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

HP Part Number 97089-90052

HEWLETT PACKARD

Hewlett-Packard Company

3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

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

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

September 1986...Edition 1

December 1986...Update 1. Device I/O Library tutorial rewritten and expanded to clarify hardware/software operation and to document new subroutines added at Series 300 HP-UX Release 5.2. Appendix F added to include a non-trivial applications programming example.

June 1987...Edition 2. Update incorporated. Native Language Support tutorial added for Series 300 and Series 800 HP-UX. UUCP tutorial rewritten to include new UUCP procedures, remove duplicate material and reorganize chapters.

October 1987...Edition 3. This new edition incorporates new features, more information and more examples.

Table of 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, whether Integral PC, Series 200/300, 500, or 800. However, because of certain inherent differences between processors and other hardware, some differences do exist. When such differences arise during an explanation, they are clearly identified by introductory headings such as:

- **Series 500 Only:**
- **Integral PC Only:**
- **Series 200/300 Only:**

Additional major differences related to a specific model or series are identified in a separate appendix for that model or series. Appendices are provided for Series 200/300, 500, 800, and the Integral PC.

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 500 Dependencies discusses hardware- and system-dependent characteristics of DIL subroutines when used with Series 500 computers. If you are using a Series 500 HP-UX system, check this appendix to ensure correct use of DIL subroutines.

Appendix B: Series 200/300 Dependencies is similar to Appendix A, but for Series 200/300 computers. Use this appendix to ensure the correct use of DIL subroutines on Series 200/300 systems.

Appendix C: Integral PC Dependencies describes hardware- and system-dependent characteristics related to the Integral PC. refer to this appendix to ensure the proper usage of DIL routines on the Integral PC.

Appendix D: 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 E: Character Codes

Appendix F: 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 -ldvio program.c
```

Similarly, for a Pascal program, use:

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

and for a FORTRAN program, use:

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

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:

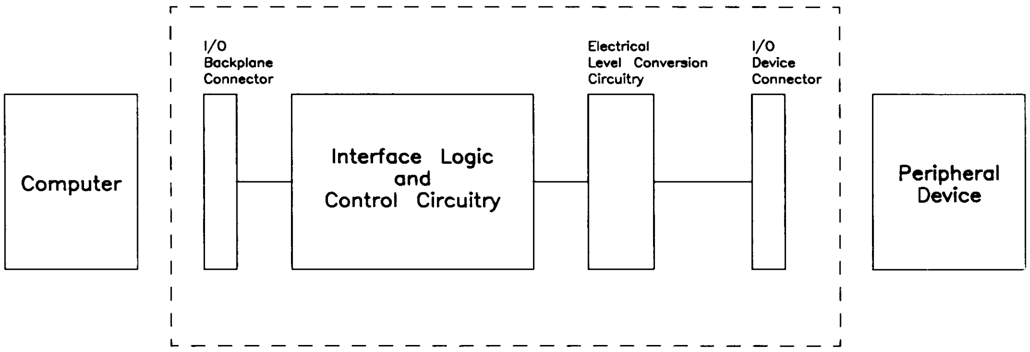


Figure 1–1. Interface Functional Diagram

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.

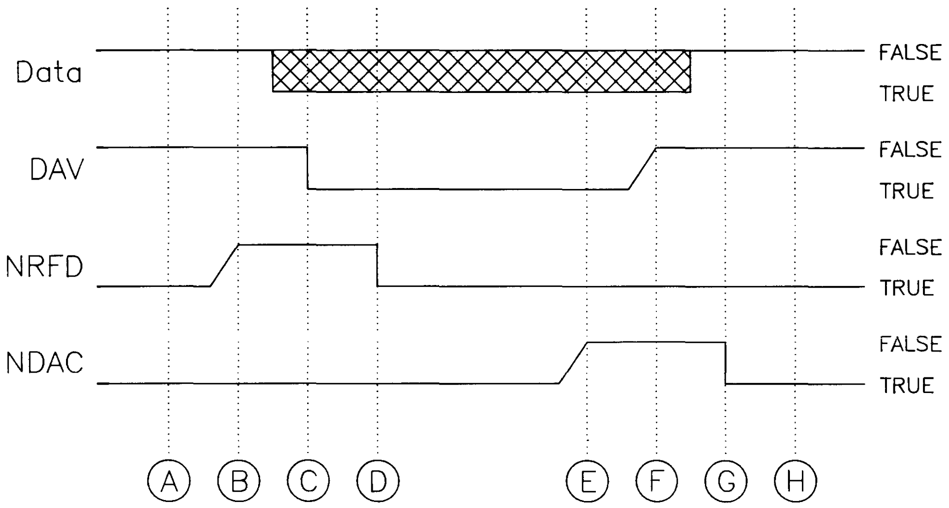


Figure 1–2. The HP-IB Handshake

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 200/300, 500, 800, or the Integral PC for information pertaining to your specific application.

General-Purpose Routines 2

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 200/300 and 500). 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 containing DIL subroutine calls for a specific interface 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.

Series 200/300 computers support an additional subroutine: *io_burst*. Refer to the *io_burst*(3I) page in the *HP-UX Reference* for details on using this subroutine.

The **Integral PC DIL library** supports several non-standard DIL subroutines in addition to the standard subroutines in Table 2-1. Refer to the "Integral PC Dependencies" appendix for details on their use.

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.

Integral PC Only: filename is the special device name for the specific GPIO or HP-IB interface created by *load_gpio* or *load_hpib*. Note that each GPIO port has a separate device file name. Refer to Appendix C, "Integral PC Dependencies," for details on using *load_gpio* and *load_hpib* to create special files for GPIO and HP-IB interfaces.

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>
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";
eid = open(buffer, O_RDWR);
```

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 NDIL-BUFFERS (default is 30). On Series 200 systems, the maximum number of open DIL files is limited to 10.

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>
main()
{
   int eid;
   :
   eid = open( "/dev/raw_hpib", O_RDWR);

   :      /* 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* 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>
main()
{
    int  eid;         /*the entity identifier*/
    char buffer[10];  /*buffer in which the read data will be placed*/
    eid = open("/dev/raw_hpib", O_RDWR);

    :  /*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>
main()
{
   int  eid;          /*the entity identifier*/
   char *buffer;      /* the buffer containing data to be written to a file*/
   eid = open("/dev/raw_hpib", O_RDWR);

   .
   :  /*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 parameters (discussed later) associated with the interface file *eid* can end the transfer before this number is reached.

Integral PC Only: When performing a *read* or *write* operation to a 16- or 32-bit GPIO port, the data must start on a word boundary.

Example

Assume that you have already created an auto-addressed special file, */dev/hpib_dev*, for an HP-IB device. Your program must first open the interface file */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) page 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

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 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");
      else
         printf("Error: file exists but cannot open");
      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*(3C) (see 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);
```

Io_reset resets the interface whose entity identifier is *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>
main()
{
   int eid;
   :
   eid = open( "/dev/raw_hpib", O_RDWR);
   if (io_reset(eid) == -1)
   {
       printf("Possible problem with interface");
       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*(2) in the *HP-UX Reference* for more information about locking and unlocking interfaces.

CAUTION

Do not place a lock on any interface that supports the 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 *eids* 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 *eids*.

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

Opening multiple *eids* 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>
main()
{
   int  eid;
   long time;

   eid = open( "/dev/raw_hpib", O_RDWR);
   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 an interface 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 interface 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 interface 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>
main()
{
    int  eid, width;
    width = 16;                /*width of new data path */

    eid = open("/dev/raw_gpio", O_RDWR);
    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 all users of the interface. Once the data path width is altered, it remains at the new value for each future opening of the file, independent of *eid*. Use *io_reset* or *io_width_ctl* to restore the default 8-bit path width. It should be obvious from this discussion that if any program on the system alters the data path width for a given interface from its default value, all programs using the interface should include a call to *io_width_ctl* to ensure correct operation. However, if a given interface requires operation at a fixed but not default path width, and is used identically by all calling programs (such as a 16-bit GPIO card connected to a single peripheral device), the call to *io_width_ctl* could be easily included in a system start-up configuration program that is executed automatically each time the system is rebooted or restarted for any reason.

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 interface file, the interface 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 that 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>
#define  ENABLE      1
main()
{
   int eid;

   eid = open("/dev/raw_hpib", O_RDWR);
   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`) is detected in incoming data. The `\n` becomes the last byte in the stored data message.

Integral PC Only: On the Integral PC, a read operation from a GPIO interface terminates only when a specified number of read operations have been performed or when the read termination pattern has been found (EOI is not recognized on the GPIO interface).

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.

Series 200, 300, and 500 Only: 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 effect 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>
   main()
   {
      int eid;
      char buffer[12];

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

       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;
        }
   }
```

Series 500 Only: On Series 500 computers, the value returned by *io_get_term_reason* only indicates the termination cause with the highest value; other causes with lower values could have occurred at the same time. See Appendix A, "Series 500 Dependencies" for more information.

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.

Currently, **interrupts are supported only on the Integral PC, Series 300, and Series 500 computers**. However, interrupts can be simulated on Series 200 systems. Refer to the appropriate hardware-specific appendix for any restrictions that may apply.

Integral PC Interrupt Support

The only interrupt condition available on the Integral PC is PIR: interrupt on assertion of the Peripheral Interrupt Request line. For hardware restrictions related to using the HP-IB interrupts on the Integral PC, refer to the *io_on_interrupt.3d* (or *.3i* if the *.3d* suffix is not present) file in the *doc* folder on the DIL disc.

Series 300 and 500 Interrupt Support

HP-IB Interrupts

Series 300 and 500 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.

Series 300 GPIO

Series 300 computers recognize the following GPIO interrupt conditions:

EIR	EIR line has been asserted.

