Device I/O and User Interfacing HP-UX Concepts and Tutorials

HP 9000 Series 300/800 Computers

HP Part Number 97089-90057

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

Legal Notices

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

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

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

Printing History

This manual printing date and part number indicate its current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. The manual part number will change when extensive changes are made.

September 1989 ... Edition 1. This edition supersedes manual part number 97089-90054. The current edition contains all information previously contained in the Device I/O and the Curses and Terminfo tutorials. In addition, the manual reflects the following changes:

- Examples in the Device I/O tutorial have been updated to reflect changes for the 7.0 release.
- The Curses and Terminfo tutorial now includes 16-bit character support.
- The NLS tutorial has been removed and placed in a separate manual, *HP-UX Native Language Support* (97089-90058).

Contents

1

Interfacing Concepts

This tutorial explains how to access arbitrary I/O devices from HP-UX through **HP-IB** (Hewlett-Packard Interface Bus) and **GPIO** (General-Purpose I/O) interfaces by using subroutines contained in the HP-UX Device I/O Library (**DIL**). Topics discussed include general I/O programming strategies, as well as strategies related specifically to HP-IB and GPIO interfaces.

It is assumed that communication with I/O devices is handled through calls to DIL subroutines from C, Pascal, or FORTRAN programs. Examples shown in this tutorial are written in C, but the techniques illustrated are easily converted for use with Pascal or FORTRAN by adding a little extra code.

Variation Between Computer Systems

In general, DIL subroutines function identically on all HP-UX computers, regardless of series or model number within a series. However, because of certain inherent differences between processors and other hardware, some differences do exist. If such differences arise during an explanation, they are clearly identified.

Additional major differences related to a specific model or series are identified in a separate appendix for that series. Separate appendices are provided for Series 300 and 800.

Manual Organization

Chapter 1: Interfacing Concepts presents basic I/O programming concepts and a description of the HP-IB and GPIO interfaces.

Chapter 2: General-Purpose Routines discusses how to access interfaces from HP-UX environment and how to implement I/O transfers.

Chapter 3: Controlling the HP-IB Interface describes I/O programming techniques for the HP-IB interface.

Chapter 4: Controlling the GPIO Interface discusses I/O programming techniques for the GPIO interface.

Appendix A: Series 300 Dependencies discusses hardware- and system-dependent characteristics of DIL subroutines when used with Series 300 computers. If you are using a Series 300 HP-UX system, check this appendix to ensure correct use of DIL subroutines.

Appendix B: Series 800 Dependencies is similar to other appendices, but for Series 800 computers. Use this appendix to ensure the correct use of DIL subroutines on Series 800 systems.

Appendix C: Character Codes

Appendix D: DIL Programming Example shows a non-trivial example of an Amigo-protocol HP-IB device driver suitable for driving HP-IB line printers that support Amigo protocol (commonly used on certain HP-IB disc drives and line printers). This example program shows good HP-UX programming practice, and illustrates a number of other techniques and features such as parsing a command with arguments.

DIL Interfacing Subroutines

As mentioned previously, Device I/O Library (DIL) subroutines provide a means for directly accessing peripheral devices through HP-IB and/or GPIO interfaces connected to your computer system. Some routines are general-purpose and can be used with any interface supported by the library, while others provide control of only certain specific HP-IB or GPIO interfaces.

Linking DIL Routines

DIL routines can be called from C, Pascal, or FORTRAN programs. However, the **-l** flag must be given when invoking the C, Pascal, or FORTRAN compiler, **cc** (1), **pc** (1), or **fc** (1). Otherwise, library subroutines are not automatically linked with your program. To link DIL subroutines to a compiled C program, invoke the C compiler as follows:

```
cc program.c -ldvio
```

Similarly, for a Pascal program, use:

```
pc program.p  -ldvio
```

and for a FORTRAN program, use:

```
fc program.f -ldvio
```

In all three cases, the **-l** option is passed to the HP-UX linker, causing it to link any DIL routines called by the program being compiled. To determine the exact location of DIL library on your HP-UX system, refer to the corresponding hardware-specific appendix in this tutorial.

Calling DIL Routines from Pascal

You must provide an **external declaration** for each DIL subroutine called from a Pascal program. An external declaration consists of the subroutine heading, including a formal parameter list and result type, followed by the Pascal `EXTERNAL` directive. For example, the C description of *open*(2) is:

```
int  open(path, oflag)
char *path;
int oflag;
```

The equivalent external declaration for the same subroutine in a Pascal program is:

```
TYPE
   PATHNAME = PACKED ARRAY [0..50] OF CHAR;

FUNCTION open
  (VAR path: PATHNAME;
   oflag: INTEGER):
   INTEGER;
   EXTERNAL;
```

Note that the *path* parameter is a VAR parameter, indicating that the parameter is passed by reference. This simulates the passing of a pointer, which is what *open*(2) expects. In general, declaring a C routine from Pascal is straightforward.

Calling DIL Routines from FORTRAN

C and FORTRAN subroutine calls are not compatible because C passes parameters *by value* while FORTRAN passes them *by reference*. This incompatibility can be easily circumvented by directing the compiler to generate a call by value through the use of FORTRAN's $ALIAS option. For example:

```
$ALIAS close = 'close' (%val)
```

If the FORTRAN compiler on your system does not support this form of $ALIAS, the parameter-passing differences can be resolved by writing an **onionskin** routine which is a C-language function written for the purpose of resolving parameter-passing irregularities between C and other languages.

For example, to access *close*(2) through an onionskin routine, use:

```
$ALIAS close = '_my_io_close'
```

then write the onionskin routine:

```
int my_io_close (eid)
/* the compiler will create the external symbol "_my_io_close"
   based on the above declaration*/
int *eid;
{
    return (close (*eid));
}
```

General Interface Concepts

The remainder of this chapter discusses interfaces in general and the HP-IB and GPIO interfaces in particular. This background information is helpful for understanding system operation, but is not prerequisite to being able to successfully use DIL routines.

Definition

An interface is a built-in or plug-in electronic subassembly that manages the transfer of information between the computer and one or more peripheral devices. It converts electrical signals from the computer to a form that is compatible with the requirements of the peripheral device and converts signals from the peripheral device to a form that can be used by the computer. The interface also controls information transfer paths and transfer timing such that data flows in an orderly manner in correct sequence.

HP 9000 computers are equipped with both built-in as well as plug-in interfaces that can be purchased as standard or optional items. Separate interface cabling connects the peripheral device(s) to the interface unless the peripheral device is built into the computer housing. The following functional block diagram illustrates the functional architecture of a typical interface:

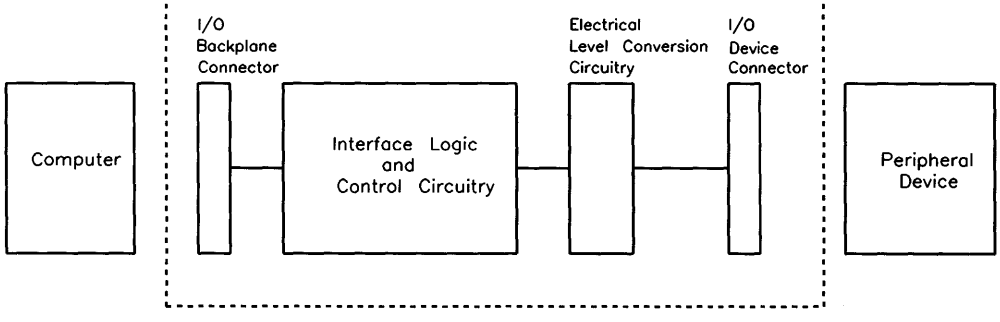


Figure 1-1. Interface Functional Diagram

Interface Functions

A usable interface must fill the following system requirements:

- **Electrical Compatibility:** The interface must convert electrical signal voltages, currents, frequencies, and timing from the computer to a form that is useful to the peripheral device, and vice-versa (unless no conversions are necessary). It must also provide any special protection that might be necessary to protect circuitry within the computer or peripheral from damage due to external effects related to the interface cable or power source.
- **Mechanical Compatibility:** The interface must be mechanically structured so that it is readily connected to both the computer and the peripheral device. This is usually accomplished by means of an interface cable that has appropriate connectors on each end.
- **Data Compatibility.** Just as two people must speak a common language before they can communicate well, the computer and peripheral must use compatible forms of communication. While in most cases, the computer operating system and the programmer are responsible for general data format, communication protocols such as those used in data communication networks and HP-IB interconnections are usually managed by the interface card, based upon various signals and commands from the computer and the peripheral device.
- **Timing Compatibility.** Peripheral devices within a given system rarely have identical data transfer rates and data transfer timing requirements. They also rarely match the timing and transfer rates in the computer or other devices in the system. For this reason, one of the most important functions of the interface is to manage and coordinate the interaction between the computer and the interface as well as timing between the interface and peripheral devices by using special timing signals that are inserted into the data being transferred (most common in data communication interfaces) or carried on separate control signal lines (typical for HP-IB and GPIO interfaces). These timing signals are used to coordinate when a transfer begins and at what rate the information is handled.
- **Processor Overhead Reduction:** Another important function of the interface card is to relieve the computer of low-level tasks, such as

performing data transfer handshakes. This distribution of tasks eases some of the computer's burden and decreases the otherwise stringent response-time requirements of external devices. The actual tasks performed by each type of interface card vary widely. The remainder of this chapter concentrates on the functions of two particular interfaces: HP-IB and GPIO.

Handshake I/O

Most HP-IB and GPIO interfaces operate by means of **handshake** transfers which operate generally as follows:

Handshake Output

- Computer sets input/output control to output and places first word or byte on I/O bus to interface.
- Computer asserts peripheral control line to interface to start transfer.
- Interface recognizes asserted control signal from computer and transfers data to output drivers and interface cable.
- Interface asserts output timing signals to peripheral device and waits for response.
- Peripheral accepts output timing signals, inputs data from interface cable, then returns flag signal indicating data has been accepted.
- Interface recognizes flag and sets flag to computer indicating the transaction is complete. If the sender and receiver do not agree upon start time and transfer rate, then the transfer is carried out via a **handshake** process: the transfer proceeds one data item at a time with the receiving device acknowledging that it received the data and that the sender can transfer the next data item. Both types of transfers are utilized with different interfaces.

Handshake Input

- Computer sets input/output control to input.
- Computer asserts peripheral control line to interface to start transfer.

- Interface recognizes asserted control signal from computer, sends data input command sequence to peripheral device, and waits for response.
- Peripheral accepts input command sequence, places data on interface cable, then returns flag signal indicating data is available.
- Interface recognizes flag, moves data to computer I/O bus, and sets flag to computer indicating the transaction is complete.

Different interfaces support variations on this basic sequence. For example, more sophisticated data communication and HP-IB cards may be equipped with a microprocessor and shared memory that is directly accessible to the computer and the interface processor. The computer moves data to and from shared memory according to program needs, while the interface processor performs similar operations to meet the demands of any data transfers in progress. Shared pointers and other flags prevent collisions between conflicting demands from the two processors, and the increased efficiency of a "smart" interface greatly reduces the complexity and overhead related to more mundane approaches to interrupt-driven handshake I/O.

For example, instead of handling each character or word as a single transaction, the computer can load a block of data into the shared memory then signal the interface that data is ready for transfer. The interface then uses the shared pointers or other means to determine how much data to transfer, handles the transfer, then signals the computer that the task is complete.

HP-IB Protocol

When a single interface is shared by multiple peripheral devices, additional signalling must be used to control which devices respond to each transaction as in HP-IB interfacing. A selection of protocol signals and device commands are used to activate or deactivate various devices on the HP-IB bus according to the needs of the bus controller (controlling interface). This signals, their functions, and the sequences in which they are used are discussed in greater detail throughout this tutorial.

The HP-IB Interface

The Hewlett-Packard Interface Bus (HP-IB) was developed at HP as the solution to an expanding need for a universal interfacing technique that could be readily adapted to a wide variety of electronic instruments. It was later expanded to include high-speed disc drives and other high-performance computer peripherals. The HP-IB architecture was subsequently proposed to and accepted by the Institute of Electrical and Electronic Engineers (IEEE) and is now widely used throughout the electronic industry. HP-IB is compatible with IEEE standard 488-1978. The number of devices that can be connected to a given HP-IB interface depends on the loading factor of each device, but in general up to 15 devices (including the interface) can be connected together while still maintaining electrical, mechanical, and timing compatibility requirements on the bus.

General Structure

IEEE Standard 488-1978 defines a set of communication rules called "bus protocol" that governs data and control operations on the bus. The defined protocol is necessary in order to ensure orderly information traffic over the bus.

Each device (peripheral or computer interface) that is connected to the HP-IB can function in one or more of the following roles:

System Controller — Master controller of the HP-IB. The computer interface is usually the bus controller when all peripheral devices on the bus are slaves to the system computer. However, any other device can become the active controller if it is equipped to act as a controller and control is passed to it by the System Controller. The System Controller is always the active bus controller at power-up.

Active Controller — Current controller of the HP-IB. At power-up or whenever IFC (InterFace Clear) is asserted by the System Controller, the System Controller is the active controller. Under certain conditions, the System controller may pass control to another device that is capable of managing the bus in which case that device becomes the new active controller. The active controller can then pass control to

	another controller or back to the System Controller. If the System Controller asserts IFC, the active controller immediately relinquishes control of the bus.
Talker	A device that has been authorized by the current active controller to place data on the bus. Only one talker can be authorized at a time.
Listener	Any device that has been programmed by the active controller to accept data from the bus. Any number of devices on the bus can be programmed by the active controller to listen simultaneously at any given time.

In typical systems, an HP-IB interface in the computer can act as a **controller, talker,** and **listener.** If more than one computer is connected to the same bus, only one interface can be configured as System Controller to prevent conflicts at power-up (this is usually accomplished by a switch or wire jumper on the interface card). A device that can only accept data from the bus (such as a line printer) usually operates as a **listener,** while a device that can only supply data to the bus (such as a voltmeter) usually operates as a **talker.** However, before any device can talk or listen (after power-up initialization), it must be authorized to do so by the current active controller. Bus configuration varies, depending on the type of activity that is prevalent at the time. However, in any case, the bus can have only one Active Controller and only one talker at a given time, though it can have any number of listeners.

HP-IB is composed of 16 lines (plus ground) that are divided into 3 groups:

- Eight data lines form a bi-directional data path to carry data, commands, and device addresses.
- Three handshake lines control the transfer of data bytes.
- The five remaining lines control bus management.

Handshake Lines

The **handshake** lines used to synchronize data transfers are:

$\overline{\text{DAV}}$	DAta Valid: Valid data has been placed on bus by talker.
$\overline{\text{NRFD}}$	Not Ready For Data: One or more listeners not yet ready to accept data from the bus.

$\overline{\text{NDAC}}$ Not Data ACcepted: One or more listeners has not yet accepted the data currently on the bus.

Note

The HP-IB interface uses negative (ground-true) logic for handshake, data, and bus management lines. This means that when the voltage on a line is at a logic LOW level, the line is *asserted* (true). When a logic HIGH voltage level is present on the line, the line is *not asserted* (false).

In general, software documentation refers to handshake and other lines by their name acronym such as DAV, NRFD, NDAC, etc. When discussing these same signal lines in hardware documents, it is customary to refer to ground-true (low-true) logic lines by their name acronym with a bar across the top such as $\overline{\text{DAV}}$, $\overline{\text{NRFD}}$, $\overline{\text{NDAC}}$, etc. In this document, both versions are used. The overbar is usually present when discussing hardware operation, but usually absent when software is being treated. In this tutorial, only the name is significant. Signal names are synonymous, with or without the overbar unless specifically noted otherwise; the overbar is used for the convenience of those readers whose experience is oriented more toward hardware than software.

The timing diagram in Figure 1-2 shows how handshake lines are used to complete a data item transfer. The discussion which follows is based on the contents of Figure 1-2.

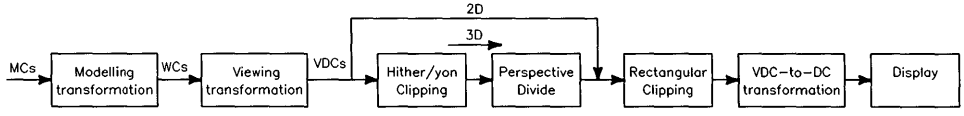


Figure 1-2. HP-IB Handshake Sequence

All handshake lines are electrically connected in a "wired-OR" configuration which means that any device can pull the line low (active or asserted) at any time, and more than one device may pull the line low simultaneously or later in a given handshake cycle. The line then remains low until every device that was previously pulling the line low has released the line, allowing it to float to its high state. At the start of the handshake cycle (point A), the handshake lines are in the following states:

- $\overline{\text{DAV}}$ is false (high), meaning that the current talker has not yet placed valid data on the bus.
- $\overline{\text{NRFD}}$ is true (low), meaning that one or more listeners is not yet ready to accept data from the bus.
- $\overline{\text{NDAC}}$ is true (low), meaning that bus data has not yet been accepted by every listener on the bus.

When a listener is ready to accept data, it releases $\overline{\text{NRFD}}$, allowing it to go high provided no other listener is still holding the line low. However (due to the "wired-OR" interconnection scheme used by HP-IB), $\overline{\text{NRFD}}$ remains LOW (true) until *every* listener releases it. When every listener is ready to accept data (indicated by $\overline{\text{NRFD}}$ being released by every listener), $\overline{\text{NRFD}}$ changes to its logic HIGH (false) state as indicated by point B in Figure 1-2.

By monitoring $\overline{\text{NRFD}}$, the talker can determine when to send data: $\overline{\text{NRFD}}$ false means that every listener is ready to accept data. The talker then places data on the data lines and asserts $\overline{\text{DAV}}$ (point C), indicating to the listeners that valid data is available on the data lines for them to accept.

As soon as each listener detects that $\overline{\text{DAV}}$ has been asserted, it asserts $\overline{\text{NRFD}}$ (point D), driving it low (true) unless $\overline{\text{NRFD}}$ has already been driven low by another listener in the same cycle.

After driving $\overline{\text{NRFD}}$ low, each listener inputs and processes the data from the data lines. When it has accepted the data, the listener releases $\overline{\text{NDAC}}$. As with the $\overline{\text{NRFD}}$ line at point B, $\overline{\text{NDAC}}$ remains low (true) until every listener on the bus has released the line, allowing it to go high (false). When $\overline{\text{NDAC}}$ goes high, the false logic state indicates to the talker that every listener has accepted the data (point E).

When the talker determines that every listener has accepted the data, it releases the $\overline{\text{DAV}}$ line which rises to its high (false) state. At the same time, the talker disables its outputs to the data lines, allowing them to rise to their high (false) state (point F).

When $\overline{\text{DAV}}$ goes false, the listeners assert $\overline{\text{NDAC}}$ (point G), driving it low. This signifies the end of the handshake (point H), at which time all bus logic lines are again at the same state as they were before the handshake started (point A).

Bus Management Control Lines

There are five bus management control lines:

$\overline{\text{ATN}}$ ATtentioN: Treat data on data lines as commands, not data.

$\overline{\text{IFC}}$ InterFace Clear: Unconditionally terminate all current bus activity.

$\overline{\text{REN}}$ Remote ENable: Place all current listeners in Remote operating mode.

$\overline{\text{EOI}}$ End Or Identify: End of data message. If $\overline{\text{ATN}}$ is true (low), Active Controller is conducting a parallel poll (Identify) of devices on the bus.

$\overline{\text{SRQ}}$ Service ReQuest: Bus device is requesting service from current Active Controller.

$\overline{\text{ATN}}$: The Attention Line

Command messages are encoded on the data lines as 7-bit ASCII characters, and are distinguished from the normal data characters by the attention ($\overline{\text{ATN}}$) line's logic state. That is, when $\overline{\text{ATN}}$ is false, the states of the data lines are interpreted as data. When $\overline{\text{ATN}}$ is true, the data lines are interpreted as commands.

$\overline{\text{IFC}}$: The Interface Clear Line

Only the System Controller sets the $\overline{\text{IFC}}$ line true. By asserting $\overline{\text{IFC}}$, all bus activity is unconditionally terminated, the System Controller becomes the Active Controller, and any current talker and all listeners become unaddressed. Normally, this line is used to terminate all current operations, or to allow the System Controller to regain control of the bus. It overrides any other activity currently taking place on the bus.

$\overline{\text{REN}}$: The Remote Enable Line

This line allows instruments on the bus to be programmed remotely by the Active Controller. Any device addressed to listen while $\overline{\text{REN}}$ is true is placed in its remote mode of operation.

$\overline{\text{EOI}}$: The End or Identify Line

If $\overline{\text{ATN}}$ is false, $\overline{\text{EOI}}$ is used by the current talker to indicate the end of a data message. Normally, data messages sent over the HP-IB are sent using strings of standard ASCII code terminated by the ASCII line-feed character. However, certain devices must handle blocks of information containing data bytes within the data message that are identical to the line-feed character bit pattern, thus

making it inappropriate to use a line-feed as the terminating character. For this reason, $\overline{\text{EOI}}$ is used to mark the end of the data message.

The Active Controller can use $\overline{\text{EOI}}$ with $\overline{\text{ATN}}$ true to conduct a parallel poll on the bus.

$\overline{\text{SRQ}}$: The Service Request Line

The Active Controller is always in charge of overall bus activity, performing such tasks as determining which devices are talkers and listeners, and so forth. If a device on the bus needs assistance from the Active Controller, it asserts $\overline{\text{SRQ}}$, driving the line low (true). $\overline{\text{SRQ}}$ is a request for service, not a demand, so the Active Controller has the option of choosing when and how the request is to be serviced. However, the device continues to assert $\overline{\text{SRQ}}$ until it has been satisfied (or until an interface clear command disables the request). Exactly what satisfies a service request depends on the requesting device, and is explained in the operating manual for the device.

The GPIO Interface

The **GPIO** (General Purpose Input/Output) interface is a very flexible parallel interface that can be used to communicate with a variety of devices. The GPIO interface utilizes data, handshake, and special-purpose lines to perform data transfers by means of various user-selectable handshaking methods.

While the GPIO interfaces used on various HP-UX computers are electrically very similar, they differ in certain important aspects. Refer to the appendices for Series 300 and 800 for information pertaining to your specific application.

2

General-Purpose Routines

The DIL library contains several general-purpose subroutines that can be used with any interface supported by the library (see Table 2-1 for a complete list). This chapter explains how to use these subroutines in application programs. Specifically, the following topics are presented:

- Basic introductory background concepts that are essential to understanding correct use of DIL library routines.
- Opening interface special files.
- Closing interface special files.
- Read/write operations to interface special files.
- Designing error-checking routines.
- Resetting an interface.
- Controlling input/output parameters.
- Determining why a read terminated.
- Handling interrupts.

Background Basics

Interface Special Files

HP-UX handles I/O to an interface or system peripheral device much like it handles read/write operations to disc storage files: every I/O interface or device is associated with an entity generally referred to as a `device file`, `special file`, or `device special file`. All three terms are used interchangeably and are usually synonymous. Any program that accesses subroutines in the DIL library cannot be used unless an appropriate device special file has been created for the corresponding interface. While the program can be written before the file exists, it cannot be used. The method used to create an interface special file depends on the model of computer being used. Refer to the appropriate hardware-specific appendix for information about creating interface special files on your system.

Entity Identifiers (eid)

Nearly all DIL routines require an **entity identifier** (*eid*) as a parameter. The entity identifier is an integer returned by the *open*(2) system call when opening the interface special file (*eid* is the file descriptor for the opened special file on Series 300 and 800). The *eid* supplied as a parameter to a DIL subroutine tells the subroutine which interface special file to use.

Programming Model

As a general rule, all programs that contain DIL subroutine calls for a specific interface should conform to the following structure:

1. Use an *open* system call to obtain the interface entity identifier (*eid*) for the special file being used. Opening an interface special file is discussed later in this chapter.
2. Use the returned *eid* as a parameter in DIL subroutine calls to perform desired tasks through the corresponding interface. Suitable techniques are discussed throughout the remainder of this tutorial.

3. When the necessary DIL subroutine calls have been completed, close the interface special file that was opened in step 1 above as discussed later in this chapter.

General-Purpose Routines

Table 2–1 provides a brief synopsis of the standard general-purpose routines discussed in this chapter. Several system calls related to the use of DIL subroutines, are also discussed: *open*(2), *close*(2), *read*(2), and *write*(2).

Table 2–1. General-Purpose Routines.

Routine	Description
`io_reset`	Reset a specified interface.
`io_timeout_ctl`	Establish a timeout period for any operation performed on a specified interface by a DIL routine.
`io_width_ctl`	Set the data path width for a specified interface.
`io_speed_ctl`	Select a data transfer speed for a specified interface.
`io_eol_ctl`	Set up a read termination character for data read from a specified interface.
`io_get_term_reason`	Determine how the last read terminated for the specified interface.
`io_on_interrupt`	Set up interrupt handling for a program.
`io_interrupt_ctl`	Enable or disable interrupts for a specified interface.
`io_lock`	Lock an interface for exclusive use by the calling process.
`io_unlock`	Unlock an interface so it can be used by other processes.

Additional Series 300 Routines

Series 300 systems also support the following DIL subroutines:

Subroutine	Description
`io_burst`	Control the data path between computer memory and an HP-IB or GPIO interface. If *flag* = 0, all data is handled through kernel calls with the normal associated overhead. If *flag* is non-zero, burst mode locks the interface and data is transferred directly between memory and the I/O mapped interface until the transfer is completed. Burst mode yields substantial improvement in efficiency when handling small amounts of data or high-speed data acquisition. This subroutine handles high-speed transfers on both HP-IB and GPIO I/O.
`io_dma_ctl`	Control usage of DMA channels by DIL devices.

Refer to the *io_burst*(3I) and *io_dma_ctl*(3I) entries in the *HP-UX Reference* for details on using these subroutines.

Opening Interface Special Files

With the exception of the default standard input, standard output, and standard error files, all read/write operations to any file from inside C, FORTRAN, or Pascal programs require that the file(s) be explicitly *open*ed before they can be used. The HP-UX *open*(2) system call is used to accomplish this as follows:

```
#include <fcntl.h>
int  eid;
:
eid = open(filename, oflag);
```

filename is either a character string containing the device file's external HP-UX name or a pointer to a buffer containing the external name.

The integer variable *oflag* specifies the access mode for the opened file, and can have one of six possible values, as defined in the `/usr/include/fcntl.h` header file: `O_RDONLY` (value = 0) requests read-only access, `O_WRONLY` (value = 1) requests write-only access, and `O_RDWR` (value = 2) requests both read and write access (three values with O_NDELAY not set, three values with O_NDELAY set – see `io_lock` (3I) in the *HP-UX Reference,* for a total of six values). To use these constants in a programs, the `#include` C-compiler directive must be present as shown in the example above.

An *open* system call on an interface special file returns an integer representing the entity identifier (*eid*) for the opened interface. As mentioned earlier, the entity identifier is required as a parameter in all DIL subroutine calls. It is also required as a parameter for all read/write operations to the opened file.

The following code defines an entity identifier called *eid* and opens an interface file called `/dev/raw_hpib` with access enabled for both reading and writing:

```
#include <fcntl.h>
#include <errno.h>
int  eid;
:
eid = open("/dev/raw_hpib", O_RDWR);
```

Special files can also be opened by placing the character string name of the file being opened in a string variable, then executing the *open* system call with a pointer to the variable as shown in the following code segment:

```
#include <fcntl.h>
int  eid;
char *buffer;
:
buffer = "/dev/raw_hpib";
if ((eid = open(buffer, O_RDWR)) == -1) {
    printf("open failed, errno = %d\n", errno);
    exit(2);
}
```

If the call to *open* succeeds, a non-negative integer is returned as the entity identifier. If an error occurs and the file is not opened, −1 is returned and *errno* is set to indicate the error.

Closing Interface Special Files

Good programming practice dictates that an open interface special file should be closed when a program is through using it by executing a *close*(2) system call. This guideline is valid even though any open files are automatically closed by the HP-UX operating system when a process terminates (via *exit*(2) or a return from the main routine).

Note HP-UX limits the number of files a given process (program) can have open at one time to `NO_FILE` as defined in the `/usr/include/param.h` header file. Series 300 systems limit the number of open DIL files in the entire system to the value of the configurable parameter *ndilbuffers* (default is 30). See the *HP-UX System Administrator Manual* for information on changing this value. Series 800 systems limit the number of open DIL files to 16 per interface.

The *close* system call requires the entity identifier corresponding to the open interface special file that is being closed. The following code segment shows how to open and close an HP-IB interface:

```
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid;
   .
   .
   .
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }

   .
   .
   .      /* Code to perform I/O operations
                  (read/write in this case) on the open interface. */
   close(eid);
}
```

Upon completion of the *close* system call, the entity identifier is no longer valid and is available for the system to assign to another file. If the file is again

opened later in the program, the system may or may not assign the same *eid* value, so appropriate caution in using *eid* values is in order.

close(2) returns a value of zero if the file is successfully closed. Otherwise, it returns a −1 and the external error variable *errno*(2) is set to indicate the error (error handling is discussed later in this chapter). The most common error returned by *close* (EINVAL) is related to an invalid value for *eid* meaning that the wrong value was used or the file is already closed.

Low-Level Read/Write Operations

Most HP-UX I/O operations to system peripheral devices is handled at a fairly high level where the system automatically provides buffering and other services that are not under the direct control of the user or program being run. However, some situations that are commonly encountered by DIL users require a much more intimate control of individual I/O transactions. These low-level operations provide no buffering or other services, and are a direct entry into the operating system. The two HP-UX system calls, *read*(2) and *write*(2), provide low-level I/O read/write capabilities. Both require three arguments:

- The entity identifier for an open file
- A buffer (string variable) in the program where data is to come from during *write* or go to during *read* (*write* empties a buffer; *read* fills a buffer).
- The number of bytes to be transferred.

Calls to `read` have the form:

```
#include <fcntl.h>
#include <errno.h>
main()
{
    int  eid;          /*the entity identifier*/
    char buffer[10];   /*buffer in which the read data will be placed*/

    if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
    }

    io_reset(eid);
    io_timeout_ctl(eid, 1000000);

    .
    .
    .  /*establish communication with the raw HP-IB device file
              as described in Chapter 3, "Controlling the HP-IB interface"*/

    read(eid, buffer, 10);  /*reads 10 bytes from a previously opened*/
}                           /*file with the entity identifier "eid". */
```

Calls to *write* are very similar:

```
#include <fcntl.h>
#include <errno.h>
main()
{
   int  eid;         /*the entity identifier*/
   char *buffer;     /* the buffer containing data to be written to a file*/
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }

   io_reset(eid);
   io_timeout_ctl(eid, 1000000);

   .
   .
   .  /*establish communication with the HP-IB interface as described
             in Chapter 3, "Controlling the HP-IB Interface"*/

   buffer = "data message";  /*message to be sent*/
   write(eid, buffer, 12);   /*12 bytes are written to previously*/
}                            /*opened file with the entity identifier "eid"*/
```

Although *read* and *write* require the number of bytes to be transferred as their third argument, other characteristics (discussed later) of the device associated with the interface file *eid* can end the transfer before this number is reached.

Example

Assume you have already created an auto-addressed special file, */dev/hpib_dev* for an HP-IB device. Your program must first open */dev/hpib_dev* for reading and writing:

```
int  eid;
eid = open("/dev/hpib_dev", O_RDWR);
```

To place data on the bus, use *write:*

```
write(eid, "This is a test", 14);
```

In this example, 14 characters are sent through *eid.* The literal string expression `This is a test` is placed in a data storage area by the compiler for later handling by the call to *write.* On output, if the number of characters requested does not match the length of the data storage space, the message is truncated (if the byte count is smaller than the data block) or extended into the next data block assigned by the compiler (if the byte count is larger than the data block).

To receive 10 bytes of data from the bus and place them in `buffer`, use:

```
char buffer[10];
read(eid, buffer, 10);
```

In this code segment, the `read` routine will attempt to read up to 10 bytes of data from the interface and place it in `buffer`.

Designing Error Checking Routines

All Device I/O Library routines return −1 and set an external HP-UX variable called *errno* if an error occurs during execution.

The errno Variable

errno is an integer variable whose value indicates what error caused the failure of a system or library routine call. It is not reset after successful routine calls, and should never be checked for value until after you have determined that an error has occurred.

Well-designed programs always include adequate error checking. However, most examples shown in this tutorial (other than in this section) do not verify successful completion of subroutine calls.

Refer to the *errno*(2) entry in the *HP-UX Reference* for complete definitions of the various errors returned when a system call fails.

Using errno

The following code segment must be present in the early part of any program that accesses *errno:*

```
#include <errno.h>
```

The errno.h Header File

The header file `/usr/include/errno.h` uses error numbers defined in header file `/usr/include/sys/errno.h`. For a complete list of errors and their associated meanings, refer to *errno*(2) in the *HP-UX Reference.*

Displaying errno

Once *errno* has been declared in a program, there are two ways to check its value if a routine fails. The simplest approach is to check the return value to determine whether or not the routine failed, then print out the value of *errno* and exit if it did. The following example illustrates this strategy:

```
#include <errno.h>
#include <fcntl.h>
main()
{
   int eid;
   .
   .
   .
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1)
   {
      printf("Error occurred. Errno = %d", errno);
      exit(1);
   }
   .
   .
   .
}
```

When this method is used, the program user must refer to the *errno*(2) entry in the *HP-UX Reference* to determine what the printed value of **errno** means.

Error Handlers

Another approach that is more complex for the programmer but much more convenient for the user is to check for specific values of **errno** and execute error routines related to the value. In most cases, only a limited number of situations can cause a particular a subroutine to fail, so there is a correspondingly small number of **errno** values that can be encountered upon failure. Possible error values are usually listed in the *HP-UX Reference* on the manual page entry for the failed subroutine.

For example, checking *open*(2) in the *HP-UX Reference* reveals that *errno* is set to `ENOENT` (defined in the `errno.h` header file) if you attempt to open a file that does not exist and you have not given the system call permission to create a new file. Armed with this information, you can incorporate the following code segment in your program:

```
#include <errno.h>
#include <fcntl.h>
main()
{
   int eid;
   .
   .
   .
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1)
   {
      if (errno == ENOENT)
         printf("Error: cannot open; file does not exist\n");
      else
         printf("Error: file exists but cannot open\n");
      exit(1);
   }
   .
   .
   .
}
```

Note that the print statements in the example above could be replaced with calls to more sophisticated error-handling routines such as `perror` (see the *perror*(3C) entry in the *HP-UX Reference*).

Resetting Interfaces

The DIL routine `io_reset` can be used to reset both HP-IB and GPIO interfaces.

The following example call to `io_reset` resets the interface whose entity identifier is *eid* where *eid* is the value that was returned when the interface special file was opened.

```
io_reset(eid);
```

Refer to the appropriate hardware-specific appendix for more information about the exact effects of `io_reset` on HP-IB and GPIO interfaces when used with various computer models.

For example, suppose that after opening an interface file you want to make sure the interface has been properly initialized. This is done by calling `io_reset` and looking at its return value:

```
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid;
   .
   .
   .
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }

   if (io_reset(eid) == -1)
   {
       printf("Possible problem with interface\n");
       exit(1);
   }
   .
   .
   .
   /* program continues if "io_reset" was successful */
}
```

Locking an Interface

Using a single interface to control multiple peripheral devices provides many advantages in convenience, cost and system operating characteristics. However, when several programs and/or several users need simultaneous access to peripherals sharing a single interface, conflicts arise. This problem is especially annoying when one user needs exclusive control of the interface during a set of critical I/O operations. Unless a mechanism is provided to lock out other users during critical program steps, useful results may be unobtainable in some cases.

Two DIL subroutines, `io_lock` and `io_unlock` are provided for this purpose. The first locks the interface so that only the process that locked it can use the interface until it is unlocked. The second unlocks the interface so other processes can again access it.

When another process attempts to access a locked interface, the process will sleep until the interface is unlocked (or a timeout occurs) if the *O_NDELAY* flag was not set at the time the requesting process executed the *open*(2) system call. If the *O_NDELAY* flag was set during the call to *open*(2) and the interface is locked, any attempts to access the locked interface fail and the DIL subroutine call from the process returns with an error.

Locks on an interface are owned by the process, and are not associated with the *eid*. This means that the same process can access a given interface through another *eid* if another *open* is performed on the device. If a process uses a *fork*(2) system call to create a child process that uses the same interface, the child does not inherit the current lock from the parent. Since it has a different process ID than the parent, it also cannot access the locked interface file until the parent unlocks it.

For good programming practice, any locks created by a process should be unlocked through a call to `io_unlock` before terminating. However, any locks held by a process are released when the process terminates, whether or not a call to `io_unlock` was executed. Refer to `io_lock`(3I) in the *HP-UX Reference* for more information about locking and unlocking interfaces.

Caution Do not place a lock on any interface that supports any system disc or swap device. Interface locks are enforced by the system, and such a condition may require rebooting in order to recover.

Controlling I/O Parameters

The Device I/O Library provides four subroutines that perform I/O control operations pertaining to timeout, data path width (usually 8 or 16 bits), transfer speed, and read termination (end-of-line) pattern. The subroutines and their functions are as follows:

Subroutine	Controlled I/O Function
`io_timeout_ctl`	Timeout: Assign a timeout value in microseconds for I/O operations (actual timeout resolution may be limited by system hardware).
`io_width_ctl`	Data Path Width: Specify width of the interface's data path or switch between supported widths for various operations.
`io_speed_ctl`	Transfer Speed: Request a minimum speed for data transfers through the interface in kilobytes (Kbytes) per second.
`io_eol_ctl`	Read Termination Pattern: Assign a pattern to be recognized as a read termination pattern.

Note

It is not uncommon for a single process to have multiple *eid*s open simultaneously (resulting from multiple calls to *open* in a single program. The subroutines `io_timeout_ctl`, `io_width_ctl` `io_speed_ctl`, and `io_eol_ctl`, can be used to conveniently configure different values for timeout, width, speed, and termination pattern on any given *eid* without disturbing the previously configured (or default) values associated with other *eid*s.

Unless specifically altered by calls to one or more of these subroutines, interface file operation uses system defaults for each *eid*.

Opening multiple *eid* s on a given interface file, then configuring each independently is an easy way to handle multiple devices that use different data formats without having to reconfigure each individual I/O operation.

Setting I/O Timeout

I/O timeout determines how long the system waits for a response from the interface or peripheral device each time an I/O operation is initiated. If the timeout limit is exceeded, the operation is aborted and a timeout error is returned. The default timeout is set to 0 which disables timeout errors.

If timeout is disabled (zero) and an error condition occurs that prevents successful completion of a data transfer or other I/O operation, the calling program may hang. Therefore, use of a non-zero timeout value is strongly recommended as good programming practice. To set or change the timeout use `io_timeout_ctl` as follows:

```
#include <fcntl.h>
#include <errno.h>
main()
{
   int  eid;
   long time;

   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);

   time = 1000000;     /*set timeout of 1 second*/
   io_timeout_ctl(eid, time);

   :              /*data transfers using "eid" are controlled by the
                         timeout value "time"*/
}
```

eid is the entity identifier associated with the open interface file, and *time* is a 32-bit long integer specifying the length of the timeout in microseconds.

Each time an I/O operation is initiated, timeout is restarted. For example, when setting up bus addressing, the system allows *timeout* microseconds for completion. Each subsequent data transfer (in or out) is given the same time limit. If a given operation is not completed within the time limit specified by the timeout value, the operation is aborted and an error indication is returned (return value of -1) and *errno* is set to *EIO* (not to be confused with EOI).

Note Be sure that the timeout limit is set to a value higher than the longest expected time to complete a transfer. If a normal transfer takes longer than the timeout limit, the operation is aborted even though system operation is correct.

Timeout is specified in microseconds (μsec) in the call to `io_timeout_ctl`, but the actual timeout used and its resolution is system-dependent. The timeout value is always rounded *up* to the nearest normal time resolution interval supported by the system executing the operation. For example, if the available system resolution is 10 milliseconds and a timeout of 25000 microseconds (25 milliseconds) is requested, the actual timeout value used is 30 milliseconds. To determine timeout resolution for your system, refer to the appropriate hardware-specific appendix.

IMPORTANT A timeout value of 0 microseconds is meaningless because no device can respond with data in less than zero time. For this reason, the default or a specified timeout value of zero is treated as a request to disable timeout and any condition that would normally cause a timeout termination is ignored by the system, usually causing the program to hang. *Specifying a timeout of zero is not recommended.*

Any interface file *eid* obtained by using the *dup*(2) system call or inherited by a *fork*(2) request shares the same timeout as the original interface file *eid* obtained from *open*(2). If the child process resulting from a *fork* inherits an *eid* then changes the timeout, the *eid* used by the parent process is likewise affected.

Setting Data Path Width

When you create a DIL special file and open it for the first time, the data path width defaults to 8 bits. Once the file is opened, `io_width_ctl` can be used to select a new width. *Allowable widths vary, depending on the computer model and interface.* Refer to the appropriate hardware-specific appendix to determine what widths are supported by specific interfaces.

