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

HP Part Number 97089-90052

HEWLETT
PACKARD

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

NOTICE

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

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

WARRANTY

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

Copyright 1986 Hewlett-Packard Company

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the Rights in Technical Data and Software clause in DAR 7-104.9(a).

Copyright 1980, 1984, AT&T, Inc.

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

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

Customer Note

Hewlett-Packard is in the process of changing the color of our documentation binders. In order to accomplish this changeover we are placing two spine inserts with this manual. Please use the insert that matches the binders you receive.

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

Printed in U.S.A.

Printing History

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

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

September 1986...Edition 1

Table of Contents

Interfacing Concepts

- Differences Between Computers ... 1
- DIL Tutorial Contents ... 2
- The DIL Interfacing Routines ... 3
  - Linking the DIL Routines ... 3
  - Calling the DIL Routines From Pascal ... 4
  - Calling the DIL Routines From FORTRAN ... 5
- General Interface Concepts ... 6
  - What Is an Interface? ... 6
  - Interface Functions ... 7
  - Additional Interface Functions ... 7
- The HP-IB Interface ... 8
  - General Structure ... 8
  - Handshake Lines ... 9
  - Bus Management Control Lines ... 11
- The GPIO Interface ... 12

General-Purpose Routines

- Concepts ... 13
  - The Interface Special File ... 13
  - The Entity Identifier (eid) ... 14
  - The Programming Model ... 14
  - General-Purpose Routines ... 14
- Opening an Interface's Special File ... 15
- Closing an Interface's Special File ... 17
- Reading and Writing ... 18
- Designing Error Checking Routines ... 19
  - The errno Variable ... 19
  - Using errno ... 20
- Resetting Interfaces ... 22
- Controlling I/O Parameters ... 23
  - Setting the I/O Timeout ... 23
  - Setting Data Path Width ... 25
  - Setting Transfer Speed ... 25
  - Setting the Read Termination Pattern ... 26
  - Removing a Read Termination Pattern ... 28

Determining Why a Read Terminated 29
Interrupts 31
    Interrupts on the Integral PC 31
    Interrupts on the Series 500 31

Controlling the HP-IB Interface
Overview of HP-IB Commands 36
Overview of the HP-IB DIL Routines 40
    Standard DIL Routines 40
    The Computer's Role on the HP-IB 41
Opening the HP-IB Interface File 42
Sending HP-IB Commands 43
The Active Controller 45
    Determining Active Controller 46
    Setting Up Talkers and Listeners 47
    Remote Control of Devices 50
    Locking Out Local Control 51
    Enabling Local Control 51
    Triggering Devices 52
    Transferring Data 52
    Clearing HP-IB Devices 54
    Servicing Requests 54
    Parallel Polling 57
    Waiting For a Parallel Poll Response 61
    Serial Polling 65
    Passing Control 67
The System Controller 68
    Determining System Controller 68
    System Controller's Duties 69
The Computer As a Non-Active Controller 71
    Determining the Controller's Status 71
    Requesting Service 73
    Responding to Parallel Polls 74
    Disabling Parallel-Poll Response 76
    Accepting Active Control 77
    Determining When You Are Addressed 79
Buffering I/O Operations 83
    Iodetail: The I/O Operation Template 84
    Allocating Space 87
    Example 88
    Locating Errors in Buffered I/O Operations 90

Controlling the GPIO Interface
Configuring Your GPIO Interface 93
Configuring the Integral PC GPIO 93
Setting the Interface Switches for Series 200/300 and 500 93
Default Configuration and Switch Settings for the Series 800 Model 840 GPIO 94
Creating the GPIO Interface File 94
Limitations on Controlling the Interface 95
Using the DIL Routines 96
Resetting the Interface 97
Performing Data Transfers 98
Using the Special-Purpose Lines 98
Controlling the Data Path Width 100
Controlling the Transfer Speed 101
Read Terminations 101
Interrupts 102
Interrupt-Driven Transfer Mode 102

Series 500 Dependencies
Location of the DIL Routines 103
The GPIO Interface 103
Data Lines 104
Handshake Lines 104
Special-Purpose Lines 104
Data Handshake Methods 104
Data-In Clock Source 105
Creating the Interface Special File 105
Creating an Interface File 105
Determining The Bus Address of the Interface Card 108
Effects of Resetting (via io_reset) 108
Entity Identifiers 108
Restrictions Using the DIL Routines 109
hpib_bus_status 109
hpib_card_ppoll_resp 110
hpib_rqst_srvce 110
hpib_send_cmnd 111
hpib_status_wait 111
hpib_wait_on_ppoll 111
io_get_term_reason 111
io_timeout_ctl 112
io_speed_ctl 112
io_width_ctl 112
Performance Tips 113

Series 200/300 Dependencies
Location of the DIL Routines 115
Linking DIL Routines 116
The GPIO Interface 116
Data Lines 116
Handshake Lines 117
Special-Purpose Lines 117
Data Handshake Methods 117
Data-In Clock Source 118
Creating the Interface Special File 119
Creating the Special File 119
Effects of Resetting (via io_reset) 122
Entity Identifiers 122
Restrictions Using the DIL Routines 123
hpib_io 123
hpib_send_cmnd 123
hpib_status 123
io_interrupt_ctl 123
io_on_interrupt 123
io_reset 124
io_speed_ctl 124
io_timeout_ctl 124
Performance Tips 125
Simulating Interrupts for the HP-IB Interface 126
Simulating Interrupts on the GPIO Interface 128

Integral PC Dependencies
Location of the DIL Routines 132
The GPIO Interface 132
Creating an Interface Special File 133
GPIO Interface Files 133
HP-IB Interface Files 133
Unloading the DIL Drivers 133
Interrupts 134
Controlling the HP-IB Interface 134
Limitations on the HP-IB Interface 134
The Computer as a Non-Active Controller 134
Non-Standard DIL Routines 135
General-Purpose Routines 135
Non-Standard HP-IB Routines 135
Non-Standard GPIO Routines 135

Restrictions Using the DIL Routines 136
hpib_bus_status 136
hpib_card_ppoll_resp 136
hpib_ppoll_resp_ctl 136
io_eol_ctl 136
io_reset 136
io_speed_ctl 137
io_timeout_ctl 137
io_width_ctl 138
open(2) 138
read(2) and write(2) 138

Series 800 Model 840 Dependencies
Compiling Programs That Use DIL 140
Accessing the Interface Special Files 140
Major Numbers 140
Minor Numbers and Logical Unit Numbers 141
Listing Special Files 142
Naming Conventions for Interface Special Files 143
Creating Interface Special Files 144
Hardware Effects on DIL Routines 145
hpib_rqst_srvce 145
io_eol_ctl 145
io_reset 145
io_speed_ctl 146
io_timeout_ctl 146
io_width_ctl 146
Return Values for Special Error Conditions 146
DIL Support of HP-IB Auto-Addressed Files 147
hpib_card_ppoll_resp 149
hpib_io 149
hpib_ren_ctl 149
hpib_send_cmd 149
hpib_spoll 149
hpib_wait_on_ppoll 149
io_on_interrupt 149

Performance Tips 150
Process Locking 150
Setting Real-Time Priority 151
Preallocating Disc Space 151
Reducing System Call Overhead 152
Setting Up Faster Data Transfers 152

Character Codes 153

Index 155

Interfacing Concepts 1

This tutorial illustrates how to access an arbitrary device through **HP-IB** (Hewlett-Packard Interface Bus) and **GPIO** (General Purpose Input/Output) interfaces on your HP-UX system using the routines in **DIL** (Device I/O Library). This tutorial covers general interfacing strategies, in addition to strategies designed specifically for HP-IB and GPIO interfaces.

The tutorial assumes you want to communicate with devices from within a program (process). All DIL routines can be called from C, Pascal, and FORTRAN programs. The examples, illustrating the use of the routines, are written in C; however, with a little extra code they can be accessed from Pascal or FORTRAN programs.