Series 500 GPIO

Series 500 computers recognize the following GPIO interrupt conditions:

SIE0	Status line 0 has been asserted.
SIE1	Status line 1 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 invoked. 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>
main()
{
   int eid;
   struct interrupt_struct cause_vec;

   eid = open ("/dev/raw_hpib", O_RDWR);
   cause_vec.cause = SRQ;
   io_on_interrupt(eid, cause_vec, handler);
   :
   :
}
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

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);

 }
```

Controlling the HP-IB Interface 3

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
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 then 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** all current talkers on the bus. No means is available for unaddressing a given talker without unaddressing all talkers on the bus. This command ensures that the bus is cleared of all talkers before addressing a new talker.

DEVICE CLEAR

DEVICE CLEAR causes all devices that recognize this command to return to a predefined, 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 14 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 (with some loss of execution speed).
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).

Additional Series 200/300 and Integral PC Routines

The Integral PC and Series 200/300 support high-speed burst I/O on HP-IB and GPIO through the following DIL subroutine:

Subroutine	Description
io_burst(eid,flag)	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.

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	●

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 200/300 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. See the Series 200/300 Appendix for more information about burst I/O.

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 locks the interface, handles the transfer as defined in the data structure (which configures the HP-IB and handles the transfer and clean-up) unlocks the interface, then returns with the result (transfer complete or transfer failed). While *hpib_io* often makes programs shorter and simpler, the added overhead associated with *hpib_io* is less efficient than when using individual DIL subroutine calls.

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>
main()
{
   int eid;
   char command[2];          /*command array*/

   eid = open("/dev/raw_hpib", O_RDWR);
   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>
main()
{
   int eid;

   eid = open("/dev/raw_hpib", O_RDWR);
   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.

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

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,4);
```

where *eid* is the entity identifier for the opened HP-IB interface device file and *4* 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 <fcntl.h>
main()
{
   int eid, status;
   eid = open("/dev/raw_hpib", O_RDWR);

   if ((status = hpib_bus_status(eid,4)) == -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 200/300 and 500 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.

The Integral PC does not support auto-addressing. All HP-IB interface files on the Integral PC are raw special files and do not support auto-addressing. Hence *hpib_send_cmnd* must be used for all HP-IB bus control operations.

Auto-Addressing on Series 200/300 and 500

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 non-raw 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*,

Series 200, 300, and 500 systems determine whether a device file is raw or auto-addressed by the address parameter 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 500 computer with an HP 27110A/B HP-IB card on select code 01 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 (a similar procedure described in Appendix B is used for Series 200/300):

```
mknod /dev/device c 12 0x010300
```

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     12 0x010300 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:

```
main()
{
   int eid;
   eid = open("/dev/device",O_RDWR);
     /*Assuming "/dev/device" has the minor number (0x010300), the*/
     /*system automatically addresses the interface card at select code 1*/
     /*as a talker and the device at bus address 3 as a listener before*/
     /*sending data*/
   write(eid, "test data",9);
}
```

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 <fcntl.h>
main()
{
   int eid, address;
   eid = open("/dev/raw_hpib", O_RDWR);
   address = hpib_bus_status(eid, 7);
   .
   .
   .
}
```

where *eid* is the entity identifier for the interface file and *7* 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, 7) + 64;
MLA = hpib_bus_status(eid, 7) + 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 <fcntl.h>
main()
{
     int  eid, MTA;
     char command[5];
     eid = open("/dev/raw_hpib", O_RDWR);
     MTA = hpib_bus_status(eid, 7) + 64; /*calculate My Talk Address*/
     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 <fcntl.h>
main()
{
   int eid, MTA;
   char command[50];

   eid = open("/dev/raw_hpib", O_RDWR);
   MTA = hpib_bus_status(eid, 7) + 64;  /*calculate 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 <fcntl.h>
main()
{
   int eid, MLA, len;
   char buffer[51];                      /*storage for data*/
   char command[4];

   eid = open("/dev/raw_hpib", O_RDWR);
   MLA = hpib_bus_status(eid, 7) + 32;  /*calculate 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.

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, 1);
```

where *eid* is the entity identifier for the open interface file and *1* 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 <fcntl.h>
main()
{
   int eid;
   eid = open("/dev/raw_hpib", O_RDWR);
   io_timeout_ctl(eid,10000000);    /*Set a 10-second timeout*/
   if (hpib_status_wait(eid, 1) == 0)
      service_routine();            /*SRQ is asserted; service the request*/
   else
      printf("Either a timeout or an error occurred");
}
```

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

```
hpib_bus_status(eid, 1);
```

where, as before, *eid* is the entity identifier for the open interface file and *1* 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 pools 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.

Integral PC Only: The parallel poll response in the HP 82998A HP-IB interface can only be set using the *hpib_card_ppoll_resp* subroutine.

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 a 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.

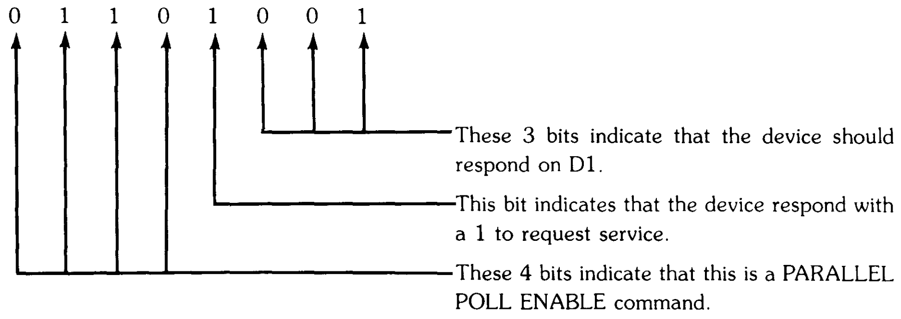
For example, to program a given device to respond to a parallel poll by placing a logic 1 on data line D0 if it needs service, use a PARALLEL POLL ENABLE command with a decimal value of 104 (binary 01101000).

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 <fcntl.h>
main()
{
   int eid, MTA;
   char command[50];

   eid = open("/dev/raw_hpib", O_RDWR);
   MTA = hpib_bus_status(eid, 7) + 64;  /*calculate 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 used above is:


0 1 1 0 1 0 0 1
These 3 bits indicate that the device should respond on D1.
This bit indicates that the device respond with a 1 to request service.
These 4 bits indicate that this is a PARALLEL POLL ENABLE command.

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>
main()
{
  int eid, status, byte;
  eid = open("/dev/raw_hpib", O_RDWR);

  if ((status = hpib_ppoll( eid)) == -1) /*conduct the parallel poll*/
  {
      printf("error taking ppoll");  /*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.

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 or a timeout occurred.

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 valuc 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>
main()
{
   int eid;
   eid = open("/dev/raw_hpib", O_RDWR);
   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");
   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>
main()
{
   int eid, status;
   eid = open("/dev/raw_hpib", O_RDWR);
   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");
        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].

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 device polled did not respond before the timeout or the interface is not Active Controller.
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].

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.

System Controller Role

When the HP-IBs System Controller is first powered on or is reset, it assumes the role of Active Controller. An HP-IB 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.

Integral PC Only: The HP 82998A HP-IB interface can be configured to power-on in the non-system-controller state by setting a switch on the interface card. Refer to the *HP 82923A HP-IB Interface Owner's Manual* for instructions. The built-in HP-IB interface on the Integral PC will always power-on in the system-controller state.

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, 3);
```

where *eid* is the entity identifier for an open interface file and *3* 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 <fcntl.h>
main()
{
   int eid, status;
   eid = open("/dev/raw_hpib", O_RDWR);

   if ((status = hpib_bus_status(eid, 3)) == -1)
       printf("Error occurred during bus status subroutine");
   else if (status == 1)
       printf("Interface is the System Controller");
   else
       printf("Interface is not the System Controller");
}
```

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 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.

The Computer As a Non-Active Controller

The information described in this section is accurate for Series 200/300 and 500 computers. For details specific to the Integral PC, refer to Appendix C, "Integral PC Dependencies."

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 controller 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
0	Is the interface in its remote state?
1	Are any devices currently requesting service? (Is SRQ asserted?)
2	Is there a listener that is not ready for data? (Is NDAC asserted?)
3	Is the interface the current System Controller?
4	Is the interface the current Active Controller?
5	Is the interface currently addressed as a talker?
6	Is the interface currently addressed as a listener?
7	What is the interface's bus address?

If the value of *status_question* is in the range 0-6, *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 7, *hpib_bus_status* returns the bus address of the computer's HP-IB interface. If the value of *status_question* is outside the allowable range of 0 through 7, **−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, 4)) == -1)
   printf("Error occurred while checking status");
else if (status == 0)
     printf("Computer is a Non-Active Controller");
else
     printf("Computer is the Active Controller");
:
```

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 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>
main()
{
   int eid;

   eid = open("/dev/raw_hpib", O_RDWR);
   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.

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.

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 how your computer responds 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 for your system to find out if 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.
- **Series 500 Only:** Interface parallel poll response cannot be altered under local program control.

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.
EINVAL	(**Series 500 Only:**) Interface cannot respond on the line indicated 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	Wait Condition
1	Wait until the SRQ line is asserted
4	Wait until this computer is the Active Controller
5	Wait until this computer is addressed as a talker
6	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 <fcntl.h>
main()
{
   int eid;

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

   if (hpib_status_wait(eid, 4) == -1) /*wait until Active Controller*/
   {
      printf("Error while waiting for status");
      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.

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.
EIO	The specified condition did not become true before a timeout occurred.

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 <fcntl.h>
main()
{
   int eid;

   eid = open("/dev/raw_hpib", O_RDWR);
   if (hpib_bus_status(eid, 5) == 1)
   {
      printf("the interface is addressed as a talker");
      write(eid, "data message", 12);   /*do the expected data transfer*/
   }
   else
      printf("the interface is not addressed as a talker");
}
```

In the above call to *hpib_bus_status*, *eid* is the entity identifier for the interface device file and *5* 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, 6) == 1)
{
   printf("the interface is addressed as a listener");
   read(eid, buffer, 12);            /*do the data transfer*/
}
else
   printf("the interface is not addressed as a listener");
:
```

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 5 causes the program to wait until the interface is addressed as a talker. With a condition value of 6 the routine waits until it is addressed to listen. How 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 occuered 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 <fcntl.h>
main()
{
   int eid, len;
   char buffer[51];                    /*storage for message*/
   eid = open("/dev/raw_hpib", O_RDWR);
   io_timeout_ctl(eid, 5000000);       /*5-second timeout*/

   if (hpib_status_wait(eid, 6) == -1)
   {
       printf("Either a timeout or an error occurred");
       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>
int eid;
char buffer[50];
main()
{
   int handler();
   int eid;
   struct interrupt_struct cause_vec;

   eid = open("/dev/raw_hpib",O_RDWR);
   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, data, 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*.

