>         /* for raster operations */

#include <scrn/wmcom.h>     /* for the window structure */

#include <scrn/plotem.h> /* structures for ioct() calls */

#include <fcntl.h>   /* for opening the graphics window */

#include <scrn/keycode.h>      /* for keycode mnemonics */

int graph_fd;    /* file descriptor for graphics window */

/*******************************************************/
```

Code Sample 6-1. Header Files for Low-Level Graphics Routines

Interestingly, not only does a graphics window behave as an HP-UX *device*, with **open()**, **ioctl()**, **write()**, and **close()** entry points, but a graphics window behaves similarly to an HP-UX terminal (tty) device. That is, you can set a graphics window to *raw mode* and exercise a great deal of control over keyboard inputs, just as you can with an alpha window.

Similarly, you can precisely control the location and characteristics of a graphics window, just as you can with an alpha window. The window manager routines discussed in the *Windows* chapter apply to both window types.

6.3.2 Opening a Graphics Window

The following **open_graph()** routine creates a graphics window of *xsize*, *ysize* pixels at location *xloc*, *yloc* on the display (also measured in pixels).

```
/*****************************************************/

open_graph (xsize, ysize, xloc, yloc)
int xsize, ysize;
int xloc, yloc;
{
     struct windio plot;   /*  window structure for the
                              **  graphics window
                     */
     /* get the default window attributes */
     ioctl (0,WMGET,&plot);

     /* assign window as Plotter type */
     plot.w_type = PETYPE;

     /* set window to autoshow and autodestroy */
     plot.w_stat |= AUTO_SH | AUTO_DEST;

     /* set size in pixels of the window */
     plot.w_width = xsize;
     plot.w_height = ysize;

     /* set the location on screen */
     plot.w_xloc = xloc;
     plot.w_yloc = yloc;

     /* use default buffer size */
     plot.w_gen1 = 0;
     plot.w_gen2 = 0;

     /* optional -- append suffix to window name */
     strcat (plot.w_path, ".plot");

     /* actually create the window */
     ioctl (0, WMCREATE, &plot);

     /* get file descriptor */
     graph_fd = open (plot.w_path, O_RDWR);
}
/*****************************************************/
```

Code Sample 6-2. Routine To Create a Graphics Window

Note that the **open()** call returns in `graph_fd` a *file descriptor* for use in subsequent reads and writes to the plotter window.

To make your code more "bullet-proof," you should test the returned values of the WMCREATE **ioctl()** and the **open()** calls. For example:

```
if ((graph_fd = open (plot.w_path, O_RDWR)) == -1)
    {
    printf("Can't open graphics window.\n");
    my_error_routine (A_NEW_GRAPH_ERROR);
    }
```

If either the **ioctl()** or the **open()** call returns a failure code (*-1*), you can take appropriate action.

Fields `w_gen1` (height) and `w_gen2` (width) of the `windio` structure set the size of the "plotter bed" of the graphics window, measured in pixels. If both parameters are 0 ("zero"), as in the above code sample, then the plotter bed will default to 512 pixels wide by 255 dots high. To avoid unpredictable results, we recommend that you use this size unless there's good reason to request a larger or smaller plotter bed.

6.3.3 Putting a Graphics Window in Raw Mode

As with an alpha window (Chapter 5), you may elect to put a graphics window in *raw mode* in order to intercept keystrokes (such as up-arrow and [Tab]) before they are sent on to the graphics "terminal" for processing.

The following **raw_graph()** routine does just that:

```
/********************************************************/

#define TRUE  1
#define FALSE 0

static struct termio plotio_2;
/* to save the original state of the graphics window */

raw_graph()
{
     struct termio plotio;

     /* Get current terminal attributes */
     ioctl(graph_fd, TCGETA, &plotio);

     /* Save attributes for later restoration */
     ioctl(graph_fd, TCGETA, &plotio_2);

     /* Turn off canonical processing */
     plotio.c_lflag &= ~ICANON;

     /* Get characters when one is ready */
     plotio.c_cc[VMIN] = 1;
     plotio.c_cc[VTIME] = 0;

     /* Ignore SIGQUIT -- the [Stop] key.
     ** (If ISIG is not cleared, then the
     ** the up-arrow key will send a two
     ** character sequence of \233 \003
     ** and \003 is SIGQUIT!)
     */
     plotio.c_lflag &= ~ISIG;

     /* Turn off echo
     ** (if you leave echo on,
     ** then the HP-GL interpreter will
     ** regard keyboard inputs as HP-GL
     ** commands--like it or not!
     */
     plotio.c_lflag &= ~ECHO;
```

```
        /* Now set the plotter window */
        ioctl(graph_fd, TCSETA, &plotio);

        /* Enable transmission of special keys */
        ioctl(graph_fd, PESTRAPA, TRUE);
    }
    /***************************************************/
```

Code Sample 6-3. Routine To Put a Graphics Window In Raw Mode

A corresponding **cook_graph()** routine is much easier--it involves only two lines of code:

```
    cook_graph() /* will cause keys to be buffered until our
              ** user hits carriage return.
              */
    {
        ioctl(graph_fd, TCSETA, &plotio_2);
        ioctl(graph_fd, PESTRAPA, FALSE);
    }
```

The reason for the simplicity is that the **raw_graph()** routine filled up the `plotio_2` structure with the original values of the plotter terminal. Thus, resetting the plotter terminal values involves a single **ioctl()** call using this `plotio_2` structure.

You can choose to ignore cooked mode altogether if your application runs the whole time in raw mode.

6.3.4 Reading Keys in Raw Mode

With the graphics window in raw mode, your application receives keystrokes as soon as the user presses them--no carriage return is necessary.

Letter, number, symbol, and other typing keys behave as they do for alpha windows. For example, if the user presses the shifted [A] key, your graphics window application will receive an ASCII 'A'.[1]

However, if the user presses an *editing* key, your graphics window application will receive a completely different set of character codes than will an alpha window application.

1. Decimal code 65; octal code 101; hexadecimal code 41.

For example, pressing the up-arrow key generates two character codes--ESC A, or (in hexadecimal):

```
   0x001B 0x041
```

--when sent to an alpha window. The same keystroke generates the following two keycodes when sent to a graphics window:

```
   0x09B 0x003
```

One difference is that the first character code in the graphics sequence (0x09B) is a *biased* ESC--an escape character with the eighth bit set.

The other difference is that the second character code in the graphics sequence is derived from a hardware mapping of the keyboard. On the other hand, the second character code in the alpha sequence has been translated by the alpha window driver into the corresponding HP 2622 terminal sequence (0x041 for uppercase "A").

You can find a complete set of mnemonics for the hardware keyboard codes in **<scrn/keycode.h>**. For example, **CURS_UP_KEY** is defined in the header file as 0x103. The way your program will see that key code is 0x09B 0x003--the second character of a two-character string.

To translate a graphics window special key into its keyboard definition, logically OR the second character code with a **CHAR_KEY** quantity, defined in **<scrn/keycode.h>** as 0x100.

The following **get_graph_key()** routine reads keystrokes from a graphics window in raw mode and returns their codes as found in **<scrn/keycode.h>**:

```
/******************************************************/

#define BIASED_ESC '\233'
get_graph_key()
{
     char ch;

     read(graph_fd, &ch, 1);
     if (ch == BIASED_ESC)
         {
         read(graph_fd, &ch, 1);
         ch |= CHAR_KEY;     /* build the keycode */
         }
     return(ch);
}
/******************************************************/
```

Code Sample 6-4. Routine To Return Graphics Window Keystrokes

Afterwards, you can use a large **switch()** statement to sort out the individual keystrokes.

6.3.5 Drawing Lines In a Graphics Window

The following **draw_vector()** routine will draw a line to a given *xloc*, *yloc* location in a graphics window. If the plotter pen is in the raised position (pen == UP), then the pen will be moved to that location and no line will be drawn.

```
/*********************************************************/

#define DOWN  '\0'
#define UP    '\1'

draw_vector (xloc, yloc, pen)

int     xloc, yloc;
char    pen;
{
        struct plot_info plinfo;

        plinfo.x = xloc;
        plinfo.y = yloc;
        plinfo.pen_down = pen;
        ioctl(graph_fd, PEPLOTABS,&plinfo);
}
/*********************************************************/
```

Code Sample 6-5. Routine To Draw a Line in a Graphics Window

6.3.6 Using Rasters in a Graphics Window

Both the graphics window and the alpha/graphics window types support *raster operations*--that is, reads and writes to the window in discrete blocks of pixels.

The following code samples show:

- How to fill raster blocks in a graphics window.
- How to write raster blocks to a graphics window.
- How to read raster blocks from the window.

A final code sample is a **RasterBlaster** program to show how the individual routines work together.

Note

The following routines are specific to the Integral Personal Computer operating system and will not port to other HP-UX systems.

Be sure that you isolate these system-dependent routines in a separate module of your software.

In the following pages, `graph_fd` is a valid file descriptor to an **open()**-ed graphics window.

6.3.6.1 Filling a Raster Block

The following **fill_raster()** routine fills a rectangular region of a graphics window with a pattern. Parameter pair (`x,y`) is the upper left corner of the fill rectangle--with the origin at the *lower left corner* of the window.

Parameters `height` and `width` specify the dimensions of the fill rectangle in pixels.

```
/*******************************************************/

fill_raster (x, y, height, width)
int x, y, height, width;
{
    struct blkfill blkfl;
    register int i;

    blkfl.x = x;
    blkfl.y = y;
    blkfl.height = height;
    blkfl.width = width;
    for (i=0; i<32; i++)
       blkfl.fill[i] = (unsigned char)0xf0;
       /* mindless pattern */
    blkfl.rr = NEW;

    ioctl(graph_fd, PEBLKFILL, &blkfl);
}
/*******************************************************/
```

Code Sample 6-6. Routine to Fill a Raster Block

Our "mindless pattern" is a set of vertical stripes across the window. The PEBLKFILL **ioctl()** call writes the whole pattern *at once* to the graphics window. This is a high performance graphics window driver routine!

The `rr` field of the `blkfill` structure specifies the *replacement rule* of the fill--whether to AND, OR, EXOR, etc. the individual pixels as we overwrite them with our raster data. Possible values for replacement rules are included in **<scrn/disp.h>**.

The NEW replacement rule in the code sample means to overwrite completely the existing data in the window with the new raster data.

6.3.6.2 Writing a Graphics Raster

The following **write_raster()** routine writes a block raster at location (`x,y`).

Other parameters are:

- Parameters `width` and `height` define the dimensions of the raster in pixels.
- Character pointer `data_ptr` is the address of the raster data.

- Parameter rr specifies one of one of the sixteen replacement rules.

```
/********************************************************/

write_raster (x, y, width, height, data_ptr, rr)
int x, y;
int height, width;
unsigned char *data_ptr;
char rr;
{
   struct blkwrite block;

   block.x = x;
   block.y = y;
   block.width = width;
   block.height = height;
   block.rr = rr;
   block.data = data_ptr;

   ioctl (graph_fd, PEBLKWR, &block);
}
/********************************************************/
```

Code Sample 6-7. Routine to Write a Raster Block

Note that the data_ptr must be aligned on a *word boundary* (16 bits); otherwise the PEBLKWR **ioctl()** call will set the errno variable to EINVAL.

Also, if your raster data is not by nature 16 bits wide, you should pad the raster data to a multiple of 16 bits per row, as the **RasterBlaster** program below will do.

6.3.6.3 Reading a Graphics Raster

The following **read_raster()** routine reads and then stores a 6 by 8 pixel pattern from the screen in the global variable back_grnd.

```
/*******************************************************/

read_raster(xloc, yloc)
{
     struct blkread blk_rd;

     blk_rd.x = xloc;
     blk_rd.y = yloc;
     blk_rd.width = 6;
     blk_rd.height = 8;

     blk_rd.data = back_grnd;
        /* use the global variable back_grnd */

     ioctl (plot_fd, PEBLKRD, &blk_rd);
}
/*******************************************************/
```

Code Sample 6-8. Routine to Read a Raster Block

Variable `back_grnd` is a 32-byte character array to hold the 16 by 16 pixel pattern.

The `width` and `height` fields are "hard-coded" to a 6 by 8 block size--your own routine can be more flexible.

6.3.6.4 Tying It All Together

The following **RasterBlaster** program sews the preceding code samples together to blast a pattern over the whole graphics window and to write a character "glyph" (the letter A) across the window.

```
/*******************************************************/

#include <fcntl.h>
#include <scrn/smsysdep.h>
#include <sys/types.h>
#include <scrn/disp.h>
#include <scrn/plotem.h>
#include <scrn/wmcom.h>

#define XSIZE  512  /* use the full size */
#define YSIZE  255
#define XLOC   0
#define YLOC   0

int  graph_fd;    /* graphics window file descriptor */

char glyph[] =   /* a 6x8 pattern,
              ** right-hand 10 bits are ignored
              */
     {
     0xc7, 0x00,    /* ..XXX. .. XXXXXXXX */
     0x9b, 0x00,    /* .XX..X .. XXXXXXXX */
     0x9b, 0x00,    /* .XX..X .. XXXXXXXX */
     0x83, 0x00,    /* .XXXXX .. XXXXXXXX */
     0x9b, 0x00,    /* .XX..X .. XXXXXXXX */
     0x9b, 0x00,    /* .XX..X .. XXXXXXXX */
     0x9b, 0x00,    /* .XX..X .. XXXXXXXX */
     0xff, 0x00,    /* ...... .. XXXXXXXX */
     };

char back_grnd[32];
/* global to store a 16x16 pixel pattern */

main ()
{
   int x,y;

   graph_fd = open_graph (XSIZE, YSIZE, XLOC, YLOC);

   fill_raster (0, YSIZE-1, YSIZE, XSIZE);
```

```
    while (1) /* forever -- use [Stop] to stop */
        {
        /* move a glyph across display */
        for (x=500; x>10; x-=6)
         {
         read_raster (x,100);
         write_raster (x, 100, 6, 8, glyph, NEW);
         for (y=0; y<2000; y++); /* pause to see glyph */
         write_raster (x, 100, 6, 8, back_grnd, NEW);
         }
        }
}
/*******************************************************/
```

Code Sample 6-9. Program To Write and Read Rasters

For more information on writing and reading rasters in a graphics window, refer to the **PLOTEM** file (for *plotter emulator*) on the C language documentation disc.

6.3.7 Closing a Graphics Window

The **close_graph()** routine simply cleans up when your program has finished doing business with a graphics window.