Assuming that the open device file has the entity identifier *eid*, `io_width_ctl` is called using a code segment similar to the following:

```
int  eid, width;
:
io_width_ctl(eid, width);
```

where **width** is the number of parallel bits in the new data path. The `io_width_ctl` returns −1 to indicate an error if the specified width is not supported on the interface identified by *eid*.

For example, to reconfigure a GPIO device to use all 16 data lines in the interface cable instead of the default lower 8 bits, use a a code segment similar to the following:

```
#include <fcntl.h>
#include <errno.h>
main()
{
    int  eid, width;
    width = 16;                /*width of new data path */

    if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
         printf("open failed, errno = %d\n", errno);
         exit(2);
    }
    io_width_ctl(eid, width); /*assign new width for GPIO bus*/

    :
    :   /*data transfers using "/dev/raw_gpio" will now
              use a 16-bit bus*/
}
```

Use of `io_width_ctl` to change interface data path width affects only the device associated with that particular *eid*. Use `io_reset` or `io_width_ctl` to restore the default 8-bit path width. On a Series 800 system, `io_width_ctl` affects all users of the interface referred to by *eid*.

Setting Minimum Data Transfer Rate

DIL provides a means for specifying a minimum acceptable data transfer rate for a given interface special file within the limits of available hardware by use of `io_speed_ctl`. The calling sequence is as follows:

```
io_speed_ctl(eid, speed);
```

where *eid* is the entity identifier for the open interface file, *speed* is an integer indicating a minimum speed in Kbytes per second, and a kilobyte equals 1 024 bytes.

`Io_speed_ctl` returns a 0 if successful, or −1 if an error occurred. For example:

```
io_speed_ctl(eid, 1);
```

requests a minimum speed of 1 024 bytes per second. While the system may use a faster transfer rate if possible, you are at assured that the rate will not be less than the specified speed.

The transfer method (such as **DMA** or interrupt) chosen by the system is determined by the minimum speed requested. The system selects a transfer method that is as fast or faster than the requested speed. If the requested speed is beyond system limitations, the fastest available transfer method is used. Refer to the appropriate hardware-specific appendix for details.

Setting the Read Termination Pattern

During read operations on an open device file, the system recognizes certain conditions as the end of a data transfer from the sending device. DIL supports three methods for identifying the end of an input operation:

- Input data byte count limit is reached.
- Hardware condition is used to identify end of data.
- Predetermined character or sequence of characters is used to identify the end of a data record.

Input termination occurs when the first termination condition is recognized, independent of the type of condition. If two or more conditions occur simultaneously, the first condition detected terminates the operation. However, this first condition along with any other simultaneous events that would also

have caused termination are recorded during clean-up at the end of the transfer for possible later use by `io_get_term_reason`.

Termination on Byte Count

Any call to *read* must specify the maximum number of data bytes that are to be accepted. When the specified number of bytes have been read, the data transfer is unconditionally terminated, whether the data is complete or not.

Termination on Hardware Condition

In many cases, the number of bytes being transferred is controlled by the peripheral device and cannot be predetermined. To make sure that no data is lost, the byte limit is set to a value higher than the longest expected input data record, and the interface is configured to recognize a condition, character, or set of characters (one or two bytes only) as the end of the incoming data. For instance, if an HP-IB interface detects that the EOI line has been asserted, it knows that the last data byte has been transferred and halts the read operation, whether or not the specified byte count has been reached.

Termination on Data Pattern

The DIL routine `io_eol_ctl` configures an interface to recognize a particular character or pair of characters as a **read termination pattern.** Whether one or two bytes are used for the pattern depends on whether the data path width is set to 8 or 16 bits. The read termination pattern is in addition to any other conditions that may already be in effect for the interface. The call to `io_eol_ctl` has the form:

```
int  eid, flag, match;
:
io_eol_ctl(eid, flag, match);
```

where *eid* is the entity identifier for the open interface file and *flag,* depending on its value, enables or disables the interface's ability to recognize a read termination pattern.

When *flag* is zero, termination pattern recognition is disabled and only EOI or a satisfied byte count can terminate a normal transfer. If *flag* is non-zero, *match* defines the new termination pattern. When using *flag* = 0 to disable eol pattern recognition, the third parameter (**match**) in the subroutine call is not used. However, it is recommended that a value (such as zero) be provided as good programming practice.

When *flag* is non-zero to enable end-of-line recognition (for example, *flag* = 1) and the interface data path width is set to 8 bits, the least-significant byte of the 4-byte integer value of **match** defines the termination pattern used to identify an end-of-line condition.

On the other hand, if the interface data path width is set to 16 bits (such as with a GPIO interface), then, for most systems, the **termination pattern** is also 16 bits, defined by the two lower (least-significant) bytes of the 4-byte integer value defined by **match.**

Remember: If any other read termination conditions defined for the interface are in effect (such as EOI for an HP-IB interface), *any* event that matches a currently active termination condition can cause a read operation to halt; independent of whether the defined eol condition has been met. Also note that the read termination pattern defined by `io_eol_ctl` is accepted as part of the valid incoming data, meaning that it is transferred to the data storage area along with the rest of the transferred data. In other words, when the interface encounters transferred data matching the **match** value, it treats the data as part of the data message but does not attempt any further data input after the matching data pattern is found. This means that if data within an incoming data stream happens to match the pattern defined by **match,** the read is terminated whether the data message is complete or not. For this reason, care must be exercised when defining eol character sequences for data transfer.

To illustrate how to use `io_eol_ctl`, suppose an HP-IB interface is being configured to recognize a backslash-n (`\n`) as a read termination pattern. First, open the HP-IB interface file and obtain the entity identifier *eid.* Second, make the call to `io_eol_ctl` using *eid* as the entity identifier, `ENABLE` as the flag, and `\n` as the match (`\n` is a one-byte value, and the data path width for all HP-IB devices is 8 bits):

```
#include <fcntl.h>
#include <errno.h>
#define  ENABLE      1
main()
{
   int eid;

   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);

   io_eol_ctl(eid, ENABLE, '\n');

   :       /*data transfers using "eid" terminate with a '\n'*/

}
```

Interface file `/dev/raw_hpib` is now configured to terminate read operations when any one of the following occurs:

- The byte count specified in the call to *read* is reached.
- The HP-IB EOI line is asserted. When the interface detects that the EOI line has been asserted, the character currently on the bus becomes the last byte in the data message.
- backslash-n (`\n`) (newline character) is detected in incoming data. The newline character becomes the last byte in the stored data message.

An interface file entity identifier returned by a *dup*(2) system call or inherited by a *fork* request shares the same read termination pattern as the entity identifier returned by the original call to *open*. If the child process resulting from a *fork* inherits an entity identifier then sets a read termination pattern for that *eid*, the *eid* used by the parent process is also affected.

If a single program or process executes more than one *open* system call on the same interface file, each entity identifier returned by *open* can have its own associated read termination pattern. Using `io_eol_ctl` on a given *eid* does not affect the others. Thus, multiple entity identifiers can be set up for a single interface to facilitate recognition of various termination characters during program execution.

Disabling a Read Termination Pattern

To disable the read termination pattern, call `io_eol_ctl` with the *flag* parameter disabled (set to 0):

```
io_eol_ctl(eid, 0, xx);
```

where xx represents a "don't care" value for the *match* argument. If the flag argument is 0, the match argument is ignored.

The following code segment defines the ASCII . character (decimal value 46) as a termination pattern, performs a read operation, then disables termination pattern recognition.

```
#include <fcntl.h>
#include <errno.h>
   main()
   {
      int eid;
      char buffer[12];

      if ((eid = open("/dev/hpib_dev", O_RDWR)) == -1) {
          printf("open failed, errno = %d\n", errno);
          exit(2);
      }
      io_reset(eid);
      io_timeout_ctl(eid, 10000000);

      io_eol_ctl(eid, 1, 46);
      read( eid, buffer, 12); /*Read operation halts when a period character
                                 "." is read or when the 12th byte is read*/
      io_eol_ctl( eid, 0, 0); /*termination pattern recognition is disabled*/
      .
      .
   }
```

Determining Why a Read Terminated

Various situations can cause termination of read operations through an interface. Upon completion of a *read,* you may want to include code to verify that the reason for termination is what you expected. This is done by using the DIL routine `io_get_term_reason`.

`io_get_term_reason` uses a single argument: the interface file entity identifier *eid,* and returns an integer. The returned value indicating how the last read operation ended, is interpreted as follows:

Returned Value	Meaning
−1	An error during the subroutine call.
0	Read terminated abnormally (for some reason other than the ones listed here).
1	Byte count limit caused termination.
2	End-of-line character pattern caused termination
4	Device-imposed condition (such as EOI asserted on HP-IB interface) caused termination.

If more than one termination condition occurred simultaneously, the bit corresponding to the above values is set for each condition, and the aggregate value of the lower three bits represents a sum equal to the combined values of the individual conditions. The three least-significant bits of the lowest byte have meanings as indicated by their associated decimal values in the table above. For example, if `io_get_term_reason` returns a value of 7, all three conditions: byte count limit, hardware termination, and termination pattern recognition occurred simultaneously.

Note If no *read* is performed on an open interface file prior to a call to `io_get_term_reason`, a value of zero is returned.

All entity identifiers descending from a single *open* request (such as from *dup* or *fork*) affect the status returned by this routine. For example, suppose that an entity identifier is inherited by a child process through a *fork*. If the parent process calls `io_get_term_reason`, the last read operation of either the parent or the child is looked at, depending on which is more recent.

Example

Suppose you want to read data through an open HP-IB interface file, but want a printout indicating the reason for termination on every transfer, whether the termination was normal or abnormal. The following code segment provides that capability:

```
#include <fcntl.h>
#include <errno.h>

/*
** possible termination reasons
** returned by io_get_term_reason
*/
#define TR_ABNORMAL  0       /* abnormal */
#define TR_COUNT     1       /* requested count was satisfied */
#define TR_MATCH     2       /* specified eol character was matched */
#define TR_CNT_MCH   3       /* TR_COUNT + TR_MATCH */
#define TR_END       4       /* EOI was detected */
#define TR_CNT_END   5       /* TR_COUNT + TR_END */
#define TR_MCH_END   6       /* TR_MATCH + TR_END */
#define TR_CNT_MCH_END       7       /* TR_COUNT + TR_MATCH + TR_END */

      main()
   {
       int eid, termination_reason, bytes_read;
       char buffer[50];

       if ((eid = open("/dev/raw_hpib", O_RDWR)) < 0) {
             printf("Open of /dev/raw_hpib failed - errno = %d\n", errno);
             exit(1);
       }
       io_reset(eid);
       io_timeout_ctl(eid, 1000000);

       bytes_read = read(eid, buffer, 50);
       termination_reason = io_get_term_reason(eid);
```

```
        switch (termination_reason) {
              case TR_ABNORMAL:       /* abnormal */
                     printf("Abnormal read termination, bytes_read = %d,
errno = %d\n", bytes_read, errno);
                     break;
              case TR_COUNT:          /* requested count was satisfied */
                     printf("Count satisfied.\n");
                     break;
              case TR_MATCH:          /* specified eol character was matched
*/
                     printf("EOL character satisfied.\n");
                     break;
              case TR_CNT_MCH:        /* TR_COUNT + TR_MATCH */
                     printf("Count and EOL character satisfied.\n");
                     break;
              case TR_END:            /* EOI was detected */
                     printf("EOI detected.\n");
                     break;
              case TR_CNT_END:        /* TR_COUNT + TR_END */
                     printf("Count satisfied and EOI detected.\n");
                     break;
              case TR_MCH_END:        /* TR_MATCH + TR_END */
                     printf("EOL character satisfied and EOI detected.\n");
                     break;
              case TR_CNT_MCH_END:    /* TR_COUNT + TR_MATCH + TR_END */
                     printf("Count and EOL character satisfied and EOI
detected.\n");
                     break;
              default:                /* io_get_term_reasoned failed */
                     printf("io_get_term_reason failed, bytes_read = %d,
errno = %d\n", bytes_read, errno);
                     break;
        }
}
```

Interrupts

DIL provides an interrupt mechanism for HP-IB and GPIO interfaces that is similar to HP-UX signal handling. Thus **interrupt handlers** can be included in programs such that they are invoked when certain conditions occur.

Series 300 and 800 Interrupt Support

HP-IB Interrupts

Series 300 and 800 computers recognize the following HP-IB interrupt conditions:

signal	Condition
SRQ	SRQ line has been asserted.
TLK	Computer HP-IB interface has been addressed to talk.
LTN	Computer HP-IB interface has been addressed to listen.
CIC	Computer HP-IB interface has received control of the bus.
IFC	IFC line has been asserted.
REN	Remote enable line has been asserted.
DCL	Computer HP-IB interface has received a device clear command.
GET	Computer HP-IB interface has received a group execution trigger command.
PPOLL	A specific parallel poll response occurred.

GPIO Interrupts

Series 300 computers recognize the following GPIO interrupt condition:

EIR EIR line has been asserted.

The Series 800 HP 27112 GPIO interface recognizes the following interrupt conditions:

SIE0 Status line 0 has been set.

SIE1 Status line 1 has been set.

The Series 800 HP 27114 GPIO interface recognizes the following interrupt condition:

EIR EIR line has been asserted.

io_on_interrupt

DIL provides two subroutines for controlling interrupts: `io_on_interrupt` and `io_interrupt_ctl`. The first, `io_on_interrupt`, sets up interrupt conditions and has the form:

```
io_on_interrupt(eid, cause_vec, handler);
```

where *eid* is the interface entity identifier for a GPIO or raw HP-IB interface. **handler** points to the function that is to be invoked when the interrupt condition occurs, and `cause_vec` is a pointer to a structure of the form:

```
struct interrupt_struct {
          int cause;
          int mask;
};
```

The **interrupt_struct** structure is defined in the include file `dvio.h`.

cause is a bit vector specifying which selectable interrupt or fault events will cause the *handler* routine to be involked Available interrupt **cause**s are usually specific to the type of interface being considered. In addition, certain exception (error) conditions can be handled by the `io_on_interrupt` subroutine. If the **cause** vector has a zero value, it, in effect, disables interrupts for that *eid.*

mask is an integer value that is used to define which parallel-poll response lines are to be recognized in an HP-IB parallel poll interrupt. The value for **mask** is formed from an 8-bit binary number, each bit of which corresponds to one

of the eight parallel-poll response lines. For example, to invoke an interrupt handler for a response on line 2 or 6, the correct binary number is 01000100 which converts to a decimal equivalent of 68, the correct value for **mask.**

When the enabled interrupt condition occurs on the specified *eid,* the process that set up the interrupt executes the interrupt-handler routine pointed to by *handler.* The entity identifier *eid* and the interrupt condition **cause** are returned to *handler* as the first and second parameters respectively.

Whenever an interrupt condition occurs for a given *eid,* the interrupt is recognized, interrupts are disabled for that *eid,* then the interrupt handler is executed. After processing the interrupt, interrupts can be re-enabled for that *eid* by calling `io_interrupt_ctl`.

Each call to `io_on_interrupt` returns a pointer to the previous handler if the new handler is successfully installed, otherwise it returns −1 and **errno** is set.

The following example illustrates how an interrupt handler can be set up to handle requests on the HP-IB service request line (SRQ):

```
#include <dvio.h>
#include <fcntl.h>
#include <stdio.h>
extern int service_routine();

handler (eid, cause_vec)
int eid;
struct interrupt_struct *cause_vec;
{
    if (cause_vec->cause == SRQ)
       service_routine(); /* application-specific service routine*/
}
main()
{
   int eid;
   struct interrupt_struct cause_vec;

   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);

   cause_vec.cause = SRQ;
   io_on_interrupt(eid, &cause_vec, handler);
   .
   .
   .
}
```

io_interrupt_ctl

Subroutine `io_interrupt_ctl` provides a convenient means for enabling and disabling interrupts on a specific *eid.* Since interrupts are automatically disabled when an interrupt occurs, `io_interrupt_ctl` is commonly used to re-enable interrupts during a series of repetitive operations that are being handled under interrupt control. The call to `io_interrupt_ctl` has the following form:

```
io_interrupt_ctl(eid, enable_flag);
```

where *eid* is the entity identifier for an open GPIO or raw HP-IB interface (device) file. The value of *enable_flag* determines whether interrupts are to be enabled or disabled: if *enable_flag* is non-zero, interrupts are enabled on the *eid;* if *enable_flag* is zero, interrupts are disabled. Attempting to use `io_interrupt_ctl` on an *eid* fails when no previous call has been made to `io_on_interrupt` for the same *eid.*

The following code segment shows how the previous example can be modified slightly so that interrupts are re-enabled at the end of the interrupt service routine:

```
handler(eid, cause_vec);
int eid;
struct interrupt_struct *cause_vec;
{
   if (cause_vec->cause == SRQ)

      service_routine(); /* application-specific service routine*/

   io_interrupt_ctl(eid,1);

}
```

3

Controlling the HP-IB Interface

The general-purpose subroutines discussed in Chapter 2 are used to set up and handle data transfers at a high level. However, they do not control the lower-level interface operations that are necessary to maintain proper bus operation and control interaction between HP-IB devices.

This chapter explains the use of subroutines in the Device I/O Library that are directly related to HP-IB interface control. Chapter 4 covers comparable material for the GPIO interface. This chapter presents a brief overview of HP-IB commands, followed by a detailed discussion of HP-IB DIL subroutines including how they are used to control bus activity and manage bus traffic.

Overview of HP-IB Commands

HP-IB commands consist of various data sequences that are sent over the eight HP-IB data lines while the ATN line is asserted (held LOW). The DIL subroutine `hpib_send_cmnd` provides a convenient means for sending bus commands by automatically handling the ATN line and the necessary handshaking operations between devices. However, `hpib_send_cmnd` can be used *only* when the computer interface to the bus is the active controller. Techniques for using `hpib_send_cmnd` are discussed later in this chapter.

Any device that is the intended recipient of an HP-IB command must have its remote enable line (REN) enabled by the System Controller (unless altered by the System Controller, REN is enabled, by default). Only the System Controller can alter the state of the REN line (see "System Controller's Duties" section later in this chapter).

HP-IB Data Bus Commands fall into four categories:

- **Universal commands** cause every properly equipped device on the bus to perform the specified interface operation, whether addressed to listen or not.
- **Addressed commands** are similar to universal commands, but are accepted only by bus devices that are currently addressed as listeners.
- **Talk and listen addresses** are commands that assign talkers and listeners on the bus.
- **Secondary commands** are commands that must always be used in conjunction with a command from one of the above groups.

The following table lists commands that can be sent with `hpib_send_cmnd`, along with the decimal and ASCII character equivalents of each command. This table is useful for reference when determining what values to use as parameters in `hpib_send_cmnd` subroutine calls.

Table 3-1. HP-IB Bus Commands

Command	Decimal Value	ASCII Character
Universal Commands:		
UNLISTEN	63	?
UNTALK	95	_
DEVICE CLEAR	20	DC4
LOCAL LOCKOUT	17	DC1
SERIAL POLL ENABLE	24	CAN
SERIAL POLL DISABLE	25	EM
PARALLEL POLL UNCONFIGURE	21	NAK
Addressed Commands:		
TRIGGER	8	BS
SELECTED DEVICE CLEAR	4	EOT
GO TO LOCAL	1	SOH
PARALLEL POLL CONFIGURE	5	ENQ
TAKE CONTROL	9	HT

Table 3-1. HP-IB Bus Commands (cont'd)

Command	Decimal Value	ASCII Character
Talk and Listen Addresses:		
Talk Addresses 0-30	64-94	@ thru ^ (uppercase ASCII)
Listen Addresses 0-30	32-62	space thru > (numbers and special characters)
Secondary Commands: (If a secondary command follows the PARALLEL POLL CONFIGURE command, it is interpreted as follows; otherwise its meaning is device-dependent)		
PARALLEL POLL ENABLE	96-111	` thru o (lowercase ASCII)
PARALLEL POLL DISABLE	112	p

UNLISTEN

UNLISTEN *unaddresses* all current listeners on the bus. No means is available for unaddressing a given listener without unaddressing all listeners on the bus. This command ensures that the bus is cleared of all listeners before addressing a new listener or group of listeners.

UNTALK

UNTALK *unaddresses* any active talkers on the bus. Since no means is available for unaddressing a given talker, the UNTALK command is sent to all devices on the bus. This ensures that no conflict with a current talker can occur when addressing a new one.

DEVICE CLEAR

DEVICE CLEAR causes all devices that recognize this command to return to a pre-defined, device-dependent state, independent of any previous addressing. The reset state for any given device after accepting this command is documented in the operating manual for the device in question.

LOCAL LOCKOUT

LOCAL LOCKOUT disables local (front panel) control on all devices that recognize this command, whether the devices have been addressed or not.

SERIAL POLL ENABLE

SERIAL POLL ENABLE establishes serial poll mode for all devices that are capable of being bus talkers, provided they recognize and support the command. This command operates independent of whether the devices being polled have been addressed to talk. When a device is addressed to talk, it returns an 8-bit status byte message.

This command is handled through the DIL subroutine `hpib_spoll`, as discussed later in this chapter.

SERIAL POLL DISABLE

SERIAL POLL DISABLE terminates serial poll mode for all devices that support this command, whether or not the individual devices have been addressed.

The DIL subroutine `hpib_spoll` that performs this function is discussed at length later in this chapter.

TRIGGER (Group Execute Trigger)

TRIGGER causes devices currently addressed as listeners to initiate a preprogrammed, device-dependent action if they are capable of doing so. Use of this function and programming procedures are documented in operating manuals for devices that support it.

SELECTED DEVICE CLEAR

SELECTED DEVICE CLEAR resets devices currently addressed as listeners to a device-dependent state, provided they support the command. Refer to the device operating manual for more information about programming and the resulting state(s).

GO TO LOCAL

GO TO LOCAL causes devices currently addressed as listeners to return to the local-control state (exit from the remote state). Devices return to remote state next time they are addressed.

PARALLEL POLL CONFIGURE

PARALLEL POLL CONFIGURE tells devices currently addressed as listeners that a secondary command follows. This secondary command must be either PARALLEL POLL ENABLE or PARALLEL POLL DISABLE.

PARALLEL POLL ENABLE

PARALLEL POLL ENABLE configures devices addressed by PARALLEL POLL CONFIGURE to respond to parallel polls with a predefined logic level on a particular data line. On some devices, the response is implemented in a local form (such as by using hardware jumper wires) that cannot be changed.

Use of this command must be preceded by a PARALLEL POLL CONFIGURE command.

PARALLEL POLL DISABLE

The PARALLEL POLL DISABLE command prevents devices previously addressed by a PARALLEL POLL CONFIGURE command from responding to parallel polls. This command must be preceded by the PARALLEL POLL CONFIGURE command.

Overview of HP-IB DIL Routines

Standard DIL Routines

These 17 subroutines, in addition to the general-purpose subroutines discussed in Chapter 2, provide full capabilities for controlling and using the HP-IB interface.

Subroutine	Description
`hpib_abort`	Stop activity on specified HP-IB select code.
`hpib_io`	Perform a series of HP-IB read, write, and SEND_CMD operations from a single subroutine call.
`hpib_ppoll`	Conduct parallel poll on HP-IB.
`hpib_spoll`	Conduct serial poll on HP-IB.
`hpib_bus_status`	Return status on HP-IB interface.
`hpib_eoi_ctl`	Control EOI mode for data transfers.
`hpib_pass_ctl`	Pass bus control to another device on the bus.
`hpib_card_ppoll_resp`	Define HP-IB card's response to a parallel poll.
`hpib_ren_ctl`	Assert or release HP-IB remote-enable (REN) line on HP-IB.
`hpib_rqst_srvce`	Initiate a service request (SRQ) when interface is not Active Controller.
`hpib_send_cmnd`	Send command message on HP-IB data lines while asserting the attention (ATN) line.
`hpib_wait_on_ppoll`	Wait until a specified device responds on its assigned parallel poll response line indicating that it needs service.
`hpib_status_wait`	Wait until any device on the bus asserts SRQ.
`hpib_ppoll_resp_ctl`	Configure and enable or disable the parallel poll response circuit on the specified device (determines how the device will respond to the next parallel poll from a remote active controller).
`hpib_atn_ctl`	Control the HP-IB ATN line.
`hpib_parity_ctl`	Set parity type to be used for `hpib_send_cmnd` calls.
`hpib_address_ctl`	Set the bus address of an HP-IB interface card.

HP-IB: The Computer's Role

Most HP-IB applications consist of a single computer and several peripheral devices connected to a given bus. However, some situations may require two or more computers on the same bus along with various shared and/or dedicated peripheral devices. This discussion applies to both configurations.

Ground Rules

The following rules are mandatory for proper HP-IB interaction:

- HP-IB allows only one *System Controller* per bus.
- Only one device on the bus can be *active controller* at any given time.
- All other devices capable of controlling the bus must be *non-active controllers* unless control is passed from another active controller.
- The computer interface is configured as System Controller. If two or more computers are interfaced to a single bus, only one can be configured as System Controller. All other interfaces must be configured as non-controllers (incapable of acting as System Controller). This is usually accomplished by programming a switch or jumper on the HP-IB interface card.

At power-up, the System Controller is the Active Controller. All other controllers on the bus are non-active controllers. If the computer interface passes control to another device, the device receiving control becomes the new active controller and the computer interface becomes a non-active controller although it remains System Controller at all times and can regain control of the bus by asserting $\overline{\text{IFC}}$ (InterFace Clear). Once control has been passed to another device, the computer remains non-active controller until control is passed back or $\overline{\text{IFC}}$ is asserted.

Available Subroutines versus Controller Role

Which DIL subroutines can be used depends on the computer's role on the HP-IB at the time. Given the three possible roles, Table 3-2 indicates which subroutines can be used with each.

Table 3-2. DIL Subroutine Availability Based on Interface Role.

Subroutine	System Controller	Active Controller	Non-Active Controller
hpib_abort	•		
hpib_io		•	
hpib_ppoll		•	
hpib_spoll		•	
hpib_bus_status	Note 1	•	•
hpib_eoi_ctl	•		
hpib_pass_ctl		•	
hpib_card_ppoll_resp		Note 2	•
hpib_ren_ctl	•		
hpib_rqst_srvce		Note 2	•
hpib_send_cmnd		•	
hpib_wait_on_ppoll		•	
hpib_status_wait	Note 1	•	•
hpib_ppoll_resp_ctl		Note 2	•
hpib_parity_ctl	Note 1	•	•
hpib_atn_ctl		•	
hpib_address_ctl	Note 1	•	•

Note 1 This command is available to the System controller, but the availability is meaningless because this command is available to any interface on the bus, independent of its role as an active or non-active controller.

Note 2 This command is available to the interface while it is active controller, but the command is meaningless except when the interface is acting in the non-active controller role.

Bus Citizenship: Surviving Multi-Device/Multi-Process HP-IB

HP-UX provides a powerful environment for creative programming. As a result, one or more users can create a large number of processes that may be running simultaneously. At the same time, HP-IB provides the capability of combining multiple devices on a single I/O channel or interface. As long as only auto-addressed HP-IB interface files are used, problems are few and infrequent. However, when processes that use DIL subroutines start accessing raw-mode HP-IB interface files, a splendid opportunity arises for competing processes to create bus addressing and access conflicts. If certain precautions are not carefully maintained, performance quickly decays to chaos.

The Device I/O Library contains several subroutines that are provided specifically for maintaining orderly HP-IB traffic and good I/O efficiency. Correct use of these subroutines is especially important when using raw interface files. They include:

- `io_lock` and `io_unlock` to take exclusive control of the HP-IB channel for the duration of a transfer,
- `io_burst` to efficiently handle short transfers without consuming large amounts of HP-UX kernel overhead,
- `hpib_io` to structure a complete bus transfer including configuration and control operations in a buffer then handle the transfer as a single subroutine call through an interface file that is automatically locked at the beginning and released at the end of the transfer.

These subroutines are discussed at length later in this chapter, but are treated here from the point of view of overall bus applications efficiency as it pertains to programming practice.

io_lock and io_unlock

When handling raw-mode (as opposed to auto-addressed) HP-IB transfers, devices must be set up to communicate (preamble) before the transfer (read/write) can be initiated, then the necessary clean-up (postamble) operations must be performed to leave the bus in an acceptable state for the next process. If you do not notify other processes that you are using the bus, they might initiate a different transfer while you are preparing for your next DIL subroutine call. A command sequence from another process (through a different *eid* but through the same interface) could completely scramble your bus configuration so your transfer request results in no data, erroneous data, or possibly even more serious results, depending on the nature of the transfer.

A simple call to `io_lock` prior to your first call to an HP-IB subroutine and a matching call to `io_unlock` after your last HP-IB subroutine call keeps competing processes from using the bus while you have control. As soon as the interface file is unlocked, it can be accessed by the next process that needs it.

io_burst

Series 300 systems support burst I/O (also called fast handshake) which bypasses the kernel by performing a high-speed non-interrupt transfer. This method can produce considerable performance improvement when handling short transfers to or from high-speed HP-IB devices. Refer to the *io_burst*(3I) manual entry in Section 3 of the *HP-UX Reference* for more information.

hpib_io

The DIL subroutine `hpib_io` is used to perform bus configuration, data transfer, and bus clean-up as a single operation through a locked interface file. When using `hpib_io`, control commands (the preamble), data to be written or a buffer for incoming data (the data message), and clean-up commands (postamble) are placed in a data structure prior to calling `hpib_io`. `hpib_io` then handles the transfer as defined in the data structure (which configures the HP-IB and handles the transfer and clean-up) then returns with the result (transfer complete or transfer failed).

Opening the HP-IB Interface File

Before DIL subroutines can be used on an HP-IB interface, the interface special file must exist and the program must obtain a corresponding entity identifier. The procedures for opening interface special files and obtaining entity identifiers is discussed in Chapter 2, "General-Purpose Routines."

Sending HP-IB Commands

Once the HP-IB interface special file has been opened and the entity identifier has been obtained, DIL subroutines can be used to send HP-IB commands to control the interface. If the computer is Active Controller, `hpib_send_cmnd` can be used to place HP-IB commands on the data bus.

One method of using this routine is to first set up a character array containing the commands being sent. Assign the decimal value of each command to an element in the array, then use a subroutine call having the form:

```
hpib_send_cmnd(eid, command, number);
```

where *eid* is the entity identifier for the open interface file, *command* is a character pointer to the first element of the array containing the HP-IB commands, and *number* is the number of elements (commands) in the array. The subroutine `hpib_send_cmnd` places each of the commands stored in the array on the bus with ATN asserted.

Notice that by changing the *number* argument and moving the *command* pointer you can send subsets of command arrays. Suppose you create an array that contains 10 HP-IB commands, command[0] through command[9]. You can now specify that only the last 5 commands in the array be sent by using:

```
hpib_send_cmnd(eid, command + 5, 5);
```

This method of sending HP-IB commands by storing them in an array uses their decimal values. Alternatively, ASCII command characters can be used by specifying a character string and using a subroutine call of the form:

```
hpib_send_cmnd(eid, "command_string", number);
```

where *eid* and *number* are the same as before but the commands to be sent are now specified by each character in the string *command_string.*

To illustrate the two methods, assume that you want to send the HP-IB UNLISTEN and UNTALK commands. With the decimal array method, first set up an array having two elements, place the decimal value for each command in the appropriate location in the array, then call `hpib_send_cmnd`:

```
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid;
   char command[2];          /*command array*/

   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid, 1000000);

   command[0] = 63;          /*decimal value for UNLISTEN*/
   command[1] = 95;          /*decimal value for UNTALK*/
   hpib_send_cmnd(eid, command, 2);
}
```

Using the ASCII character string method, the same effect is achieved using:

```
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid;

   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid, 1000000);

   hpib_send_cmnd(eid, "?_", 2);  /*? is ASCII for UNLISTEN and*/
                                  /*_ is ASCII for UNTALK      */
}
```

The array method is usually preferred when sending a large number of commands or sending the same set of commands several times in the program because the entire set of commands can be stored once then used whenever needed. When the string method is used, the entire set of commands must be specified as a string in each call to `hpib_send_cmnd`. It is preferred when sending only a few commands or sending a set of commands only once in a program.

Errors While Sending Commands

Normally, `hpib_send_cmnd` returns a 0 if successful. It returns a −1 if any one of the following error conditions exist:

- Computer interface is not Active Controller.
- *eid* entity identifier does not refer to an HP-IB raw interface file.
- *eid* entity identifier does not refer to an open file.
- A timeout occurs.
- The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid.*
- The command length specified by *number* is invalid.

To determine which of these conditions caused the error, check the value of *errno,* an external integer variable used by HP-UX system calls. Error-checking routines are discussed at length in Chapter 2.

The following table lists *errno* values corresponding to the conditions above when detected by `hpib_send_cmnd`:

errno Value	Error Condition
EBADF	*eid* did not refer to an open file
ENOTTY	*eid* did not refer to a raw interface file
EIO	The interface was not the Active Controller (EACCES on Series 800)
ETIMEDOUT	A timeout occurred (EIO on Series 300)
EACCES	The interface associated with this *eid* was locked by another process and `O_NDELAY` was set for this *eid*
EINVAL	*number* was invalid, either less than or equal to 0 or greater than `MAX_HPIB_COMMANDS` as defined in `dvio.h`

Changing Parity on Commands

By default, bus commands sent across the bus using `hpib_send_cmnd` are sent using odd parity. On the Series 300, you can disable the use of parity on bus commands using the `hpib_parity_ctl` routine.

The following sequence illustrates the use of `hpib_parity_ctl` to disable the sending of parity and use eight bit command bytes:

```
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid;
   char command[2];          /*command array*/

   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid, 1000000);

   hpib_parity_ctl(eid, 0);
   command[0] = 63;          /*decimal value for UNLISTEN*/
   command[1] = 95;          /*decimal value for UNTALK*/
   hpib_send_cmnd(eid, command, 2);
}
```

Active Controller Role

The Active Controller is responsible for originating all commands handled on the bus and responding to requests for service from other devices. `hpib_send_cmnd` is used to send HP-IB commands. Other DIL subroutines are used for the remaining bus control tasks. Active Controller operations discussed in this chapter include:

- Addressing individual devices to talk or listen.
- Switching devices to remote control operation.

- Locking out local front-panel control on devices.
- Switching devices to local front-panel control.
- Triggering devices to initiate device-dependent operations.
- Transferring data in or out.
- Clearing (resetting) devices
- Responding to service requests from devices.
- Conducting parallel and serial polls.
- Passing active control of the bus to another device.

Determining Active Controller

A computer interface must be the Active Controller before it can handle any bus management activities. If any other device on the bus is capable of being Active Controller, use the `hpib_bus_status` subroutine to determine whether the interface is the current Active Controller. Use the following subroutine call form:

```
hpib_bus_status(eid,ACT_CONT_STATUS);
```

where *eid* is the entity identifier for the opened HP-IB interface device file and *ACT_CONT_STATUS* tells the subroutine to examine interface status and determine whether or not the card is the Active Controller. The value returned by the subroutine can be tested as indicated in the example source code which follows.

`hpib_bus_status` returns 0 if the condition being tested is false; 1 if true, and −1 if an error occurred. The code that follows shows a straightforward way of interpreting the returned value:

```
#include <dvio.h>
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid, status;
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
```

```
    if ((status = hpib_bus_status(eid,ACT_CONT_STATUS)) == -1)
    .                   /*an error occurred; error-handling code*/
    .                   /*goes here.                            */
else if (status == 0)
    .                   /*not Active Controller; code to request  */
    .                   /*Active Controller status goes here */
else
    .                   /*Active Controller; bus-management code */
    .                   /*goes here    */
}
```

Setting Up Talkers and Listeners

Before data can be transferred over HP-IB, one talker and one or more listeners must be assigned to handle the transfer. In addition, some HP-IB commands are recognized only by those devices that are currently addressed as listeners, which means that the Active Controller must specify the listeners before sending such commands. Only one talker at a time is allowed on the bus, but the number of listeners is not restricted.

Series 300 and 800 computers provide two methods for addressing listeners and talkers on HP-IB: auto-addressing and command addressing.

When an HP-IB interface device file is set up as an auto-addressed file (determined by the value of the minor number used when creating the file), any read/write operations to or from the file automatically set up the bus talk and listen address commands prior to transferring data. The interface must be the Active Controller when auto-addressing is used.

The alternate method uses `hpib_send_cmnd` to directly control the bus from the user program itself. However, this method of control can only be used on raw device special files.

Auto-Addressing

Much of the tedium of addressing devices to talk or listen can be avoided by using auto-addressed device special files to take advantage of HP-UX

auto-addressing capabilities for many peripherals. Auto-addressing is performed only on auto-addressed HP-IB device files. Some DIL subroutines require a *raw* HP-IB device file, and will fail if you attempt to use them on an auto-addressed device file. DIL subroutines that can be used with auto addressed device files include `hpib_eoi_ctl`, `hpib_eol_ctl`, `io_burst`, `io_get_term_reason`, `io_lock`, `io_unlock`, `io_speed_ctl`, and `io_timeout_ctl`. Systems determine whether a device file is raw or auto-addressed by the minor number used when the file is created. Address 31 (hexadecimal 1f) is reserved for raw files. Any address in the range 0 through 30 is auto-addressed. Refer to the appropriate appendix for procedures used to create device and interface special files.

For example, suppose you are using a Series 300 computer with an HP 98624 HP-IB card on select code 08 to access a peripheral device located at bus address 03. Use `mknod` to create a new device file named *device* for the peripheral device and place the file in directory `dev` underneath the root directory as explained in Appendix A:

```
mknod /dev/device c 21 0x080300
```

Once the file exists, it can be listed by using the ll(1) command. In this case, the device file named */dev/device* is listed (along with other files in the /dev directory) together with its permissions and attributes:

```
crw-rw-rw-   1 root    other      21 0x080300 Nov  22 1986 /dev/device
```

Since the bus address is less than decimal 31, the file is a non-raw device file and is auto-addressable. The following code segment illustrates how to use auto-addressing with such a device file:

```
#include <errno.h>
#include <fcntl.h>

main()
{
   int eid;

   if ((eid = open("/dev/device",O_RDWR) < 0)) {
        printf("Open of /dev/device failed, errno = %d\n", errno);
        exit(1);
   }

   /*
   ** Assuming "/dev/device" has the minor number (0x080300), the
   ** system automatically addresses the interface card at select code 8
   ** as a talker and the device at bus address 3 as a listener before
   ** sending data
   */

   if (write(eid, "test data", sizeof("test data")) < 0) {
      printf("write failed, errno = %d\n", errno);
      exit(2);
   }
}
```

Using hpib_send_cmnd

Talkers and listeners can be configured under program control by forming HP-IB command sequences from the talk and listen addresses of the devices being used. However, before addressing talkers and listeners, clear the bus of any talkers and listeners that might be left over from previous transactions by issuing UNTALK and UNLISTEN commands (whenever a talk address appears on the bus, well-mannered devices should recognize the address and automatically untalk if the address is for a different device. However, not all devices are necessarily well-mannered, so an UNTALK is considered good programming practice). To configure a new talker and listeners:

1. Send an UNTALK command to remove any previous talkers.
2. Send an UNLISTEN command to remove any previous listeners.
3. Send the talk address of the device that will be sending data. There can only be one talker.
4. Send the listen address of each device that is to receive the data.

After data transfer is complete, issue an UNTALK and UNLISTEN command on the bus (repeat steps 1 and 2) to leave it in a clean state for subsequent transactions.

DIL subroutine `hpib_send_cmnd` is used to perform these tasks.

Calculating Talk and Listen Addresses

Before devices can be addressed to talk or listen, their HP-IB bus addresses must be known. The bus address of the computer interface is easily obtained by using `hpib_bus_status` as shown in this program code segment:

```
#include <dvio.h>
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid, address;
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   address = hpib_bus_status(eid, CURRENT_BUS_ADDRESS);
   .
   .
   .
}
```

where *eid* is the entity identifier for the interface file and *CURRENT_BUS_ADDRESS* indicates a request for the interface HP-IB bus address.

To determine the bus address of other devices on the bus, refer to installation and operating manuals for each device being used (certain HP-IB addresses may be reserved for specific devices on some systems).

Once device addresses are known for all devices of interest, setting up talk and listen addresses is a fairly simple matter.

HP-IB commands are set up as a single ASCII character transmitted while $\overline{\text{ATN}}$ is asserted. However, it is usually much easier to calculate addresses based on bus address rather than looking up the corresponding ASCII character for each address. Bus addresses range from 0 through 30, and talk and listen addresses are derived through decimal addition as follows:

```
talk_address = 64 + bus_address
listen_address = 32 + bus_address
```

where *talk_address* is the decimal equivalent of the binary bit pattern that represents the ASCII talk address command character. Likewise, *listen_address* is the decimal representation of the ASCII listen address command character.

bus_address is the decimal value of the HP-IB bus address for the device being addressed.

