HEWLETT PACKARD

HP 9000
Computers

Device I/O: User's Guide

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Device I/O:
User's Guide

HP 9000 Computers

HEWLETT
PACKARD

HP Part No. B1864-90002
Printed in USA January 1991

First Edition
E0191

Notices

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

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

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

Printing History

New editions of this manual will incorporate all material updated since the previous edition.

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.

January, 1991 ... Edition 1. This Edition documents material related to device I/O relevant to the 8.X release of HP-UX.

- For DIL, it supersedes the "Device I/O Library" part of manual part number 97089-90057, Edition 1, dated September 1989. New information was added regarding the Centronics-compatible Parallel interface.
- For HP-HIL, it supersedes the "Using HP-HIL Devices" part of manual part number 97089-90081, Edition 3, dated September 1987. Information regarding use of the Sound Generator (beeper) was added to the HP-HIL part.

Contents

Appendixes

Figures

Tables

1

Introduction to Device I/O

What is Device I/O?

For purposes of this User's Guide, device I/O involves access to arbitrary input/output devices from HP-UX. This access may be through one of the following interfaces:

- Hewlett-Packard Interface Bus (HP-IB)
- General-Purpose Input/Output (GPIO)
- Hewlett-Packard Human Interface Link (HP-HIL)

What is DIL?

DIL is the Device I/O Library. This is a library of subroutines used for interfacing with I/O devices. The DIL part of this User's Guide not only discusses interfacing strategies using the HP-IB and GPIO interfaces, but also strategies for general purpose I/O programming using DIL routines.

What is HP-HIL?

HP-HIL is the Hewlett-Packard Human Interface Link. The HP-HIL part of this User's Guide discusses communication using the HP-HIL interface, other functions provided by the HP-HIL peripheral processor, and describes a few of the HP-UX supported HP-HIL devices.

Choosing DIL or HP-HIL

If you want to interface with devices using GPIO, HP-IB or other protocols that would need DIL routines, use the DIL part of this manual. You cannot, however, access HP-HIL devices using DIL.

If you want to access HP-HIL devices, use the HP-HIL part of this User's Guide. You can access only HP-HIL devices with the HP-HIL interface.

Part I
DIL

The Device I/O Library

- Interfacing Concepts
- General-Purpose Routines
- Controlling the HP-IB Interface
- Controlling the GPIO Interface
- Controlling the Parallel Interface

2

Interfacing Concepts

This tutorial explains how to access arbitrary I/O devices from HP-UX through **HP-IB** (Hewlett-Packard Interface Bus), **GPIO** (General-Purpose I/O), and Centronics-compatible **Parallel** 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, GPIO, and Parallel interfaces.

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

Variation Between Computer Systems

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

Additional major differences related to a specific model or series are identified in a separate appendix for that series. Separate appendices are provided for Series 300/400 and 600/800.

Manual Organization

Chapter 2: Interfacing Concepts presents basic I/O programming concepts and a description of the HP-IB, GPIO, and Parallel interfaces.

Chapter 3: General-Purpose Routines discusses how to access interfaces from HP-UX environment and how to implement I/O transfers.

Chapter 4: Controlling the HP-IB Interface describes I/O programming techniques for the HP-IB interface.

Chapter 5: Controlling the GPIO Interface discusses I/O programming techniques for the GPIO interface.

Chapter 6: Controlling the Parallel Interface describes I/O programming techniques for the Centronics-compatible Parallel interface.

Appendix A: Series 300/400 Dependencies discusses hardware- and system-dependent characteristics of DIL subroutines when used with Series 300/400 computers. If you are using a Series 300/400 HP-UX system, check this appendix to ensure correct use of DIL subroutines.

Appendix B: Series 600/800 Dependencies is similar to other appendices, but for Series 600/800 computers. Use this appendix to ensure the correct use of DIL subroutines on Series 600/800 systems.

Appendix C: Character Codes

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

DIL Interfacing Subroutines

As mentioned previously, Device I/O Library (DIL) subroutines provide a means for directly accessing peripheral devices through HP-IB, GPIO, and/or Parallel 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, GPIO, or Parallel interfaces.

Linking DIL Routines

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

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

Similarly, for a Pascal program, use:

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

and for a FORTRAN program, use:

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

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

Calling DIL Routines from Pascal

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

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

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

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

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

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

Calling DIL Routines from FORTRAN

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

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

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

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

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

then write the onionskin routine:

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

General Interface Concepts

The remainder of this chapter discusses interfaces in general and the HP-IB, GPIO, and Centronics-compatible Parallel 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:

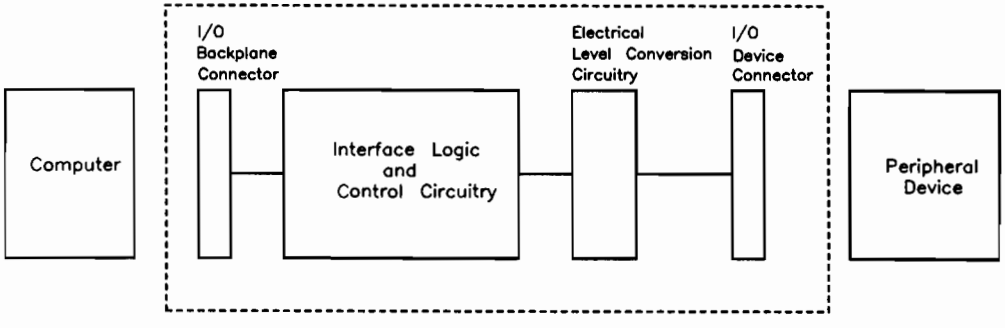


Figure 2-1. Interface Functional Diagram

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, GPIO, and Parallel 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 three particular interfaces: HP-IB, GPIO, and Parallel.

Handshake I/O

Most HP-IB, GPIO, and Parallel 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 signaling 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.

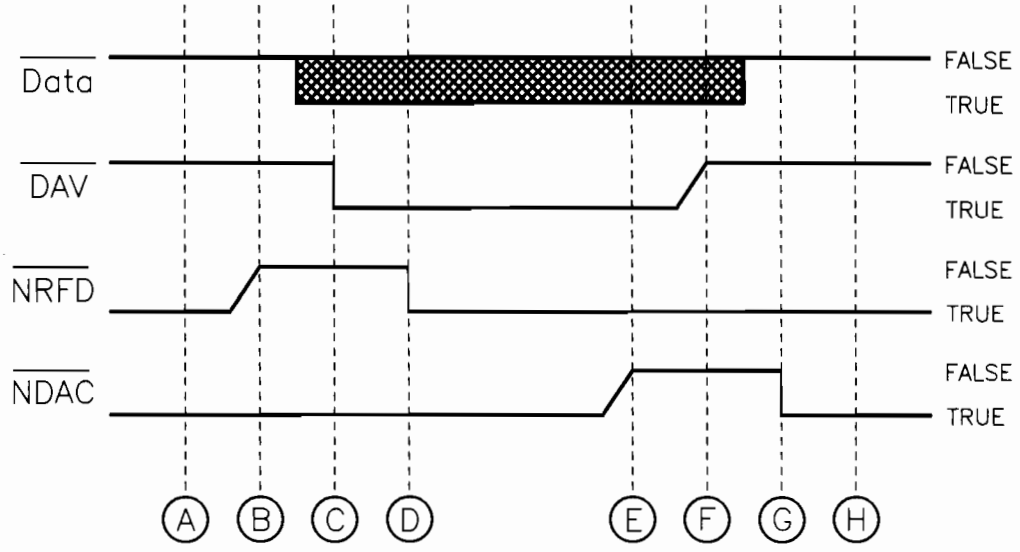


Figure 2-2. HP-IB Handshake Sequence

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

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

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

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

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

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

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

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

Bus Management Control Lines

There are five bus management control lines:

$\overline{\text{ATN}}$	ATtentioN: Treat data on data lines as commands, not data.
$\overline{\text{IFC}}$	InterFace Clear: Unconditionally terminate all current bus activity.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The GPIO Interface

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

While the GPIO interfaces used on various HP-UX computers are electrically very similar, they differ in certain important aspects. Refer to the appendices for Series 300/400 and 600/800 for information pertaining to your specific application.

The Centronics-Compatible Parallel Interface

The **Parallel** interface is a very flexible Centronics-compatible bi-directional interface that can be used to communicate with a variety of devices. The Parallel interface utilizes data, handshake, and special-purpose lines to perform data transfers by means of various user-selectable handshaking methods.

While the Parallel interfaces used on various HP-UX computers are electrically very similar, they differ in certain important aspects. Refer to the appendices for Series 300/400 and 600/800 for information pertaining to your specific application.

3

General-Purpose Routines

The DIL library contains several general-purpose subroutines that can be used with any interface supported by the library (see Table 3-1 for a complete list). This chapter explains how to use these subroutines in application programs. Specifically, the following topics are presented:

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

Background Basics

Interface Special Files

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

Entity Identifiers (eid)

Nearly all DIL routines require an **entity identifier** (`eid`) as a parameter. The entity identifier is an integer returned by the *open*(2) system call when opening the interface special file (`eid` is the file descriptor for the opened special file on Series 300/400 and 600/800). The `eid` supplied as a parameter to a DIL subroutine tells the subroutine which interface special file to use.

Programming Model

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

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

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

General-Purpose Routines

Table 3-1 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 3-1. General-Purpose Routines.

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

Additional Series 300/400 Routines

Series 300/400 systems also support the following DIL subroutines:

Subroutine	Description
`io_burst`	Control the data path between computer memory and an HP-IB, GPIO, or Parallel 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 HP-IB, GPIO, and Parallel I/O.
`io_dma_ctl`	Control usage of DMA channels by DIL devices.

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

Opening Interface Special Files

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

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

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

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

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

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

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

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

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

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

Closing Interface Special Files

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

Note HP-UX limits the number of files a given process (program) can have open at one time to `NO_FILE` as defined in the `/usr/include/param.h` header file. Series 300/400 systems limit the number of open DIL files in the entire system to the value of the configurable parameter `ndilbuffers` (default is 30). See the *HP-UX System Administrator Manual* for information on changing this value. Series 600/800 systems limit the number of open DIL files to 16 per interface.

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

```
#include <fcntl.h>
#include <errno.h>
main()
{
   int eid;
   .
   .
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1) {
        printf("open failed, errno = %d\n", errno);
        exit(2);
   }
   .
   .
        /* Code to perform I/O operations
                (read/write in this case) on the open interface. */
   close(eid);
}
```

Upon completion of the `close` system call, the entity identifier is no longer valid and is available for the system to assign to another file. If the file is again opened later in the program, the system may or may not assign the same `eid` value, so appropriate caution in using `eid` values is in order.

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

Low-Level Read/Write Operations

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

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

Calls to read have the form:

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

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

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

    .
    .  /*establish communication with the raw HP-IB device file
             (see the next chapter, "Controlling the HP-IB interface")*/

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

Calls to write are very similar:

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

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

   .
   .  /*establish communication with the HP-IB interface
            (see the next chapter, "Controlling the HP-IB Interface")*/

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

Although read and write require the number of bytes to be transferred as their third argument, other characteristics (discussed later) of the device

associated with the interface file `eid` can end the transfer before this number is reached.

Example

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

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

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

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

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

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

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

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

Designing Error Checking Routines

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

The errno Variable

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

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

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

Using errno

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

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

The errno.h Header File

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

Displaying errno

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

```
#include <errno.h>
#include <fcntl.h>
main()
```

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

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

Error Handlers

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

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

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

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

Resetting Interfaces

The DIL routine `io_reset` can be used to reset HP-IB, GPIO, and Centronics-compatible Parallel interfaces.

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

```
io_reset(eid);
```

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

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

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

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

Locking an Interface

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

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

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

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

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

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

Controlling I/O Parameters

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

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

Note

It is not uncommon for a single process to have multiple `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`.

An easy way to handle multiple devices that use different data formats without having to reconfigure each individual I/O operation is to open more than one `eid` on a given interface file, then configure each `eid` independently.

Setting I/O Timeout

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

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

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

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

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

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

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

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

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

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

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

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

Setting Data Path Width

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

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

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

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

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

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

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

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

Use of `io_width_ctl` to change interface data path width affects only the device associated with that particular `eid`. Use `io_reset` or `io_width_ctl` to restore the default 8-bit path width. On a Series 600/800 system, `io_width_ctl` affects all users of the interface referred to by `eid`.

Setting Minimum Data Transfer Rate

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

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

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

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

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

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

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

Setting the Read Termination Pattern

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

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

Input termination occurs when the first termination condition is recognized, independent of the type of condition. If two or more conditions occur simultaneously, the first condition detected terminates the operation. However, this first condition along with any other simultaneous events that would also have caused termination are recorded during clean-up at the end of the transfer for possible later use by `io_get_term_reason`.

Termination on Byte Count

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

Termination on Hardware Condition

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

Termination on Data Pattern

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

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

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

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

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

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

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

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

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

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

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

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

}
```

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

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

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

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

Disabling a Read Termination Pattern

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

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

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

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

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

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

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

Determining Why a Read Terminated

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

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

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

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

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

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

Example

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

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

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

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

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

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

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

Interrupts

DIL provides an interrupt mechanism for HP-IB, GPIO, and Parallel 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.

HP-IB Interrupts

Series 300/400 and 600/800 computers recognize the following HP-IB interrupt conditions:

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

GPIO Interrupts

- Series 300/400 computers recognize the following GPIO interrupt condition:

Signal	Condition
EIR	EIR line has been asserted.

- The Series 600/800 HP 27112 GPIO interface recognizes the following interrupt conditions:

Signal	Condition
SIE0	Status line 0 has been set.
SIE1	Status line 1 has been set.

- The Series 600/800 HP 27114 GPIO interface recognizes the following interrupt condition:

Signal	Condition
EIR	EIR line has been asserted.

Parallel Interrupts

Series 300/400 computers recognize the following Parallel interrupt conditions:

Signal	Condition
NERROR	NERROR line has changed from high to low or from low to high.
SELECT	SELECT line has changed from high to low or from low to high.
PERROR	PERROR line has changed from high to low or from low to high.

The io_on_interrupt Subroutine

The `io_on_interrupt` subroutine sets up interrupt conditions. It has the form:

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

where **eid** is the interface entity identifier for a GPIO, HP-IB or Parallel 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>
extern int service_routine();

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

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

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

The io_interrupt_ctl Subroutine

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);

}
```

4

Controlling the HP-IB Interface

The general-purpose subroutines discussed in Chapter 3 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 5 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.

Table 4-1 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 4-1. HP-IB Bus Commands

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

Table 4-1. HP-IB Bus Commands (continued)

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

UNLISTEN

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

UNTALK

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

DEVICE CLEAR

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

LOCAL LOCKOUT

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

SERIAL POLL ENABLE

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

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

SERIAL POLL DISABLE

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

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

TRIGGER (Group Execute Trigger)

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

SELECTED DEVICE CLEAR

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

GO TO LOCAL

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

PARALLEL POLL CONFIGURE

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

PARALLEL POLL ENABLE

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

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

PARALLEL POLL DISABLE

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

Overview of HP-IB DIL Routines

The 17 subroutines in Table 4-2, in addition to the general-purpose subroutines discussed in Chapter 3, provide full capabilities for controlling and using the HP-IB interface.

Table 4-2. HP-IB DIL Routines

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

HP-IB: The Computer's Role

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

Ground Rules

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

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

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

Available Subroutines versus Controller Role

Which DIL subroutines can be used depends on the computer's role on the HP-IB at the time. Given the three possible roles, Table 4-3 indicates which subroutines can be used with each.

Table 4-3. DIL Subroutine Availability Based on Interface Role

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

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

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

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

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

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

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

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

io_lock and io_unlock

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

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

io_burst

Series 300/400 systems support burst I/O (also called fast handshake) which bypasses the kernel by performing a high-speed non-interrupt transfer. This method can produce considerable performance improvement when handling short transfers to or from high-speed HP-IB devices. Refer to the *io_burst*(3I) manual entry in Section 3 of the *HP-UX Reference* for more information.

hpib_io

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

Opening the HP-IB Interface File

Before DIL subroutines can be used on an HP-IB interface, the interface special file must exist and the program must obtain a corresponding entity identifier. The procedures for opening interface special files and obtaining entity identifiers is discussed in Chapter 3, "General-Purpose Routines."

Sending HP-IB Commands

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Errors While Sending Commands

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

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

To determine which of these conditions caused the error, check the value of `errno`, an external integer variable used by HP-UX system calls. Error-checking routines are discussed at length in Chapter 3.

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

errno Value	Error Condition
EBADF	`eid` did not refer to an open file
ENOTTY	`eid` did not refer to a raw interface file
EIO	The interface was not the Active Controller (EACCES on Series 600/800)
ETIMEDOUT	A timeout occurred (EIO on Series 300/400)
EACCES	The interface associated with this `eid` was locked by another process and `O_NDELAY` was set for this `eid`
EINVAL	`number` was invalid, either less than or equal to 0 or greater than `MAX_HPIB_COMMANDS` as defined in `dvio.h`

Changing Parity on Commands

By default, bus commands sent across the bus using **hpib_send_cmnd** are sent using odd parity. On the Series 300/400, you can disable the use of parity on bus commands using the **hpib_parity_ctl** routine.

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

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

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

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

Active Controller Role

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

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

Determining Active Controller

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

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

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

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

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

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

Setting Up Talkers and Listeners

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

Series 300/400 and 600/800 computers provide two methods for addressing listeners and talkers on HP-IB: auto-addressing and command addressing.

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

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

Auto-Addressing

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

For example, suppose you are using a Series 300/400 computer with an HP 98624 HP-IB card on select code 08 to access a peripheral device located at bus address 03. Use **mknod** to create a new device file named **device** for the peripheral device and place the file in directory **dev** underneath the root directory as explained in Appendix A:

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

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

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

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

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

main()
{
   int eid;

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

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

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

Using hpib_send_cmnd

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

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

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

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

Calculating Talk and Listen Addresses

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

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

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

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

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

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

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

where `talk_address` is the decimal equivalent of the binary bit pattern that represents the ASCII talk address command character. Likewise, `listen_address` is the decimal representation of the ASCII listen address command character. `bus_address` is the decimal value of the HP-IB bus address for the device being addressed.