A call to *hpib_io* has the form:

```
#include <dvio.h>
/* on the Integral PC, the include directive would be:
 *
 *       #include <libdvio.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*.

On the Integral PC, the include file is *libdvio.h* instead of *dvio.h*, as shown in the example above.

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:

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*:

```
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 4 structures can be built and implemented.

```
   #include <fcntl.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 and bus address to
listen*/
      temp->buf = buffer[0];
      temp->count = strlen(temp->buf);
   /*Build structure 2 -- Sending the data message*/
      temp++;     /*use temp pointer so that iovec remains pointing to the*/
                  /*first structure but temp now points to the 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(temp->buf);


   /*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(temp->buf);


   /*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*/
      eid = open("/dev/raw_hpib", O_RDWR);

      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 is changed 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 10 *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 <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*/
   :
   eid = open("/dev/raw_hpib", O_RDWR); /*open the interface file*/

   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");
   }
   else
       printf("No error occurred during execution of hpib_io");
}
```

Controlling the GPIO Interface 4

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.

Configuring the GPIO Interface

On Series 200/300 and 500 computers, the GPIO interface is configured by setting several switches on the interface card. On the other hand, the HP 82923A GPIO interface used on the Integral PC is configured by using DIL routines instead of switches.

Configuring the Integral PC GPIO

As mentioned, DIL subroutines are used to configure the the HP 82923A GPIO interface on the Integral PC. The functions that can be configured are:

- Data logic sense (use *gpio_normalize* subroutine),
- Data handshake mode (use *gpio_handshake_ctl* subroutine),
- Delay time (use *gpio_delay_time_ctl* subroutine).

For information about these routines, refer to the documentation files in the *doc* folder on the DIL disc.

Setting Interface Switches

Series 200/300 and 500 computer GPIO interface cards have several configuration switches that are used to set up the interface. 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.

Note

On Series 200/300 systems, the GPIO interface select code is determined by a switch setting on the interface card. Refer to the appropriate hardware-specific appendix to see if a switch configuration is required. On Series 500 systems, no switch setting is required; the select code is determined by which I/O slot you use when installing the interface card.

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.
- **Series 500 Only**: You cannot recognize interrupts sent by the peripheral over the External Interrupt Request line (EIR).

Integral PC Only: The HP 82923A GPIO card has several capabilities not supported by the DIL routines. Because of this, the following limitations exist:

- 24-bit port paths are not supported,
- Flag line cannot be read directly,
- Fast-handshake transfer mode described in the *HP 82923A GPIO Interface Owner's Manual* is not supported.

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.

In addition to these standard GPIO DIL routines, the **Integral PC supports non-standard routines for controlling the HP 82923A GPIO interface**. Refer to the appendix "Integral PC Dependencies" for information about these routines.

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 are disabled on Series 200/300.

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.

Integral PC Only: The *io_reset* routine has the following effect on the HP 82923A GPIO interface:

- Read termination character is cleared,
- Timeout value is set to 0,
- Width for all ports is set to 8 bits,
- Normalization is set to positive true,
- Delay time is set to 1 μ-sec,
- Handshake mode is set to 1,
- Data lines are set to 0,
- Speed is set to the flag transfer mode,
- I/$\overline{\text{O}}$ line remains unchanged.

Performing Data Transfers

DIL subroutines *read* and *write* 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.

```
main()
{
   int  eid;                                 /*entity identifier*/
   char read_buffer[16], write_buffer[16];  /*buffers to hold data*/

   eid  =  open( "/dev/raw_gpio", O_RDWR);  /*open interface file*/
   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 ($\overline{\mathrm{CTL0}}$ and $\overline{\mathrm{CTL1}}$), and two are for input ($\overline{\mathrm{STI0}}$ and $\overline{\mathrm{STI1}}$). The routine *gpio_set_ctl* allows you to control the values of $\overline{\mathrm{CTL0}}$ and $\overline{\mathrm{CTL1}}$, while the routine *gpio_get_status* allows you to read the values of $\overline{\mathrm{STI0}}$ and $\overline{\mathrm{STI1}}$.

The Integral PC's HP 82923A GPIO interface does not provide any equivalent special-purpose lines. Each port, however, does have a single status line and a single control line. The status and control lines in unused ports can be used with active ports to perform the same function as the special-purpose lines. For example, if you have specified a port **b** data width of 16 bits, both ports **a** and **b** will be active. The status and control lines on ports **c** and **d** can then be used by first opening either port **c** or **d** then using the *gpio_get_status* and *gpio_set_ctl* routines to monitor or control those lines.

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

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 $\overline{\mathrm{CTL0}}$ (bit 0) and $\overline{\mathrm{CTL1}}$ (bit 1). Both $\overline{\mathrm{CTL0}}$ and $\overline{\mathrm{CTL1}}$ 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: $\overline{\mathrm{CTL0}}$ and $\overline{\mathrm{CTL1}}$ both false (HIGH logic level)
- If *value* =1: $\overline{\mathrm{CTL0}}$ true (LOW logic level) and $\overline{\mathrm{CTL1}}$ false (HIGH logic level)
- If *value* =2: $\overline{\mathrm{CTL0}}$ false (HIGH logic level) and $\overline{\mathrm{CTL1}}$ true (LOW logic level)
- If *value* =3: $\overline{\mathrm{CTL0}}$ and $\overline{\mathrm{CTL1}}$ 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*. Series 500 systems always use DMA, so use of this subroutine on Series 500 is meaningless, although it is supported for software compatibility reasons.

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 EIO. When a timeout error occurs, use *io_reset* to reset the GPIO interface before attempting another transfer.

Burst Transfers

The Integral PC and Series 200/300 support high-speed burst I/O on HP-IB and GPIO interfaces. Burst I/O is meaningless on Series 500 systems because they use DMA for GPIO transfers. 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 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.*

Interrupt-Driven Transfer Mode

Implemented on Integral PC Only:

The Integral PC supports two transfer modes on the HP 82923A GPIO interface: **flag-driven mode** and **interrupt-driven mode**. To select interrupt-driven mode, set the speed to zero using the *io_speed_ctl* subroutine.

When operating in interrupt-driven mode, *read* and *write* calls to the GPIO interface cause the calling process to go to sleep until an interrupt occurs at the completion of the *read* or *write.*

Series 500 Dependencies A

This appendix contains the following information which is specific to Series 500 systems:

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

Device I/O Library Location

The DIL subroutine library is contained in file */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 HP-IB and GPIO interfaces.

The GPIO Interface

The **GPIO** (General Purpose Input/Output) interface is a very flexible parallel interface that supports communication with a variety of devices. On Series 500 systems, the interface sends and receives up to 16 bits of parallel data with a choice of several handshake methods. External interrupt and user-definable signal lines provide additional flexibility.

The GPIO interface provides the following lines data and signalling lines:

- 16 parallel data input lines
- 16 parallel data output lines
- 4 handshake lines
- 4 special-purpose (status and control) lines.

Data Lines

There are 32 separate data lines: 16 for input and 16 for output. These lines normally use ground-true logic (LOW indicates true, HIGH indicates false). The logic can be changed so that a HIGH indicates true by changing the setting of the interface configuration option switches. Refer to the GPIO interface installation manual for more information.

Handshake Lines

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

- PCTL — Peripheral ConTroL
- PFLG — Peripheral FLaG
- I/$\overline{\text{O}}$ — Input/$\overline{\text{Output}}$.

The Peripheral Control (**PCTL**) line is driven by the interface and used to initiate data transfers. The Peripheral Flag (**PFLG**) line is driven by the peripheral device and used to indicate that a signal from the computer interface has been received and processed by the peripheral and the peripheral is ready for the next operation.

The Input/$\overline{\text{Output}}$ (**I/$\overline{\textbf{O}}$**) line is used to indicate direction of data flow.

The fourth handshake line is the External Interrupt Request (**EIR**) line. This line is used by the peripheral to signal interrupt service requests to the computer.

Special-Purpose (Control and Status) Lines

Four interface signal lines are available for any use you desire. Two are driven by the peripheral device and sensed by the computer; the other two are driven by the computer and sensed by the peripheral. These lines are most commonly used to transmit and receive control and status information beyond that which is normally available through PCTL and PFLG, hence their names $\overline{\text{CTL0}}$, $\overline{\text{CTL1}}$, $\overline{\text{STI0}}$, and $\overline{\text{STI0}}$

Data Handshake Methods

PCTL and PFLG support two handshake methods used to synchronize data transfers: **pulse-mode** and **full-mode** handshaking. If the peripheral uses pulses to handshake data transfers and meets certain hardware timing requirements, the pulse-mode handshake is used. Full-mode handshake should be used if the peripheral does not meet pulse-mode timing requirements. Refer to the GPIO interface installation manual for more information.

Latching Data Transfers

The GPIO interface design assumes very little on the part of the peripheral device. It has built-in data latching to hold data to and from the peripheral to ensure that no data is lost. Latching is performed as follows:

- When data is being output to the peripheral, the interface output register latches the data and holds it. The interface then asserts PCTL with I/$\overline{\text{O}}$ held LOW to indicate an output operation is in progress, and holds the data until PFLG is returned by the peripheral. The peripheral device must make proper use of the data, storing it if necessary, before data is removed upon receipt of PFLG.
- When data is being input from the peripheral, PCTL signals the peripheral (with I/$\overline{\text{O}}$ held HIGH to indicate an input operation) that the computer is ready to receive data. The peripheral must then place input data on the input lines to the computer then assert PFLG to indicate that the data is valid. PFLG is used to clock the input latches which means the peripheral can remove the data from the lines as soon as it has asserted PFLG.

The logic sense (ground-true or high-true logic) of the control and flag lines PCTL and PFLG is defined by the configuration switch for each line on the interface card. Consult the interface installation manual for more information about switch settings.

Creating the Interface Special File

HP-UX handles I/O to an interface the same way it handles I/O to any peripheral device: the interface must have a device 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 for special files associated with an interface.

Creating an Interface File

Special files are created using the *mknod(*1M) command which requires super-user access. When creating 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. The remaining parameters in the *mknod* command are as follows:

pathname

The *pathname* parameter specifies the name being given to the new interface special file. **pathname** identifies the interface file itself, not a peripheral connected to the interface. Special files are usually kept in the directory */dev.* This HP-UX convention is used because some commands expect to find device special files in the */dev* directory and fail if the file is 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
12	HP 27110A/B HP-IB Interface
18	HP 27110A GPIO Interface
37	Model 550 Internal HP-IB Interface.

minor_number

The *minor number* parameter identifies the location of the interface for *mknod.* The minor number is constructed as follows:

```
0xScAdUV
```

where:

`0x`	Identifies the remainder of the expression as a hexadecimal number. The two characters (zero followed by x) are entered exactly as shown.
`Sc`	A two-digit hexadecimal value specifying the select code of the interface card. The select code corresponds to the I/O slot in which the interface card resides.
`Ad`	A two-digit hexadecimal value specifying the device bus address. To use DIL subroutines 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, the file is *not* created as a raw file but rather as an auto-addressable file (in which 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 (as when using the GPIO interface), this component of the minor number is ignored (use `00` instead of a device bus address).
`U`	A single-digit hexadecimal value specifying a secondary address such as a device unit number. This component of the minor number is not used when creating interface special files; set it to 0.
`V`	A single-digit hexadecimal value specifying a secondary address such as a device volume number. This component of the minor number is not used when creating interface special files; set it to 0.

Creating an HP-IB Interface File

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

- Pathname is */dev/raw_hpib.*
- Internal HP-IB interface has major number 12.
- Interface card is located in slot 2 (select code 02), so the `Sc` component of the minor number is 02.
- Special file must be a **raw** special file in order to use DIL subroutines with it which means that the `Ad` portion of the minor number must be 31 (1f in hexadecimal).

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

```
mknod /dev/raw_hpib c 12 0x021f00
```

To further illustrate the use of *mknod*, suppose you have two HP 27110A HP-IB interface cards (major number = 12) installed in slots 2 and 3. The following two *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 12 0x021f00
mknod /dev/raw_hpib2 c 12 0x031f00
```

Creating a GPIO Interface File

Now suppose you also have a GPIO interface on the same Series 500 computer that you want to access using DIL subroutines.

The GPIO interface is does not use a bus architecture, so the usual bus address (`Ad`) and secondary address (`UV`) components of the minor number are ignored, and you need only determine the select code value before using *mknod.*

Assume that the GPIO interface is located in the I/O slot corresponding to select code 04 on your Series 500. The following *mknod* command creates the appropriate special file, named */dev/raw_gpio*:

```
mknod  /dev/raw_gpio  c  18  0x040000
```

Determining Interface Card Bus Address

The HP 27110A/B card always assumes bus address 30 when it is Active Controller. If control is passed to another device, the card assumes the address specified by the interface card configuration switch setting. The value of the current setting is easily determined by a call to *hpib_bus_status* which always returns the current bus address.

Effects of Resetting (via io_reset)

When *io_reset* is used on a Series 500 HP-IB interface,

- REN is cleared,
- The Interface Clear (IFC) line is pulsed,
- REN is reset,
- Interrupt mask is cleared, and
- The Peripheral Reset line (PRESET) is pulsed.

In addition, an interface self-test is performed. If the test fails, *io_reset* returns −1. If the interface successfully resets and completes self-test, *io_reset* returns 0.

Entity Identifiers

On Series 500, interface file entity identifiers used by DIL subroutines are equivalent to HP-UX file descriptors. This means that you can obtain entity identifiers for your interface files with the system routines *dup*, *fcntl*, and *pipe* as well as *open*.

DIL Subroutine Use Restrictions

This section presents various restrictions related to using DIL subroutines on Series 500 computers. Restrictions are arranged under headings named after the subroutine to which they apply. Subroutine names are treated in alphabetical order.

hpib_bus_status

A bug in the HP 27110A HP-IB interface card can cause an erroneous SRQ line state report. This error can occur during a narrow time window that allows *hpib_bus_status(eid, 1)* to report that the line is clear when in reality it is set. Since the subroutine never can report that the line is set when in reality it is clear, ORing successive readings of the SRQ line state minimizes the possibility of error. ORing five successive readings provides a result that is approximately 99% accurate. This bug has been fixed in the HP 27110B card.

On Series 500 systems, it is possible to check the SRQ line using *hpib_bus_status* and not see it asserted when it actually is. Because of this, the SRQ line should be checked at least 5 times to accurately determining whether or not it is asserted. If it is true any one of the 5 times, then the line is asserted (it never can be reported as asserted when it actually isn't). For example:

```
#include <fcntl.h>
main()
{
   int eid, value, i;

   eid = open("/dev/raw_hpib", O_RDWR);
   value = 0;
   for ( i=0; i<5; ++i)
       value = hpib_bus_status(eid,1) + value;
  /*Note that if SRQ is ever true during this test, "value" will be
  greater than 0*/

   if (value>0)
      service_routine();           /*SRQ is asserted; service the request*/
   else
      printf("No one is requesting service");
}
```

hpib_card_ppoll_resp

HP 27110A/B HP-IB interface cards do not support configuration of their parallel poll response under program control by use of *hpib_card_ppoll_resp*. Configuration can only be performed by the current Active Controller.

Unless programmed otherwise by the Active Controller, default *sense* of the HP 27110A/B interface's parallel poll response is always 1. If the interface's bus address (when not Active Controller) is 7 or less, the address determines the response line number as follows: bus data lines are labeled D0 through D7 corresponding to addresses 7 through 0, respectively (note the reverse order).

Thus, the parallel poll response of an HP 27110A/B at bus address 0 is a logic 1 on data line D7. An identical interface at bus address 7 responds with a 1 on line D0. If the address of the interface is greater than 7, there is no default line for it to respond on, so, unless its response is configured remotely by the Active Controller, it cannot respond at all.

If you want the interface to respond with a sense of 0 or on a different (non-default) line, it must be remotely configured by an Active Controller (this can be done by using *hpib_send_cmnd* while the interface is Active Controller).

hpib_rqst_srvce

This subroutine provides the capability for configuring an HP-IB interface's 8-bit response to serial polls. However, the HP 27110A/B HP-IB interface only allows you to set bit 6 of the response; all the other bits are cleared. If you set bit 6 of the serial response (where the response bits are labeled bit D0-D7) and the interface is not the Active Controller, the SRQ line is immediately asserted. The line remains asserted until the interface is serially polled or you clear bit 6 with *hpib_rqst_srvce.* If you set bit 6 and the interface is Active Controller, the interface remembers the response and asserts SRQ as soon as control passes to another controller.

Since you can only control bit 6 of the serial poll response, only the bit corresponding to decimal 64 in the response argument for *hpib_rqst_srvce* has any effect. Thus:

```
hpib_rqst_srvce(eid, 64);
```

sets bit 6 of the interface's serial poll response and:

```
hpib_rqst_srvce(eid, 0);
```

clears it.

hpib_send_cmnd

HP 27110A/B HP-IB and Model 550 Internal HP-IB interfaces send all *hpib_send_cmnd* commands with odd parity by overwriting 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 parity.

hpib_status_wait

hpib_status_wait holds off all other activity on the interface card associated with *eid* while it is in progress. Any other processes attempting to access the same interface will hang until the wait is terminated. Therefore, it is strongly recommended that a non-zero timeout be activated before calling *hpib_status_wait.*

hpib_wait_on_ppoll

hpib_wait_on_ppoll also holds off all other activity on the interface card while executing. As with *hpib_status_wait*, other processes attempting to access the interface card will hang, so it is recommended that a non-zero timeout be in effect before calling *hpib_wait_on_ppoll.*

io_get_term_reason

Subroutine *io_get_term_reason* is able to indicate one, two, or three reasons for a read termination by combining each reason into the three least significant bits in the returned value:

Set Bit	Decimal	Meaning
(none)	0	Abnormal terminaion.
Bit 0	1	Number of bytes requested were read.
Bit 1	2	Specified termination character was detected.
Bit 2	4	Device-imposed termination condition was detected (e.g., EOI on HP-IB).

For example, if *io_get_term_reason* returns 7 you know that the read terminated for three reasons: the byte count was reached, a termination character was encountered, and a termination condition was detected.

However, the Series 500 HP-IB interface does not return more than one termination reason to *io_get_term_reason*, but, rather, returns only the highest numbered reason. Consequently, *io_get_term_reason* can only return the values .0, 1, 2, or 4 (or −1 if an error occurs). For instance, if a 4 is returned, it indicates that a device-imposed termination condition occurred, but no mechanism exists for determining whether the byte count was reached or if a termination character was read as well.

When a Series 500 GPIO interface is set to use a 16-bit data path width, the termination character is only on byte (8 bits) wide (the least significant byte of the match value). During read operations, if the termination character arrives as the lower byte in a data transfer, data is handled and stored smoothly; both the upper and lower bytes of the transfer are received and the count of received bytes is incremented by two. However, if the termination character arrives in the upper-byte position of the transfer, both the upper byte and the lower byte are still read. However, the count of received bytes is only incremented by one, indicating that the termination character was located in the upper byte position.

io_on_interrupt

The internal HP-IB interface supplied with the Model 550 does not support talker-addressed, listener-addressed, controller-in-charge, or remote-enable interrupts. GPIO interrupts on the EIR line also are not supported.

io_timeout_ctl

io_timeout_ctl is used to set a time limit for I/O operations on a given entity identifier associated with an interface file. The timeout value specified in the subroutine call is a 32-bit long integer that determines the maximum timeout waiting period in microseconds. However, the effective timing resolution for timeouts is system-dependent. On Series 500 systems, timeout is rounded up to the nearest 10-millisecond boundary which means, for example, that if you specify a timeout of 155000 microseconds (155 milliseconds), the effective timeout is rounded up to 160 milliseconds.