The talk and listen addresses MTA ("my talk address") and MLA ("my listen address") for the computer interface are derived similarly as follows:

```
MTA = hpib_bus_status(eid, CURRENT_BUS_ADDRESS) + 64;
MLA = hpib_bus_status(eid, CURRENT_BUS_ADDRESS) + 32;
```

An Example Configuration

Assuming that the computer's HP-IB interface is currently the Active Controller, the following code segment establishes the interface as the bus talker. Two devices at HP-IB addresses 4 and 8 are designated as bus listeners.

```
#include <dvio.h>
#include <fcntl.h>
#include <errno.h>
main()
{
     int  eid, MTA;
     char command[5];
     if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
          printf("open failed, errno = %d\n", errno);
          exit(2);
     }

     /*calculate My Talk Address*/
     MTA = hpib_bus_status(eid, CURRENT_BUS_ADDRESS) + 64;
     command[0] = 95;       /* UNTALK command*/
     command[1] = 63;       /* UNLISTEN command*/
     command[2] = MTA;      /* interface talk address*/
     command[3] = 32 + 4;   /* listen address for device at bus address 4*/
     command[4] = 32 + 8;   /* listen address for device at bus address 8*/
     hpib_send_cmnd(eid, command, 5);
}
```

Remote Control of Devices

Most HP-IB devices can be controlled from either their front panel or the bus. If the device's front-panel controls are currently operational, the device is in *local* state. If it is being controlled through the HP-IB, it is in *remote* state. Pressing the device's front-panel LOCAL key returns the device to local control unless it has been placed in local lockout state (described in the next section).

Whether the HP-IB remote enable (REN) line is asserted or not determines whether or not a device can respond to remote program control. While REN is asserted, any device that is addressed to listen is automatically placed in remote state. Only the System Controller can assert or release the REN line. REN, by default, is asserted at power-up and remains asserted unless changed as discussed later in this chapter under the topic *System Controller Operations.*

Locking Out Local Control

The LOCAL LOCKOUT command inhibits the LOCAL key or switch present on the front panel of most HP-IB devices, thus preventing anyone from interfering with system operations by pressing front-panel control buttons. All devices that support local lockout are locked, whether addressed or not, and cannot be returned to local control from their front panels.

The following code segment shows one method for sending the LOCAL LOCKOUT command:

```
.
.
.
command[0] = 17;            /* Decimal value of LOCAL LOCKOUT*/
hpib_send_cmnd(eid, command, 1);
.
.
.
```

The GO TO LOCAL command can be used to place a device in local (front-panel control) state.

Enabling Local Control

During system operation, it may be necessary to place certain devices in local state for direct operator control such as when making special tests or troubleshooting. The GO TO LOCAL command returns all devices currently addressed as listeners to their local state.

For example, the following code segment places devices at bus addresses 3 and 5 in *local* state.

```
:
command[0] = 63;              /* the UNLISTEN command*/
command[1] = 32 + 3;          /* listen address for device at address 3*/
command[2] = 32 + 5;          /* listen address for device at address 5*/
command[3] = 1;               /* the GO TO LOCAL command*/
hpib_send_cmnd(eid, command, 4);
:
```

Triggering Devices

The HP-IB TRIGGER command tells devices currently addressed as listeners to initiate some device-dependent action. A typical use is triggering a measurement cycle on a digital voltmeter. Since device response to a TRIGGER command is strictly device-dependent, HP-IB has no direct control over the type of action being initiated.

The following code triggers the device at bus address 5:

```
:
command[0] = 63;              /* UNLISTEN command*/
command[1] = 32 + 5;          /* listen address for device at address 5*/
command[2] = 8;               /* TRIGGER command*/
hpib_send_cmnd(eid, command, 3);
:
```

Transferring Data

Data Output

To output data from an Active Controller the controller must:

1. Send a bus UNTALK command.
2. Send a bus UNLISTEN command.
3. Send its own talk address (MTA).
4. Send the listen address of the device that is to receive the data. One listen address is sent for every device that is to receive the data.
5. Send the data.
6. Repeat steps 1 and 2 to clean up the bus.

The first 3 steps are accomplished using `hpib_send_cmnd`. The system subroutine `write` takes care of the fourth.

The following code segment illustrates how character data can be sent to a device at HP-IB address 5.

```
#include <dvio.h>
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid, MTA;
   char command[50];

   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid, 1000000);

   MTA = hpib_bus_status(eid, CURRENT_BUS_ADDRESS) + 64; /*compute MTA*/
   command[0] = 95;                    /*UNTALK command*/
   command[1] = 63;                    /*UNLISTEN command*/
   command[2] = MTA;                   /*address interface to talk*/
   command[3] = 32 + 5;                /*listen address of device at*/
                                       /*address 5                  */
   hpib_send_cmnd(eid, command, 4);
   write(eid, "data message", 12);   /*send the data*/
   hpib_send_cmnd(eid, command, 2);    /*clear talkers and listeners*/
}
```

Data Input

Assume that you expect to receive 50 bytes of data from another device on the bus. The following code segment programs the interface to receive character data from a device at bus address 5. The integer variable *MLA* contains the interface listen address.

```
#include <dvio.h>
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid, MLA, len;
   char buffer[51];                        /*storage for data*/
   char command[4];

   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid, 1000000);

   MLA = hpib_bus_status(eid, CURRENT_BUS_ADDRESS) + 32; /*compute MLA*/
   command[0] = 95;                        /*UNTALK command*/
   command[1] = 63;                        /*UNLISTEN command*/
   command[2] = 64 + 5;                    /*address device at address 5*/
                                           /*to talk                    */
   command[3] = MLA;                       /*address interface to listen*/
   hpib_send_cmnd(eid, command, 4);
   len = read(eid, buffer, 50);           /*store the data in "buffer"*/
   buffer[ len ] = '\0';                   /*terminate with NULL for printf*/
   hpib_send_cmnd(eid, command, 2);
   printf("Data read is: %s", buffer);    /*print message*/
}
```

Clearing HP-IB Devices

Two HP-IB commands are used to reset devices to pre-defined, device-dependent states. The first, DEVICE CLEAR, causes all devices that recognize the command to be reset, whether addressed or not. Care should be used not to use this command on an HP-IB bus with a system (non-DIL) device attached.

To reset all devices on an HP-IB accessed through an interface file having entity identifier *eid,* use a code segment similar to:

```
:
command[0] = 20;                /* DEVICE CLEAR command*/
hpib_send_cmnd(eid, command, 1);
:
```

The second command for resetting devices is SELECTED DEVICE CLEAR. This command resets only those devices that are currently addressed as listeners.

To reset a device at HP-IB address 7, use a code segment such as this (the interface must already be addressed to talk):

```
:
command[0] = 63;                /* the UNLISTEN command*/
command[1] = 32 + 7;            /* the listen address for device at*/
                                /* address 7                       */
command[2] = 4;                 /* the SELECTED DEVICE CLEAR command*/
hpib_send_cmnd(eid, command, 3);
:
```

Responding to Service Requests

Most HP-IB devices, such as voltmeters, frequency counters, and spectrum analyzers, are capable of generating a *service request* when they require the Active Controller to take some action. *Service requests* are generally made after the device has completed a task (such as taking a measurement) or when an error condition exists (such as a printer being out of paper). The operating or programming manual for each device describes the device's capability to request service and the conditions under which it requests service.

Monitoring the SRQ Line

To request service, a device asserts the bus Service Request ($\overline{\text{SRQ}}$) line. To determine if SRQ is being asserted, check the status of the line, wait for SRQ, or set up an interrupt handler for SRQ. The `hpib_status_wait` subroutine provides a means for suspending program operation until the SRQ line is asserted then continuing. To structure a program so that it waits until SRQ line is asserted, invoke `hpib_status_wait` as follows:

```
hpib_status_wait(eid, WAIT_FOR_SRQ);
```

where *eid* is the entity identifier for the open interface file and *WAIT_FOR_SRQ* indicates that the event that you are waiting for is the assertion of SRQ. The subroutine returns 0 when the condition requested becomes true or -1 if a timeout or an error occurred.

The following code segment illustrates the use of `hpib_status_wait`:

```
#include <dvio.h>
#include <fcntl.h>
#include <errno.h>
extern int service_routine();
main()
{
   int eid;
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid,10000000);    /*Set a 10-second timeout*/

   if (hpib_status_wait(eid, WAIT_FOR_SRQ) == 0)
      service_routine();            /*SRQ is asserted; service the request*/
   else
      printf("Either a timeout or an error occurred\n");
}
```

Another solution is to periodically check the value of the SRQ line by calling `hpib_bus_status` as follows:

```
hpib_bus_status(eid, SRQ_STATUS);
```

where, as before, *eid* is the entity identifier for the open interface file and *SRQ_STATUS* indicates that you want the logical value of the SRQ line returned. `hpib_bus_status` returns 1 if SRQ is asserted, 0 if not, and −1 if an error occurred.

The most practical way to monitor SRQ is to set up an interrupt handler for that condition (see "Interrupts" section of Chapter 2).

Processing the Service Request

Once a device has asserted the SRQ line, it continues to assert the line until its request has been satisfied. How a service request is satisfied is device-dependent. Serial polling the device can provide the information as to what kind of service it requires.

Many devices are designed so that they automatically clear their SRQ output whenever they are serially polled. These devices treat the serial poll as an acknowledgement from the Active Controller that the request has been recognized and is being processed by the Active Controller.

If there is more than one device on the bus when SRQ is asserted, the Active Controller must first determine which device needs service before it can properly undertake any service related activity. There are two strategies for doing this:

- Serial poll each individual device in sequence until the one that is requesting service is found. This approach is reasonable if there are only a few devices on the bus.
- Conduct a parallel poll to locate the device requesting service. Normally each device (when capable) is programmed to respond on a given data line. However, up to 15 devices can reside on the bus which has only 8 data lines. Therefore it is sometimes necessary for more than one device to respond on a given line.

  If two or more devices are programmed to respond on a given parallel poll line and the parallel poll shows that line asserted, the Active Controller must then serially poll each device that is programmed to respond on that line until it determines which device is requesting service.

Thus, the Active Controller responds to SRQ by:

- Conducting a serial poll of individual devices on the bus,
- Conducting a parallel poll of return data lines to determine which line is being asserted, or
- Conducting a parallel poll to identify the asserted data line followed by a serial poll of devices programmed to assert that line when SRQ is being asserted by the same device.

HP-IB parallel and serial polls are conducted by the DIL subroutines `hpib_ppoll` and `hpib_spoll`, respectively. The next section explains how to use these subroutines.

Parallel Polling

The parallel poll is the fastest means of determining which device needs service when several devices are connected to the bus. Each device on the bus that is capable of responding to parallel polls can be programmed to respond to parallel polls by asserting a given data line, thus making it possible to obtain the status of several devices in a single operation. If a given device responds to the poll with a data line response (*I need service*), more information about its specific status can be obtained by conducting a subsequent serial poll of that device.

Configuring Parallel Poll Responses

HP-IB devices fall into three general categories:

1. Those devices that can be remotely programmed by the Active Controller to respond to a parallel poll in a certain way, The next several pages explain how to program these devices.
2. Devices whose parallel poll response is configured by internal hardware, whether by setting of configuration switches, or based on device bus address. A significant number of Hewlett-Packard products fall into this grouping. In general, they are HP-IB devices that support secondary commands such as SS/80 and CS/80 mass storage devices, CYPER printers, and Amigo protocol devices including several disc drives and printers. Some important information about these devices follows in the next few paragraphs.
3. Devices that are not capable of responding to parallel polls, so discussing their configuration is meaningless.

A number of operating rules have been established for devices in Category 2:

- No two devices can respond on the same data line. This means that only eight or fewer devices in this category can reside simultaneously on a given bus. If fewer than eight are present, data lines not used by these devices for parallel poll response can be shared among remaining devices on the bus if any are present.
- Each device in this category responds to a parallel poll on an assigned data line determined by the device's HP-IB address. Devices residing at HP-IB addresses 0 through 7 respond on data lines DI7 through DI0, respectively (note the reversed numbering sequencing).
- Devices in this category respond to parallel polls when they need service by driving the specified data line LOW to its ground-true logic state (the sense cannot be reversed to high-true).

Note also that some models of HP-IB devices can be switched between normal HP-IB operating mode and "Amigo" or "Secondary" mode (terminology varies as well as the implementation). Refer to the device installation and operating manuals for more information about how to configure the device for your application and to determine whether the device supports remote configuration

by the Active Controller, uses internal configuration, or does not support parallel poll.

To configure the parallel poll response for a given device by remote control from the Active Controller, use the HP-IB command sequences PARALLEL POLL CONFIGURE followed by PARALLEL POLL ENABLE. This combination of two commands tells all devices currently addressed as listeners to respond to any future parallel polls by asserting a specific data line with a specific logic level. Most devices that do not support remote configuration programming have internal configuration switches or jumpers that perform an equivalent function but which cannot be changed remotely by the Active Controller.

Devices that can be remotely configured can be programmed to respond with a logic 0 or logic 1 level on any one of eight data lines. Thus there are 16 possible combinations of lines and logic levels since there are two possible levels on each line and only one line can be asserted during a parallel poll. The PARALLEL POLL ENABLE command consists of an 8-bit byte whose bits are arranged as follows (the decimal equivalent value of the byte falls in the range of 96 through 111):

D7	D6	D5	D4	D3	D2	D1	D0	Decimal Range
0	1	1	0	L	X	X	X	96-111

where:

- The upper four bits are a fixed pattern of logical 0 (bits D7 and D4) and logical 1 (bits D6 and D5).
- Bit D3 (response logic level) determines whether data line D3 is to be asserted (driven to its ground-true state) or released (allowed to float to its high-false state) by the device when responding to a parallel poll if service is needed. If bit D3 is set (1), the device responding to the poll drives the data line low if service is needed. If D3 is not set (0), the device responding to the poll drives the data line low if service is *not* needed (bit value = 0). This bit is most commonly set to a value of 1.

- Bits D2, D1, and D0 are the 3-bit (value range 0 through 7) value representing which data line (D0 through D7 respectively) is to be used when responding to a parallel poll.

For example, to program a given device to respond to a parallel poll by placing a logic 1 on data line D0 if it needs service, use a PARALLEL POLL ENABLE command with a decimal value of 104 (binary 01101000).

The following code segment shows how to configure a device at bus address 5 to respond to a parallel poll by asserting data line D1 with a logic 1 if it needs service.

```
#include <dvio.h>
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid, MTA;
   char command[50];

   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   MTA = hpib_bus_status(eid, CURRENT_BUS_ADDRESS) + 64; /*compute MTA*/
   command[0] = MTA;          /*talk address of interface*/
   command[1] = 63;           /* the UNLISTEN command*/
   command[2] = 32 + 5;       /* the listen address for device at*/
                              /* address 5                       */
   command[3] = 5;            /* the PARALLEL POLL CONFIGURE command*/
   command[4] = 105;          /* the PARALLEL POLL ENABLE command*/
   hpib_send_cmnd(eid, command, 5);
}
```

Notice that the bit pattern for the PARALLEL POLL ENABLE command 105 (binary 01101001) used above is constructed as follows:

Bit Position	H	G	F	E	D	C	B	A
Bit Value	0	1	1	0	1	0	0	1

Where:

- Bits H through E (0110) indicate that this is a PARALLEL POLL ENABLE command.
- Bit D (1) indicates that the device respond with a 1 to request service.
- Bits C through A (001) indicate that the device should respond on D1.

When the computer interface is the Active Controller, it can configure its own parallel poll response by addressing itself as both talker and listener. However, the configuration is meaningless until the interface is no longer Active Controller because the Active Controller never responds to parallel polls.

Disabling Parallel Poll Responses

A device whose parallel poll response can be remotely configured by the Active Controller can also be disabled from responding.

To disable a device from responding to subsequent parallel polls, the Active Controller must first send a PARALLEL POLL CONFIGURE command followed by PARALLEL POLL DISABLE. This sequence disables all devices that are currently addressed to listen.

In the previous example a device at bus address 5 was configured to respond to parallel polls on data line D1. To disable parallel poll response on the same device, use a code segment similar to the following:

```
.
.
.
command[0] = MTA;          /*talk address of interface*/
command[1] = 63;           /* the UNLISTEN command*/
command[2] = 32 + 5;       /* the listen address for device at*/
                           /* address 5                       */
command[3] = 5;            /* the PARALLEL POLL CONFIGURE command*/
command[4] = 112;          /* the PARALLEL POLL DISABLE command*/
hpib_send_cmnd(eid, command, 5);
.
.
.
```

Conducting a Parallel Poll

Once parallel poll responses have been (remotely or internally) configured for all devices on the bus that are capable of responding to parallel polls, you can use `hpib_ppoll` to conduct a parallel poll on the bus, provided the computer is the current Active Controller.

The `hpib_ppoll` subroutine returns an integer whose least significant byte contains the 8-bit response to the parallel poll. Each device that is enabled to respond to a parallel poll places its status bit (service needed or not needed) on the data line defined by its current parallel poll response configuration. The subroutine returns −1 if an error occurs during the poll.

`hpib_ppoll` is invoked as follows:

```
hpib_ppoll(eid);
```

where *eid* is the entity identifier for the open interface file associated with the bus.

The following code segment shows how to interpret the byte returned by `hpib_ppoll`. Suppose a device at address 6 was previously configured to respond to a parallel poll by setting D0 to logic 1 (low) level if it needs service and a device at address 7 was configured to respond similarly on D1. Assuming that these are the only two devices capable of responding to a parallel poll, only the values of the 2 least significant bits of the integer returned by `hpib_ppoll` are of interest. This example code segment handles the results of the parallel poll, but does not include the code needed to handled the requested service.

```
#include <fcntl.h>
#include <errno.h>
main()
{
  int eid, status, byte;
  if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
  }
  io_reset(eid);
  io_timeout_ctl(eid, 10000000);

  if ((status = hpib_ppoll( eid)) == -1) /*conduct the parallel poll*/
  {
      printf("error taking ppoll\n");  /*if -1 returned then error occurred*/
      exit(1);
   }
   byte = status & 3;           /*set all but the least significant*/
                                /*2 bits to zero                   */
   switch (byte) {
                 case 0:        /*neither device is requesting service*/
                 .
                 .
                 .
                 break;
                 case 1:        /*device at address 6 wants service*/
                 .
                 .
                 .
                 break;
                 case 2:        /*device at address 7 wants service*/
                 .
                 .
                 .
                 break;
                 case 3:        /*both devices want service*/
                 .
                 .
                 .
```

```
                    break;
                }
    }
```

Errors During Parallel Polls

`hpib_ppoll` returns the value −1 if any one of the following error conditions are encountered:

- Timeout defined by `io_timeout_ctl` occurred before all devices responded.
- Computer's interface is not the Active Controller.
- Entity identifier *eid* does not refer to a raw HP-IB interface file.
- Entity identifier *eid* does not refer to an open file.
- A timeout occurs.

To find out which of these conditions caused the error, your program should check for the following values of *errno:*

errno Value	Error Condition
EBADF	*eid* does not refer to an open file.
ENOTTY	*eid* does not refer to a raw interface file.
EIO	Interface is not Active Controller. (EACCES on Series 800)
ETIMEDOUT	A timeout occurred. (EIO on Series 300)

Waiting For a Parallel Poll Response

Subroutine `hpib_wait_on_ppoll` allows you to wait for a specific parallel poll response from one or more devices. The effect of this is similar to using `hpib_status_wait` to wait for assertion of SRQ as discussed earlier. `hpib_wait_on_ppoll` provides a mechanism for waiting until a specific device requests service while `hpib_status_wait` only waits until any device requests service.

To call `hpib_wait_on_ppoll`, use the form:

```
hpib_wait_on_ppoll(eid, mask, sense);
```

where *eid* is the entity identifier for an open interface file, *mask* is an integer whose binary value identifies which parallel poll lines are to be monitored for a request, and *sense* is an integer whose binary value identifies which lines respond with an inverted logic sense (device responds with 0 when it wants service instead of the usual 1). `hpib_wait_on_ppoll` returns the response byte *XOR*ed with the *sense* value then *AND*ed with the *mask* value, unless an error occurs, in which case it returns −1.

Calculating the mask

`hpib_wait_on_ppoll` uses only the least significant byte of the *mask* integer, which means that the integer's remaining bytes can contain anything. For simplicity, the examples in this discussion set the upper bytes to zero.

The value for *mask* is determined as follows:

1. Decide which parallel poll lines (the 8 data lines labelled D0 through D7) are to be monitored for service requests.
2. Set up an 8-bit binary number where the bits associated with each line being monitored are set to 1 and all remaining bits are 0. (D0 is associated with the least significant bit of the binary number, and D7 with the most significant.)
3. Given the binary number from step 2, calculate its decimal value. The result is the correct value for *mask*.

For example, suppose that you want to wait for device A or device B to request service. You know that device A has been configured to respond on parallel poll line D0 and device B has been configured to respond on line D4. The correct binary value for *mask* is:

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	1	0	0	0	1

The decimal equivalent of this binary number is 17; the correct value for *mask*.

Consider a *mask* value of 0 which indicates that you do not want to wait for a request on any of the parallel poll lines. In such a case, a call to `hpib_wait_on_ppoll` using a *mask* of 0 is meaningless and has no effect.

Calculating the sense

The subroutine `hpib_wait_on_ppoll` also only looks at the least significant byte of the *sense* integer. For simplicity, the examples in this discussion set the upper bytes to zero.

The value for *sense* is determined as follows:

1. Decide which parallel poll lines (the 8 data lines) are to be monitored for service requests as discussed earlier.
2. Determine which of these lines will indicate a service request by a logic 0 response. This means that you must know the *sense* with which the associated devices are configured to respond to parallel polls.
3. Define an 8-bit binary number where the bits associated with the lines that use a 0 to indicate a service request are set to 1 and all of remaining bits are 0. (D0 is associated with the least significant bit of the binary number, and D7 with the most significant.)
4. Given the binary number from step 3, calculate its decimal value. The resulting value is the *sense* integer you should use with hpib_wait_on_ppoll.

Using the previous example for calculating the *mask* value, device A is configured to respond on line D0 with a 1 when it wants service, but device B requests service by placing a 0 on line D4. The binary value for *sense* is:

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	1	0	0	0	0

The decimal equivalent of this number is 16; the correct value for *sense*.

If all devices on the bus respond to parallel polls with a 1 to request service, the value for *sense* can always be 0, regardless of which parallel poll lines are being monitored. If, on the other hand, all of devices request service with a 0, the *sense* value can always be 255 (11111111 in binary). You need calculate a special value for *sense* only if various devices on the bus respond with dissimilar logic senses.

Example

Assume that you want to use `hpib_wait_on_ppoll` to wait for one of the four devices on a bus to request service where the bus is configured as follows:

Device	Bus Address	Parallel Poll Response Line	Requests Service with a:
A	5	D0	1
B	7	D1	0
C	9	D2	0
D	11	D3	1

Begin by calculating the mask value for `hpib_wait_on_ppoll`. Since responses can be expected on lines D0, D1, D2, and D3, the correct *mask* value is:

Binary:	Decimal:
0 0 0 0 1 1 1 1	15

The four devices on the bus use mixed (both ground- and high-true logic), the *sense* value must be determined. Devices responding on lines D1 and D2 use 0 to request service, so the *sense* value is:

Binary:	Decimal:
0 0 0 0 0 1 1 0	6

Now that the *mask* and *sense* values have been determined, the code segment that makes the call to `hpib_wait_on_ppoll` can be written:

```
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid;
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid,10000000);   /*Set a 10-second timeout*/

   if (hpib_wait_on_ppoll(eid, 15, 6) == -1)
      printf("either a timeout or error occurred\n");
   else
      service_routine();
}
```

In the code segment shown, `service_routine` is executed only if one of the four devices requests service during the parallel poll. `service_routine` should contain code segments to service all devices on the bus, either individually or as a group. See the appropriate hardware-specific appendix for any restrictions that may apply to your system.

Serial Polling

A sequential poll of individual devices on the bus is known as a **serial poll.** One entire status byte is returned by the polled device in response to a serial poll. This byte is called the *status byte message* and, depending on the device, may indicate an overload, a request for service, printer out of paper, or some other condition. The particular response of each device depends on the device.

Not all devices can respond to a serial poll. To find out whether a particular device can be serially polled, consult operating manuals for the device. Attempting to serially poll a device that cannot respond to the poll causes a timeout or suspends your program indefinitely.

The Active Controller cannot poll itself.

Unlike parallel poll responses, serial poll responses cannot be configured remotely by the Active Controller. Responses vary, depending on the type of device being polled. Refer to device manual for more information.

Conducting a Serial Poll

Subroutine `hpib_spoll` performs a serial poll on a specified device. It is called with the form:

```
hpib_spoll(eid, address);
```

where *eid* is the entity identifier for an open interface file and *address* is the bus address of the device being polled. The subroutine returns an integer, the lowest byte of which contains the status byte message (the serial poll response) from the addressed device. Only one device can be polled per call to `hpib_spoll`.

Although the status byte message supplied by the addressed device is device-dependent, bit D6 (of bits D0 through D7) always indicates whether or not the device is currently asserting SRQ. If SRQ is currently being asserted by the device, indicating that it needs service, be sure to handle the request properly because the serial poll also clears SRQ so that a subsequent poll will show no service request, whether or not the current request has been satisfied.

The following code segment shows how `hpib_spoll` can be used to determine whether a device at bus address 5 is requesting service. The determination is made by simply examining D6 which indicates whether SRQ is being asserted.

```
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid, status;
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid,100000);   /*Set a 0.1-second timeout*/

   if ((status = hpib_spoll(eid, 5)) == -1)     /*conduct serial poll*/
   {    printf("error during serial poll\n");
        exit(1);
   }
   if (status & 64)                  /*after setting every bit except D6*/
                                     /*to zero; if D6 is set the device*/
     service_routine();              /*is requesting service */
}
```

Errors During Serial Poll

If any of the following error conditions are encountered during a call to `hpib_spoll`, the subroutine returns −1:

- Addressed device did not respond to serial poll before the timeout limit defined by `io_timeout_ctl` was exceeded.
- Computer interface is not current Active Controller.
- Entity identifier *eid* does not refer to an HP-IB raw interface file.
- Entity identifier *eid* does not refer to an open file.
- Address is outside the range [0,30].
- The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid.*

To determine which of these conditions caused the error, your program should check for the following values of *errno:*

errno Value	Error Condition
EBADF	*eid* does not refer to an open file.
ENOTTY	*eid* does not refer to a raw interface file.
EIO	The interface was not the Active Controller. (EACCES on Series 800)
ETIMEDOUT	A timeout occurred. (EIO on Series 300)
EACCES	The interface associated with this *eid* was locked by another process and `O_NDELAY` was set for this *eid.*
EINVAL	Invalid bus address.

Passing Control

The subroutine `hpib_pass_ctl` can be used to pass control of the bus from the computer (which must be the current Active Controller) to a *Non-Active Controller*. A *Non-Active Controller* is a device capable of becoming Active Controller, which usually means it is another computer.

`hpib_pass_ctl` is called as follows:

```
hpib_pass_ctl(eid, address);
```

where *eid* is the entity identifier for an open interface file that is currently the Active Controller and *address* is the bus address of a Non-Active Controller. Upon completion, the Non-Active Controller becomes the new Active Controller and the local interface is a Non-Active Controller.

While `hpib_pass_ctl` can pass active control capability, it cannot pass system control capability.

What If Control Is Not Accepted?

Your program is not suspended if the Non-Active Controller that you address does not accept active control of the bus, but the computer still loses active control meaning that the bus no longer has an Active Controller. If this happens, the computer must use its position as System Controller to assume the role of Active Controller by executing `hpib_abort` (see System Controller Role section which follows) or `io_reset`.

No error is returned by `hpib_pass_ctl` if the device that you address does not accept active control, and there is no direct way to determine in advance whether a given device can accept active control. There is also no way for the computer, after initiating `hpib_pass_ctl`, to determine whether active control has been accepted. However, if the computer that has passed control immediately requests service after passing control and detects a timeout before the request is acknowledged, this indicates that active control may not have been accepted.

Errors While Passing Control

If any of the following errors are encountered, `hpib_pass_ctl` returns −1:

- Computer interface is not Active Controller.
- Entity identifier *eid* does not refer to an HP-IB raw interface file.
- Entity identifier *eid* does not refer to an open file.
- Address is outside the range [0,30].
- A timeout occurs.
- The interface associated with thie *eid* is locked by another process and `O_NDELAY` is set for this *eid.*

To find out which of these conditions caused the error, your program should check for the following values of *errno:*

errno Value	Error Condition
EBADF	*eid* does not refer to an open file.
ENOTTY	*eid* does not refer to a raw interface file.
EIO	Interface is not Active Controller.
EINVAL	Invalid bus address.
ETIMEDOUT	A timeout occurred (EIO on Series 300)
EACCES	The interface associated with thie *eid* was locked by another process and `O_NDELAY` was set for this *eid*

Controlling the ATN Line

On a Series 300, the subroutine `hpib_atn_ctl` can be used to control the ATN line on the HP-IB bus. This routine is particularly useful when setting up two non-active controllers for a data transfer.

`hpib_atn_ctl` is called as follows:

```
hpib_atn_ctl(eid, flag);
```

where *eid* is the entity identifier for an open interface file that is currently active controller and *flag* is either a *0* or a *1*. A *flag* value of *1* enables ATN; a value of *0* disables it.

Changing the Interface Bus Address

On a Series 300, the subroutine `hpib_address_ctl` can be used to programmatically change the bus address of an HP-IB interface card.

`hpib_address_ctl` is called as follows:

```
hpib_address_ctl(eid, ba);
```

where *eid* is the new bus address for the interface card. *ba* must be in the range 0-30.

System Controller Role

When the HP-IBs System Controller is first powered on or is reset, it assumes the role of Active Controller. Any given HP-IB bus can have only one System Controller. The System Controller cannot pass system control to any other controller (computer) on the bus. However, it can pass active control to another controller.

Determining System Controller

To determine whether your computer's HP-IB interface is the System Controller, use the `hpib_bus_status` subroutine which must be called as follows:

```
hpib_bus_status(eid, SYS_CONT_STATUS);
```

where *eid* is the entity identifier for an open interface file and `SYS_CONT_STATUS` indicates that you want to determine whether it is the System Controller. The subroutine returns 1 if it is the System Controller, 0 if not, and −1 if an error occurs.

The following code segment prints a message indicating whether the interface is System Controller:

```
#include <dvio.h>
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid, status;
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid, 1000000);

   if ((status = hpib_bus_status(eid, SYS_CONT_STATUS)) == -1)
       printf("Error occurred during bus status subroutine\n");
   else if (status == 1)
       printf("Interface is the System Controller\n");
   else
       printf("Interface is not the System Controller\n");
}
```

System Controller's Duties

The HP-IB System Controller has three major functions:

- It assumes the role of Active Controller at power-up and reset.
- It can cancel talkers and listeners from the bus and assume the role of Active Controller by executing `hpib_abort`.
- It can control the logic level of the remote enable line (REN) with `hpib_ren_ctl`.

hpib_abort

A call to `hpib_abort` performs the following actions:

- Terminates activity on the bus by pulsing the Interface Clear (IFC) line. This unaddresses all talkers and listeners on the bus.
- Sets the REN line so that devices on the bus will be placed in their remote state when addressed as listeners.
- Clears the ATN line if it was left set by the previous Active Controller.
- System Controller then becomes Active Controller.
- Returns all devices on the bus to their local state.

`hpib_abort` leaves the SRQ line unchanged, meaning that any device requesting service before `hpib_abort` is executed is still requesting service when the subroutine is finished.

To use `hpib_abort` on a particular HP-IB, the computer must be the System Controller of that bus. It does not have to be the Active Controller.

One situation where `hpib_abort` is useful is when the current Active Controller passes active control to another device, but the device does not accept active control (this can occur when the device addressed to receive control is not another controller). Consequently, the bus is left without any Active Controller, leaving the System Controller to assume that role by using `hpib_abort`.

`hpib_abort` is called as follows:

```
hpib_abort(eid);
```

where *eid* is the entity identifier for an open interface file.

hpib_ren_ctl

`hpib_ren_ctl` is used to enable or disable the REN line on the HP-IB. If the REN line is enabled, all devices capable of remote operation (meaning that they can interpret HP-IB commands) can be placed in their remote state by the Active Controller addressing them as talkers or listeners. When REN is disabled, all devices on the bus return to their local state and cannot be accessed remotely.

The REN line is enabled by default by the System Controller at power-up or reset. It is also enabled whenever the System Controller executes `hpib_abort`.

To use `hpib_ren_ctl` on a particular HP-IB, the computer must System Controller on that bus. It does not have to be the Active Controller.

`hpib_ren_ctl` is called as follows:

```
hpib_ren_ctl(eid, flag);
```

where *eid* is the file descriptor for an open interface file and *flag* is an integer. If *flag* is zero, the REN line is disabled. If it has any other value, REN is enabled.

Errors During hpib_abort and hpib_ren_ctl

If any of the following errors is encountered, `hpib_abort` and `hpib_ren_ctl` both return −1:

- Computer interface is not System Controller.
- Entity identifier *eid* does not refer to an HP-IB raw interface file.
- Entity identifier *eid* does not refer to an open file.

To determine which of these conditions caused the error, your program should check for the following values of *errno:*

errno Value	Error Condition
EBADF	*eid* does not refer to an open file.
ENOTTY	*eid* does not refer to a raw interface file.
EIO	Interface is not System Controller.

In addition, `hpib_abort` can return the following values for *errno:*

errno Value	Error Condition
ETIMEDOUT	A timeout occurred (EIO on Series 300)
EACCES	The interface associated with this *eid* was locked by another process and `O_NDELAY` was set for this *eid*

The Computer As a Non-Active Controller

Checking Controller Status

Subroutine `hpib_bus_status` is used to obtain information about the current status of the HP-IB interface card and the HP-IB, and can be used by any device on the bus, whether it is the current Active Controller or System Controller or not. `hpib_bus_status` is mentioned briefly in previous discussions about Active and System Controllers. The discussion that follows is a broader treatment of how the routine is used.

The call to `hpib_bus_status` has the form:

```
hpib_bus_status(eid, status_question);
```

where *eid* is the entity identifier for an open interface file and *status_question* is an integer that indicates what question you want answered. The value of *status_question* must be within the range of 0 through 7 where the relationship between value and the nature of the status inquiry are as follows:

Value	Status Question
REMOTE_STATUS	Is the interface in its remote state?
SRQ_STATUS	Are any devices currently requesting service? (Is SRQ asserted?)
NDAC_STATUS	Is there a listener that is not ready for data? (Is NDAC asserted?)
SYS_CONT_STATUS	Is the interface the current System Controller?
ACT_CONT_STATUS	Is the interface the current Active Controller?
TALKER_STATUS	Is the interface currently addressed as a talker?
LISTENER_STATUS	Is the interface currently addressed as a listener?
CURRENT_BUS_ADDRESS	What is the interface's bus address?

For all values of *status_question* except CURRENT_BUS_ADDRESS, hpib_bus_status returns *1* if the answer to the question is yes, or *0* if the answer is no. If the value of *status_question* is CURRENT_BUS_ADDRESS, hpib_bus_status returns the bus address of the computer's HP-IB interface. If the value of *status_question* is outside the allowable set of values, −1 is returned, indicating an error.

For example, to determine if your interface is a Non-Active Controller on the bus, use a calling sequence similar to the following code segment:

```
:
if ((status = hpib_bus_status(eid, ACT_CONT_STATUS)) == -1)
   printf("Error occurred while checking status\n");
else if (status == 0)
   printf("Computer is a Non-Active Controller\n");
else
   printf("Computer is the Active Controller\n");
:
```

Requesting Service

When your computer is a Non-Active Controller it can request service from the current Active Controller by asserting the SRQ line. This is done with the hpib_rqst_srvce routine which is called as follows:

```
hpib_rqst_srvce(eid, response);
```

where *eid* is the entity identifier for an open interface file and the lowest byte of *response* is the integer value of the 8-bit response that the computer gives if it is serially polled. The upper bytes of *response* are ignored by the hpib_rqst_srvce. Using the labels d0 through D7 for the data bus byte, bit D6 sets the SRQ line. The defined values for the remaining 7 bits varies, depending on the application. This section only discusses how to use D6 (integer value of 64) to set and clear the SRQ line.

To request service, invoke `hpib_rqst_srvce` as follows:

```
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid;

   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid, 10000000);
   hpib_rqst_srvce(eid, 64);    /*Bit 6 of serial poll response is set*/
                                /*and SRQ is asserted                 */
}
```

Note that by setting *response* to 64, the only information that the Active Controller receives when it serially polls your computer is that you are asserting the SRQ line. Therefore, other data bits in *response* must be set or cleared to indicate the type of service you are requesting, and the program controlling the current Active Controller must be capable of interpreting the data correctly before transfer of control between computers connected to the same bus can be handled in an orderly manner.

`hpib_rqst_srvce` returns *0* if it executes correctly or *-1* if an error occurred.

Once you have asserted SRQ, the line remains asserted until the Active Controller serially polls you or you call `hpib_rqst_srvce` again and clear bit 6 using a sequence such as `hpib_rqst_srvce(eid, 0)` . Once the serial poll response is configured, your computer's HP-IB interface responds automatically to any serial polls from the Active Controller.

A couple of notes of caution are in order here:

If another device on the bus is also asserting SRQ when your service request is detected by the current Active Controller, SRQ remains asserted, even after your service request is processed by the Active Controller. Thus, if you receive control of the bus before the requesting device is serviced, you must handle that device's service request correctly in order to maintain correct bus operation.

On the other hand, if you call `hpib_rqst_srvce` while you are Active Controller, the interface receives the service request sequence from the computer but does not place an SRQ on the bus as long as you are still Active Controller. However, if active control is passed to another controller on the bus, as soon as the interface changes to non-controller it immediately sets SRQ and readies the specified *response* data byte for the first serial poll from the new Active Controller.

When an Active Controller detects an asserted SRQ line, it usually conducts a parallel poll of devices on the bus to determine which one is requesting service. The next section discusses how to configure the HP-IB interface card for correct response to parallel polls.

When an HP-IB device responds to a parallel poll with an *I need service* message, the Active Controller then performs a serial poll to determine what type of service is required. If two or more devices are configured to respond to a parallel poll on a single data line and the Active Controller detects a service request on that line, the controller *must* perform a serial poll of all devices that respond on that line in order to determine which device is requesting service.

Errors While Requesting Service

If any of the following error conditions occurs, `hpib_rqst_srvce` returns −1:

- Entity identifier *eid* does not refer to an HP-IB raw interface file.
- Entity identifier *eid* does not refer to an open file.
- A timeout occurs.
- The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid*.

To determine which of these conditions caused the error, your program should check for the following values of *errno:*

errno Value	Error Condition
EBADF	*eid* does not refer to an open file.
ENOTTY	*eid* does not refer to a raw interface file.
ETIMEDOUT	A timeout occurred. (EIO on Series 300)
EACCES	The interface associated with this *eid* was locked by another process and `O_NDELAY` was set for this *eid.*

Responding to Parallel Polls

Before the HP-IB interface on your computer can respond correctly to a parallel poll from another Active Controller, the response must be configured on the interface. This can be programmed remotely by the Active Controller as discussed previously in the Active Controller section of this chapter, or locally using `hpib_card_ppoll_resp`.

To configure a parallel-poll response:

- Specify the logic sense of the response (i.e. whether a 1 means the device does or doesn't need service).
- Specify which data line the device responds on. Two or more devices can be configured to respond on a single line.

To locally configure response to parallel polls, call `hpib_card_ppoll_resp` as follows:

```
hpib_card_ppoll_resp(eid, response);
```

where *eid* is the entity identifier of an open interface file and *response* is an integer whose binary value configures the response.

Calculating the Response

The value for *response* is found by first forming an 8-bit binary number, then using the decimal equivalent of that number where the bits in the binary number are defined as follows:

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	S	P	P	P

where:

S sets the logic sense of the response. Thus, if *S* is 1, the device responds with a logic 1 in response to a parallel poll if it requires service. Likewise, if *S* is 0, the interface places a logic 0 on the assigned data line in response to a parallel poll if it requires service.

P is a 3-bit binary number (value range from 0 through 7) that specifies which of the eight available parallel poll response lines (D0-D7) is to be used when responding to a parallel poll.

Of course, this configuration capability is possible only on those interfaces that support it. Refer to the appropriate appendix for more information about specific systems.

Limitations of hpib_card_ppoll_resp

Hardware limitations on certain devices restrict the use of `hpib_card_ppoll_resp` to configure parallel poll responses. Refer to the appendix related to your system to determine whether any restrictions apply. If there are restrictions on your system, you may find it easier to configure the interface parallel poll response remotely from another Active Controller. Don't forget that the Active Controller can configure its own response, but the response remains dormant until control is passed to another device.