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

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

An Example Configuration

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

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

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

Remote Control of Devices

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

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

Locking Out Local Control

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

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

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

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

Enabling Local Control

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

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

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

Triggering Devices

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

The following code triggers the device at bus address 5:

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

Transferring Data

Data Output

To output data from an Active Controller the controller must:

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

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

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

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

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

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

Data Input

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

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

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

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

Clearing HP-IB Devices

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

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

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

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

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

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

Responding to Service Requests

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

Monitoring the SRQ Line

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

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

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

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

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

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

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

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

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

The most practical way to monitor SRQ is to set up an interrupt handler for that condition (see "Interrupts" section of Chapter 3).

Processing the Service Request

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

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

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

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

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

Thus, the Active Controller responds to SRQ by:

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

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

Parallel Polling

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

Configuring Parallel Poll Responses

HP-IB devices fall into three general categories:

1. Those devices that can be remotely programmed by the Active Controller to respond to a parallel poll in a certain way, The next several pages explain how to program these devices.
2. Devices whose parallel poll response is configured by internal hardware, whether by setting of configuration switches, or based on device bus address. A significant number of Hewlett-Packard products fall into this grouping. In general, they are HP-IB devices that support secondary commands such as SS/80 and CS/80 mass storage devices, CYPER printers, and Amigo protocol devices including several disc drives and printers. Some important information about these devices follows in the next few paragraphs.

3. Devices that are not capable of responding to parallel polls, so discussing their configuration is meaningless.

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

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

Note also that some models of HP-IB devices can be switched between normal HP-IB operating mode and "Amigo" or "Secondary" mode (terminology varies as well as the implementation). Refer to the device installation and operating manuals for more information about how to configure the device for your application and to determine whether the device supports remote configuration by the Active Controller, uses internal configuration, or does not support parallel poll.

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

Devices that can be remotely configured can be programmed to respond with a logic 0 or logic 1 level on any one of eight data lines. Thus there are 16 possible combinations of lines and logic levels since there are two possible levels on each line and only one line can be asserted during a parallel poll. The PARALLEL POLL ENABLE command consists of an 8-bit byte whose bits are arranged as in Table 4-4 (the decimal equivalent value of the byte falls in the range of 96 through 111).

Table 4-4. PARALLEL POLL ENABLE Bits

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

where:

- The upper four bits are a fixed pattern of logical 0 (bits D7 and D4) and logical 1 (bits D6 and D5).
- Bit D3 (response logic level) determines whether data line D3 is to be asserted (driven to its ground-true state) or released (allowed to float to its high-false state) by the device when responding to a parallel poll if service is needed. If bit D3 is set (1), the device responding to the poll drives the data line low if service is needed. If D3 is not set (0), the device responding to the poll drives the data line low if service is *not* needed (bit value = 0). This bit is most commonly set to a value of 1.
- Bits D2, D1, and D0 are the 3-bit (value range 0 through 7) value representing which data line (D0 through D7 respectively) is to be used when responding to a parallel poll.

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

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

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

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

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

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

Where:

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

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

Disabling Parallel Poll Responses

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

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

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

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

Conducting a Parallel Poll

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

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

`hpib_ppoll` is invoked as follows:

```
hpib_ppoll(eid);
```

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

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

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

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

Errors During Parallel Polls

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

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

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

errno Value	Error Condition
EBADF	`eid` does not refer to an open file.
ENOTTY	`eid` does not refer to a raw interface file.
EIO	Interface is not Active Controller. (EACCES on Series 600/800)
ETIMEDOUT	A timeout occurred. (EIO on Series 300/400)

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

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

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

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

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

Calculating the sense

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

The value for `sense` is determined as follows:

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

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

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

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

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

Example

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

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

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

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

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

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

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

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

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

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

Serial Polling

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

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

The Active Controller cannot poll itself.

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

Conducting a Serial Poll

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

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

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

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

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

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

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

Errors During Serial Poll

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

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

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

errno Value	Error Condition
EBADF	`eid` does not refer to an open file.
ENOTTY	`eid` does not refer to a raw interface file.
EIO	The interface was not the Active Controller. (EACCES on Series 600/800)
ETIMEDOUT	A timeout occurred. (EIO on Series 300/400)
EACCES	The interface associated with this `eid` was locked by another process and `O_NDELAY` was set for this `eid`.
EINVAL	Invalid bus address.

Passing Control

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

`hpib_pass_ctl` is called as follows:

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

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

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

What If Control Is Not Accepted?

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

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

Errors While Passing Control

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

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

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

errno Value	Error Condition
EBADF	`eid` does not refer to an open file.
ENOTTY	`eid` does not refer to a raw interface file.
EIO	Interface is not Active Controller.
EINVAL	Invalid bus address.
ETIMEDOUT	A timeout occurred (EIO on Series 300/400)
EACCES	The interface associated with this `eid` was locked by another process and `O_NDELAY` was set for this `eid`

Controlling the ATN Line

On a Series 300/400, the subroutine `hpib_atn_ctl` can be used to control the ATN line on the HP-IB bus. This routine is particularly useful when setting up two non-active controllers for a data transfer.

`hpib_atn_ctl` is called as follows:

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

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

Changing the Interface Bus Address

On a Series 300/400, the subroutine `hpib_address_ctl` can be used to programmatically change the bus address of an HP-IB interface card.

`hpib_address_ctl` is called as follows:

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

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

System Controller Role

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

Determining System Controller

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

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

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

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

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

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

System Controller's Duties

The HP-IB System Controller has three major functions:

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

hpib_abort

A call to `hpib_abort` performs the following actions:

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

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

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

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

`hpib_abort` is called as follows:

```
hpib_abort(eid);
```

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

hpib_ren_ctl

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

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

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

`hpib_ren_ctl` is called as follows:

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

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

Errors During hpib_abort and hpib_ren_ctl

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

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

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

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

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

errno Value	Error Condition
ETIMEDOUT	A timeout occurred (EIO on Series 300/400)
EACCES	The interface associated with this `eid` was locked by another process and `O_NDELAY` was set for this `eid`

The Computer As a Non-Active Controller

Checking Controller Status

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

The call to `hpib_bus_status` has the form:

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

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

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

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

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

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

Requesting Service

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

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

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

To request service, invoke `hpib_rqst_srvce` as follows:

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

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

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

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

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

A couple of notes of caution are in order here:

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

On the other hand, if you call `hpib_rqst_srvce` while you are Active Controller, the interface receives the service request sequence from the

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

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

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

Errors While Requesting Service

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

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

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

errno Value	Error Condition
EBADF	`eid` does not refer to an open file.
ENOTTY	`eid` does not refer to a raw interface file.
ETIMEDOUT	A timeout occurred. (EIO on Series 300/400)
EACCES	The interface associated with this `eid` was locked by another process and `O_NDELAY` was set for this `eid`.

Responding to Parallel Polls

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

To configure a parallel-poll response:

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

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

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

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

Calculating the Response

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

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

where:

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

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

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

Limitations of hpib_card_ppoll_resp

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

Error Conditions

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

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

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

errno Value	Error Condition
EBADF	`eid` does not refer to an open file.
ENOTTY	`eid` does not refer to a raw interface file.
ETIMEDOUT	A timeout occurred. (EIO on Series 300/400)
EACCES	The interface associated with this `eid` was locked by another process and `O_NDELAY` was set for this `eid`.
EINVAL	The device cannot respond on the line number specified by `response`.

hpib_ppoll_resp_ctl

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

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

Parallel poll response is set as follows:

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

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

Disabling Parallel-Poll Response

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

For example, the subroutine call:

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

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

Accepting Active Control

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

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

Call `hpib_status_wait` as follows:

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

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

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

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

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

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

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

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

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

Errors While Waiting on Status

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

- A timeout occurred before the condition the routine was waiting for became true.
- The value specified by `status` is undefined.
- Entity identifier `eid` does not refer to a raw HP-IB interface file.
- Entity identifier `eid` does not refer to an open file.
- The interface associated with this `eid` is locked by another process and `O_NDELAY` is set for this `eid`.
- The device is active controller and `status` specifies WAIT_FOR_TALKER or WAIT_FOR_LISTENER. (Series 300/400 only)

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

errno Value	Error Condition
EBADF	`eid` does not refer to an open file.
ENOTTY	`eid` does not refer to a raw HP-IB interface file.
EINVAL	`status` contains an invalid value.
ETIMEDOUT	The specified condition did not become true before a timeout occurred. (EIO on Series 300/400)
EACCES	The interface associated with this `eid` was locked by another process and `O_NDELAY` was set for this `eid`.
EIO	The device is active controller and `status` specifies WAIT_FOR_TALKER or WAIT_FOR_LISTENER (Series 300/400 only).

Determining When You Are Addressed

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

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

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

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

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

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

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

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

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

As with `hpib_bus_status`, a condition value of WAIT_FOR_TALKER causes the program to wait until the interface is addressed as a talker. With a condition value of WAIT_FOR_LISTENER the routine waits until it is addressed to listen. The maximum time that the routine can wait for the specified condition is controlled by the timeout value that was previously set for the entity identifier using subroutine `io_timeout_ctl` (discussed in Chapter 3). `hpib_status_wait` returns 0 if the wait condition terminated the wait or `-1` if a timeout or other error occurred before the wait condition was fulfilled.

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

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

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

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

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

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

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

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

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

Combining I/O Operations into a Single Subroutine Call

`hpib_io` is a high-level DIL subroutine that provides a mechanism for conveniently collecting a series of HP-IB I/O operations in a data structure then using a simple subroutine call to `hpib_io` to handle interface and bus management operations. This feature eliminates the need for using several long tedious series of subroutine calls to `io_lock`, `hpib_send_cmnd`, `read`, `write`, and `io_unlock` and makes these operations atomic on the Series 300/400.

A call to `hpib_io` has the form:

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

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

Iodetail: The I/O Operation Template

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

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

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

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

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

```
iovec->component
```

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

The Mode Component

The `mode` describes what type of I/O operation is to be performed on the data pointed to by the `buf` component. To determine its value, *OR* appropriate constants from a set defined in the include file `dvio.h`. You can choose from the constants in Table 4-5:

Table 4-5. Constants for Constructing `mode`

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

Note

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

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

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

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

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

The Terminator Component

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

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

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

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

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

The Count Component

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

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

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

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

The Buf Component

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

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

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

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

Allocating Space

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

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

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

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

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

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

Example

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Locating Errors in Buffered I/O Operations

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

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

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

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

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

5

Controlling the GPIO Interface

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

Interface Configuration

The Series 300/400 GPIO interface is configured by setting several switches on the interface card. The interface installation manual explains how each switch is used and how it should be configured. Configurable functions associated with these switches include:

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

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

Creating the GPIO Interface File

After setting the necessary switches on your GPIO interface, install the card in the computer then create an interface file for it as explained in Chapter 3. An appropriate interface file must be created before the interface can be accessed from HP-UX.

Interface Control Limitations

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

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

Using DIL Subroutines

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

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

General-purpose routines are listed and described in detail in Chapter 3. They are used in this chapter to illustrate various aspects of controlling GPIO interfaces from an HP-UX process.

Two DIL routines used exclusively with GPIO interfaces:

- `gpio_get_status`
- `gpio_set_ctl`.

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

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

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

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

Resetting the Interface

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

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

This has the following effect:

- Peripheral Reset line (PRESET) is pulsed low,
- PCTL line is placed in the clear state,
- If the DOUT CLEAR jumper is installed, the Data Out lines are all cleared (set to logical 0),
- Interrupts from the controlled interface are disabled on Series 300/400 systems.

Lines that are left unchanged are:

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

Performing Data Transfers

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

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

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

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

Using Status and Control Lines

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

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

The call to `gpio_set_ctl` has the following form:

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

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

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

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

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

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

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

Reading STI0 and STI1

The call to `gpio_get_status` has the following form:

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

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

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

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

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

To illustrate:

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

Controlling Data Path Width

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

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

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

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

```
int eid;

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

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

Controlling Transfer Speed

You can request a minimum speed for the data transfer across a GPIO interface by issuing a call to `io_speed_ctl`. Your system rounds the specified speed up to the nearest defined speed. If you specify a speed that is faster than your system allows, the highest available speed is used instead. Refer to Chapter 3 for more information about `io_speed_ctl`.

GPIO Timeouts

If a non-zero timeout limit has been established for a given *eid* and that limit is exceeded during a data transfer request, an error condition results. When the subroutine handling the transfer detects the timeout error, it returns -1 and sets *errno* to ETIMEDOUT (EIO on Series 300/400). When a timeout error occurs, use `io_reset` to reset the GPIO interface before attempting another transfer.

Burst Transfers

Series 300/400 systems support high-speed burst I/O on HP-IB and GPIO interfaces. The call to `io_burst` is structured as follows:

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

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

Read Terminations

Determining Why a Read Operation Terminated

Subroutine `io_get_term_reason`, described in Chapter 3, is used to determine why the last read performed on a particular *eid* terminated. Possible reasons include:

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

Specifying a Read Termination Pattern

Chapter 3 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 3. They are used to set up and control interrupt handlers for the GPIO status line or for a particular GPIO interface file *eid*.

6

Controlling the Parallel Interface

This chapter discusses the capabilities and limitations of DIL subroutines when controlling the Parallel interface.

Interface Control Limitations

Device I/O Library (DIL) subroutines provide a means for using a Centronics-compatible Parallel 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 limitation that there is no direct access to interface handshake lines: STROBE line, BUSY line, and ACK line. These handshake lines are controlled by the interface Input/Output FIFO hardware.

Using DIL Subroutines

Several DIL subroutines can be used to control the Parallel interface. They are the general-purpose routines usable with HP-IB, GPIO, and Parallel interfaces, which are listed and described in detail in Chapter 3. They are used in this chapter to illustrate various aspects of controlling Parallel interfaces from an HP-UX process.

By using the DIL general-purpose routines you can:

- Reset the interface.
- Perform data transfers.
- Control the data handshake mode.
- Set a timeout for data transfers.
- Set a read termination character.
- Get the termination reason.
- Set up the interrupts.
- Enable or disable interrupts.

Resetting the Interface

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

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

This has the following effect:

- The NINIT signal is held low for 50 microseconds.
- The interface is reset to its power-on state.
- User interrupts enabled via `io_on_interrupt` are enabled (unmasked).

Performing Data Transfers

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

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

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

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

Controlling Transfer Speed

You can request a minimum speed for the data transfer across a Parallel 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 3 for more information about `io_speed_ctl`.

Timeouts

If a non-zero timeout limit has been established for a given *eid* and that limit is exceeded during a data transfer request, an error condition results. When the subroutine handling the transfer detects the timeout error, it returns -1 and sets *errno* to ETIMEDOUT (EIO on Series 300/400). When a timeout error occurs, use `io_reset` to reset the Parallel interface before attempting another transfer.

Burst Transfers

Series 300/400 systems support high-speed burst I/O on HP-IB, GPIO, and Parallel interfaces. The call to `io_burst` is structured as follows:

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

`io_burst` controls the data path between computer memory and the HP-IB, GPIO or Parallel 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 3, 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.
- An assertion of the NACK line was seen.
- Some abnormal condition occurred, such as an I/O timeout.

Specifying a Read Termination Pattern

Chapter 3 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 Parallel interface file *eid*.

Interrupts

Subroutines `io_on_interrupt` and `io_interrupt_ctl` are described in Chapter 3. They are used to set up and control interrupt handlers for a particular Parallel interface file *eid*.

Index

E

F

G

H

I

L

N

Part II
HP-HIL

The Hewlett-Packard Human Interface Link

- The Interface to HP-HIL Devices
- Typical HP-HIL Devices
- Using HP-HIL Devices
- HP-HIL Commands
- Keycode Set 1

7

Using HP-HIL Devices with HP-UX

The Interface to HP-HIL Devices

This part of the User's Guide describes communication via the Hewlett-Packard Human Interface Link (HP-HIL), and other functions provided by the HP-HIL System Device Controller (8042). It is primarily a description of the enhancements added to handle the HP-HIL interface. This interface is capable of supporting up to seven peripherals, such as graphics input, system ID Modules, and other devices generally related to human input, as well as the system keyboard.

Before launching into a discussion of the workings of the HP-HIL interface, a general overview should be presented. Figure 1-1 illustrates the basic components.

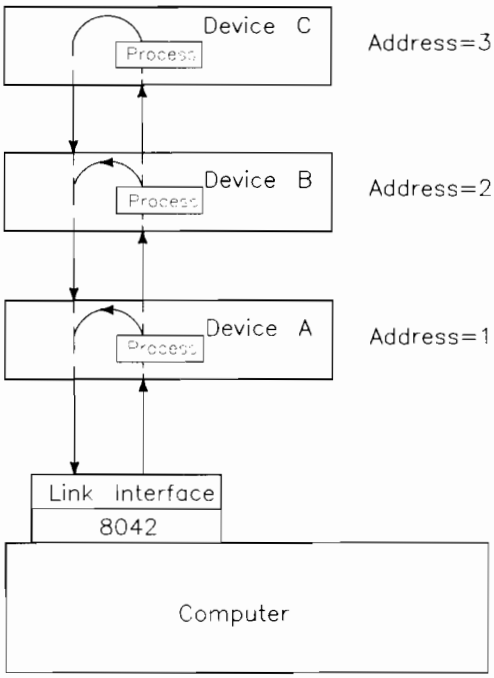


Figure 7-1.
Hewlett-Packard Human Interface Link

HP-HIL initialization takes place in the following manner. The peripheral devices in the HP-HIL link are powered up when the computer is turned on. Next, because of the "loop-back mode" each device is recognized by the computer as the "last" device in the link. The "loop-back mode" is where the computer sends out a signal and the HP-HIL device sends that same signal back to the computer through the return side of the device. Each device in the link is checked in this manner until there are no additional devices to check in the link. Note that the computer *does not* know the type of each device in the link; it merely knows that there is a device at that location in the link.

To further explain the HP-HIL initialization process, the following example is given. Assume the 8042 has sent out a signal looking for the first device on the link. (The 8042 is the HP-HIL system device controller, *not* the MC68000/10/20 microprocessor.) In our previously shown diagram, it would find Device A. Being the first device on the link, it's address is considered to be

1. The 8042 then instructs Device A to *exit* loop-back mode; that is, send the signals through to a possible next device. The 8042 then attempts to contact the second device on the link. Device B responds and is assigned address 2. The 8042 now knows that there are at least two devices on the link. The 8042 commands Device B to exit loop-back mode, and attempts to contact the next device. Successful, the 8042 now knows about Device C. As our diagram illustrates, Device C is the last device on the link, so the process proceeds differently at this point.

The 8042 instructs Device C to exit loop-back mode, and attempts to contact (nonexistent) Device D. Since it is not there, a timeout occurs, and the 8042 deduces that Device C is the last device on the link. Therefore, it instructs Device C to once again enter loop-back mode, and the link is configured.

The link can deal with a maximum of seven devices at any one time (see the NOTE in the section, "Typical HP-HIL Devices"). If there are eight or more devices physically connected, the devices after number seven are not found.

As the above discussion indicates, the address of a particular device is merely its sequential order of placement along the link. In the above diagram, Device A has address 1, B has address 2, and C has address 3. This is only a result of their physical order of connection. If Device C had been connected between Devices A and B, Device A would still have been address 1, but Device C would be address 2, and B would be address 3. The type of device is irrelevant to the address assigned to it.

After the link is operational, and during subsequent link operations, each device looks at the data being sent down the link. If a device notices that the destination address associated with the link data is the same as that device's address, that device receives and acts on the data. Otherwise, the data is merely shuttled along to the next device.

Typical HP-HIL Devices

This section provides a brief description of a few of the HP-UX supported HP-HIL devices. You can make use of these devices by writing special programs in C Language, FORTRAN, or Pascal to control them.

Before you can use an HP-HIL device, your terminal or computer must meet the following requirements:

- It should have a built-in HP-HIL interface or HP-HIL interface card present. This is the case for the HP Models 217, 237, 310, 318, 320, 330, and 350, Integral Personal Computer, and Model 550 with an HP 98700H Graphics Display Station. The HP 2393 and HP 2397 terminals also have built-in HP-HIL interfaces.
- If you are using a Model 220, it should have an HP 9920 Option 535 (HP 09920-66535, HP-HIL Keyboard/HP-IB Interface) card inserted in its backplane.

The following is a list of HP-HIL devices supported by the HP-UX system. It also provides the maximum current which each device uses.

Note

The total current your HP-HIL link can use before it stops working is 750 milliamps. This current limit is true for all HP-UX computers except the Integral PC and HP 98700A/H which have a total current limit of 520 milliamps.

When determining the total current used by your HP-HIL link, you should note that the current limits listed in this section are maximum current limits. The typical current used by each HP-HIL device is approximately two-thirds of this value, so when calculating the total current used by your HP-HIL link you need to take two-thirds of the sum of the maximum current values.

To determine the total current which your HP-HIL link draws when connected to the HP-HIL interface, add up the maximum current used by each device in the link and multiply the result by two-thirds. Again use the following list to determine the maximum current each device uses.

- HP-HIL/Touchscreen model HP 35723A—This is a screen bezel which is placed over the bezel of the HP 35731 (medium resolution black and white monitor) and HP 35741 (medium resolution color monitor) 12-inch video monitors. It can be programmed to select various functions by simply touching the screen. The maximum current this device uses is 200 milliamps.
- HP-HIL Keyboard model HP 46021A—This keyboard has alphabetic and numeric keys similar to those on a typewriter. Note that this keyboard replaces the HP 46020A keyboard and that all of the keys in each key group of the HP 46021A function the same as those of the HP 46020A. The key groups you will find on this keyboard are as follows:
  - Character Entry Group—allows alphabetic and numeric characters, as well as mathematical and commercial signs to be entered. It also contains data control keys such as [Back space] and [Return].
  - Numeric Group—provides for rapid entry of numeric data.
  - Display Control Group—controls the location of the cursor on the display.
  - Edit Group—allows data to be inserted in and deleted from the display.
  - Function Key Group—provides you with system defined function key labels, as well as with user defined function key labels.
  - System Control Group—controls system functions related to display operations, such as using the [Stop] key to suspend the display.

  The maximum current this device uses is 70 milliamps.
- HP Mouse model HP 46060A or HP 46060B—The mouse is a relative graphics input device for some graphics programs. It is commonly used to move the cursor to any position on the CRT (display) without using arrow keys or a Rotary Control Knob model HP 46083A. The HP 46060A is a two-button mouse, the HP 46060B is a three-button mouse. The maximum current use for the HP 46060A is 200 milliamps; 120 milliamps for the HP 46060B.

Note The HP 46060A Two-Button Mouse or the HP 46060B Three-Button Mouse must be the *last* device on the HP-HIL link.

- Extension Module model HP 46080A—The Extension Module allows you to increase the distance between HP-HIL devices by eight feet. Note that the HP-HIL link is capable of handling seven addresses and that the Extension Module *does not* occupy one of these addresses. The maximum current this device uses is 25 milliamps.
- HP-HIL/Audio Extension model HP 46081A —The Audio Extension allows a separation of 2.4 meters (8 feet) between the host computer and another HP-HIL device. This device also contains a speaker. Note that the HP-HIL link is capable of handling seven addresses and that the Audio Extension *does not* occupy one of these address. The maximum current that this device uses is 25 milliamps.
- HP-HIL/Audio Remote Extension model HP 46082A/B The HP 46082A allows a separation of 15 meters (49.2 feet) between the host computer and the graphics display station, and the HP 46082B allows a separation of 30 meters (98.4 feet). The Audio Remote Extension also contains a speaker. Note that the HP-HIL link is capable of handling seven addresses and that the Audio Remote Extension module *does not* occupy one of these address. The maximum current that each of these devices use is 50 milliamps.
- Rotary Control Knob model HP 46083A- This device provides the additional feature of a rotary control knob to your system. Note that a switch is provided which toggles the knob from X-axis data to Y-axis data. The maximum current this device uses is 110 milliamps.
- HP-HIL ID Module model HP 46084A- The HP 46084A Module is an HP-HIL device that returns an identification number for identifying you as the computer user. The identification number is unique to your particular ID Module. This allows application programs to use the ID Module to control access to program functions, data bases, and networks.

Note The identification number is the product/exchange and serial numbers returned in a packed format as explained in the section entitled. "Report Security Code."

The maximum current this device uses is 60 milliamps.

- Control Dials model HP 46085A -This module provides nine user-definable knobs. These knobs can be software defined to provide zooming, panning.

rotation, horizontal and vertical motion, color translation, and menu control when using graphics. Each knob on the HP 46085A can be defined in software to do functions other than those mentioned.

Note The HP 46085A Module occupies 3 addresses on the link. Each horizontal row corresponds to one address (bottom to top).

The maximum current this device uses is 320 milliamps.

- Function Box model HP 46086A—This module provides 32 function keys to select software-defined functions. A status LED, which is controlled by software, provides an indication of when the device is sending or receiving data. This device uses a non-standard keycode set (Keycode Set 2) which is shown in Figure 7-2. The maximum current this device uses is 80 milliamps.

	0/1	2/3	4/5	6/7	
8/9	10/11	12/13	14/15	16/17	18/19
20/21	22/23	24/25	26/27	28/29	30/31
32/33	34/35	36/37	38/39	40/41	42/43
44/45	46/47	48/49	50/51	52/53	54/55
	56/57	58/59	60/61	62/63	

Figure 7-2. Keycode Set 2

- A-size Digitizer model HP 46087A—The A-size Digitizer allows data entry from an ISO A4 or ANSI A-size drawing, or free-hand graphics input. It uses either a pen-like stylus or an optional Four-Button Cursor model HP 46089A . The maximum current this device uses is 200 milliamps.
- B-size Digitizer model HP 46088A—The B-size Digitizer allows data entry from an ISO A3 or ANSI B-size drawing, or free-hand graphics input. It uses

either a pen-like stylus or an the optional HP 46089A Four-Button Cursor. The maximum current this device uses is 200 milliamps.

- Four-Button Cursor model HP 46089A—This device is a four switch puck that may be used on the A or B-size Digitizer in place of the stylus. This device *does not* take an address space, and it *does not* use any additional current.
- HP-HIL/Quadrature Port model HP 46094A—This device allows interfacing an off-the-shelf 3 Button Mouse, Trackball (or any other device which provides an output in quadrature) to the HP-HIL link. The maximum current this device uses is 200 milliamps.
- Keyboard model HP 98203C—This keyboard functions the same as the HP 98203B keyboard. For information on the HP 98203B keyboard read the section entitled, "The Series 200/300 ITE as System Console" in this manual. The maximum current this device uses is 90 milliamps.
- Bar-Code Reader model HP 92916A—This device reads all standard bar-codes using a wand as the input mechanism. It provides an effective and reliable alternative to the keyboard for data entry. Note that HP-UX supports this device in the keyboard mode, in which the input from the device looks like keycodes. The keycodes, which can be read by the Bar-Code Reader, are: 3 of 9, Interleaved 2 out of 5, UPC/EAN, and Codabars USD-4 and ABC. The maximum current this device uses is 200 milliamps.

For more information on these devices, call your local HP Sales or Service Representatives.

Using HP-HIL Devices

This section gives a procedure for creating special (device) files for your HP-HIL devices, provides programs for identifying HP-HIL devices, and includes tables for interpreting data for the sample programs. This section also includes a discussion of the commands (opcodes) used in the macros located in the file `/usr/include/sys/hilioctl.h`.

Note HP-HIL devices can be added to or removed from the HP-HIL link without affecting the HP-UX operating system while it is running. However, if you are running an application which requires the use of that particular device and you:

- remove the device from the link, or
- open the link to the device, or
- open the link to add a new device

your application might not recognize the change and as a result it will not work as expected. An HP-HIL device can be added anywhere in the HP-HIL link provided it is not a non-extendible device (e.g. HP 46060A, HP Mouse), in this case the device can only be added to the end of the link.

A Few Terms

The following terms will be used throughout this part of the User's Guide:

- A **special (device) file** is a file associated with an I/O device. Special (device) files are read and written just like "ordinary files" (a type of HP-UX file containing either a program, text or data), but requests to read or write result in activation of a driver of the associated device. Entries for each file normally reside in the `/dev` directory. In this documentation, you will find that a special (device) file is referred to as a device file or a special file.
- A **macro** is a command which contains a set of instructions to be performed. The term macro was derived from the word macroinstruction.

- A **frame** is the way information travels through the HP-HIL link. It consists of 15 bits of information which include: start (1 bit), stop (1 bit), command (1 bit), parity (1 bit), address (3 bits), and data (8 bits). The frame is transmitted around the link at the rate of 10 micro-seconds per bit, or 150 micro-seconds per frame.

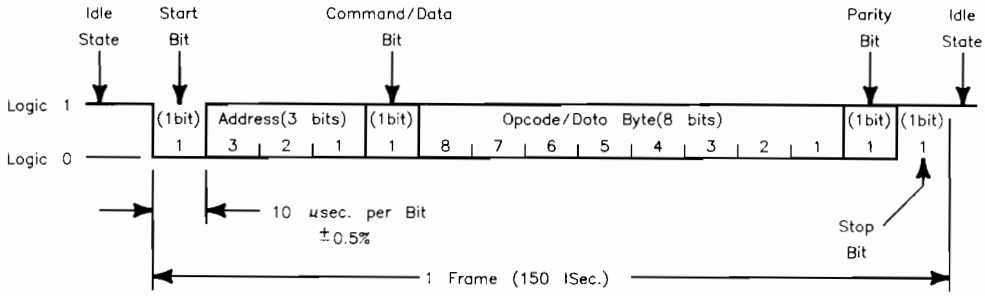


Figure 7-3. Frame

- A **command**(opcode) in this documentation is an operational code used in a lower level programming language (assembly) to perform an operation, such as incrementation, inversion or multiplication, on one or more operands. This is the definition for the term opcode; however, in this manual it will be used as the definition for the term command.
- A **path name** is a sequence of directories and "ordinary files" (HP-UX files containing either programs, text or data) separated by /'s which map out a path leading to a destination file.
- A **select code** is part of an address used for devices; a number determined by a setting on an interface card to which a peripheral device is connected. Multiple peripherals connected to the same interface card share the same select code.

Creating a Special Device File for HP-HIL Devices

Each device on the HP-HIL link has a unique address based on its position in the link (e.g. the first addressable HP-HIL device is address 1 and so on). There may be up to seven devices connected to a single HP-HIL driver board allowing device file names in the form `hil1`, `hil2`, ... , `hil7`. To access a device, you must first create a special (device) file using the `mknod` command.

For the Series 300

The Series 300 `mknod` parameters should create a character device with a major (driver) number of 24 and a minor (select code) number of 0x0000*a*0 where *a* is the device's one digit address i.e. its position on the HP-HIL loop from the computer interface card.

The format for using this command is:

```
/etc/mknod /dev/hila c 24 0x0000a0
```

where:

`hil_`	is the name you give to identify the HP-HIL address for which you are creating a device file.
`c`	specifies the character mode rather than the block mode.
`24`	is the major (driver) number used with the device you are creating. Series 300, 700 and 800 computers use major (driver) number 24 for communicating with HP-HIL devices.
`0x0000`	is the Series 300 select code of the device.
`a`	is the HP-HIL address of the device to which you wish to talk. The address ranges from one to seven. An addressable device is *not* an extension device, such as the HP 46080A (Extension Module) and the HP 46081A (HP-HIL/Audio Extension).

For the Series 700

The Series 700 `mknod` parameters should create a character device that is similar to that of the Series 300. The difference is the minor (select code) number of 0x2030*a*0 where *a* is the device's one digit address; again, its position on the HP-HIL loop from the computer interface card.

7

The format for the Series 700 command is:

```
/etc/mknod /dev/hila c 24 0x2030a0
```

For the Series 800

The Series 800 `mknod` parameters should create a character device that is similar to that of the Series 300 and 700. The difference is the minor (select code) number of 0x00*lua*0 where *lu* is the two-digit hardware logical unit and *a* is the device's one digit address.

The format for the Series 800 command is:

```
/etc/mknod /dev/hila c 24 0x00lua0
```

or

```
/etc/mknod /dev/hillu.a c 24 0x00lua0
```

You may need to create a special (device) file for the 8042 driver so you can talk to the timer, talk to the beeper, or change the keyboard repeat rate. To do this, change the device address (24) to 23 and use the appropriate form of the `mknod` command:

For Series 300

```
/etc/mknod /dev/hila c 23 0x000a0
```

Using the Sound Generator

This section describes how to implement and control the Sound Generator associated with the HP-HIL System Device Controller (8042). The Sound Generator ("beeper") can produce tone outputs in varying frequencies, duration and loudness. A sample program is included in this section to help you create various sounds that can be used as user prompts, warnings or other audio signals.

Sample Beeper Program

The following sample program, written in C Language, provides the structure for controlling the Sound Generator. You may alter this program to fit your needs. Remember to type in the program exactly as it appears (omit line numbers as they are *not* part of the program) and compile it.

```
1   #include <fcntl.h>
2   #include <sys/beeper.h>
3   main(argc, argv)
4   int argc;
5   char *argv[];
6   {
7       int fd;                          /*freq dura vol*/
8       static struct beep_info ring = {440, 500, 100};
9       fd=open("/dev/rhil",O_RDWR);
10      ioctl(fd, DOBEEP, &ring );
11  }
```

Lines 1 and 2 identify the include files for the macros that execute specific functions required by this program. The macro `fcntl.h` provides the O_RDWR File Access Mode and `beeper.h` sends data to the beeper.

Lines 3, 4 and 5 declare the variables (argc, argv and fd) to be used in the program.

Line 8 is where you may insert the three variables that control frequency duration and volume of your tone. Values entered here will produce a tone equal to A above middle C for 500 milliseconds at maximum volume.

Frequency, Duration and Volume of Tones

The three parameters asked for in the above C program line:

```
static struct beep_info ring = {aaa, bbb, ccc};
```

control the tone frequency or pitch (aaa), duration (bbb) and volume (ccc). By altering these values in your program you can control the Sound Generator.

To Set Frequency. Enter the frequency value, in Hertz, of the tone desired into the above program. The typical range is from 40 to 5 208 Hz depending on your computer hardware. For a baseline, 440Hz translates to A above Middle C on the musical scale (see sample program).

To Set Duration. Enter the tone's duration value, in milliseconds, into the above program. Tone durations can range from 2 550 (2.55 seconds) down to 1 (0.001 second) and 0 (Off).

To Set Volume. Enter the volume value, from 0 to 100, into the above program. Values can range from 1 (softest) through 100 (loudest) and 0 (Off).

Note Frequency, duration and volume of tone are subject to the resolution of the Sound Generator. The frequency specified is rounded to the nearest frequency achievable by the hardware. Some versions of hardware are limited as to frequency, duration and volume ranges.

Additional Considerations

When you execute a long listing of the `/dev` file, you will find the device files for HP-HIL devices on Series 300 computers are as follows:

```
crw-rw-rw-   1 root     root      24 0x000010 Oct 29 09:02 hil1
crw-rw-rw-   2 root     root      24 0x000020 May 22  1985 hil2
crw-rw-rw-   1 root     root      24 0x000030 May 22  1985 hil3
crw-rw-rw-   1 root     root      24 0x000040 May 22  1985 hil4
crw-rw-rw-   1 root     root      24 0x000050 May 22  1985 hil5
crw-rw-rw-   1 root     root      24 0x000060 May 22  1985 hil6
crw-rw-rw-   1 root     root      24 0x000070 May 22  1985 hil7
crw-rw-rw-   1 root     root      25 0x000080 May 22  1985 hilkbd
```

Note that the last device file (`hilkbd`) listed has a different major (driver) number. This is the device file for the HP-HIL "cooked" keyboard driver. The HP-HIL "cooked" keyboard driver does not require a new keyboard; it simply provides a protocol conversion for using your present HP-HIL keyboard. This protocol only recognizes the down stroke of a key when it is pressed (not both up and down keystrokes). Using this protocol conversion with programs that trap individual keystrokes (by reading from the HP-HIL interface) makes the application programs more compact, because they are keeping track of fewer keystrokes.

Note that the keyboard sends a set of data which consist of a four byte time stamp, one byte that contains status information as follows:

- 1000xxxx—both Shift and CTRL have been pressed,
- 1001xxxx—only CTRL has been pressed,
- 1010xxxx—only Shift has been pressed,
- 1011xxxx—neither Shift nor CTRL have been pressed,

and a final byte that contains a keycode taken from the table below. The following is a table of the keycodes for the HP-HIL "cooked" keyboard driver.

Table 7-1. Keycodes for the HP-HIL "Cooked" Keyboard Driver

Keycodes in		Key Label	
hex	decimal	Unshifted	Shift
00	0	Unused	
01	1	-	.
02	2	[	]
03	3	ESC	DEL
04	4	Unused	
05	5	Break	Reset
06	6	Stop	
07	7	Select	
08	8	Enter	
09	9	Tab	
0A	10	(Blank 1)	
0B	11	(Blank 2)	
0C	12	(Blank 3)	
0D	13	(Blank 4)	
0E	14	&home;	
0F	15	Prev	
10	16	Next	
11	17	Enter	Print
12	18	Extend char (left)	
13	19	Extend char (right)	
14	20	System	User
15	21	Menu	
16	22	Clear line	
17	23	Clear display	
18	24	Caps	

Continued on next page ...

Note that decimal keycodes 8 - 13 are numeric keypad keys

Table 7-1.
Keycodes for the HP-HIL "Cooked" Keyboard Driver (continued)

Keycodes in		Key Label	
hex	decimal	Unshifted	Shift
19	25	Tab	
1A	26	Unused	
1B	27	f1	
1C	28	f2	
1D	29	f5	
1E	30	f6	
1F	31	f7	
20	32	f3	
21	33	f4	
22	34	▼	
23	35	▲	
24	36	f8	
25	37	Unused	
26	38	◀	
27	39	▶	
28	40	Insert line	
29	41	Delete line	
2A	42	Unused	
2B	43	Insert char	
2C	44	Delete char	
2D	45	Unused	
2E	46	Back space	
2F	47	Unused	
30	48	Unused	
31	49	Unused	
32	50	Unused	

Continued on next page ...

Table 7-1.
Keycodes for the HP-HIL "Cooked" Keyboard Driver (continued)

Keycodes in		Key Label	
hex	decimal	Unshifted	Shift
33	51	Unused	
34	52	Unused	
35	53	Unused	
36	54	Unused	
37	55	Unused	
38	56	Unused	
39	57	Enter	
3A	58	Unused	
3B	59	Unused	
3C	60	0	
3D	61	▢	
3E	62	.	
3F	63	+	
40	64	1	
41	65	2	
42	66	3	
43	67	-	
44	68	4	
45	69	5	
46	70	6	
47	71	*	
48	72	7	
49	73	8	
4A	74	9	
4B	75	/	

Continued on next page ...

Note that decimal keycodes 60 - 75 are numeric keypad keys

Table 7-1.
Keycodes for the HP-HIL "Cooked" Keyboard Driver (continued)

Keycodes in		Key Label	
hex	decimal	Unshifted	Shift
4C	76	Unused	
4D	77	Unused	
4E	78	Unused	
4F	79	Unused	
50	80	1	!
51	81	2	@
52	82	3	#
53	83	4	$
54	84	5	%
55	85	6	^
56	86	7	&
57	87	8	*
58	88	9	(
59	89	0	)
5A	90	-	_
5B	91	=	+
5C	92	[	{
5D	93	\ ]	}
5E	94	;	:
5F	95	'	"
60	96	,	<
61	97	.	>
62	98	/	?
63	99	(Space)	
64	100	`	

Continued on next page ...

Table 7-1.
Keycodes for the HP-HIL "Cooked" Keyboard Driver (continued)

Keycodes in		Key Label	
hex	decimal	Unshifted	Shift
65	101	P	
66	102	K	
67	103	L	
68	104	Q	
69	105	W	
6A	106	E	
6B	107	R	
6C	108	T	
6D	109	Y	
6E	110	U	
6F	111	I	
70	112	A	
71	113	S	
72	114	D	
73	115	F	
74	116	G	
75	117	H	
76	118	J	
77	119	M	
78	120	Z	
79	121	X	
7A	122	C	
7B	123	V	
7C	124	B	
7D	125	N	
7E	126	Unused	
7F	127	Unused	

If, for some reason, you need to connect more than one Keyboard to an HP-HIL link, you must keep the following facts in mind:

- When you have multiple Keyboards in the HP-HIL link all operating in the "cooked" mode, key presses from these keyboards are sequentially merged together.
- When you have multiple Keyboards in the HP-HIL link all operating in the "raw" mode, key presses from each keyboard is specific to that device.

If a process accesses an keyboard in the "raw" mode and another process accesses a separate keyboard in the "cooked" mode, these processes will not interfere with each other. However, if there are two separate processes accessing the same Keyboard and one process accesses it in the "raw" mode and the other accesses it in the "cooked" mode, all of the data will go to the "raw" mode process.

Communicating with HP-HIL Devices

This section provides sample programs which can be used to identify and describe all of the HP-HIL devices supported by HP-UX and it explains the data read by the programs.

To use the programs in this section, you must type them in using either the `vi` editor or HP-UX editor of your choice (*the line numbers are not part of the program*). Next, compile the programs using either the C Language compiler command `cc`, the FORTRAN compiler command `fc`, or the Pascal compiler command `pc`. The programs will return the device status and 5 hexadecimal values of the Describe Record. Next, they wait for you to move the HP Mouse around on your desk or type in something from the HP 46021A Keyboard before returning any data. Note that this same process and programs may be used to test all the HP-HIL devices supported on the HP-UX operating system.

Sample C Language Program

The sample program presented in this section is used to open a link to the device located at link address 1, it requests that an Identify and Describe be performed on the device, reads some data, and then closes the link to the device. To communicate with an HP-HIL device from the C Language, these intrinsics are used: `open`, `close`, `read`, and `ioctl`.

C Program Listing.

```
1   #include <sys/hilioctl.h>
2   main()
3   {
4      int fd, status, index, bytes_read, done;
5      unsigned char describe[11], buffer[10]
6
7      /*Open the device file for the first device on the loop.*/
8
9      fd = open("/dev/hil1", 0);
10
11     for (index = 0; index < 12; index++)
12     {
13        describe[index] = ' ';
14     }
15
16     /*This ioctl system call requests a describe record from
17       the device.  The describe record contains information
18       describing the amount and types of data that can be
19       returned by the device.*/
20
21     status = ioctl(fd, HILID, &describe[0]);
22     printf("status %d    \n",status);
23     printf("describe record\n");
24
25     for (index=0; index<12; index++)
26     {
27        printf("      %x\n", describe[index]);
28     }
29     printf("\n", ' ');
30
31     /*Read at least 21 bytes of data from the device.*/
32
33     done = 0;
34     while (done < 21)
35     {
36        bytes_read = read(fd, buffer, 1);
37        done += bytes_read;
38
39        for (index = 0; index < bytes_read; index++)
40        {
41           printf("      %x\n", buffer[index]);
42        }
43     }
44     close(fd);
45  }
```

The following is an explanation of the program:

Line 1 provides the `include` file (`sys/hilioctl.h`) which contains a list of macros that execute specific functions when used within a C program. The `HILID` macro (*line 21*) which is called in the above program executes the identify and describe function. Note that the macro `HILID` can be found in the file `hilioctl.h`. The path name for the `hilioctl.h` file is `/usr/include/sys/hilioctl.h`. If you execute the `cat` or `more` command on this path name, you will receive a screen listing of this file's contents.

Lines 4 and 5 declare the variables to be used in the program.

Line 9 opens the file `/dev/hil1` for reading.

Lines 11 through *14* initialize the describe array.

Line 21 uses the system call `ioctl` to call the macro `HILID` which when executed returns a describe array in the argument `describe[0]`. A function return value is assigned to the variable `status`.

Line 22 prints the value for `status` and *Line 23* prints the column header for the `describe record` listing.

Lines 25 through *28* are a `for` loop which list the contents of the `describe` array. The contents listed are the device ID, the describe record header, and the I/O descriptor byte.

Line 29 prints a blank and executes a carriage return.

Line 33 initializes the variable `done`.

Lines 34 through *43* are a `while` loop which reads 21 bytes of data from the device one byte at a time and then list this data on the standard output (CRT). The data returned is the number of bytes in each packet, a time stamp, and a poll record header and it's parameters.

Line 44 closes the file `/dev/hil1`.

The results of using this program to read data from an HP Mouse are similar to:

```
status  0
describe record
      68
      12
      C2
      1E
      2
      0
      0
      0
      0
      0
      0
      0

      8
      2
      82
      D9
      D3
      2
      1
      0
      8
      2
      82
      D9
      DB
      2
      1
      1
      8
      2
      82
      D9
      DD
```

An explanation for these results can be found in the section entitled, "Description of the Data Returned by the Programs."

Sample Pascal Program

The sample program presented in this section is used to open a link to the device located at link address 1. It requests that an Identify and Describe be performed on the device, read some data, and then close the link to the device. To communicate with an HP-HIL device from Pascal, the HP-HIL device file is opened by executing a Pascal `reset`. There is also the need to use the Pascal `alias` directive to make function and procedure calls to the C Language commands: `ioctl` and `sprintf`. The other Pascal directive used is `sysprog` on.

Pascal Listing for Series 300.

```
 1 $sysprog on$
 2
 3 program hildev (input, output);
 4
 5 const
 6    maxstr = 255;
 7    maxdes = 12;
 8    loopcount = 21;
 9    hil_id = 1074554883;
10    format = '     %X'#0;
11
12 type
13    anystr = packed array [1..maxstr] of char;
14    des_array = packed array[ 1..maxdes ] of char;
15
16 var
17    device_f                  : file of char;
18    buf                       : char;
19    describe                  : des_array;
20    index, status             : integer;
21    format_str                : packed array[ 1..8] of char;
22    line                      : anystr;
23
24 function ioctl $alias '_ioctl'$ (fd, hilid : integer;
25                                   var des : des_array): integer; external;
26
27 procedure sprintf $alias '_sprintf'$ (anyvar str, format : anystr;
28                                       num : integer); external;
29
30 begin
31   (*Open the device file for the first device on the loop.*)
32
33   reset (device_f, '/dev/hil1');
34   for index := 1 to maxdes do describe[index] := chr(0);
35   format_str := format;
```

```
36
37   (*This ioctl system call requests a describe record from
38     the device.  The describe record contains information
39     describing the amount and types of data that can be
40     returned by the device.*)
41
42   status := ioctl(3, hil_id, describe);
43   writeln ('status = ', status:2);
44   writeln ('describe record');
45   for index := 1 to maxdes do
46     begin
47       sprintf (line, format_str, ord(describe[index]));
48       writeln (line);
49     end;
50   writeln;
51
52   (*Read at least 21 bytes of data from the device.*)
53
54   index := 0;
55   while index < loopcount do
56     begin
57       read (device_f, buf);
58       sprintf (line, format_str, ord(buf));
59       writeln (line);
60       index := index + 1;
61     end;
62   close(device_f);
63 end.
```

The following is an explanation of the program:

Line 1 $sysprog on$ is the Series 200/300 Pascal directive which allows you to use ANYVAR. The ANYVAR parameter specifier in a function or procedure relaxes type compatibility checking when the routine is called.

Lines 5 through *10* are the constants defined as follows:

- `maxstr` is the maximum string length for a packed array of characters.
- `maxdes` is the maximum string length for a packed array of characters containing the describe record.
- `loopcount` is the number of times the while loop of *lines 55* through *61* is to be executed.
- `hil_id` is the decimal value of the HILID command. The decimal values for the other HP-HIL commands can be found in a table in the section, "Identify and Describe Command (HILID)." Note that it is only necessary to use these

decimal values of the HP-HIL commands when using the FORTRAN and Pascal programming languages.

- `format` is the character string used to format the output of the `sprintf` command.

Lines 13 through *14* are the type declarations for the program. They are defined as follows:

- `anystr` is a packed array of characters. This packed array of characters is used to declare the variables `line`, `str`, and `format`.
- `des_array` is the packed array of characters used to declare the variable `describe`.

Lines 17 through *22* are a list of the variables declared for this program. They are defined as follows:

- `device_f` is the file name assigned to the device file `/dev/hil1`.
- `buf` is the character variable returned from reading the `device_f` file.
- `describe` is the packed array of characters which is assigned the describe record. The describe record is explained in the section, "Identify and Describe Command (HILID)."
- `index` is a loop control variable.
- `status` is the variable which is assigned the status of the opened file `device_f` (`/dev/hil1`) after executing the HILID command.
- `format_str` is the packed array of characters assigned the value of the constant called `format` (`'  %X'#0`).
- `line` is the packed array of characters that is assigned the value of the character string that is returned when the `sprintf` command is executed.

Line 24 is a function which references the external HP-UX command `ioctl`. Note that the string parameter in the ALIAS directive has an under score preappended to it. This is only true for Series 300 computers.

Line 27 is a procedure which references the external HP-UX command `sprintf`. Note that the string parameter in the ALIAS directive has an under score preappended to it. This is only true for Series 300 computers.

Line 33 opens the device file `/dev/hil1` and assigns it the name `device_f`.

Line 34 initializes the packed array `describe`.

Line 35 assigns the constant `format` to the packed array of characters `format_str`.

Line 42 executes the function `ioctl` and assigns its status value to the variable `status`.

Line 43 causes the variable `status` to be output to the display.

Line 44 outputs the describe record label.

Lines 45 through *49* are a for loop which causes the describe record to be sent to the display or standard output (CRT).

Line 54 initializes the count variable `index` to zero.

Lines 55 through *61* are a while loop which causes the data read from the device file `device_f` to be sent to the display (CRT).

Line 62 closes the device file `device_f`.

The results from using this program to read data from an HP Mouse are similar to:

```
status  0
describe record
      68
      12
      C2
      1E
      2
      0
      0
      0
      0
      0
      0
      0

      8
      2
      82
      D9
      D3
      2
      1
      0
      8
      2
      82
      D9
      DB
      2
      1
      1
      8
      2
      82
      D9
      DD
```

An explanation for these results can be found in the section entitled, "Description of the Data Returned by the Programs".

Sample FORTRAN Program

The sample program covered in this section is used to open a link to the device located at link address 1, it requests an identify and describe be performed on the device, reads some data, and then closes the link to the device. To

communicate with an HP-HIL device from FORTRAN, the HP-HIL device file is opened by executing a FORTRAN OPEN statement. There is also the need to use the FORTRAN ALIAS directive to make calls to the C Language commands: `ioctl` and `read`.

The program is as follows:

FORTRAN Program Listing.

```
 1        program hildev
 2 $alias ioctl='ioctl'(%val,%val,%ref)
 3 $alias read='read'(%val,%ref,%val)
 4        integer fd, istatus, index, HILID, count
 5        parameter (HILID = 1074554883)
 6        character*12 describe
 7        character*1 ch1
 8        open(unit=10,file="/dev/hil1")       ! open for first hil device
 9        fd=fnum(10)                          ! get the file descriptor
10 C                                          ! for unit 10 (/dev/hil1)
11        count = 0                            ! initialize count variable
12        do index = 1, 12                     ! initialize character array
13          describe(index:index) = ' '        ! describe
14        end do
15        istatus=ioctl(fd,HILID,describe)     ! check the status
16        print*,"status = ",istatus           ! output the status
17        print*,"describe record"             ! label describe record
18        do 110 index = 1,12
19          write(6,100) describe(index:index)
20 100     format(' ',6X,Z2.1)                  ! output the describe
21 110   continue                               ! record
22        write(6,'()')
23
24        do while (count .LT. 21)             ! read 21 bytes of data
25          call read (fd, ch1, 1)             ! from the device
26          write(6,'(7X,Z2.1)') ch1
27          count = count + 1
28        end do
29        end
```

The following is an explanation of the program:

Line 1 is the `program` statement which defines the name of the program. The program name for this program is `hildev`.

Lines 2 and *3* are ALIAS directives used to assign internal function names to the external HP-UX commands `ioctl` and `read`.

Line 4 declares the variables contained in it to be integers. These variables are defined as follows:

- `fd` is file descriptor associated with unit number 10.
- `istatus` is the variable assigned the status value returned by the `ioctl` command.
- `index` is the control variable for the DO loop in *lines 18* through *21* of the program.
- `HILID` is the variable associated with the Identify and Describe Command. This variable is assigned the value shown in the PARAMETER statement of *line 5*. The decimal code values for the various HP-HIL commands can be found in the section entitled, "Identify and Describe Command (HILID)."
- `count` is the count variable for the DO WHILE loop of *lines 23* through *27*.

Lines 6 and *7* are character variables defined as follows:

- `describe` is a string of characters twelve characters long. This variable is assigned the describe record when the `ioctl` function is called.
- `ch1` is a one character string. This variable is assigned the data received when a call to the `read` command is made in the DO WHILE loop of *lines 23* through *27*.

Line 8 opens the device file `/dev/hil1`.

Line 9 assigns the value of unit number 10 to `fd`.

Line 11 initializes the count variable to zero.

Lines 12 through *14* form a `do` loop which initializes each character of `describe` to blank.

Line 15 assigns the value of the `ioctl` function call to the variable `istatus`.

Line 16 displays the value assigned to `istatus` on the standard output (CRT).

Line 17 displays the describe record label on the standard output.

Lines 18 through *21* display the contents of the character string `describe` on the standard output.

Lines 24 through *28* read the data from the device file `/dev/hil1` and display it on the standard output.

The results from using this program to read data from an HP Mouse are similar to:

```
status  0
describe record
      68
      12
      C2
      1E
      2
       0
       0
       0
       0
       0
       0
       0

       8
       2
      82
      D9
      D3
       2
       1
       0
       8
       2
      82
      D9
      DB
       2
       1
       1
       8
       2
      82
      D9
      DD
```

An explanation for these results can be found in the next section.

Description of the Data Returned by the Programs

This section of the documentation provides you with an interpretation of the data obtained from running any one of the previously discussed programs. To understand how to interpret the Identify and Describe Command data returned

by these programs, read the section entitled, "Identify and Describe Command (HILID)."

If you used the previously discussed programs to identify and describe an HP 46060A (HP Mouse), the first set of data returned to you is created by lines 21 through 29 of the C language program, 41 through 48 of the Pascal program and 15 through 21 of the FORTRAN program. The data looks similar to the following:

```
status    0
describe record
      68
      12
      c2
      1e
      2
      0
      0
      0
      0
      0
      0
      0
```

where the `status` returned is `0` which indicates that you can communicate with the HP-HIL device. If you receive a `-1`, it means you are not able to communicate with the HP-HIL device. The remaining data created by the `for` loop of lines 25 through 28 is explained as follows:

`68` is the device ID for an HP 46060A (HP Mouse).

`12` is the describe record header which supplies some of the parameters of the device and provides an indication of how much additional information is to follow this parameter. The 8 bit character string for the hexadecimal value 12 is: 00010010. Reading this bit string from right to left you find that bit 0 is 0, bit 1 is 1, and bit 5 is 0. This says that the header uses 16 bits describing the resolution of the device in counts per meter, and axes X and Y will be reported. Bit 4 is set because the I/O descriptor byte is to appear later on in the Describe Record.

`c2` is the lower-byte resolution.

1e is the higher-byte resolution. To determine the total counts per meter multiply the higher byte by 256 (100 hex) and add the result to the lower byte read above.