```
/*******************************************************/

close_graph()
{
     close (graph_fd);
}
/*******************************************************/
```

Code Sample 6-10. Routine To Close a Graphics Window

Note that the initial **open_graph()** routine has set the graphics window to **AUTO_DEST** so that the graphics window will be eliminated when the **close()** call is made.

6.4 HP-GL Plotter Commands

Following is a brief summary of the HP-GL plotter commands that are available to you, as recognized by the HP-GL interpreter:

Command	Description
CA	Designate alternate character set.
CP	Character plot.
CS	Designate standard character set.
DC	Digitize clear.
DF	Default instruction.
DI	Absolute direction.
DP	Digitize point.
DR	Relative direction.
DT	Define Terminator.
IM	Input mask.
IN	Initialize.
IW	Input window.
LB	Label.
LT	Line Type.
OA	Output actual position and pen status.
OC	Output commanded position and pen status.
OD	Output digitized point and pen status.
OE	Output error.
OF	Output factors.
OI	Output identification.
OO	Output options.
OP	Output P1 and P2.
OS	Output status.
OW	Output window.
PA	Plot absolute.
PD	Pen down.
PG	Page.
PR	Plot relative.
SA	Select alternate character set.
SC	Scale.
SI	Absolute character size.
SL	Character slant.
SM	Symbol mode.
SP	Pen select.
SR	Relative character size.
SS	Select standard character set.

Command	Description
TL	Tick length.
XT	X-tick.
YT	Y-tick.

Refer to the *HP-GL Reference Manual* for detailed descriptions of individual HP-GL commands.

6.5 Controlling a Graphics Window Using HP-GL Commands

Here are the steps in controlling a graphics window by means of HP-GL command strings:

1. Load the HP-GL interpreter into system memory by running the **load_hpgl** program supplied on the system disc.[2]
2. Create an HP-GL graphics window by running the **plotter_is** program supplied on the system disc.
3. Open the **/dev/plotter** device using an **open()** call.
4. Use **write()** statements afterwards consisting of three parts:
   - The *file descriptor* returned from the **open()** call.
   - The character string that is the HP-GL command.
   - The number of bytes that are in the character string.

Alternately, you can use **fprintf()** statements that specify a *file pointer* to the graphics window to write out your HP-GL command strings.

6.6 An HP-GL Example

The following "spiral" program draws a spiral in a graphics window consisting

2. This step is necessary only for Release 1.0.0 of the operating system.

of about 10,000 individual HP-GL line segments.

Remember that you need to have run the **plotter_is** utility to create a graphics window before running this program:

```
/*******************************************************/

#include <stdio.h>
#include <math.h>    /* Note: add '-lm' option in the
              ** 'cc' command line to link in
                 ** the math library routines.
                 */

FILE *gw_fp, *fopen();

main()
/*
** Use HP-GL commands to draw spirals
*/
{
    float j, ell;
    int k, k1, k2, k3;

    /* open the plotter device as a 'stream' */
    gw_fp = fopen("/dev/plotter","r+");

    init_graph();

    k1=10000; k2=10; k3=1; ell=0;

    for (k = ell; k <= k1; k += k2, ell += k3)
       {
       j = k/57.29577951;
       fprintf(gw_fp,"PA%3.0f,", sin(j)*ell*.2 + ell*.2);
       fprintf(gw_fp,"%3.0f;", cos(j)*ell*.1 + ell*.1);
       }
}
init_graph() /* initialize the graphics window */
    {
    fprintf (gw_fp, "in; pg; pu; pa0,0; pd;");
    }
/*******************************************************/
```

Code Sample 6-11. Program To Draw Spirals Using HP-GL Commands

6.7 Further Information

For more information on the **plotter_is** utility, refer to the owner's documentation for the computer.

For information on individual HP-GL commands, refer to the *HP-GL Reference Manual.*

For more information on individual low-level graphics requests, refer to the **PLOTEM** file on the C language documentation disc.

For a listing of keycode mnemonics for graphics windows, refer to the **<scrn/keycode.h>** header file.

Chapter 7

Alpha/Graphics Windows

7.1 Concepts

The Integral Personal Computer offers a third window type that enables you to mix text and graphics in the same window.[1] *Alpha/Graphics windows* offer the following capabilities:

- Support for both vector (line-oriented) operations and raster (block-oriented) operations.
- Positioning and writing of text.
- Support for downloadable character fonts.
- Support for terminal *raw mode* and other tty characteristics.
- Significant performance gains over the *graphics window* type.

The disadvantages of alpha/graphics windows are:

- Currently, the only way to program alpha/graphics windows is by means of low-level, *non-portable* system calls, specific to the Integral Personal Computer.
- In order to achieve its performance, alpha/graphics windows do not handle *moving* and *stretching* as gracefully as other window types. You will probably want to disallow moving and stretching of an alpha/graphics window altogether.

For most applications, the advantages of alpha/graphics windows far outweigh the disadvantages. This is especially true if your application mixes text and graphics. We recommend that you use alpha/graphics windows if you're constructing a graphically oriented application specifically for the Integral Personal Computer.

1. The Alpha/Graphics windows type is not available on Release 1.0.0 of the computer.

This chapter shows how to create and use alpha/graphics window. For information on downloadable fonts, refer to Chapter 8, *User-Definable Fonts*.

Note

The following routines are specific to the Integral Personal Computer operating system and will not port to other HP-UX systems.

Be sure that you isolate these system-dependent routines in a separate module of your software.

7.2 Creating an Alpha/Graphics Window

The following declarations will be used throughout this chapter:

```
/*******************************************************/

#include <stdio.h>   /* optional--for writing out */
#include <errno.h>   /*             error messages */
#include <sys/types.h>
#include <termio.h>           /* for raw mode code */
#include <scrn/keycode.h>   /* for keyboard input */
#include <scrn/wmcom.h>   /* for window structure */
#include <scrn/disp.h>
#include <scrn/ag.h>  /* for ag window structures */
#include <fcntl.h>          /* to open() the window */

extern int errno;   /* necessary on Release 1.0.0 */

int ag_fd;       /* file descriptor for ag window */

/*******************************************************/
```

Code Sample 7-1. Declarations for an Alpha/Graphics Window

Creating an alpha/graphics window involves the same sequence of steps as that for other window types:

1. Use a **WMGET ioctl()** call to "get" the existing parameters for the window that your application is running in.
2. Set the various fields of the `windio` structure to the desired values.
3. Use a **WMCREATE ioctl()** call to create the window according to the new values.
4. Use an **open()** system call to open the window for subsequent writes and reads.

The following **ag_open()** routine enables you to create an alpha/graphics window according to a given size, location, and name:

```
/*******************************************************/

ag_open (xsize, ysize, xloc, yloc, w_name)
int xsize, ysize, xloc, yloc;
char *w_name;
{
    struct windio ag_win;

    extern char *strrchr();
    char *dummy;

    /* get default window attributes */
    ioctl (0, WMGET, &ag_win);

    /* now set the fields -- first the name */
    /* find the last slash in the path name */
    dummy = strrchr(ag_win.w_path, '/');

    /* zero out the end of the current name */
    *(++dummy) = 0;

    /* now add the name our calling routine wants */
    strcat (ag_win.w_path, w_name);

    /* it's an alpha/graphics window */
    ag_win.w_type = AGTYPE;

    ag_win.w_stat |= AUTO_SH | AUTO_ACT | AUTO_DEST;
    ag_win.w_width = xsize;
    ag_win.w_height = ysize;
    ag_win.w_xloc = xloc;
    ag_win.w_yloc = yloc;
    ag_win.w_gen1 = 0;
    ag_win.w_gen2 = 0;

    /* try to create the window */
    if (ioctl (0, WMCREATE, &ag_win) < 0)
       {
       printf("unable to create %s\n",ag_win.w_path);
       printf ("errno --> %d\n",errno);
       ag_fd = -1;
       }
    else
       {
       /* open the window for reading/writing */
       ag_fd = open (ag_win.w_path, O_RDWR);
       if (ag_fd < 0)
       printf("unable to open %s\n",ag_win.w_path);
       }
}
/*****************************************************/
```

Code Sample 7-2. Routine To Create an Alpha/Graphics Window

Note that the **open()** call returns in global variable `ag_fd` the file descriptor for use in subsequent reads and writes to the alpha/graphics window.

Fields `w_gen1` and `w_gen2` set the size of the "plotter bed" of the alpha/graphics window, measured in pixels. If both parameters are `0` ("zero"), then the plotter bed will default to 512 pixels wide by 255 dots high. In order to avoid unexpected results, we recommend that you always use the default size by setting `w_gen1` and `w_gen2` to zero.

The code sample uses **printf()** statements to report errors. Note that **printf()** uses **stdout** as its file descriptor, which will be the *alpha window* associated with your application.

7.3 On Moving and Stretching Alpha/Graphics Windows

Normally, the user has a great deal of freedom in moving windows around the display and *stretching* (or changing the size of) windows with the $\boxed{\text{Move}}$ and $\boxed{\text{Stretch}}$ function keys on the System menu.

A certain amount of freedom needs to be sacrificed to accommodate alpha/graphics windows. In particular, the user should not be allowed to move an alpha/graphics window partly off the display--any data written to the offscreen portion of the window will be lost!

Similarly, complications may arise if the user is allowed to stretch an alpha/graphics window.

To forestall any problems, we recommend that you set the NO_MOVESTRETCH bit of the `windio` structure for your alpha/graphics window. Refer to Chapter 3, *Windows*, for information on setting window attributes.

With the NO_MOVESTRETCH bit set, pressing the $\boxed{\text{Move}}$ and $\boxed{\text{Stretch}}$ function keys will have no effect on the window.[2]

7.4 Setting the Replacement Rule

The replacement rule controls the logic used to determine whether the resulting display pixels are turned on or off. There are sixteen possible pixel operations, as defined in **<scrn/disp.h>**:

Rule	Operation
FORCE_ZERO	Paint it black (or, if window color is inverted, paint it white).
AND	(New Display Data) AND (Old Display Data)
AND_NOT_OLD	(New Display Data) AND ~(Old Display Data)

2. It is also possible to set up a signal interrupt to detect when a move or a stretch is attempted and then limit a move to keep the window totally onscreen.

Rule	Operation
NEW	Overwrite with New Display Data.
AND_NOT_NEW	~(New Display Data) AND (Old Display Data)
OLD	Leave Old Display Data.
EXOR	(New Display Data) EXOR (Old Display Data)
OR	(New Display Data) OR (Old Display Data)
NOR	(New Display Data) NOR (Old Display Data)
EXNOR	(New Display Data) EXNOR (Old Display Data)
NOT_OLD	~(Old Display Data)
OR_NOT_OLD	(New Display Data) OR ~(Old Display Data)
NOT_NEW	~(New Display Data)
OR_NOT_NEW	~(New Display Data) OR (Old Display Data)
NAND	(New Display Data) NAND (Old Display Data)
FORCE_ONE	Paint it white (or, if window color is inverted, paint it black).

These are the same sixteen rules mentioned in the description of the **fill_raster()** and **write_raster()** routines in Chapter 6, *Graphics Windows*.

The setting of the replacement rule effects the results of all calls to **ag_write_raster()**, **ag_draw_line()**, and **ag_draw_polylines()**.

The most recently set replacement rule is the one in effect for a given window.

You can use the following **ag_replace_rule()** routine to set the replacement rule for an alpha/graphics window:

```
/*******************************************************/

ag_replace_rule (fd, rr)
int fd;     /* file descriptor for target window */
int rr;     /* desired replacement rule */
{
    ioctl (fd, AGWRRR, rr);
}
/*******************************************************/
```

Code Sample 7-3. Routine To Set the Replacement Rule

7.5 Writing Characters to an Alpha/Graphics Window

Once you've created and opened an alpha/graphics window, you can start using the window, as for writing alphanumeric characters.

Chapter 8, *User-Definable Fonts*, shows how to download *font files* into memory, how to access different fonts from an alpha/graphics window, and how to write the characters in a given font to the alpha/graphics window.

In a nutshell, the technique is to use the **AGMVWRW ioctl()** call to move character data to the window. Refer to Chapter 8 for more information.

7.6 Drawing Vectors in an Alpha/Graphics Window

Alpha/graphics windows support both vector and raster operations. The next few pages cover vector operations and describe two different ways that you can draw vectors (or lines) in an alpha/graphics window.

The assumption is that you're using a valid `ag_fd` file descriptor for the alpha/graphics window.

7.6.1 Drawing Lines

The following **ag_draw_line()** routine draws a line to a given `x`, `y` location in an alpha/graphics window, *beginning from the current pen position.*

If the pen is in the up position, then the pen will be moved to that location and no line will be drawn.