Error Conditions

If any of the following error conditions is encountered by `hpib_card_ppoll_resp`, it returns −1:

- Entity identifier *eid* does not refer to an HP-IB raw interface file.
- Entity identifier *eid* does not refer to an open file.
- A timeout occurs.
- The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid.*
- The device cannot respond on the line number specified by *response.*

To find out which of these conditions caused the error, your program should check for the following values of *errno:*

errno Value	Error Condition
EBADF	*eid* does not refer to an open file.
ENOTTY	*eid* does not refer to a raw interface file.
ETIMEDOUT	A timeout occurred. (EIO on Series 300)
EACCES	The interface associated with this *eid* was locked by another process and `O_NDELAY` was set for this *eid.*
EINVAL	The device cannot respond on the line number stecified by *response.*

hpib_ppoll_resp_ctl

The subroutine `hpib_ppoll_resp_ctl` is used to control how the HP-IB interface will respond to the next parallel poll:

- Assert the assigned data line with the previously configured logic sense if service is required, or
- Place the opposite logic level on the same data line if the interface does not need to interact with the Active Controller.

Parallel poll response is set as follows:

```
hpib_ppoll_resp_ctl(eid, response_value);
```

where *eid* is the entity identifier of an open interface file and *response_value* is an integer that indicates how the interface is to respond to the next parallel poll. If *response_value* is non-zero, the computer will respond to the next parallel poll with a request for service. If *response_value* is zero, the next response will be set to indicate that no service is needed.

Disabling Parallel-Poll Response

You can also disable responses to parallel polls from another Active Controller by using `hpib_card_ppoll_resp` by setting bit D4 in the routine's *response* value. When D4 is 0 the interface is set to respond to parallel polls with a service-needed logic level. When D4 is 1, the interface responds to parallel polls with the opposite (service not needed) level. Thus, a flag value of 16 disables the need-service response.

For example, the subroutine call:

```
⋮
hpib_card_ppoll_resp(eid, 16);   /*disable parallel poll response*/
⋮
```

disables the HP-IB interface associated with entity identifier *eid* from responding to any parallel polls with a service request.

Accepting Active Control

Any Active Controller can pass control to any other device on the bus, but only a Non-Active Controller can accept control. When an Active Controller interface passes control to a Non-Active Controller interface, the Non-Active interface automatically accepts control and the former Active Controller becomes a Non-Active Controller. However, when this transfer of control occurs, the interface receiving control does not automatically notify the computer that control has been received unless the necessary interrupts have been set up by the application program by use of subroutines `hpib_bus_status`, `hpib_status_wait`, and `io_on_interrupt`.

`hpib_status_wait` has been mentioned in previous discussions about the Active Controller and System Controller. The following discussion provides a look at its uses.

Call `hpib_status_wait` as follows:

```
hpib_status_wait(eid, status);
```

where *eid* is the entity identifier for an open interface file and *status* is an integer indicating what condition you want to wait for. The following values for *status* are defined:

Value	Condition
WAIT_FOR_SRQ	Wait until the SRQ line is asserted
WAIT_FOR_CONTROL	Wait until this computer is the Active Controller
WAIT_FOR_TALKER	Wait until this computer is addressed as a talker
WAIT_FOR_LISTENER	Wait until this computer is addressed as a listener

Suppose you are designing a program to handle a situation where the current Active Controller is programmed such that when your computer requests service, it passes active control to you. The following code segment shows how you can program your computer to request service then wait until it becomes the new Active Controller before it continues.

```
include <dvio.h>
include <fcntl.h>
include <errno.h>
ain()

  int eid;

  if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
       printf("open failed, errno = %d\n", errno);
       exit(2);
  }
  io_reset(eid);
  io_timeout_ctl(eid, 10000000);

  if (hpib_rqst_srvce(eid, 64) == -1)   /*set SRQ line to request service*/
  {
     printf("Error while requesting service\n");
     exit(1);
  }

  if (hpib_status_wait(eid, WAIT_FOR_CONTROL) == -1) /*wait until Active Controller*/
  {
     printf("Error while waiting for status\n");
     exit(1);
  }
  .
  .
  .          /*Computer is now the Active Controller*/
```

Note that for `hpib_status_wait` to have returned −1 (caused by an unexpected timeout), a timeout value would have to have been set using `io_timeout_ctl` after the interface file was opened. Since this example does not contain a call to `io_timeout_ctl`, no timeout occurs.

Errors While Waiting on Status

`hpib_status_wait` returns −1 indicating an error if any of the following error conditions are encountered:

- A timeout occurred before the condition the routine was waiting for became true.
- The value specified by *status* is undefined.
- Entity identifier *eid* does not refer to a raw HP-IB interface file.
- Entity identifier *eid* does not refer to an open file.
- The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid.*
- The device is active controller and *status* specifies WAIT_FOR_TALKER or WAIT_FOR_LISTENER. (Series 300 only)

To find out which of these conditions caused the error, your program should check for the following values of *errno:*

errno Value	Error Condition
EBADF	*eid* does not refer to an open file.
ENOTTY	*eid* does not refer to a raw HP-IB interface file.
EINVAL	*status* contains an invalid value.
ETIMEDOUT	The specified condition did not become true before a timeout occurred. (EIO on Series 300)
EACCES	The interface associated with this *eid* was locked by another process and `O_NDELAY` was set for this *eid*
EIO	The device is active controller and *status* specifies WAIT_FOR_TALKER or WAIT_FOR_LISTENER (Series 300 only).

Determining When You Are Addressed

As a Non-Active Controller you may be addressed at any time by the current Active Controller to become a bus talker or listener for data transfer. The DIL routines `hpib_bus_status`, `hpib_status_wait`, and `io_ on_interrupt` are used to determine that the interface is currently being addressed and provide proper notification to the controlling program.

The following code segment determines whether the interface is currently addressed as a bus talker:

```
#include <dvio.h>
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid;

   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   if (hpib_bus_status(eid, TALKER_STATUS) == 1)
   {
      printf("the interface is addressed as a talker\n");
      write(eid, "data message", 12);   /*do the expected data transfer*/
   }
   else
      printf("the interface is not addressed as a talker\n");
}
```

In the above call to `hpib_bus_status`, *eid* is the entity identifier for the interface device file and TALKER_STATUS indicates that you want to know if it is addressed to talk. The routine returns the value 1 if the answer is yes; 0 if not.

To determine whether the interface is currently addressed as a bus listener use the following:

```
:
if (hpib_bus_status(eid, LISTENER_STATUS) == 1)
{
   printf("the interface is addressed as a listener\n");
   read(eid, buffer, 12);            /*do the data transfer*/
}
else
   printf("the interface is not addressed as a listener\n");
:
```

If you need to wait until the interface is addressed as either a talker or listener, then handle an appropriate data transfer, use the DIL subroutine `hpib_status_wait`, specifying both the entity identifier of the interface device file and the bus condition that is being used to terminate the wait.

```
hpib_status_wait(eid, condition);
```

As with `hpib_bus_status`, a condition value of WAIT_FOR_TALKER causes the program to wait until the interface is addressed as a talker. With a condition value of WAIT_FOR_LISTENER the routine waits until it is addressed to listen. The maximum time that the routine can wait for the specified condition is controlled by the timeout value that was previously set for the entity identifier using subroutine `io_timeout_ctl` (discussed in Chapter 2). `hpib_status_wait` returns 0 if the wait condition terminated the wait or `-1` if a timeout or other error occurred before the wait condition was fulfilled.

In the following example code segment, the program waits for the interface to become a bus listener, then reads a 50-byte message.

```
#include <dvio.h>
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid, len;
   char buffer[51];                    /*storage for message*/
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid, 5000000);       /*5-second timeout*/

   if (hpib_status_wait(eid, WAIT_FOR_LISTENER) == -1)
   {
       printf("Either a timeout or an error occurred\n");
       exit(1);
   }

   len = read(eid, buffer, 50);            /*read data into buffer*/
   buffer[ len ] = '\0';
   printf("Message is: %s", buffer);       /*print data message*/
}
```

Note that in this example a timeout value is set for the interface file's entity identifier so that the program cannot hang indefinitely while waiting for the interface to be addressed as a bus listener should the condition not occur as expected.

The following example illustrates how to use `io_on_interrupt` to set up an interrupt handler to handle a data transfer:

```
#include <dvio.h>
#include <fcntl.h>
#include <errno.h>
char buffer[50];
main()
{
   int handler();
   int eid;
   struct interrupt_struct cause_vec;

   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);

   cause_vec.cause = LTN;
   io_on_interrupt(eid, &cause_vec, handler);
   .
   .
   .
}
handler(eid, cause_vec)
int eid;
struct interrupt_struct *cause_vec;
{
   if (cause_vec->cause == LTN)
       read(eid, buffer, 50);
}
```

Combining I/O Operations into a Single Subroutine Call

`hpib_io` is a high-level DIL subroutine that provides a mechanism for conveniently collecting a series of HP-IB I/O operations in a data structure then using a simple subroutine call to `hpib_io` to handle interface and bus management operations. This feature eliminates the need for using several long tedious series of subroutine calls to `io_lock`, `hpib_send_cmnd`, *read, write,* and `io_unlock` and makes these operations atomic on the Seris 300.

A call to `hpib_io` has the form:

```
#include <dvio.h>
main()
{
   int eid;
   struct  iodetail *iovec;
   int iolen;
   .
   .
   .
   hpib_io(eid, iovec, iolen);
   .
   .
   .
}
```

where *eid* is the entity identifier of an open interface file, `iovec` is a pointer to an array of I/O operation structures, and `iolen` is the number of structures in the array. The name of the template for the I/O operation structures is `iodetail` and it is defined in the include file *dvio.h.*

Iodetail: The I/O Operation Template

The form of the `iodetail` structure that holds I/O operations is:

```
struct iodetail {
      char mode;
      char terminator;
      int count;
      char *buf;
     };
```

Where the components in structure `iodetail` have the following meanings:

mode	Describes what kind of I/O operation the structure contains.
terminator	Specifies whether or not there is a read termination character for the I/O operation, and if so it specifies the value.
count	How many bytes are to be transferred during the I/O operation.
buf	A pointer to an array containing the bytes of data to be transferred.

Components of a particular `iodetail` structure are referenced with:

```
iovec->component
```

where `iovec` is a pointer to an array of `iodetail` structures and `component` is either `mode`, `terminator`, `count`, or `buf`.

The Mode Component

The `mode` describes what type of I/O operation is to be performed on the data pointed to by the `buf` component. To determine its value, *OR* appropriate constants from a set defined in the include file *dvio.h*. You can choose from the following constants:

Table 3-3.

Name	Description
HPIBREAD	Perform a read operation and place the data into the accompanying buffer pointed to by *buf*. Can be by itself or *OR*-ed with HPIBCHAR.
HPIBWRITE	Perform a write operation using the data in the accompanying buffer pointed to by *buf*. Can be by itself or *OR*-ed with either HPIBATN or HPIBEOI but not both.
HPIBATN	If you are performing a write operation, the data is placed on the bus with ATN asserted (you are sending a bus command). It only has effect if you also specify HPIBWRITE.
HPIBEOI	If you are performing a write operation, the EOI line is asserted when the last byte of data is sent. It only has effect if you also specify HPIBWRITE.
HPIBCHAR	If you are performing a read operation, the transfer is halted when the *terminator* component value of the `iodetail` structure is read. The *terminator* component only has effect if you *OR* HPIBCHAR and HPIBREAD. The HPIBCHAR constant only has effect if also specify HPIBREAD.

Note

When you construct *mode,* you must use either HPIBREAD or HPIBWRITE, but not both. Optionally, you can *OR* one of the other three constants with either HPIBREAD or HPIBWRITE, but they are not required. HPIBCHAR has effect only when it is *OR*ed with HPIBREAD, while HPIBATN and HPIBEOI have effect only when they are *OR*ed with HPIBWRITE (but not both at the same time).

The *mode* component allows you to specify conditions under which an I/O operation terminates. All I/O operations terminate when the maximum number of bytes specified by the *count* component of the `iodetail` structure is reached. However, additional termination conditions are possible:

- If you specify HPIBREAD and HPIBCHAR: detection of the termination character defined by the *terminator* component also causes termination.
- If you specify HPIBWRITE and HPIBEOI: when the count value is reached EOI is asserted at the time that the last byte of data is sent (unless you also specify HPIBATN).

To illustrate, assume that `iovec` points to an `iodetail` structure that you are building and you want the structure to send several HP-IB commands. The *mode* component of the structure is assigned the necessary value as follows:

```
iovec->mode = HPIBWRITE | HPIBATN;
```

The Terminator Component

The *terminator* component of the `iodetail` structure specifies a character that causes the termination of a read operation when it is detected. The *terminator* only has effect if HPIBREAD | HPIBCHAR is specified as the structure's associated *mode* component.

Assign a value to the *terminator* component in the structure pointed to by `iovec` with:

```
iovec->terminator = value;
```

For example, to define the ASCII period character (.) the termination character, use the statement:

```
iovec->terminator = '.';
```

The Count Component

count is an integer that defines the maximum number of bytes to be transferred during the structure's I/O operation. Reading or writing always terminates when this value is reached, but additional termination conditions can be set up using the structure's associated *mode* component.

To set a maximum number of bytes for a structure's data transfer:

```
iovec->count = max_value;
```

where `iovec` is a pointer to the structure and *max_value* is an integer.

The Buf Component

The *buf* component points to a character array where data is to be stored from a read operation (HPIBREAD) or a character array containing data to be written to during a write operation (HPIBWRITE).

Note The value of a structure's *count* component should *never* exceed the size of the array. If this restriction is violated, unpredictable results and/or data loss are likely.

One way to store a message in the *buf* array is:

```
iovec->buf = "data message";
```

Allocating Space

Before building `iodetail` structures for I/O operations, storage space in memory must be allocated. The easiest way to do this (if you are programming in C) is to write a routine that allocates space for *n* `iodetail` structures and returns a pointer to the first one.

Here is a sample code segment for such a routine, `io_alloc`:

```
#include <dvio.h>
struct iodetail *io_alloc(n)
int n;
{
  char *malloc();
  return((struct iodetail *) malloc(sizeof(struct iodetail) * n));
}
```

Refer to the *HP-UX Reference* for a description of *malloc*(3C).

For example, to use `io_alloc` to allocate memory space for 10 `iodetail` structures your program should contain the statements:

```
struct iodetail *iovec;     /*define an iodetail pointer*/
iovec = io_alloc(10);      /*allocate space for 10 iodetail structures*/
```

Example

Assume the HP-IB interface is Active Controller and located at HP-IB address 30. A data message is to be sent to a device at HP-IB address 7 then a subsequent message is to be received from the same device by use of the `hpib_io` subroutine. Such a sequence requires four `iodetail` structures:

1. The first structure configures the bus so that the interface is the talker and the device at address 7 is the listener.
2. The second structure sends the data message from the interface to the device.
3. The third structure configures the bus so that the device at address 7 is the talker and the interface is the listener.
4. The fourth structure receives the data message from the device.

The following code segment illustrates how the four structures can be built and implemented.

```
#include <fcntl.h>
#include <errno.h>
#include <dvio.h>              /*contains definitions for iodetail*/
struct iodetail *io_alloc(n)
int n;
{
    char *malloc();
    return ((struct iodetail *) malloc(sizeof (struct iodetail) *n));
}

main()
{
   extern int errno;
   int eid;
   char buffer[4][12];
   struct iodetail *iovec, *temp;  /*2 pointers to iodetail structures*/

/*Allocate space for 4 iodetail structures*/
   iovec = io_alloc(4);           /* use the routine described earlier */
   temp = iovec;
```

```
/*Build structure 1 -- Configuring the bus*/
   temp->mode = HPIBWRITE | HPIBATN; /*you want to send commands*/
   strcpy(buffer[0],"?^'"); /*address computer to talk; bus address to listen*/
   temp->buf = buffer[0];
   temp->count = strlen(buffer[0]);

/*Build structure 2 -- Sending the data message*/
   temp++;     /*use temp pointer so iovec keeps pointing to*/
               /*first structure but temp now points to next one*/

   temp->mode = HPIBWRITE | HPIBEOI; /*assert EOI when the transfer is
complete*/
   strcpy(buffer[1],"data message");
   temp->buf = buffer[1];
   temp->count = strlen(buffer[1]);

/*Build structure 3 -- Configuring the bus*/
   temp++;                              /*increment structure
pointer*/
   temp->mode = HPIBWRITE | HPIBATN;    /*to send commands*/
   strcpy(buffer[2],"?G>");
   temp->buf = buffer[2];
   temp->count = strlen(buffer[2]);

/*Build structure 4 -- Receiving data message*/
   temp++;                /*increment structure pointer*/
   temp->mode = HPIBREAD; /*read data until count limit is reached*/
   temp->count = 10;      /*accept message up to 10-bytes in length*/
   temp->buf = buffer[3];

/*Implement the I/O operations stored in the iodetail structures*/
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid, 10000000);

   if (hpib_io(eid, iovec, 4) == -1)
   {
       printf ("hpib_io failed\n");
       printf ("errno = %d\n",errno);
       exit(1);
   }
```

```
/*Print data message received from the device. Note that temp still*/
/*points to the last iodetail structure, the one that did the read */

   printf("%s", temp->buf);
}
```

One comment about the C language: Subroutine parameters are passed by value; not by reference. This means that after `hpib_io` is executed, the `iovec` parameter still points to the first `iodetail` structure, just as it did before the subroutine was executed. Thus, another way to print out the data message that was read into the *buf* component of the fourth `iodetail` structure in the example above is:

```
printf("%s", (iovec + 3)->buf);
```

Locating Errors in Buffered I/O Operations

If all I/O operations specified in the array of `iodetail` structures complete successfully, `hpib_io` returns 0 and updates the *count* component of each structure to reflect the actual number of bytes read or written.

If an error occurs during one of the I/O operations, `hpib_io` immediately returns a −1 indicating the error. To determine which `iodetail` structure operation was associated with the error, examine the structures' *count* components. When `hpib_io` encounters an error, it updates the *count* component of the structure that caused the error to −1. Thus, once you have located a structure with a count of −1, you know that all previous structures were completed successfully and all of the structures after it were not executed at all.

For example, suppose an array of ten `iodetail` structures has been built to execute a sequence of I/O operations. The following code segment executes the operations then checks for errors. If an error occurs, the number of the structure that caused it (the first structure in the array is number 1) is printed.

```
#include <fcntl.h>
#include <errno.h>
#include <dvio.h>
main()
{
   int FOUND, number, eid;
   struct  iodetail *iovec, *temp;
   :
   /*space is allocated for the 10 structures then they are*/
   /*built. "Iovec" is left pointing to the first structure*/
   :
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
 printf("open failed, errno = %d\n", errno);
 exit(2);
   }
   io_reset(eid);
   io_timeout_ctl(eid, 10000000);

   if (hpib_io(eid, iovec, 10) == -1) /*execute the operations.  If a -1*/
                                      /*is returned, an error occurred*/
   {
       number = 1;              /*initialize counter*/
       FOUND  = 0;              /*initialize Boolean flag*/
       temp = iovec;            /*set temporary pointer to first structure*/
       while (number <= 10 &&  FOUND != 1)
          if (temp->count == -1)     /*found structure that caused error*/
             FOUND = 1;
          else
             {
             temp++;     /*move pointer to next structure*/
             number++;              /*increment counter*/
             }
       if (FOUND == 1)
          printf("Structure number %d caused error", number);
       else
          printf("Error but couldn't find structure that caused it\n");
   }
   else
       printf("No error occurred during execution of hpib_io\n");
```

```
}
```

4

Controlling the GPIO Interface

This chapter briefly describes how to configure the GPIO interface before accessing it from a program by use of DIL subroutines. It then discusses the capabilities and limitations of DIL subroutines when controlling the GPIO interface.

Interface Configuration

The Series 300 GPIO interface is configured by setting several switches on the interface card. The interface installation manual explains how each switch is used and how it should be configured. Configurable functions associated with these switches include:

- Data logic sense,
- Data handshake mode,
- Input data clock source.

Set the configuration switches according to the directions found in the GPIO interface installation manual.

Creating the GPIO Interface File

After setting the necessary switches on your GPIO interface, install the card in the computer then create an interface file for it as explained in Chapter 2. An appropriate interface file must be created before the interface can be accessed from HP-UX.

Interface Control Limitations

Device I/O Library (DIL) subroutines provide a means for using a GPIO interface to communicate with devices that are not supported on your HP-UX system. However, they do not provide full control of the interface, so you are faced with the following limitations:

- There is no direct access to interface handshake lines: Peripheral Control (PCTL) line, Peripheral Flag (PFLG) line, and Input/Output (I/$\overline{O}$) line.
- You cannot read the value of the Peripheral Status line (PSTS) directly.

Using DIL Subroutines

Several DIL subroutines can be used to control the GPIO interface. They are divided into two groups:

- General-purpose routines usable with both HP-IB and GPIO interfaces,
- GPIO routines: routines specifically designed for use with a GPIO interface.

General-purpose routines are listed and described in detail in Chapter 2. They are used in this chapter to illustrate various aspects of controlling GPIO interfaces from an HP-UX process.

Two DIL routines used exclusively with GPIO interfaces:

- `gpio_get_status`
- `gpio_set_ctl`.

The GPIO interface has four special-purpose lines that are used in various ways, depending on the needs of the device connected to the interface. Two incoming lines, $\overline{\text{STI0}}$ and $\overline{\text{STI1}}$, are driven by the peripheral device and are usually used to provide device status information. Two outgoing lines, $\overline{\text{CTL0}}$ and $\overline{\text{CTL1}}$ are driven by the computer, usually to control the device.

The subroutines `gpio_get_status` and `gpio_set_ctl` are used to access these four special-purpose lines. `gpio_get_status` reads $\overline{\text{STI0}}$ and $\overline{\text{STI1}}$, and `gpio_set_ctl` sets the values of $\overline{\text{CTL0}}$ and $\overline{\text{CTL1}}$. Both routines are described later in this chapter in the section *Using Status and Control Lines.*

By using the DIL general-purpose routines and these two GPIO-specific routines you can:

- Reset the interface,
- Perform data transfers,
- Use the interface's 4 special purpose lines,
- Control the data path width and data transfer speed,
- Set a timeout for data transfers,
- Set a read termination character,
- Get the termination reason,
- Set up the interrupts,
- Enable or disable interrupts.

Resetting the Interface

The interface should always be reset before it is used, to ensure that it is in a known state. All interfaces are automatically reset when the computer is powered up, but you can also reset them from your I/O process by using the `io_reset` subroutine. For example, the following code segment resets a GPIO interface:

```
int  eid;                              /*entity identifier*/
eid = open( "/dev/raw_gpio", O_RDWR); /*open GPIO interface file*/
io_reset(eid);                         /*reset the interface*/
```

This has the following effect:

- Peripheral Reset line (PRESET) is pulsed low,
- PCTL line is placed in the clear state,
- If the DOUT CLEAR jumper is installed, the Data Out lines are all cleared (set to logical 0),
- Interrupts from the controlled interface are disabled on Series 300 systems.

Lines that are left unchanged are:

- $\overline{\text{CTL0}}$ and $\overline{\text{CTL1}}$ output lines,
- I/$\overline{\text{O}}$ line,
- Data Out lines if the DOUT CLEAR jumper is not installed.

Performing Data Transfers

The `read` and `write` system calls are used to transfer ASCII data to and from the GPIO interface. The following code segment illustrates how to use these routines to write 16 bytes to the interface, then read 16 bytes back in.

```
#include <fcntl.h>
#include <errno.h>
main()
{
   int  eid;                                /*entity identifier*/
   char read_buffer[16], write_buffer[16]; /*buffers to hold data*/

   if ((eid = open("/dev/raw_gpio", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   io_reset(eid);

   write_buffer = "message to write";       /*data message to send*/
   write( eid,write_buffer, 16);            /*send message*/
   read( eid, read_buffer, 16);             /*receive message*/
   printf("%s", read_buffer);               /*print received message*/
}
```

Using Status and Control Lines

Four special-purpose (status and control) signal lines are available for a variety of uses. Two of the lines are for output ($\overline{\text{CTL0}}$ and $\overline{\text{CTL1}}$), and two are for input ($\overline{\text{STI0}}$ and $\overline{\text{STI1}}$). The routine `gpio_set_ctl` allows you to control the values of $\overline{\text{CTL0}}$ and $\overline{\text{CTL1}}$, while the routine `gpio_get_status` allows you to read the values of $\overline{\text{STI0}}$ and $\overline{\text{STI1}}$.

Driving $\overline{\text{CTL0}}$ and $\overline{\text{CTL1}}$

The call to `gpio_set_ctl` has the following form:

```
gpio_set_ctl(eid, value);
```

where *eid* is the entity identifier for an open GPIO interface file and *value* is an integer whose least significant two bits are mapped to $\overline{\text{CTL0}}$ (bit 0) and $\overline{\text{CTL1}}$ (bit 1). Both $\overline{\text{CTL0}}$ and $\overline{\text{CTL1}}$ are ground-true logic meaning that they are at a logic LOW level when asserted. This logic polarity cannot be changed. Logic sense of the two lines is related to *value* as follows:

- If *value* =0: $\overline{\text{CTL0}}$ and $\overline{\text{CTL1}}$ both false (HIGH logic level)
- If *value* =1: $\overline{\text{CTL0}}$ true (LOW logic level) and $\overline{\text{CTL1}}$ false (HIGH logic level)
- If *value* =2: $\overline{\text{CTL0}}$ false (HIGH logic level) and $\overline{\text{CTL1}}$ true (LOW logic level)
- If *value* =3: $\overline{\text{CTL0}}$ and $\overline{\text{CTL1}}$ both true (LOW logic level)

This example code segment asserts both lines, setting them at a logic LOW level:

```
int eid;                /*entity identifier*/
eid = open("/dev/raw_gpio", O_RDWR);  /*open interface file*/
gpio_set_ctl( eid, 3);              /*assert CTL0 and CTL1*/
```

To set both lines to a logic HIGH level, call `gpio_set_ctl` as follows:

```
gpio_set_ctl( eid, 0);
```

Reading STI0 and STI1

The call to `gpio_get_status` has the following form:

```
int  eid, value;
value = gpio_get_status(eid);
```

where *eid* is the entity identifier for an open GPIO interface file. `gpio_get_status` returns an integer whose least significant two bits are the values of $\overline{\text{STI0}}$ and $\overline{\text{STI1}}$.

Like $\overline{\text{CTL0}}$ and $\overline{\text{CTL1}}$, $\overline{\text{STI0}}$ and $\overline{\text{STI1}}$ are ground-true logic meaning they are at a logic LOW level when asserted. Thus the *value* returned by `gpio_get_status` is as follows (be sure to AND *value* with 3 to clear upper bits before testing):

- If *value* =0: $\overline{\text{STI0}}$ and $\overline{\text{STI1}}$ both false (HIGH logic level)
- If *value* =1: $\overline{\text{STI0}}$ true (LOW logic level) and $\overline{\text{STI1}}$ false (HIGH logic level)
- If *value* =2: $\overline{\text{STI0}}$ false (HIGH logic level) and $\overline{\text{STI1}}$ true (LOW logic level)
- If *value* =3: $\overline{\text{STI0}}$ and $\overline{\text{STI1}}$ both true (LOW logic level)

To illustrate:

```
int eid;                /*entity identifier*/
int value, bits;
eid = open("/dev/raw_gpio", O_RDWR);   /*open interface file*/
value = gpio_get_status(eid);         /*look at STI0 and STI1*/
bits = value & 03 /*clear all but the 2 least significant bits*/
if (bits == 3)    /*and see if they are both set*/
   :
 /*insert code that handles case when both STI0 and STI1 are asserted*/
else if (bits == 1)                   /*only STI0 is asserted*/
   :
 /*insert code that handles case when STI0 is asserted*/
   :
else if (bits == 2)                   /*only STI1 is asserted*/
   :
 /*insert code that handles case when STI1 is asserted*/
   :
else                                  /*neither are asserted*/
   :
 /*insert code that handles case when neither STI0 nor STI1 is asserted*/
```

Controlling Data Path Width

DIL subroutine `io_width_ctl` is used to specify 8-bit or 16-bit data path widths for the GPIO interface. The call has the following form:

```
io_width_ctl( eid, width);
```

where *eid* is the entity identifier for an open GPIO interface file and *width* is either 8 or 16. If any other *width* value is specified, `io_width_ctl` returns -1 and sets *errno* to *EINVAL*. The GPIO interface is set to a default 8-bit path width when the interface file is opened.

The following code segment illustrates data transfers using a 16-bit data path width.

```
int eid;

eid = open("/dev/raw_gpio", O_RDWR);        /*open the interface file*/
io_width_ctl( eid, 16);                  /*set path width to 16 bits*/
write( eid, "data message", 12);         /*perform data transfer*/
```

Since the interface data path width is 16 bits, 2 ASCII characters are transferred during each handshake cycle. In the first 16-bit transfer, *d* is sent in the upper byte and *a* is sent in the lower. The actual logic sense (ground-true or high-true) of the GPIO data output lines depends on how the lines were configured during interface card installation.

Controlling Transfer Speed

You can request a minimum speed for the data transfer across a GPIO interface by issuing a call to `io_speed_ctl`. Your system rounds the specified speed up to the nearest defined speed. If you specify a speed that is faster than your system allows, the highest available speed is used instead. Refer to Chapter 2 for more information about `io_speed_ctl`.

GPIO Timeouts

If a non-zero timeout limit has been established for a given *eid* and that limit is exceeded during a data transfer request, an error condition results. When the subroutine handling the transfer detects the timeout error, it returns −1 and sets *errno* to ETIMEDOUT (EIO on Series 300). When a timeout error occurs, use `io_reset` to reset the GPIO interface before attempting another transfer.

Burst Transfers

Series 300 systems support high-speed burst I/O on HP-IB and GPIO interfaces. The call to `io_burst` is structured as follows:

```
io_burst(eid, flag)
```

`io_burst` controls the data path between computer memory and the HP-IB or GPIO interface. If *flag* = 0, all data is handled through kernel calls with the normal associated overhead. If *flag* is non-zero, burst mode locks the interface and data is transferred directly between memory and the I/O mapped interface until the transfer is completed. Burst mode yields substantial improvement in efficiency when handling small amounts of data or high-speed data acquisition.

Read Terminations

Determining Why a Read Operation Terminated

Subroutine `io_get_term_reason`, described in Chapter 2, is used to determine why the last read performed on a particular *eid* terminated. Possible reasons include:

- The requested number of bytes were read
- A specified read termination character was seen
- A assertion of the PSTS line was seen
- Some abnormal condition occurred, such as an I/O timeout.

Specifying a Read Termination Pattern

Chapter 2 describes subroutine `io_eol_ctl` which is used to specify a character or string of characters (called a read termination pattern) that, when encountered during a read, terminates the read operation currently underway on a particular GPIO interface file *eid*.

Interrupts

Subroutines `io_on_interrupt` and `io_interrupt_ctl` are described in Chapter 2. They are used to set up and control interrupt handlers for the GPIO status line or for a particular GPIO interface file *eid*.

A

Series 300 Dependencies

The following information, specific to Series 300 computers, is discussed in this appendix:

- Location of the DIL subroutines,
- Information about creating interface special files used by DIL subroutines,
- Relationship between entity identifiers and file descriptors,
- Restrictions imposed by the hardware on using the DIL subroutines,
- Techniques for improving data transfer performance when using DIL subroutines.

Location of the DIL Subroutines

The DIL subroutines that provide direct control of your computer's interfaces are contained in the library `/usr/lib/libdvio.a`. Some of these subroutines are general-purpose and can be used with any interface supported by the library, while others provide control of specific interfaces. The Device I/O Library (DIL) currently supports the HP-IB and GPIO interfaces.

Linking DIL Subroutines

The `libdvio.a` library redefines the *read, write, fcntl, dup,* and *ioctl* entry points. For DIL to work properly, the DIL library must be linked *before* `libc`.

The GPIO Interface

The **GPIO** (General Purpose Input/Output) interface is a very flexible parallel interface that allows communication with a variety of devices. On Series 300 computers, the interface sends and receives up to 16 bits of data with a choice of several handshake methods. External interrupt and user-definable signal lines provide additional flexibility.

The GPIO interface is comprised of the following lines:

- 16 parallel data input lines
- 16 parallel data output lines
- 4 handshake lines
- 4 special-purpose lines.

Data Lines

There are 32 data lines: 16 for input and 16 for output. These lines normally use negative logic (0 indicates true, 1 indicates false). The logic can be changed so that a 1 indicates true with the interface's Option Switches. Refer to your GPIO interface manual to see how to do this.

Handshake Lines

Although four lines fall into this group, only three are used for controlling the transfer of data:

- PCTL—Peripheral ConTroL
- PFLG—Peripheral FLaG
- I/O—Input/Output.

The Peripheral Control (**PCTL**) line is controlled by the interface and used to initiate data transfers. The Peripheral Flag (**PFLG**) line is controlled by the peripheral device and used to signal the peripheral's readiness to continue the transfer process. The Input/Output (**I/O**) line is used to indicate direction of data flow.

Special-Purpose Lines

Four lines are available for any purpose you desire; two are controlled by the peripheral device and sensed by the computer, and two are controlled by the computer and sensed by the peripheral.

Data Handshake Methods

There are two handshake methods using PCTL and PFLG to synchronize data transfers: **pulse-mode handshakes** and **full-mode.** If the peripheral uses pulses to handshake data transfers and meets certain hardware timing requirements, the pulse-mode handshake is used. The full-mode handshake should be used if the peripheral does not meet the pulse-mode timing requirements. Refer to the GPIO interface's documentation for a description of these handshake methods.

Data-In Clock Source

Ensuring that data is *valid* when read by the receiving device differs slightly depending on what direction the data is flowing. When *writing data out* from the computer the interface generally holds data valid while PCTL is in the asserted state, the peripheral must read the data during this period.

When *reading data from* the peripheral, the peripheral must hold the data valid until it can signal that the data is valid or until the data is read by the computer. The peripheral signals that the data is valid using the PFLG line. This clocks the data into the interface's Data-In registers.

You can specify the logic level of the PFLG line that indicates valid data by setting the **FLAG** switches on the interface card. Refer to the card's installation manual to find out how to do this.

Creating the Interface Special File

HP-UX treats I/O to an interface the same way it treats I/O to any input/output device: the interface must have a special file. The general process of creating special files is described in the *HP-UX System Administrator Manual* for your system. The following discussion points out specific requirements needed for a special file associated with a given HP-IB or GPIO interface.

Creating the Special File

Special files are created using the `mknod(1M)` command; you must be super-user to execute this command. When used to create an interface special file, `mknod` has the following syntax:

`mknod` *pathname* `c` *major_number minor_number*

The `c` parameter to `mknod` tells the system to create the file as a character special file. Descriptions of the remaining parameters to the `mknod` command follow.

pathname

The *pathname* parameter specifies the name to be given to the newly created interface special file. The *pathname* identifies the interface itself, not a peripheral on the interface. Special files are usually kept in the directory `/dev`. This is basically an HP-UX convention; some commands expect to find special files in the `/dev` directory and fail if they are not there.

major_number

The *major number* specifies which device driver to use with the interface. The following table shows the major number used for each supported interface:

Major Number	Interface
21	HP-IB Interface
22	GPIO Interface

minor_number

The *minor number* parameter tells `mknod` the location of the interface. The minor number has the following syntax:

```
0xScAdUV
```

where:

`0x` specifies that the characters which follow represent hexadecimal values. These two characters (zero and x) are entered as shown.

`Sc` a two-digit hexadecimal value specifying the select code of the interface card. The select code is determined by switch settings on the HP-IB interface card.

`Ad` a two-digit hexadecimal value specifying a bus address. To use DIL routines with the interface, the special file should be created as a **raw** special file: the `Ad` component of the minor number should be 31 (1f in hexadecimal). If `Ad` is less than 31, then the file is *not* created as a raw file; it is created as an auto-addressable file. (In this case, `Ad` specifies the bus address of the device for which the special file is created.) If only one device can be connected to the interface (e.g., the GPIO interface), the component of the minor number is ignored.

`U` a single-digit hexadecimal value specifying a secondary address. This component of the minor number is ignored when the special file you are creating is for an interface; you should set it to 0.

`V` a single-digit hexadecimal value specifying a secondary address, such as the volume number in a multi-volume drive. This component of the minor number is ignored also; you should set it to 0.

Creating an HP-IB Interface File

Suppose you want to create an HP-IB interface special file with the following characteristics:

- the pathname is `/dev/raw_hpib`
- because the interface is HP-IB, the major number is 21
- the card's select code switches are set to select code 2—i.e., the `Sc` component of the minor number is 02

- the special file must be a **raw** special file in order to use DIL subroutines with it; therefore, the `Ad` portion of the minor number must be 31 (1f in hexadecimal).

Based on this information, you would use `mknod` as follows to create the special file for the interface:

```
mknod /dev/raw_hpib c 21 0x021f00
```

To further illustrate the use of `mknod`, suppose you have two HP-IB interfaces (major number = 21) whose switches are set to select codes 2 and 3. The following `mknod` commands set up a special file for the interface at select code 02 (`/dev/raw_hpib1`) and select code 03 (`/dev/raw_hpib2`):

```
mknod /dev/raw_hpib1 c 21 0x021f00

mknod /dev/raw_hpib2 c 21 0x031f00
```

Creating a GPIO Interface File

Now suppose you have a GPIO interface that you want to access with the DIL subroutines on the same computer.

Because the GPIO interface does not use a bus architecture, the usual bus address (`Ad`) and secondary address (`UV`) components of `mknod`'s minor number are ignored, and you need only determine the select code value.

Assuming that you have set the interface select code switches to 04 on the Series 300 GPIO card, the following `mknod` command will create the appropriate special file, named `/dev/raw_gpio`:

```
mknod  /dev/raw_gpio  c  22  0x040000
```

Effects of Resetting (via io_reset)

For an HP-IB interface on Series 300 computers, resetting involves clearing REN, pulsing its Interface Clear line (IFC), and resetting REN; for a GPIO interface the Peripheral Reset line (PRESET) is pulsed. If `io_reset` fails, the routine returns a −1; otherwise the routine returns a 0.

Entity Identifiers

On Series 300 computers, an entity identifier for a file used by a DIL routine is equivalent to an HP-UX file descriptor. This means that you can obtain entity identifiers for your interface files with the system subroutines `dup`, `fcntl`, and `creat`, in addition to `open`.

Restrictions Using the DIL Subroutines

This section presents some restrictions on using the DIL subroutines on Series 300 computers. These restrictions are organized under the routine to which they apply. The subroutines are presented in alphabetical order.

hpib_send_cmnd

By default, the Series 300 HP-IB interface card uses odd parity when you send commands via `hpib_send_cmnd`. To do this, it overwrites the most-significant bit of each command byte with a parity bit. This should not cause a problem since all HP-IB commands use only 7 bits, and the eighth is free for use as a parity bit. The behavior of `hpib_send_cmnd` can be modified to use all eight bits for commands using the `hpib_parity_ctl` subroutine.

hpib_status

The `hpib_status` routine cannot sense lines being driven (output) by the interface. In other words, listeners cannot senses NDAC and non-controllers cannot sense SRQ.

io_reset

When an HP-IB interface is reset via `io_reset`, the interrupt mask is set to 0, the parallel poll response is set to 0, the serial poll response is set to 0, the HP-IB address is assigned, the IFC line is pulsed (if system controller), the card is put on line, and REN is set (if system controller).

When a GPIO interface is reset, the peripheral request line is pulled low, the PTCL line is placed in the clear state, and if the DOUT CLEAR jumper is installed, the data out lines are all cleared. The interrupt enable bit is also cleared.

io_speed_ctl

If the I/O transfer speed is set less than 7Kb/sec (i.e., the *speed* parameter is less than 7), then the interface will use interrupt transfer mode. If the transfer speed is set greater than 140Kb/sec (*speed* $>$ 140), then the system chooses the fastest mode possible. If the speed is between 7Kb and 140Kb/sec (7Kb $\leq$ *speed* $\leq$ 140), then DMA transfer mode is used.

IMPORTANT If you are using pattern termination, via `io_eol_ctl`, then you'll always get interrupt mode, regardless of speed.

---

io_timeout_ctl

This routine allows you to set a time limit for I/O operations on an entity identifier associated with an interface file. The timeout value that you specify is a 32-bit long integer that indicates the length of the timeout in microseconds. However, the resolution of the effective timeout is system-dependent. On the Series 300 computers the timeout is rounded up to the nearest 20-millisecond boundary. For example, if you specify a timeout of

150000 microseconds (150 milliseconds), the effective timeout is rounded up to 160 milliseconds.