```
lower-byte resolution -- c2 hex = 194 decimal
higher-byte resolution -- 1e hex = 30 decimal
100 hex = 256 decimal
```

Your total counts per meter reading is determined as follows:

```
30 × 256) + 194 = 7874 counts per meter
```

2 is the I/O descriptor byte. The 2 reading indicates the buttons for which the device reports keycodes. The data here indicates that keycodes are reported for buttons one and two.

Note that the zeros following the I/O descriptor byte (2) provide no information. In the case of the FORTRAN sample program, blanks were returned. These zeros and blanks fill in the data locations of the remaining seven empty bytes of data which may be obtained from the Identify and Describe Command depending on the device file you are reading. The Identify and Describe Command can return up to 12 bytes of data:

Identify and Describe Command Output

Device ID byte
Describe Record Header
Number of counts per centimeter (meter) low byte
Number of counts per centimeter (meter) high byte
Maximum count of X-axis low byte
Maximum count of X-axis high byte
Maximum count of Y-axis low byte
Maximum count of Y-axis high byte
Maximum count of Z-axis low byte
Maximum count of Z-axis high byte
I/O descriptor byte
Identify and describe command

The interpretation of the Identify and Describe bytes can be found in the section entitled, "Identify and Describe Command."

The second set of data received from the aforementioned program looks similar to the following display. Note that leading zeros of the two digit hexadecimal values have been omitted.

```
8
2
82
D9
D3
2
1
0
8
2
82
D9
DB
2
1
1
8
82
D9
DD
```

where:

8	is the number of bytes contained in the packet which was read including the length byte. Eight bytes of data were read in this packet.
2 82 D9 D3	is a time stamp in tens of milliseconds since power-up. The time stamp since power-up for this packet of data is 42 129 875 milliseconds.
2	is the poll record header. This header indicates to the System the type and quantity of information to follow, as well as reporting simple status information. The 8 bit character string for the hexadecimal value of 02 is: 00000010. This shows bit 0 is 0 and bit 1 is 1, which indicates that the coordinate axes the device is reporting are: X and Y.

1	is the X coordinate relative position of 1. Note that this reading is how much the X coordinate has moved since the last poll.
0	is the Y coordinate relative position of zero (0). Note that this reading is how much the Y coordinate has moved since the last poll variable.
8	is the number of bytes in the second packet of data sent. Eight bytes of data were read in this packet.
2 82 D9 DB	is a time stamp in tens of milliseconds since power-up. The time stamp since power-up for this packet of data is 42 129 883 milliseconds.
2	is the poll record header. This header indicates to the System the type and quantity of information to follow, as well as it reports simple status information. The 8 bit character string for the hexadecimal value of 02 is: 00000010. This shows bit 0 is 0 and bit 1 is 1, which indicates that the coordinate axes the device is reporting are: X and Y.
0	is the X coordinate position of one (1).
1	is the Y coordinate position of one (1).
8	is the number of bytes in the second packet of data sent. Eight bytes of data were read in this packet.
2 82 D9 DD	is a time stamp in tens of milliseconds since power-up. The time stamp since power-up for this packet of data is 42 129 885 milliseconds.

The last three bytes of data that were not shown in the previous data listing were truncated by the program. These bytes of data if read would return the Poll Record Header, X-axis coordinate position and the Y-axis coordinate position.

If you were using an HP 46020A (Keyboard) instead of an HP 46060A (HP Mouse), then the second set of data read from the device would look similar to the following display. Note that leading zeroes of the two-digit hexadecimal values have been omitted. It should also be noted that the describe record header returned for the HP 46020A is the hexadecimal value DF, which indicates that it is an Extended Keyboard. The trailing zeros again should be ignored, as they do not provide any information.

```
7
A2
7D
37
B6
40
F2
7
A2
7D
37
BA
40
F3
7
A2
7D
49
20
40
F2
```

where:

7	is the number of bytes contained in the packet which was read. Seven bytes of data were read in this packet.
A2 7D 37 B6	is a time stamp in tens of milliseconds since power-up. The time stamp since power-up for this packet of data is 2 726 115 254 milliseconds (A2 7D 37 B6 hexadecimal = 2 726 115 254 decimal).
40	is the poll record header. This header indicates to the system the type and quantity of information to follow, as well as reporting simple status information. The 8 bit character string for the hexadecimal value of 40 is: 0100 0000. Bit 6 is 1 (rightmost bit is bit 0), which indicates that the information read is up to 8 bytes and that the data produced can be

	interpreted by using Keycode Set 1 (Keycode Set 1 is provided at the end of this part of the User's Guide).
f2	is the (space bar) which has been pressed down.
7	is the number of bytes contained in the second packet which was read. Seven bytes of data were read in this packet.
A2 7D 37 BA	is a time stamp in tens of milliseconds since power-up. The time stamp since power-up for this packet of data is 2 726 115 258 milliseconds (A2 7D 37 BA hexadecimal = 2 726 115 258 decimal).
40	is the poll record header. This header indicates to the system the type and quantity of information to follow, as well as reporting simple status information. The 8 bit character string for the hexadecimal value of 40 is: 0100 0000. Bit 6 is 1 (rightmost bit is bit 0), which indicates that the information read is up to 8 bytes and that the data produced can be interpreted by using Keycode Set 1 (Keycode Set 1 is provided at the end of this part of the User's Guide).
f3	is the (space bar) which has been released.
7	is the number of bytes contained in the packet which was read. Seven bytes of data were read in this packet.
A2 7D 49 20	is a time stamp in tens of milliseconds since power-up. The time stamp since power-up for this packet of data is 2 726 119 712 milliseconds (A2 7D 49 20 hexadecimal = 2 726 119 712 decimal).
40	is the poll record header. This header indicates to the system the type and quantity of information to follow, as well as reporting simple status information. The 8 bit character string for the hexadecimal value of 40 is: 0100 0000. Bit 6 is 1 (rightmost bit is bit 0), which indicates that the information read is up to 8 bytes and that the data produced can be interpreted by using Keycode Set 1 (Keycode Set 1 is provided at the end of this part of the User's Guide).
f2	is the (space bar) which has been pressed down.

HP-HIL Commands

This section contains descriptions of the various HP-HIL commands (opcodes), including their usage, effects, and format when transmitted and received. Note that the term **command** as it relates to opcode is explained in the section entitled, "A Few Terms." The commands covered in this section are used within the macros located in the file `/usr/include/sys/hilioctl.h`. As stated earlier these macros are called by using them as parameters in an `ioctl` command. For example:

```
ioctl(fd, HILID, &describe[0])
```

The following is a table of the macros included in the file `/usr/include/sys/hilioctl.h`. This table contains the hexadecimal commands (opcodes) for these macros, as well as a brief description of each macro. A detailed description of the commands is provided in the sections which follow this table.

Table 7-2. HP-HIL Macros

Macro	Opcode (hexadecimal)	Description
HILID	03h	Identify and Describe
HILPST	05h	Perform Self Test
HILRR	06h	Read Register
HILWR	07h	Write Register
HILRN	30h	Report Name
HILRS	31h	Report Status
HILED	32h	Extended Describe
HILSC	33h	Report Security Code
HILDKR	3Dh	Disable Keyswitch Auto Repeat
HILER1	3Eh	Enable Keyswitch Auto Repeat, cursor key repeat rate = 1/30 second. This is based on a system poll rate of 1/60 second.
HILER2	3Fh	Enable Keyswitch Auto Repeat, cursor key repeat rate = 1/60 second. This is based on a system poll rate of 1/60 second.
HILP1..HILP7	40h..46h	Prompt 1 through Prompt 7
HILP	47h	Prompt (General Purpose)
HILA1..HILA7	48h..4Eh	Acknowledge 1 through Acknowledge 7
HILA	4Fh	Acknowledge (General Purpose)

The above macro names cannot be used directly as parameters for an `ioctl` system call from Pascal or FORTRAN. However, if you assign the decimal value of the code to a declared integer variable and then use it as a parameter to an `ioctl` command, you can use the macro within a Pascal or FORTRAN program. The following table provides you with a listing of these macros in their decimal form.

Table 7-3. HP-HIL Macros and Their Decimal Equivalent

Macros	Decimal Equivalent
HILID	1074554883
HILPST	1073833989
HILRR	-1073649658
HILWR	-2147391481
HILRN	1074817072
HILRS	1074817073
HILED	1074817074
HILSC	1074817075
HILDKR	-2147456963
HILER1	-2147456962
HILER2	-2147456961
HILP1	-2147456960
HILP2	-2147456959
HILP3	-2147456958
HILP4	-2147456957
HILP5	-2147456956
HILP6	-2147456955
HILP7	-2147456954
HILP	-2147456953
HILA1	-2147456952
HILA2	-2147456951
HILA3	-2147456950
HILA4	-2147456949
HILA5	-2147456948
HILA6	-2147456947
HILA7	-2147456946
HILA	-2147456945

Identify and Describe Command (HILID)

This command is used to determine the type of devices that are connected to the HP-HIL link, as well as the characteristics of these devices. Each device responds to the Identify and Describe command with a series of data which can vary in length from 2 to 11 bytes. The content of this data is as follows:

- Device ID Byte,
- Describe Record Header,
- I/O Descriptor Byte.

The remainder of this section will cover how to interpret the information contained in these bytes and header.

Device ID Byte

This is used to identify the general class of device and its nationality in the case of a keyboard or keypad. Since the sample programs in this documentation return only a two digit hexadecimal value as a device ID, you need a table which can be used to interpret what this value means. The following is a table of HP-HIL devices and the hexadecimal values which correspond to these devices.

Table 7-4. HP-HIL Device Identification Codes

Device Type	ID Range (hex)	Device Description
Keyboard	E0 ... FFh	Standard Keyboard (85-87 keys)
	C0 ... DFh	Extended Keyboard (107-109 keys)
	A0 ... BFh	Compressed Keyboard (91-93 keys)
Absolute Positioners	98 ... 9Fh	Undefined
	90 ... 97h	Graphics Tablet and Digitizer
	8C ... 8Fh	Touchscreen
	88 ... 8Bh	Touch-pad
	80 ... 87h	Undefined
Relative Positioners	70 ... 7Fh	Undefined
	6C ... 6Fh	Undefined
	68 ... 6Bh	HP Mouse
	60 ... 67h	Generic Quadrature Devices (e.g., Control Dials, Quad Port, etc.)
Character Entry	5C ... 5Fh	Barcode Reader
	50 ... 5Bh	Undefined
	40 ... 4Fh	Undefined
Other	30 ... 3Fh	32-button Module and ID Module
	2C ... 2Fh	Undefined
	20 ... 2Bh	Undefined
	00 ... 1F	Vectra Keyboard

If the HP-HIL device you are using with the sample program is a keyboard or keypad, then you can to use the following table to determine its nationality.

Table 7-5. HP-HIL Keyboard Nationality Codes

Lower 5 bits of Device ID (hex)	Nationality of Keyboard or Keypad
00 ... 02h	Undefined
03h	Swiss/French
04h	Portuguese
05h	Arabic
06h	Hebrew
07h	Canadian/English
08h	Turkish
09h	Greek
0Ah	Thai (Thailand)
0Bh	Italian
0Ch	Hangul (Korea)
0Dh	Dutch
0Eh	Swedish
0Fh	German
10h	Chinese - PRC (China)
11h	Chinese - ROC (Taiwan)
12h	Swiss/French II
13h	Spanish
14h	Swiss/German II
15h	Belgian (Flemish)
16h	Finnish
17h	United Kingdom

Continued on next page ...

Table 7-5. HP-HIL Keyboard Nationality Codes (continued)

Lower 5 bits of Device ID (hex)	Nationality of Keyboard or Keypad
18h	French/Canadian
19h	Swiss/German
1Ah	Norwegian
1Bh	French
1Ch	Danish
1Dh	Katakana
1Eh	Latin American/Spanish
1Fh	United States
20 ... FFh	Undefined

To use Table 7-5, assume the hexadecimal ID number returned is `DF`. Use the "Device Identification Codes" table to determine the type of device being used. Reading down the second column of this table you find that the device corresponding to `DF` is an Extended Keyboard (107-109 keys). You next need to determine its nationality since it is a keyboard. To do this, use the "Keyboard Nationality Codes" table and the lower 5 bits of the hexadecimal value `DF`. In the case of this example the lower 5 bits are `11111` binary or `1f` hexadecimal. In the table `1F` corresponds to the United States; therefore, the keyboard is designed for use in the United States.

7

Describe Record Header

This is used to supply additional information about the axes used by the device if it is intended to return coordinates, provide information about the I/O Descriptor Byte, and give information about additional commands. The Describe Record Header is the second hexadecimal value read by the sample program in this documentation and each of the 8 bits of this value represents an important piece of data. The following table can be used to interpret this data.

	Description of Describe Record Header
Bit 7	Set if the device contains two independent sets of coordinate axes. An example of two independent sets of HP-HIL devices is a device with two joysticks connected to it. Each of the joysticks have their own set of coordinate axes. Note that currently joysticks are not supported on HP products. It is assumed, however, that both sets of coordinate axes share common characteristics as identified in the remainder of the record. Default (clear) indicates a maximum of one set of axes.
Bit 6	Set if the device is to return absolute positional data (unsigned integers). Default (clear) indicates relative data (2's complement).
Bit 5	Set if the device returns all positional information at 16-bits/axis. Default (clear) is 8-bits/axis.
Bit 4	Set if the I/O Descriptor Byte is to follow later in the Identify and Describe Record. Default (clear) indicates that the device has no buttons, no proximity detection, and no prompt/acknowledge functionality, with no I/O Descriptor Byte to follow.
Bit 3	Set if the device supports the Extended Describe command. This command is covered later on in this documentation. Default (clear) indicates that the Extended Describe command is not supported.
Bit 2	Set if the device supports the Report Security Code command. This command is covered later on in this documentation.

Default (clear) indicates that the Report Security Code command is not supported.

Bits 1 and 0 — Bits 1 and 0 indicate the coordinate axes the device reports. If these bits are nonzero and bit 5 is set, then following the header will be 16 bits (2 bytes) of data describing the resolution of the device in counts per centimeter. However, if bit 5 is clear, the resolution will be in counts per meter. If the device is an absolute positioner, there will be an additional 16 bits/axis detailing the extent of each coordinate axis. For example, if the HP-HIL device has X, Y, and Z axes then there will be a maximum count/axis given for the lower and higher bytes of each axis. This is true no matter if the data is being report in 8 bit or 16 bit format. To determine the number of axes used by an HP-HIL device, use the following table:

bit 1	bit 0	Axes Reported
0	0	None
0	1	X
1	0	X and Y
1	1	X, Y, and Z

You now have the information necessary for interpreting a Describe Record Header. An example using this Identify and Describe Record parameter can be found in the section entitled, "Description of the Sample Program's Data."

I/O Descriptor Byte

This is used to indicate the buttons the device reports keycodes for, whether the device has proximity detection, and what Prompt/Acknowledge functions, if any, are implemented in the device. Proximity detection is a way of determining whether the stylus is in contact with the X and Y axis sensing device. Note that Prompt and Acknowledge are treated as a set, and no device may indicate support of any particular Prompt or Acknowledge without also supporting its counterpart. If none of the above features are implemented, the I/O Descriptor Byte is not transmitted. The following is the description of this byte:

Description of Bits for I/O Descriptor Byte

Bit 7

Set if the device implements the general purpose Prompt and Acknowledge functions. Generally speaking, a Prompt is an audible or visual indication to you that the HP-UX system is ready for some form of input, and Acknowledge is an indication to you that the input has been received by the HP-UX system. Default (clear) implies these functions are not implemented.

Bits 6, 5, and 4

Indicate specific Prompt/Acknowledges (Prompt 1..7 and Acknowledge 1..7) implemented in the device. Default (clear) indicates none. Use the following table to determine Prompt/Acknowledges:

Bit 6	Bit 5	Bit 4	Prompt/Acks. Implemented
0	0	0	None
0	0	1	1
0	1	0	1 and 2
0	1	1	1, 2, and 3
1	0	0	1 .. 4
1	0	1	1 .. 5
1	1	0	1 .. 6
1	1	1	1 .. 7

Bit 3

Set if the device reports the Proximity In/Out keycodes. Proximity In/Out keycodes describe the stylus or pointers position as it moves into or out of contact with an X and Y axis sensing device (e.g. digitizer or touchscreen). Default (clear) indicates no proximity detection. Proximity detection is a way of determining whether the stylus is in contact with the X and Y axis sensing device.

Bits 2, 1, and 0 — Indicate the buttons for which the device reports keycodes. A button report table is given below.

Bit 2	Bit 1	Bit 0	Buttons Reported
0	0	0	None
0	0	1	1
0	1	0	1 and 2
0	1	1	1, 2, and 3
1	0	0	1 .. 4
1	0	1	1 .. 5
1	1	0	1 .. 6
1	1	1	1 .. 7

You now have the information necessary for interpreting an I/O Descriptor Byte. An example using this Identify and Describe Record parameter can be found in the section entitled, "Description of the Sample Program's Data."

In the description of the Sample Program found in this documentation, you should recall that the first `for` loop returned the Identify and Describe Record data and the `while` loop and its internal `for` loop returned data read when using the device. To interpret the latter set of data, you need to know how to read the information included with the Poll Record Header. The Poll Record Header returns information on X, Y, and Z coordinate positions, and keycodes sets being used. The Poll Record Header bits are assigned as follows:

Description of the Poll Record Header

Bit 7

Set if the device is reporting data from the second set of coordinate axes. Default (clear) indicates data from set 1.

Bits 6, 5, and 4

Based on the value of these bits, you can determine the data type. The following is a table for determining the data type:

Bit 6	Bit 5	Bit 4	Data Type
0	0	0	No character data to follow
0	0	1	Reserved Character Set 1
0	1	0	US ASCII Characters
0	1	1	Binary Data
1	0	0	Keycode Set 1
1	0	1	Reserved Character Set 2
1	1	0	Keycode Set 2
1	1	1	Keycode Set 3

(*Note that Set 2 Keycodes are device dependent and user definable.* See the section entitled, "A Few HP-HIL Devices.")

Bit 3

Set indicates request for status check. Default (clear) indicates status unchanged.

Bit 2

Set indicates device ready for data. Default (clear) indicates not ready for data transfer at this time.

Bits 1 and 0

Indicate the coordinate axes the device is reporting. The following table provides information for determining which axes are to be reported.

Bit 1	Bit 0	Axes Reported
0	0	None
0	1	X
1	0	X and Y
1	1	X, Y, and Z

The Poll Record Header is followed by device data. If the device indicated that it would report coordinate information as 16-bits/axis in the Describe Record Header, then for each axis reported there is a lower byte and then a higher byte coordinate. Otherwise, the higher byte is not transmitted. In general, the Poll Record format indicates the maximum data which can be reported. Note that most devices only transmit a subset each time.

7

Once the positional data has been given, the next set of data provided by the Poll Record is 8 bytes of character data, as specified by the Poll Record Header Bits 5 and 6.

Perform Self Test (HILPST)

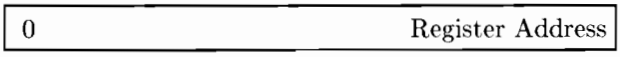
This command causes the addressed device(s) to perform a self test, returning 1 to 15 data frames. A single data frame of 00h indicates a successful test, with nonzero values representing device-specific failures.

Read Register (HILRR)

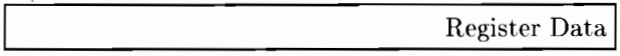
The Read Register provides a means for interaction with more complex devices via HP-HIL, allowing for types of data transfer not generally supported by the HP-HIL devices.

A device indicates support of the Read Register command in the Extended Describe Record (covered later in this documentation), it also indicates the specific read registers contained in the device. To perform a register read, the system transmits a data frame containing the address of the register it is to read, followed by the Read Register command. The device, upon receiving the command, transmits the contents of the register indicated by the data frame. The following diagrams illustrate this process.

System sends:



Device responds:



Devices which do not support the Read Register command discard all data frames.

Write Register (HILWR)

Write Register provides a means of setting the contents of individual registers in devices supporting this advanced feature, as well as providing the means for transmitting a large amount of data to a device at speeds approaching the links maximum of 6 500 bytes/second.

There are two forms of the Write Register command which may be used separately or mixed. Devices indicate support of either of these two forms (or both) in the Extended Describe Record.

To write to individual registers in a device, the system transmits the register address, followed by the intended contents of that register. This is called Write Register Type 1. Several register address/data pairs may be transmitted in sequence.

System sends:

0	Register Address
	Register Data
0	Register Address
	Register Data
0	Register Address
	Register Data
	.
	.
	.
0	Register Address
	Register Data

The system may also wish to write several bytes to a single register in sequence, utilizing Write Register Type 2. Type 2 transfers are sent to the device by setting the most significant bit in the byte containing the register address, followed by as many data frames as desired (up to the maximum Write Buffer Length of the device, reported in the Extended Describe Record). Although it is possible to write several bytes to the same register using Type 1 transfers in devices supporting both types (repeating the register address before each byte of data), the intention of the two types of data transfers is quite different, and use of Type 2 is recommended. Write Register Type 2 has the following format:

System Sends:

1	Register Address
	Register Data
	Register Data
	Register Data
	.
	.
	.
	Register Data

The two formats may also be mixed for devices supporting both Write Register Type 1 and Type 2. There is, however, the restriction that all writes to individual registers occur first, because once the device detects the most significant bit in the register address byte to be set it assumes all following data frames to be data for that register. The following diagram illustrates the mixing of the two Types.

System sends:

Mixing of Register Type 1 and Type 2

0	Register Address
	Register Data
0	Register Address
	Register Data
	. . .
1	Register Address
	Register Data
	Register Data
	. . .
	Register Data

Devices which do not support the Write Register command will discard all data frames.

Report Name (HILRN)

Report Name is used to request a string of up to 15 characters (8-bit US ASCII) which would aid in describing the device to the user. Devices indicate support of the Report Name command in the Extended Describe Record.

Report Status (HILRS)

Report Status is used to extract device-specific status information from devices configured on the link. Devices indicate support of the Report Status command in the Extended Describe Record. Devices supporting the command respond with from 1 to 15 bytes of device-specific status information. Interpretation of the status bytes will necessarily depend upon the device in question.

Extended Describe (HILED)

Support of the Extended Describe command is indicated in the Describe Record header. It provides additional information concerning more advanced device features which may not be required for basic operation.

Devices supporting the Extended Describe command respond with a series of data types referred to as the Extended Describe Record. The record length may vary from 1 to 15 bytes (although only 5 bytes are currently defined). The Extended Describe Record has the following format:

Extended Describe Record

	Extended Describe Record Header
0	Maximum Read Register Supported
0	Maximum Write Register Supported
	Maximum Write Buffer Length Low Byte
	Maximum Write Buffer Length High Byte
	Localization Code

Devices responding to the Extended Describe command return at least 1 byte of data, the Extended Describe Record Header. Devices supporting Read Register to Write Register will need to report additional information so that their capabilities may be more fully defined. The Extended Describe Record Header both supplies some of the parameters of the device and provides an indication of how much additional information is to follow. The meanings of the individual bits in the Header are as follows:

Table 7-6. Description of Extended Describe Record Header

bit 7	Reserved for future use. Default will be clear.
bit 6	Set if the Localization Code is supported. If set, then following the Maximum Write Buffer Length High Byte will be one byte indicating the nationality of the device (keyboard). See the section entitled, "Identify and Describe Command (HILID)" for a listing of the Localization Codes (HP-HIL Keyboard Nationality Codes). Default (clear) indicates that the Localization Code is not supported.
bit 5	Set if the Report Status command is supported. Default (clear) indicates Report Status not supported.
bit 4	Set if the Report Name command is supported. Default (clear) indicated Report Name not supported.
bit 3	Reserved for future use. Default will be clear.
bit 2	Set if Read Register supported. If set, immediately following the Header is a byte indicating the registers supported for reading in the device. Default is clear, indicating Read Register not supported.
bits 1 and 0	Bit 1 and bit 0 indicate support of the Write Register command. If bit 1 is set, Write Register Type 2 is supported by the device. If bit 0 is set, Write Register Type 1 is supported. If both bits are set, then the device supports both Type 1 and Type 2. If either bit 1 or bit 0 is set, then following in the Record will be information indicating the registers supported for writing in the device. If bit 1 is set, then an additional 16 bits will be returned indicating the maximum number of data bytes which may be written to the device at a time using Write Register Type 2 without data loss.

If the device indicated support for the Read Register command in the Header, then following the Header is a byte indicating the read registers supported by the device. The maximum Read Register supported byte indicates the largest read register address supported. Note that it is assumed that all addresses less than this maximum are also supported. Thus a byte of 0Fh indicates that the device contains 16 read registers, addressed as read registers 0..15. HP-HIL protocol allows for devices containing up to 128 read registers, addressed as 0..127.

If Write Register (Type 1 or Type 2) support is indicated, then next is a byte indicating the write registers supported. The maximum Write Register supported byte indicates the largest write register address supported in the device. It is assumed that all addresses less than the maximum are also supported. Up to 128 write registers, addressed as 0..127, are supported in the HP-HIL protocol.

If Write Register Type 2 is supported, as indicated by bit 1 of the Extended Describe Record Header being set, then following the maximum Write Register supported byte is 16 bits of data indicating the maximum number of bytes which may be transmitted to the device in a Type 2 transfer without overflowing the device's internal buffer. This number, transmitted first low byte, then high byte, represents the buffer length of the device minus 1. Thus a device capable of buffering 1024 bytes of data would transmit a maximum Buffer Length Low Byte of FFh and a maximum Buffer Length High Byte of 03h.

If the Localization code is supported, then the Localization Code byte will be included in the Extended Describe Record. The Localization Code is an 8-bit number which corresponds to a nationality (language) of a keyboard.

Report Security Code (HILSC)

The Report Security Code command is used to extract a unique identifier from the device. Support of the command is indicated in the Describe Record Header. The security code is a series of from 1 to 15 bytes which uniquely identify the device in question. Similar in purpose to a serial number, it may also contain information related to user identity, network address, or other information which is unique to a particular user or environment.

The only data transmitted by the ID Module is in response to the Report Security Command and a self test command.

The Report Security command invokes a data response according to the HP-HIL specifications. The following information applies to any device that supports the report security command.

The Data Format Type 1 consists of a one byte header and eight bytes of binary data. The eight data bytes are the packed product and serial numbers of the HP-HIL device. In the case where ID Module is an exchange module signified by a ten digit part number, the five digit prefix number remains the same and the product number letter is replaced by the least significant digit of the part number.

The product, exchange and serial number formats are:

Table 7-7. Product, Exchange and Serial Number Formats

Header	: H	(1 byte header)
Product number is	: DDDDDA	(5 digits and 1 ASCII character)
Exchange part number	: DDDDDd	(5 digits and 1 ASCII character)
Serial number is	: YYWW@NNNNN	(9 digits and 1 ASCII character)

where:

H	is the data header.
DDDDD	is the product number (e.g. 46084).
A	is the product number alpha character.
d	is the least significant numeric character of the exchange number.
YY	is the year code (year less 60).
WW	is the week code (0 to 51).
@	is the letter designation of the country manufacturing the device.
NNNNN	is the serial suffix (0 to 99,999)

The header byte is transmitted before the eight data bytes. The header's purpose is to allow for other data formats; however, none are currently implemented.

The five digits of the product or exchange part prefix number are converted to a two-byte binary number and the high-order bit of a third byte. The remaining lower seven bits of the third byte contain the ASCII character. In products where two alpha characters are used in the product number, only the first character is used in the data format. The order of the bytes have been arranged to transmit the least significant byte of the number first.

In a similar manner, the nine digits of the serial number are converted to a four byte binary number. The country code of manufacturing is in the last byte to be transmitted and is an ASCII character.

The Report Security data bytes are transmitted starting with byte 1 and going through byte 9. Bits are numbered starting with bit 0 at the right most position of the byte (least significant bit) and going through bit 7 (most significant bit), left-most position.

The report security Data Format Type 1 is:

Table 7-8. Report Security Data Format Type 1

Byte	Bit	Description
1	7 - 0	The first byte is the header containing the number 10 hexadecimal for the following format. The general scheme for the header is: Bits 7 - 4 assigned as format variations, where format 1 is the only assignment. Bits 3 - 0 undefined, but set to zero.
2 3 4	7 - 0 7 - 0 7	The second and third bytes and the 7th bit of the fourth byte represent the 5 digits of the product or exchange part number DDDDD in binary form. The least significant bit is bit 0 of byte two.
4	6 - 0	The least significant seven bits of byte four represent the product letter or the least significant digit of the exchange number numeric character. The character is the US ASCII 7 bit representation of the character.
5 6 7 8	7 - 0 7 - 0 7 - 0 5 - 0	The fifth, sixth, seventh bytes and the six least significant bits of byte eight represent the 9 digits of the serial number YYWWNNNNN in binary form, without the alpha character. The least significant bit is bit 0 of byte 5.
8	7 - 6	The two most significant bits of byte eight are reserved for future use and are set to zero.
9	6 - 0	The least significant seven bits of byte 9 represent the serial number letter. The character is the US ASCII 7 bit representation of the character.
9	7	The most significant bit of byte nine is reserved for future use and is set to zero.

Sample of Report Security Format for a Product Module

The following information is returned upon receiving a Report Security command for a Product Module. The data is based on the data format described in the last section. Byte 1 is the first byte sent from the module to the host.

The sample results given are based on the product number 46084A and serial number 2519A00001. The serial number corresponds with the year of 1985, week 19, and serial number suffix 00001. Note that by adding 60 to the above serial numbers first two digits you get the year 85.

Table 7-9. Sample Report Security Results for a Product Module

Byte No.	Data (hex)	Description
1	10	Header
2	04	Part of product number 46084
3	B4	Part of product number 46084
4	41	Product letter "A" and part of product number 46084
5	61	Part of serial number
6	B0	Part of serial number
7	03	Part of serial number
8	0F	Part of serial number
9	41	Country of Manufacturing Code

Sample of Report Security Format for an Exchange Module

The following information is returned upon receiving a Report Security command for an Exchange Module. The data is based on the data format described in the section, "Report Security Code (HILSC)." Byte 1 is the first byte sent from the module to the host.

The sample results given are based on the exchange number 46084-69901 and serial number 2519A00001. The serial number corresponds with the year of 1985, week 19, and serial number suffix 00001. Note that by adding 60 to the above serial numbers first two digits you get the year 85.

Table 7-10.
Sample Report Security Results for an Exchange Module

Byte No.	Data (hex)	Description
1	10	Header
2	04	Part of product number 46084
3	B4	Part of product number 46084
4	31	US ASCII character "1" which is part of the product number 460841
5	61	Part of serial number
6	B0	Part of serial number
7	03	Part of serial number
8	0F	Part of serial number
9	41	Country of Manufacturing Code

Since the sample is an exchange module, the exchange part number transmitted is 460841. Byte 4 is the hexadecimal value of 31 which represents the US ASCII character "1". Note, the prefix number 46084 does not change from the sample of the product module and the character "1" is really an ASCII character. When the number is displayed, the character string "-69901" should be inserted into the part number at the appropriate place.

Sample Report Security Program

This program returns information upon receiving a Report Security command for a Product Module. The data is based on the data format described at the beginning of the section, "Report Security Code (HILSC)." Byte 1 is the first byte sent from the module to the host.

The sample results given are based on the product number 46084A and serial number 2519A00093. The serial number corresponds with the year of 1985, week 19, and serial number suffix 00093. Note that by adding 60 to the above serial numbers first two digits you get the year 85.

The sample program is as follows:

```
#include <sys/hilioctl.h>
#define BUF_SIZ      15
#define MAX_DEV      '7'

unsigned char   dev_name[] = { "/dev/hil1" };

main()
{
  extern int    errno;
  int           index, hil_fd, product_no, serial_no,
                serial_hi, serial_lo;
  unsigned char hil_info[BUF_SIZ], product_let, country;

  while (dev_name[8] <= MAX_DEV)          /* device search loop */
  {
     printf("\n");
     if((hil_fd = open(dev_name,0)) > 0)
     {                                    /* open device file. */
       printf("%s -",dev_name);           /* print device file name. */
       if(ioctl(hil_fd,HILID,hil_info) < 0)
         printf("ioctl error %d\n",errno); /* invalid ioctl call. */
       else
       {
         for(index = 0; index < BUF_SIZ; printf("%.2X",hil_info[index++]))
         ;                                /* print id desc info. */
         printf("\n");
         if((hil_info[1] & ~(~04)) == 04)
         {                                /* if dev reports sec code. */
           printf("     ID NUMBER: ");
           if(ioctl(hil_fd,HILSC,hil_info) < 0) /* return sec code. */
             printf("ioctl error %d\n",errno);  /* invalid ioctl call. */
           else
           {
             for(index = 0; index < BUF_SIZ - 6; printf(" %.2X",
```

```
35                  hil_info[index++]))  /* print security code. */
36              ;
37              printf("\n");
38              product_no = hil_info[1] | hil_info [2] << 8 |
39                           (hil_info[3] >> 7) << 16;
40              product_let = hil_info[3] & 0x7f;
41              serial_no = hil_info[4] | hil_info[5] << 8 |
42                          hil_info[6] << 16 | (hil_info[7] & 0x3f) << 24;
43              serial_hi = serial_no/100000;
44              serial_lo = serial_no - serial_hi * 100000;
45              country   = hil_info[8] & 0x7f;
46              printf("     product number = %u%c",product_no,product_let);
47              printf("     serial number = %u%c%.5u\n",
48                           serial_hi,country,serial_lo);
49            }
50          }
51        }
52       close(hil_fd);
53      }
54      else printf("%s - open error: %d\n",dev_name,errno);
55                                          /* unable to open dev file. */
56      ++(dev_name[8]);           /* select next device file. */
57   }
58 }
```

The following is an explanation of the program:

Line 1 provides the `include` file (`sys/hilioctl.h`) which contains a list of macros that execute specific functions when used within a C Language program. HILID and HILSC are two macros which are included in this file and are used in this program. The path name for the `hilioctl.h` file is `/usr/include/sys/hilioctl.h`. If you execute the `cat` or `more` command on this path name, you will receive a screen listing of this file's contents.

Lines 2 and *3* define the constants BUF_SIZ and MAX_DEV. BUF_SIZ is assigned the value of 15 and MAX_DEV is assigned the value of 7.

Line 5 assigns the character string "`/dev/hil1`" to the unsigned character array `dev_name`.

Lines 9 through *12* declare the variables used in the program. These variables are defined as follows:

- `errno` is the external variable that receives the error number if there is an invalid `ioctl` call made.
- `index` is the control variable used in the `for` loops of *lines 25* and *35*.
- `hil_fd` is the file descriptor used in the `ioctl` parameter fields.
- `product_no` is the product number of the ID Module being used.
- `serial_no` is the serial number of the ID Module being used.
- `serial_hi` is the higher order set of bits being manipulated to interpret the serial number.
- `serial_lo` is the lower order set of bits being manipulated to interpret the serial number.
- `hil_info` is the unsigned character array which is assigned information return from the HP-HIL device.
- `product_let` is the product letter suffixed to the product number of the HP-HIL ID Module.
- `country` is the unsigned character which designates the country of the manufacturing code for the HP-HIL ID Module.

Line 14 is a HP-HIL device search loop (while loop) which tries to open an HP-HIL device file. If the device file can be opened it continues with the remainder of the while loop; otherwise, it prints an error message (*line 54*) and increments the eighth element (*line 56*) of the character string `dev_name` and searches for a new device.

Line 17 is an IF construct which tries to open the file `dev_name`. If the file can be opened the variable `hil_id` is assigned the file descriptor returned from opening the file. If the `dev_name` can not be opened then an error message (*line 54*) is displayed and the eighth element of the character array `dev_name` is incremented by one (*line 56*). The while loop then searches for the next HP-HIL device.

Line 19 prints the device file name.

Line 20 checks for a valid `ioctl` call. If it is valid the program continues from the `else` statement with the next set of instructions; otherwise, an error message is displayed (*line 21*).

Line 24 is a `for` loop which displays the describe record information after the label created by *line 19*.

Line 27 test to see if the device is an ID Module if it is not an ID Module then nothing is printed and the *while* loop searches for the next device. However, if the device is an ID Module then the program continues by making an `ioctl` call using the macro HILSC.

Line 29 displays the label `ID NUMBER`.

Line 30 test for a valid `ioctl` call. If the call made is invalid then an error message is displayed (*line 31*). If the call made was valid then the program continues.

Lines 34 and *35* are a `for` loop which displays the hexadecimal value of the Security Code after the `ID NUMBER` label.

Lines 37 through *48* make up the remainder of the `if-then-else` statement. Their purpose is to unscramble the Security Code information by performing several bit manipulations on the data. The result of all this bit manipulation is a readable product number and serial number being displayed.

A sample display would appear as follows assuming that an ID Module is the first device in the HP-HIL link, an Extended Keyboard is the second device in the HP-HIL link, and an HP Mouse is the third device in the HP-HIL link. Note that the remaining devices shown in the display indicate an open error. This is because these devices did not exist in the HP-HIL link.