```
/*****************************************************/

ag_draw_line (ag_fd, x, y, pen_state)
int ag_fd, x, y, pen_state;
{
     struct vect_info next_vector;

     next_vector.x = x; /* end points for the line */
     next_vector.y = y;
     next_vector.pen_down = pen_state;

     ioctl (ag_fd, AGLINEW, &next_vector);
}
/*****************************************************/
```

Code Sample 7-4. Routine To Draw a Line in an Alpha/Graphics Window

Note that this routine takes the coordinates of the *end point* of the line.

If `pen_state` is set to zero (FALSE), then the pen will be up and no line will be drawn. If `pen_state` is set to non-zero (TRUE), then the pen will be down during the move, and a line will be drawn.

7.6.2 Drawing Polylines

One of the capabilities of the alpha/graphics window type is its ability to draw *polylines*--that is, many lines at once.

If you can assemble a group of line definitions within your application, then you can "blast" them all to the alpha/graphics window with a single **ioctl()** call. The result is a significant performance increase over single vector draws.

The following **ag_draw_polylines()** takes a pointer to an array of line definitions and writes them to the alpha/graphics window.

```
/*******************************************************/

ag_draw_polylines(ag_fd, count, list)
int ag_fd, count;
struct vect_info list[];
{
   struct poly_info line_info;

   line_info.count = count;
   line_info.cp = (char *)list;
   ioctl (ag_fd, AGPOLYLINEW, &line_info);
}
/*******************************************************/
```

Code Sample 7-5. Routine To Draw Many Lines at Once

The pass parameter `list` is a pointer to an array of `vect_info` structures, which are defined in **<scrn/ag.h>** as follows:

```
/*******************************************************/

struct vect_info {
     short x;
     short y;
     short pen_down;
}
/*******************************************************/
```

Code Sample 7-6. The Structure of a Vector

The alpha/graphics window driver checks to make sure that the first and last elements of the vector list do, in fact, reference valid memory.[3] Otherwise, no data integrity checks are made.

Starting from the current position, the pen is moved to each `x`, `y` position indicated in order. If `pen_down` is TRUE (non-zero), a line will be drawn. otherwise, the pen is simply moved to the specified point.

3. Attempting to reference nonexistent memory will cause `EFAULT` to be returned in the `errno` variable.

The drawing/moving will continue until `count` number of lines has been processed.

7.7 Writing and Reading Raster Data

Alpha/graphics windows enable you to write raster data and to read raster data. As for graphics windows, raster operations in alpha/graphics windows use *one bit per pixel.*

7.7.1 Writing Raster Data

The following **ag_write_raster()** routine writes a block of raster data to the alpha/graphics window specified by the `ag_fd` file descriptor. Note the similarity between this routine and the **write_raster()** routine from Chapter 6, *Graphics Windows*:

```
/****************************************************/

ag_write_raster
(x,y, height,width, bit_offset,addr_offset, write_data)

int x, y, height, width, bit_offset, addr_offset;
char *write_data;
{
    struct agblk blk_write;

    blk_write.x = x;
    blk_write.y = y;
    blk_write.height = height;
    blk_write.width = width;
    blk_write.boffset = bit_offset;
    blk_write.aoffset = addr_offset;
    blk_write.data = write_data;

    if (ioctl (ag_fd, AGBLKWRW, &blk_write) < 0)
       printf ("Sorry--write/raster problem.\n");
}
/****************************************************/
```

Code Sample 7-7. Routine To Write a Raster Block

The parameters to the **ag_write_raster()** routine are as follows:

`x`	Horizontal position in window of block write (0 is left).
`y`	Vertical position in window of block write (0 is bottom).
`height`	Height in pixels of the raster write.
`width`	Width in pixels of the raster write.
`bit_offset`	Bit offset from word boundary.
`addr_offset`	Address offset between lines of pixels.
`write_data`	Buffer in which the raster data is placed.

The **ag_write_raster()** routine works exactly like the **write_raster()** routine from Chapter 6, *Graphics Windows*, except for the addition of the `bit_offset` and `addr_offset` parameters. If these parameters are set to 0 (zero), then the routine works the same as the **write_raster()** routine.

The rectangle on the display that is affected by **ag_write_raster()** is defined by the `x`, `y`, `height`, and `width` parameters. Parameters `x` and `y` determine the upper left corner of the affected rectangle, while parameters `height` and `width` determine the size of the rectangle.

The memory locations that hold the raster image to be written are defined by the `bit_offset`, `addr_offset`, `write_data`, `height`, and `width` parameters.

Perhaps a picture will help:

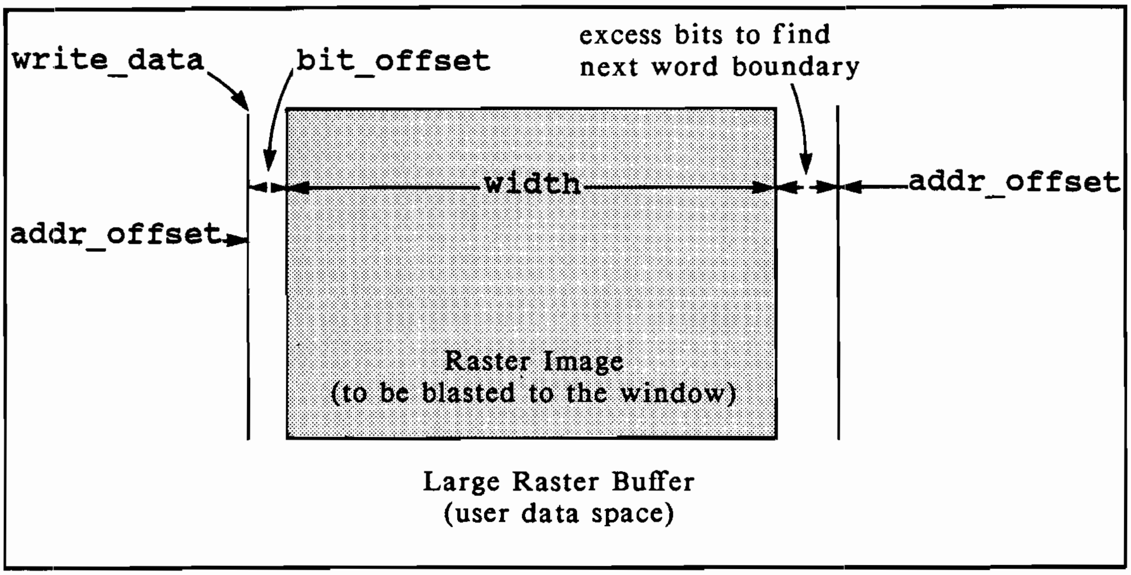
write_data
bit_offset
excess bits to find next word boundary
width
addr_offset
addr_offset
Raster Image (to be blasted to the window)
Large Raster Buffer (user data space)

Figure 7-1. The Layout of a Raster Image in an Alpha/Graphics Window

7.7.2 A Simple Example

Consider this example of using **ag_write_raster()**:

```
/********************************************************/

#define FACE_WIDTH  32
#define FACE_HEIGHT 28
#define XSIZE 100
#define YSIZE 100
#define XLOC 200
#define YLOC 75

char face[] =
{   /* raster image of a cartoon face */
    /* array of 4 bytes of 28 rows */
    0xff,0xff,0xff,0xff,
    0xfe,0x00,0x16,0xff,
    0xfd,0x50,0x1b,0x7f,
    0xfa,0xb5,0x4d,0xbf,
    0xfb,0xd6,0xb6,0xdf,
    0xf7,0xfb,0x5b,0x5f,
    0xef,0xfd,0xfd,0xef,
    0xdf,0xff,0xff,0xf7,
    0xb8,0x1f,0xf0,0x3b,
    0x7f,0xef,0xef,0xfb,

    /* the next line begins at address &face[40] */
    0x7c,0x3f,0xf8,0x7d,
    0x7b,0xdf,0xf7,0xbd,
    0x76,0x6f,0xec,0xdd,
    0x6c,0x36,0xd8,0x6d,
    0x76,0x6e,0xec,0xdd,
    0x7b,0xde,0xf7,0xbd,
    0x7c,0x3d,0x78,0x7b,
    0x7f,0xfb,0xbf,0xfb,
    0xbf,0xff,0xff,0xf7,
    0xdf,0x1d,0x73,0xef,
    0xef,0x62,0x8b,0xef,
    0xf7,0xbf,0xf7,0xdf,
    0xfb,0xdf,0xef,0xbf,
    0xfd,0xe0,0x1f,0x7f,
    0xfe,0xff,0xfe,0xff,
    0xff,0x7f,0xfd,0xff,
    0xff,0x9f,0xf3,0xff,
    0xff,0xe0,0x0f,0xff,
};

main()
{
    /* open an a/g window called 'faces' */
    ag_open (XSIZE, YSIZE, XLOC, YLOC, "faces");

    /* set the replacement rule to NEW */
    ag_replace_rule (ag_fd, NEW);

    /* draw the face raster on the display */
    ag_write_raster
      (30,70, FACE_HEIGHT,FACE_WIDTH, 0,0, face);

    /* draw an eye from the face raster on the display */
    ag_write_raster (80,70, 7,10, 3,2, &face[40]);

    /* let go of the CPU */
    pause();
}
/******************************************************/
```

Code Sample 7-8. Program To Write Raster Blocks

When run, the program will create an alpha/graphics window and display a little picture.

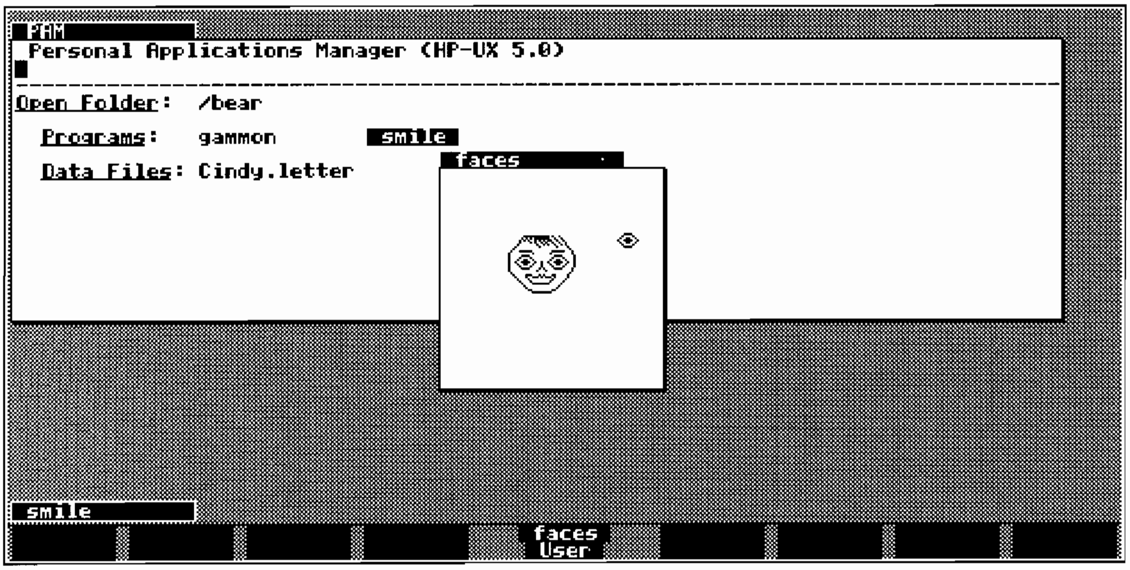

Figure 7-2. Program Output to an Alpha/Graphics Window

Consider the following from the code example:

```
ag_write_raster (30,70,FACE_HEIGHT,FACE_WIDTH,0,0,face);
```

This call causes a raster image to be written at (30,70) in the alpha/graphics window, with a height of 28 (FACE_HEIGHT) and a width of 32 (FACE_WIDTH). Beginning at the base address of `face`, every 32 bits (4 bytes) defines a row of corresponding pixels on the display.

7.7.3 Writing Small Rectangles From a Large Raster Image

You may want to maintain in your application a large bit image of a raster--larger, even, than the display size. To output a specific part of your bit image, you will want to define a "subrectangle" of the bit image to write to the display.

At such times the `bit_offset` and `addr_offset` parameters are useful. Consider the line from the code example:

```
ag_write_raster (80,70, 7,10, 3,2, &face[40]);
```

This call extracts the left eye from the `face` raster image and writes it on the display. The left eye image begins at the address of `face[40]`. However, since the `bit_offset` is now 3, then the first three bits are *not* included in the raster image. Since the width of the raster image is 10, then next 10 bits are used to form a row of the image. Any remaining bits left in the current word (16 bits) are also ignored.

Then, we add `addr_offset` to the current address being used, in order to find the start of the next row. In this code example, the `addr_offset` is 2, causing the 2 bytes forming the right eye of the face to be ignored.

Note that the `addr_offset` should always be 0 or even, because it represents the number of bytes between word boundaries.

7.7.4 Reading Raster Data

The alpha/graphics window provides the ability to read raster images from the display and to store the results at an address supplied by your program.

Reading rasters is exactly the inverse of writing rasters to the display. The following **ag_read_raster()** routine parallels the **ag_write_raster()** routine:

```
/*******************************************************/

ag_read_raster
(x,y, height,width, bit_offset,addr_offset, read_data)

int x, y, height, width, bit_offset, addr_offset;
char *read_data;
{
    struct agblk blk_read;

    blk_read.x = x;
    blk_read.y = y;
    blk_read.height = height;
    blk_read.width = width;
    blk_read.boffset = bit_offset;
    blk_read.aoffset = addr_offset;
    blk_read.data = read_data;

    if (ioctl (ag_fd, AGBLKRDW, &blk_read) < 0)
       printf ("Sorry--read/raster problem.\n");
}
/*******************************************************/
```

Code Sample 7-9. Routine To Read Raster Data

In the preceding routine the parameter `read_data` is the address of a buffer supplied by the program that is large enough to hold the raster image including any padding required by non-zero `addr_offset`, or `bit_offset`.

Also, `x`, `y`, `height`, and `width` still determine the rectangle effected on the display. The memory locations used to store the pixel image are determined by `read_data`, `bit_offset`, `width`, and `addr_offset`.

7.8 Handling Keyboard Input

As with other window types, you can put an alpha/graphics window in *raw mode* in order to intercept keystrokes (such as up-arrow and [Tab]) before they are sent on to the graphics "terminal" for processing.

The following **ag_set_raw()** routine does just that:

```
/*******************************************************/

ag_set_raw()
{
     struct termio agio;

     /* Get current terminal attributes */
     ioctl(ag_fd, TCGETA, &agio);

     /* Turn off canonical processing */
     /* This is the key! */
     agio.c_lflag &= ~ICANON;

     /* Get characters when one is ready */
     agio.c_cc[VMIN] = 1;
     agio.c_cc[VTIME] = 0;

     /* Ignore SIGQUIT -- the [Stop] key.
     ** (If ISIG is not cleared, then the
     ** the up-arrow key will send a two
     ** character sequence of \233 \003
     ** and \003 is SIGQUIT!)
     */
     agio.c_lflag &= ~ISIG;

     /* Turn off echo */
     agio.c_lflag &= ~ECHO;

     /* Now set the window */
     ioctl(ag_fd, TCSETA, &agio);
}
/*******************************************************/
```

Code Sample 7-10. Routine To Put an Alpha/Graphics Window In Raw Mode

Without this routine, the alpha/graphics window processes keyboard input *canonically*--the system will wait for the user to press a carriage return before allowing your application to look at a keystroke.

As with the other window types, you can set up an alpha/graphics window for either *blocking* or *non-blocking* reads.

For a discussion of how to read keystrokes in raw mode, refer to Chapter 6, *Graphics Windows.*

We don't offer the corresponding **ag_set_cook()** routine--we assume that you'll want to run your alpha/graphics application in raw mode only.

7.9 Closing an Alpha/Graphics Window

When done with your alpha/graphics window, you can close it with the following **ag_close()** routine.

```
/*******************************************************/

ag_close()
{
     close (ag_fd);
}
/*******************************************************/
```

Code Sample 7-11. Routine To Close an Alpha/Graphics Window

Since the initial **ag_open()** routine in this chapter created the alpha/graphics window as an autodestroy (AUTO_DEST) window, the **close()** call in this routine will eliminate the alpha/graphics window from the display.

7.10 Further Information

To learn the more intricate details of the alpha/graphics window driver, refer to the *Alpha/Graphics Window Reference Manual.*

Chapter 8
User-Definable Fonts

8.1 Concepts

The Integral Personal Computer enables you to:

- Download ready-made character fonts from disc into system memory.
- Select among fonts when writing to *alpha windows* and to *alpha/graphics* windows.[1]
- Create and modify custom fonts for your application, with an easy-to-use *font editor* program.

A *font* is a collection of character patterns (or *glyphs*) that specify up to 256 shapes, corresponding to ASCII decimal codes 0 through 255. Character patterns may be as large as 16 by 16 pixels.

A *font file* is a disc-based binary file consisting of width, height, and identification information for a particular font. All the characters in a given font must fit in the same-sized *cell*. For example, in the power-on system font, all characters fit in a cell size of 6 by 8 pixels.

The operating system is equipped with a system-wide *font pool*, which grows and shrinks dynamically in RAM, according to the number of fonts in use. At power on, the font pool is loaded from a Read-Only Memory file (**/rom/font**) and made ready for use.

When your application is using either an alpha window or an alpha/graphics window, you can issue programmatic calls to download a named font into the font pool.

Afterwards, you can write to the alpha window or alpha/graphics window using the newly installed font.

1. The third window type, graphics windows, offers a single HP-GL character font.

This chapter shows how to download a font from both alpha window applications and alpha/graphics window applications and how to select that font for use. (Refer to Chapter 6, *Graphics Windows*, for information on the HP-GL character set.)

8.2 Alpha Window Fonts

The fast alpha library (**/c/lib/libfa.a** on the C language translator disc) enables you to download and use disc-based font files in alpha windows.

Following is a brief summary of the fast alpha font manipulation routines:

Fast Alpha Call	Description
fainit()	Initializes the fast alpha driver.
fafontload()	Downloads a font file.
fafontactivate()	Activates a font in the font pool for fast alpha writes.
fafontremove()	Frees the memory used by the specified font.

Note that you *don't* need to worry about any other fast alpha routines in order to use the above font manipulation routines.

The general procedure for installing and using a disc-based font file in an alpha window is as follows:

1. Create the alpha window using the window manager routines described in Chapter 3.
2. Use a **fa_init()** call to initialize the fast alpha window driver for that window.
3. Use a **fafontload()** call to download the desired font file into the system font pool.
4. Use a **fafontactivate()** call to begin using fast-alpha writes in that font--*or*, alternately, use HP escape sequences to select the desired font.

8.2.1 Creating the Alpha Window

Chapter 3, *Windows*, shows how to create an alpha window (of type TOTYPE) to a given height and width.

Each alpha window can support only fonts of the *same size*. Thus, when you create an alpha window, you should specify the size of your *window* based on the size of *font* you wish to use.

For example, if you want to create a 72 column by 20 line window and want to use a 7 by 11 cell-size for the font, then you should choose:

72 x 7 = 504 pixels for the `w_width` field, and

20 x 11 = 220 pixels for the `w_height` field.

Chapter 3 includes a **NewWindow()** code sample that you can use to create and set the size of a new alpha window.

8.2.2 Initializing the Fast Alpha Driver

After creating the alpha window, you need to initialize the window for fast alpha calls. Chapter 5, *Alpha Windows*, contains an **alpha_open()** code sample that shows the use of the fast alpha initialization call, **fainit()**. Refer to Chapter 5 for more information.

By the way, the code samples on the next few pages use the following fast alpha declarations:

```
/*******************************************************/

#include <fa.h> /* Be sure to use the '-lfa' option in
                ** the 'cc' command line to link in the
                ** fast alpha library routines.
                */

int alpha_fd;   /* file des. for fast alpha driver */

int font_id_1;  /* font identifiers */
int font_id_2;

/*******************************************************/
```

Code Sample 8-1. Fast Alpha Declarations

From this point, we assume that variable `alpha_fd` is the file descriptor returned from a valid call to **fainit()**.

Variables `font_id_1`, `font_id_2`, etc. are small integers that will serve as "handles" for specifying which font we wish to access.

You may instead choose more descriptive names for the font identifiers, such as `Bold`, `Regular`, or `Math`.

8.2.3 Downloading a Font Using Fast Alpha

The fast alpha library contains the the **faloadfont()** routine for downloading fonts easily. To use the **faloadfont()** routine, you need to:

1. Create your alpha window (see above).
2. Initialize the fast alpha driver for that window (see above).
3. Call the **faloadfont()** routine using the path name of an online font file.

The following **alpha_load_font()** routine shows how:

```
/*******************************************************/

alpha_load_font (fontname)
char *fontname;
{
    int pass_font;
    if ((pass_font = fafontload (alpha_fd, fontname))
        == -1)
       printf("Sorry--can't load %s.\n",fontname);
    return(pass_font);
}
/*******************************************************/
```

Code Sample 8-2. Fast Alpha Routine To Download a Font

A typical invocation of **alpha_load_font()** is:

```
font_id_1 = alpha_load_font("/apps/fonts/block12x16");
```

--where `font_id_1` will be returned as the identifier for the named disc file.

If the **fafontload()** routine fails for any reason, the returned value will be a *-1*. You can handle that error as you see fit.

8.2.4 Using a Font in an Alpha Window

After you've downloaded a font into the font pool, you can select the font for your alpha window in one of two ways:

- By invoking the **fafontactivate()** fast alpha routine.
- By writing the appropriate HP alpha escape sequence to the alpha window.

8.2.5 Selecting Fonts With a Fast Alpha Routine

If you are using fast alpha routines to write characters to an alpha window, then you can use the **fafontactivate()** call to switch to a different font.

The syntax is:

```
fafontactivate(alpha_fd, font_id);
```

--where `font_id` is the identification number returned from a valid **fafontload()** call.

Use **fafontactivate()** when you are using either **fawrite()** or **farectwrite()** calls to write to the alpha window.

8.2.6 Selecting Alpha Fonts With HP Escape Sequences

You can select fonts for your alpha window by writing HP escape sequences to the window with standard C library **write()** and **printf()** routines.

Assuming that you've already downloaded one or more fonts into system memory, here are the key concepts:

- At a given time, an alpha window has *direct* access to two, 256-character sets:
  - The *base* font, which is the primary (or default) character set for the window.
  - The *alternate* font, which is a secondary character set for the window.

  For example, you may use a base font of normal roman characters and an alternate font of *italic* characters.
- To select a font, you write an escape sequence that specifies a particular font as being either the base or the alternate font.
- You switch between base and alternate fonts by sending simple control codes to the alpha window.

Note: When you initialize the fast alpha driver with **fainit()**, you automatically set the alpha window to ANSI mode. To use HP escape sequences, however, you need to set the alpha window back to HP mode. To do so, write the following escape sequence to the alpha window:[2]

```
ESC & k 0 \
```

To select the primary font, write the following escape sequence to the alpha window:

```
ESC ( <font_number> <A | B | ... Z>
```

--where the (is an "open parenthesis."

For example, to select the math symbol set as the primary font, use the following four-byte escape sequence:

```
write(1, "\033(8M", 4);
```

You can determine the `font_number` and the character identifier (A through Z) for a particular font by using the font editor **fedit** to examine those two parameters for the font.

To select the secondary font, write the following escape sequence to the alpha window:

```
ESC ) <font_number> <A | B | ... Z>
```

--where the) is a "close parenthesis."

For example, to select the courier 9 by 15 character set as the alternate font, use the following escape sequence:

```
write(1, "\033)7F", 4);
```

Note that if the specified font is not in the font pool, then these escape sequences are ignored and no font changes occur.

Finally, switching between base and alternate fonts is easy:

1. To switch to the alternate font, send an ASCII SO[3] to the alpha window.

2. We include embedded blanks for readability. Omit the blanks when actually writing the string.

3. Decimal code 14; octal code 16.

2. To switch back to the base font, send an ASCII SI[4] to the alpha window.

Accessing a third or a fourth font is typically a matter of selecting a new alternate font and "shifting out" to it.

If there is no alternate font, then the SO/SI character sequences will be ignored and the alpha window will use only the primary font.

For more information on selecting fonts with HP escape sequences, refer to the *Term0 Reference Manual.*

For information on using fast alpha routines to write characters and character enhancements, refer to Chapter 5, *Alpha Windows.*

8.2.7 Removing a Font From System Memory

The **fafontremove()** fast alpha call enables you to remove a font from the system font pool. The syntax is:

```
fafontremove(font_id);
```

If no other window is using that font, then the system reclaims the memory formerly occupied by the specified font.

8.3 Alpha/Graphics Window Fonts

Like alpha windows, alpha/graphics windows allow you to download font files from disc and to select fonts for use.[5] Both window types recognize the same font file formats on disc and access the same pool of fonts in system memory.

Unlike alpha windows, alpha/graphics windows allow more than one font size at a time to be written to the same window.

Another difference is that you have to make a series of low-level **ioctl()** calls to the alpha/graphics window driver to manipulate character fonts. Unlike the fast alpha library, no such alpha/graphics window library is currently available.

4. Decimal code 15; octal code 17.

5. Note that the material on these pages does not apply to Release 1.0.0 of the computer.

Note

The following routines are specific to the Integral Personal Computer operating system and will not port to other HP-UX systems.

Be sure that you isolate these system-dependent routines in a separate module of your software.

Even at a low level, manipulating fonts for alpha/graphics windows is fairly straight-forward:

1. After creating an alpha/graphics window, download a font with an **AGCREATEFONT ioctl()** call.
2. Select the new font with an **AGSELFONT ioctl()** call.
3. Write to the alpha/graphics window with a series of **AGMVRW** and **AGSETENHMASK ioctl()** calls.
4. When done writing, remove the font from the system font pool with a **AGREMFONT ioctl()** call.

8.3.1 Creating the Alpha/Graphics Window

Chapter 7, *Alpha/Graphics Windows*, shows how to create an alpha/graphics window (of type **AGTYPE**) with a specified location and size. Refer to the **ag_open()** code sample in Chapter 7 for more information.

By the way, the code samples on the next few pages use the following alpha/graphics window declarations:

```
/*****************************************************/

#include <scrn/disp.h>
#include <scrn/ag.h>

int ag_fd;     /* file descriptor for ag window */
long ag_font_id;  /* identifier for the desired font */

/*****************************************************/
```

Code Sample 8-3. Alpha/Graphics Window Font Declarations

From this point, we assume that variable `ag_fd` is the file descriptor returned from a valid **open()** call.

8.3.2 Downloading a Font Into an Alpha/Graphics Window

Once you've created the alpha/graphics window, use the following **ag_load_font()** routine to download a disc-based font file into the system font pool.