Performance Tips

Device I/O performance on Series 300 computers using DIL subroutines can be improved by following these guidelines:

- Use `io_burst` for many small data transfers (less than 4 Kbytes).
- For processes running with an effective user ID of super-user, lock the process in memory by using *plock*(2) (see *HP-UX Reference*) which informs the system that the process code, data, or both are not to be swapped out of memory. Here is an example illustrating the use of `plock`:

```
#include <sys/lock.h>
main()
{
   int plock();
   plock(PROCLOCK); /* lock text and data segments into memory*/
   :
   plock(UNLOCK);   /* unlock my process*/
}
```

- Use auto-addressing for all read and write operations (refer to Chapter 3 under the topic "Setting up Talkers and Listeners").
- Use *rtprio*(2) to increase the system priority of an I/O process. `rtprio` requires that the process be running with an effective user ID of super-user. The real-time priorities available with `rtprio` are non-degrading priorities. Be careful when using real-time priorities. Increasing I/O process priorities above system processes may cause undesirable behavior. For example, requesting a real-time priority in the range of 0-63 places your process at a higher priority than the DIL interrupt handler system process. This means that interrupts could be lost if available CPU resources are insufficient. The following example places the calling process at the lowest (least important) real-time priority:

```
#include <sys/rtprio.h>
main()
{
   int rtprio(), my_proc;

   my_proc = 0;       /* specifying process number zero tells rtprio */
                      /* to refer to the calling process. */
   rtprio(my_proc, 127); /* priority 127 = lowest real-time priority*/
   .
   .
   .
   rtprio(my_proc, RTPRIO_RTOFF); /* turn off real-time priority*/
}
```

B

Series 800 Dependencies

The following information, specific to the Device I/O Library (DIL) on Series 800 computers, is discussed in this appendix:

- compiling programs that use DIL routines
- accessing the special files for the interfaces that you plan to use with DIL
- creating special files for the interfaces that you plan to use with DIL
- DIL routines affected by the Series 800 hardware
- DIL support of HP-IB auto-addressed files
- improving performance of DIL programs

Compiling Programs That Use DIL

The DIL routines are located in the library `/usr/lib/libdvio.a`. Thus, programs can be linked as:

```
cc test.c -ldvio
```

Accessing the Interface Special Files

The Series 800 kernel is shipped with a default I/O configuration. This means a default set of special files is made for you. For example, the `/dev/hpib` directory contains special files created for use with HP-IB instruments connected to the HP 27110B HP-IB interface. The special file */dev/gpio0* is created for use with instruments or peripherals connected to the HP27114A Asynchronous FIFO interface (AFI). The `insf` command is used to install these special files all at one time. `Mknod` could also be used to create them one at a time. For more information on `insf` and `mknod` refer to the *HP-UX Reference*.

Major Numbers

Major numbers map the hardware I/O cards to the software I/O driver for the type of I/O application the card will be doing. The driver used to talk to the HP-IB card for instrument I/O is called `instr0`, and corresponds to major number 21. The HP-IB card talks to different drivers (which use different major numbers) to do I/O to other kinds of devices, such as disc drives or printers. All default special files in the `/dev/hpib` directory use major number 21. The driver that talks to the AFI card is called `gpio0`, and corresponds to major number 22. The */dev/gpio0* special file uses major number 22.

Minor Numbers and Logical Unit Numbers

Drivers use minor numbers to map the hardware I/O cards to their locations in the Series 800 I/O backplane. The default I/O configuration shipped with your Series 800 creates special files accessing a subset of the available backplane slots. For the HP-IB card, two slots are available for instrument I/O, and one slot is available for the AFI card. Slot information is accessed through the device's **logical unit** number. The logical unit number is mapped into the special file's minor number. For HP-IB special files, the HP-IB bus address is also mapped into the minor number.

The minor number syntax for an HP-IB special file is:

```
0x00LuBa
```

where **Lu** is the device's logical unit number, and **Ba** is the bus address of the HP-IB device. Both numbers are in hexadecimal.

The minor number syntax for an AFI special file is:

```
0x00Lu  00
```

where **Lu** is the device's logical unit number in hexadecimal.

For example, a long listing of the special file */dev/hpib/0a16* shows

```
$ ll /dev/hpib/0a16
crw-rw-rw-   1 root     root     21 0x000010 Mar 11 15:19 0a16
```

The logical unit number is 0, and bus address 16 is 10 in hexadecimal.

Listing Special Files

The Series 800 I/O architecture is based on a hierarchical design. The use of logical numbers in conjunction with the major and minor number allows the system to keep track of all the information about the I/O structure. The `lssf` command, list special file, is a tool that makes it easy to read information about a special file without decoding it by hand.

The syntax of `lssf` is:

```
lssf [-f dev_file] path
```

where **path** is the pathname of the special file. `Lssf` uses the major number from the special file to find the name of the device driver in a file called */etc/devices*. If you use the *-f* option, `lssf` looks in *dev_file* instead of `/etc/devices`. It then decodes the minor number, outputs the logical unit number, the device bus address (if there is one), and the corresponding CIO slot address for the actual card in the I/O backplane.

Using the default special file */dev/hpib/0a16* as an example, the following output is produced:

```
$ lssf /dev/hpib/0a16
instr0 lu 0 bus address 16 address 8.2.16 /dev/hpib/0a16
```

where `instr0` is the name of the instrument HP-IB driver, the logical unit number is *0,* the HP-IB bus address is *16,* and the backplane address of the HP-IB card is *8.2.16.* This says that the CIO channel card is in mid-bus address *8,* and the HP-IB card should be in slot *2* of that CIO channel. There

are 12 CIO slots available, numbered 0-11. The last digit, in this case *16,* is the HP-IB bus address of the device *0a16.*

The default HP-IB special files are set up for cards in slot 2 or slot 7 of the CIO channel at mid-bus address 8. A special file for each possible bus address (0-31) is made for each card. The special files for the card at slot 2 all have a logical unit number of 0, and the special files for the card in slot 7 all have a logical unit number of 1.

The default GPIO special file is set up for an AFI card in slot 5 of the CIO channel at mid-bus address 8, and uses a logical unit number of 0.

For more information on `lssf` refer to the *HP-UX Reference.*

Naming Conventions for Interface Special Files

If your Series 800 computer was configured correctly, the special files discussed above will already have been created.

By convention, HP-IB special files reside in the `/dev/hpib` directory. Also by convention, the default special files for the HP-IB **raw bus** (a HP-IB card itself) are named */dev/hpib/***X,** where **X** is the bus's logical unit. `Auto-addressed` files are named */dev/hpib/***X****a****Y,*** where **X** is the logical unit, **a** stands for an auto-addressed file, and **Y** is the file's associated HP-IB bus address (see the "DIL Support of HP-IB Auto-Addressed Files" section of this appendix).

The naming convention for the GPIO default special files is */dev/gpio***X,** where **X** is the device's logical unit.

If you cannot locate the default special files on your system, refer to the next section for how to create them.

Creating Interface Special Files

If the special files you need for HP-IB or GPIO are not available on your system, you can use the `mksf` command to create them. `Mksf` is a high-level command implemented for the Series 800, that can be used instead of `mknod`. Like `lssf`, `mksf` frees you from having to know the major number and minor number format. `Mksf` makes the special file creation process consistent for all classes of devices. The syntax of `mksf` is:

```
mksf -d driver -l lu var|other_flags| ... sfname
```

where `driver` is the name of the driver associated with the special file, `lu` is the file's logical unit, and *sfname* is the name of the special file you wish to create.

Each class of device can have additional class-dependent attributes (such as the bus address for an HP-IB auto-addressed file).

For HP-IB devices, the driver is `instr0`. Thus, to create a special file named */dev/bus* for HP-IB lu 1, you use the command:

```
mksf -d instr0 -l 1 /dev/bus
```

When creating auto-addressed HP-IB special files, you add another option `-a` to associate the address with the device. For example, to create an auto-addressed special file called */dev/plotter,* at bus address 7 on HP-IB lu 2, you could type:

```
mksf -d instr0 -l 2 -a 7 /dev/plotter
```

For the AFI card, the driver is `gpio0`. Thus, to create a special file named */dev/afi* for GPIO lu 0, you could use the command:

```
mksf -d gpio0 -l 0 /dev/afi
```

For more information on `mksf` or `mknod`, refer to the *HP-UX Reference.*

Hardware Effects on DIL Routines

The HP-IB card supported on the Series 800 is the HP 27110B HP-IB interface; the GPIO card is the HP 27114A Asynchronous FIFO Interface (**AFI**).

This section presents some restrictions on using the DIL routines on Series 800 computers. These restrictions are organized under the DIL routine to which they apply. The routines are presented in alphabetical order. A list of *errno* error names can be found in section two of the *HP-UX Reference*. *Errno* numeric values are defined in the file `/usr/include/sys/errno.h`.

hpib_rqst_srvce

The `hpib_rqst_srvce` routine only permits bit 6 of the serial poll *response* to be set. If `hpib_rqst_srvce` is called with a *response* having bit 6 set, the interface sends <01000000> (64 decimal) in response to serial poll; if bit 6 is not set in *response,* the interface sends <10000000> (128 decimal). See "The Computer as a Non-Active Controller" in Chapter 3.

hpib_io

The atomicity of `hpib_io` calls is not guaranteed.

hpib_atn_ctl, hpib-address_ctl, hpib_parity_ctl

These routines are not currently supported on the Series 800.

io_eol_ctl

The AFI driver does not support pattern matching on reads; all `io_eol_ctl` calls return -1 and set *errno* to **EINVAL.**

io_reset

When an HP-IB interface is reset via `io_reset`, the card's parallel poll response is set to 0; its serial poll response is set to 128; its HP-IB address is read off the hardware switches; and the card is put on-line. Any enabled interrupts are preserved. If the card is configured as system controller, then Interface Clear (IFC) is pulsed and Remote Enable (REN) is asserted.

When an AFI interface is reset via `io_reset`, each of the three control output lines is reset to zero, the incoming Attention Request (ARQ) is disabled, the ARQ flip flop is cleared, the ARQ enable flip flop and the handshake to the peripheral are disabled, and the FIFO buffer is flushed out.

io_speed_ctl

The `io_speed_ctl` routine is not supported on Series 800 computers; transfer is always done via DMA.

io_timeout_ctl

On Series 800 computers, the timeout you specify via `io_timeout_ctl` is rounded up to the nearest 10-millisecond boundary. For example, if you specify a timeout of 125000 microseconds (125 milliseconds), the effective timeout is rounded up to 130 milliseconds.

DIL functions, *read,* or *write* requests that time out, return a value of -1 and set *errno* to either **ETIMEDOUT** or **EINTR.** If the request can be aborted normally, then *errno* is set to **ETIMEDOUT** . Otherwise, the HP-IB card is reset and **EINTR** is returned.

io_width_ctl

The only allowable data path width for HP-IB devices is 8. AFI devices support 8-bit and 16-bit data paths. If you specify any other width, `io_width_ctl` returns an error indication.

Return Values for Special Error Conditions

On specific error conditions, the Series 800 sets *errno* values which are different from what is expected from the DIL as documented in the HP-UX Standard. For example, when any request times out, *errno* is set to **ETIMEDOUT** ("connection timed out") or instead of setting it to **EOI.** Also, upon HP-IB requests that require the interface to be the active controller or the system controller, set *errno* to **EACCES** ("permission denied"). Requests that are aborted due to system power failure set *errno* to **EINTR** ("interrupted system call"); in addition, your process receives the signal **SIGPWR,** which indicates recovery of system power.

DIL Support of HP-IB Auto-Addressed Files

As noted in Chapter 3 in the section called "Setting Up Talkers and Listeners," one class of HP-IB special files, known as *auto-addressed* files, are associated with a given address on the bus. For *read* and *write* requests to these files, addressing is done automatically; that is, the sequence of talk and listen bus commands is generated for you.

In general, the DIL functions are not defined for auto-addressed files. On the Series 800, however, many of them are implemented, but with more device-oriented actions.

Important The DIL Standard does not currently specify a functional definition for the support of auto-addressed files. When support for auto-addressed files becomes part of the DIL Standard, the specific functionality implemented may differ from the implementation described here for the Series 800. Please keep this in mind when developing programs which take advantage of this new functionality.

The following table shows which DIL functions are supported on auto-addressed files. Entries in the first column work the same on both auto-addressed and non-auto-addressed (also called **raw bus**) files. Entries in the second column are somewhat different for auto-addressed files; entries in

the third column are not supported on HP-IB auto-addressed files and will return an error indication if used.

Table B-1.

Routine	Same Effect	Different Effect	Not Allowed
hpib_abort	•		
hpib_bus_status	•		
hpib_card_ppoll_resp		•	
hpib_eoi_ctl	•		
hpib_io		•	
hpib_pass_ctl			•
hpib_ppoll	•		
hpib_ppoll_resp_ctl			•
hpib_ren_ctl		•	
hpib_rqst_srvce			•
hpib_send_cmd		•	
hpib_spoll		•	
hpib_status_wait			•
hpib_wait_on_ppoll		•	
io_eol_ctl	•		
io_get_term_reason	•		
io_interrupt_ctl	•		
io_on_interrupt		•	
io_reset			•
io_speed_ctl	•		
io_timeout_ctl	•		
io_width_ctl	•		

Those functions in the second column, which operate differently on raw bus and auto-addressed special files, are discussed below.

hpib_card_ppoll_resp

Calling `hpib_card_ppoll_resp` on an auto-addressed file does not configure the HP-IB interface card; rather, it configures the device associated with the file with the appropriate addressing and Parallel Poll configuration commands.

hpib_io

For those `iodetail` structures that send commands (by setting the `mode` flag to HPIBWRITE or HPIBATN), `hpib_io` prefixes the command buffer `buf` with the appropriate device addressing (see `hpib_send_cmd`, below). For data transfers (with `mode` set to HPIBREAD or HPIBWRITE) using auto-addressed files, the addressing is also done for you.

hpib_ren_ctl

Setting REN (by setting the *flag* parameter to a non-zero value) on an auto-addressed file addresses the associated device *before* asserting REN. Clearing REN (by setting *flag* to a zero) addresses the device and sends it a Go To Local command, in lieu of clearing REN.

hpib_send_cmd

Sending HP-IB commands to an auto-addressed file via `hpib_send_cmd` does the appropriate device addressing for you. The `command` buffer you pass down to the device is preceded by the commands necessary to remove any previous listeners on the bus, address the Active Controller to talk, and configure the file's associated device to listen.

hpib_spoll

Performing a serial poll on an auto-addressed file polls the associated device; any bus address passed via the `ba` argument is ignored.

hpib_wait_on_ppoll

For auto-addressed files, the `mask` argument is ignored; only the address associated with the device is polled. In addition, the `sense` argument only specifies the sense of the particular device's assertion. Successful completion of the `hpib_wait_on_ppoll` request implies that the device responded to parallel poll.

io_on_interrupt

The only allowable interrupt for auto-addressed files is SRQ.

Performance Tips

DIL performance improvements for the Series 800 fall into two categories: those that keep your process from waiting for resources, and those that actually improve your I/O performance. The first three of the tips described below fall into the first category; the last two are in the second category.

Process Locking

Normally, the operating system swaps processes in and out of memory; you can circumvent this swapping by using the *plock* system call.

If you are running as the super-user (or have the **PRIV_MLOCK** capability), you can use *plock* to lock your process in memory; *plock* prevents the system from swapping out the process's code, data, or both.

The following example illustrates its use:

```
#include <sys/lock.h>
int plock();

main() {

     plock(PROCLOCK); /* lock text and data segments into memory */
     :
     plock(UNLOCK);    /* unlock the process */
}
```

Refer to `plock(2)` and `getprivgrp(2)` in the *HP-UX Reference* for more information.

Setting Real-Time Priority

The operating system schedules processes based on their priority. Under normal circumstances, the priority of a process drops over time, allowing newer processes a greater share of CPU time. You can assign a higher priority to your process and keep its priority from dropping by using the `rtprio` system call.

If you are running as the super-user (or have the **PRIV_RTPRIO** capability), you can use `rtprio` to give your process a real-time priority. Real-time

processes run at a higher priority than normal user processes; they get preempted only by voluntarily giving up the CPU or by being interrupted by a higher priority process or interrupt.

You must be careful when using real-time priorities because you can increase your priority above those of important system processes. The following example places the calling process at the lowest (least important) real-time priority:

```
#include <sys/rtprio.h>
#define  ME 0 /* a zero process ID means this process */
int rtprio();

main() {
    rtprio(ME, 127);          /* Turn on real-time priority for ME  */
    :
    rtprio(ME, RTPRIO_RTOFF); /* Turn off real-time priority for ME */
}
```

Refer to `rtprio(2)` and `getprivgrp(2)` in the *HP-UX Reference* for more information.

Preallocating Disc Space

if your process is reading large amounts of data and writing it to a file, you can block while the operating system allocates disc space. However, you can allocate disc space in advance by using the `prealloc` system call. The following example opens a file and preallocates 65536 bytes of space for that file:

```
#include <fcntl.h>
#define MAX_SIZE 65536
int prealloc();

main() {
    int eid;

    eid = open("data_file", O_WRONLY);
    prealloc(eid, MAX_SIZE);    /* preallocate space to write into */
    :
}
```

Refer to `prealloc(2)` in the *HP-UX Reference* for more information.

Reducing System Call Overhead

Most DIL function calls you make on the Series 800 map into system calls. Therefore, you can cut down on operating system overhead by using fewer library calls. In particular, use auto-addressed files for all read and write operations, rather than using an extra call to `hpib_send_cmd` to do addressing.

Setting Up Faster Data Transfers

Because of the I/O architecture of the Series 800, data transfers run more efficiently if your data buffers are aligned on a page boundary. The number of bytes per page is defined as **NBPG** and can be referenced by including `<sys/param.h>`. The following example shows how to allocate and page-align a data buffer:

```
#include <sys/param.h> /* defines NBPG and roundup(x, y)          */
#define REAL_SIZE 1024 /* amount of memory we want to page-align */
char *malloc();

main() {
    char *malloc_ptr, *align_ptr;

    :
    malloc_ptr = malloc(NBPG + REAL_SIZE);   /* allocate memory       */
    align_ptr  = roundup(malloc_ptr, NBPG);  /* and round up the ptr  */
                    /* in future data transfers, use align_ptr      */
    :
    free(malloc_ptr);  /* when we're done with the data */
}
```

In addition, even count transfers run more quickly than odd count transfers.

C

ASCII Character Codes

This appendix contains two tables:

- The first table lists ASCII control characters and how to obtain them by pressing the specified key while holding the [Ctrl] key or [Ctrl] and [Shift] keys down.
- The second table fills two pages and lists all ASCII characters with their decimal, binary, octal, and hexadecimal equivalent values as well as their corresponding HP-IB name.

Table C-1. Obtaining ASCII Control Characters

Keys	ASCII	Dec	Oct	Hex	Keys	ASCII	Dec	Oct	Hex
[Ctrl]-[Shift]-[@]	NUL	00	000	00	[Ctrl]-[P]	DLE	16	020	10
[Ctrl]-[A]	SOH	01	001	01	[Ctrl]-[Q]	DC1	17	021	11
[Ctrl]-[B]	STX	02	002	02	[Ctrl]-[R]	DC2	18	022	12
[Ctrl]-[C]	ETX	03	003	03	[Ctrl]-[S]	DC3	19	023	13
[Ctrl]-[D]	EOT	04	004	04	[Ctrl]-[T]	DC4	20	024	14
[Ctrl]-[E]	ENQ	05	005	05	[Ctrl]-[U]	NAK	21	025	15
[Ctrl]-[F]	ACK	06	006	06	[Ctrl]-[V]	SYNC	22	026	16
[Ctrl]-[G]	BEL	07	007	07	[Ctrl]-[W]	ETB	23	027	17
[Ctrl]-[H]	BS	08	010	08	[Ctrl]-[X]	CAN	24	030	18
[Ctrl]-[I]	HT	09	011	09	[Ctrl]-[Y]	EM	25	031	19
[Ctrl]-[J]	LF	10	012	0A	[Ctrl]-[Z]	SUB	26	032	1A
[Ctrl]-[K]	VT	11	013	0B	[Ctrl]-[[]	ESC	27	033	1B
[Ctrl]-[L]	FF	12	014	0C	[Ctrl]-[\]	FS	28	034	1C
[Ctrl]-[M]	CR	13	015	0D	[Ctrl]-[]]	GS	29	035	1D
[Ctrl]-[N]	SO	14	016	0E	[Ctrl]-[Shift]-[^]	RS	30	036	1E
[Ctrl]-[O]	SI	15	017	0F	[Ctrl]-[Shift]-[_]	US	31	037	1F

Table C-2. ASCII Character Codes

ASCII	Dec	Binary	Oct	Hex	HP-IB	ASCII	Dec	Binary	Oct	Hex	HP-IB
NUL	00	00000000	000	00		space	32	00100000	040	20	LA0
SOH	01	00000001	001	01	GTL	!	33	00100001	041	21	LA1
STX	02	00000010	002	02		"	34	00100010	042	22	LA2
ETX	03	00000011	003	03		#	35	00100011	043	23	LA3
EOT	04	00000100	004	04	SDC	$	36	00100100	044	24	LA4
ENQ	05	00000101	005	05	PPC	&	37	00100101	045	25	LA5
ACK	06	00000110	006	06		%	38	00100110	046	26	LA6
BEL	07	00000111	007	07		'	39	00100111	047	27	LA7
BS	08	00001000	010	08	GET	(	40	00101000	050	28	LA8
HT	09	00001001	011	09	TCT	)	41	00101001	051	29	LA9
LF	10	00001010	012	0A		*	42	00101010	052	2A	LA10
VT	11	00001011	013	0B		+	43	00101011	053	2B	LA11
FF	12	00001100	014	0C		,	44	00101100	054	2C	LA12
CR	13	00001101	015	0D		-	45	00101101	055	2D	LA13
SO	14	00001110	016	0E		.	46	00101110	056	2E	LA14
SI	15	00001111	017	0F		/	47	00101111	057	2F	LA15
DLE	16	00010000	020	10		0	48	00110000	060	30	LA16
DC1	17	00010001	021	11	LLO	1	49	00110001	061	31	LA17
DC2	18	00010010	022	12		2	50	00110010	062	32	LA18
DC3	19	00010011	023	13		3	51	00110011	063	33	LA19
DC4	20	00010100	024	14	DCL	4	52	00110100	064	34	LA20
NAK	21	00010101	025	15	PPU	5	53	00110101	065	35	LA21
SYNC	22	00010110	026	16		6	54	00110110	066	36	LA22
ETB	23	00010111	027	17		7	55	00110111	067	37	LA23
CAN	24	00011000	030	18	SPE	8	56	00111000	070	38	LA24
EM	25	00011001	031	19	SPD	9	57	00111001	071	39	LA25
SUB	26	00011010	032	1A		:	58	00111010	072	3A	LA26
ESC	27	00011011	033	1B		;	59	00111011	073	3B	LA27
FS	28	00011100	034	1C		<	60	00111100	074	3C	LA28
GS	29	00011101	035	1D		=	61	00111101	075	3D	LA29
RS	30	00011110	036	1E		>	62	00111110	076	3E	LA30
US	31	00011111	037	1F		?	63	00111111	077	3F	UNL

Table C-2. ASCII Character Codes
Continued

ASCII	Dec	Binary	Oct	Hex	HP-IB	ASCII	Dec	Binary	Oct	Hex	HP-IB
@	64	01000000	100	40	TA0	`	96	01100000	140	60	SC0
A	65	01000001	101	41	TA1	a	97	01100001	141	61	SC1
B	66	01000010	102	42	TA2	b	98	01100010	142	62	SC2
C	67	01000011	103	43	TA3	c	99	01100011	143	63	SC3
D	68	01000100	104	44	TA4	d	100	01100100	144	64	SC4
E	69	01000101	105	45	TA5	e	101	01100101	145	65	SC5
F	70	01000110	106	46	TA6	f	102	01100110	146	66	SC6
G	71	01000111	107	47	TA7	g	103	01100111	147	67	SC7
H	72	01001000	110	48	TA8	h	104	01101000	150	68	SC8
I	73	01001001	111	49	TA9	i	105	01101001	151	69	SC9
J	74	01001010	112	4A	TA10	j	106	01101010	152	6A	SC10
K	75	01001011	113	4B	TA11	k	107	01101011	153	6B	SC11
L	76	01001100	114	4C	TA12	l	108	01101100	154	6C	SC12
M	77	01001101	115	4D	TA13	m	109	01101101	155	6D	SC13
N	78	01001110	116	4E	TA14	n	110	01101110	156	6E	SC14
O	79	01001111	117	4F	TA15	o	111	01101111	157	6F	SC15
P	80	01000000	120	50	TA16	p	112	01110000	160	70	SC16
Q	81	01000001	121	51	TA17	q	113	01110001	161	71	SC17
R	82	01000010	122	52	TA18	r	114	01110010	162	72	SC18
S	83	01000011	123	53	TA19	s	115	01110011	163	73	SC19
T	84	01010100	124	54	TA20	t	116	01110100	164	74	SC20
U	85	01010101	125	55	TA21	u	117	01110101	165	75	SC21
V	86	01010110	126	56	TA22	v	118	01110110	166	76	SC22
W	87	01010111	127	57	TA23	w	119	01110111	167	77	SC23
X	88	01011000	130	58	TA24	x	120	01111000	170	78	SC24
Y	89	01011001	131	59	TA25	y	121	01111001	171	79	SC25
Z	90	01011010	132	5A	TA26	z	122	01111010	172	7A	SC26
[	91	01011011	133	5B	TA27	{	123	01111011	173	7B	SC27
\	92	01011100	134	5C	TA28	|	124	01111100	174	7C	SC28
]	93	01011101	135	5D	TA29	}	125	01111101	175	7D	SC29
^	94	01011110	136	5E	TA30	~	126	01111110	176	7E	SC30
_	95	01011111	137	5F	UNT	DEL	127	01111111	177	7F	SC31

D

DIL Programming Example

This appendix contains a program listing for an HP-IB driver that uses Device I/O Library subroutines to drive various models of Hewlett-Packard Amigo protocol HP-IB printers. It is provided solely for illustrative use, and is not to be construed as optimum programming technique nor necessarily totally bug-free although the program has been extensively tested.

It contains not only examples of DIL subroutine usage, but also other useful programming techniques and structures that can make the task of writing specialized I/O programs much easier.

```
/*************************************************************************/
/* This example Amigo printer driver uses a byte stream as standard      */
/* input and Amigo protocol as output to HP-IB driver (21). Any special  */
/* character handling should be done by a filter that feeds this driver. */
/*                                                                       */
/* This example program is provided for solely illustrative purposes to  */
/* demonstrate typical use of Device I/O Library (DIL) subroutines.  No  */
/* representations are made as to its suitability for any given          */
/* application.                                                          */
/*                                                                       */
/* While the program is intended to show good programming practice, it   */
/* does not necessarily represent optimum programming efficiency.        */
/*************************************************************************/

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/sysmacros.h>

```

```
22  /* HP-IB addressing group bases */
23  #define LAG_BASE    0x20  /*  listener address base */
24  #define TAG_BASE    0x40  /*  talker address base */
25  #define SCG_BASE    0x60  /*  secondary address base */
26
27  /* HP-IB command equates in odd parity */
28  #define GTL         0x01 /*  go to local */
29  #define SDC         0x04 /*  selective device clear */
30  #define DCL         0x94 /*  device clear */
31  #define UNL         0xbf /*  unlisten */
32  #define UNT         0xdf /*  untalk */
33
34  /* HP-IB secondary commands */
35  #define  PR_SEC_DSJ  SCG_BASE+16
36  #define  PR_SEC_DATA  SCG_BASE+0
37  #define  PR_SEC_RSTA  SCG_BASE+14
38  #define  PR_SEC_MASK  SCG_BASE+01
39  #define  PR_SEC_STRD SCG_BASE+10 /* 2608A */
40
41  /* output of DSJ operation 2608A */
42  #define  PR_ATTEN 0x0001
43  #define  PR_RIBBON 0x0002
44  #define  PR_ATT_PAR 0x0003
45  #define  PR_PAPERF 0x0010
46  #define  PR_SELF  0x0020
47  #define  PR_PRINT 0x0040
48
49  /* output of DSJ operation the rest of the printers */
50  #define  PR_RFDATA   0x0000
51  #define  PR_SDS    0x0001
52  #define  PR_RIOSTAT      0x0002
53
54  /* ppoll mask bits */
55  #define  PR_M_RFD 0x0010
56  #define  PR_M_STATUS 0x0020
57  #define  PR_M_POWER 0x0040
58  #define  PR_M_PAPER 0x0080
59
60  /* default parallel poll mask */
61  unsigned char pmask[1] = {PR_M_PAPER+PR_M_POWER+PR_M_STATUS+PR_M_RFD};
62
```

```
63  /* masks for io status byte in case of 2608A */
64  #define  PR_I_POW 0x0001
65  #define  PR_I_OPSTAT 0x0040
66  #define  PR_I_LINE 0x0080
67
68  /* masks for io status byte the rest of the printers */
69  #define  PR_I_POWER 0x0001
70  #define  PR_I_PAPER 0x0002
71  #define  PR_I_PARITY 0x0008
72  #define  PR_I_RFD 0x0040
73  #define  PR_I_ONLINE 0x0080
74
75  /* define printer types */
76  #define T2608A  1
77  #define T2631A  2
78  #define T2631B  3
79  #define T2673A  4
80  #define QjetPlus 5
81  #define T2632A  6
82  #define T2634A  7
83
84  int ptr_type; /* type of printer */
85
86  /* setup defines for fatal returns */
87  #define F_RTRN  1
88  #define F_EXIT  0
89
90  /* setup defines for HP-IB_msg */
91  #define H_READ  1
92  #define H_WRITE  2
93  #define H_CMND  4
94
95  /* default timeout value (in seconds) to infinity */
96  int timeout = 0;
97
98  /* default size of output buffer to printer */
99  int bufsz = 32;
100
```

```
/* device file suffix for raw hpib dev */
char ptr_raw[] = "_00";

/* default output dev to printer */
char ptr_dev[100] = "/dev/lp";

extern char *optarg;
extern int optind;
extern int errno;

/* file id for raw HP-IB dev */
int eid;

/* configured listen and talk commands */
int MTA; /* my talk address */
int MLA; /* my listen address */
int DTA; /* device (printer) talk address */
int DLA; /* device (printer) listen address */

/* device bus address & my bus address */
int devba, myba;

/* my name */
char *procnam;

int Debug = 0;

main(argc, argv)
int argc;
char *argv[];
{

 register i, c;
 register unsigned char *outbuf; /* output buffer pointer */
 int status;
 int selcode;   /* select code of printer */
 struct stat statbuf;
 int errflg = 0;

 procnam = argv[0];  /* save pointer to my name */

```

```
142   /* GET USER SUPPLIED OPTIONS AND PRINTER FILE NAME */
143   while ((i = getopt(argc, argv, "b:t:p:D")) != EOF) {
144    switch (i) {
145    /* set the buffer size to output to printer */
146    case 'b': if ((bufsz = atoi(optarg)) <= 0) errflg++;
147     break;
148
149    /* get the new timeout value in seconds */
150    case 't': if ((timeout = atoi(optarg)) < 0) errflg++;
151     break;
152
153    /* Set the parallel poll pmask (mostly for debugging) */
154    case 'p': if ((pmask[0] = atoi(optarg)) < 0) errflg++;
155     break;
156
157    case 'D': Debug++; break;
158
159    case '?': errflg++;break;
160    }
161   }
162   /* get printer dev if supplied */
163   if (optind < argc)
164    strcpy(ptr_dev, argv[optind]);
165
166   if (errflg) {
167  fprintf(stderr, "usage: %s [-bbufsz -ttmout] [printer_dev]\n", procnam);
168  fprintf(stderr, "-b bufsz > Output buf size to printer (%d)\n", bufsz);
169  fprintf(stderr, "-t tmout > Max seconds to output buffer (%d)\n",
timeout);
170  fprintf(stderr, "printer_dev > Printer device file     (%s)\n", ptr_dev);
171  fprintf(stderr, "-p ppoll_mask > Parallel poll mask
(0x%02x)\n",pmask[0]);
172    exit(2);
173   }
174   /* get memory for the output buffer */
175   outbuf = (unsigned char *)malloc (bufsz + 4);
176  /*
177    NOTE: Printer device file (/dev/lp) is used only to get printer select
178    code and HP-IB bus address.  This is because attention-true (ATN)
179    requests can only be sent to an "HP-IB raw bus device file".  Therefore
180    after getting the SC and BA we will use a "HP-IB raw bus device file" to
181    do all the work, but it must exist with a name similar to the printer
182    device; i.e. "/dev/lp" is changed to "/dev/lp_07", where the "07" is the
183    select code.
184  */
```

```
185   /* check if printer device exists */
186   if (stat(ptr_dev, &statbuf) < 0)
187    fatal_err("stat", ptr_dev, F_EXIT);
188
189   /* check if it is a character device file */
190   if ((statbuf.st_mode & S_IFMT) != S_IFCHR)
191    fatal_err("Must be a char_special file", ptr_dev, F_EXIT);
192
193   /* extract selectcode from the printer device */
194   selcode = m_selcode(statbuf.st_rdev);
195
196   /* make the HP-IB raw bus device file name from selectcode */
197   ptr_raw[1] += selcode / 16;
198   ptr_raw[2] += selcode % 16;
199   if ((selcode % 16) >= 10) ptr_raw[2] += ('a' - '0' -10);
200   strcat(ptr_dev, ptr_raw);
201
202   /* get device BA from the printer device and config control bytes */
203   devba = m_busaddr(statbuf.st_rdev);
204   DLA = LAG_BASE + devba; /* device listen address */
205   DTA = TAG_BASE + devba; /* device talk address */
206
207   /* open the HP-IB raw bus device */
208   if ((eid = open(ptr_dev, O_RDWR)) <0) {
209    fatal_err("Raw HP-IB open", ptr_dev, F_RTRN);
210  fprintf(stderr,
211  " The following commands executed as a super user may be necessary\n\n");
212  fprintf(stderr, "     # mknod %s c 21 0x%s1f00\n", ptr_dev, &ptr_raw[1]);
213  fprintf(stderr, "     # chmod 555 %s\n", ptr_dev);
214  fprintf(stderr, "     # chown lp  %s\n", ptr_dev);
215    exit(2);
216   }
217   /* get (my) BA of the controller and configure control bytes */
218   if ((myba = hpib_bus_status(eid, 7)) < 0)
219    fatal_err("Must be raw hpib driver (21)", ptr_dev,F_EXIT);
220   MLA = LAG_BASE + myba; /* controller (my) listen address */
221   MTA = TAG_BASE + myba; /* controller (my) talk address */
222
223   /* go do the Amigo identify */
224   ptr_type = amigo_identify();
225
```

```
226  if (Debug) {
227   printf("%s Identified ", ptr_dev);
228   switch(ptr_type) {
229   case T2608A:  printf("2608A");  break;
230    case T2631A:  printf("2631A");  break;
231    case T2631B:  printf("2631B");  break;
232    case T2673A:  printf("2673A");  break;
233    case QjetPlus:  printf("QuietJet Plus");break;
234    case T2632A:  printf("2632A");  break;
235    case T2634A:  printf("2634A");  break;
236   default: printf("You forgot one dummy"); break;
237   }
238   printf(" printer\n");
239  }
240  /* set the timeout to user requested value */
241  if (io_timeout_ctl(eid, timeout * 1000000) < 0)
242   fatal_err("io_timeout_ctl", ptr_dev, F_EXIT);
243
244  /* always tag last output data byte with EOI */
245  if (hpib_eoi_ctl(eid, 1) < 0)
246   fatal_err("hpib_eoi_ctl", ptr_dev, F_EXIT);
247
248  /* clear out the status bits */
249  amigo_clear();
250
251  /* check the status bits */
252  status = amigo_status();
253  if (Debug) printf("%s Printer status = 0x%x\n", ptr_dev, status);
254
255  /* set the ppoll mask required by some printers */
256  amigo_set_pmask();
257
```

```
258   /* MAIN OUTPUT LOOP */
259   i = 0;
260   while ((c = getchar()) != EOF) {
261    if (i == bufsz) {
262     amigo_write(outbuf, i);
263     i = 0;
264    }
265    outbuf[i++] = c;
266   }
267   /* post remaining buffer */
268   if (i) amigo_write(outbuf, i);
269   exit(0);
270  }
271
272  /* ROUTINE TO DO THE MAIN I/O TO THE BUSS */
273  /* lock bus, do preamble, read/write, do postamble and unlock bus */
274  /* preamble must be 3 or 4 bytes, postamble must be 1 or 2 bytes */
275  int
276  HPIB_msg(rw_flag, pcm1, pcm2, pcm3, buffer, length, ocm0, ocm1)
277  int rw_flag;
278  int  pcm1;
279  int  pcm2;
280  int  pcm3;
281  char  *buffer;
282  int  length;
283  int  ocm0;
284  int  ocm1;
285  {
286   unsigned char pre_cmd[4];
287   unsigned char post_cmd[2];
288   int tlog = -1;
289
290   pre_cmd[0] = UNL;  /* always issue unlisten command first */
291   pre_cmd[1] = pcm1;
292   pre_cmd[2] = pcm2;
293   pre_cmd[3] = pcm3;
294
295   post_cmd[0] = ocm0;
296   post_cmd[1] = ocm1;
297
298   /* first get exclusive use of the bus */
299   if (io_lock(eid) < 0)
300    fatal_err("io_lock", ptr_dev, F_EXIT);
301
```

```
302   /* send the preamble 3 or 4 bytes with attention true */
303   if (hpib_send_cmnd(eid, pre_cmd, (pcm3 ? 4 : 3)) < 0)
304    fatal_err("hpib_send_cmnd preamble", ptr_dev, F_EXIT);
305
306   switch (rw_flag) {
307   case H_READ:
308    if ((tlog = read(eid, buffer, length)) < 0)
309     fatal_err("read", ptr_dev, F_EXIT);
310    break;
311
312   case H_WRITE:
313    if ((tlog = write(eid, buffer, length)) < 0)
314     fatal_err("write", ptr_dev, F_EXIT);
315    break;
316
317   case H_CMND:
318    return(0);
319   default:
320    return(-1);
321   }
322   /* send the postamble 1 or 2 bytes with attention true */
323   if (hpib_send_cmnd(eid, post_cmd, (ocm1 ? 2 : 1)) < 0)
324    fatal_err("hpib_send_cmnd postamble", ptr_dev, F_EXIT);
325
326   /* at last unlock the bus so other bus users can access it */
327   if (io_unlock(eid) < 0)
328    fatal_err("io_unlock", ptr_dev, F_EXIT);
329
330   return(tlog);
331  }
332
333  int
334  amigo_identify()
335  {
336   unsigned char identify[2];
337
```

```
338  /* TLK31 (UNT) is special for amigo identify */
339  /* finish with a MTA (UNT is not save for non-amigo devices) */
340  HPIB_msg(H_READ, MLA, UNT, SCG_BASE + devba, identify, 2, MTA, 0);
341
342  switch(identify[0]) {
343  case 32:
344   /* Amigo identify */
345   switch(identify[1]) {
346   case 1:  return(T2608A);
347   case 2:  return(T2631A);
348   case 9:  return(T2631B);
349   case 11: return(T2673A);
350   case 13: return(QjetPlus);
351   case 16: return(T2632A);
352   case 17: return(T2634A);
353   default:
354    printf("Unrecognized Amigo printer, ID2 = %d\n",
355     identify[1]); break;
356   }
357   break;
358  case 33:
359   if (identify[1] == 1)
360    printf("Ciper printer not supported yet!\n");
361   break;
362  default:
363 printf("Unrecognized Amigo Printer identify, ID1 = %d, ID2 = %d\n",
364    identify[0], identify[1]);
365   break;
366  }
367  exit(2);
368 }
369
370 /* set the parallel poll mask value */
371 amigo_set_pmask()
372 {
373  HPIB_msg(H_WRITE, MTA, DLA, PR_SEC_MASK, pmask, 1, UNL, 0);
374 }
375
```

```
376 /* do the amigo clear followed by selective device clear */
377 amigo_clear()
378 {
379  HPIB_msg(H_WRITE, MTA, DLA, SCG_BASE + 16, "\0", 1, SDC, UNL);
380 }
381
382 /* get the dsj byte */
383 int
384 amigo_dsj()
385 {
386  unsigned char dsj_byte[1];
387
388  HPIB_msg(H_READ, MLA, DTA, PR_SEC_DSJ, dsj_byte, 1, UNT, 0);
389  return(dsj_byte[0]);
390 }
391
392 /* return the amigo status byte */
393 int
394 amigo_status()
395 {
396  unsigned char status_byte[1];
397
398  HPIB_msg(H_READ, MLA, DTA, PR_SEC_RSTA, status_byte, 1, UNT, 0);
399  return(status_byte[0]);
400 }
401
402 /* output a buffer to printer */
403 amigo_write(buffer, length)
404 char *buffer;
405 int length;
406 {
407  int status, dsj = 0;
408
409  /* write the buffer */
410  HPIB_msg(H_WRITE, MTA, DLA, PR_SEC_DATA, buffer, length, UNL, 0);
411 again:
412  /* now wait for parallel poll response */
413  if (Debug)  printf("%s Ppoll wait\n", ptr_dev);
414  if (hpib_wait_on_ppoll(eid, 0x80>>devba, 0) < 0)
415   fatal_err("hpib_wait_on_ppoll", ptr_dev, F_EXIT);
416
```

```
417   /* a DSJ is required to remove the ppoll response from device */
418   if (dsj = amigo_dsj()) {
419    if (Debug) printf("%s DSJ = 0x%x\n", ptr_dev, dsj);
420
421    status = amigo_status();
422    if (Debug) printf("%s STATUS = 0x%x\n", ptr_dev, status);
423    goto again;
424   }
425  }
426
427  /* output error message and conditionally abort */
428  fatal_err(message, fname, flag)
429  char *message;
430  char *fname;
431  {
432   fprintf(stderr, "%s: Error - %s of %s ", procnam, message, fname);
433   if   (errno) perror("");
434   else fprintf(stderr, "\n");
435
436   if (flag == F_RTRN) return;
437   if (flag == F_EXIT) exit(2);
438   exit(3);
439  }
```

Index

A

B

H

I

T

U

W

1

Using Curses and Terminfo

Introduction

This tutorial describes the operation of the `curses` library (see *curses*(3x) entry in *HP-UX Reference*) and the `terminfo` routines and corresponding database (see *terminfo*(5) entry). It is intended for use by programmers who are interested in writing screen-oriented software using the `curses` library package. `curses` uses the `terminfo` routines and database when interacting with a given terminal in the system and when formatting display data for subsequent output to the terminal display.