When an I/O operation is aborted due to a timeout, *errno* is set to EIO. However, EIO is defined as a general I/O error, and can be set by many other error conditions. On Series 500 systems, more information can be obtained by looking at the external HP-UX variable *errinfo* which is set to the value 56 when a timeout occurs.

io_speed_ctl

The Series 500 always uses DMA for HP-IB and GPIO transfers, thus ensuring the fastest possible I/O speeds. Consequently, *io_speed_ctl* is meaningless when used in software intended for use on Series 500 systems. However, it is included in the Series 500 Device I/O Library to enhance software compatibility with Series 200/300 and other systems.

io_width_ctl

Although this subroutine can be called for any interface, the path width specified in the call to *io_width_ctl* must be compatible with the related interface. On Series 500 systems, only the GPIO interface supports multiple data path widths and only two widths are supported: 8 bits and 16 bits. *io_width_ctl* returns an error if a width is specified that is not available on the interface.

Performance Tips

When using DIL subroutines on Series 500 systems, overall I/O performance can be improved by following the basic guidelines listed in this section.

- Use buffers to hold data being written to an interface. Transferring data previously stored in a buffer is considerably faster than specifying a data string when invoking the transfer. For example, a data transfer handled by this code segment:

  ```
  int eid;       /*entity identifier descriptor*/
  char *buffer;      /*data storage buffer*/

  eid = open("/dev/raw_hpib", O_RDWR);
  buffer = "data message";   /*store data in buffer*/
  write(eid, buffer, 12); /*transfer data*/
  ```

  is faster than a data transfer handled by this equivalent code segment:

  ```
  int eid;       /*entity identifier descriptor*/

  eid = open("/dev/raw_hpib", O_RDWR);
  write(eid, "data message", 12); /*transfer data*/
  ```

- Make the number of bytes transferred equal to an integer multiple of the number of bytes per word in system memory. Data transfers, both in and out, are faster if the number of bytes being transferred fall on a word boundary. Series 500 memory is arranged in 4-byte words which means that the following code segment will perform optimally because the byte counts are integer multiples of 4.

  ```
  write(eid, buffer1, 12);
  read(eid, buffer2, 40);
  ```

- If you are super-user, you can use the *memlck*(2) routine (see *HP-UX Reference: Section 2*) to lock I/O process address space into physical memory. Data transfer times are reduced because transfers are handled directly from the user area without first moving data to the system area. However, one cannot lock an arbitrarily large amount of space for a given process because there is a point at which system performance begins to degrade.

- If a given process is running with an effective user ID of super-user, the process can be locked in memory by using *plock*(2) described in the *HP-UX Reference.* This lock is different than *memlck* mentioned previously. *plock*(2) informs the system that the process code, data, or both are not to be swapped out of memory. The following example illustrates the use of *plock*:

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

- Use auto-addressing for all read and write operations (see the Chapter 3 section "Setting up Talkers and Listeners" for details).
- *rtprio*(2) can be used 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. However, caution must be observed when using real time priorities because one can increase their priority above system processes resulting in possibly undesirable behavior.

  For example, if you request a real-time priority for your process that lies in the range of 0 through 63, your process has a higher priority than the system process that handles DIL interrupts. Such condition would cause interrupts to be lost if demand for CPU time became high enough that there was no available time to handle the interrupt.

  The following example code segment places the calling process at the lowest (least important) real time priority:

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

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

Series 200/300 Dependencies B

The following information, specific to Series 200/300 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.
- Information on how to simulate I/O interrupt programming on Series 200/300 computers.

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 200/300 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 an 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 wish 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) installed in slots 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 is 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 200/300 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 200/300 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 it fails, the routine returns a −1; otherwise the routine returns a 0.

Entity Identifiers

On Series 200/300 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 200/300 computers. These restrictions are organized under the routine to which they apply. The subroutines are presented in alphabetical order.

hpib_io

After calling *hpib_io*, the effects of any previous calls to *hpib_eoi_ctl* and *io_eol_ctl* are nullified. In other words, EOI mode is disabled for the specified *eid* and the read termination pattern is disabled. Therefore, if you want these to remain in effect after calling *hpib_io*, you must set them again with *hpib_eoi_ctl* and *io_eol_ctl.*

hpib_send_cmnd

The Series 200/300 HP-IB interface card uses odd parity when you send commands via *hpib_send_cmd.* 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.

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_interrupt_ctl

The *io_interrupt_ctl* routine is not supported on Series 200/300 computers.

io_on_interrupt

The *io_on_interrupt* routine is not supported on Series 200/300 computers.

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 200/300 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

The performance of your I/O process on a Series 200/300 computer using DIL subroutines can be improved by following the guidelines below:

- Use the *io_burst* routine for small data transfers. ("Small" on a Series 300 Model 310 is less than 1Kb; "small" on a Series 300 Model 320 is less than 4Kb.)
- If you are the super-user, you can use the *memlck(2)* routine (see *HP-UX Reference: Section 2*) to lock your I/O process's address space into physical memory. Data transfer times are reduced because they are carried out directly from the user area and do not have to be first moved to the system area. However, you cannot lock an arbitrarily large amount of space for your process since there is a point at which your system's performance will begin to degrade.
- For processes running with an effective user ID of super-user, it is possible to lock the process in memory with *plock(2)* (see *HP-UX Reference*). This lock is different than *memlck* (as mentioned above). *plock(2)* informs the system that the process code, data, or both are not to be swapped out of memory. The following example illustrates the use of *plock*:

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

- Use auto-addressing for all read and write operations. (See the section "Setting up Talkers and Listeners" of Chapter 3, "Controlling the HP-IB Interface," for details.)
- Increasing the system priority of an I/O process can be accomplished by using *rtprio(2)*. *rtprio* requires the process to be running with an effective user *ID* of super-user. The real time priorities available with *rtprio* are non-degrading priorities. Caution must be observed when using real time priorities since one can increase their priority above system processes. This may cause undesirable behavior. For example, requesting a real time priority in the range of 0-63 places your process in a higher priority than the DIL interrupt handler system process. This means that interrupts could be lost if there is not sufficient CPU resource available. 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;      /* a zero process # 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*/
}
```

Simulating Interrupts for the HP-IB Interface

Although Series 200 HP-UX does not allow you to set interrupts, the use of four system subroutines *fork(2)*, *signal(2)*, *kill(2)*, and *getpid(2)* allows you to simulate their effect. The purpose of this section is not to describe how these subroutines work, but merely to present a specific application that uses them. Refer to *HP-UX Reference: Section 2* for a complete description of the four system subroutines.

You can simulate setting an interrupt by creating a child process that waits for the interrupt condition. When that condition occurs, the child process sends a signal back to the parent process and then terminates. While the child process is waiting for the specified condition, the parent process can continue executing until it receives the signal from the child, at which time it jumps to a specified service routine.

The code below illustrates how you can use *fork* to spawn a child process that waits for a particular bus condition. Here the child process calls *hpib_status_wait* to wait until the SRQ line is asserted. Since no timeout has been set for the interface file's entity identifier, there is no limit to how long the child process waits for the specified condition. When the SRQ line is seen, the child process sends the signal SIGINT to the parent process using *kill*. Since *kill* requires the process ID of the process that is to receive the signal, *getpid* is called. *Getpid* returns the process ID of the calling process's parent process. The child process terminates after the signal is sent. *Signal* allows you to specify in the parent process what signal it is to look for and what routine it is to execute when the signal is received. The code for *service_routine* is not shown here. After *service_routine* is executed, the parent process resumes execution at the point where it was interrupted.

```
#include <signal.h>                      /*defines various signals*/
main()
{
   int eid;
   eid = open( "/dev/raw_hpib", O_RDWR);  /*open interface file*/

   /*create a new process that will look for service requests*/
   if (fork() == 0)     /*this is the child process*/
   {
       hpib_status_wait( eid, 1);       /*note that no timeout is set--it
                                        will wait indefinitely for SRQ*/
       kill(getpid(), SIGINT);
   }

   else                                 /*this is the parent*/
   {
       signal(SIGINT, service_routine);
          :
   /*parent process can now do other things while the child waits
     for SRQ. When the parent receives the SIGINT signal the function
     service_routine will be executed.*/
          :
   }
}
```

Some additional points about simulating interrupts in this way are:

- The code for the child process can be distinguished from that of the parent process by the value returned by *fork*. *Fork* returns a 0 in the child process and the process ID of the child process to the parent process.
- The include file *signal.h* must appear near the beginning of your program if the program calls *signal*.
- If the interface file is opened before the *fork* call, the child process inherits the file's entity identifier. If *fork* is called before the interface file is opened, then both the child and the parent processes must open it.

Simulating Interrupts on the GPIO Interface

Chapter 3: Controlling the HP-IB Interface discusses the use of four system subroutines *fork*, *signal*, *kill* and *getpid* to simulate the effect of an interrupt when a certain condition occurs on an HP-IB interface. This same technique can be used to simulate an interrupt given a certain condition on a GPIO interface, such as a certain value of the STI0 and STI1 special purpose status lines.

Fork is used to spawn a child process that waits for a specified condition to occur, leaving the parent free to continue executing. When the condition occurs, the child process sends a signal via *kill* to the parent which then implements whatever service routine is required. The parent process uses *signal* to recognize when the signal is sent and the child process uses *getpid* to find out the process ID of the parent so that it knows where to send the signal. The code below illustrates generating an **interrupt** when a peripheral connected to the GPIO interface asserts STI0.