```
/*****************************************************/

ag_load_font (font_file_name)
char font_file_name[];
{
     int i, pass_font;
     struct font_data afont; /* from <scrn/disp.h> */

     /* build the name string */
     for (i=0; font_file_name[i]; i++)
         afont.filenm[i] = font_file_name[i];
     afont.filenm[i] = 0;

     /* do it! */
     if (ioctl(ag_fd, AGCREATEFONT, &afont) < 0)
        {
        printf("Can't access file %s\n",afont.filenm);
        return (-1);
        }
     else /* return the font id number for later use
          ** in font selection
          */
            return(afont.id);
}
/*****************************************************/
```

Code Sample 8-4. Low-Level Routine To Download a Font

A typical invocation of **ag_load_font()** is:

```
ag_font_id = ag_load_font ("/util/fonts/math6x8");
```

--where global variable `ag_font_id` will be your "handle" for subsequent font operations.

Note that the alpha/graphics window driver will first search the fonts already in the system font pool for the specified font before going out to the disc to look.

The `font_data` structure declared in **<scrn/disp.h>** consists of a number of fields:

```
/****************************************************/

struct font_data { /* data for getting font id */

     int esc_num;
        /* number used in ESC sequence for font */

     unsigned char esc_char;
        /* character used in ESC sequence for font */

     unsigned char width;
        /* width of character */

     unsigned char height;
        /* height of character */

     unsigned char cursval;
        /* cursor value */

     unsigned char curs_top;
        /* top of cursor */

     unsigned char curs_bot;
        /* bottom of cursor */

     long id;
        /* unique id associated with this font */

     unsigned char filenm[FILENAME_SIZE];
        /* font file path name */
};
/****************************************************/
```

Code Sample 8-5. The Structure of a Font

This underlying font structure is common to *both* alpha windows and alpha/graphics windows.

Only two of the `font_data` fields concern us here:

- The `filenm[]` field is a character array that contains the path name of the file on disc. It's the only parameter that you must supply to the above **ag_load_font()** routine.
- The `id` field is the "handle" for subsequent low-level font manipulations.

Although alpha windows and alpha/graphics windows share the same `font_data` structures when accessing the same font, the font `id` value is *not* the same as the value returned from a fast alpha **fafontload()** routine. Normally, you shouldn't have to worry about confusing the two id values.

8.3.3 Selecting a Font for an Alpha/Graphics Window

Once the font has been loaded, you need to select the font with another low-level call. The following **ag_select_font()** routine shows how:

```
/*******************************************************/

ag_select_font (font_num)
long font_num;
{
     if (ioctl(ag_fd, AGSELFONT, font_num) < 0)
        {
        printf("Can't select font\n");
        return(-1);
        }
}
/*******************************************************/
```

Code Sample 8-6. Low-Level Routine To Select a Font

Note that we can pass into the **ag_select_font()** routine the font identifier that was stored in global variable `ag_font_id` during the earlier **ag_load_font()** routine.

A typical invocation is as follows:

```
ag_select_font(ag_font_id);
```

8.3.4 Writing Characters to an Alpha/Graphics Window

Finally, we get to *use* the font that we've just selected. The following **write_char()** routine writes a character string to an alpha/graphics window, beginning at a specified *x,y* location and using the currently selected font.

```
/*******************************************************/

write_char (x, y, a_string)
int x, y;
char *a_string;
{
     struct mv_wr mywrite;

     mywrite.x = x;
     mywrite.y = y;
     mywrite.data = a_string;

     ioctl(ag_fd, AGMVWRW, &mywrite);
}
/*******************************************************/
```

Code Sample 8-7. Routine To Write Characters to an Alpha/Graphics Window

Note the following:

- The mnemonic AGMVWRW stands for *move and write*--the alpha/graphics window "pen" moves to the specified location and then starts writing characters.
- The `x` and `y` values are measured in *pixels*, relative to *0,0* (the origin at the *lower* left corner of the window).
- A negative `x` value will cause the pen to start "plotting" from its current location.
- The pen will be moved after each character to the beginning of the next character position.
- Data that would cause the pen to move out-of-bounds will be "clipped."
- Control characters such as carriage return, linefeed, and tab will have no effect on the position of the pen.
- If no font has been explicitly selected, the alpha/graphics window will use the system 6 by 8 ROMAN8 character set.

- If you have many "move and writes" to perform at the same time, you can use the **AGPOLYMVWRW ioctl()** call to increase performance.

For more information on the AGMVWRW request, refer to the *Alpha/Graphics Window Reference Manual.*

8.3.5 Setting Display Enhancements

You can set display enhancements (inverse and underline) for your character strings before writing them to the alpha/graphics window. The following **set_enhancement()** routine causes subsequent characters to be written according to the enhancement(s) passed in as `mask`.

```
/*******************************************************/

set_enhancement (mask)
int mask;
{
     ioctl(ag_fd, AGSETENHMASK, mask);
}
/*******************************************************/
```

Code Sample 8-8. Low-Level Routine To Set Display Enhancements

Display enhancements in an alpha graphics window are strictly *character-oriented.*[6] That is, once the alpha/graphics window is set to a given enhancement, it will keep writing subsequent characters in that enhancement until you explicitly change it.

Possible values for the display enhancement mask are defined in **<scrn/ag.h>** and include:

- AGINVERSE (inverse video).
- AGUNDERLINE (underline).
- AGINVERSE | AGUNDERLINE (the two values OR-ed together).

For more information on setting display enhancements, refer to the **ag** file on the C language documentation disc.

6. Alpha windows support both character-oriented and field-oriented display enhancements.

8.3.6 Removing a Font From an Alpha/Graphics Window

When done with a font, you can remove it from the system font pool with a low-level call to the alpha/graphics window driver. The following **ag_remove_font()** routine removes the font as specified by the `id` field in the `font_data` structure:

```
/*****************************************************/

ag_remove_font (font_num)
long font_num;
{
     struct font_data afont;

     afont.id = font_num;
     if (ioctl(ag_fd, AGREMFONT, &afont) < 0)
        {
        printf("Trouble while removing font\n");
        return(-1);
        }
}
/*****************************************************/
```

Code Sample 8-9. Low-Level Routine To Remove a Font

If no other application--including any alpha window application--is currently using the specified font, then the system will reclaim the memory from the font point. Otherwise, the font will remain in the font pool.

8.4 Creating Customized Fonts

The font editor (**fedit**) is an easy-to-use program that enables you to create and modify custom fonts for the computer.

Refer to the owner's documentation for the computer and to the online **fedit.doc** file to learn how to use **fedit**.

A particularly useful function key is [? Ident] (shifted function key [f7]) which displays the current values of the `font_data` structure and allows you to edit the font fields.

8.5 Miscellaneous

Note that each installed font requires from less than 2K to more than 7K of system memory, depending on the number of unique glyphs in that file and on the cell size. Usually, the size of a font is not an issue.

You can download a large number of fonts into system memory--the font pool will grow to accommodate them. However, due to memory constraints of the graphics processor unit, there may be a noticeable performance lag if you switch rapidly between three or more fonts.

8.6 Further Information

To learn more about fast alpha font routines, refer to the fast alpha files the C language documentation disc.

To learn more about low-level alpha/graphics window font operations, refer to the *Alpha/Graphics Window Reference Manual.*

Chapter 9

Memory Management

9.1 Introduction

Faced with designing a low-cost, flexible-disc based, transportable HP-UX computer having only 512K bytes of RAM, the engineering team had to be creative.

The result is that the memory organization of the Integral Personal Computer differs from other HP-UX implementations.

Note that in the following discussion the term *process* means your application--including its text and data space--as it is executed in system memory.

9.2 Memory Constraints

There are two major constraints in the memory organization of the computer:

- **Limited stack space:** When loading an application from disc, the Integral Personal Computer divides process space into a *text* segment (relocatable object code), a *data* segment (for static variables), and a *stack* segment (for automatic variables and function return addresses). As in other HP-UX systems, the data space is virtually unbounded; unlike other HP-UX systems, however, the stack space *is* bounded.

- **No bounds checking:** The Integral Personal Computer memory management hardware consists of four base registers, two for user text and data and two for supervisor data and stack segments.[1] *No bounds checking registers exist.* It's the responsibility of your application to run in a "well-behaved" manner and not to reference illegal memory addresses.

1. These registers are separate from, and in addition to, the MC68000 CPU registers.

It's possible for an out-of-control or malicious process to overwrite another's or the system's data space. Something as innocent as an illegal pointer reference can cause unexpected results.

Normally, however, neither of these constraints presents any problem.

9.3 Stack Space

By default, the operating system allocates each process a fixed-size 6K-byte user stack. The stack holds automatic variables (those defined within functions) and return addresses.

If insufficient stack space becomes a problem, manifested by segmentation faults or bus errors, then you may have to ensure that your application has enough stack space.

You can address the stack space issue in a number of ways:

1. By keeping the number of local variables to a minimum. For starters, consider declaring large arrays to be global and using `static` declarations to reserve data space for your variables.
2. By using the **malloc()** system call in subroutines when you feel comfortable handling memory allocations on your own.
3. By running in *shared text.* The operating system allows processes to share text (pure executable code segments) in memory. A pleasant side effect is that shared text programs are allotted about 512K bytes of stack space, which should be plenty of room.

   To create a shared text program, simply include the linker `-n` flag when invoking the C compiler. Example:

   ```
   cc -n -o output source.c
   ```

   This example causes an executable shared text file to be created (**output**) from the **source.c** source file. It will have a large stack space available to it.

   Note that shared text processes have *separate* (non-contiguous) text and data segments. It's *not* possible for program execution to jump to the data segment of a shared text program and continue executing.
4. By explicitly reserving for itself a larger chunk of stack space with the linker -R flag.[2]

The default stack grows from a relative address of 0x1800 toward 0x0000--for 6K bytes of available space. By specifying a larger starting address in the command line, you direct the linker to create relocatable code having a larger stack size. Examples:

```
cc -o myprog -R 3000 myprog.o

ld -o myprog -R 3000 /usr/lib/crt0.o myprog.o -lc
```

--both create a program with an extra 0x1000 (or 4,096) bytes of memory. (The area between 0x2000 and 0x1800 is reserved for the per process system stack.)

9.4 Efficient Forking: vfork()

It's common for an application to **fork()** a child process in a multi-tasking environment.

The standard **fork()** system routine makes a carbon copy of the parent process--text and data segments of parent and child are identical.

In a system with limited memory resources, you may choose to use the **vfork()** system routine instead. During a **vfork()**, only the *u_area*[3] and 2K bytes of stack space are duplicated, and parent and child share the same data and text space.[4]

On the Integral Personal Computer **vfork()**-ed children have a unique 2K-byte stack allocated to them. Any stack space used beyond this 2K limit is shared between the child and the parent.

The upshot is that using **vfork()** instead of **fork()** can save memory.

Chapter 2, *Your Software Development Environment*, includes a code sample in which a **vfork()** is issued. Note that the child process should use the **_exit()** call after a **vfork()** to exit safely.

2. Not possible on Release 1.0.0 of the operating system.

3. You may think of the u_area as the process description area.

4. At least until the child issues an exec() to overlay itself with a new process.

We recommend that you refer to the **vfork()** entry in the system call section of the *HP-UX Reference Manual* or on the C language documentation disc in order to examine the limitations of the **vfork()** system call. For example, it's possible for the parent and a **vfork()**-ed child process to corrupt each other's data space.

9.5 Sticky Bits?

You can set the *sticky bit* for a given process to improve system performance.

Normally, all vestiges of a process are removed from system memory when the process exits. Release 5.0 of the operating system enables you to keep frequently used process text segments in core. The result is that re-executing the same application requires less time for disc access.

A process must be compiled shared text (discussed above) in order to be made "sticky." That is, the process text segment will not cleared when the *user count* drops to zero unless the memory is needed by another process.

The way to change an application to "sticky" is to exercise the **chmod** utility with the proper bits specified:

```
chmod u+t myprog
```

This example sets the sticky bit for `myprog`.

After a program with a sticky bit is loaded into memory, the text segment will stay there until you reset or power off the computer.

As an alternative, you can simply load heavily used programs onto the electronic disc. The advantage is that you can easily delete programs from the Edisc when done with them. The disadvantage is that each Edisc program requires room both for its object file on the Edisc and for its text/data segments in core.

9.6 How Much Is Left?

As a transportable computer with a fixed amount of memory, the Integral Personal Computer has been designed to handle out-of-memory conditions gracefully.

For example, PAM simply won't allow an application to begin executing if there's not enough memory for it.

The **status** utility, shipped on the utilities disc, displays the current amount of system memory available for user programs and files. This is a dynamically changing quantity, and using **status** can only provide a snapshot of available memory.

9.7 System V Enhancements

Release 5.0 of the operating system conforms to the *System V Interface Definition: Kernel Extension* in its implementation of shared memory, semaphores, process control, and message control.

Following is a list of added System V calls, several of which enable processes to share physical memory space:

System Call	Description
acct()	Enable or disable process accounting.
msgctl()	Control message operations.
msgget()	Get a message queue.
msgsnd()	Send a message to a queue.
msgrcv()	Read a messsage from a queue.
plock()	Lock a process, text, or data in memory.
profil()	Profile execution time.
ptrace()	Trace a process.
semctl()	Control semaphore operations.
semget()	Get a set of semaphores.

System Call	Description
semop()	Perform operations on a set of semaphores.
shmctl()	Control shared memory.
shmget()	Get a shared memory segment.
shmat()	Attach a shared memory segment to the data space of the calling process.
shmdt()	Detach a memory segment.

9.8 Core Dumps

Release 5.0 of the operating system enables your programs to *dump core* when an abnormal termination of execution occurs.[5] Core dumps are useful in conjunction with program debugging tools to examine what went wrong where.

There are available two utilities, **core_on** and **core_off**, to switch the core dumping capability of your computer on and off, on a system-wide basis.

9.9 Further Information

For information on process and memory management, refer to the system call section of the *HP-UX Reference Manual* or the C language documentation disc.

5. Release 1.0.0 of the operating system does not create core dumps.

Chapter 10

The File System

10.1 Introduction

This chapter describes the characteristics and programmatic interface of the Integral Personal Computer file system.

Although the term *file system* sometimes refers to the group of files existing on a single disc *volume*, in this guide *file system* refers to the hierarchical collection of *all* files on *all* mass storage devices available to the computer.

10.2 File System Characteristics

With few exceptions, the Integral Personal Computer file system conforms to the *Bell System V Interface Definition.*

10.2.1 Disc Drivers

The operating system includes two built-in disc drivers:

- A disc driver for controlling discs that understand the *Amigo* protocol.
- A disc driver for controlling discs that understand the *Subset-80* protocol.

The operating system calls the appropriate disc driver whenever your application issues a file system call, such as **open()** or **lseek()**.

The disc driver in turn calls the HP-IB driver with the appropriate information for the appropriate disc, and lets the HP-IB driver handle the details.

At the application level, you don't need to concern yourself with the operating characteristics of either the disc driver or the HP-IB driver routines. Your programmatic access to the file system is by means of the two dozen or so *system level calls.*

10.2.2 Block Sizes

A *logical block* of storage is the smallest unit of information that can be accessed (allocated/read/written to) on a disc. Logical blocks consist of 1K (1,024) bytes and are the same for flexible discs, hard discs, and the *electronic disc.*

Although the logical block size may vary from the *physical block size* of a disc device, the operating system groups physical blocks into 1K "chunks" for your application.

10.2.3 File Formats

The *HP-UX Reference Manual* describes common file formats for HP-UX files. For example, the linker creates executable code files in an **a.out** format, as described in the *HP-UX Reference Manual.*

For files that your application creates, however, HP-UX puts no constraints on file formats, such as fixed record lengths. From your application's point of view, then, each file consists of nothing but a series of *bytes,* one after the other. Whatever structure a file may have is strictly defined by your application program.

10.2.4 Major and Minor Device Numbers

A principal difference between the Integral Personal Computer file system and other HP-UX systems is a difference in the representation of major and minor device numbers.

On the Integral Personal Computer major device numbers are *8-bit* and minor device numbers are *24-bit* quantities. The dev data structure used in the **bstat()** and **mknod** system calls, for example, reflect the difference in size.

10.3 Finding One's Way Around the File System

Chapter 2, *Your Development System Environment*, suggests a comfortable arrangement for your own file system as a software developer.

It's reasonable to assume that your end user has properly set the switch settings on any external discs drives, properly connected the drives, and has powered up the drives and the computer so that the file system is properly configured.

If your application resides on a single flexible disc, then you can also assume that the user knows how to insert the disc in the internal disc drive and how to start the application.[1]

If disc swapping is necessary during your application, then you can use the **bstat()** system call (described later in this chapter) to ensure that the proper disc is online.

10.3.1 The Electronic Disc

The electronic disc (or *Edisc*) functions similarly to a physical disc volume mounted in your file system. Your application can treat it as another disc drive.

Here are some differences:

- The electronic disc grows and shrinks dynamically, according to the number of files and directories the user sets up within it.
- At power off, all information within the electronic disc is *lost.* It's important that you save all important information to a physical disc volume before switching off your machine.

The Edisc uses up to *three-quarters* of the available user memory at a given time,[2] depending on how much space is requested for mass storage. The operating system periodically (every 10 seconds or so) frees up unused memory from the Edisc for general use. Thus, if the user isn't storing files in the Edisc, then the Edisc size is very small.

For performance reasons, the space used by Edisc files is not compacted when unused space is freed. Consequently, memory can be fragmented by heavy use of the Edisc. This can be cured by temporarily unloading the Edisc to another type of media, deleting the Edisc files (wait a few seconds here to allow the system to free up unused Edisc space), and then reloading the Edisc.

Note that a system reset ([Shift][Reset]) has no effect on the contents of the electronic disc.

10.3.2 The Root Directory

At power on, the root directory of your file system is mounted in the Edisc. Consequently, all references to absolute path names and directories that begin

1. If you think the user may lack this information, you can refer him or her to the online tutor or the getting started guide for the computer.
2. Up to one-half of available user memory on Release 1.0.0 of the operating system.

with slash ("/") will be based in the Edisc.

The **chroot** utility shipped with the computer and the **chroot()** system call enable you to change the root to another directory, although the change affects only the process that invokes the command and its children. Because of the workings of the dynamic RAM file system, we recommend that you leave the root just as it is in the Edisc.

10.3.3 The Internal Disc Drive

The internal disc drive has a special device name, **/dev/internal**, that is installed in the device directory during the power on sequence.

The *automount* capabilities of the internal disc drive are described below.

10.3.4 External Disc Drives

During the power on sequence, external disc drives are given device names based on their HP-IB interface address and switch settings. For example, D040 specifies an external disc.

Refer to the owner's documentation for the computer for more information on external disc drives.

10.3.5 The scan_discs Utility

At power on, the **scan_discs** program (in **/rom/scan_discs**) is automatically run to mount the disc in the internal disc drive (if any) and all external discs.

10.3.6 Automounting vs. Manual Mounting

The operating system supports automounting and dismounting of discs in the internal disc drive. To this end, the operating system always keeps *up to date* information on operations involving the internal disc drive.

The same is *not* true of discs in external flexible disc drives. Although external discs are automatically mounted at power on, the user is expected to run the **unmount_disc** utility before removing flexible discs from external drives. *Failure to do so can lead to corrupt data blocks on flexible discs.*

The **unmount_disc** utility is shipped with the computer for the purpose of writing out any buffered data to an external disc and properly closing the files on that disc. There is also a **mount_disc** utility for mounting external flexible discs *after* powering up the system.

For safety, you should check the return code of a file operation whenever attempting to read from a file or write to a file that may be on a flexible disc in the internal disc drive or in an external disc drive. Because users can easily remove flexible discs, it's your application's responsibility to ensure that a disc is still mounted in the file system when needed.

10.3.7 Copying Directories

The Integral Personal Computer does not provide a "copy directory" utility. However, you can easily write a simple shell script to do the task for you.

The following **copydir** shell script copies all the files and subdirectories in the current directory (recursively) to the specified destination directory:

```
/*******************************************************/

echo "Copying current directory to $1"
find . -print | cpio -pdv $1
echo "All done."

/*******************************************************/
```

Code Sample 10-1. Shell Script To Copy Directories

The script relies on the handy **find** and **cpio** utilities shipped with the HP-UX tools discs. The beauty of this script is that the file hierarchy of the source directory is preserved during the transfer to the target directory.

If, for example, you wanted to transfer all the files on a flexible disc named /**MyDisc** to the top level directory of a hard disc named /**usr**, then you would:

- Insert the /**MyDisc** flexible disc in the internal disc drive and let the computer automount the disc for you.
- Change your working directory to /**MyDisc**.
- Enter the following command in the shell window (assuming that **copydir** is on your search path):

  ```
  copydir /usr
  ```

The shell will use the **find** and **cpio** utilities to copy the subdirectories and files one by one while reporting on its progress.

10.4 Transferring Files Between Computers

There are two primary ways of exchanging files between computers:

- By *media* transfers (the disc containing the desired files is removed from one system and moved to the target system).
- By *data communications* transfers (the desired file is *uploaded* or *downloaded* electronically from one file system to another).

A number of utilities are available for supporting media transfers between the Integral Personal Computer and other computers.

10.4.1 LIF Utilities

LIF utilities (for *Logical Interchange Format*) enable you to transfer disc files from one HP-UX system to another system (including other Series 200, Series 300, and Series 500 computers).

The idea is to use a LIF utility to copy a file from the source system to a disc formated for LIF files, walk the disc to the target system, and use the corresponding LIF utility on the target system to copy the LIF file into the target file system.

10.4.2 MS-DOS Utilities

MS-DOS utilities enable you to move character data both ways between MS-DOS computers and HP-UX computers. Contact your Hewlett-Packard Sales Representative for information on MS-DOS/HP-UX utility programs.

10.4.3 The Datacomm Program

The Datacomm program provides both terminal capabilities for connecting the Integral Personal Computer to a host computer and file transfer capabilities for uploading and downloading files.

The Datacomm program provides both unprotocoled and protocoled transfers, using the Christensen umodem protocol.

A public domain software program, **umodem**, is widely available, which you can quickly port to a host machine in order to effect safe and easy file transfers.

The umodem protocol guarantees the data integrity of both 7-bit text files and 8-bit binary files. Data are packaged in 128-byte blocks with a checksum generated at the end of each block. If source and destination checksums fail to match, the block is retransmitted repeatedly for a specified number of tries.

Using the `FileUp` and `FileDn` capabilities of the Datacomm program is straightforward. For example, if you wanted to upload a binary file to the host computer using the **umodem** program, you would:

- Establish a Datacomm session with the host computer.
- Fill out the $\boxed{\text{?FileUp}}$ Datacomm form, specifying valid path names for the local (existing) file and the remote (to be created) file.
- Specify a `binary` file type.
- Use `umodem -rb` as the remote invocation, so that the host computer will expect to *receive binary.*
- Return to the Transfer menu of the Datacomm program and press the $\boxed{\text{FileUp}}$ key. Datacomm will handle the details of the transfer.

10.5 File System Routines

Following are *brief* summaries of the built-in system calls for the Integral Personal Computer that you can use to create, access, update, and erase individual files in the file system.

All the system calls return a function result indicating the status of the call. Typically, a zero or positive result indicates that the call completed successfully, and a *-1* (negative one) indicates an error. In the case of an error, the external variable `errno` will be set to indicate the cause of the problem.

The following paragraphs exist mainly to list the available file system operations and to point out implementation dependencies, if any.

In general, the main difference between Integral Personal Computer system calls and system calls for other HP-UX computers is that the Integral Personal Computer allows normal applications all capabilities previously reserved for the *super user.* The reason is obvious: The Integral Personal Computer is a single-user personal computer.

Refer to the "system calls" section of the *HP-UX Reference Manual* for complete descriptions of the file system calls. You can also refer to the individual entries on the C language documentation disc.

10.5.1 access()

The **access()** system call checks whether the specified file has the specified access permission.

```
int access (path, amode)
char *path;
int amode;
```

You don't need to be super user to use this system call on the Integral Personal Computer.

Note also that a file currently open for execution is considered writable.

10.5.2 chdir()

The **chdir()** system call causes a specified directory to become the working directory for the current application and any of its children.

```
int chdir (path)
char *path;
```

10.5.3 chmod()

The **chmod()** system call enables you to change the access permission bits on a file.

```
int chmod (path, mode)
char *path;
int mode;
```

You don't need to be super user to use this system call on the Integral Personal Computer.

Note also that the "save text image after execution" bit (sometimes called the *sticky bit*) is not supported on Release 1.0.0 of the operating system. Refer to Chapter 9, *Memory Management*, for more information.

10.5.4 chown() and fchown()

The **chown()** and **fchown()** system calls enable you to change the ownership of a file.

```
int chown (path, owner, group)
char *path;
int owner, group;

int fchown (fd, owner, group)
int fd, owner, group;
```

You don't need to be super user to use this system call on the Integral Personal Computer.

10.5.5 chroot()

The **chroot** system call causes the named directory to be considered the root directory for the current application. That is, all absolute path name searches will begin from the named directory.

```
int chroot (path)
char *path;
```

You don't need to be super user to use this system call on the Integral Personal Computer.

10.5.6 close()

The **close()** system call closes a specified file descriptor.

```
int close (fildes)
int fildes;
```

10.5.7 creat()

The **creat()** system call is used to create new files on mounted discs and to rewrite existing files.

```
int creat (path, mode)
char *path;
int mode;
```

You can also use the **open()** system call to perform the function of **creat()**.

On the Integral Personal Computer an application may have up to 20 files open at a time.

On the Integral Personal Computer you can't guarantee that a newly created file will physically exist on a disc until you close that file.

10.5.8 dup()

The **dup()** system call duplicates an open file descriptor. Typically, you duplicate a file descriptor for the purpose of having multiple accesses to the same file.

```
int dup (fildes)
int fildes;
```

On the Integral Personal Computer an application may have up to 20 files open at a time.

10.5.9 fcntl()

The **fcntl()** system call provides control over open files, at a fairly low--but still portable--level.

```
#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;
```

On the Integral Personal Computer an application may have up to 20 files open at a time.

Examine the **<fcntl.h>** header file for a list of possible **fcntl()** requests.

10.5.10 fsync()

The **fsync()** system call synchronizes and updates the information on the specified disc file with the information in system memory.

```
int fsync (int)
int fd;
```

10.5.11 link()

The **link()** system call creates a new directory entry for an existing file.

```
int link (path1, path2)
char *path1, *path2;
```

10.5.12 lseek()

The **lseek()** system call causes the file pointer for an open file to seek to a new physical location within the file.

```
long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;
```

10.5.13 mknod()

The **mknod** system call creates a directory, a special file (typically a device file), or an ordinary file.

```
#include <mknod.h>

int mknod (path, mode, dev)
char *path;
int mode;
dev_t dev;
```

You don't need to be super user to use this system call on the Integral Personal Computer.

On the Integral Personal Computer major device numbers are *8-bit* and minor device numbers are *24-bit* quantities.

10.5.14 mount()

The **mount()** system call enables your application to mount a disc in the file system.