`curses` is a versatile cursor and screen control package that has many capabilities. It is designed to efficiently utilize terminal screen control and display capabilities, thus limiting its demand for computer CPU resources. It can create and move windows and subwindows, use display highlighting features, and support other terminal capabilities that enhance visual interaction with display terminal users. All interaction with a given terminal is tailored to the terminal type which is obtained from the environment variable `TERM`).

`curses` also interacts with the terminal keyboard, and can handle user inputs. Its ability to handle keys that produce multi-character sequences (such as arrow keys) as ordinary keys can be used to add versatility to application programs.

Display Data Handling

Output Data Structure

curses uses data structures called windows to collect display text, then transfers the data structures to the terminal display screen during execution of refresh routines. Each window contains a two-dimensional data array for storing text and character highlighting attributes. Other data structures associated with the window contain the current cursor position and various pointers, and fill other curses needs.

Two windows are always present when curses is active. **Current screen** is named curscr for programming purposes, and represents the current screen. It is used as a reference when optimizing output operations to the CRT screen. The **standard screen** window, named stdscr, is the default destination for all text output operations that are not directed to a window specified in the function. Both curscr and stdscr have the same row and column dimensions as the physical display screen.

Additional program-definable windows can be created and dimensioned as programming needs dictate. Such windows can be any size, provided they do not exceed the row and/or column capacity of the physical display screen.

When a program requires a window that is larger than the available display screen, pads are used. Pads have the same structure and characteristics as a window, but they can be any size within the limits of reasonable memory usage (each pad requires two bytes per character position plus data structure overhead).

Text and Highlighting Data Format

Every window data structure contains, among other things, a two-dimensional array of 32-bit data words, each word corresponding to a displayable character in the window. The lower eight bits in each 32-bit word contain 8-bit character code of the character associated with the corresponding screen display position. The middle eight bits contain NLS attributes. The remaining sixteen bits specify which highlighting attributes, if any, are to be used when the character is displayed. The window data structure also contains a set of current attributes that are used to form the attribute bits as each word is placed in

the array by `addch` or its equivalent. If text highlighting is to be changed for a given character or set of characters, an update to the current attribute set must be performed by `attrset` (or its equivalent) before `addch` is performed. The beginning default attribute set disables all highlighting.

16-bit Data Handling

`curses` uses an NLS environment when displaying 16-bit characters. This means `curses` uses the HP-15 code scheme for interfacing (Input/Output), and users must set up their own NLS environment. Also, `curses` treats a 16-bit character code as an upper 8-bit code and a lower 8-bit code. In short, `curses` can pass only 8-bit characters at one time.

`curses` displays and handles 16-bit characters as follows:

- Overwrite on a 16-bit character.
  If the write of a character starts from either the right half or the left half of a 16-bit character, the remaining half is changed to a space character.
- Insert character on a 16-bit character.
  If the insert of a character starts from the right half of a 16-bit character, the left half and the right half are changed to a space character.
- Write a 16-bit character on the right boundary.
  When a 16-bit character is added to the right margin, a space character is added to the right margin and a 16-bit character is added to the left margin of the next line.
- Delete a 16-bit character.
  If the cursor is at either the right half or left half of a 16-bit character, the remaining half is changed to a space character.
- Clear a 16-bit character.
  If the cursor is at the right half of a 16-bit character, the left half is changed to a space character.
- Move a cursor on a 16-bit character.
  When the specified row (y) and column (x) is on the right half of a 16-bit character, the cursor position (the left half or the right half of a 16-bit character) depends on the terminal's facility. But when the

cursor position is on the left half or the right half of a 16-bit character, an internal cursor position of the curses library is the specified position.

- Display half of a 16-bit character.
  A space character code is used to display a 16-bit character that is hiding half of a 16-bit character in the other window.
- Illegal 16-bit character.
  If the second byte is illegal and the first byte is legal, the two bytes are treated as two 8-bit codes, not one 16-bit code.
- Display enhancement of a 16-bit character.
  The change of the display enhancement is not done on the 16-bit character: the change is done in character units, not byte units. When the change of display enhancement is done on a 16-bit character, the change takes effect from the next character onward.

Applications Program Structure

Consider the following example of an application program structure that uses curses:

```
#include <curses.h>
...
   initscr(); /* Initialization */

   cbreak();/* Various optional mode settings */
   nonl();
   noecho();
...
   while (!done) {/* Main body of program */
...
      /* Sample calls to draw on screen */
      move(row,col);
      addch(ch);
      printw("Formatted print with value %d\n", value);
...
      /* Flush output */
      refresh();
...
   }

   endwin();/* Clean up */
```

```
exit(0);
```

One of curses' major advantages is its ability to optimize the process of updating terminal screen contents, thus reducing the demand for CPU and I/O resources by reducing the amount of data handling required for requested changes in displayed text. This is accomplished by comparing the current screen contents with the window being transferred, then transmitting only those text and control characters that are needed to most efficiently update the screen. Other screen contents remain undisturbed.

Note Most terminals are equipped with hardware scrolling whose operating characteristics make it impossible to write characters in the extreme lower right-hand character position.

In order to optimize screen updates, curses must have access to a data base that reflects current screen contents. When an application program starts execution, the current screen is unknown. To provide a starting current screen reference, a screen clearing operation must be set up early in the program by a call to initscr() which identifies the terminal, initializes data structures, and enables the clearok option in curses so that the screen is cleared during the first *refresh* operation in the program. Upon completion of the first refresh operation, the terminal screen is an exact replica of the text stored in the current screen data base. Use of initscr() in a typical program is shown in the preceding sample program structure example.

When initialization is complete, other operating modes and options can be selected as dictated by program needs. Available operating modes include cbreak() and idlok(stdscr,TRUE) which are explained in detail later. During program execution, screen output is handled through routines such as addch(*ch*) and printw(*fmt,args*). They are equivalent to putchar and printf, respectively, but use curses in addition to the usual other system facilities. Cursor and character positioning are performed by move and other similar calls.

All of the routines mentioned send their output to program-specified window data structures; not directly to the display screen. The window data structure represents all or part of a CRT display screen, and contains the following items:

- An array of characters to be displayed on the screen area defined by the window boundaries,

- Present cursor location,
- Current set of video attributes, and
- Various operating modes and options.

There is little need to be concerned with windows (unless you use several windows during program operation), except to recognize that the data structure corresponding to a given window acts as a buffer/data accumulator for display output requests.

Accumulated contents of a window data structure are sent to the display screen by use of `refresh()` or an equivalent function for windows and pads (functionally similar to a `flush`). `curses` considers many different ways of handling the output operation, taking into account the various available terminal characteristics, similarities between the current screen display and the desired pattern, and other factors. Refresh operations are usually handled using as few characters as possible, but not always.

When the application program is finished, certain clean-up operations should be performed before termination. While the amount of clean-up needed varies, depending on program structure and capabilities, termination should always include a call to `endwin()`. `endwin()` restores all terminal settings to their original state prior to program execution, places the cursor at the bottom left corner of the screen, and dismantles data structures that are no longer needed.

Among the example programs at the end of this tutorial is a program named `scatter` that reads a file and displays the file contents in random order on the CRT display screen. While some application programs assume that terminals have twenty-four 80-character lines of available display space, many terminals do not. To accommodate display terminals having various screen sizes, the variables `LINES` and `COLS` are defined by `initscr` to specify the current screen size. Application programs should always use screen-size variables rather than assuming a 24×80 display screen.

Applications Program Operation

During program operation, no data is output to the display terminal until `refresh` is called. Instead, program routines such as `move` and `addch` place data in a window data structure called `stdscr` (standard screen) that is maintained by `curses`. `curses` also maintains a replica of what is on the current physical screen in `curscr` for updating purposes.

When `refresh` or an equivalent function is called, `curses` compares the `curscr` window with what is presently contained in `stdscr` (or other specified window or pad). The results of the comparison are combined with terminal hardware capabilities to construct character streams that most efficiently update the physical display to the desired contents. Available terminal capabilities are considered while comparing `stdscr` and `curscr` so that the most efficient means of updating the screen can be determined. This sequence is referred to as cursor optimization, and is the basis for naming the `curses` package. During the update operation, `curscr` is also changed to reflect the contents of the updated screen.

Keyboard Input

`curses` capabilities include more than screen writing functions. Several keyboard input functions are also supported, including special handling of certain keys that normally generate a sequence of two or more characters (usually an escape code followed by a single character, but not always). Such keys can then be treated as ordinary single-character keys for improved programming versatility.

The most commonly used keyboard input function is `getch()` which waits for the terminal user to type a character on the terminal keyboard, then returns the character to the calling program. `getch` is similar to `getchar`, except that it uses `curses` instead of other HP-UX facilities. `getch` is particularly useful in programs that use `cbreak()` or `noecho()` options because `getch` supports several terminal- and system-dependent options that are not accessible through `getchar`. Available `getch` options include:

- `keypad` enables programmers to use non-typing keys such as arrow keys, function keys, and other special keys that transmit escape sequences or other multi-character sequences as ordinary single-character keys. Keypad character code length requires 16-bit integer variables for storage.
- `nodelay` enabled option causes `getch` to return immediately with the value -1 if no input character is waiting. This avoids program delays that would otherwise result when no response from the terminal is available.
- `getstr` can be used to input an entire string of characters up to a newline instead of a single character. It also handles echo, erase, and kill character functions associated with the input operation.

Example programs at the end of this tutorial show how these options are used.

Keypad Character Handling

When keypad is enabled, keypad character sequence conversion tables in the `terminfo` data base are used to map keypad character sequences into corresponding single, 16-bit character form. Each supported keypad key must produce a unique character or character sequence when pressed. All convertible sequences must be included in the `terminfo` data base. If any sequence is absent from the table, it cannot be converted, so it is handled in unaltered form. The following special keys are assigned the values and names indicated. Some of the keys listed may not be supported on given terminals, depending on the terminal model and its internal operating characteristics, and whether the conversion sequence is in `terminfo`.

Note Keypad character codes do not fit in a normal 8-bit data element. Therefore a *char* variable cannot be used. Use a larger (16-bit) variable for storing and handling keypad character codes.

Keypad Character Code Values

Character Name	Octal Value	Key name
KEY_BREAK	0401	Break key (unreliable)
KEY_DOWN	0402	Down Arrow key
KEY_UP	0403	Up Arrow key
KEY_LEFT	0404	Left Arrow key
KEY_RIGHT	0405	Right Arrow key
KEY_HOME	0406	Home Up (to upper left corner) key
KEY_BACKSPACE	0407	Backspace key (unreliable)
KEY_F0	0410	Function Key 0
⋮	⋮	⋮
KEY_F(n)	0410+(n)	Function Key (n)
KEY_DL	0510	Delete Line key
KEY_IL	0511	Insert Line key
KEY_DC	0512	Delete Character key
KEY_IC	0513	Insert Character or Enter Insert Mode key
KEY_EIC	0514	Exit Insert-character Mode Key
KEY_CLEAR	0515	Clear Screen key
KEY_EOS	0516	Clear to End-of-Screen key
KEY_EOL	0517	Clear to End-of-line key
KEY_SF	0520	Scroll Forward 1 Line
KEY_SR	0521	Scroll Reverse (backwards) 1 line
KEY_NPAGE	0522	Next Page key
KEY_PPAGE	0523	Previous Page key
KEY_STAB	0524	Set Tab key
KEY_CTAB	0525	Clear Tab key
KEY_CATAB	0526	Clear All Tabs key
KEY_ENTER	0527	Enter or Send key (unreliable)
KEY_SRESET	0530	Soft (partial) Reset key (unreliable)
KEY_RESET	0531	Reset or Hard Reset key (unreliable)
KEY_PRINT	0532	Print or Copy key
KEY_LL	0533	Home Down (to lower left) key

Keyboard Input Program Example

The example program `show` at the end of this tutorial contains an example use of `getch`. `Show` displays a file, one screen at a time; advancing to the next page each time the space bar is pressed. Nearly any exercise for `curses` can be created by constructing an input file that contains a series of 24-line pages, each page varying slightly from the previous page.

In the `show` program:

- `cbreak` is used so that only the space bar need be pressed (use of [Return] is unnecessary).
- `noecho` is used to prevent the character transmitted by the space bar from being echoed during `refresh` calls so that refresh operations are not adversely affected.
- `nonl` is called to enable additional screen optimization.
- `idlok` allows insert and delete line. This capability helps streamline updates in some instances, but produces undesirable effects in other cases. Therefore an option to allow or disallow the capability has been provided.
- `clrtoeol` clears from cursor to end of current line.
- `clrtobot` clears from cursor to end of current line, then clears all subsequent lines to the bottom of the screen.

Display Highlighting

curses supports nine highlighting attributes, each of which has a corresponding 32-bit integer constant named in the include file <curses.h>. The value of each constant is selected such that one bit (corresponding to the attribute) in the 32-bit integer is set while all other bits are cleared. Below is a list of the nine attributes with their corresponding enable-bit positions. The name and octal value of each constant is also shown (note that only eleven digits are needed to represent the 32-bit value; the leading zero identifies the constant as an octal value).

- Standout (bit 23):
  A_STANDOUT = 000040000000
- Underlining (bit 24):
  A_UNDERLINE = 000100000000
- Inverse Video (bit 25):
  A_REVERSE = 000200000000
- Blinking (bit 26):
  A_BLINK = 000400000000
- Dim (bit 27):
  A_DIM = 001000000000
- Bold (bit 28):
  A_BOLD = 002000000000
- Invisible (bit 29):
  A_REVERSE = 004000000000
- No print or display (bit 30):
  A_PROTECT = 010000000000
- Alternate Character Set (bit 31):
  A_ALTCHARSET = 020000000000

addch and waddchr store window characters as 32-bit data words where the lower eight bits (0-7) of each word contain the character code and the upper sixteen bits (16-31), when set, enable the corresponding display highlighting attributes when that character is displayed on a terminal. Each attribute bit corresponds to one of the highlighting functions listed above. Obviously, any selected highlighting feature that is not available on a given terminal cannot be used even though the capability is standard fare for curses. However, when a requested attribute is not available on a given terminal, curses attempts to identify and use a suitable substitute. If none is possible, the attribute is ignored.

Three other constants in <curses.h> are also useful:

- A_NORMAL (value = 000000000000) can be used as an argument for attrset to disable all attributes. attrset(A_NORMAL) is equivalent to attrset(0), but more descriptive.
- A_ATTRIBUTES has an octal value of 037740000000. It can be used in a bit-level logical AND to remove character bits and NLS attributes, isolating the attributes attached to a given character.
- A_CHARTEXT has an octal value of 000000000377. It is useful in a bit-level logical AND to discard all except the lower eight bits of the data word; in effect, separating the character from its highlighting attributes.

curses maintains a set of **current attributes** for each window. Whenever text is being placed in a given window by the program, the current attribute bits for the selected window are added to each character of text data, forming a 32-bit word for each character handled. To select a specific combination of attributes, a program call to attrset (or attron) with new attribute values must precede text output to the window. This can be used to enable one or more attributes when all were previously disabled, disable all currently enabled attributes (attrset(0)), or change the current set to any other new current set.

To enable one or more attributes in the current set without altering other active or inactive attributes, call attron. A call to attroff performs the opposite function, disabling the selected attributes without disturbing any other attributes in the current set.

curses always uses current attribute values, so a call to attrset, attron, or attroff (or their related window functions) must be used whenever you begin, end, or change any selected highlighting option. Here is an example program segment that illustrates how to set a word in boldface then restore normal display attributes for remaining text:

```
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
 ...
refresh();
```

In this example, the space characters before and after the word `boldface` are included in text blocks outside (before and after) the `attrset` calls. This technique prevents `curses` from applying display highlights to the spaces, thus avoiding possible undesirable effects; especially in situations where `curses` attempts to substitute an alternative for unavailable highlighting features.

The attribute `A_STANDOUT` offers unique program flexibility. In many interactive programs, displayed text needs to be enhanced to attract attention. However, it is not critical that the text be displayed with specific attributes. Many multi-terminal systems contain various terminal models that do not support identical highlighting features. For versatility, `A_STANDOUT` uses the terminal characteristics stored in the `terminfo` data base to determine the most pleasing highlighting feature available on the terminal being addressed (usually bold or inverse video), then uses that feature when sending corresponding text to the selected window on the terminal display screen. Two functions, `standout()` and `standend()` are provided so you can conveniently enable and disable `A_STANDOUT` highlighting.

`attrset` can be used to select only one (such as `A_BOLD`, shown in the earlier example in this section) or multiple attributes (such as `A_REVERSE` and `A_BLINK` for blinking inverse video). To change only one attribute or a certain combination of attributes while leaving the others undisturbed, use `attron()` and `attroff()`.

The example program `highlight` at the end of this tutorial demonstrates typical use of attributes. The program uses a text file as input, and embedded escape sequences in the file to control attributes. In the example program, `\U` enables underlining, `\B` selects bold, and `\N` restores normal text. An initial call to `scrollok` allows the terminal to scroll if the text file exceeds the capacity of a single display screen. When `scrollok` is active, if any text extends beyond the lower screen boundary, `curses` automatically scrolls the internally stored window up one line, then calls `refresh` to update the terminal display screen each time a line of input text exceeds the lower screen boundary. The scrolling process continues until end-of-file is reached on the input file.

The `highlight` program comes about as close to being a filter as is possible with `curses`. It is not a true filter because `curses` interacts directly with the terminal screen. `curses`' ability to optimize interaction between HP-UX programs and terminals is inherently linked to its direct monitoring of the

current CRT screen and the windows where display text is being held for output through `refresh` operations. This capability requires that `curses` clear the screen as part of the first `refresh` operation so that it has a known beginning reference condition, then maintain a continually up-to-date data structure that reflects current screen contents and cursor location.

NLS Attributes

`curses` supports two NLS attributes, each of which has a corresponding 32-bit integer constant named in the include file <curses.h>. Below is a list of the two attributes with their corresponding enable-bit positions. The name and octal value of each constant is also shown (note that only eleven digits are needed to represent the 32-bit value; the leading zero identifies the constant as a octal value).

- First byte of a 16-bit character code (bit 14):
  `A_FIRSTOF2` = 000000040000

- Second byte of a 16-bit character code (bit 15):
  `A_SECOF2` = 000000100000

These NLS attributes can be found in a 16-bit character (not character code). These attributes might be returned by the function `inch`. And they cannot be passed to the `curses` functions.

Another constant in <curses.h> is also useful:

- `A_NLSATTR` has an octal value of 000000177400. It can be used in a bit-level logical AND to remove character bits and highlighting attributes, isolating the NLS attributes attached to a given character.

Multiple Windows

A window is a data structure that represents all or part of the CRT display screen. It contains a two-dimensional array of 32-bit character data words, a cursor, a set of current attributes, and several flags. Each 32-bit character data word contains:

- An 8-bit character code in the lower eight bits, and
- An 8-bit NLS attributes code in the middle eight bits, and
- A 16-bit video highlighting code in the upper sixteen bits. Each bit enables one of sixteen attributes when set, each attribute represented by one of the respective bits.

`curses` provides a full-screen window called `stdscr` and a set of functions that use `stdscr`. Another window called `curscr` that represents the current physical display screen is also provided.

It is important that you clearly understand that a window is only a data structure. Use of more than one window does not imply the presence of more than one terminal, nor does it involve more than one process. A window is nothing more than a data object that can be copied to all or part of the terminal screen. `curses`, as presently implemented, cannot handle windows that are larger than the available display screen (use pads for such applications).

Pads

Pads are data structures that are essentially identical to windows, except that they can be larger than the available terminal screen size, and, as a result, must be handled differently. For example, a special refresh function is required that knows how to transfer only a specified part of the total pad area to the current screen instead of the entire pad. Other window operations do not depend on the size of the structure, so they can treat windows and pads identically. In such instances, a single function supports pads and windows (such as `addch`, `delwin`, and similar functions).

Creating Windows

Additional windows can be created so that the applications program can maintain several different screen images. Images can then be alternated under program control as needs dictate. Windows can be useful in editors, games, and other applications such as when handling interactive processes involving multiple users on multiple terminals.

Overlapping windows can also be constructed so that changes to one window are easily copied onto the overlapping area of the second. Several `curses` routines have been provided specifically to handle such cases. `overlay` and `overwrite` copy one window onto the second, each handling the copy operation differently. `wrefresh` can be used to refresh the terminal screen, but in some cases it is more efficient and pleasing to perform a series of internal window operations that are equivalent to refresh, but which do not update the screen. This is done by using a series of calls to `wnoutrefresh` (or its equivalent for pads), followed by a single `doupdate` that copies the series of refreshes onto the physical screen in a single operation. This is readily provided because `refresh` is really a call to `wnoutrefresh` followed by a call to `doupdate`.

To create a new window, use the function:

`newwin(`*lines, cols, begin_row, begin_col*`)`

The `newwin` function call returns a pointer to the newly created window whose dimensions are *lines* by *cols,* and whose upper left-hand corner is positioned at screen location *begin_row* and *begin_col.*

Using Multiple Windows

All operations that affect `stdscr` have a corresponding function for use with other named windows. These functions' names are formed by adding the letter `w` in front of the `stdscr` function name. For example, the window function that corresponds to `addch` is named:

`waddch(`*mywin, c*`)`

To update the contents of the currently displayed screen to match the contents of a window, use:

`wrefresh(`*mywin*`)`

Whenever the boundaries of two or more windows overlap and thus conflict, the most recently refreshed window becomes the currently displayed screen in that area of the display area that is defined by the window size and location.

Any call to the non-w version of any window function (`stdscr` function calls) is converted to its w-prefixed counterpart. Thus, a call to `addch(`*c*`)` produces a call to `waddch(stdscr, `*c*`)`, automatically adding the `stdscr` argument in the process.

The example program `window` at the end of this tutorial shows how windowing can be handled. The main display is kept in `stdscr`. When the user wants to put something else on the screen, a new window is created that covers part of the screen. A call to `wrefresh` on that window causes the window to be written over `stdscr` on the display screen. A subsequent call to `refresh` on `stdscr` causes the original window to be fully restored to the screen, eliminating the temporarily displayed window.

Examine the `touchwin` calls in `window` that precede refresh calls on overlapping windows. `touchwin` calls prevent optimization by `curses`, thus forcing `wrefresh` to completely overwrite the entire window area on the physical screen (previously displayed data is thus erased in the window area only). In some situations, if the `touchwin` call is omitted, only part of the window is written and existing information from a previous window may remain in the newly written window area.

For improved screen addressability, a set of move functions are available in conjunction with most common window functions. They produce a call to `move` before the other function is called, so that the cursor can be relocated before the window function is executed. Here are some examples:

- `mvaddch(`*row,col,ch*`)` is equivalent to `move(`*row,col*`); addch(`*ch*`)`
- `mvwaddch(`*row,col,win,ch*`)` is equivalent to `wmove(`*win,row,col*`); waddch(`*win,ch*`)`.

Refer to the `curses` routines section of this tutorial for more detailed descriptions of the window routines and their related move functions.

Subwindows

Subwindows can be created within any existing window or pad. Subwindows are identical to normal windows except that the subwindow's character data structure occupies the same memory locations as the corresponding character positions in the main window. This means that whenever a character is placed in a subwindow, the main window automatically contains the same character in the same location with the same highlighting attributes. In fact, as a result of shared character storage, any character stored in the character array automatically receives the current attributes for the window or subwindow through which it was stored, regardless of how many subwindows overlap the storage location. This feature greatly simplifies combining windows in a single display for some types of applications.

Each subwindow has its own cursor location, can be configured with a soft scrolling region, and generally has the same capabilities as any normal window, but, except for shared character storage, is completely independent of the original window it is associated with. Because of shared character data structures, `curses` does not allow deletion of any window (`delwin(`*win*`)` or pad that has one or more undeleted subwindows.

If subwindows are created within a pad, care must be exercised in the choice of correct refresh functions and other program characteristics to ensure correct data handling.

Multiple Terminals

`curses` can produce simultaneous output on multiple terminals. This capability is useful in single-process programs that access a common data base such as multi-player games. Output to multiple terminals is a complex issue, and `curses` does not solve all of the related programming problems. For example, it is the program's responsibility to determine the special file name for each terminal line and what type of terminal is connected to that line. The normal method, checking the environment variable `$TERM`, does not work because each process can only examine its own environment. Another issue that must be addressed is the case of multiple programs reading data from a single terminal line, a situation that produces race conditions which

must be avoided because a program that wants to take over a terminal cannot arbitrarily stop whatever program is currently running on that terminal (particularly where security considerations make this action inappropriate, though it is appropriate for some applications such as inter-terminal communication programs).

Race conditions may or may not be a problem, depending on the overall relationships of running programs and processes. For example, if a `curses` program is looking for input from a terminal, there *must* be no other program looking for input from the same terminal (such as a shell). On the other hand, if two programs are sending output to the same terminal at the same time, the result is usually no worse than an unusable screen display. In any event, for interaction with the terminal to flow smoothly, conflicts in terminal access must be prevented.

A typical solution requires the user logged onto each terminal line to run a program that notifies the master program that the user is interested in joining the master program. The master program is given the notification program's process id, the name of the tty link, and the type of terminal being used. The notification program then goes to sleep until the master program finishes. During termination, the master program wakes up the notification program and all programs exit.

`curses` handles multiple terminals by always having a **current terminal.** All function calls always pertain to the current terminal. The master program should set up each terminal, saving a reference (pointer) to the terminal in its own variables. When it is ready to interact with a given terminal, the master program should set the current terminal (use `set_term`) according to program needs, then use ordinary `curses` routines.

Terminal references have type `struct screen *`. To initialize a new terminal, call `newterm(`*`type,fd`*`)`. `newterm` returns a screen reference to the terminal being set up. `type` is a character string that names the kind of terminal being used. `fd` is a stdio file descriptor to be used for input and output to the terminal (if only output is needed, the file can be opened for output only). The `newterm` call replaces the normal call to `initscr`.

To select a new current terminal, call `set_term(`*`sp`*`)` where `sp` is the screen reference returned by `newterm` for the terminal being selected. `set_term` returns a screen reference to the previous terminal.

A full set of windows and options must be maintained for each terminal according to program needs. Each terminal must be initialized separately with its own `newterm` call. Options such as `cbreak` and `noecho`, and functions such as `endwin` and `refresh` must be set (or called) separately for each terminal. Here is a typical scenario for sending a message to each terminal:

```
for (i=0; i <nterm; i++) {
   set_term(terms[i]);
   mvaddstr(0,0,"Important message");
   refresh();
}
```

The sample program `two` at the end of this tutorial contains a full example of how this technique is implemented. The program pages through a file, showing one page to the first terminal; the next page to the second. It then waits for a space character to be typed on either terminal, then sends the next page to the terminal that sent the space character. Each terminal has to be put into nodelay mode separately. No standard multiplexer is available in current HP-UX versions, so it is necessary to busy wait or call `sleep`(1); between each check for keyboard input. `two` waits one second between checks for available terminal keyboard characters.

`two` is only a simple example of two-terminal `curses`. It does not handle notification as described above; instead, it requires the name and type of the second terminal on the program procedure line. As written, `two` requires that the command `sleep 100000` be typed on the second terminal to put it to sleep while the program runs, and the the first-terminal user must have read and write permission on the second terminal.

Low-Level Terminfo Usage

Some programs need access to lower-level primitives than those offered by `curses`. For such programs, the **terminfo-level** interface is provided. This interface does not manage the CRT screen, but gives programs access to strings and capabilities that can be used to manipulate the terminal.

Use of `terminfo`-level routines is discouraged. Whenever possible, higher-level `curses` routines should be used instead, in order to maintain portability to

other systems and handle a wider variety of terminal types. `curses` takes care of all of the anomolies, glitches, and personality defects present in physical terminals, but at the `terminfo` level they must be dealt with in the program. Also, there is no guarantee that the `terminfo` interface will not change with new releases of HP-UX or be upward compatible with previous releases.

There are two circumstances where use of `terminfo` routines is appropriate. On instance is where a special-purpose program sends a special string to the terminal (such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line). The second is when writing a filter. A typical filter performs one transformation on the input stream without clearing the screen or addressing the cursor. If this transformation is terminal-dependent and clearing the screen is inappropriate, `terminfo` routines are preferred.

A program written at the `terminfo` level uses the framework shown here:

```
#include <curses.h>
#include <term.h>
 ...
   Setupterm(0,1,0);
 ...
   putp(clear_screen);
 ...
   reset_shell_mode();
   exit(0);
```

The call to `setupterm` handles initialization (`setupterm`(0,1,0) invokes reasonable defaults). If `setupterm` cannot determine the terminal type, it prints an error message and exits. The calling program should call `reset_shell_mode` before exiting.

Global variables with such names as *clear_screen* and *cursor_address* are defined during the call to `setupterm`. When outputting these variables, use calls to `putp` or `tputs` for better programmer control during output. Global variable strings should not be output to the terminal through `printf` because they contain padding information that must be processed. A program (*such as printf*) that transmits unprocessed strings will fail on terminals that require padding or use Xon/Xoff flow-control protocol.

Higher-level routines described previously are not available at the `terminfo` level. The programmer must determine output needs and structure programs

accordingly. For a list of `terminfo` capabilities and their descriptions, see *terminfo*(5) in the *HP-UX Reference*.

The example program `termhl` at the end of this tutorial shows simple use of `terminfo`. It is similar to `highlight`, but uses `terminfo` instead of `curses`. This version can be used as a filter. The strings used to enter bold and underline mode, and to disable all highlighting attributes are demonstrated.

The program was made more complex than necessary in order to illustrate several `terminfo` properties. For example, `vidattr` could have been used instead of directly outputting `enter_bold_mode`, `enter_underline_mode`, and `exit_attribute_mode`. In fact, the program could easily be made more robust by using `vidattr` because there are several ways to change video attributes. However, this program was structured only to illustrate typical use of `terminfo` routines.

The function `tputs(`*cap,affcnt,outc*`)` adds padding information to the capability `cap`. Some capabilities contain strings such as $<20>, which means to pad for 20 milliseconds. `tputs` adds enough pad characters to produce the desired delay. `cap` is the string capability to be output; `affcnt` is the number of lines affected by the output (for example, `insert_line` may have to copy all lines below the current line, and may require time proportional to the number of lines being copied). By convention, `affcnt` is 1 if no lines are affected rather than 0 because `affcnt` is multiplied by the amount of time required per item, and a zero time may be undesirable. `outc` is the name of a routine that is to be called with each character being sent.

In many simple programs, `affcnt` is set to 1, and `outc` just calls `putchar`. For such programs, the `terminfo` routine `putp(`*cap*`)` is a convenient abbreviation. The example program `termhl` could be simplified by using `putp`.

Note the special check for the `underline_char` capability. Some terminals, rather than having a code to start underlining and a code to stop underlining, use a code to underline the current character. `termhl` keeps track of the current mode, and outputs `underline_char`, if necessary, whenever the current character is to be underlined. Low-level details such as this are a major reason why `curses` routines are preferred over `terminfo` routines. `curses` takes care of all the different terminal keyboard and display functions and highlighting sequences instead of forcing such details onto the application program.

A Larger Example

The example program `editor` is a very simple screen editor that has been patterned after the `vi` editor and illustrates how `curses` can be used for such applications. `editor` uses `stdscr` as a buffer for simplicity, whereas a more useful editor would maintain a separate data structure for editing operations, then display the pertinent contents of that separate structure on the screen. `Editor`, as written, requires a file size equal to screen size. It also cannot handle lines longer than the screen, and has no provision for control characters in the file.

Several program characteristics are of interest. The routine that writes the file back to the file system shows how `mvinch` is used to retrieve characters from given window positions. The data structure used does not provide for keeping track of the number of characters in a line nor the number of lines in the file, so trailing blanks are eliminated when the file is written out.

`editor` uses built-in `curses` functions `insch`, `delch`, `insertln`, and `deleteln`. These functions behave much like equivalent functions on intelligent terminals when inserting and deleting characters and lines.

The command interpreter accepts not only ASCII characters, but also special (non-typing) keys. This is important—a good program accepts both. Defining the keyboard so that every special key has its function defined on a normal typing key as well provides a desirable increase in flexibility. The benefit for new users, for example, is that they can use arrow keys without having to remember that the same functions are available on h, j, k, and l keys in the normal typing area. On the other hand, an experienced user may prefer to keep his fingers on the home typing row where he can work faster, so the typing key equivalent of special keys is appreciated. Handling both classes of keys also widens the variety of terminals the program can interact with because some terminals may not be equipped with arrow or other special keys on the keyboard. Providing an ASCII character synonym for each special keypad key provides better overall program and system flexibility, and makes the program more salable and easier to learn.

Note the call to `mvaddstr` in the input routine. `addstr` is roughly equivalent to the `fputs` function in C. Like `fputs`, `addstr` does not add a trailing newline. It is equivalent to a series of calls to `addch`, using the characters in the string.

`mvaddstr` moves the current cursor position to the specified location in the window before writing the string into the data structure.

The control-L command demonstrates a feature that most programs using `curses` should include. Frequently, an independent program operating beyond the control of `curses` may write something to the terminal screen, or some other event such as line noise causes the physical screen to be altered without `curses` being notified. In such a case, CTRL-L can be used to clear and redraw the current screen at the user's request. This is accomplished by a call to `clearok(curscr)` which sets a flag that causes the next `refresh` to clear the screen. A call to `refresh` follows immediately so that the screen is immediately redrawn using the data in `curscr` so that there is no wait for other program activities or completion of a pending keyboard input. There is also no loss of current screen data.

Note also the call to `flash()` which flashes the screen (unless the terminal has no flashing capability, in which case it rings the bell instead). Replacing the bell with the flashing capability is useful in environments where the sound of the bell is objectionable or distracting. Still, there may be instances where an audible signal is still needed for certain purposes, even in quiet environments. In such cases, the `beep()` routine can still be called instead whenever a real beep is preferred. If `beep` is called and the terminal is not equipped to process the call, `curses` substitutes the `flash` in its place if possible, and vice versa. Thus, a terminal with no beep capability receives a flash sequence when beep is called; a terminal that cannot flash receives a beep sequence when flash is called. If the terminal has neither capability, – well, some situations do present certain limitations – do without or get a different terminal because both are ignored in such a case.

Use of Escape in Program Control

Another important programming practice is terminating the input command with Ctrl-D; not escape. It is very tempting to use escape as a command because the escape key is one of the few special keys that is available on nearly every terminal keyboard (return and break are the only others). However, using escape as a separate key introduces an ambiguity which is handled by `curses` as follows:

Most terminals use sequences of characters beginning with an escape character (called escape sequences) to control the terminal. They also use similar escape

sequences to transmit special keys to the computer. If the computer sees an escape character from the terminal, it cannot immediately determine whether the user pressed the escape key, or whether a special key was pressed instead. `curses` handles the ambiguity by waiting for up to one second. If another character is received within the one-second time limit, the escape and second character are compared with possible escape sequences. If the character pair represents a valid possibility, the wait is extended for up to one more second, or until the next character is received. The cycle continues until a valid special key sequence is completed or a character is received that could not be part of a valid sequence (or the time limit expires). While this technique works well most of the time, it is not foolproof. For example, a user could press the escape key then press one or more other keys that represent a valid sequence before the time limits expired (less than one second between successive key strokes). `curses` would then think that a special key had been pressed. Another disadvantage is the inevitable delay from the time a key is pressed until it can be processed by the program when an escape key is pressed, possibly even accidentally.

Many existing programs use escape as a fundamental command which often cannot be changed without incurring the wrath of a large group of users. Such programs cannot make use of special keys without dealing with the aforementioned ambiguity, and must, at best, resort to a timeout solution. The pathway is clear. When designing new programs and updating older ones, avoid using the escape key for program control whenever possible.

Program Routines

This and the following sections describe `curses` routines that are available to programmers. In this section, the routines are discussed in groups by function in the context of program operation. The next sections list `curses`, `terminfo`, and `termcap` compatibility routines alphabetically for easy reference, and each is discussed in greater detail. Both are helpful as tutorial and reference information, expanding on the information contained in the *curses*(3X) and *terminfo*(5) entries in the *HP-UX Reference*.

The `curses` routines discussed in this section operate on pads, windows, and subwindows. In general, windows and subwindows are treated identically by

most routines. Subwindows share character data structures with the original window, but have their own cursor location and other non-character data structures. Unless indicated otherwise, all references to windows during discussion of window routines apply equally to windows and subwindows.

Program Structure Considerations

All programs using `curses` should include the file `<curses.h>` which defines several `curses` functions as macros and establishes needed global variables as well as the datatype `WINDOW` (window references are always of type `WINDOW*`). `curses` also defines the `WINDOW *` constants `stdscr` (the standard screen that is used as a default for all routines that interact with windows) and `curscr` (the current screen, used as a reference for low-level operations when updating the current display or clearing and redrawing a scrambled display. The integer constants `LINES` and `COLS` are defined, and contain values equal to the number of available lines and columns in the physical display. The constants `TRUE` and `FALSE` are also defined with the values 1 and 0, respectively. Two additional constants are defined; the values returned by most `curses` routines. `OK` is returned when the routine was able to successfully complete its assigned task. `ERR` indicates that an error occurred (such as an attempt to place the cursor outside a defined window boundary or create a window larger than the physical screen); thus, the task was not successfully completed.

The include file `<curses.h>` that must be specified at the beginning of the program automatically includes `<stdio.h>` and an appropriate tty driver interface file, presently `<termio.h>`. Including `<stdio.h>` again in a subsequent program statement is harmless though wasteful, but including a tty driver interface file could cause a fatal error if the file is not the same as the one selected by `curses`.

Any program that uses curses should include the loader option

```
-lcurses
```

in its makefile, whether the program operates at the `curses` or `terminfo` level. If the program only needs `curses`' screen output and optimization capabilities, and no non-default windows are involved, you can improve output speed and processing efficiency by restricting the program to the mini-curses package. Mini-curses is selected by using the compilation flag

```
-DMINICURSES
```

Routines supported by mini-curses are marked by asterisks in the complete list of `curses` routines at the beginning of the `curses` Routines section of this tutorial. They are also similarly marked in the *curses*(3X) entry in the *HP-UX Reference*.

Terminal Initialization Routines

Program entry and exit states must be handled correctly to maintain system integrity and proper terminal operation. If the program interacts with only one user/terminal, `initscr` should be the first function call in the program. It sets up the necessary data structures and makes sure that terminal handling and screen clearing are properly initialized. The program should call `endwin` before terminating, ensuring that the terminal is restored to its original operating state and the cursor is placed in the lower left corner of the screen. `endwin` also dismantles data structures and other program entities that were created by `curses` and are no longer needed.

If the program must interact with multiple terminals during operation, `newterm` should be used for each terminal instead of the single call to `initscr`. `newterm` returns a variable of type `SCREEN *` which should be saved and used each time that terminal is referenced. Two file descriptors must be present, one for input, and one for output. Use `endwin` for each terminal prior to program termination to restore previous terminal states and dismantle data structures that were created by `curses` and are no longer needed. During program operation with multiple terminals, `set_term` is used to switch between terminals.

Another initialization function is `longname` which returns a pointer to a static area containing a verbose description of the current terminal upon completion of a call to `initscr`, `newterm`, or `setupterm`.

Option Setting Routines

These routines set up options within `curses`. Arguments specify the window to which the option applies, and the boolean flag which must be `TRUE` or `FALSE` (not 1 or 0) specifies whether the option is enabled or disabled. Default for all functions in this group is `FALSE` (disabled).

- `clearok(`*win,boolean_flag*`)`, when set, clears and redraws the entire screen on the next call to `refresh` or `wrefresh`.

- `idlok(`*win,boolean_flag*`)`, when set, allows `curses` to use the insert/delete line features of the terminal if they are available. This feature tends to be visually annoying if used in applications where it is not really needed. Insert/delete character capabilities are always considered by `curses`, and are not related to insert/delete line considerations.

- `keypad(`*win,boolean_flag*`)`, when set, enables handling of special keys from the terminal keyboard as single values instead of character sequences.

- `leaveok(`*win,boolean_flag*`)`, when set, allows `curses` to ignore cursor position and relocation at the end of an operation. This feature helps simplify program operation when the cursor is not used or cursor position is not important.