Differences Between Computers

For the most part, DIL routines function the same on different computers; that is, the routines should work basically the same for the Integral PC, Series 200/300, Series 500, and Series 800 computers. However, some differences do exist.

Where differences do exist, you'll be alerted by bold introductory phrases such as:

- **Integral PC Only:**
- **Series 200/300 Only:**
- **Series 500 Only:**
- **Series 800 Only:**

In addition, major differences are outlined in an appendix for each computer system on which DIL routines run—Series 500, Series 200/300, Integral PC, and Series 800.

DIL Tutorial Contents

Chapter 1: Interfacing Concepts presents basic interfacing concepts and a description of the HP-IB and GPIO interfaces.

Chapter 2: General-Purpose Routines discusses how the interfaces are accessed in the HP-UX environment and how basic data transfers are implemented.

Chapter 3: Controlling the HP-IB Interface describes interfacing techniques for the HP-IB interface.

Chapter 4: Controlling the GPIO Interface covers interfacing techniques for the GPIO interface.

Appendix A: Series 500 Dependencies covers hardware- and system-dependent information for Series 500 computers. If you use DIL routines on a Series 500 computer, you should check this appendix to ensure the correct use of DIL routines.

Appendix B: Series 200/300 Dependencies describes hardware- and system-dependent information. If you use DIL routines on a Series 200/300computer, you should check this appendix to ensure the correct use of DIL routines.

Appendix C: Integral PC Dependencies describes hardware- and system-dependent information specific to the Integral PC. If you use DIL routines on an Integral PC, you should check this appendix to ensure the proper usage of DIL routines.

Appendix D: Series 800 Model 840 Dependencies describes hardware- and system-dependent information specific to the Series 800 Model 840. If you use DIL routines on a Model 840, you should check this appendix to ensure the proper usage of DIL routines.

Appendix E: Character Codes

The DIL Interfacing Routines

As mentioned previously, Device I/O Library (DIL) routines allow you to access devices directly through HP-IB and/or GPIO interfaces connected to your computer system. Some routines are general-purpose and can be used with any interface supported by the library, while others provide control of specific supported interfaces. DIL currently supports the HP-IB and GPIO interfaces.