```
#include <signal.h>                /*defines various signals*/
main()
{
   int eid;                        /*entity identifier*/

   eid = open("/dev/raw_gpio", O_RDWR);  /*open GPIO interface file*/
   /*create a child process that looks for assertion of STI0*/

   if (fork() == 0)                /*this is the child process*/
   {
      wait_on_STI0(eid);           /*call a routine that waits for STI0*/
      kill(getpid(), SIGINT);      /*send signal to parent process*/
   }
   else                            /*this is the parent process*/
   {
      signal(SIGINT, service_routine());
      :
      /*parent process can now do other things while the child waits for
       STI0. When the parent receives the signal SIGINT, the function
       ''service_routine'' will be executed*/ ... ... } } /*end of main*/

      /*"wait_on_STI0" repeatedly calls gpio_get_status until it sees that
        STI0 is asserted and then it returns to the calling routine*/

       wait_on_STI0(eid)
       int eid;
        {
        int value;               /*Variable to hold value of STI0 and STI1*/
        int flag = 0;            /*Boolean flag initialized to 0 (false)*/

        while (flag == 0)
        {
             value = gpio_get_status(eid); /*read STI0 and STI1 lines*/
             if (value & 1)        /*clear all but the first bit*/
                flag = 1;          /*when STI0 is asserted, set flag to 1*/

   }
}
```

Integral PC Dependencies C

The following information, specific to the Integral PC, is discussed in this appendix:

- location of the DIL routines
- the GPIO interface
- creating an interface special file
- interrupts
- controlling the HP-IB interface
- non-standard DIL routines
- restrictions using the DIL routines

Location of the DIL Routines

The DIL routines are supplied in the *libdvio.a* library on the DIL disc. To use this library with your compiler, move the *libdvio.a* library, along with the include files, to the appropriate folder for your compiler, usually */usr/lib*.

The GPIO Interface

The HP 82923A GPIO interface used on the Integral PC is different in a number of key areas from the GPIO used on Series 200/300 and 500 computers. Refer to the *HP 82923A GPIO Interface Owner's Manual* for a complete description of the hardware. Note that the HP 82923A GPIO interface has the following features:

- parameters are set using DIL routines, not switches; these DIL routines are non-standard DIL routines and are only provided on the Integral PC
- four 8-bit bidirectional data ports (which can be configured in 8-, 16-, or 32-bit ports)
- 2 handshaking lines for each port
- 1 peripheral interrupt line (PIR) for each port
- 1 reset line (RES) for each port
- 1 status line for each port
- 1 data direction line ($I/\overline{O}$) for each port.

The HP 82923A GPIO interface has six handshake types. The handshake type is selected using the *gpio_handshake_ctl* routine.

Creating an Interface Special File

Two utility programs, *load_hpib* and *load_gpio*, must be used to create the appropriate special files for your HP-IB and GPIO interfaces, respectively. These routines create a special (device) file for each HP-IB or GPIO interface found, and load the appropriate DIL driver. The data files containing the DIL drivers, *dhpib.data* and *dgpio.data*, must be in the search path defined by your PATH variable when the load utility is invoked. For more information on *load_hpib* and *load_gpio* refer to the *load_hpib.1* and *load_gpio.1* files provided in the *doc* folder on the DIL disc.

GPIO Interface Files

The special files for GPIO interfaces have the following form:

`/dev/gpio`*GPIO_port*.*IO_port*

where *GPIO_port* is the letter designation for GPIO ports `a`, `b`, `c`, or `d`; and *IO_port* is a one- or two-character designation (`a`, `b`, `a1`, `a2`,...) for the Integral PC I/O port. Note that the top port on the Integral PC is port `a`, the bottom port is port `b`, while the bus expander ports have a combination letter and number designation as shown below.

HP-IB Interface Files

The special (device) files for HP-IB interfaces have two forms:

`/dev/dhpib.i`	for the built-in HP-IB interface
`/dev/dhpib.`*IO_port*	for the plug-in HP-IB interface, where *IO_port* is the Integral PC I/O port designator (`a`, `b`, `a1`, `a2`,...) described above.

Unloading the DIL Drivers

Two additional utilities, *unload_hpib* and *unload_gpio*, are provided on the DIL disc. These utilities are used to remove both the DIL drivers and the special files created by *load_hpib* and *load_gpio*. For more information on using these utility programs, refer to *load_hpib.1* and *load_gpio.1* in the *doc* folder on the DIL disc.

Interrupts

Unlike the Series 500, the Integral PC supports only one interrupt condition, PIR, meaning that the Peripheral Interface Request line has been asserted. For hardware restrictions on using the HP-IB interrupts on the Integral PC, refer to the *io_on_interrupt.3d* file in the *doc* folder on the DIL disc.

Controlling the HP-IB Interface

Limitations on the HP-IB Interface

The use of DIL routines with the built-in HP-IB interface has the following limitations:

- The user must not pass control when using the DIL routines with the built-in HP-IB interface. The built-in HP-IB interface *must* always be the System Controller/Active Controller.
- Loading the DIL drivers and then opening the *built-in* HP-IB interface special file prevents the operating system from accessing printers, plotters, and mass-storage drives on the built-in HP-IB interface until the built-in HP-IB interface special file is closed. This means that any operation using a printer, plotter, or mass-storage device on the built-in HP-IB interface will be suspended until the built-in HP-IB device file is closed. This limitation can result in a deadlock situation if your program both uses the DIL routines with the built-in HP-IB interface and attempts to use a printer, plotter, or mass-storage drive on the built-in HP-IB interface.

To avoid these limitations, we recommend that you **use the HP-IB DIL routines only with the HP 82998A HP-IB interface**.

The Computer as a Non-Active Controller

The built-in HP-IB interface must be in the system controller, active controller state to use the DIL routines on the Integral PC.

Non-Standard DIL Routines

The Integral PC DIL library supports several routines that are not part of the DIL standard. This section describes these routines.

General-Purpose Routines

In addition to the standard DIL routines, the Integral PC DIL library supports the following two routines:

io_lock	Locks the interface port to the calling process until the *io_unlock* routine is called.
io_unlock	Used by the calling process to remove the lock created by *io_lock.*

For details on using these routines, refer to the *io_lock.3d* file located in the *doc* folder on the DIL disc supplied with your Integral PC.

Non-Standard HP-IB Routines

In addition to the standard DIL routines for controlling the HP-IB interface, the Integral PC supports the following non-standard DIL routine:

io_burst(eid, flag)	Used to control the high-speed HP-IB mode. If *flag* = 0, high-speed mode is turned off; otherwise it is turned on.

For information on the *io_burst* routine, refer to the *io_burst.3d* file in the *doc* folder on the DIL disc.

Non-Standard GPIO Routines

The following non-standard DIL routines have been added to control the HP 82923A GPIO interface:

- *gpio_handshake_ctl*
- *gpio_normalize_ctl*
- *gpio_delay_time_ctl*

A description of these routines is provided in the *doc* folder on the DIL disc.

Restrictions Using the DIL Routines

This section presents some restrictions on using DIL routines with the Integral PC computer. Restrictions on using system routines, such as *open(2)*, are also discussed here. These restrictions are organized under the routine to which they apply; the routines are presented in alphabetical order.

hpib_bus_status

On the Integral PC, it is not possible to determine the status of the NDAC and SRQ lines under certain conditions. This can result in incorrect results when using the *hpib_bus_status* routine to determine the status of these two lines. If the HP-IB interface is talk-addressed, the SRQ status is incorrect; if it is listen-addressed, the NDAC status is incorrect.

hpib_card_ppoll_resp

The parallel poll response of the HP 82998A HP-IB interface can *not* be remotely programmed. Instead, use the *hpib_card_ppoll_resp* routine.

hpib_ppoll_resp_ctl

The "sense" bit of the flag value for the *hpib_ppoll_resp_ctl* routine determines whether a zero or non-zero "response value" means that the computer requires service. If the "s" bit is a 0, then a zero response value means service is needed.

io_eol_ctl

On the Integral PC, a read operation from a GPIO interface will terminate only when a specified number of read operations have been performed, or when the read termination pattern has been found.

The Integral PC does not support different read termination patterns on multiple opens to the same *eid.*

io_reset

When used to reset a GPIO interface, the *io_reset* routine will pulse the $\overline{\text{RES}}$ (reset) line only on the GPIO controller port specified by the *eid.*

io_speed_ctl

GPIO

Setting the speed on a GPIO interface determines the transfer mode used by the driver: either interrupt-driven, flag-driven handshake, or "fast handshake" mode. (Note that the driver's fast handshake mode is not the same as the fast handshake mode described in the *HP 82923A GPIO Owner's Manual*; it refers to a flag-driven mode where the EOL and timeout settings are ignored to achieve a faster transfer rate.)

DMA transfers are not available on the Integral PC.

Interrupt-Driven Transfer Mode

Two transfer modes exist between the Integral PC and the HP 82923A GPIO interface: flag-driven mode and interrupt-driven mode. To select the interrupt-driven mode, use *io_speed_ctl* to set the speed to 0.

While in the interrupt-driven mode, *read* and *write* calls to the GPIO interface will cause the user's process to go to sleep until an interrupt occurs at the completion of the *read* or *write*.

HP-IB

The DIL routines on the Integral PC support two HP-IB transfer modes: flag-driven mode and high-speed transfer mode. The default mode is the flag-driven mode until it is set to the high-speed transfer mode using the *io_burst* routine.

In the high-speed transfer mode, the driver talks directly to the interface without going through the operating system. For more information on *io_burst*, refer to the documentation provided in the *io_burst.3d* file in the *doc* folder on the DIL disc.

io_timeout_ctl

This routine allows you to set a time limit for operations carried out by DIL routines on a specified entity identifier. The timeout value you specify is a 32-bit long integer that indicates the length of the timeout in microseconds (μ-secs). However, the resolution of the effective timeout is system-dependent. On the Integral PC, the timeout resolution on both the HP 82923A GPIO interface and the HP 82998A HP-IB interface is 1 millisecond (msec).

For example, suppose you specify a timeout of 99 999 microseconds (99.999 milliseconds). Then the effective timeout is rounded up to 100 milliseconds.

io_width_ctl

The data path width for the HP-IB interface is always 8 bits on the Integral PC. However, the four 8-bit ports on the HP 82923A GPIO interface can be combined to form 8-, 16-, or 32-bit data paths.

For 16- or 32-bit ports, only one port acts as a controller; that port's *eid* is used in the *io_width_ctl* routine. The allowable data path widths for each port are shown in the following table.

GPIO Data Path Widths

Data Path Width	Controller Port	Data Ports*
8-bit	a	a
	b	b
	c	c
	d	d
16-bit	b	b a
	d	d c
32-bit	b	b a d c

* Data ports are listed in order, left to right, from most-significant byte to least-significant byte.

Combinations of 8- and 16-bit or two 16-bit ports are also allowed on the same GPIO interface. 24-bit ports are not allowed.

open(2)

When opening the special file for an interface, you must use the special file name for the specific GPIO or HP-IB interface created by *load_hpib* or *load_gpio*. Note that each GPIO port has a separate special file name. For details on interface special file names, see the previous section "Creating an Interface Special File."

read(2) and write(2)

During a read or write operation to a 16- or 32-bit **GPIO** port, the data must start on a word boundary. This restriction applies only to the GPIO interface.

Series 800 Dependencies D

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 Model 840 I/O backplane. The default I/O configuration shipped with your Model 840 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:

```
0x00Lu00
```

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** *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.

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.

Routine	Same Effect	Different Effect	Not Allowed
hpib_abort	X		
hpib_bus_status	X		
hpib_card_ppoll_resp		X	
hpib_eoi_ctl	X		
hpib_io		X	
hpib_pass_ctl			X
hpib_ppoll	X		
hpib_ppoll_resp_ctl			X
hpib_ren_ctl		X	
hpib_rqst_srvce			X
hpib_send_cmd		X	
hpib_spoll		X	
hpib_status_wait			X
hpib_wait_on_ppoll		X	
io_eol_ctl	X		
io_get_term_reason	X		
io_interrupt_ctl	X		
io_on_interrupt		X	
io_reset			X
io_speed_ctl	X		
io_timeout_ctl	X		
io_width_ctl	X		

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.

ASCII Character Codes

E

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
NUL	0	00000000	000	00	
SOH	1	00000001	001	01	GTL
STX	2	00000010	002	02	
ETX	3	00000011	003	03	
EOT	4	00000100	004	04	SDC
ENQ	5	00000101	005	05	PPC
ACK	6	00000110	006	06	
BEL	7	00000111	007	07	
BS	8	00001000	010	08	GET
HT	9	00001001	011	09	TCT
LF	10	00001010	012	0A	
VT	11	00001011	013	0B	
FF	12	00001100	014	0C	
CR	13	00001101	015	0D	
SO	14	00001110	016	0E	
SI	15	00001111	017	0F	
DLE	16	00010000	020	10	
DC1	17	00010001	021	11	LLO
DC2	18	00010010	022	12	
DC3	19	00010011	023	13	
DC4	20	00010100	024	14	DCL
NAK	21	00010101	025	15	PPU
SYNC	22	00010110	026	16	
ETB	23	00010111	027	17	
CAN	24	00011000	030	18	SPE
EM	25	00011001	031	19	SPD
SUB	26	00011010	032	1A	
ESC	27	00011011	033	1B	
FS	28	00011100	034	1C	
GS	29	00011101	035	1D	
RS	30	00011110	036	1E	
US	31	00011111	037	1F	

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
space	32	00100000	040	20	LA0
!	33	00100001	041	21	LA1
"	34	00100010	042	22	LA2
#	35	00100011	043	23	LA3
$	36	00100100	044	24	LA4
%	37	00100101	045	25	LA5
&	38	00100110	046	26	LA6
'	39	00100111	047	27	LA7
(	40	00101000	050	28	LA8
)	41	00101001	051	29	LA9
*	42	00101010	052	2A	LA10
+	43	00101011	053	2B	LA11
,	44	00101100	054	2C	LA12
–	45	00101101	055	2D	LA13
.	46	00101110	056	2E	LA14
/	47	00101111	057	2F	LA15
0	48	00110000	060	30	LA16
1	49	00110001	061	31	LA17
2	50	00110010	062	32	LA18
3	51	00110011	063	33	LA19
4	52	00110100	064	34	LA20
5	53	00110101	065	35	LA21
6	54	00110110	066	36	LA22
7	55	00110111	067	37	LA23
8	56	00111000	070	38	LA24
9	57	00111001	071	39	LA25
:	58	00111010	072	3A	LA26
;	59	00111011	073	3B	LA27
<	60	00111100	074	3C	LA28
=	61	00111101	075	3D	LA29
>	62	00111110	076	3E	LA30
?	63	00111111	077	3F	UNL

Character Codes (cont.)

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
@	64	01000000	100	40	TA0
A	65	01000001	101	41	TA1
B	66	01000010	102	42	TA2
C	67	01000011	103	43	TA3
D	68	01000100	104	44	TA4
E	69	01000101	105	45	TA5
F	70	01000110	106	46	TA6
G	71	01000111	107	47	TA7
H	72	01001000	110	48	TA8
I	73	01001001	111	49	TA9
J	74	01001010	112	4A	TA10
K	75	01001011	113	4B	TA11
L	76	01001100	114	4C	TA12
M	77	01001101	115	4D	TA13
N	78	01001110	116	4E	TA14
O	79	01001111	117	4F	TA15
P	80	01010000	120	50	TA16
Q	81	01010001	121	51	TA17
R	82	01010010	122	52	TA18
S	83	01010011	123	53	TA19
T	84	01010100	124	54	TA20
U	85	01010101	125	55	TA21
V	86	01010110	126	56	TA22
W	87	01010111	127	57	TA23
X	88	01011000	130	58	TA24
Y	89	01011001	131	59	TA25
Z	90	01011010	132	5A	TA26
[	91	01011011	133	5B	TA27
\	92	01011100	134	5C	TA28
]	93	01011101	135	5D	TA29
^	94	01011110	136	5E	TA30
_	95	01011111	137	5F	UNT

ASCII Char.	EQUIVALENT FORMS				HP-IB
	Dec	Binary	Oct	Hex	
`	96	01100000	140	60	SC0
a	97	01100001	141	61	SC1
b	98	01100010	142	62	SC2
c	99	01100011	143	63	SC3
d	100	01100100	144	64	SC4
e	101	01100101	145	65	SC5
f	102	01100110	146	66	SC6
g	103	01100111	147	67	SC7
h	104	01101000	150	68	SC8
i	105	01101001	151	69	SC9
j	106	01101010	152	6A	SC10
k	107	01101011	153	6B	SC11
l	108	01101100	154	6C	SC12
m	109	01101101	155	6D	SC13
n	110	01101110	156	6E	SC14
o	111	01101111	157	6F	SC15
p	112	01110000	160	70	SC16
q	113	01110001	161	71	SC17
r	114	01110010	162	72	SC18
s	115	01110011	163	73	SC19
t	116	01110100	164	74	SC20
u	117	01110101	165	75	SC21
v	118	01110110	166	76	SC22
w	119	01110111	167	77	SC23
x	120	01111000	170	78	SC24
y	121	01111001	171	79	SC25
z	122	01111010	172	7A	SC26
{	123	01111011	173	7B	SC27
|	124	01111100	174	7C	SC28
}	125	01111101	175	7D	SC29
~	126	01111110	176	7E	SC30
DEL	127	01111111	177	7F	SC31

DIL Programming Example

F

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>

/* HP-IB addressing group bases */
#define    LAG_BASE       0x20     /*  listener address base */
#define    TAG_BASE       0x40     /*  talker address base */
#define    SCG_BASE       0x60     /*  secondary address base */

/* HP-IB command equates in odd parity */
#define    GTL         0x01       /*  go to local */
#define    SDC         0x04       /*  selective device clear */
#define    DCL         0x94       /*  device clear */
#define    UNL         0xbf       /*  unlisten */
#define    UNT         0xdf       /*  untalk */

```

```
/* HP-IB secondary commands */
#define            PR_SEC_DSJ      SCG_BASE+16
#define            PR_SEC_DATA     SCG_BASE+0
#define            PR_SEC_RSTA     SCG_BASE+14
#define            PR_SEC_MASK     SCG_BASE+01
#define            PR_SEC_STRD     SCG_BASE+10     /* 2608A */

/* output of DSJ operation 2608A */
#define            PR_ATTEN        0x0001
#define            PR_RIBBON       0x0002
#define            PR_ATT_PAR      0x0003
#define            PR_PAPERF       0x0010
#define            PR_SELF         0x0020
#define            PR_PRINT        0x0040

/* output of DSJ operation the rest of the printers */
#define            PR_RFDATA       0x0000
#define            PR_SDS          0x0001
#define            PR_RIOSTAT      0x0002

/* ppoll mask bits */
#define            PR_M_RFD        0x0010
#define            PR_M_STATUS     0x0020
#define            PR_M_POWER      0x0040
#define            PR_M_PAPER      0x0080

/* default parallel poll mask */
unsigned char pmask[1] = {PR_M_PAPER+PR_M_POWER+PR_M_STATUS+PR_M_RFD};

/* masks for io status byte in case of 2608A */
#define    PR_I_POW        0x0001
#define            PR_I_OPSTAT     0x0040
#define            PR_I_LINE       0x0080

/* masks for io status byte the rest of the printers */
#define    PR_I_POWER      0x0001
#define            PR_I_PAPER      0x0002
#define            PR_I_PARITY     0x0008
#define            PR_I_RFD        0x0040
#define            PR_I_ONLINE     0x0080

/* define printer types */
#define T2608A             1
#define T2631A             2
#define T2631B             3
#define T2673A             4
#define QjetPlus   5
#define T2632A             6
#define T2634A             7