```
int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;
```

You don't need to be super user to use this system call on the Integral Personal Computer.

The **mount()** and **umount()** system calls are not part of the automounting strategy of the internal disc drive and do *not* work well with the **mount_disc** and the **unmount_disc** utilities shipped with the computer.

Normally, these system calls should be unnecessary, and you should avoid using them in your application.

10.5.15 open()

The **open()** system call opens a file for reading or writing or both.

```
#include <fcntl.h>

int open (path, oflag [,mode])
char *path;
int oflag, mode;
```

On the Integral Personal Computer an application may have up to 20 files open at a time.

Examine the **<fcntl.h>** header file for a list of possible **open()** requests.

10.5.16 pipe()

The **pipe()** system call creates a temporary file (or *pipe*) for use in interprocess communication.

```
int pipe (fildes)
int fildes[2];
```

On the Integral Personal Computer an application may have up to 20 files open at a time.

Writes of up to 10,240 bytes of data are buffered by the pipe before the writing process is blocked.

10.5.17 read()

The **read()** system call reads data from a file.

```
int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

Information read from a disc by the operating system is cached in memory to speed up access to information in files. Consequently, not every read operation causes the system to access the physical medium. If you want to access the medium synchronously, you should execute a **sync()** before executing the **read()**.

10.5.18 stat() and fstat()

The **stat()** and **fstat()** system calls check the status of files and directories.

```
#include <sys/types.h>
#include <sys/stat.h>

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;

int bstat (path, buf)
char *path;
struct bstat *buf;

int bfstat (fildes, buf)
int fildes;
struct bstat *buf;
```

You may want or need to use the Integral Personal Computer **bstat()** and **bfstat()** calls in place of the standard HP-UX **stat()** and **fstat()** calls. The reason is that the `st_dev` and `st_rdev` fields in the `bstat` structure for the Integral Personal Computer are larger than the corresponding fields in the (conventional) `stat` structure to accommodate the larger major and minor device numbers of the computer.

Important: If your application examines the `st_dev` or `st_rdev` fields, then you should use the **bstat()** or **fbstat()** calls.

Note that the **bstat()** and **fbstat()** calls are implemented *only* on the Integral Personal Computer.

Refer to the **<sys/stat.h>** header file to examine the file information stored in the `stat` and `bstat` structures and to examine the differences between the two structures.

10.5.19 sync()

The **sync()** system call causes all information in memory that should be on disc to be written out.

```
sync()
```

10.5.20 truncate()

The **truncate()** system call causes a specified file to be truncated.

```
truncate (path, length)
char *path;
int length;
```

The Integral Personal Computer does not implement this system call.

10.5.21 umask()

The **umask** system call sets an application's file mode creation mask.

```
int umask (cmask)
int cmask;
```

10.5.22 umount()

The **umount()** system call unmounts a disc from the file system.

```
int umount (spec)
char *spec;
```

You don't need to be super user to use this system call on the Integral Personal Computer.

The **mount()** and **umount()** system calls are not part of the automounting strategy of the internal disc drive and do *not* work well with the **mount_disc** and the **unmount_disc** utilities shipped with the computer.

Normally, these system calls should be unnecessary, and you should avoid using them in your application.

10.5.23 unlink()

The **unlink()** system call removes directory entries and deletes files.

```
int unlink (path)
char *path;
```

10.5.24 ustat()

The **ustat()** system call returns file system statistics.

```
#include <sys/types.h>
#include <sys/ustat.h>

int ustat (dev, buf)
dev_t dev;
struct ustat *buf;
```

On the Integral Personal Computer major device numbers are *8-bit* and minor device numbers are *24-bit* quantities.

10.5.25 write()

The write() system call writes data on a file.

```
write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

If `O_NDELAY` is set in the `termio` structure, then `nbyte` can be less than or equal to 10,240 bytes.

10.6 Further Information

To learn more about the end user's view of the file system, the electronic disc, and the automounting capabilities of the computer, refer to the owner's documentation for the computer.

For complete descriptions of file system calls, refer to the "system calls" section of the *HP-UX Reference Manual.*

To learn more about the file attributes you can check and set, refer to the individual header files on the C language preprocessor disc.

Chapter 11

External Devices

11.1 Concepts

The Integral Personal Computer operating system enables you to control a variety of external devices by means of built-in and installable *device drivers.*

The operating system supports:

- A built-in printer driver for the internal printer and for external printers.
- A built-in plotter driver for external plotters.
- A built-in HP-IB driver for the built-in HP-IB interface.
- A built-in serial driver for the optional RS-232C serial interface.[1]
- Installable drivers for GP-IO, BCD, HP-IB, and HP-IL interfaces.
- An installable driver for the plug-in modem.
- A built-in HP-HIL driver for HP-HIL devices, such as the keyboard and mouse.
- Two built-in disc drivers for internal and external discs.
- A Device Independent Library (*DIL*) for programmatic access to certain interface drivers.

Each peripheral device requires the following pieces:

- The *hardware interface*, either built into the computer, installed in the back plane, or installed in an expander box.
- The peripheral *device* itself, with the appropriate switch settings and proper connection to the interface.

1. For Release 1.0.0 of the operating system, the serial driver is an installable driver.

- The prewritten *device drivers*--the software that translates **open()**, **read()**, **write()**, **ioctl()**, and **close()** calls from your application program into lower level calls.
- The optional DIL *device library*, which provides a number of high level routines that your application program can call to control external devices.
- Your *application program*, which will include calls to the device driver (or, optionally, calls to the device library).

This chapter shows how to control external devices, such as printers and plotters, at a programmatic level.

This chapter does *not* detail the operation of the disc drivers, the workings of the DIL library, programmatic access to the interface drivers, or the way to write your own device drivers.

11.2 Device Drivers

Application programs control external devices by means of *device drivers* for the individual devices. There is a one-to-one correspondence between a device type (such as an interface card or a printer) and the software device driver that controls that device type. (It's perfectly reasonable, however, for the same driver to control more than one device at a time.)

Device drivers enable you to address external devices in a way consistent across *all* HP-UX devices. For example, you can include **write()** statements to write graphics commands to an external plotter without having to know anything about interface registers--just as you can include **write()** statements to write strings of characters to an alpha window without knowing the details of the GPU.

Your application program deals with just the device driver, which handles all the details of communicating with the external device.

11.2.1 Using a Device Driver

Your application program establishes communication with a given device driver by means of a special file in the **/dev** directory, such as **/dev/plotter** and **/dev/lp** (for *line printer*).

When the application *opens* a device driver by means of the special **/dev** file, the operating system returns to the application a *file descriptor*--a small integer--that the application will use from then on to specify the particular device driver.

11.2.2 Sequence of Events

The application, operating system, and device driver work together in a well-defined manner. Following is a typical sequence of events that occurs when an application writes a string of characters to an HP-IB plotter.

1. After opening **/dev/plotter**, the application executes a **write()** call that specifies the file descriptor for the plotter driver and the desired character string.
2. Execution of the **write()** call passes control to the operating system, which in turns calls the plotter driver.
3. The plotter driver accepts the **write()** string and in *its* turn calls the built-in HP-IB driver.
4. The HP-IB driver sends the character string to the HP-IB interface, which finally ships the bytes out to the plotter device.

11.2.3 Interfaces and Devices

The above example showed that the plotter driver called the HP-IB driver. The plotter driver could just as well have called the *serial* driver (for a serial plotter) or the *HP-IL* driver (for an HP-IL plotter).

The idea is that you need to have the interface driver readily available in order to talk to the physical device connected to the interface. Both the HP-IB driver and the serial driver are part of Read-Only Memory (ROM) and are always available. Other interface drivers need to be loaded into system RAM by means of a utility, such as **load_modem**, before they can be used.

11.2.4 Talking Directly to an Interface Driver

It's possible to control external devices, such as an HP-IB voltmeter, directly through the HP-IB driver.

The main reason for calling interface driver routines directly is for building your own device driver, such as a customized instrument driver.

However, doing so requires an extensive knowledge of interface protocols and low-level i/o programming. We recommend that you use the DIL library for programming the desired interface.

11.2.5 Writing to Hardware Registers

At an even lower level, it's possible to write values directly to the hardware registers of a particular interface and to read values directly from those registers.

The main reason for accessing hardware registers directly is for performance.

Again, doing so is not recommended. DIL provides a set of *logical* registers to offer the same functionality at almost the same performance level, while eliminating the complexity of dealing directly with the hardware.

11.3 Controlling External Printers and Plotters

The following discussion describes programmatic control of external printers and plotters.

In order to start with a known configuration, you should assume that the end user has done the following before running your application:

1. Installed the interface in the computer.
2. Properly connected and powered the external device and the computer.
3. Loaded the appropriate interface driver. (The HP-IB and serial drivers are built into the computer and do not require loading.)
4. Properly configured the printer or the plotter. The **printer_is** and **plotter_is** utilities are shipped with the computer for this purpose.

11.4 Controlling a Printer Device

Controlling a printer device is easy:

1. You *open* the printer device by means of an **open()** call that specifies the **/dev/lp** printer driver.
2. You *write* to the printer device by specifying the printer file descriptor in a **write()** statement.
3. When you're finished with the printer, you *close* the printer device.

11.4.1 Initializing the Printer

Following are the relevant declarations for the printer code samples in this chapter:

```
/*******************************************************/

#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/lp.h>
#include <sys/ioctl.h>

int pr_fd; /* global printer file descriptor */

/*******************************************************/
```

Code Sample 11-1. Printer Declarations

The following **PrinterOpen()** routine returns the file descriptor for the currently active printer device in global variable `pr_fd`.

```
/*******************************************************/

PrinterOpen()  /* Open the printer driver */
{
     pr_fd = open("/dev/lp", O_RDWR);
     if (pr_fd == -1)
        {
        printf("Can't open printer.\n");
        error_routine(BAD_PRINT_OPEN);
        }
}
/*******************************************************/
```

Code Sample 11-2. Routine To Open the Printer

Note that you should assume that your end user has taken responsibility for setting up the active printer device, addressed by **/dev/lp**. The **printer_is** utility is shipped with the computer for this express purpose. (The default printer device is the built-in printer.) Refer to the owner's documentation for the computer for a description of the **printer_is** utility.

Note that the **open()** of **/dev/lp** will always succeed, even if the end user hasn't properly set up the printer. If you want to check for a valid output device, then you should check the return values of subsequent **write()**'s or **fprint()**'s to that device.

11.4.2 Writing Characters to the Printer

After your application has opened the active printer device, your application can start writing to the printer.

The following **PrintChars()** routine accepts a string of characters and writes those characters to the printer.

```
/*****************************************************/

PrintChars(out_string)
char *out_string;
{
     write(pr_fd, out_string, strlen(out_string));
}
/*****************************************************/
```

Code Sample 11-3. Routine to Write a String to /dev/lp

For formated output and convenience, you can use **fprintf()** statements instead of **write()** statements to send characters to the printer.

Note that **write()** statements rely on the *file descriptor* for the device, whereas **fprintf()** statements rely on the *file pointer* for the device.

If you've already gotten a file descriptor for the printer device using an **open()** statement *and* if you want to start using **fprintf** statements, then you should include a **fdopen()** call to associate a file pointer (or *stream*) with the printer file descriptor.

For example:

```
/*******************************************************/

{
     FILE *fl;   /* fl is file pointer */

     PrinterOpen();    /* from an earlier code sample */

     fl = fdopen(pr_fd,"r+");     /* open for update */

     fprintf(fl,"Hello, printers everywhere!\n");
     fprintf(fl,"pr_fd is %d\n", pr_fd);

     fflush(fl);  /* to flush output */
}
/*******************************************************/
```

Code Sample 11-4. Using fdopen() To Return a File Pointer

The **fdopen()** and **fflush()** routines are part of the standard C runtime support library.

Note that the output from **fprintf()** statements is buffered; it needs to be flushed before you attempt any more **write()** statements to the same device (**write()** statements are unbuffered.)

Note also that doing formated output using **fprintf()** causes a sizable library to be linked into your application. To keep the size of your application down, you may wish to eschew the use of **fprintf()** statements.

11.4.3 Printers and Escape Sequences

The command language understood by all Hewlett-Packard printers consists of *escape sequences*--that is, strings of characters beginning with an ASCII ESC character[2] and followed by one or more parameter substrings. For example, the HP escape sequence to turn on underlining consists of four characters:

```
ESC & d D
```

The escape sequence to turn off underlining consists of the following characters:

2. Decimal code 27; octal code 33.