Linking the DIL Routines

You can make calls to the DIL routines from C, Pascal, or FORTRAN programs. However, the library is not automatically linked with your program when you compile the program with *cc(1)*, *pc(1)*, or *fc(1)*. You must use the `-l` flag to specify that the library be linked with the program. To compile a C program and then link the DIL routines with it, use:

```
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. For the exact location of DIL library on your computer system, see the appropriate hardware-specific appendix in this tutorial.

Calling the DIL Routines From Pascal

You must give an **external declaration** for each DIL routine called from a Pascal program. An external declaration consists of the routine 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 external declaration in a Pascal program for the routine 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 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 the DIL Routines From FORTRAN

C and FORTRAN routine calls are not compatible because C passes parameters **by value** while FORTRAN passes them **by reference**.

To overcome this incompatibility, direct the compiler to generate a call by value using FORTRAN's *$ALIAS* option. For example:

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

If your system's FORTRAN compiler does not support this form of *$ALIAS*, you may need to solve the parameter-passing differences by writing an **onionskin** routine. An onionskin routine 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)* using an onionskin routine, use:

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

and 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

This and the remaining sections in this chapter provide concepts information concerning interfaces in general and the HP-IB and GPIO interfaces in particular. This information is provided as background information only; it is not required before using the DIL routines (although you may find some of the information useful). You can skip the remainder of this chapter without serious detrimental effects.

What Is an Interface?

The primary function of an interface is to provide a data communication path between the computer and its associated peripherals. Interfaces act as intermediaries between peripherals by handling part of the **bookkeeping** work and ensuring that the communication process flows smoothly

On HP's 9000 family of computers, the interface connects directly to the computer either hard-wired or as a card that fits in your computer's backplane slots. Peripherals are connected to the interface via cables. The functions of an interface are shown in the following block diagram (Figure 1-1).

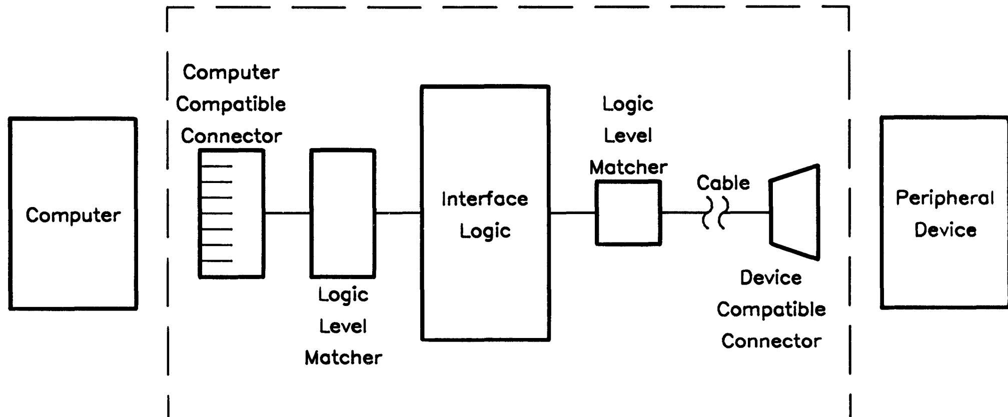


Figure 1-1. Functional Diagram of an Interface

Interface Functions

In general, an interface performs the following functions:

- **Electrical and Mechanical Compatibility.** This simply means that you can attach compatible peripherals to an interface, and doing so won't destroy the peripheral or the interface.

- **Data Compatibility.** Just as two people must speak a common language to communicate, the computer and peripheral must agree upon the format of data before communicating. Ensuring proper data format is the responsibility of the programmer. Most interfaces merely move agreed-upon quantities of data between the computer and peripheral.

- **Timing Compatibility.** Since all devices do *not* have the same data transfer rates, nor do they agree as to when data should be transferred, there must be synchronization between peripherals and the interface: data transfers can be started at a time agreed upon by the interface and the peripheral, and the data must be transferred at a mutually agreeable rate.

  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.

Additional Interface Functions

Another powerful feature of an 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 next sections concentrate on the functions of two particular interfaces: the HP-IB and the GPIO.

The HP-IB Interface

The Hewlett-Packard Interface Bus (**HP-IB**) is an interface that provides compatibility between the computer and external devices conforming to the IEEE 488-1978 standard. Electrical, mechanical, and timing compatibility requirements are satisfied by the bus, which allows you to connect up to 15 devices to one interface.

General Structure

Communications through the HP-IB are made according to a precise set of rules defined by the IEEE 488-1978 standard. These rules ensure orderly communication. There are three types of devices on the HP-IB:

- controller
- talker
- listener.

These types are actually attributes that exist alone or in combinations in one device. For example, the HP-IB interface allows a desktop computer to be a **controller, talker**, and **listener**. A device that accepts data from the bus (for example, a printer) is usually a **listener**, while a device that supplies data to the bus (for example, a voltmeter) is usually a **talker**. At any one time, the bus has only one Active Controller and only one talker, but it can have any number of listeners.

The HP-IB is composed of 16 lines which are divided into 3 groups: 8 lines form a bi-directional data path which carries data, commands, and device addresses; 3 lines control the transfer of data bytes (handshake lines); and the 5 remaining lines control bus management.

Handshake Lines

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

- DAV — DAta Valid
- NRFD — Not Ready For Data
- NDAC — Not Data ACcepted.

NOTE

The HP-IB interface uses negative logic for handshake, data, and bus management lines. This means that a line is **asserted** (true) when its voltage is low; when a line's voltage is high, the line is **not asserted** (false).

The timing diagram in Figure 1-1 illustrates how the handshake lines are used to complete a data item transfer. You should refer to this diagram when reading the subsequent discussion of the HP-IB handshake.

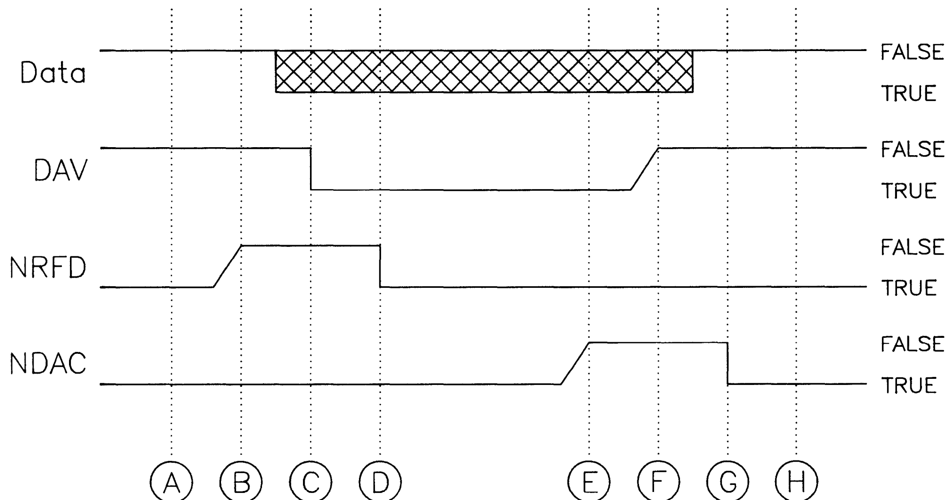


Figure 1-2. The HP-IB Handshake

At the start of the handshake (point A), the handshake lines are in the following states:

- DAV is false — there is no valid data on the data lines.
- NRFD is true — none of the listeners are ready to accept data.
- NDAC is true — there is no data for the listeners to accept.

When a listener is ready to accept data, it de-asserts NRFD—that is, it lets NRFD float high. However, NRFD remains asserted (true) until *every* listener de-asserts it[1]. When every listener is ready to accept data (that is, when NRFD is de-asserted by every listener), NRFD becomes false (point B).

By looking at NRFD, the talker knows when it can send data: when NRFD is false, the talker knows that every listener is ready to accept data; the talker then puts data on the data lines and asserts DAV (point C), thus telling the listeners that there is valid data on the data lines to be accepted.

As soon as a listener senses the assertion of DAV, the listener asserts NRFD (point D), thus driving NRFD low (true).

After point D, each listener accepts the data on the data lines. When a listener has accepted the data, it de-asserts NDAC. As with the NRFD line at point B, NDAC remains asserted (true) until every listener on the bus de-asserts (makes false) the NDAC line. When every listener has de-asserted NDAC, the line becomes false (de-asserted), thus telling the talker that every listener has accepted the data (point E).

When the talker sees that every listener has accepted the data, the talker de-asserts (makes false) the DAV line and takes data off the data lines (point F).

As soon as a listener senses that data is no longer valid, it asserts NDAC (point G), thus signifying the end of the handshake (point H). When the handshake is finished, all lines are at the values they had before the handshake started (point A).

[1] The reason NRFD remains asserted until every listener de-asserts it is because an active low voltage on the bus line (asserted) overrides a passive high voltage (de-asserted). Therefore, the line remains asserted until every listener sets the line voltage to a passive high (de-asserted).

Bus Management Control Lines

There are five bus management control lines:

- ATN — ATtentioN
- IFC — InterFace Clear
- REN — Remote ENable
- EOI — End Or Identify
- SRQ — Service ReQuest.

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 line's (**ATN**'s) logic state. That is, when ATN is false, the states of the data lines are interpreted as data. When ATN is true, the data lines are interpreted as commands.

IFC: The Interface Clear Line

Only the System Controller sets the **IFC** line true. By asserting IFC, all bus activity is unconditionally terminated, the System Controller becomes the Active Controller, and any current talker and 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.

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 **REN** is true is placed in its remote mode of operation.

EOI: The End or Identify Line

The **EOI** line is used to indicate the end of a data message. Normally, data messages sent over the HP-IB are sent using standard ASCII code and are terminated by the ASCII line-feed character. However, certain devices need to send blocks of information containing data bytes which have the line-feed character bit pattern as part of the data message. Thus, no bit pattern can be designated as a **terminating character**, since it could occur anywhere in the data stream. For this reason, the EOI line is used to mark the end of the data message.

Another function of EOI is that, when it is asserted along with the ATN line, a parallel poll is taken of the bus.

SRQ: The Service Request Line

The Active Controller is always in charge of ordering events that occur on the HP-IB. If a device on the bus needs the Active Controller's help, it sets the **SRQ** line true. The SRQ line sends a request for service, not a demand, and it is up to the Active Controller to choose when and how it services the request. However, the device continues to assert SRQ until it has been satisfied. Exactly what satisfies a service request depends on the requesting device, and is explained in the device's operating manual.

The GPIO Interface

The **GPIO** (General Purpose Input/Output) interface is a very flexible parallel interface that allows communication with a variety of devices. The GPIO interface utilizes data, handshake, and special-purpose lines to perform data transfers via user-selectable handshaking methods.

Four GPIO interfaces are supported by DIL routines: the GPIO for Series 200 and 300, the GPIO for Series 500 computers, the GPIO for the Integral Personal Computer, and the AFI card for the Series 800 Model 840. You should refer to the appropriate hardware-specific appendix for details on each GPIO.

General-Purpose Routines 2

The DIL library contains several general-purpose routines that can be used with any interface supported by the library. (These routines are listed in Table 2-1.) This chapter discusses how to use these routines from your programs. Specifically, the following topics are presented:

- concepts essential to understanding the use of DIL library routines
- opening an interface's special file
- closing an interface's special file
- reading from and writing to an interface's special file
- designing error-checking routines
- resetting an interface
- controlling input/output parameters
- determining why a read terminated
- handling interrupts

Concepts

The Interface Special File

HP-UX treats I/O to an interface the same way it treats I/O to an input/output device: the interface must have a special file. Before you can write programs that call DIL routines to communicate with an interface, the interface must have an appropriate special file.

The special file for a device interface must be created before calling DIL routines to communicate with the interface. The method for creating an interface's special file depends on which model of computer you use. You should refer to the appropriate hardware-specific appendix for details on creating the interface special file for your system.

The Entity Identifier (eid)

Nearly all DIL routines require an **entity identifier** (*eid*) as a parameter. The entity identifier is an integer returned from opening (via the *open(2)* system call) an interface's special file. When supplied as a parameter to a DIL routine, the entity identifier tells the routine which interface special file to work with.

The Programming Model

As a general rule, all programs that call DIL routines to operate on a specific interface conform to the following structure:

1. Get the entity identifier (*eid*) for the interface with which you wish to communicate. This is done by opening the interface's special file. For details on obtaining an interface's entity identifier, see the section "Opening an Interface's Special File."
2. After obtaining the *eid*, your program can call DIL routines to perform various tasks with the corresponding interface. This and the remaining chapters in this tutorial describe how to use the various routines. (General-purpose routines covered in this chapter are described briefly in the following "General-Purpose Routines" section.)
3. When finished calling DIL routines, your program should close the interface's special file, opened in step 1 above. For details on closing this special file, see the section "Closing an Interface's Special File."

General-Purpose Routines

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

Table 2-1. General-Purpose Routines.

Routine	Description
io_reset	Reset a specified interface.
io_timeout_ctl	Establish a timeout period for any operation performed to a specified interface by a DIL routine.
io_width_ctl	Set the width of the data path 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 a specified interface.
io_on_interrupt	Set up interrupt handling for a program.
io_interrupt_ctl	Allow enabling and disabling of interrupts for a specified interface.

Series 200/300 computers support an additional routine, *io_burst*; you should refer to the *io_burst(3D)* page of the *HP-UX Reference* for details on using this routine.

In addition to the above standard DIL routines, the **Integral PC DIL library** supports non-standard DIL routines. You should refer to the appendix "Integral PC Dependencies" for details on these routines.

Opening an Interface's Special File

Other than the default standard input, standard output, and standard error files, you must explicitly open files in order to read and write to them from inside C, FORTRAN, or Pascal programs. The HP-UX system routine for opening files is *open(2)*. It is called as follows:

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

The *filename* is either a character string representing a file's external HP-UX name or a pointer to a buffer that contains the external name.

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

The integer *oflag* specifies the access mode for opening the file. It can have one of three possible values, as defined in the */usr/include/fcntl.h* header file: `O_RDONLY` (0) requests read-only access, `O_WRONLY` (1) requests write-only access, and `O_RDWR` (2) requests both read and write access. To use these constants in your programs, you must use the `#include` C-compiler directive, as shown in the above example.

When used on an interface's special file, the *open* system call returns an integer representing the interface's entity identifier (*eid*). As mentioned in the "Concepts" section of this chapter, the entity identifier is required as a parameter to DIL routines; it is also required as a parameter when reading from or writing to an interface's special file.

The following code defines an entity identifier called *eid* and opens an interface file called */dev/raw_hpib* with read and write access:

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

As an alternative to specifying the character string name of the HP-UX file in the call to *open*, you can place the name in a buffer and then call *open* with a pointer to the buffer. For example, the following code also opens the HP-IB interface file:

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

If a file is successfully opened, *open* returns a non-negative integer as the entity identifier. However, if an error occurs and the file is not opened, a −1 is returned.

Closing an Interface's Special File

When your program is finished with an opened interface special file, the special file should be closed using the *close(2)* system call.

Normally, when a process terminates (via *exit(2)* or a return from the main routine), any of its open files are automatically closed by the HP-UX operating system. However, it is still good programming practice to close a file when you're finished using it.

NOTE

HP-UX limits the number of files one process (program) can have open at one time to `NOFILE`, as defined in the */usr/include/sys/param.h* header file.

The *close* routine requires the entity identifier for the opened interface special file you wish to close. The following code shows how an HP-IB interface can be opened and closed:

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

   :      /* You can now call routines to
                   communicate with the interface. */
   close(eid);
}
```