```

```
int ptr_type;      /* type of printer */

/* setup defines for fatal returns */
#define F_RTRN     1
#define F_EXIT     0

/* setup defines for HP-IB_msg */
#define H_READ             1
#define H_WRITE    2
#define H_CMND             4

/* default timeout value (in seconds) to infinity */
int timeout =      0;

/* default size of output buffer to printer */
int bufsz =        32;

/* 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 */

   /* GET USER SUPPLIED OPTIONS AND PRINTER FILE NAME */
   while ((i = getopt(argc, argv, "b:t:p:D")) != EOF) {
           switch (i) {
           /* set the buffer size to output to printer */
           case 'b': if ((bufsz = atoi(optarg)) <= 0) errflg++;
                   break;

           /* get the new timeout value in seconds */
           case 't': if ((timeout = atoi(optarg)) < 0) errflg++;
                   break;

           /* Set the parallel poll pmask (mostly for debugging) */
           case 'p': if ((pmask[0] = atoi(optarg)) < 0) errflg++;
                   break;

           case 'D': Debug++; break;

           case '?': errflg++;break;
           }
   }
   /* get printer dev if supplied */
   if (optind < argc)
           strcpy(ptr_dev, argv[optind]);

   if (errflg) {
fprintf(stderr, "usage: %s [-bbufsz -ttmout] [printer_dev]\n", procnam);
fprintf(stderr, "-b bufsz > Output buf size to printer (%d)\n", bufsz);
fprintf(stderr, "-t tmout > Max seconds to output buffer (%d)\n", timeout);
fprintf(stderr, "printer_dev > Printer device file     (%s)\n", ptr_dev);
fprintf(stderr, "-p ppoll_mask > Parallel poll mask  (0x%02x)\n",pmask[0]);
           exit(2);
   }
   /* get memory for the output buffer */
   outbuf = (unsigned char *)malloc (bufsz + 4);
/*
   NOTE: Printer device file (/dev/lp) is used only to get printer select
   code and HP-IB bus address.  This is because attention-true (ATN)
   requests can only be sent to an "HP-IB raw bus device file".  Therefore
   after getting the SC and BA we will use a "HP-IB raw bus device file" to
   do all the work, but it must exist with a name similar to the printer
   device; i.e. "/dev/lp" is changed to "/dev/lp_07", where the "07" is the
   select code.
```

```
*/
   /* check if printer device exists */
   if (stat(ptr_dev, &statbuf) < 0)
           fatal_err("stat", ptr_dev, F_EXIT);

   /* check if it is a character device file */
   if ((statbuf.st_mode & S_IFMT) != S_IFCHR)
           fatal_err("Must be a char_special file", ptr_dev, F_EXIT);

   /* extract selectcode from the printer device */
   selcode =       m_selcode(statbuf.st_rdev);

   /* make the HP-IB raw bus device file name from selectcode */
   ptr_raw[1] += selcode / 16;
   ptr_raw[2] += selcode % 16;
   if ((selcode % 16) >= 10) ptr_raw[2] += ('a' - '0' -10);
   strcat(ptr_dev, ptr_raw);

   /* get device BA from the printer device and config control bytes */
   devba = m_busaddr(statbuf.st_rdev);
   DLA = LAG_BASE + devba; /* device listen address */
   DTA = TAG_BASE + devba; /* device talk address */

   /* open the HP-IB raw bus device */
   if ((eid = open(ptr_dev, O_RDWR)) <0) {
           fatal_err("Raw HP-IB open", ptr_dev, F_RTRN);
fprintf(stderr,
" The following commands executed as a super user may be necessary\n\n");
fprintf(stderr, "     # mknod %s c 21 0x%s1f00\n", ptr_dev, &ptr_raw[1]);
fprintf(stderr, "     # chmod 555 %s\n", ptr_dev);
fprintf(stderr, "     # chown lp  %s\n", ptr_dev);
           exit(2);
   }
   /* get (my) BA of the controller and configure control bytes */
   if ((myba = hpib_bus_status(eid, 7)) < 0)
           fatal_err("Must be raw hpib driver (21)", ptr_dev,F_EXIT);
   MLA = LAG_BASE + myba; /* controller (my) listen address */
   MTA = TAG_BASE + myba; /* controller (my) talk address */

   /* go do the Amigo identify */
   ptr_type = amigo_identify();

   if (Debug) {
           printf("%s Identified ", ptr_dev);
           switch(ptr_type) {
           case T2608A:     printf("2608A");        break;
           case T2631A:     printf("2631A");        break;
           case T2631B:     printf("2631B");        break;
           case T2673A:     printf("2673A");        break;
           case QjetPlus:   printf("QuietJet Plus");break;
```

```
		case T2632A:	printf("2632A");	break;
		case T2634A:	printf("2634A");	break;
		default: printf("You forgot one dummy"); break;
		}
		printf(" printer\n");
	}
	/* set the timeout to user requested value */
	if (io_timeout_ctl(eid, timeout * 1000000) < 0)
		fatal_err("io_timeout_ctl", ptr_dev, F_EXIT);

	/* always tag last output data byte with EOI */
	if (hpib_eoi_ctl(eid, 1) < 0)
		fatal_err("hpib_eoi_ctl", ptr_dev, F_EXIT);

	/* clear out the status bits */
	amigo_clear();

	/* check the status bits */
	status = amigo_status();
	if (Debug) printf("%s Printer status = 0x%x\n", ptr_dev, status);

	/* set the ppoll mask required by some printers */
	amigo_set_pmask();

	/* MAIN OUTPUT LOOP */
	i = 0;
	while ((c = getchar()) != EOF) {
		if (i == bufsz) {
			amigo_write(outbuf, i);
			i = 0;
		}
		outbuf[i++] = c;
	}
	/* post remaining buffer */
	if (i) amigo_write(outbuf, i);
	exit(0);
}

/* ROUTINE TO DO THE MAIN I/O TO THE BUSS */
/* lock bus, do preamble, read/write, do postamble and unlock bus */
/* preamble must be 3 or 4 bytes, postamble must be 1 or 2 bytes */
int
HPIB_msg(rw_flag, pcm1, pcm2, pcm3, buffer, length, ocm0, ocm1)
int		rw_flag;
int		pcm1;
int		pcm2;
int		pcm3;
char		*buffer;
int		length;
int		ocm0;
```

```
int        ocm1;
{
   unsigned char pre_cmd[4];
   unsigned char post_cmd[2];
   int tlog = -1;

   pre_cmd[0] = UNL;       /* always issue unlisten command first */
   pre_cmd[1] = pcm1;
   pre_cmd[2] = pcm2;
   pre_cmd[3] = pcm3;

   post_cmd[0] = ocm0;
   post_cmd[1] = ocm1;

   /* first get exclusive use of the bus */
   if (io_lock(eid) < 0)
           fatal_err("io_lock", ptr_dev, F_EXIT);

   /* send the preamble 3 or 4 bytes with attention true */
   if (hpib_send_cmnd(eid, pre_cmd, (pcm3 ? 4 : 3)) < 0)
           fatal_err("hpib_send_cmnd preamble", ptr_dev, F_EXIT);

   switch (rw_flag) {
   case H_READ:
           if ((tlog = read(eid, buffer, length)) < 0)
                   fatal_err("read", ptr_dev, F_EXIT);
           break;

   case H_WRITE:
           if ((tlog = write(eid, buffer, length)) < 0)
                   fatal_err("write", ptr_dev, F_EXIT);
           break;

   case H_CMND:
           return(0);
   default:
           return(-1);
   }
   /* send the postamble 1 or 2 bytes with attention true */
   if (hpib_send_cmnd(eid, post_cmd, (ocm1 ? 2 : 1)) < 0)
           fatal_err("hpib_send_cmnd postamble", ptr_dev, F_EXIT);

   /* at last unlock the bus so other bus users can access it */
   if (io_unlock(eid) < 0)
           fatal_err("io_unlock", ptr_dev, F_EXIT);

   return(tlog);
}

int
```

```
amigo_identify()
{
   unsigned char identify[2];

   /* TLK31 (UNT) is special for amigo identify */
   /* finish with a MTA (UNT is not save for non-amigo devices) */
   HPIB_msg(H_READ, MLA, UNT, SCG_BASE + devba, identify, 2, MTA, 0);

   switch(identify[0]) {
   case 32:
           /* Amigo identify */
           switch(identify[1]) {
           case 1:  return(T2608A);
           case 2:  return(T2631A);
           case 9:  return(T2631B);
           case 11: return(T2673A);
           case 13: return(QjetPlus);
           case 16: return(T2632A);
           case 17: return(T2634A);
           default:
                   printf("Unrecognized Amigo printer, ID2 = %d\n",
                           identify[1]); break;
           }
           break;
   case 33:
           if (identify[1] == 1)
                   printf("Ciper printer not supported yet!\n");
           break;
   default:
printf("Unrecognized Amigo Printer identify, ID1 = %d, ID2 = %d\n",
                   identify[0], identify[1]);
           break;
   }
   exit(2);
}

/* set the parallel poll mask value */
amigo_set_pmask()
{
   HPIB_msg(H_WRITE, MTA, DLA, PR_SEC_MASK, pmask, 1, UNL, 0);
}

/* do the amigo clear followed by selective device clear */
amigo_clear()
{
   HPIB_msg(H_WRITE, MTA, DLA, SCG_BASE + 16, "\0", 1, SDC, UNL);
}

/* get the dsj byte */
int
```

```
amigo_dsj()
{
   unsigned char dsj_byte[1];

   HPIB_msg(H_READ, MLA, DTA, PR_SEC_DSJ, dsj_byte, 1, UNT, 0);
   return(dsj_byte[0]);
}

/* return the amigo status byte */
int
amigo_status()
{
   unsigned char status_byte[1];

   HPIB_msg(H_READ, MLA, DTA, PR_SEC_RSTA, status_byte, 1, UNT, 0);
   return(status_byte[0]);
}

/* output a buffer to printer */
amigo_write(buffer, length)
char *buffer;
int length;
{
   int status, dsj = 0;

   /* write the buffer */
   HPIB_msg(H_WRITE, MTA, DLA, PR_SEC_DATA, buffer, length, UNL, 0);
again:
   /* now wait for parallel poll response */
   if (Debug)  printf("%s Ppoll wait\n", ptr_dev);
   if (hpib_wait_on_ppoll(eid, 0x80>>devba, 0) < 0)
           fatal_err("hpib_wait_on_ppoll", ptr_dev, F_EXIT);

   /* a DSJ is required to remove the ppoll response from device */
   if (dsj = amigo_dsj()) {
           if (Debug) printf("%s DSJ = 0x%x\n", ptr_dev, dsj);

           status = amigo_status();
           if (Debug) printf("%s STATUS = 0x%x\n", ptr_dev, status);
           goto again;
   }
}

/* output error message and conditionally abort */
fatal_err(message, fname, flag)
char *message;
char *fname;
{
   fprintf(stderr, "%s: Error - %s of %s ", procnam, message, fname);
   if   (errno) perror("");
```

```
    else fprintf(stderr, "\n");

    if (flag == F_RTRN) return;
    if (flag == F_EXIT) exit(2);
    exit(3);
}
```

Index