```
/dev/hil1 -34 04 00 00 00 00 00 00 00 00 00 00 00 00 00
     ID NUMBER: 10 04 B4 41 BD B0 03 0F 41
     product number = 46084A   serial number = 2519A00093

/dev/hil2 -DF 00 00 00 00 00 00 00 00 00 00 00 00 00 00

/dev/hil3 -68 12 C2 1E 02 00 00 00 00 00 00 00 00 00 00

/dev/hil4 - open error: 6

/dev/hil5 - open error: 6

/dev/hil6 - open error: 6

/dev/hil7 - open error: 6
```

The hexadecimal information displayed after each of the device file names shown above can be interpreted by reading the section entitled, "Identify and Describe Command (HILID)."

Disable Keyswitch Auto-repeat (HILDKR)

This command is used to disable the "repeating keys" feature in the addressed device, reducing returned data to one report per keyswitch transition.

Enable Keyswitch Auto-repeat 1 and 2 (HILER1 and HILER2)

These two commands are used to enable the "repeating keys" feature in the addressed device (if the feature is supported). Generally keys repeat at the rate of one report every 1/30 of a second (based on a system poll rate of 1/60 of a second). Some keys, termed "Modifier" keys, will not repeat, while based on the opcode of the Enable Keyswitch Auto-repeat command the Cursor Keys (cursor left, right, up and down) will repeat at either 1/30 of a second interval

(opcode 3Eh) or 1/60 of a second intervals (opcode 3Fh). Most keys "repeat" by generating repeated down transitions corresponding to the key position being repeated, although repeating cursor keys report a keycode of 02h.

Prompt 1 through Prompt 7 (HILP1 through HILP7)

These commands are used to provide an audible or visual stimulus to the user, perhaps indicating that the system is ready for a particular type of input. Although intended to be directly associated with Acknowledge 1 through Acknowledge 7 and Button 1 through Button 7, this association is not a requirement.

The Prompts supported by the device are indicated in the Describe Record, and all unsupported Prompts will be treated the same as other unsupported commands.

Prompt (HILP)

Intended as a general-purpose stimulus to the user, Prompt is not intended to be associated with a particular Button as are Prompt 1 through Prompt 7. A device indicates support of Prompt in the Describe Record.

Acknowledge 1 through Acknowledge 7 (HILA1 through HILA7)

These commands, though similar to the Prompt 1 through Prompt 7 commands, are intended to provide an audio or visual response to the user, and are generally directly associated with the corresponding Prompt and Button of the same number, although this is not a requirement. Unsupported Acknowledge commands are ignored. Since there is no explicit "Prompt Off" function provided, this functionality may be part of the Acknowledge definition for a particular device, if required.

Acknowledge (HILA)

This command is similar to Prompt, however, Acknowledge is not associated with any particular Button, but merely as a general purpose audio or visual response to the user.

Keycode Set 1

This section provides the table, "Keycode Set 1".

Table 7-11. Keycode Set 1

Keycode for Transition (hex)		United States Legend		
Down	**Up**	**Unshifted**		**Shifted**
00h	01h	5		+ Char
02h	03h	<Repeat Cursor>		
04h	05h	Extend Char	(right)	
06h	07h	Extend Char	(left)	
08h	09h	Shift	(right)	
0Ah	0Bh	Shift	(left)	
0Ch	0Dh	CTRL		
0Eh	0Fh	Break		Reset
10h	11h	4	(keypad)	
12h	13h	8	(keypad)	
14h	15h	5	(keypad)	
16h	17h	9	(keypad)	
18h	19h	6	(keypad)	
1Ah	1Bh	7	(keypad)	
1Ch	1Dh	.	(keypad)	
1Eh	1Fh	Enter	(keypad)	
20h	21h	1	(keypad)	
22h	23h	/	(keypad)	
24h	25h	2	(keypad)	
26h	27h	+	(keypad)	
28h	29h	3	(keypad)	
2Ah	2Bh	*	(keypad)	

Continued on next page ...

Note that Transition Keycode 02h is reserved for Cursor Repeat.

Table 7-11. Keycode Set 1 (continued)

Keycode for Transition (hex)		United States Legend		
Down	**Up**	**Unshifted**		**Shifted**
2Ch	2Dh	0	(keypad)	
2Eh	2Fh	-	(keypad)	
30h	31h	B		
32h	33h	V		
34h	35h	C		
36h	37h	X		
38h	39h	Z		
3Ah	3Bh	<NOT LOADED>	(left)	
3Ch	3Dh	<NOT USED>		
3Eh	3Fh	ESC		
40h	41h	6		- Char
42h	43h	<blank/f10>	(keypad)	
44h	45h	3	(keypad)	Prev
46h	47h	<blank/f11>	(keypad)	
48h	49h	.	(keypad)	
4Ah	4Bh	<blank/f9>	(keypad)	
4Ch	4Dh	Tab >|	(keypad)	|<
4Eh	4Fh	<blank/f12>	(keypad)	
50h	51h	H		
52h	53h	G		
54h	55h	F		
56h	57h	D		
58h	59h	S		
5Ah	5Bh	A		
5Ch	5Dh	<NOT USED>		

Continued on next page ...

Note that the <NOT LOADED> key positions are located next to Extend Char, *below* Shift, *and are generally covered by non-operative filler keys.*

Table 7-11. Keycode Set 1 (continued)

Keycode for Transition (hex)		United States Legend	
Down	**Up**	**Unshifted**	**Shifted**
5EH	5Fh	Caps	
60h	61h	U	
62h	63h	Y	
64h	65h	T	
66h	67h	R	
68h	69h	E	
6Ah	6Bh	W	
6Ch	6Dh	Q	
6Eh	6Fh	Tab >|	|<
70h	71h	7	&
72h	73h	6	^
74h	75h	5	%
76h	77h	4	$
78h	79h	3	#
7Ah	7Bh	2	@
7Ch	7Dh	1	!
7Eh	7Fh	`	~
80h	81h	<BUTTON 1>	
82h	83h	<BUTTON 2>	
84h	85h	<BUTTON 3>	
86h	87h	<BUTTON 4>	
88h	89h	<BUTTON 5>	
8Ah	8Bh	<BUTTON 6>	
8Ch	8Dh	<BUTTON 7>	
8Eh	8Fh	<PROXIMITY IN/OUT>	
90h	91h	Menu	
92h	93h	f4	
94h	95h	f3	

Continued on next page ...

Table 7-11. Keycode Set 1 (continued)

Keycode for Transition (hex)		United States Legend	
Down	**Up**	**Unshifted**	**Shifted**
96h	97h	f2	
98h	99h	f1	
9Ah	9Bh	8	+ Line
9Ch	9Dh	Stop	
9Eh	9Fh	Enter	Print
A0h	A1h	System	User
A2h	A3h	f5	
A4h	A5h	f6	
A6h	A7h	f7	
A8h	A9h	f8	
AAh	ABh	9	- Line
ACh	ADh	Clear line	
AEh	AFh	Clear display	
B0h	B1h	8	*
B2h	B3h	9	(
B4h	B5h	0	)
B6h	B7h	-	_
B8h	B9h	=	+
BAh	BBh	Back space	
BCh	BDh	Insert line	
BEh	BFh	Delete line	
C0h	C1h	I	
C2h	C3h	O	
C4h	C5h	P	
C6h	C7h	[	{
C8h	C9h	]	}
CAh	CBh	\	|
CCh	CDh	Insert char	

Continued on next page ...

Table 7-11. Keycode Set 1 (continued)

Keycode for Transition (hex)		United States Legend	
Down	**Up**	**Unshifted**	**Shifted**
CEh	CFh	Delete char	
D0h	D1h	J	
D2h	D3h	K	
D4h	D5h	L	
D6h	D7h	;	:
D8h	D9h	`	"
DAh	DBh	Return	
DCh	DDh	<home cursor>	
DEh	DFh	Prev	
E0h	E1h	M	
E2h	E3h	,	<
E4h	E5h	.	>
E6h	E7h	/	?
E8h	E9h	<NOT USED>	
EAh	EBh	Select	
ECh	EDh	<NOT USED>	
EEh	EFh	Next	
F0h	F1h	N	
F2h	F3h	<space bar>	
F4h	F5h	.	
F6h	F7h	<NOT LOADED> (right)	
F8h	F9h	<left cursor>	
FAh	FBh	<down cursor>	
FCh	FDh	<up cursor>	
FEh	FFh	<right cursor>	

Continued on next page ...

Note that the <NOT LOADED> key positions are located next to [Extend Char], *below* [Shift], *and are generally covered by non-operative filler keys.*

Index

F

H

W

Appendixes

A

Series 300/400 Dependencies

The following information, specific to Series 300/400 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.
- Hardware effects on DIL subroutines.
- Techniques for improving data transfer performance when using DIL subroutines.

Location of the DIL Subroutines

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

A

The GPIO Interface on Series 300/400 Computers

The **GPIO** (General Purpose Input/Output) interface is a very flexible parallel interface that allows communication with a variety of devices. On Series 300/400 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 Administration Tasks* manual for your system. The following discussion points out specific requirements needed for a special file associated with a given HP-IB, GPIO, or Centronics-compatible Parallel 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 major number for the HP-IB, GPIO, and Parallel interfaces is 21.

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, GPIO, or Parallel 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 or Parallel interface), the component of the minor number is ignored.

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

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

Creating an HP-IB Interface File

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

- The pathname is `/dev/raw_hpib`.
- The major number is 21.
- The card's select code switches are set to select code 2 (that is, 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
```

A

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

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

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

Creating a GPIO Interface File

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

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

Assuming that you have set the interface select code switches to 04 on the Series 300/400 GPIO card, the following `mknod` command will create the appropriate special file, named `/dev/raw_gpio`:

```
mknod  /dev/raw_gpio  c  21  0x040000
```

Creating a Centronics-compatible Parallel Interface File

If you have a Centronics-compatible interface that you want to access with the DIL subroutines on the same computer, here's what you do.

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

Assuming that you have a Parallel interface at select code 05 on the Series 300/400 system, the following `mknod` command will create the appropriate special file, named `/dev/parallel`:

```
mknod  /dev/parallel  c  21  0x050000
```

Entity Identifiers

On Series 300/400 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`.

Hardware Effects on DIL Subroutines

This section presents characteristics of the DIL subroutines specific to Series 300/400 computers. These dependencies are organized under the routine to which they apply. The subroutines are presented in alphabetical order.

hpib_send_cmnd

By default, the Series 300/400 HP-IB interface card uses odd parity when you send commands via `hpib_send_cmnd`. To do this, it overwrites the most-significant bit of each command byte with a parity bit. This should not cause a problem since all HP-IB commands use only 7 bits, and the eighth is free for use as a parity bit. The behavior of `hpib_send_cmnd` can be modified to use all eight bits for commands using the `hpib_parity_ctl` subroutine.

hpib_status

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

io_get_term_reason

For the GPIO interface, PSTS is checked only at the beginning of a transfer. An interrupt caused by an EIR will also terminate a transfer. The value of the termination reason in this case is also 4.

For the Centronics-compatible Parallel interface, a termination reason value of 4 indicates that the transfer terminated because the peripheral asserted the ACK line.

io_on_interrupt

For the HP 98622 GPIO interface, only the EIR interrupt is available.

For the HP 98265A/B HP-IB interface, the IFC and GET interrupts are not available.

A

io_reset

Interface self-test is not supported.

When an HP-IB interface is reset via `io_reset`, Remote Enable (REN) is cleared, 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 Interface Clear (IFC) line is pulsed (if system controller), the card is put on line, and REN is reset (if system controller).

When a GPIO interface is reset, the peripheral request line is pulsed 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, and the Peripheral Reset (PRESET) line is pulsed.

io_speed_ctl

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

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

---

io_timeout_ctl

This routine allows you to set a time limit for I/O operations on an entity identifier associated with an interface file. The timeout value that you specify is a 32-bit long integer that indicates the length of the timeout in microseconds. However, the resolution of the effective timeout is system-dependent. On the Series 300/400 computers the timeout is rounded up to the nearest 20-millisecond boundary. For example, if you specify a timeout of 150000 microseconds (150 milliseconds), the effective timeout is rounded up to 160 milliseconds.

Performance Tips

Device I/O performance on Series 300/400 computers using DIL subroutines can be improved by following these guidelines:

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

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

- Use auto-addressing for all read and write operations (refer to the "setting up talkers and listeners" index entry under "Active Controller.")
- Use *rtprio*(2) to increase the system priority of an I/O process. `rtprio` requires that the process be running with an effective user ID of super-user. The real-time priorities available with `rtprio` are non-degrading priorities. Be careful when using real-time priorities. Increasing I/O process priorities above system processes may cause undesirable behavior. For example, requesting a real-time priority in the range of 0-63 places your process at a

higher priority than the DIL interrupt handler system process. This means that interrupts could be lost if available CPU resources are insufficient. The following example places the calling process at the lowest (least important) real-time priority:

A

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

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

B

Series 600/800 Dependencies

The following information, specific to the Device I/O Library (DIL) on Series 600/800 computers, is discussed in this appendix:

- Compiling programs that use DIL subroutines.
- 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 subroutines affected by the Series 600/800 hardware.
- DIL support of HP-IB auto-addressed files.
- Improving performance of DIL programs.

Compiling Programs That Use DIL

The DIL subroutines 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 600/800 kernel is shipped with a default I/O configuration. This means a default set of special files is made for you. For example, the `/dev/hpib` directory contains special files created for use with HP-IB instruments connected to the HP 27110B HP-IB interface. The special file `/dev/gpio0` is created for use with instruments or peripherals connected to the HP27114A Asynchronous FIFO interface (AFI).

The `insf` command is used to install these special files all at one time. `Mknod` could also be used to create them one at a time. For more information on `insf` and `mknod` refer to the *HP-UX Reference*.

Major Numbers

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

Minor Numbers and Logical Unit Numbers

Drivers use minor numbers to map the hardware I/O cards to their locations in the Series 600/800 I/O backplane. The default I/O configuration shipped with your Series 600/800 creates special files accessing a subset of the available backplane slots. For the HP-IB card, two slots are available for instrument I/O, and one slot is available for the AFI card. Slot information is accessed through the device's **logical unit** number. The logical unit number is mapped into the special file's minor number. For HP-IB special files, the HP-IB bus address is also mapped into the minor number.

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

```
0x00LuBa
```

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

The minor number syntax for an AFI special file is:

```
0x00Lu    00
```

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

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

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

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

Listing Special Files

The Series 600/800 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 600/800 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/`*XaY*, 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, GPIO, or Centronics-compatible Parallel interfaces 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 600/800, 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 Subroutines

The HP-IB card supported on the Series 600/800 is the HP 27110B HP-IB interface; the GPIO card is the HP 27114A Asynchronous FIFO Interface (AFI).

B

This section presents some restrictions on using the DIL subroutines on Series 600/800 computers. These restrictions are organized under the DIL subroutine to which they apply. The subroutines 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` subroutine 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 "requesting service" index entry under "Non-Active Controller.")

hpib_io

The atomicity of `hpib_io` calls is not guaranteed.

hpib_atn_ctl, hpib-address_ctl, hpib_parity_ctl

These subroutines are not currently supported on the Series 600/800.

io_eol_ctl

The AFI driver does not support pattern matching on reads; all `io_eol_ctl` calls return -1 and set `errno` to `EINVAL`.

io_reset

When an HP-IB interface is reset via `io_reset`, the card's parallel poll response is set to 0; its serial poll response is set to 128; its HP-IB address is read off the hardware switches; and the card is put on-line. Any enabled interrupts are preserved. If the card is configured as system controller, then Interface Clear (IFC) is pulsed and Remote Enable (REN) is asserted.

When an AFI interface is reset via `io_reset`, each of the three control output lines is reset to zero, the incoming Attention Request (ARQ) is disabled, the ARQ flip flop is cleared, the ARQ enable flip flop and the handshake to the peripheral are disabled, and the FIFO buffer is flushed out.

io_speed_ctl

DMA is the only supported transfer method.

io_timeout_ctl

On Series 600/800 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 600/800 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") 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 referenced in the "auto-addressing" index entry under "Active Controller," 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 600/800, 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 600/800. Please keep this in mind when developing programs which take advantage of this new functionality.

Table B-1 shows which DIL functions are supported on auto-addressed files. Entries in the first column work the same on both auto-addressed and non-auto-addressed (also called **raw bus**) files. Entries in the second column are somewhat different for auto-addressed files; entries in the third column are

not supported on HP-IB auto-addressed files and will return an error indication if used.

Table B-1. DIL Auto-addressed Support

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

Those functions in the second column, which operate differently on raw bus and auto-addressed special files, are discussed below.

hpib_card_ppoll_resp

Calling `hpib_card_ppoll_resp` on an auto-addressed file does not configure the HP-IB interface card; rather, it configures the device associated with the file with the appropriate addressing and Parallel Poll configuration commands.

B

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 600/800 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.

B

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 600/800 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.

B

Setting Up Faster Data Transfers

Because of the I/O architecture of the Series 600/800, data transfers run more efficiently if your data buffers are aligned on a page boundary. The number of bytes per page is defined as **NBPG** and can be referenced by including `<sys/param.h>`. The following example shows how to allocate and page-align a data buffer:

```
#include <sys/param.h> /* defines NBPG and roundup(x, y)         */
#define REAL_SIZE 1024 /* amount of memory we want to page-align */
char *malloc();

main() {
    char *malloc_ptr, *align_ptr;

    :
    malloc_ptr = malloc(NBPG + REAL_SIZE);   /* allocate memory       */
    align_ptr  = roundup(malloc_ptr, NBPG);  /* and round up the ptr  */
                 /* in future data transfers, use align_ptr       */
    :
    free(malloc_ptr);  /* when we're done with the data */
}
```

In addition, even count transfers run more quickly than odd count transfers.

C

ASCII Character Codes

This appendix contains two tables:

- Table C-1 lists ASCII control characters and how to obtain them by pressing the specified key while holding the Ctrl key or Ctrl and Shift keys down.
- Table C-2 lists all ASCII characters with their decimal, binary, octal, and hexadecimal equivalent values as well as their corresponding HP-IB name.