The connection between the entity identifier and the open file is now broken, and the entity identifier is available for the system to assign to another file. A file that is opened on two separate occasions need not be assigned the same entity identifier both times by the system.

If the routine successfully closes the specified file, it returns a 0; if not, it returns a −1 and the external error variable *errno(2)* is set to indicate the error (see the section "Designing Error Checking Routines"). A common cause of the routine failing is using an argument that is not a valid entity identifier for an open interface file.

Reading and Writing

The lowest level of I/O in HP-UX provides no buffering or other services; it is a direct entry into the operating system. Two HP-UX system routines provide low-level I/O read/write capabilities: *read(2)* and *write(2)*. Both require three arguments:

- an entity identifier of an open file
- a buffer in your program where the 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.

The call to *read* has this form:

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

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

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

The call to *write* is very similar:

```
#include <fcntl.h>
main()
{
   int  eid;        /*the entity identifier*/
   char *buffer;    /* the buffer containing data to be written to a file*/
   eid = open("/dev/raw_hpib", O_RDWR);

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

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

Although *read* and *write* required the number of bytes to be transferred as their third argument, other parameters, discussed later, associated with the interface file's *eid* can end the transfer before this number is reached. Both *read* and *write* return the number of bytes transferred.

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

Example

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

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

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

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

The number of bytes to be sent is 14 because there are 14 characters in the data string. To receive 10 bytes of data from the bus you use:

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

The *read* routine will attempt to read 10 bytes of data from the interface and put the data in *buffer*.

Designing Error Checking Routines

All Device I/O Library routines return a −1 to indicate that an error occurred during the routine's execution. If this happens, the routine sets an external HP-UX variable called *errno*.

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; therefore, you should only check its value after you have determined an error occurred.

Except for this section, most examples in this manual do not check for the successful completion of routine calls. However, as good programming practice you should include error checking in your own programs.

The *errno(2)* page in the *HP-UX Reference* defines the various errors returned when a system call fails. You should refer to this documentation for a complete description of errors.

Using errno

To access *errno* from your program you must include the following code at the beginning of the program:

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

The errno.h Header File

The header file */usr/include/errno.h* uses error number definitions from the */usr/include/sys/errno.h* header file. Refer to the *errno(2)* entry in the *HP-UX Reference* to see this list and to find out the meaning associated with each value.

Displaying errno

Once you have declared *errno*, there are two ways you can check its value if a routine fails. The simplest way is to check to see if the routine failed, and if so, to print out the value of *errno* and then exit. The example below 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);
   }
   :
}
```

If an error occurs and *errno*'s value is printed, you must then refer to *errno*'s entry in the *HP-UX Reference* to find out what the number means.

Error Handlers

Another approach is to check for specific values of *errno* and execute different error routines depending on its value. Only a limited number of situations can cause the failure of a particular routine; thus, a routine usually has a small set of values that it can assign to *errno*. To find out what this set is, refer to the routine's entry in the *HP-UX Reference.*

For example, in the *HP-UX Reference* you find that *errno* is set to `ENOENT` (defined in the *errno.h* header file) when you try to open a file that doesn't exist. Once this is known, you can incorporate the following code into the program:

```
#include <errno.h>
#include <fcntl.h>
main()
{
   int eid;
   :
   if ((eid = open("/dev/raw_hpib", O_RDWR)) == -1)
   {
      if (errno == ENOENT)
         printf("Error occurred because file doesn't exist to open");
      else
         printf("File exists to open, but still an error occurred");
      exit(1);
   }
   :
}
```

Notice the print statements in the example above could be replaced with calls to more-complicated error handling routines such as *perror(3)* (see the *HP-UX Reference*).

Resetting Interfaces

The DIL routine for resetting an interface is *io_reset.* This routine is used on either HP-IB or GPIO interfaces.

The following call to *io_reset* resets the interface whose entity identifier is *eid*—i.e., *eid* was returned from opening the interface's special file.

```
io_reset(eid);
```

io_reset resets the interface whose entity identifier is *eid.* You should refer to the appropriate hardware-specific appendix for details on the exact effects of *io_reset* on your computer's HP-IB and GPIO.

Assume that after opening an interface file you want to make sure the interface operates correctly. This is done by calling *io_reset* and looking at its return value:

```
#include <fcntl.h>
main()
{
   int eid;
   :
   eid = open( "/dev/raw_hpib", O_RDWR);
   if (io_reset(eid) == -1)
   {
       printf("Possible problem with interface");
       exit(1);
   }
   :   /* program continues if "io_reset" was successful */
}
```

Controlling I/O Parameters

The Device I/O Library provides four routines that allow you to control three different parameters involved in data transfers between an interface card and the devices connected to it. The routines and the parameters they control are listed below.

Routine	I/O Parameter
io_timeout_ctl	Timeout: Assign a timeout value for I/O operations.
io_width_ctl	Data Path Width: Specify width of the interface's data path.
io_speed_ctl	Transfer Speed: Request a minimum speed for data transfers through the interface.
io_eol_ctl	Read Termination Pattern: Assign a pattern to be recognized as a read termination pattern.

When you use one of these four routines, its effect is associated with the open interface file for the interface. If you close the file the effect is lost and the I/O parameter returns to its default state the next time the file is opened.

Setting the I/O Timeout

The I/O timeout parameter controls how long an interface spends trying to complete a data transfer with a device connected to it. When you open the interface file associated with the interface, the timeout is set at 0 by default, indicating that the system never causes a timeout.

If timeout is zero and an error condition occurs which keeps a data transfer from completing, your program may hang. It is recommended you set a timeout for the interface. To set or change the timeout use *io_timeout_ctl*:

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

   eid = open( "/dev/raw_hpib", O_RDWR);
   time = 1000000;     /*set timeout of 1 second*/
   io_timeout_ctl(eid, time);

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

eid is the entity identifier for the open interface file; *time* is a 32-bit long integer specifying the length of the timeout in microseconds.