- `meta(`*win,boolean_flag*`)`, when set, handles characters from the `(getch)` function as 8-bit entities instead of the usual seven. However, this feature has no value if other programs and networks interacting with the data can only pass 7-bit characters.

  This feature is useful for applications where an extended non-text character set is needed and the terminal has a meta shift key available. Curses takes whatever measures are needed to handle the 8-bit input, including the use of raw mode, if necessary. In most cases, the character size is set to 8, parity checking disabled, and 8th-bit stripping is disabled. For the data to continue unaltered, all programs using it must also be capable of handling 8-bit character codes.

- `nodelay(`*win,boolean_flag*`)`, when set, makes `getch` a non-blocking call. When enabled, `getch` returns immediately with the value -1 if no input is ready. If not enabled, the program hangs until a terminal key is pressed.

- `intrflush(`*win,boolean_flag*`)`, when set, flushes all output in the tty driver queue if an interrupt key (interrupt, quit, or suspend, if available on the system) is pressed on the terminal keyboard. While this capability provides faster interrupt response, the flush destroys the representative relationship between `curscr` and the current physical display contents.

- `typeahead(`*file_descriptor*`)`, when set, enables typeahead for the specified file where `file_descriptor` is the terminal input file. A file descriptor value of zero selects `stdin`; −1 disables typeahead checking.
- `scrollok(`*win,boolean_flag*`)`, when set, enables scrolling on the specified window whenever the cursor position exceeds the lower boundary of the window (or scrolling region, if set). Boundary crossing results when a newline occurs on the bottom line or a character is placed in the last character position of the bottom line. If `scrollok` is enabled, the window or scrolling region is scrolled up one line, and a `refresh` operation is performed to update the terminal screen. `idlok` must be enabled on the terminal to get a physical scrolling effect on the visible display. If `scrollok` is disabled, the cursor is left on the bottom line, and no advances are allowed beyond the last character position.
- `setscrreg(`*top,bottom*`)` and `wsetscrreg(`*win,top,bottom*`)` are used to set software scrolling regions within a given window. If this option and `scrollok` are both active, the scrolling region is scrolled up one line and `refresh` is called to update the screen whenever the cursor position is moved beyond the lower limit of the scrolling region in the window. To get a scrolling effect on the terminal screen, `idlok` must also be enabled.

Terminal Configuration Routines

These routines are used to set or disable various operating modes that are supported by the terminal being used.

- `cbreak()` and `nocbreak()` enable and disable single-character mode. When `cbreak` is enabled, characters are received and processed from the terminal keyboard as they are typed. When `nobreak` is active, characters are held by the tty driver until a newline key is received before making the line available to the program. Interrupt and flow control characters are not affected by either option. `cbreak` enabled is the preferred operating mode for most interactive programs. Default is `nobreak` active.
- `echo()` and `noecho()` select direct echoing of characters back to the terminal display as they are received by the tty driver, or transfer the characters to the program without returning them to the terminal

display. `noecho` can be used to process incoming text under program control then echo selected characters to a controlled area of the screen or not echo at all.

- `nl()` and `nonl()` select or disable conversion of newline characters into a carriage-return line-feed sequence on output and conversion of incoming return character(s) into newlines. By disabling newline conversions, `curses` can use line-feed capability more effectively, resulting in better cursor motion.
- `raw()` and `noraw()` select or disable raw mode. Raw mode is similar to `cbreak` in that characters are passed to the program as they are typed, but interrupt, quit, and suspend characters are not interpreted, so they do not generate a signal. Raw mode also handles characters as 8-bit entities. BREAK handling is not affected.
- `resetty()` and `savetty()` restore and save tty modes. `savetty` saves the current state in a buffer. `resetty` restores the terminal to the state that was obtained by the last previous call to `savetty`.

Window Manipulation Routines

Window manipulation routines are used to create, move, and delete windows, subwindows, and pads, and perform certain other operations. `newwin`, `newpad`, and `subwin` create new structures. `delwin` deletes window, pad, and subwindow structures, and `mvwin` relocates a window to a different area within the physical screen boundary. `touchwin`, `overlay`, and `overwrite` affect optimization and character replacement during refresh and window copying operations as follows:

- `touchwin` forces the entire window to be rewritten to the screen during refresh.
- `overlay` copies non-blank characters from one window onto the overlapping area of another.
- `overwrite` overwrites all characters from one window onto the overlapping area of another.

Pad functions are related to window functions, with some differences. Pads are essentially the same as windows but usually larger than the available screen size so that only part of the pad can be displayed at any given time. Pads

cannot be directly transferred to the terminal screen by use of window `refresh` functions. Pad refresh functions must be used instead, so that the appropriate area of the pad can be specified for display.

When a new window, subwindow, or pad is created, the function returns a pointer that should be stored in a variable for later use when accessing the window or pad. The returned variable then becomes the `win` argument for writing to the window (or pad), deleting the window (or pad), and for other text and cursor operations that include `win` as an argument. Except for `prefresh`, `pnoutrefresh`, and `newpad`, all pad operations use the appropriate window function for all text and cursor manipulations and other pad/window activities.

Terminal Data Output Routines

All data transfers from a pad or window to the terminal display are handled by pad and window refresh/update functions:

- `refresh()` and `wrefresh(`*win*`)` transfer the contents of the default or specified window to the current screen window and to the terminal display.
- `doupdate()` and `wnoutrefresh (`*win*`)` are used to accumulate several window copy operations to the standard screen window by using multiple calls to `wnoutrefresh (`*win*`)`, then transferring the current screen window to the terminal screen by calling `doupdate()`.
- `prefresh( ... )` and `pnoutrefresh( ... )` are equivalent to `wrefresh` and `wnoutrefresh`, except that the pad and area within the pad are specified. `pnoutrefresh` is followed by the `doupdate` function that is normally used with window updates.

Window Writing Routines

Placing Text in the Window

These routines are used to write data in windows, subwindows, and pads. Only the root function is listed here. Other related functions are listed with the root function in the alphabetical `curses` Routines section later in this tutorial.

Routines that use the `win` argument operate on the `stdscr` window if `win` is not specified. The cursor can be relocated before a function is executed by adding `mv` onto the beginning of the function name. This produces a `move(`*y,x*`)` or `wmove(`*win,y,x*`)` call on the default or specified window associated with the function, followed by a call to the remaining window writing routine. Row (y) and column (x) coordinates begin with (0,0) in the upper left-hand corner of the window or screen, not (1,1). Use of the `mv` prefix was also discussed earlier. See the section, Using Multiple Windows.

- `move(`*y,x*`)` and `wmove(`*win,y,x*`)` move the cursor in the given window or pad. `move(`*y,x*`)` is equivalent to `wmove(stdscr,`*y,x*`)`.
- `addch(`*ch*`)` and related functions (see `curses` routines section for related functions) write a single character in the given window or pad. `mv` prefixed to the base function name causes the current cursor/character position to be changed to the specified `y,x` location before the character is placed. Cursor position after the placement is determined by the type of character written.
- `addstr(`*str*`)` and related functions place the specified string in the selected window. `mv` prefixed to the base function name causes the current cursor/character position to be changed to the specified `y,x` location before the string is placed. Cursor position after the placement is determined by the characters contained in the written string.
- `erase()` and `werase(`*win*`)` place blanks in the entire window or pad, destroying all previous window contents.
- `clear()` and `wclear(`*win*`)` are similar to `erase()`. They erase the window by filling it with blanks, but they also call `clearok()` which clears the terminal screen on the next `refresh()` for that window.
- `clrtoeol()` and `clrtobot()` and their related window/pad functions erase the specified window/pad from the present cursor position to the end of the cursor line or to the end of the window or pad, respectively.

Inserting and Deleting Text in the Window

The following routines are used to insert and delete lines and characters in the window. These operations are performed on the window only, and have no effect on the terminal at the time of execution.

- `delch` and related window and move routines delete a single character from the current or specified new cursor position.
- `deleteln()` and `wdeleteln(`*win*`)` remove the current cursor line from the default or specified window.
- `insch(`*c*`)` and related routines insert the specified character in front of the current cursor position and move succeeding text appropriately to accommodate the new character.
- `insertln()` and `winsertln(`*win*`)` insert a blank line at the present cursor line position and move the existing cursor line (and subsequent lines) down one position. The bottom line in the window is lost. The inserted line becomes the new cursor line.

Formatted Output to the Window

`printw` is functionally similar to `printf` except the output is handled by `addch` which places the formatted data in the window.

Miscellaneous Window Operations

`scrollw(`*win*`)` is used to scroll a given window up one line each time the function is called. `box(`*win,vert,hor*`)` uses the specified characters to draw a box around the specified window. When the window is boxed, the top and bottom rows and left and right columns in the window are no longer available for normal text use.

Window Data Input Routines

Two functions are available that are used to obtain data from a given window. `getyx(`*y,x*`)` is used to obtain the present cursor position for use by the program. `inch()` and related functions can be used to retrieve any character in a given window. The returned character includes video highlighting attribute bits, each of which is set or cleared according to the original highlighting attributes that were stored with the character when it was written to the window.

Terminal Data Input Routines

`getch` and its related window and move routines are the basic building block for all program input from the terminal. `getch` handles individual characters, one at a time, returning a character as a 16-bit integer value each time it returns from a call.

If echo is enabled, `getch` also places each character at the current cursor position in the window associated with the function and updates the terminal screen with a `refresh` on the window as the character is received and processed (the cursor is advanced as each character is written to the window). If `noecho` is active instead, input character(s) are not placed in the window.

`getstr` and its related functions generate a series of calls to `getch` to read an entire line, one character at a time, up to the terminating newline character. The line is stored in the specified string before `getstr` returns to the calling program.

`scanw` and its related functions perform formatted processing on the input line after it has been placed in a special buffer used by `getstr`. (If echo is enabled, the string is also placed in the associated window, but only the characters stored in the buffer are used by `scanw`. When scanning is complete, the processed results string results are placed in the specified `args` variables.

Video Highlighting Attribute Routines

Each character written into a window is stored as a 32-bit word. The lower eight bits contain the character code; the middle eight bits contain the NLS attributes; the remaining nine bits control video highlighting. As each word is stored, the 8-bit character code is combined (through a bit-level logical OR operation) with the current set of nine video highlighting attributes to obtain the 32-bit result. Video attribute routines are used to construct the current attribute set that is used during character storage.

Highlighting attributes can be specified as a complete set by using `attrset` or `wattrset`. Using 0 (or `A_NORMAL`) as an argument for `attrset` disables all highlighting.

Highlighting can be altered from the present state by turning individual attributes on or off without altering the state of other attributes in the set. This is done with `attron`, `attroff`, `wattron`, and `wattroff`.

As characters are stored in a given window, the current attributes are attached to each character. To change highlighting, attributes must be changed before the next character is written to the window. When deciding where to change highlighting attributes, remember that highlighting applies to non-printing space and tab characters as well as visible characters.

`standout` and `standend` provide easy access to the `A_STANDOUT` attribute. `standout` is equivalent to a call to `attron(A_STANDOUT)`, and adds `A_STANDOUT` to the currently active set of attributes (if any are active). However, `standend` is not the opposite. `standend` is equivalent to `attrset(0)`, not `attroff(A_STANDOUT)`. Thus, a call to `standout` with underlining on would maintain underlining until another highlighting call. `standend`, on the other hand, would not only terminate the previous `standout` call, but would terminate underlining as well.

Attribute functions and arguments must be logically conceived. For example, `attron(A_NORMAL)` and `attroff(A_NORMAL)`, though executable, do nothing because all bits in `A_NORMAL` are cleared (value is zero). The bit-level logical OR of `attron` has no effect (all bits zero), and `attroff` is ineffectual because `A_NORMAL` is inverted (all bits set to 1) before a bit-level logical AND is used to clear the selected highlighting attribute.

Miscellaneous Functions

beep/flash

`beep()` and `flash()` are used to signal the terminal operator. If the terminal does not support the called function, the other is substituted where possible. Thus a call to `beep` flashes the screen if the terminal has no beep capability; a call to `flash` produces a beep if no flashing video capability is available.

Portability Functions

Several functions have been included to aid portability of `curses` between various systems:

- `baudrate()` returns the terminal datacomm line speed as an integer baud rate value. The returned value can then be used for program and system configuration purposes.

- `erasechar()` returns the terminal erase character that has been chosen by the user. This character is used to cancel the last previous character. Interactive programs should include cancellation capabilities so users can correct typographical errors during keyboard inputs.
- `killchar()` is similar to the erase character, but cancels the entire line where the character appears.
- `flushinp()` discards any typeahead characters when an interrupt character is detected. This enables users to interrupt a series of commands or other activities that have accumulated in the typeahead buffer and terminate the current process without waiting for the typeahead queue to empty. Normally used for aborts, this function and the related program structure must be handled carefully to ensure proper termination of program processes before the program exits.

Delay Functions

Delay functions are not highly portable, but are frequently needed by programs that use `curses`, especially real-time interactive response programs. Use of these functions should be avoided where possible:

- `draino(`*ms*`)` is used to reduce the amount of data being held in the output queue. The main purpose of this function is to keep the program (and keyboard) from getting ahead of the screen. With careful program design, use of this function should be unnecessary in most cases.
- `napms(`*ms*`)` suspends program operation for a specified time. It is similar to `sleep`, but offers higher resolution (resolution varies, depending on system resources). `napms` uses a call to `select` for its time base reference.

2

Curses Routines

`curses` supports the following functions. Those marked with an asterisk are also supported by `mini-curses` (some unmarked routines might work, but are not officially supported by `mini-curses`. Proceed at your own risk if you try them).

addch(*ch*)*
addstr(*str*)*
attroff(*attrs*)*
attron(*attrs*)*
attrset(*attrs*)*
baudrate()*
beep()*
box(*win,vert,hor*)
cbreak()*
clear()
clearok(*win,boolean_flag*)
clrtobot()
clrtoeol()
delay_output(*ms*)*
delch()
deleteln()
del_term(*oterm*)
delwin(*win*)
doupdate()
draino(*ms*)
echo()*
endwin()*
erase()
erasechar()
fixterm()
flash()
flushinp()*
mvprintw(*y,x,fmt,args*)
mvscanw(*y,x,fmt,args*)
mvwaddch(*win,y,x,ch*)
mvwaddstr(*win,y,x,str*)
getch()*
getstr(*str*)
gettmode()
getyx(*win,y,x*)
has_ic()
has_il()
idlok(*win,boolean_flag*)
inch()
initscr()*
insch(*c*)
insertln()
intrflush(*win,boolean_flag*)
keypad(*win,boolean_flag*)
killchar()
leaveok(*win,boolean_flag*)
longname()
meta(*win,boolean_flag*)*
move(*y,x*)*
mvaddch(*y,x,ch*)
mvaddstr(*y,x,str*)
mvcur(*oldrow,oldcol,newrow,newcol*)
mvdelch(*y,x*)
mvgetch(*y,x*)
mvgetstr(*y,x,str*)
mvinch(*y,x*)
mvinsch(*y,x,c*)
setscrreg(*t,b*)
set_curterm(*nterm*)
set_term(*new*)
setterm(*type*)
setupterm(*term,filenum,errret*)

mvwdelch(*win,y,x*)
mvwgetch(*win,y,x*)
mvwgetstr(*win,y,x,str*)

mvwin(*win,beg_y,beg_x*)
mvwinch(*win,y,x*)
mvwinsch(*win,y,x,c*)
mvwprintw(*win,y,x,fmt,args*)
mvwscanw(*win,y,x,fmt,args*)
napms(*ms*)
newpad(*num_lines,num_col*)
newterm(*type,fdout,fdin*)
newwin(*num_lines,num_cols,beg_x,beg_y*)
nl()*
nocbreak()*
nodelay(*win,boolean_flag*)
noecho()*
nonl()*
noraw()*
overlay(*win1,win2*)
overwrite(*win1,win2*)
pnoutrefresh(*pad,pminrow,pmincol, sminrow,smincol,smaxrow,smaxcol*)
printw(*fmt,args*)
raw()*
refresh()*
resetterm()*
resetty()*
saveterm()*
savetty()*
scanw(*fmt,args*)
scroll(*win*)
scrollok(*win,boolean_flag*)
standend()*
standout()*
subwin(*orig_win,n_lines,n_cols, beg_y,beg_x*)
touchwin(*win*)
traceoff()
traceon()
typeahead(*fd*)
unctrl(*ch*)*
waddch(*win,ch*)
waddstr(*win,str*)
wattroff(*win,attrs*)
wattron(*win,attrs*)
wattrset(*win,attrs*)
wclear(*win*)
wclrtobot(*win*)
wclrtoeol(*win*)
wdelch(*win,c*)
wdeleteln(*win*)
werase(*win*)
wgetch(*win*)
wgetstr(*win,str*)

winch(*win*)
winsch(*win,c*)
winsertln(*win*)
wmove(*win,y,x*)
wnoutrefresh(*win*)
wprintw(*win,fmt,args*)
wrefresh(*win*)
wscanw(*win,fmt,args*)
wsetscrreg(*win,t,b*)
wstandend(*win*)
wstandout(*win*)

Description of Routines

The `curses` package includes the following functions. Function names that are associated with operations on user-specified windows contain a `w` or `mvw` prefix, and the window must be included as a parameter in the function call. If no `w` or `mvw` prefix is present, or if the window is not specified in the parameter set, the operation is performed on the default window `stdscr`. Programs that use the `curses` package are subject to the normal rules of C compiler statement syntax.

Routines are listed alphabetically by function keyword which is printed in slanted bold type. When two or more functions are related to a common keyword, the root keyword is listed in bold, followed by a list of related function names in normal italics. The individual related functions are also included elsewhere in the list with references back to the root keyword where a detailed explanation of all keywords related to the root keyword is located.

`addch(`*ch*`)`
`waddch(`*win,ch*`)`
`mvaddch(`*y,x,ch*`)`
`mvwaddch(`*win,y,x,ch*`)`

Places character *ch* in window at current cursor position for that window then advances cursor to next position. If *ch* is a tab, newline, backspace, the cursor is moved appropriately, but no text is altered. If `ch` is a control character other than tab, newline, or backspace, the character is drawn using `^`*x* `notation (where` *x* `is a printable character preceded by ^` to indicate a control character – see `unctrl(`*ch*`)`). If the character is placed at the right margin, an automatic newline is performed. At the bottom of the scrolling region, the region is scrolled up one line if `scrollok` is enabled.

ch parameter is an integer; not a character. `addch` performs a bit-level logical OR between the 16-bit character and the current attributes if any are active. Highlighting of individual characters can also be handled by the program if the current attributes are all zero (disabled) by performing an equivalent bit-level logical OR operation between the 7-bit character code in bit positions 0 through 6 and selected video attribute bits in bit positions 7 through 15 to create a single 16-bit integer representing the character and its associated highlighting attributes. If no highlighting attributes for the window are currently active, any attributes added to the character by the program or already present from the source are preserved. If any are active, they are added to the character and any attached attributes without altering other attributes. Thus, you can copy text (including attributes) from one place to another with `inch` and `addch`.

`addch` is used with *stdscr* window; `waddch` with window *win;* `mvaddch` moves the cursor to row *y,* column *x,* then places the character at that location; `mvwaddch` is identical to `mvaddch`, but operates on a specified window *win.* If *win* is not specified, default is to *stdscr.* All row and column references are relative to the upper left corner whose corner character position is represented by row 0, column 0.

Description of Curses Routines (Continued)

`addstr(`*str*`)`
`waddstr(`*win,str*`)`
`mvaddstr(`*y,x,str*`)`
`mvwaddstr(`*win,y,x,str*`)`
	Places the character string specified by *str* at the current cursor position (`addstr` and `waddstr`) or at the specified location in the window (`mvaddstr` and `mvwaddstr`). String placement consists of a series of character placements using the `addchx` routine. *str* must be terminated by a null character.

`attroff(`*attrs*`)` `wattroff(`*win,attrs*`)`	Disables the specified video highlighting attributes without affecting other attributes. Any or all of the following attributes can be specified (multiple attributes must be separated by the C logical OR operator, | which performs a bit-level logical OR on all attributes specified in the function call): `A_STANDOUT`, `A_UNDERLINE`, `A_REVERSE`, `A_BLINK`, `A_DIM`, `A_BOLD`, `A_INVIS` (invisible), `A_PROTECT`, and `A_ALTCHARSET`.
`attron(`*attrs*`)` `wattron(`*win,attrs*`)`	Enables the specified video highlighting attributes without affecting other attributes. Any or all of the following attributes can be specified (multiple attributes must be separated by the C logical OR operator, | which performs a bit-level logical OR on all attributes specified in the function call): `A_STANDOUT`, `A_UNDERLINE`, `A_REVERSE`, `A_BLINK`, `A_DIM`, `A_BOLD`, `A_INVIS` (invisible), `A_PROTECT`, and `A_ALTCHARSET`.
`attrset(`*attrs*`)` `wattrset(`*win,attrs*`)`	Enables specified video highlighting attributes, and disables all others. Any or all of the following attributes can be specified (multiple attributes must be separated by the C logical OR operator, | which performs a bit-level logical OR on all attributes specified in the function call): `A_STANDOUT`, `A_UNDERLINE`, `A_REVERSE`, `A_BLINK`, `A_DIM`, `A_BOLD`, `A_INVIS` (invisible), `A_PROTECT`, and `A_ALTCHARSET`. `attrset(0)`, `attrset(A_NORMAL)`, and `standend()` (or `standend(`*win*`)`) are equivalent functions that disable all attributes (normal display). See `standend()`.

Description of Curses Routines (Continued)

`baudrate()`	Returns the terminal serial I/O datacomm speed. The value returned is the integer baud rate (such as 9600) rather than a table index value (such as B9600). If the baud rate is External A or External B, the value −1 is returned instead.
`beep()`	Used to signal the terminal user with an audible signal. If no audible signal is available on the terminal, the screen is flashed instead (see `flash()`). If neither capability is available, no output is sent to the terminal.
`box(`*win,vert,hor*`)`	Draws a box around the specified window. `vert` specifies the character to be used for left and right columns; `hor` specifies the character for top and bottom rows. Usable window space is reduced by two lines and columns when a box is present.
`cbreak()` `nocbreak()`	These functions place the terminal in and out of `CBREAK` mode, respectively. When `cbreak` (character-mode operation) is active, each typed character is immediately available to the program. If disabled (`nocbreak`), the tty driver holds characters until a newline character is received, then releases the entire line to the program (line-mode operation). Interrupt and flow control characters are not affected by `cbreak`; default is `nocbreak`, but most interactive programs that use `curses` run with `cbreak` enabled.
`clear()` `wclear(`*win*`)`	Similar to `erase` and `werase`, but `clearok` is also called so that the terminal screen is cleared by the next call to `refresh` for that window. `clearok` sets a flag to clear the screen, blanks are placed in the window, and the next call to `refresh` outputs a screen clearing operation or blanks or both to the terminal, depending on terminal capabilities.

`clearok(`*win, boolean_flag*`)`	If set, the next `wrefresh` call for the specified window clears and redraws the entire screen (instead of just the area represented by the specified window). If `win` specifies `curscr`, the next call to `wrefresh` for *any* window clears and redraws the entire screen. This is useful when current screen contents are uncertain, or in some cases for a more pleasing visual effect.
`clrtobot()` `wclrtobot(`*win*`)`	Clears all character positions from the current cursor position to the right margin, and all lines below the current cursor line to the end of the window.
`clrtoeol()` `wclrtoeol(`*win*`)`	Clears all character positions from the current cursor position to the right margin. The rest of the window remains undisturbed.
`delay_output(`*ms*`)`	See `terminfo` routines in the next section of this chapter.

`delch()`
`wdelch(`*win*`)`
`mvdelch(`*y,x*`)`
`mvwdelch(`*win,y,x*`)`
	The character at the present cursor position is deleted. All remaining characters on the line to the right of the deleted character are moved left one position. Other lines are not disturbed. The operation is performed only on the window, and does not use the terminal hardware delete-character feature because no terminal operation has been performed.

`deleteln()` `wdeleteln(`*win*`)`	The present cursor line is deleted. All remaining lines in the window below the cursor line are moved up one position, leaving a blank line at the bottom of the window. This window operation does not interact directly with the terminal when performed, so no terminal hardware delete-line feature is used.
`del_term(`*oterm*`)`	*oterm* is of type TERMINAL*. `del_term()` frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as `cur_term`, then references to any of *terminfo(4)* boolean, numeric and string variables thereafter may refer to invalid memory locations until another `setupterm()` has been called.
`delwin(`*win*`)`	Deletes the specified window and releases all memory associated with it. If the window contains subwindows, all subwindows must be deleted first.

`doupdate()`
`wnoutrefresh(`*win*`)`
`pnoutrefresh(`*pad,...*`)`
	`wnoutrefresh` (or `pnoutrefresh`) and `doupdate` essentially divide `wrefresh` into two independent functions that can be called separately for more efficient handling of multiple output operations to windows and pads. In normal operation, `wrefresh(`*win*`)` calls `wnoutrefresh(`*win*`)` to copy the named window to the virtual screen, then uses `doupdate` to update the physical screen to match the virtual screen. When outputting multiple windows, `wnoutrefresh(`*win*`)` can be used successively, once for each window; followed by a single `doupdate()` to transfer the new screen to the terminal, probably with fewer characters transmitted. `pnoutrefresh` is used similarly when writing to pads.

`draino(`*ms*`)`	Suspends program operation until the output queue has been reduced sufficiently ("drained") so that the remaining characters can be transmitted in not more than *ms* milliseconds. For example, `draino`(50) at 1200 baud would suspend program execution until no more than 6 characters remain to be sent (6 characters @ 1200 baud require about 50 ms transmit time). This routine is used to keep the program (and thus the keyboard) from getting ahead of the screen. If the operating system does not support the I/O controls (`ioctls`) that are needed to implement `draino`, the value ERR is returned; otherwise OK is returned.
`echo()` `noecho()`	Enables or disables echoing of characters by `getch` through the specified window and back to the terminal as each character is typed on the keyboard and subsequently processed by `getch`. Default is `echo` (enabled). In some interactive programs, it is preferable to suppress echoing by `getch` (`noecho`), then let the program place incoming characters in a controlled area of the screen or not return them at all, as needs dictate.

Description of Curses Routines (Continued)

`endwin()`	`endwin` should always be called before exiting from a `curses`-based program. Restores tty modes, places the cursor in the lower left corner of the terminal screen, resets the terminal into the proper non-visual mode, and removes data structures that are no longer needed by the exiting program.
`erase()` `werase(`*win*`)`	Copies blanks to every character position in the specified or default window. As each blank is stored in the window, the highlighting attribute bits are set to zero (disabled).
`erasechar()`	Returns the user's chosen erase character from the terminfo data base. The returned character should be interpreted by the program as an "erase previous character" command whenever it is received from the terminal.
`fixterm()`	Restores the current terminal to the state it was in prior to the most recent call to `resetterm()`. State information stored by the most recent previous call to `saveterm()` provides the needed restoration information. See `resetterm()`.
`flash()`	Used to signal the terminal user by flashing the screen. If the terminal has no screen flashing feature, the audible signal is sounded instead (see `beep()`). If neither capability is available, no output is sent to the terminal.
`flushinp()`	Discards any typeahead characters in the typeahead buffer (characters that have been typed on the terminal but are still waiting to be handled by the program.

`getch()`
`wgetch(`*win*`)`
`mvgetch()`
`mvwgetch(`*win*`)`

Takes a character from the terminal keyboard input buffer as a 16-bit integer, processes it, and returns it to the program as a 16-bit integer. Character processing and return conditions vary as follows:

If `mv` is placed in front of `getch` or `wgetch`, the cursor position for the selected window is moved to the specified location which becomes the new current cursor position. This operation is completed before any character processing begins.

If `echo` is active and the character is a normal typing character (keypad and meta characters are discussed later), the character is placed in the current cursor position by a call to `waddch` from `getch`. During character placement in the window, a bit-level logical OR in `waddch` attaches current highlighting attributes to the character. `waddch` is followed immediately by a call to `wrefresh` which updates the terminal screen with the echo character.

If an escape character is received, special timeouts are set up to determine whether the character is part of a multiple-character keypad sequence. See Use of Escape in Program Control topic earlier in this tutorial for a detailed discussion of how escape is handled.

If `meta` is *enabled* and the character is not a keypad sequence, the 16-bit input character is logical ORed with octal 0377 to mask the upper bits to zero and return an 8-bit text character value. The eighth bit interferes with the `A_STANDOUT` highlighting attribute bit in the same position, so `noecho` is usually chosen for programs that operate with meta active.

If `meta` is *not enabled,* text characters are logical ORed with octal 0177 to mask the upper bits to zero and return a 7-bit character value. Echoing is handled in the normal manner if enabled.

If `keypad` is *not enabled,* function key sequences are treated as individual characters and handled as normal text.

If keypad is *enabled,* each function key sequence (usually an escape sequence) is handled as a single-character keycode which is assigned a 16-bit integer value in a range beginning at 0401 (octal) and a name that starts with *KEY_* (a complete list of keypad character value and name definitions is included in the keypad discussion near the beginning of this tutorial). The character value is *not* placed in the window for echoing, even if echo is enabled.

If nodelay is active: if no input is available in the keyboard input buffer when getch is called, getch returns with the value −1 and no other action is taken. If nodelay is not active, the program hangs until text is available in the buffer. Depending on the current cbreak setting, text is made available to the program as each character is received (cbreak), or incoming characters are held by the tty driver until a newline is received then they are made available to the program (nocbreak).

`getstr(`*str*`)`
`wgetstr(`*win,str*`)`
`mvgetstr(`*y,x,str*`)`
`mvwgetstr(`*win,y,x,str*`)`

This routine is used to input an entire line from the terminal. It is equivalent to getch, except that it handles an entire string instead of single characters. Handling of each character is identical to getch except that text and meta characters are packed into the string variable *str* instead of being returned to the program as individual 16-bit integers. Keypad characters (except for kill, erase, key_left (left arrow), and backspace) are not recognized and cannot be handled through getstr.

During execution, getstr generates a series of calls to getch until a newline is received, at which time it returns. The 16-bit integers returned by successive calls to getch are stripped of their unneeded upper bits (except recognized keypad keys) before packing into a string variable beginning at the location identified by the character pointer *str.*

If echo is enabled, incoming string characters are also placed in the associated window (by getch) as they are received and processed, and echoed to the terminal (by refresh). If noecho is active, characters are not placed in the window; they are only placed in *str.*

Description of Curses Routines (Continued)

`gettmode()`	(Get tty mode). Dummy entry point. Performs no useful function.
`getyx(`*win,y,x*`)`	Places the current cursor position of the specified window in the specified two integer variables `y` and `x`. This is a macro, so no `&` is necessary.
`has_ic()`	Returns a value indicating whether or not the terminal has insert/delete character capability. Zero value indicates the capability is not present; non-zero: capability present.
`has_il()`	Returns a value indicating whether or not the terminal has insert/delete line capability. Zero value indicates the capability is not present; non-zero: capability present.
`idlok(`*win,boolean_flag*`)`	Insert and Delete Line OK. If enabled, `curses` can use hardware insert/delete line capabilities when the terminal is so equipped. If disabled, `curses` does not use the capability. Use only when the program requires it (such as a screen editor). `idlok` is disabled by default because it tends to be annoying when used in applications where it is not really needed. If insert/delete line cannot be used, `curses` redraws changed portions of all lines that do not match the desired result.

`inch()`
`winch(`*win*`)`
`mvinch(`*y,x*`)`
`mvwinch(`*win,y,x*`)`
	Returns the character located at the current or specified position in the specified window as a 16-bit integer. If any attributes are set for that position, their values are included in the value returned. To extract only the character or the attributes, perform a bit-level logical AND on the returned value, using the predefined constant `A_CHARTEXT` (octal 0177) or `A_ATTRIBUTES` (octal 0177600).

`initscr()`	The first function called in `curses`-based programs. Determines terminal type, and initializes `curses` data structures as appropriate. Also sets indicators so that the first call to `refresh` clears the terminal screen and updates `curscr` to reflect the cleared screen.

Description of Curses Routines (Continued)

`insch(`*c*`)`
`winsch(`*win,c*`)`
`mvinsch(`*y,x,c*`)`
`mvwinsch(`*win,y,x,c*`)`
	Inserts the character (byte, usually a 7-bit code) specified by *c* at the current cursor position position or at the specified location in the standard or specified window (current attributes are attached during the placement operation). All characters beginning at the insertion location are moved right one position for the remainder of the line. If the line is full, the rightmost character is discarded. This does not interact with the terminal so no hardware insert-character feature is used.

`insertln()` `winsertln(`*win*`)`	Inserts a blank line between the current cursor line and the line above it. The current line and subsequent lines of text in the window are moved down one position, and the blank line becomes the new current cursor line. The bottom line of text is discarded if it cannot fit inside the window. This is a window operation that does not interact with the terminal, so no hardware insert-line feature is used.
`intrflush(`*win, boolean_flag*`)`	Causes tty driver queue to be flushed on interrupt. When enabled, an interrupt, quit, or suspend keypress from the terminal flushes all output from the tty driver queue, providing a faster response to the interrupt. However, `curses` loses its record of what is currently displayed on the screen when the interrupt occurs. Disabling the option prevents the flush. Default is flush enabled. Requires proper support from the underlying driver.
`keypad(`*win, boolean_flag*`)`	Enables keypad character handling for the user terminal associated with `win`. When true, the terminal operator can press any key that generates multiple-character sequences (such as a function key), and `getch` returns a single 16-bit integer value representing the function key (the returned character must be handled as a 16-bit value). If `keypad` is disabled (default), `curses` handles keypad sequences as normal text. `keypad` also enables and disables keypad keys on the terminal if the terminal hardware is equipped to support such command sequences from the external computer.

`killchar()`	Returns the line-kill character chosen by the terminal user. This character, when typed by the user, is a command to the program to cancel the entire line being typed.
`leaveok(`*win, boolean_flag*`)`	Upon completion of normal `refresh` operations (`leaveok` disabled) the terminal hardware cursor is placed at the current cursor location for the window being refreshed. A call to `leaveok(`*win,* `TRUE)` prior to `refresh` allows refresh operations to leave the terminal hardware cursor in any convenient position instead of updating it to the current window cursor position when refresh is finished. This is useful for applications where the cursor is not used because it reduces the need for cursor movements. When possible, the cursor is made invisible when `leaveok` is specified for the window. Once `leaveok` is set `TRUE` for a given window, it remains active for the duration of the program or until another call sets it `FALSE`.
`longname()`	Returns a pointer to a static area containing a verbose description of the current terminal. This static area is defined only after a call to `initscr`, `newterm`, or `setupterm`.
`meta(`*win,boolean_flag*`)`	When enabled, text characters are returned by `getch` as 8-bit character codes (masked by octal 0377) instead of 7-bit (masked by octal 0177) characters. Returns the value `OK` if the request succeeds; `ERR` if the terminal or system cannot handle 8-bit character codes. `meta` is useful for extending the non-text command set in applications where the terminal has a meta shift key. `curses` takes whatever measures are necessary to arrange for 8-bit input. When `meta` is true, HP-UX sets datacomm configuration to 8-bit character length, no parity checking, and disables 8th-bit stripping. Remember that if any program or facility handling the data can only pass 7-bit codes or strips the 8th bit, 8-bit handling is not possible.

Description of Curses Routines (Continued)

`move(`*y,x*`)` `wmove(`*win,y,x*`)`	Places the cursor associated with the specified or default window at the specified row (`y`) and column (`x`) where the upper left corner of the window is row 0, column 0. The cursor is not moved on the display screen until a `refresh` or equivalent function is executed.
`mvaddch(`*y,x,ch*`)`	Same as `move(`*y,x*`)`; `addch(`*ch*`)`. See `addch(`*ch*`)`.
`mvaddstr(`*y,x,str*`)`	Same as `move(`*y,x*`)`; `addstr(`*str*`)`. See `addstr(`*str*`)`.
`mvcur(`*oldrow,oldcol, newrow,newcol*`)`	Optimally moves the cursor from (*oldrow, oldcol*) to (*newrow, newcol*). The user program is expected to keep track of the current cursor position. Unless a full-screen image is kept, `curses` must make pessimistic assumptions that sometimes result in less than optimal cursor motion. For example, if the cursor needs to be moved a few spaces to the right, the task could be accomplished by retransmitting the characters between the present and the desired position; but if `curses` cannot access the screen image, it cannot determine what those characters are.
`mvdelch(`*y,x*`)`	Same as `move(`*y,x*`)`; `delch()`. See `delch()`.
`mvgetch(`*y,x*`)`	Same as `move(`*y,x*`)`; `getch()`. See `getch()`.
`mvgetstr(`*y,x,str*`)`	Same as `move(`*y,x*`)`; `getstr(`*str*`)`. See `getstr(`*str*`)`.
`mvinch(`*y,x*`)`	Same as `move(`*y,x*`)`; `inch()`. See `inch()`.
`mvinsch(`*y,x,c*`)`	Same as `move(`*y,x*`)`; `insch(`*c*`)` . See `insch(`*c*`)`.
`mvprintw(`*y,x,fmt,args*`)`	Same as `move(`*y,x*`)`; `printw(`*fmt,args*`)`. See `printw(`*fmt,args*`)`.
`mvscanw(`*y,x,fmt,args*`)`	Same as `move(`*y,x*`)`; `scanw(`*fmt,args*`)`. See `scanw(`*fmt,args*`)`.
`mvwaddch(`*win,y,x,ch*`)`	Same as `wmove(`*win,y,x*`)`; `waddch(`*win,ch*`)`. See `addch(`*ch*`)`.
`mvwaddstr(`*win,y,x,str*`)`	Same as `wmove(`*win,y,x*`)`; `waddstr(`*win,str*`)`. See `addstr(`*str*`)`.
`mvwdelch(`*win,y,x*`)`	Same as `wmove(`*win,y,x*`)`; `addch(`*ch*`)`. See `delch()`.

Description of Curses Routines (Continued)

`mvwgetch(`*win,y,x*`)`	Same as `wmove(`*win,y,x*`)`; `wgetch(`*win*`)`. See `getch()`.
`mvwgetstr(`*win,y,x,str*`)`	Same as `wmove(`*win,y,x*`)`; `wgetstr(`*win,str*`)`. See `getstr(`*str*`)`.
`mvwin(`*win,beg_y,beg_x*`)`	Moves the specified window so that the upper left-hand corner is located at character position (*beg_y, beg_x*). If the move causes any part of the relocated window to lie outside the physical screen boundary, the command is considered to be in error, and the window remains in its original location.
`mvwinch(`*win,y,x*`)`	Same as `wmove(`*win,y,x*`)`; `winch(`*win*`)`. See `inch()`.
`mvwinsch(`*win,y,x,c*`)`	Same as `wmove(`*win,y,x*`)`; `winsch(`*win,c*`)`. See `insch(`*c*`)`.
`mvwprintw(`*win,y,x, fmt,args*`)`	Same as `wmove(`*win,y,x*`)`; `wprintw(`*win,fmt,args*`)`. See `printw(`*fmt,args*`)`.
`mvwscanw(`*win,y,x, fmt,args*`)`	Same as `wmove(`*win,y,x*`)`; `wscanw(`*win,fmt,args*`)`. See `scanw(`*fmt,args*`)`.
`napms(`*ms*`)`	Suspends program operation for `ms` milliseconds. `napms` is similar to `sleep`, but has higher resolution. The resolution actually provided depends on the resolution of available operating system facilities. If a resolution of at least 0.100 sec is not available, the routine rounds to the next higher second, calls sleep, and returns ERR. Otherwise the value OK is returned.
`newpad(`*num_lines, num_cols*`)`	Creates a new **pad** data structure. A pad is similar to a window, but it is not restricted by physical screen size nor is it associated with a particular part of the screen. Pads are useful when a large window is needed and only part of the window will be displayed at any given time. Automatic refreshes from pads (such as scrolling or input echo) do not occur. Refresh cannot be used with a pad as an argument. Instead, the routines `prefresh` and `pnoutrefresh` are used. Pad refresh routines require additional parameters to specify what part of the pad to display, and where to display it on the screen.

Description of Curses Routines (Continued)

`newterm(`*type,fpout,fpin*`)`	Used instead of `initscr` in programs that output to more than one terminal. `newterm` should be called once for each terminal. It returns a variable of type `struct screen*` which should be saved for use as a reference to that terminal. Arguments are: a string defining the terminal type, a file pointer for the output file, and another for the input file if needed (interactive terminal).
`newwin(`*num_lines, num_cols,beg_y,beg_x*`)`	Create a new window with the specified number of lines and columns whose upper left-hand corner is located at the specified row and column of the physical screen, and return a window pointer (the upper left-hand corner of the physical screen is row 0, column 0). If the number of lines and/or columns is specified as zero, the default value `LINES` minus *beg_y* and `COLS` minus *beg_x* is used instead. A screen buffer for the window is also created. To create a new full-screen window, use `newwin(0,0,0,0)`.
`nl()` `nonl()`	Defines handling of newline characters. When enabled (`nl`), newline is translated into a carriage-return and line-feed on output, and carriage-return is translated into a newline character on input. `curses` initially enables newline, but if it is disabled by `nonl`, `curses` can make better use of line feed capability, resulting in faster cursor motion.
`nocbreak()`	See `cbreak()`.
`nodelay(`*win, boolean_flag*`)`	Makes `getch` a non-blocking call. When enabled, if no input is ready, a call to `getch` returns −1. If disabled, `getch` hangs until a key is pressed.
`noecho()`	See `echo()`.
`nonl()`	See `nl()`.
`noraw()`	See `raw()`.