Table C-1. Obtaining ASCII Control Characters

Keys	ASCII	Dec	Oct	Hex	Keys	ASCII	Dec	Oct	Hex
Ctrl-Shift-@	NUL	00	000	00	Ctrl-P	DLE	16	020	10
Ctrl-A	SOH	01	001	01	Ctrl-Q	DC1	17	021	11
Ctrl-B	STX	02	002	02	Ctrl-R	DC2	18	022	12
Ctrl-C	ETX	03	003	03	Ctrl-S	DC3	19	023	13
Ctrl-D	EOT	04	004	04	Ctrl-T	DC4	20	024	14
Ctrl-E	ENQ	05	005	05	Ctrl-U	NAK	21	025	15
Ctrl-F	ACK	06	006	06	Ctrl-V	SYNC	22	026	16
Ctrl-G	BEL	07	007	07	Ctrl-W	ETB	23	027	17
Ctrl-H	BS	08	010	08	Ctrl-X	CAN	24	030	18
Ctrl-I	HT	09	011	09	Ctrl-Y	EM	25	031	19
Ctrl-J	LF	10	012	0A	Ctrl-Z	SUB	26	032	1A
Ctrl-K	VT	11	013	0B	Ctrl-[	ESC	27	033	1B
Ctrl-L	FF	12	014	0C	Ctrl-\	FS	28	034	1C
Ctrl-M	CR	13	015	0D	Ctrl-]	GS	29	035	1D
Ctrl-N	SO	14	016	0E	Ctrl-Shift-^	RS	30	036	1E
Ctrl-O	SI	15	017	0F	Ctrl-Shift-_	US	31	037	1F

C

Table C-2. ASCII Character Codes

ASCII	Dec	Binary	Oct	Hex	HP-IB	ASCII	Dec	Binary	Oct	Hex	HP-IB
NUL	00	00000000	000	00		space	32	00100000	040	20	LA0
SOH	01	00000001	001	01	GTL	!	33	00100001	041	21	LA1
STX	02	00000010	002	02		"	34	00100010	042	22	LA2
ETX	03	00000011	003	03		#	35	00100011	043	23	LA3
EOT	04	00000100	004	04	SDC	$	36	00100100	044	24	LA4
ENQ	05	00000101	005	05	PPC	&	37	00100101	045	25	LA5
ACK	06	00000110	006	06		%	38	00100110	046	26	LA6
BEL	07	00000111	007	07		'	39	00100111	047	27	LA7
BS	08	00001000	010	08	GET	(	40	00101000	050	28	LA8
HT	09	00001001	011	09	TCT	)	41	00101001	051	29	LA9
LF	10	00001010	012	0A		*	42	00101010	052	2A	LA10
VT	11	00001011	013	0B		+	43	00101011	053	2B	LA11
FF	12	00001100	014	0C		,	44	00101100	054	2C	LA12
CR	13	00001101	015	0D		-	45	00101101	055	2D	LA13
SO	14	00001110	016	0E		.	46	00101110	056	2E	LA14
SI	15	00001111	017	0F		/	47	00101111	057	2F	LA15
DLE	16	00010000	020	10		0	48	00110000	060	30	LA16
DC1	17	00010001	021	11	LLO	1	49	00110001	061	31	LA17
DC2	18	00010010	022	12		2	50	00110010	062	32	LA18
DC3	19	00010011	023	13		3	51	00110011	063	33	LA19
DC4	20	00010100	024	14	DCL	4	52	00110100	064	34	LA20
NAK	21	00010101	025	15	PPU	5	53	00110101	065	35	LA21
SYNC	22	00010110	026	16		6	54	00110110	066	36	LA22
ETB	23	00010111	027	17		7	55	00110111	067	37	LA23
CAN	24	00011000	030	18	SPE	8	56	00111000	070	38	LA24
EM	25	00011001	031	19	SPD	9	57	00111001	071	39	LA25
SUB	26	00011010	032	1A		:	58	00111010	072	3A	LA26
ESC	27	00011011	033	1B		;	59	00111011	073	3B	LA27
FS	28	00011100	034	1C		<	60	00111100	074	3C	LA28
GS	29	00011101	035	1D		=	61	00111101	075	3D	LA29
RS	30	00011110	036	1E		>	62	00111110	076	3E	LA30
US	31	00011111	037	1F		?	63	00111111	077	3F	UNL

Table C-2. ASCII Character Codes (continued)

ASCII	Dec	Binary	Oct	Hex	HP-IB	ASCII	Dec	Binary	Oct	Hex	HP-IB
@	64	01000000	100	40	TA0	`	96	01100000	140	60	SC0
A	65	01000001	101	41	TA1	a	97	01100001	141	61	SC1
B	66	01000010	102	42	TA2	b	98	01100010	142	62	SC2
C	67	01000011	103	43	TA3	c	99	01100011	143	63	SC3
D	68	01000100	104	44	TA4	d	100	01100100	144	64	SC4
E	69	01000101	105	45	TA5	e	101	01100101	145	65	SC5
F	70	01000110	106	46	TA6	f	102	01100110	146	66	SC6
G	71	01000111	107	47	TA7	g	103	01100111	147	67	SC7
H	72	01001000	110	48	TA8	h	104	01101000	150	68	SC8
I	73	01001001	111	49	TA9	i	105	01101001	151	69	SC9
J	74	01001010	112	4A	TA10	j	106	01101010	152	6A	SC10
K	75	01001011	113	4B	TA11	k	107	01101011	153	6B	SC11
L	76	01001100	114	4C	TA12	l	108	01101100	154	6C	SC12
M	77	01001101	115	4D	TA13	m	109	01101101	155	6D	SC13
N	78	01001110	116	4E	TA14	n	110	01101110	156	6E	SC14
O	79	01001111	117	4F	TA15	o	111	01101111	157	6F	SC15
P	80	01000000	120	50	TA16	p	112	01110000	160	70	SC16
Q	81	01000001	121	51	TA17	q	113	01110001	161	71	SC17
R	82	01000010	122	52	TA18	r	114	01110010	162	72	SC18
S	83	01000011	123	53	TA19	s	115	01110011	163	73	SC19
T	84	01010100	124	54	TA20	t	116	01110100	164	74	SC20
U	85	01010101	125	55	TA21	u	117	01110101	165	75	SC21
V	86	01010110	126	56	TA22	v	118	01110110	166	76	SC22
W	87	01010111	127	57	TA23	w	119	01110111	167	77	SC23
X	88	01011000	130	58	TA24	x	120	01111000	170	78	SC24
Y	89	01011001	131	59	TA25	y	121	01111001	171	79	SC25
Z	90	01011010	132	5A	TA26	z	122	01111010	172	7A	SC26
[	91	01011011	133	5B	TA27	{	123	01111011	173	7B	SC27
\	92	01011100	134	5C	TA28	|	124	01111100	174	7C	SC28
]	93	01011101	135	5D	TA29	}	125	01111101	175	7D	SC29
^	94	01011110	136	5E	TA30	~	126	01111110	176	7E	SC30
_	95	01011111	137	5F	UNT	DEL	127	01111111	177	7F	SC31

D

DIL Programming Example

This appendix contains a program listing for an HP-IB driver that uses Device I/O Library subroutines to drive various models of Hewlett-Packard Amigo protocol HP-IB printers. It is provided solely for illustrative use, and is not to be construed as optimum programming technique nor necessarily totally bug-free although the program has been extensively tested.

It contains not only examples of DIL subroutine usage, but also other useful programming techniques and structures that can make the task of writing specialized I/O programs much easier.

```
/*************************************************************************/
/* This example Amigo printer driver uses a byte stream as standard      */
/* input and Amigo protocol as output to HP-IB driver (21). Any special  */
/* character handling should be done by a filter that feeds this driver. */
/*                                                                       */
/* This example program is provided for solely illustrative purposes to  */
/* demonstrate typical use of Device I/O Library (DIL) subroutines.  No  */
/* representations are made as to its suitability for any given          */
/* application.                                                          */
/*                                                                       */
/* While the program is intended to show good programming practice, it   */
/* does not necessarily represent optimum programming efficiency.        */
/*************************************************************************/

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/sysmacros.h>

```

```
/* 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 */

```

```
142   /* GET USER SUPPLIED OPTIONS AND PRINTER FILE NAME */
143   while ((i = getopt(argc, argv, "b:t:p:D")) != EOF) {
144    switch (i) {
145    /* set the buffer size to output to printer */
146    case 'b': if ((bufsz = atoi(optarg)) <= 0) errflg++;
147     break;
148
149    /* get the new timeout value in seconds */
150    case 't': if ((timeout = atoi(optarg)) < 0) errflg++;
151     break;
152
153    /* Set the parallel poll pmask (mostly for debugging) */
154    case 'p': if ((pmask[0] = atoi(optarg)) < 0) errflg++;
155     break;
156
157    case 'D': Debug++; break;
158
159    case '?': errflg++;break;
160    }
161   }
162   /* get printer dev if supplied */
163   if (optind < argc)
164    strcpy(ptr_dev, argv[optind]);
165
166   if (errflg) {
167  fprintf(stderr, "usage: %s [-bbufsz -ttmout] [printer_dev]\n", procnam);
168  fprintf(stderr, "-b bufsz > Output buf size to printer (%d)\n", bufsz);
169  fprintf(stderr, "-t tmout > Max seconds to output buffer (%d)\n",
timeout);
170  fprintf(stderr, "printer_dev > Printer device file     (%s)\n", ptr_dev);
171  fprintf(stderr, "-p ppoll_mask > Parallel poll mask
(0x%02x)\n",pmask[0]);
172    exit(2);
173   }
174   /* get memory for the output buffer */
175   outbuf = (unsigned char *)malloc (bufsz + 4);
176  /*
177    NOTE: Printer device file (/dev/lp) is used only to get printer select
178    code and HP-IB bus address.  This is because attention-true (ATN)
179    requests can only be sent to an "HP-IB raw bus device file".  Therefore
180    after getting the SC and BA we will use a "HP-IB raw bus device file" to
181    do all the work, but it must exist with a name similar to the printer
182    device; i.e. "/dev/lp" is changed to "/dev/lp_07", where the "07" is the
183    select code.
184  */
```

```
185  /* check if printer device exists */
186  if (stat(ptr_dev, &statbuf) < 0)
187   fatal_err("stat", ptr_dev, F_EXIT);
188
189  /* check if it is a character device file */
190  if ((statbuf.st_mode & S_IFMT) != S_IFCHR)
191   fatal_err("Must be a char_special file", ptr_dev, F_EXIT);
192
193  /* extract selectcode from the printer device */
194  selcode = m_selcode(statbuf.st_rdev);
195
196  /* make the HP-IB raw bus device file name from selectcode */
197  ptr_raw[1] += selcode / 16;
198  ptr_raw[2] += selcode % 16;
199  if ((selcode % 16) >= 10) ptr_raw[2] += ('a' - '0' -10);
200  strcat(ptr_dev, ptr_raw);
201
202  /* get device BA from the printer device and config control bytes */
203  devba = m_busaddr(statbuf.st_rdev);
204  DLA = LAG_BASE + devba; /* device listen address */
205  DTA = TAG_BASE + devba; /* device talk address */
206
207  /* open the HP-IB raw bus device */
208  if ((eid = open(ptr_dev, O_RDWR)) <0) {
209   fatal_err("Raw HP-IB open", ptr_dev, F_RTRN);
210 fprintf(stderr,
211 " The following commands executed as a super user may be necessary\n\n");
212 fprintf(stderr, "     # mknod %s c 21 0x%s1f00\n", ptr_dev, &ptr_raw[1]);
213 fprintf(stderr, "     # chmod 555 %s\n", ptr_dev);
214 fprintf(stderr, "     # chown lp  %s\n", ptr_dev);
215   exit(2);
216  }
217  /* get (my) BA of the controller and configure control bytes */
218  if ((myba = hpib_bus_status(eid, 7)) < 0)
219   fatal_err("Must be raw hpib driver (21)", ptr_dev,F_EXIT);
220  MLA = LAG_BASE + myba; /* controller (my) listen address */
221  MTA = TAG_BASE + myba; /* controller (my) talk address */
222
223  /* go do the Amigo identify */
224  ptr_type = amigo_identify();
225
```

```
226  if (Debug) {
227   printf("%s Identified ", ptr_dev);
228   switch(ptr_type) {
229   case T2608A:  printf("2608A");  break;
230    case T2631A:  printf("2631A");  break;
231    case T2631B:  printf("2631B");  break;
232    case T2673A:  printf("2673A");  break;
233    case QjetPlus:  printf("QuietJet Plus");break;
234    case T2632A:  printf("2632A");  break;
235    case T2634A:  printf("2634A");  break;
236   default: printf("You forgot one dummy"); break;
237   }
238   printf(" printer\n");
239  }
240  /* set the timeout to user requested value */
241  if (io_timeout_ctl(eid, timeout * 1000000) < 0)
242   fatal_err("io_timeout_ctl", ptr_dev, F_EXIT);
243
244  /* always tag last output data byte with EOI */
245  if (hpib_eoi_ctl(eid, 1) < 0)
246   fatal_err("hpib_eoi_ctl", ptr_dev, F_EXIT);
247
248  /* clear out the status bits */
249  amigo_clear();
250
251  /* check the status bits */
252  status = amigo_status();
253  if (Debug) printf("%s Printer status = 0x%x\n", ptr_dev, status);
254
255  /* set the ppoll mask required by some printers */
256  amigo_set_pmask();
257
```

```
258   /* MAIN OUTPUT LOOP */
259   i = 0;
260   while ((c = getchar()) != EOF) {
261    if (i == bufsz) {
262     amigo_write(outbuf, i);
263     i = 0;
264    }
265    outbuf[i++] = c;
266   }
267   /* post remaining buffer */
268   if (i) amigo_write(outbuf, i);
269   exit(0);
270  }
271
272  /* ROUTINE TO DO THE MAIN I/O TO THE BUSS */
273  /* lock bus, do preamble, read/write, do postamble and unlock bus */
274  /* preamble must be 3 or 4 bytes, postamble must be 1 or 2 bytes */
275  int
276  HPIB_msg(rw_flag, pcm1, pcm2, pcm3, buffer, length, ocm0, ocm1)
277  int rw_flag;
278  int  pcm1;
279  int  pcm2;
280  int  pcm3;
281  char  *buffer;
282  int  length;
283  int  ocm0;
284  int  ocm1;
285  {
286   unsigned char pre_cmd[4];
287   unsigned char post_cmd[2];
288   int tlog = -1;
289
290   pre_cmd[0] = UNL;  /* always issue unlisten command first */
291   pre_cmd[1] = pcm1;
292   pre_cmd[2] = pcm2;
293   pre_cmd[3] = pcm3;
294
295   post_cmd[0] = ocm0;
296   post_cmd[1] = ocm1;
297
298   /* first get exclusive use of the bus */
299   if (io_lock(eid) < 0)
300    fatal_err("io_lock", ptr_dev, F_EXIT);
301
```

```
302  /* send the preamble 3 or 4 bytes with attention true */
303  if (hpib_send_cmnd(eid, pre_cmd, (pcm3 ? 4 : 3)) < 0)
304   fatal_err("hpib_send_cmnd preamble", ptr_dev, F_EXIT);
305
306  switch (rw_flag) {
307  case H_READ:
308   if ((tlog = read(eid, buffer, length)) < 0)
309    fatal_err("read", ptr_dev, F_EXIT);
310   break;
311
312  case H_WRITE:
313   if ((tlog = write(eid, buffer, length)) < 0)
314    fatal_err("write", ptr_dev, F_EXIT);
315   break;
316
317  case H_CMND:
318   return(0);
319  default:
320   return(-1);
321  }
322  /* send the postamble 1 or 2 bytes with attention true */
323  if (hpib_send_cmnd(eid, post_cmd, (ocm1 ? 2 : 1)) < 0)
324   fatal_err("hpib_send_cmnd postamble", ptr_dev, F_EXIT);
325
326  /* at last unlock the bus so other bus users can access it */
327  if (io_unlock(eid) < 0)
328   fatal_err("io_unlock", ptr_dev, F_EXIT);
329
330  return(tlog);
331 }
332
333 int
334 amigo_identify()
335 {
336  unsigned char identify[2];
337
```

```
338  /* TLK31 (UNT) is special for amigo identify */
339  /* finish with a MTA (UNT is not save for non-amigo devices) */
340  HPIB_msg(H_READ, MLA, UNT, SCG_BASE + devba, identify, 2, MTA, 0);
341
342  switch(identify[0]) {
343  case 32:
344   /* Amigo identify */
345   switch(identify[1]) {
346   case 1:  return(T2608A);
347   case 2:  return(T2631A);
348   case 9:  return(T2631B);
349   case 11: return(T2673A);
350   case 13: return(QjetPlus);
351   case 16: return(T2632A);
352   case 17: return(T2634A);
353   default:
354    printf("Unrecognized Amigo printer, ID2 = %d\n",
355     identify[1]); break;
356   }
357   break;
358  case 33:
359   if (identify[1] == 1)
360    printf("Ciper printer not supported yet!\n");
361   break;
362  default:
363 printf("Unrecognized Amigo Printer identify, ID1 = %d, ID2 = %d\n",
364    identify[0], identify[1]);
365   break;
366  }
367  exit(2);
368 }
369
370 /* set the parallel poll mask value */
371 amigo_set_pmask()
372 {
373  HPIB_msg(H_WRITE, MTA, DLA, PR_SEC_MASK, pmask, 1, UNL, 0);
374 }
375
```

```
/* 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);

```

```
417   /* a DSJ is required to remove the ppoll response from device */
418   if (dsj = amigo_dsj()) {
419    if (Debug) printf("%s DSJ = 0x%x\n", ptr_dev, dsj);
420
421    status = amigo_status();
422    if (Debug) printf("%s STATUS = 0x%x\n", ptr_dev, status);
423    goto again;
424   }
425  }
426
427  /* output error message and conditionally abort */
428  fatal_err(message, fname, flag)
429  char *message;
430  char *fname;
431  {
432   fprintf(stderr, "%s: Error - %s of %s ", procnam, message, fname);
433   if   (errno) perror("");
434   else fprintf(stderr, "\n");
435
436   if (flag == F_RTRN) return;
437   if (flag == F_EXIT) exit(2);
438   exit(3);
439  }
```

Master Index

Index

D

E

F

G

I

K

L

M

N

T

U

W