If *read* or *write* requests do not complete within the time limit specified by the timeout value, the requests are aborted and an error indication is returned (a return value of -1). If a routine fails due to the timeout occurring, *errno* is set to **EIO** (not to be confused with EOI).

Although you specify the timeout value in microseconds (μ-secs) when you call *io_timeout_ctl*, the resolution of the effective timeout is system-dependent. The time-out value is rounded up to your system's time resolution boundary. For example, if your system's resolution is 10 milliseconds and you request a timeout of 25000 microseconds (25 milliseconds), the effective timeout is set at 30 milliseconds. To determine the time resolution on your computer system, refer to the appropriate hardware-specific appendix.

IMPORTANT

An actual timeout of 0 microseconds (i.e., timeout occurs as soon as a routine is called) is **not** possible. However, specifying a timeout of zero sets an infinite timeout; the system will never cause a timeout. **Specifying a timeout of zero is not recommended.**

An entity identifier for an interface file obtained with the HP-UX routine *dup(2)* or inherited by a *fork(2)* request shares the same timeout as the original entity identifier for the file obtained with *open*. If the child process resulting from a *fork* inherits an entity identifier and then changes the timeout, the entity identifier used by the parent process is also affected.

Series 200, 300, and 500 Only: If your program has used *open* more than once to open the same interface file, the entity identifiers returned by *open* can each have their own timeout associated with them. Using *io_timeout_ctl* with one entity identifier does not affect the other entity identifiers.

Setting Data Path Width

When you create an interface file and then open it for the first time, the data path width defaults to 8 bits. Once the file is opened, *io_width_ctl* lets you select a new width. **Allowable widths are system and hardware dependent**; you should refer to the hardware-specific appendix for your system to determine what widths are allowed for various interfaces.

Assuming that the open interface file has the entity identifier *eid*, *io_width_ctl* is called with:

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

where *width* is the number of bits that are in the new data path. The routine returns a −1 to indicate an error if the width that you specify is not supported on the specified interface.

For example, to change the width of a GPIO data bus from 8 to 16 bits you can use:

```
#include <fcntl.h>
main()
{
    int  eid, width;
    width = 16;               /*width of new data path */
    eid = open("/dev/raw_gpio", O_RDWR);
    io_width_ctl(eid, width); /*assign new width for GPIO bus*/
    :   /*data transfers using "/dev/raw_gpio" will now
                use a 16-bit bus*/
}
```

Changing the data path width of an interface with this routine affects all users of the interface. Once you change the data path width, it stays at the new value for each future opening of the file. Either *io_reset* or *io_width_ctl* can be used to change the path back to the default of 8 bits.

Setting Transfer Speed

You can set the minimum transfer speed that is used on the interface (within the limits of the hardware) with the routine *io_speed_ctl*:

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

where *eid* is the entity identifier for the open interface file and *speed* is an integer indicating a minimum speed in kilobytes (Kb) per second[1].

The routine returns a 0 if it is successful, and a −1 if an error occurred. For example:

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

requests a minimum speed of 1 024 bytes per second. The system may use a faster transfer rate, but you are at least supplied with that speed.

The transfer method (e.g., **DMA**, interrupt) chosen by your system is determined by the minimum speed that you request. The system selects a transfer method that is as fast or faster than the speed you requested. If you request a speed that is beyond the limitations of the system, the fastest transfer method possible is used. See the appropriate hardware-specific appendix for details.

Setting the Read Termination Pattern

When you perform read operations on an open interface file, certain conditions cause the interface to recognize the end of data transfer from a sending device. When you call *read*, you must specify how many bytes you expect to read. After the specified number of bytes have been read, the data transfer halts.

The interface you are accessing can also be configured to recognize a special read termination condition. For instance, if an HP-IB interface sees the EOI line asserted, it knows that it has received the last data byte in the transfer and the read operation halts, whether or not the specified byte count has been reached.

The DIL routine *io_eol_ctl* causes an interface to recognize a particular character or string of characters as a **read termination pattern**, in addition to any other termination conditions already in effect for the interface. The call to the routine 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* either enables or disables the interface's ability to recognize a special read termination pattern.

[1] A kilobyte equals 1 024 bytes.

When *flag* = 0, any previously set read termination pattern is disabled. If *flag* has any other value, then *match* is the new termination pattern.

When *flag* indicates enable mode (e.g., *flag* = 1) and the interface's data path is 8 bits, the least-significant byte of **match** is the integer equivalent of the termination pattern that you want to set.

If the data path for the interface is set at 16 bits (such as with a GPIO interface), then for most systems the **termination pattern** is also 16 bits. It is taken from the 2 least-significant bytes of the specified **match** value.

Note that if any special read termination condition defined for the interface is still in effect (e.g., EOI for an HP-IB). Either it or the termination pattern that you have defined could cause a read operation to halt. Also note the read termination pattern you set up is interpreted by the interface as the last byte of data. In other words, the interface sees it as part of the data message but does not try to read past it.

To illustrate using *io_eol_ctl*, assume that you want to set up an HP-IB interface to recognize a backslash-n (`\n`) as a read termination pattern. First, you must open the HP-IB interface file and obtain the entity identifier *eid*. Second, make the call to *io_eol_ctl* in your program using *eid* as the entity identifier, `ENABLE` as the flag, and `\n` as the match:

```
#include <fcntl.h>
#define  ENABLE     1
main()
{
   int eid;
   eid = open("/dev/raw_hpib", O_RDWR);
   io_eol_ctl(eid, ENABLE, '\n');
   :      /*data transfers using "eid" terminate with a '\n'*/

}
```

Now when data is read from */dev/raw_hpib*, the read operation is terminated when any one of the following occurs:

- The byte count specified in the call to *read* is reached.
- The HP-IB's EOI line is asserted. The character on the bus, when the interface sees the line's assertion, becomes the last byte in the data message.
- A backslash-n (`\n`) is read. The backslash-n (`\n`) becomes the last byte in the data message.

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

An entity identifier for an interface file obtained with the HP-UX system routine *dup* or inherited by a *fork* request shares the same read termination pattern as the original entity identifier. If the child process resulting from a *fork* inherits an entity identifier and then sets a read termination pattern for it, the entity identifier used by the parent process is also affected.

Series 200, 300, and 500 Only: If your program has used *open* more than once to open the same interface file, the entity identifiers returned by *open* can each have their own read termination pattern associated with them. Using *io_eol_ctl* with one entity identifier does not effect the others. Thus, you can set up several entity identifiers for the same interface that recognize different termination characters.

Removing 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);
```

The *XX* indicates a **don't care** value for the match argument. If the flag is 0, then the match value is not looked at by the routine.

The following code sets up the ASCII '.' (decimal value 46) as a termination pattern, does a read operation, and then disables the termination pattern.

```
#include <fcntl.h>
   main()
   {
      int eid;
      char buffer[12];
      eid = open("/dev/hpib_dev", O_RDWR);
      io_eol_ctl(eid, 1, 46);
      read( eid, buffer, 12);   /*Read operation halts when either a
                                  "." is read or when the 12th byte is read*/
      io_eol_ctl( eid, 0, 0);          /*termination pattern is removed*/
      :
   }
```

Determining Why a Read Terminated

There are several situations which can terminate read operations through an interface. After your program completes a *read*, you may want to include code that verifies the cause of the *read*'s termination is what you expected. The DIL routine that allows you to do this is *io_get_term_reason*.

io_get_term_reason accepts the entity identifier of the interface file as an argument and returns an integer. The returned value indicates how the last read operation ended, as shown below.

Returned Value	Meaning
−1	An error occurred while making this routine call.
0	The last read terminated abnormally (for some reason other than the ones covered below).
1	The last read terminated by reading the number of bytes requested.
2	The last read terminated by detecting a previously determined read termination pattern.
4	The last read terminated by detecting some device-imposed termination condition, for example, the assertion of EOI for an HP-IB interface.

If a read terminated for multiple reasons, the bits that are set indicate each of the reasons. The three least-significant bits of the lowest byte have the meanings indicated by their associated decimal values in the table above. For example, if *io_get_term_reason* returns a 7 you know that the specified number of bytes were read, the last byte read was a read termination pattern, and also a device-defined termination condition occurred.

NOTE

If no *read* is performed on an interface file once it is opened and you call *io_get_term_reason*, the routine returns a 0.

All entity identifiers descending from one *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 from a device on an HP-IB and need to guarantee that a specific number of bytes are read. The following code reads 50 bytes through an opened interface file and makes sure that *read* wasn't terminated before all 50 were read.