```
ESC & d @
```

Thus, to print an underlined message, your application would send the following sequence of bytes to the printer:

```
write(pr_fd, "\033&dD", 4); /* underlining ON */

write(pr_fd, message, strlen(message));

write(pr_fd, "\033&d@", 4); /* underlining Off */
```

Refer to the owner's documentation for the target printer to determine the escape sequence "vocabulary" of that printer.

Note that if a Hewlett-Packard printer does not understand a given escape sequence, the printer will simply ignore it and begin printing subsequent characters.

11.4.4 Redirecting Standard Output to a Printer

If your application is running in an alpha window, you may occasionally wish to redirect standard output from the alpha window to the printer device.

The following **OutputDevice()** routine enables standard output to be toggled between the alpha window, **/dev/tty**, and the active printer device, **/dev/lp**:

```
/*******************************************************/

#define PRINTER       1
#define ALPHA_WINDOW  0

OutputDevice(desired)
int desired;

{
   freopen((desired ? "/dev/lp" : "/dev/tty"),"w",stdout);
}
/*******************************************************/
```

Code Sample 11-5. Routine To Redirect Standard Output

Thus, by including the following statement:

```
OutputDevice(PRINTER);
```

--you cause standard output to go to the line printer.

The **freopen()** routine is part of the standard C runtime support library.

11.4.5 Print Spooling

Up to now, we've considered a single application sending its output to a printer. Because HP-UX is multi-tasking, it's common for two or more applications to be running at the same time.

However, if two or more applications are writing to **/dev/lp** at the same time, the output may become garbled. A way to avoid this problem is to let the built-in PAM shell oversee the work for you.

PAM supports *print spooling*--that is, the printing of files one by one in an orderly manner. A straightforward way to let PAM handle print spooling for your application involves these steps:

1. Write out to a temporary file all the lines of text that you want to print. (You can create this temporary file either in **/tmp** or on a physical disc.)
2. Perform a **fork()** to create a child process to print the file. (The parent process executes a **wait()** statement to wait until the printing is over.)
3. Have the child process **exec()** a copy a PAM to carry out the PAM **print** command. You can use the **execl()** system call to do the trick.

   For example:

   ```
   execl("/rom/PAM","PAM","-c","print",tempfile,";",0);
   ```

   --causes the system to execute a copy of PAM and to pass:

   ```
   print <tempfile> ;
   ```

   to the duplicate PAM as command line arguments.

   (The 0 at the end of the **execl()** parameter list is a "zero.")
4. After the printing is over, have the parent process purge the temporary file using **unlink()**.

11.4.6 Sending Raster Data to a Printer

There may be times when you wish to take advantage of the graphics capabilities of a printer. Doing so involves sending eight-bit character data to the printer in an almost random pattern of bits. (Refer to the owner's manual for the printer to determine its graphics capabilities and instruction formats.)

However, because the printer driver gives special meaning to newline characters, tab characters, etc., your raster output may be confused with printer control codes.

To avoid this problem, you can use the following **PrinterRaw()** routine to set the printer driver for your program to "raw mode."

Note

The following routines are specific to the Integral Personal Computer operating system and will not port to other HP-UX systems.

Be sure that you isolate these system-dependent routines in a separate module of your software.

```
/*******************************************************/

static struct set_ctl lp_save;

PrinterRaw()
{
     struct set_ctl lp_ctl;

     lp_save.pid = getpid();
     ioctl(pr_fd, LP_GETCTLBITS, &lp_save);

     lp_ctl.pid = getpid();
     ioctl(pr_fd, LP_GETCTLBITS, &lp_ctl);
     lp_ctl.control &= ~LP_COOKED;
     ioctl(pr_fd, LP_SETCTLBITS, &lp_ctl);
}
/*******************************************************/
```

Code Sample 11-6. Routine To Set the Printer Driver to Raw Mode

The **getpid()** (*get process id*) call is used by the printer driver to keep track of the printer device on a per application basis. That is, how your application sets the printer driver will *not* affect the status of other applications.

The printer driver for your application will remain in raw mode until you change it. That is, newline characters, tab characters, and other special characters will be shipped *as is* to the active printer device unless you restore the printer driver to its previous state.

The following **PrinterRestore()** routine does just that:

```
/*******************************************************/

PrinterRestore()
{
     ioctl(pr_fd, LP_SETCTLBITS, &lp_save);
}
/*******************************************************/
```

Code Sample 11-7. Routine to Restore the Printer Driver to Cooked Mode

Since we've saved the original settings (or *control bits*) in the `lp_save` structure, it's an easy matter to restore them.

11.4.7 Closing the Printer Device

When done with the printer, use the following **PrinterClose()** routine to clean things up.

```
/*******************************************************/

PrinterClose()   /* Close the printer */
{
     close(pr_fd);
}
/*******************************************************/
```

Code Sample 11-8. Routine to Close the Printer (/dev/lp)

If you've been using buffered output (with **fprintf()** statements), then you can use the **fclose()** statement instead.

11.5 Controlling a Plotter Device

Controlling a plotter device is much like controlling a printer device:

1. You *open* the plotter device driver, namely, **/dev/plotter**.
2. You *write* to the plotter device by specifying the plotter file descriptor in a **write()** statement.

3. When you're finished with the plotter, you *close* the plotter device.

11.5.1 Initializing the Plotter

Following are the relevant declarations for the plotter code samples in this chapter:

```
/*****************************************************/

#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/plotter.h>
#include <sys/ioctl.h>

static int pl_fd; /* global plotter file descriptor */

/*****************************************************/
```

Code Sample 11-9. Plotter Declarations

The following **PlotterOpen()** routine returns the file descriptor of the currently active printer in variable **pl_fd**.

```
/*****************************************************/

PlotterOpen()  /* Open the plotter driver */
{
     pl_fd = open("/dev/plotter", O_RDWR);
}
/*****************************************************/
```

Code Sample 11-10. Routine To Open the Plotter

Note that you should assume that your end user has taken responsibility for setting up the active plotter device, **/dev/plotter**.

The **plotter_is** utility is shipped with the computer for this express purpose. Refer to the owner's documentation for the computer for a description of the **plotter_is** utility.

Note that the **open()** of **/dev/plotter** will always succeed, even if there is no plotter connected to the computer. If you want to check for a valid output device, then you should check the return values of subsequent **write()**'s or

fprint()'s to that device.

11.5.2 Writing Strings to the Plotter

The following **PlotString()** routine accepts a string of characters and writes those characters to the active plotter device.

```
/*******************************************************/

PlotString(out_string)
char *out_string;
{
     write(pl_fd, out_string, strlen(out_string));
}
/*******************************************************/
```

Code Sample 11-11. Routine To Write a String to /dev/plotter

Note the similarity of this **PlotString()** routine to the **PrintChars** routine earlier in this chapter. The primary difference is the *file descriptor*--one specifies the **/dev/plotter** device for your application and the other specifies the **/dev/lp** device for your application.

For convenience, you can use **fprintf()** statements instead of **write()** statements to send characters to the plotter. Before doing so, you should first include a **fdopen()** call to associate a file pointer (or *stream*) with the plotter file descriptor. For more information, refer to the description of the **fdopen()** statement earlier in this chapter.

11.5.3 Plotters and HP-GL Command Strings

The command language understood by all Hewlett-Packard plotters consists of HP-GL *command strings*--that is, simple strings of ASCII characters that are interpreted by, and direct the action of, external plotters.

For example, the string "PD" is recognized by all HP-GL plotters as a *Pen Down* command--the plotter will lower the pen to the plotter bed.

The following **Triangles** program uses some of the plotter declarations of this chapter and the **PlotterOpen()** and the **PlotString()** routines to plot a simple picture of triangles with HP-GL character strings. The assumption is that the plotter is properly connected addressed before this program is executed.

```
/******************************************************/

/* refer to preceding plotter declarations and functions*/

main()
{
     Plotter_Open();  /* an earlier code sample */
     init_plotter();
     set_scale();
     prep_pen();
     draw_triangle_1();
     draw_triangle_2();
     init_text();
     plot_text("LB Over and Out^G");
     Plotter_Close(); /* see following code sample */
}
/*** Begin Subroutines ***/

init_plotter() /* initialize and "eject page" */
    {
    PlotString("IN; PG"); /* an earlier code sample  */
    }
set_scale() /* scale the plotter bed */
    {
    PlotString("SC 750,9000,750,9000");
    }
prep_pen() /* pen up, move to start, pen down */
    {
    PlotString("PU; PA 1000,1000; PD");
    }
draw_triangle_1() /* plot absolute x,y coordinates */
    {
    PlotString("PA 9000,1000,5000,7000,1000,1000");
    }
draw_triangle_2() /* go elsewhere, change line type,
    {                ** and plot
                    */
    PlotString("PU; PA 3000,2000; LT 4");
    PlotString("PD 7000,2000,5000,5000,3000,2000");
    }
init_text() /* set up size, slant, and end-text mark */
```

```
        {
        PlotString("PU; PA 1750,2500;");
        PlotString("SI 0.75,2; SL .5; DT^G");
        }
    plot_text(label); /* write that text */
    char *label;
        {
        PlotString(label);
        }
    /*****************************************************/
```

Code Sample 11-12. Routines To Draw an HP-GL Picture

For complete explanations of what these HP-GL strings actually *do*, refer to the *HP-GL Reference Manual.*

For example, the **^G** in the text label represents the ASCII BEL character (decimal code 7). We use the HP-GL **DT** command to define the BEL character as the delimiter for text labels.

In a well-structured application, you would rewrite and extend the above subroutines to handle parameter passing and to offer greater functionality. For example, you might write a general purpose **move_to_xy()** routine, a **draw_to_xy()** routine, and a **set_pen_number()** routine.

Remember, finally, that these same HP-GL strings can be redirected to a *graphics window* to draw the same picture as on an external plotter bed.

11.5.4 Closing the Plotter Device

When done with the plotter, use the following **PlotterClose()** routine to clean things up.

```
    /*****************************************************/

    PlotterClose()   /* Close the plotter */
    {
         close(pl_fd);
    }
    /*****************************************************/
```

Code Sample 11-13. Routine To Close the Plotter (/dev/plotter)

If you've been using buffered output (with **fprintf()** statements), then use **fclose()** instead.

11.6 Further Information

To learn more about the friendly **printer_is** and **plotter_is** utilities shipped with the computer, refer the owner's documentation for the computer.

To learn more about low-level printer driver system calls, refer to the **LP** file on the C language documentation disc.

To learn more about low-level plotter driver system calls, refer to the **PLOTTER** file on the C language documentation disc.

Chapter 12
The Speaker

12.1 Concepts

The computer features a built-in speaker and speaker chip. You can control both the *pitch* and the *duration* of tones from the speaker reasonably well from 100 Hz to 5 KHz, in resolution of 1/100 seconds. (It's not possible to vary the loudness of the speaker.)

To access the speaker from your programs, regard the speaker as a normal HP-UX device (installed as **/dev/beeper**). That is, you *open* the speaker and use the *file descriptor* returned by the **open()** call to control the speaker with low-level **ioctl()** calls. When you've finished with the speaker, you *close* the speaker device.

The code samples in this chapter make it easy to control the speaker.

Note

The following routines are specific to the Integral Personal Computer operating system and will not port to other HP-UX systems.

Be sure that you isolate these system-dependent routines in a separate module of your software.

A limitation of the following code samples is that they do no error or bounds checking; however, "well-behaved" beeping shouldn't cause any problems.

Note that the speaker is a system intensive resource. If your application is making a lot of noise, then other applications running at the same time will have a harder time getting their work done.

12.2 Global Information

Following are the relevant header file names and the structures used throughout this chapter:

```
/*****************************************************/

#include <fcntl.h>
#include <sys/beep.h>
#include <sys/ioctl.h>

#define NULL 0

static int sp_fd = -1;
/* speaker file descriptor */

static struct freqdur speakdata;
/* used to set frequency and duration */

static struct freqdur speakstat;
/* used to check the status of the speaker. */

/*****************************************************/
```

Code Sample 12-1. Declarations for the Speaker

The `freqdur` structure contains two elements: the *frequency* (in Hertz) and the *duration* of a tone (in hundredths of a second).

12.3 Initializing the Speaker

The following **SpeakerOpen()** routine returns the *file descriptor* of the speaker in variable **sp_fd** and then initializes the speaker.

```
/******************************************************/

SpeakerOpen()
/* Open and initialize the speaker */
{
    sp_fd = open("/dev/beeper",O_RDWR);
    ioctl (sp_fd, ALL_INIT, NULL);
}
/******************************************************/
```

Code Sample 12-2. Routine To Initialize the Speaker

From now on, we'll use the **sp_fd** file descriptor to control the speaker.

12.4 Using the Speaker to Beep Immediately

The following **BeepImmediate()** routine uses frequency and duration parameters to sound tones.

```
/******************************************************/

BeepImmediate(freq,dur)
int freq,dur;
/* Beep a note now! Terminate any existing beep */
{
    /* copy parameters to speaker structure */
    speakdata.frequency = (unsigned short) freq;
    speakdata.duration  = (unsigned short) dur;

    ioctl (sp_fd, BEEP_NOW, &speakdata);
}
/******************************************************/
```

Code Sample 12-3. Routine to Beep Immediately

Note that this routine will start a note immediately, and "step on the toes" of a preceding note, even if that note has not completed. For this reason, you may choose not to use this code sample.

12.5 A Second Way to Sound Tones

The following **WaitAndBeep()** routine also uses frequency and duration parameters to sound tones.

```
/****************************************************/

WaitAndBeep(freq,dur)
/* wait until speaker is idle and beep a note */
int freq,dur;
{
    /* copy parameters to speaker structure */
    speakdata.frequency = (unsigned short) freq;
    speakdata.duration  = (unsigned short) dur;

    /* wait for previous beeps to finish */
     do {
          ioctl (sp_fd, BEEP_READ, &speakstat);
     }
     while (speakstat.duration > 0);

     /* now do it! */
     ioctl (sp_fd, BEEP, &speakdata);
}
/****************************************************/
```

Code Sample 12-4. Routine to Wait and Then Beep

The difference is that this routine loops indefinitely, waiting for the previous sound to finish. This is the normal way of doing things--wait for the previous note to finish before starting the next.

12.6 Closing the Speaker

Finally, when done with the speaker, you want to clean things up. Use the following **SpeakerClose()** routine.

```
/*****************************************************/

SpeakerClose()     /* close the speaker */
{
    close(sp_fd);
}
/*****************************************************/
```

Code Sample 12-5. Routine to Close the Speaker

That's it! Simple, eh?

Index