Description of Curses Routines (Continued)

`overlay(`*win1,win2*`)` `overwrite(`*win1,win2*`)`	Copies *win1* onto *win2* for all screen area where the two windows overlap. `overlay` copies only visible (non-blank) text, and does not disturb those *win2* character positions where *win1* is blank. `overwrite` copies all of overlapping *win1* onto *win2*, including blanks, thus destroying all original data in the overlapping area of *win2*.
`overwrite(`*win1,win2*`)`	See `overlay`.
`pnoutrefresh(`*pad, pminrow,pmincol, sminrow,smincol, smaxrow,smaxcol*`)`	See `prefresh`.
`prefresh(`*pad, pminrow,pmincol, sminrow,smincol, smaxrow,smaxcol*`)` `pnoutrefresh`(same parameters)	Analogous to `wrefresh` and `wnoutrefresh`, except that pads are involved instead of windows. Additional parameters specify what part of the pad and screen are to be used. *pminrow* and *pmincol* identify the upper left corner of the pad area to be displayed. *sminrow, smincol, smaxrow,* and *smaxcol* define the display boundaries on the physical screen. The lower right-hand corner of the pad area being displayed is calculated from the screen boundary parameters because both rectangles must be the same size. Both rectangles must lie completely within their respective structures.

`printw(`*fmt,args*`)`
`wprintw(`*win,fmt,args*`)`
`mvprintw(`*y,x,fmt,args*`)`
`mvwprintw(`*win,y,x, fmt,args*`)`
	These commands are functionally equivalent to `printf`. Characters that would normally be output by `printf` are instead output by `waddch` on the associated window.

`raw()` `noraw()`	Places the terminal in or out of raw mode. Raw mode is similar to `cbreak` mode in that characters are immediately passed to the user program as they are typed on the terminal keyboard, except that interrupt and quit characters are passed as normal text instead of generating a special interrupt signal. Raw mode handles all terminal I/O as 8-bit characters instead of 7. Break key behavior may vary, depending on the terminal.

Description of Curses Routines (Continued)

`refresh()` `wrefresh(`*win*`)`	These functions output window data to the terminal (other routines only manipulate data structures). `wrefresh` copies the named window to the physical screen on the terminal by using `wnoutrefresh(`*win*`)` followed by `doupdate()`, taking into account what is already on the screen in order to optimize the transfer. `refresh()` is similar, except it uses `stdscr` as the default screen. Unless `leaveok` is enabled, the cursor is placed at the location of the window cursor when the operation is complete.

`resetterm()`
`saveterm()`
`fixterm()`
	`resetterm` restores the current terminal to the operating condition it was in when `curses` was started. The "current `curses` state" is saved by `saveterm()` for possible future use by `fixterm()`. `resetterm` and `fixterm` should be used in all shell escapes. Equivalent routines are also available at the `terminfo` level.

`resetty()` `savetty()`	Restores (resets) the tty modes to those stored in the buffer by the last previous `savetty()` command. This means that only one set of states can be stored at any given time. See `savetty()`.
`saveterm()`	Preserves the current terminal `curses` state for possible future use by `fixterm`. See `resetterm()` and `fixterm()`.
`savetty()`	Saves the current state of the tty modes in a buffer for possible later use by `resetty()`. See `resetty()`.

`scanw(`*fmt,args*`)`
`wscanw(`*win,fmt,args*`)`
`mvscanw(`*y,x,fmt,args*`)`
`mvwscanw(`*win,y,x,*
fmt,args`)`
	Corresponds to `scanf` (see *scanf*(3S) entry in the *HP-UX Reference*). Calls `wgetstr` which inputs characters from the terminal and places them in a buffer until newline is received. When newline is received, the string in the buffer serves as input for the scan which processes the buffered string and places the result in the appropriate `args`. Uses `getch` for character input and echo handling.

Description of Curses Routines (Continued)

`scroll(`*win*`)`	Scrolls the window up one line by moving the lines in the window data structure. As an optimization, if the window being scrolled is `stdscr`, and the scrolling region is the entire window, the physical screen is scrolled at the same time.
`scrollok(`*win, boolean_flag*`)`	Controls window handling when the cursor advances beyond the bottom boundary of the window or scrolling region due to a newline in the bottom line or a character placed in the last character position of the bottom line. If scrolling is disabled, the cursor is left on the bottom line (characters are accepted until the bottom line is full, but newlines are ignored). If the cursor crosses the bottom boundary while `scrollok` is enabled, a `wrefresh` is performed on the window, then the window and terminal are scrolled up one line. `idlok` must also be called before a physical scrolling effect can be produced on the terminal screen.
`setscrreg(`*t,b*`)` `wsetscrreg(`*win,t,b*`)`	Sets up a software scrolling area in window `win` or `stdscr`. `t` and `b` are the top and bottom lines of the scrolling region (line 0 is the top line of the window). If this option and `scrollok` are both enabled, an attempt to move off the bottom margin causes all lines in the scrolling region to scroll up one line. Note that this process has nothing to do with the physical scrolling region capability that exists in some terminals (only the text in the window is scrolled). If the terminal has scrolling region or insert/delete line capabilities, they will probably be used by the output routines during refresh. `idlok` must be enabled before a scrolling effect can be produced on the terminal screen (see `scrollok`).
`setterm(`*type*`)`	Low-level interface used by old `curses` and included here for compatibility with earlier software.
`setupterm(`*term, filenum,errret*`)`	`terminfo` routine. See `terminfo` routines in the next section of this tutorial for description.

Description of Curses Routines (Continued)

`set_curterm(`*nterm*`)`	*nterm* is of type TERMINAL*. `set_curterm()` sets the variable `cur_term` to *nterm* and makes all the *terminfo(4)* boolean numeric and string variables use the values from *nterm*.
`set_term(`*new*`)`	Switches to a different terminal. The screen reference `new` becomes the new current terminal, and the function returns the previous terminal. All other calls affect only the current terminal. This function is used to handle multiple terminals interacting with a single program.
`standend()` `wstandend(`*win*`)`	Equivalent to `attrset(0)` and `attrset(A_NORMAL)`. Turns off all video highlighting attributes for the default (`standend`) or specified (`wstandend`) window.
`standout()` `wstandout(`*win*`)`	Equivalent to `attron(A_STANDOUT)`. Turns on the video highlighting attributes used for standout highlighting for the terminal being used. Does not alter other attributes in effect at the time. `standout` applies to the default window `stdscr`. `wstandout` affects the specified window.
`subwin(`*orig_win, num_lines,num_cols, beg_y,beg_x*`)`	Creates a new window containing the specified number of lines and columns within existing window `orig_win`. `beg_y` and `beg_x` specify the starting row and column position of the window on the physical screen (not relative to window `orig_win`). The subwindow uses that part of the main window character data storage structure that corresponds to its own area (each window maintains its own pointers, cursor location, and other items pertaining to window operation; only character storage is shared). Thus, the subwindow always contains character data (including highlighting attributes) that is identical to the data contained in the corresponding area of the original window, regardless of which window is the target of a write operation (highlighting bits are determined by the current attributes in effect for the window through which each character was stored). When using subwindows, it is often necessary to call `touchwin` before `refresh` in order to maintain correct display contents.

Description of Curses Routines (Continued)

`touchwin(`*win*`)`	Discards optimization information on the specified window so that the entire window must be completely rewritten during refresh. This is sometimes necessary when using overlapping windows because changes to one window do not update the overlapping window structure in such a manner that a subsequent refresh operation can be handled correctly.
`traceoff()`	Dummy entry point. Performs no useful function.
`traceon()`	Dummy entry point. Performs no useful function.
`typeahead(`*fd*`)`	Sets the file descriptor for typeahead check. `fd` is an integer obtained from `open` or `fileno`. Setting typeahead to −1 disables typeahead check. Default file descriptor is 0 (standard input). Typeahead is checked independently for each screen; for multiple interactive terminals, it should be set to the appropriate input for each screen. A call to `typeahead` always affects only the current screen.
`unctrl(`*ch*`)`	Converts the character code represented by `ch` into a printable form if it is an unprintable control character. The converted character is printed as an alpha-numeric character preceded by ^ where ^ represents the control key, and the alpha-numeric character corresponds to a key that can be pressed in conjunction with the control key to produce the control character.
`waddch(`*win,ch*`)`	See `addch(`*ch*`)`.
`waddstr(`*win,str*`)`	See `addstr(`*str*`)`.
`wattroff(`*win,attrs*`)`	See `attroff(`*attrs*`)`.
`wattron(`*win,attrs*`)`	See `attron(`*attrs*`)`.
`wattrset(`*win,attrs*`)`	See `attrset(`*attrs*`)`.
`wclear(`*win*`)`	See `clear()`.
`wclrtobot(`*win*`)`	See `clrtobot()`.
`wclrtoeol(`*win*`)`	See `clrtoeol()`.

Description of Curses Routines (Continued)

`wdelch(`*win*`)`	See `delch()`.
`wdeleteln(`*win*`)`	See `deleteln()`.
`werase(`*win*`)`	See `erase()`.
`wgetch(`*win*`)`	See `getch()`
`wgetstr(`*win,str*`)`	See `getstr(`*str*`)`
`winch(`*win*`)`	See `inch()`
`winsch(`*win,c*`)`	See `insch(`*c*`)`.
`winsertln(`*win*`)`	See `insertln()`.
`wmove(`*win,y,x*`)`	See `move(`*y,x*`)`.
`wnoutrefresh(`*win*`)`	See `doupdate()`.
`wprintw(`*win,fmt,args*`)`	See `printw(`*fmt,args*`)`.
`wrefresh(`*win*`)`	See `refresh()`. See also `doupdate()`.
`wscanw(`*win,fmt,args*`)`	See `scanw(`*fmt,args*`)`.
`wsetscrreg(`*win,t,b*`)`	See `setscrreg(`*t,b*`)`.
`wstandend(`*win*`)`	See `standend()`.
`wstandout(`*win*`)`	See `standout()`.

Terminfo Routines

	Description of Terminfo Routines
`delay_output(`*ms*`)`	Inserts a delay into the output stream for the specified number of milliseconds by inserting sufficient pad characters to effect the delay. This should not be used in place of a high-resolution `sleep`, but rather to slow down or hold off the output. Due to system buffering, it is unlikely that a delay can result in a process actually sleeping. `ms` should not exceed about 500 because of the large number of pad characters used to produce such delays.
`putp(`*str*`)`	Outputs a string capability without use of an `affcnt` (see `tputs`). The string is sent to `putchar` with an `affcnt` of 1. It is used in simple applications that do not require the output processing capability of `tputs`.
`setupterm(`*term, filenum,errret*`)`	Initializes the specified terminal. *term* is the character string representing the name or model of the terminal; *filenum* is the HP-UX file descriptor of the terminal being used for output; *errret* is a pointer to the integer in which a success/failure indication is returned. The values returned can be: 1 (initialize complete); −1 (`terminfo` database not found); or 0 (no such terminal). If 0 is given as the value of *term*, the default value of `TERM` is obtained from the enviroment. *errret* can be specified as 0 if no error code is wanted. If *errret* is default (0), and something goes wrong, `setupterm` prints an appropriate error message and exits rather than returning. Thus, a simple program can call `setupterm`(0,1,0) and not provide for initialization errors. If the environment variable `TERMINFO` is set to a path name, `setupterm` checks for a compiled `terminfo` description of the terminal under that path before checking */etc/term*. Otherwise, only */etc/term* is checked.

`setupterm` uses *filenum* to check the tty driver mode bits, and changes any that might prevent correct operation of low-level `curses` routines. Tabs are not expanded into spaces because various terminals exhibit inconsistent uses of the tab character. If the HP-UX system is expanding tabs, `setupterm` removes the definition of the [tab] and [backtab] functions because they may not be set correctly in the terminal. Other system-dependent changes such as disabling a virtual terminal driver may also be made here, if deemed appropriate by `setupterm`.

`setupterm` also initializes the global variable `ttytype` (an array of characters) to the value of the list of names for the terminal in question. The list is obtained from the begining of the `terminfo` description.

Upon completion of `setupterm`, the global variable *cur_term* points to the current structure of terminal capabilities. A program can use two or more terminals at once by calling `setupterm` for each terminal, and saving and restoring *cur_term*.

`nl()` is enabled, so newlines are converted to carriage return-line feed sequences on output. Programs that use *cursor_down* or *scroll_forward* should avoid these two capabilities or disable the mode with `nonl()`. `setupterm` calls `reset_prog_mode` after any changes are made.

`tparm(`*instring,p1,p2,p3, p4,p5,p6,p7,p8,p9*`)` Instantiates a parameterized string. Up to nine parameters can passed (in addition to the input string) that define what operations are to be performed on `instring` by `tparm`. The resultant string is suitable for output processing by `tput`.

Description of Terminfo Routines (Continued)

`tputs(`*cp,affcnt,outc*`)`	Processes `terminfo` capability strings for terminal devices (see *terminfo*(5) entry in *HP-UX Reference*). The padding specification, if present, is replaced by enough padding characters to produce the specified time delay. The resulting string is passed, one character at a time, to the routine `outc` which expects a single character parameter each time it is called. Often, `outc` simply calls `putchar` to complete its task. `cp` is the capability string, and *affcnt* is the number of units affected (such as lines or characters). For example, the *affcnt* for *insert_line* is the number of lines on the screen below the inserted line; that is, the number of lines that will have to be moved on the terminal. In certain cases, *affcnt* is used to determine the number of padding characters that must be created in the output string to produce the required delay(s), based on known terminal characteristics (obtained from the terminal identification data base).
`vidattr(`*attrs*`)`	Transmits the appropriate string to `stdout` to activate the specified video attributes which can include any or all of the following: `A_STANDOUT`, `A_UNDERLINE`, `A_REVERSE`, `A_BLINK`, `A_DIM`, `A_BOLD`, `A_BLANK` (invisible), `A_PROTECT`, and `A_ALTCHARSET` (multiple attributes must be separated by the C logical OR operator |).
`vidputs(`*attrs,putc*`)`	Transmits the appropriate string to the terminal, activating the specified video highlighting attributes. `attrs` can include any or all of the following (multiple attributes must be separated by the C logical OR operator |): `A_STANDOUT`, `A_UNDERLINE`, `A_REVERSE`, `A_BLINK`, `A_DIM`, `A_BOLD`, `A_BLANK` (invisible), `A_PROTECT`, and `A_ALTCHARSET`. `putc` is a `putchar`-like function. Previous highlighting attributes are preserved by this routine and restored upon return.

Termcap Compatibility Routines

Several routines have been included in `curses` that support programs written with calls to `termcap` routines. Calling parameters are the same as for equivalent `termcap` calls, but the routines are emulated using the `terminfo` data base. These routines may be removed in future releases of HP-UX.

Description of Termcap Compatibility Routines

`tgetent(`*bp,name*`)`	Obtains and returns with `termcap` entry for *name*.
`tgetflag(`*id*`)`	Returns the boolean entry for *id*.
`tgetnum(`*id*`)`	Returns the numeric entry for *id*.
`tgetstr(`*id,area*`)`	Returns the string entry for *id* and places the result in *area*.
`tgoto(`*cap,col,row*`)`	Attaches *col* and *row* parameters to the capability *cap*.
`tputs(`*cap,affcnt,fn*`)`	Equivalent to the `terminfo` routine `tputs`. Parameters are identical for both cases.

3

Program Operation

This section describes how curses routines behave and how they are used in a typical programming environment.

Insert/Delete Line

The output optimization routines associated with curses use terminal hardware insert/delete line capabilities provided the routine

```
idlok(stdscr,TRUE);
```

has been called to enable the capability. By default, insert/delete line during refresh is disabled (FALSE); not for performance reasons (there is no speed penalty involved), but because experience has shown that not only is insert/delete line frequently not needed (especially in simple programs); it can sometimes be visually annoying when used by curses. Insert/delete character is *always* available to curses if it is supported by the terminal.

Additional Terminals

Curses can be used, even when absolute cursor addressing is not provided on the terminal, as long as the cursor can be moved from any location to any other location. `curses` considers available cursor control options such as local motions, parameterized motions, home, and carriage return.

`curses` is intended for use with full-duplex, alphamumeric, video display terminals. No attempt is made to handle half-duplex, synchronous, hard copy, or bitmapped terminals. Bitmapped terminals can be handled by programming the bitmapped terminal to emulate an ordinary alphanumeric terminal. This prevents `curses` from using the bitmap capabilities, but `curses` was not designed for bitmapping.

`curses` can also deal with terminals that have the "magic cookie" glitch in their display highlighting behavior. The term "magic cookie" means that changes in highlighting are controlled by storing a "magic cookie" character in a location on the screen. While this "cookie" takes up a space, preventing an exact implementation of what the programmer wanted, `curses` takes the extra character space into account, and moves part of the line to the right when necessary. In some cases, this unavoidably results in losing text along the right-hand edge of the screen, but `curses` compensates where possible by omitting extra spaces.

Multiple Terminals

Some applications require that text be displayed on more than one terminal at the same time from the same process. This is easily accomplished, even when the terminals are different types.

`curses` maintains all information about the current terminal in a global variable called

```
struct screen *SP;
```

Although the screen structure is hidden from the user, the C compiler accepts declarations of variables that are pointers. The user program should declare

one screen pointer variable for each terminal that is to be handled. The routine:

```
struct screen *
newterm(type,fdout,fdin)
```

sets up a new terminal of the specified `type` and output is handled through file descriptor `fdout`. This is comparable to the usual program call to `initscr` which is essentially equivalent to

```
newterm(getenv("TERM"),stdout)
```

A program that uses multiple terminals should call `newterm` for each terminal, and save the value returned as a reference to that terminal for other calls.

To change to a different terminal, call

```
set_term(term)
```

which returns the old value of variable `SP`. Do not assign to `SP` because certain other global variables must also be changed.

All `curses` routines always interact with the current terminal. `set_term` is used to change from one terminal to the next in a multi-terminal environment. When the program is ready to terminate, each terminal should be selected in turn by a call to `set_term`, then cleaned up with screen clearing and cursor locating routines, followed by a call to `endwin()` for that terminal. Repeat the sequence for each additional terminal used by the program. The example program *TWO* demonstrates the technique.

Video Highlighting

Video highlighting attributes can be displayed in any combination on terminals that support the various attribute capabilities. Each character position in screen data structures is allotted 32 bits: eight for the character code; eight for the NLS attributes; the remaining sixteen for highlighting attributes, one bit per attribute. Each respective bit is associated with one of the following attributes: standout, underline, inverse video, blink, dim, bold, invisble, protect, and alternate character set. Standout selects the visually most pleasing highlighting method, and should be used by all programs that do

not need a specific highlighting combination. Underlining, inverse video, blinking, dim, and bold are standard features on most popular terminals, though they are not usually all present on a single terminal (for example, no current terminal implements both bold and dim). Invisible means that visible characters are displayed as blanks for security reasons (such as when echoing passwords). Protected and Alternate Character Set are subject to the characteristics of the terminal being used. Invisible, protected, and alternate character set attributes are subject to change or substitution by `curses`, and should be avoided unless necessary.

When characters are stored, each character is combined with the **current attributes** variable associated with the window. The variable is formed by using one of the following routines:

`attrset(`*attrs*`)`	`wattrset(`*win,attrs*`)`
`attron(`*attrs*`)`	`wattron(`*win,attrs*`)`
`attroff(`*attrs*`)`	`wattroff(`*win,attrs*`)`
`standout()`	`wstandout(`*win*`)`
`standend()`	`wstandend(`*win*`)`

The following attributes can be specified in the `attrs` argument for corresponding attribute set/on/off routines.

`A_STANDOUT`	`A_BLINK`	`A_INVIS`
`A_UNDERLINE`	`A_DIM`	`A_PROTECT`
`A_REVERSE`	`A_BOLD`	`A_ALTCHARSET`

When specifying multiple attributes, they should be separated by the C logical OR operator (|). Thus, to specify blinking underline and disable all other attributes on the `stdscr` window, use `attrset(A_BLINK|A_UNDERLINE)`.

`curses` forms the current attributes word as follows:

- Each attribute (such as `A_UNDERLINE`) is stored as a 32-bit word where all bits are zero except the bit that represents the corresponding attribute in a stored character word (for example, 00000100000000000000000000000000 controls blinking).

- All attributes forming the `attrs` argument are combined using the logical OR operator to create a single 32-bit word containing all attributes in the argument. For example, the three attribute words

  00000100000000000000000000000000,
  00010000000000000000000000000000, and
  00000010000000000000000000000000 are combined to form
  00010110000000000000000000000000 which identifies the new attributes.

- Three things can be done with the new attributes word. It can be used as the new current attributes `attrset` or `wattrset`); or the new attributes can be added to any currently active attributes (`attron` or `wattron`), or deleted from the currently active attributes (`attroff` or `wattroff`).
- If `attrset` (or `wattrset`) was called, the routine stores the new attributes in the current attributes variable and returns. The previous set of current attributes is destroyed.
- If `attron` (or `wattron`) was called, the routine performs a logical OR of the current attributes with the new attributes, then places the result in the current attributes variable and returns. The revised current attributes variable contains all previously active attributes plus the new attributes.
- If `attroff` (or `wattroff`) was called, the routine inverts the new attributes, performs a logical AND on the inverted new attributes and the current attributes, then places the result in the current attributes variable and returns. The altered current attributes variable contains all previously active attributes except those specified in the call, which are now disabled.
- `standout` and `wstandout` obtain their highlighting attributes from the `terminfo` data base, then perform the same operation as `attron` prior to returning.
- `standend` and `wstandend` disable all attributes then return. They are equivalent to `attrset(0)` and `attrset(A_NORMAL)`.

- `attrset(0)` and `wattrset(`*win,0*`)` set the 32-bit current attributes variable value to zero which disables all attributes. `A_NORMAL` can be substituted for zero as an argument.

The preceding scenarios assume that the specified attributes are available on the current terminal. In every case, the `terminfo` data base is used to determine whether the selected attribute is present. If it is not, `curses` attempts to find a suitable substitute before forming the new attribute set. If the terminal has no highlighting capabilities, all highlighting commands are ignored.

Three other constants (defined in `<curses.h>`), in addition to the previously listed attributes are also available for program use if needed:

- `A_NORMAL` has the octal value 000000000000, and can be used as an attribute argument for `attrset` to restore normal text display. `attrset(0)` is easier to type, but less descriptive. Both are equivalent.
- `A_ATTRIBUTES` has the octal value 037740000000. It can be logically ANDed with a character data word to isolate the attribute bits and discard the character.
- `A_CHARTEXT` has the octal value 000000000377. It can be logically ANDed with a character data word to isolate the character code and discard the attributes.

Special Keys

Most terminals have special keys, such as arrow keys, screen/line clearing keys, insert and delete line or character keys, and keys for user functions. The character sequences that such keys generate and send to the host computer vary from terminal to terminal. `curses` provides a convenient means for handling such keys through the use of `keypad` routines. Keypad capabilities are enabled by the call:

```
keypad(stdscr,TRUE)
```

during program initialization, or

```
keypad(win,TRUE)
```

when setting up and initializing other windows, as appropriate. When keypad is enabled, keypad character sequences are passed to the program by `getch`, but they are converted to special character values starting at 0401 octal (keypad character codes are listed in the keypad discussion early in this tutorial). Keypad codes are 16-bit values, and must not be stored in a *char* type variable because the upper bits must be preserved.

When keypad keys are used in a program, avoid using the escape key for program control because most keypad sequences begin with escape. If escape is used for program control, an ambiguity results that is not easily dealt with, and, at best, results in sluggish program response to all escape sequences as well as significant potential for incorrect program operation.

Scrolling Regions

Each window has a programmer-accessible scrolling region that is normally set to include the entire window. `curses` contains a routine that can be used to change the scrolling region to any location in the window by specifying the top and bottom margin lines. The routines are called by

```
setscrreg(top,bottom)
```

for the `stdscr` window, or

```
wsetscrreg(win,top,bottom)
```

for other windows. When the cursor advances beyond the bottom line in the region, all lines in the region are moved up one line (destroying the top line in the process) and a new line at the bottom of the region becomes the new cursor line. If scrolling has been enabled by a call to `scrollok` for that window, scrolling takes place, but only within the window boundary (if `scrollok` is not enabled, the cursor stays on the bottom line and no scrolling can occur). The scrolling region is a software feature only, and only causes a given window data structure to scroll. It may or may not translate to use of the hardware scrolling region that is featured on some terminals or hardware insert/delete line capabilities on the terminal.

Mini-Curses

All calls to `refresh` copy the current window to an internal screen image (`stdscr`). For simpler applications where window capabilities are not important and all operations can be handled by the standard screen, the screen output optimization capabilities of `curses` can be obtained through the low-level `curses` interface routines supported by `mini-curses`. Mini-`curses` is a subset of full `curses`, so any program that runs on the subset can also run on full `curses` without modification.

A complete list of commands is shown at the beginning of the `curses` commands section in this tutorial. Commands that are supported by `mini-curses` are marked with an asterisk (some that are not marked may also be accessible – if a program calls routines that are not, an error message showing undefined calls is produced by the compiler at compile time).

`mini-curses` routines are limited to commands that deal with the `stdscr` window. Certain other high-level functions that are convenient but not essential (such as `scanw`, `printw`, and `getch`) are not available, as well as all commands that begin with `w`. Low-level routines such as hardware insert/delete line and video attributes are supported, as are mode-setting routines such as `noecho`.

To access `mini-curses`, add `-DMINICURSES` to the CFLAGS in the makefile. If any routines are requested that are not available in `mini-curses`, an error diagnostic such as

```
Undefined:
m_getch
m_waddch
```

is listed to indicate that the program contains calls (in this case to `getch` and `waddch`) that cannot be linked because they are not available.

Remember that the preprocessor is involved in the implementation of `mini-curses`, so any programs that are compiled for use with `mini-curses` must be recompiled if they are to be used with full `curses`.

TTY Mode Functions

In addition to the save/restore functions `savetty()` and `resetty()`, other standard routines are provided by `curses` for entering and exiting normal tty mode.

- `resetterm()` restores the terminal to its state prior to `curses`' start-up.
- `fixterm` performs the equivalent of an `undo` on the previous `fixterm` on that terminal; it restores the "current curses mode" using the results of the most recent call to `saveterm()`.
- `endwin` automatically calls `resetterm`.
- Routines that handle control-Z (on systems that have process control) also use `resetterm()` and `fixterm()`.

Programs that use `curses` should use these routines before and after shell escapes, and also if the program has its own routines for dealing with control-Z. These routines are also available at the `terminfo` level.

Typeahead Check

When a user types something during a screen update, the update stops, pending a future update. This is useful when several keys are pressed in sequence, each of which produces a large amount of output. For example in a screen editor, the "forward screen" (or "next page") key draws the next screenful of text. If the key is pressed several times in rapid succession, rather than drawing several screens of text, `curses` cuts the updates short and only displays the last requested full screen. This feature is automatic, and cannot be disabled. It requires support by certain routines in the HP-UX operating system.

`getstr` Routine

No matter whether echo is enabled or disabled, strings typed and input by `getstr` are echoed at the current cursor location. Erase and kill characters assigned by the user for his (or her) terminal are considered when handling input strings. Thus it is unnecessary for interactive programs to deal directly

with erase, echo, and kill when processing a line of text from the terminal keyboard.

`longname`

The longname function does not require any arguments. It returns a pointer to a static storage area that contains the actual long (verbose) terminal name.

Nodelay Mode

The program call

```
nodelay(stdscr,TRUE)
```

puts the terminal in "no delay" mode. When nodelay is active, any call to getch returns the value -1 if there is nothing available for immediate input. This feature is helpful for real-time situations where a user is watching terminal screen outputs and presses a key when he wants to respond. For example, a program can be producing a text pattern on the screen while maintaining an open opportunity for the user to press certain keys to alter the output pattern, change cursor direction, or produce some other effect.

Example Programs: SCATTER

This program takes the first 23 lines from the standard input, then displays them in random order on the display terminal screen.

```
#include <curses.h>
#define    MAXLINES     120
#define    MAXCOLS      160
char    s[MAXLINES][MAXCOLS];    /* Screen Array */

main()
{
        register int  row = 0,
                      col = 0;
        register char c;
        int           char_count = 0; /* count non-blank characters */
        long          t;
        char          buf[BUFSIZ];

        initscr();
        for (row = 0; row < MAXLINES; row++)     /* initialize
screen array */
                for (col = 0; col < MAXCOLS;  col++)
                        s[row][col] = ' ';

        row = 0;
        col = 0;
        /* Read screen in */
        while ( (c = getchar()) != EOF && row < LINES) {
                if (c != '\n' && col < COLS) {
                        /* Place char in screen array */
                        s[row][col++] = c;
                        if (c != ' ')
                                char_count++;
                } else {
                        col = 0;
                        row++;
                }
        }

        time(&t);       /* Seed the random number generator */
        srand((int)(t&0177777L));

        while (char_count) {
```

```
                row = rand() % LINES;
                col = (rand() >> 2) % COLS;
                if (s[row][col] != ' ' && s[row][col] != EOF) {
                        move(row,col);
                        addch(s[row][col]);
                        s[row][col] = EOF;
                        char_count--;
                        refresh();
                }
        }

        endwin();
        exit(0);
}
```

Example Program: SHOW

This example program displays a file taken from the standard input, one screen at a time. Press the terminal space bar to advance to the next screen.

```
#include <curses.h>
#include <signal.h>
main(argc,argv)
        int     argc;
        char    *argv[];
{
        FILE    *fd;
        char    linebuf[BUFSIZ];
        int     line;
        void    done(),perror(),exit();

        if (argc != 2) {
                fprintf(stderr, "usage: %s file\n", argv[0]);
                exit(1);
        }

        if ( (fd = fopen(argv[1], "r")) == NULL) {
                perror(argv[1]);
                exit(2);
        }

        signal(SIGINT, done);
        initscr();
        noecho();
        cbreak();
        nonl();                                 /* enable more screen optimization
*/
        idlok(stdscr,TRUE);             /* allow insert/delete line */

        while (1) {
                move(0,0);
                for (line = 0; line < LINES; line++) {
                        if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
                                clrtobot();
                                done();
                        }
                        move(line,0);
                        printw("%s", linebuf);
                }
```

```
                refresh();
                if (getch() == 'q')
                        done();
        }
}
        void
done()
{
        move(LINES-1, 0);
        clrtoeol();
        refresh();
        endwin();
        exit(0);
}
```

Example Program: HIGHLIGHT

This example program displays text taken from the standard input. Highlighting is determined by embedded character sequences in the file. \U starts underlining, \B starts bold highlighting, and \N restores normal display characteristics.

```
#include <curses.h>

main(argc,argv)
        char    **argv;
{
        FILE    *fd;
        int     c,c2;

        if (argc != 2) {
                fprintf(stderr, "Usage: highlight file\n");
                exit(1);
        }

        fd = fopen(argv[1],"r");
        if (fd == NULL) {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        scrollok(stdscr,TRUE);

        for (;;) {
                c = getc(fd);
                if (c == EOF)
                        break;

                if (c == '\\') {
                        c2 = getc(fd);
                        switch(c2) {
                        case 'B':
                                attrset(A_BOLD);
                                continue;
                        case 'U':
                                attrset(A_UNDERLINE);
                                continue;
                        case 'N':
```

```
                                attrset(0);
                                continue;
                        }

                        addch(c);
                        addch(c2);
                } else
                        addch(c);
        }

        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

Example Program: WINDOW

This program demonstrates the use of multiple windows.

```
#include <curses.h>

WINDOW    *cmdwin;

main()
{
        int     i,c;
        char    buf[120];

        initscr();
        nonl();
        noecho();
        cbreak();

        cmdwin = newwin(3,COLS,0,0);    /* top 3 lines */
        for (i=0; i < LINES; i++)
                mvprintw(i,0,"This is line %d of stdscr",i);

        for (;;) {
                refresh();
                c = getch();
                switch(c) {
                case 'c':    /* Enter command from keyboard */
                        werase(cmdwin);         /* clear window */
                        wprintw(cmdwin,"Enter command:");
                        wmove(cmdwin,2,0);
                        for (i=0; i < COLS; i++)
                                waddch(cmdwin,'-');

                        wmove(cmdwin,1,0);
                        touchwin(cmdwin);
                        wrefresh(cmdwin);
                        wgetstr(cmdwin,buf);
                        touchwin(stdscr);

                        /*
                         * The command is now in buf.
                         * It should be processed here.
                         */
                        erase();
```

```
                        for (i=0; i < LINES; i++)
                                mvprintw(i,0,"%s",buf);
                        refresh();
                        break;
                case 'q':
                        endwin();
                        exit(0);
                }
        }
}
```

Example Program: TWO

This program shows how to handle two terminals from a single program.

```
#include <curses.h>
#include <signal.h>

struct screen *me, *you;
struct screen *set_term();

FILE    *fd, *fdyou;
char    linebuf[512];

main(argc,argv)
        char    **argv;
{
        int     done();
        int     c;

        if (argc != 4) {
                fprintf(stderr,"Usage: two othertty otherttytype inputfile\n");
                exit(1);
        }

        fd = fopen(argv[3],"r");
        fdyou = fopen(argv[1],"w+");
        signal(SIGINT, done);    /* die gracefully */

        me = newterm(getenv("TERM"),stdout,stdin);    /* initialize my tty */
        you = newterm(argv[2],fdyou,fdyou);    /* Initialize his/her terminal*/

        set_term(me);           /* Set modes for my terminal */
        noecho();               /* turn off tty echo */
        cbreak();               /* enter cbreak mode */
        nonl();                 /* Allow linefeed */
        nodelay(stdscr,TRUE);   /* No hang on input */

        set_term(you);
        noecho();
        cbreak();
        nonl();
        nodelay(stdscr,TRUE);

        /* Dump first screen full on my terminal */
```

```
        dump_page(me);

        /* Dump second screen full on his/her terminal */
        dump_page(you);

        for (;;) {    /* for each screen full */
                set_term(me);
                c = getch();
                if (c == 'q')    /* wait for user to read it */
                        done();
                if (c == ' ')
                        dump_page(me);

                set_term(you);
                c = getch();
                if (c == 'q')    /* wait for user to read it */
                        done();
                if (c == ' ')
                        dump_page(you);
                sleep(1);
        }
}

dump_page(term)
        struct screen *term;
{
        int    line;

        set_term(term);
        move(0,0);
        for (line=0; line < LINES-1; line++) {
                if (fgets(linebuf,sizeof linebuf,fd) == NULL) {
                        clrtobot();
                        done();
                }
                mvprintw(line,0,"%s",linebuf);
        }

        standout();
        mvprintw(LINES-1,0,"--More--");
        standend();
        refresh();    /* sync screen */
}

/*
```

```
 * Clean up and exit.
 */
done()
{
        /* Clean up first terminal */
        set_term(you);
        move(LINES-1,0);        /* to lower left corner */
        clrtoeol();             /* clear bottom line */
        refresh();              /* flush out everything */
        endwin();               /* curses clean up */

        /* Clean up second terminal */
        set_term(me);
        move(LINES-1,0);        /* to lower left corner */
        clrtoeol();             /* clear bottom line */
        refresh();              /* flush out everything */
        endwin();               /* curses clean up */

        exit(0);
}
```

Example Program: TERMHL

This program is equivalent to the earlier example program HIGHLIGHT, but uses terminfo routines instead.

```
#include <curses.h>
#include <term.h>

int     ulmode = 0;     /* Currently underlining */

main(argc, argv)
        char    **argv;
{
        FILE    *fd;
        int     c,c2;
        int     outch();

        if (argc > 2) {
                fprintf(stderr, "Usage: termhl [file]\n");
                exit(1);
        }

        if (argc == 2) {
                fd = fopen(argv[1],"r");
                if (fd == NULL) {
                        perror(argv[1]);
                        exit(2);
                }
        } else {
                fd = stdin;
        }

        setupterm(0,1,0);
        for (;;) {
                c = getc(fd);
                if (c == EOF)
                        break;

                if (c == '\\') {
                        c2 = getc(fd);
                        switch(c2) {
                        case 'B':
                                tputs(enter_bold_mode,1,outch);
                                continue;
```

```
                        case 'U':
                                tputs(enter_underline_mode,1,outch);
                                ulmode = 1;
                                continue;
                        case 'N':
                                tputs(exit_attribute_mode,1,outch);
                                ulmode = 0;
                                continue;
                        }
                        putch(c);
                        putch(c2);
                } else
                        putch(c);
        }

        fclose(fd);
        fflush(stdout);
        resetterm();
        exit(0);
}

/*
 * This function is like putchar, but it checks for underlining.
 */
putch(c)
        int    c;
{
        outch(c);
        if (ulmode && underline_char) {
                outch('\b');
                tputs(underline_char,1,outch);
        }
 }

 /*
  * Outchar is a function version of putchar that can be passed to
  * tputs as a routine to call.
  */
 outch(c)
         int    c;
 {
         putchar(c);
}
```

Example Program: EDITOR

This program is a very simple screen-oriented editor that is similar to a small subset of `vi`. For simplicity, the `stdscr` window is also used as the editing buffer.

```
#include <curses.h>
#define CTRL(c) ('c'&037)
main(argc,argv)
        char    **argv;
{
        int     i,n,l;
        int     c;
        FILE    *fd;

        if (argc != 2) {
                fprintf(stderr,"Usage: edit file\n");
                exit(1);
        }

        fd = fopen(argv[1],"r");
        if (fd == NULL) {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        cbreak();
        nonl();
        noecho();
        idlok(stdscr, TRUE);
        keypad(stdscr, TRUE);

        /* Read in the file */
        while ((c = getc(fd)) != EOF)
                addch(c);
        fclose(fd);

        move(0,0);
        refresh();
        edit();

        /* Write out the file */
        fd = fopen(argv[1],"w");
```

```
        for (l=0; l < LINES; l++) {
                n = len(l);
                for (i=0; i<n; i++)
                        putc(mvinch(l,i),fd);
                putc('\n',fd);
        }
        fclose(fd);
        endwin();
        exit(0);
}
len(lineno)
        int    lineno;
{
        int    linelen = COLS-1;

        while (linelen >= 0 && mvinch(lineno,linelen) == ' ')
                linelen--;
        return linelen + 1;
}

/* Global value of current cursor position */
int    row,col;

edit()
{
        int    c;
        for (;;) {
                move(row,col);
                refresh();
                c = getch();
                switch(c) {    /* Editor commands */

                /* hjkl and arrow keys: move cursor */
                /* in direction indicated */
                case    'h':
                case    KEY_LEFT:
                        if (col > 0)
                                col--;
                        break;

                case    'j':
                case    KEY_DOWN:
                        if (row < LINES-1)
                                row++;
                        break;
```

```
case    'k':
case    KEY_UP:
        if (row > 0)
                row--;
        break;

case    'l':
case    KEY_RIGHT:
        if (col < COLS-1)
                col++;
        break;

/* i: enter input mode */
case    KEY_IC:
case    'i':
        input();
        break;

/* x: delete current character */
case    KEY_DC:
case    'x':
        delch();
        break;

/* o: open up a new line and enter input mode */
case    KEY_IL:
case    'o':
        move(++row,col=0);
        insertln();
        input();
        break;

/* d: delete current line */
case    KEY_DL:
case    'd':
        deleteln();
        break;

/* ^L: redraw screen */
case    KEY_CLEAR:
case    CTRL(L):
        clearok(curscr);
        refresh();
        break;
```

```
                /* w: write and quit */
                case    'w':
                        return;

                /* q: quit without writing */
                case    'q':
                        endwin();
                        exit(1);
                default:
                        flash();
                        break;
                }
        }
}
/*
 * Insert mode: accept characters and insert them.
 * End with ^D or EIC.
 */
input()
{
        int     c;
        standout();
        mvaddstr(LINES-1, COLS-20, "INPUT MODE");
        standend();
        move(row,col);
        refresh();

        for (;;) {
                c = getch();
                if (c == CTRL(D) || c == KEY_EIC)
                        break;
                insch(c);
                move(row, ++col);
                refresh();
        }
        move(LINES-1, COLS-20);
        clrtoeol();
        move(row,col);
        refresh();
}
```

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

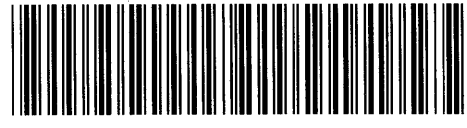
U

V

W

Contents

HP Part Number
97089-90057
Microfiche No. 97089-99057
Printed in U.S.A. E0989



97089-90660
For Internal Use Only