```
#include <fcntl.h>
main()
{
   int  eid, condition;
   char buffer[50];        /*storage for data*/

   eid = open("/dev/raw_hpib", O_RDWR);
   read(eid, buffer, 50);   /*perform read and put data in "buffer"*/
   if ((condition = io_get_term_reason(eid)) > 1)
      /*Terminated due to seeing a read termination pattern or the
        assertion of EOI. However, the event could have occurred at the
        same time as the 50th byte was read*/
       printf("Possible termination before all of data was read");

   else if (condition < 1)
        {
          if (condition == 0)
             /*Termination due to some abnormal condition*/
             printf ("Last read terminated abnormally");

           else
             printf ("io_get_term_reason call failed");
        }
   else
     /*Termination due to reading the 50th byte*/
       printf("All of data was read into buffer");
 }
```

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

Interrupts

DIL provides an **interrupt** mechanism that is similar to HP-UX signal handling. The user is able to set up **interrupt handlers** to be invoked when certain conditions occur. DIL currently supports interrupts for HP-IB and GPIO interfaces.

Currently, **interrupts are supported only on the Integral PC, Series 500, and Series 800 computers**; however, you can simulate interrupts on Series 200/300 computers. You should check the hardware-specific appendix for your system for any restrictions that may apply.

Interrupts on the Integral PC

The only interrupt condition available on the Integral PC is PIR, meaning the Peripheral Interrupt Request has been asserted. For hardware restrictions on using the HP-IB interrupts on the Integral PC, refer to the *io_on_interrupt.3d* file in the *doc* folder on the DIL disc.

Interrupts on the Series 500

The following interrupt conditions are available for HP-IB interfaces on Series 500 computers:

Name	**Meaning**
SRQ	SRQ line has been asserted
TLK	The computer has been addressed to talk
LTN	The computer has been addressed to listen
CIC	The computer has received control of the bus
IFC	The IFC line has been asserted
REN	The remote enable line has been asserted
DCL	The computer has received a device clear command
GET	The computer has received a group execution trigger command
PPOLL	A specific parallel poll response occurred

The following interrupt conditions are available for the GPIO interface:

SIE0	Status line 0 has been asserted
SIE1	Status line 1 has been asserted
EIR	Enable Interrupt, ATTN line has been asserted

io_on_interrupt

DIL provides two routines for controlling interrupts. The first routine, *io_on_interrupt*, sets up the interrupt information and has the form:

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

where *eid* is an entity identifier for a GPIO or raw HP-IB interface. The parameter *handler* points to a function to be invoked when the condition occurs. Then *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.*

The *cause* parameter is a bit vector specifying which of the interrupt or fault events will cause the *handler* routine to be invoked. The interrupt *causes* are often specific to the type of interface being considered. Also, certain exception (error) conditions can be handled using the *io_on_interrupt* capability. Specifying a zero-valued *cause_vec* vector effectively turns off the interrupt for that *eid.*

The *mask* parameter is used when an HP-IB parallel poll interrupt is being defined. The integer *mask* specifies which parallel poll response lines are of interest. *mask*'s value is obtained from an 8-bit binary number, each bit of which corresponds to one of the eight lines. For example, if you want an interrupt handler invoked for a response on lines 2 or 6, the correct binary number is 01000100. This converts to a decimal equivalent of 68, which is the number you should assign to *mask.*

Upon occurrence of an enabled interrupt condition on the specified *eid*, the receiving process 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.

An interrupt for a given *eid* is implicitly disabled after the event occurs. The interrupt condition can be re-enabled with *io_interrupt_ctl.*

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

The following example illustrates how an interrupt handler can be set up to handle assertion of the service request line (SRQ):

```
#include <dvio.h>
#include <fcntl.h>
#include <stdio.h>
main()
{
   int eid;
   struct interrupt_struct cause_vec;

   eid = open ("/dev/raw_hpib", O_RDWR);
   cause_vec.cause = SRQ;
   io_on_interrupt(eid, &cause_vec, handler);
   :
}
handler (eid, cause_vec)
int eid;
struct interrupt_struct *cause_vec;
{
    if (cause_vec->cause == SRQ)
       service_routine(); /* user specific routine*/
}
```

io_interrupt_ctl

The *io_interrupt_ctl* routine allows the user to enable or disable interrupts on a specific *eid.* Since interrupts are automatically disabled when an interrupt occurs, *io_interrupt_ctl* is commonly used when the user wants to repeatedly handle a specific event. The call to *io_interrupt_ctl* has the following form:

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

where *eid* is an entity identifier for an open GPIO or raw HP-IB device file. To control enabling and disabling of the interrupts, *enable_flag* is used. If *enable_flag* is non-zero, then interrupts are enabled on the *eid.* If *enable_flag* is zero, then interrupts are disabled on the *eid.* **Note** that attempting to use *io_interrupt_ctl* on an *eid* that has not had an *io_on_interrupt* applied to it, fails.

The following example modifies the handler from the previous example to re-enable interrupts:

```
handler(eid, cause_vec)
int eid;
struct interrupt_struct *cause_vec;
{
   if (cause_vec->cause == SRQ)
   {
      service routine(); /* user specific routine*/
      io_interrupt_ctl(eid,1);
    }
 }
```

Controlling the HP-IB Interface 3

To gain a full range of control over your computer's HP-IB interface you must use:

- the general purpose I/O routines in DIL discussed in Chapter 2, "General-Purpose Routines"
- the DIL routines, described in this chapter, designed specifically for controlling the HP-IB interface.

Besides the various routines, you must know about the commands that are interpreted on an HP-IB. This chapter provides some general information about HP-IB commands and introduces the DIL routines that specifically control the HP-IB. Then it relates this information to the information provided in Chapter 2, "General-Purpose Routines," to illustrate some HP-IB interfacing strategies.

Overview of HP-IB Commands

This section discusses the HP-IB commands that are sent over the 8 data lines while the ATN line is asserted. You can send all of these commands using a DIL routine called *hpib_send_cmnd*. This routine takes care of the assertion of ATN and the necessary handshaking between devices. The computer's interface must be the Active Controller before *hpib_send_cmnd* is used and any of the HP-IB commands sent. How *hpib_send_cmnd* is called from your program is discussed later in this chapter.

In order for the commands to be interpreted by devices on the HP-IB, the bus's remote enable line (REN) must be in its enabled state. Only the System Controller changes the state of this line (see the "System Controller's Duties" section later in this chapter). By default, REN is enabled.

Commands sent on the bus's data line form 4 groups:

- **Universal commands** cause every device, so equipped, to perform a specific interface operation. The devices do not have to be addressed as listeners.
- **Addressed commands** are similar to the universal commands, except they affect only those devices currently addressed as listeners.
- **Talk and listen addresses** are commands that assign talkers and listeners on the bus.
- **Secondary commands** are commands that must always be used in conjunction with a command from one of the above groups.

The table below lists the commands that you can send with *hpib_send_cmnd.* Later, when you use the routine, you may need to refer back to this table for the decimal or ASCII character value of particular commands.

Table 3.1 Bus Commands

Command	Decimal Value	ASCII Character
Universal Commands:		
UNLISTEN	63	?
UNTALK	95	_
DEVICE CLEAR	20	DC4
LOCAL LOCKOUT	17	DC1
SERIAL POLL ENABLE	24	CAN
SERIAL POLL DISABLE	25	EM
PARALLEL POLL UNCONFIGURE	21	NAK
Addressed Commands:		
TRIGGER	8	BS
SELECTED DEVICE CLEAR	4	EOT
GO TO LOCAL	1	SOH
PARALLEL POLL CONFIGURE	5	ENQ
TAKE CONTROL	9	HT
Talk and Listen Addresses:		
Talk Addresses 0-30	64-94	@ thru ^ (uppercase ASCII)
Listen Addresses 0-30	32-62	space thru > (numbers and special characters)
Secondary Commands: (If a secondary command follows the PARALLEL POLL CONFIGURE command then it is interpreted as follows, otherwise its meaning is device dependent)		
PARALLEL POLL ENABLE	96-111	‘ thru o (lowercase ASCII)
PARALLEL POLL DISABLE	112	p

UNLISTEN

The UNLISTEN command **unaddresses** all current listeners on the bus. Single listeners cannot be unaddressed without unaddressing all listeners. It is necessary to use this command to guarantee only desired listeners are addressed.

UNTALK

The UNTALK command unaddresses the current talker. Sending an unused talk address accomplishes the same thing. This command is provided for convenience since addressing one talker automatically unaddresses others.

DEVICE CLEAR

The DEVICE CLEAR command causes all **recognizing devices** to return to a pre-defined, device-dependent state. Recognizing devices respond whether or not they are addressed. Device manuals define the reset state for each device that recognizes the command.

LOCAL LOCKOUT

The LOCAL LOCKOUT command disables local control on all devices that recognize this command. Recognizing devices respond to the command whether or not they are addressed.

SERIAL POLL ENABLE

The SERIAL POLL ENABLE command establishes serial poll mode for all responding devices capable of being bus talkers. Recognizing devices respond to the command whether or not they are addressed. When a device is addressed to talk, it returns a 8-bit status byte message.

This command is not discussed any further since its function is accomplished by a DIL routine called *hpib_spoll* (discussed later in this chapter).

SERIAL POLL DISABLE

The SERIAL POLL DISABLE command terminates serial poll mode for all responding devices. Recognizing devices respond to the command whether or not they are addressed.

This command is not discussed any further since its function is accomplished by a DIL routine called *hpib_spoll* (discussed later in this chapter).

TRIGGER (Group Execute Trigger)

The TRIGGER command causes the devices that are currently addressed as listeners to initiate a preprogrammed, device-dependent action if they are capable. Device manuals indicate whether or not a particular device is capable of responding to the TRIGGER command and if it can, how to program it to do so.

SELECTED DEVICE CLEAR

The SELECTED DEVICE CLEAR command resets devices currently addressed as listeners to a device-dependent state, if they are capable. A device's documentation indicates whether or not the device recognizes this command and if so, it defines the reset state.

GO TO LOCAL

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

PARALLEL POLL CONFIGURE

The PARALLEL POLL CONFIGURE command tells the 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

The PARALLEL POLL ENABLE command configures devices addressed by the PARALLEL POLL CONFIGURE command to respond to parallel polls on a particular data line and with a particular logic level. Some devices implement a local form of this message (for example, jumpers) that cannot be changed.

This command must be preceded by the PARALLEL POLL CONFIGURE command.

PARALLEL POLL DISABLE

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

Overview of the HP-IB DIL Routines

Standard DIL Routines

Besides the general purpose routines described in Chapter 2, "General-Purpose Routines," DIL also provides routines that allow you to fully access the capabilities of the HP-IB interface. There are 14 of these routines:

Routine	Description
hpib_abort	Stops activity on a specified HP-IB select code.
hpib_io	Performs a mixture of HP-IB read, write, and SEND_CMD activities.
hpib_ppoll	Conducts parallel poll on HP-IB.
hpib_spoll	Conducts serial poll on HP-IB.
hpib_bus_status	Returns status on HP-IB interface.
hpib_eoi_ctl	Controls EOI mode for data transfers.
hpib_pass_ctl	Changes active controllers on HP-IB.
hpib_card_ppoll_resp	Configures it owns response to a parallel poll.
hpib_ren_ctl	Controls remote enable line (REN) on HP-IB.
hpib_rqst_srvce	Allows interface to generate an SRQ request on HP-IB.
hpib_send_cmnd	Sends characters on HP-IB with the attention line (ATN) line asserted.
hpib_wait_on_ppoll	Lets you wait for a particular parallel poll value to occur.
hpib_status_wait	Lets you wait until a particular status condition is true.
hpib_ppoll_resp_ctl	Defines interface parallel poll response as yes or no.

Additional Series 200/300 and Integral PC Routines

In addition to the standard HP-IB routines, the Integral PC and Series 200/300 support the following DIL routine:

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

For details on using this routine, refer to the appropriate hardware-specific appendix.

The Computer's Role on the HP-IB

Your computer must currently have one of the following two roles on the HP-IB:

- It is the **Active Controller**.
- If it isn't the Active Controller, it is a **Non-Active Controller**.

There can be only one Active Controller on an HP-IB interface at a given time. Since Active Controller status is passed between bus controller devices, your computer's status can change from **active** to **non-active**, or from **non-active** to **active**.

In addition to being either an Active or Non-Active Controller, your computer can also be the bus's **System Controller**. Once a controller is configured as the System Controller, it cannot be unconfigured without powering down the system. The System Controller is either the Active Controller or a Non-Active Controller. When the System Controller is initially powered up, it assumes the role of Active Controller.

Which of the DIL routines you can use depends on your computer's role on the HP-IB. Given the three role designations, Table 3-2 indicates which routines can be used with them.

Table 3-2. DIL Routine Role Designations.

Routine	System Controller	Active Controller	Non-Active Controller
hpib_abort	X		
hpib_io		X	
hpib_ppoll		X	
hpib_spoll		X	
hpib_bus_status	(X)	X	X
hpib_eoi_ctl	X		
hpib_pass_ctl		X	
hpib_card_ppoll_resp		X*	X
hpib_ren_ctl	X		
hpib_rqst_srvce		X*	X
hpib_send_cmnd		X	
hpib_wait_on_ppoll		X	
hpib_status_wait	(X)	X	X
hpib_ppoll_resp_ctl		X*	X

* means that the routine can be used if the computer is the Active Controller but there is no affect until it becomes a Non-Active Controller.

(X) means that the **X** isn't required since the System Controller must be either **active** or **non-active** and both of these roles can use the routine (i.e., the **System Controller** role is not required to use the routine).

Opening the HP-IB Interface File

Before you can use DIL routines on an HP-IB interface, the special file for the interface must exist. In addition, your program must open this special file and obtain its entity identifier. For details on creating an HP-IB special file and opening an interface's special file and obtaining its entity identifier, you should refer to the "Concepts" and "Opening an Interface's Special File" sections of Chapter 2, "General-Purpose Routines."

Sending HP-IB Commands

After your program has opened the special file for the HP-IB interface and obtained its entity identifier, you can call DIL routines to send HP-IB commands to the HP-IB interface. The DIL routine that allows you to place HP-IB commands on the data bus is *hpib_send_cmnd*. Your computer must be the Active Controller to use this routine.

One method of using this routine is to first set up a character array containing the commands that you want to send. You assign the decimal value for the commands to the elements of the array. The routine call then has 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 routine *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 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, the commands' ASCII character values can be used by specifying a character string. In this case, the routine call has the form:

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

where *eid* and *number* are the same as above. However, 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 you first set up an array with two elements, the decimal values for the commands, and then call *hpib_send_cmnd*:

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

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

If the ASCII character string method is used, the same effect is achieved with the code:

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

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

Since the array method allows you to store a list of commands, it should be used if you are sending a large number of commands or if you are sending the same set of commands several times in a program. With the string method, the entire set of commands must be specified as a string in the call to *hpib_send_cmnd*. It is useful if you are sending only a few commands or if a particular set of commands is only sent once in a program.

Errors While Sending Commands

Normally, *hpib_send_cmnd* returns a 0 if it executes successfully. However, it returns a −1 if any one of the following error conditions are true:

- The computer's interface is not the Active Controller.
- The *eid* entity identifier does not refer to an HP-IB raw interface file.
- The *eid* entity identifier does not refer to an open file.

To find out which of these conditions caused the error, the program should check the value of *errno*, an external integer variable used by HP-UX system calls. Chapter 2, "General-Purpose Routines" discusses how you can design an error checking routine that looks at the value of *errno*.

The following table indicates the value that *errno* will have given that one of the above conditions occurred during the call to *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

The Active Controller

Acting as Active Controller of the bus involves sending the HP-IB commands with *hpib_send_cmnd* and making calls to several other DIL routines. The functions of the Active Controller discussed in this chapter are:

- Setting up devices as talkers and listeners
- Gaining remote control of devices
- Locking out local control of devices
- Enabling local control of devices
- Triggering devices to initiate device-dependent actions
- Transferring data
- Clearing devices
- Servicing requests from devices
- Conducting parallel and serial polls
- Passing active control of the bus to another controller

Determining Active Controller

To carry out the Active Controller's bus management activities, the computer's HP-IB interface must be the Active Controller of its bus. If other devices on the bus are capable of being the Active Controller, you can use the *hpib_bus_status* routine to determine if the interface is currently the Active Controller.

To find out if the interface is the Active Controller, the call to *hpib_bus_status* must have the form:

```
hpib_bus_status(eid,4);
```

where *eid* is the entity identifier for the opened HP-IB interface device file and the *4* tells the routine to determine if the interface is the Active Controller. This routine returns a value that can be tested, see source code below.

hpib_bus_status returns 0 if the answer is no, 1 if the answer is yes, and −1 if an error occurred. The code that follows shows a straightforward way of interpreting the returned value:

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

   if ((status = hpib_bus_status(eid,4)) == -1)
     :                  /*an error occurred -- insert code that*/
     :                  /*flags it.                           */
   else if (status == 0)
     :                  /*not Active Controller -- insert code  */
     :                  /*that requests Active Controller status*/
   else
     :                  /*is Active Controller -- insert code for*/
     :                  /*the bus management routine required    */
}
```

Setting Up Talkers and Listeners

One talker and one or more listeners must be configured on the bus before data can be transferred. Also, some HP-IB commands effect only those devices currently addressed as listeners, which means that the Active Controller must specify the listeners before using them. There can be only one talker at a time on the bus, but there can be any number of listeners.

On Series 200/300, 500, and 800 computers, two methods exist for addressing listeners and talkers on an HP-IB. The first method, referred to as **auto-addressing**, instructs the computer to handle addressing for you.

The Integral PC does not support auto-addressing. This is because all HP-IB interface files on the Integral PC are raw special files and, therefore, do not support auto-addressing.

The second method requires using the *hpib_send_cmnd* function to **manually** address the bus. This is the only method available to Integral PC users.

Auto-Addressing on Series 200/300, 500, and 800

The system performs auto-addressing on normal (**non-raw**) HP-IB device files. Except for certain cases on the Series 800, note that DIL routines require a **raw** HP-IB device file. Therefore, while you can *open*, *close*, *read*, and *write* from a non-raw HP-IB device file, the DIL functions will fail. Please refer to the Model 840 appendix for information on Series 800 exceptions.

On Series 200, 300, and 500 systems, you can create a special file that informs the system to perform auto-addressing (see the appropriate hardware-specific appendix for details).

For example, suppose you've created an auto-addressable special file for a specific device on select code 1 at bus address 3; the device is an HP27110A/B card on a Series 500 computer and uses driver 12; the special file is named */dev/device*. If you list this file using *ll(1)*, you would see:

```
crw-rw-rw-   1 root   other     12 0x010300 Apr   5 1985 /dev/device
```

The following code illustrates auto-addressing using this device file:

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

Using hpib_send_cmnd

Talkers and listeners may be manually configured with the HP-IB commands formed by the talk and listen addresses of the devices. First, however, you should remove any previous listeners from the bus with the UNLISTEN command. To configure the bus's talker and listeners, the following steps are required:

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

To send the HP-IB commands necessary for this process you can use the *hpib_send_cmnd* routine.

Calculating Talk and Listen Addresses

A talker is specified on the bus by sending the talk address for the device, and a device is specified as a listener by sending its listen address. Talk addresses and listen addresses are both considered HP-IB commands, which means you should send them with the *hpib_send_cmnd* routine.

To calculate either the talk or the listen address for a device, you must first know its HP-IB address. The HP-IB address for the computer's interface card can be found using the *hpib_bus_status* routine:

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

where *eid* is the entity identifier for the interface file and *7* indicates that you want the routine to return the interface's HP-IB address. To find out the bus address of some other device, refer to its installation and operation documentation.

Once you have the device's HP-IB address, its *talk_address* (in decimal) is found using the formula:

```
talk_address = 64 + bus_address
```

where *bus_address* is the HP-IB bus address for the device. Bus addresses range from 0 to 30.

The listen address for a device (in decimal) is found similarly using the formula:

```
listen_address = 32 + bus_address
```

Thus, **My Talk Address** (*MTA*) for the computer is calculated with:

```
MTA = hpib_bus_status(eid, 7) + 64;
```

and **My Listen Address** (*MLA*) is calculated with:

```
MLA = hpib_bus_status(eid, 7) + 32;
```

An Example Configuration

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

```
#include <fcntl.h>
main()
{
     int  eid, MTA;
     char command[4];
     eid = open("/dev/raw_hpib", O_RDWR);
     MTA = hpib_bus_status(eid, 7) + 64; /*calculate My Talk Address*/
     command[0] = 63;       /* the UNLISTEN command*/
     command[1] = MTA;      /* the talk address for the interface*/
     command[2] = 32 + 4;   /* the listen address for device at HP-IB address
4*/
     command[3] = 32 + 8;   /* the listen address for device at HP-IB address
8*/
     hpib_send_cmnd(eid, command, 4);
 }
```

Remote Control of Devices

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

The level of the remote enable (REN) line of the HP-IB bus controls whether or not a device can respond to remote program control. If the REN line is enabled, any device that is addressed (as a listener) is automatically placed in the remote state. Only the System Controller can change the level of the REN line (see *System Controller's Duties* later in this chapter). By default, the line is enabled when the System Controller is powered up.

Locking Out Local Control

The LOCAL LOCKOUT command effectively locks out the **local** switch present on most HP-IB front panels, preventing a device's user from interfering with the system operations by pressing buttons. All devices that recognize this command are affected, whether they are addressed or not, and cannot be returned to local control from their front panels.

The following code shows one way of sending the LOCAL LOCKOUT command:

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

The local lockout state is cancelled by sending a GO TO LOCAL command to a device.

Enabling Local Control

During system operation, it may be necessary for an operator to interact with one or more devices in the local state. For instance, an operator might need to work from the front panel to make special tests or to troubleshoot. The GO TO LOCAL command returns all of the devices currently addressed as listeners to the local state.

For example, the code below places the devices at HP-IB addresses 3 and 5 into their 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 the devices currently addressed as listeners to initiate some device-dependent action. For example, it can be used to trigger a digital voltmeter to perform its measurement cycle. Because the response of a device to a TRIGGER command is strictly device-dependent, you can not specify with the command what action is to be initiated.

The following code triggers the device at bus address 5 to initiate some action:

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

Transferring Data

For the Active Controller to send data to another device it must:

1. Send an UNLISTEN command.
2. Send its own talk address (MTA).
3. 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.
4. Send the data.

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

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

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

   eid = open("/dev/raw_hpib", O_RDWR);
   MTA = hpib_bus_status(eid, 7) + 64;   /*calculate MTA*/
   command[0] = 63;                      /*the UNLISTEN command*/
   command[1] = MTA;                     /*talk address of interface*/
   command[2] = 32 + 5;                  /*listen address of device at*/
                                         /*address 5                 */
   hpib_send_cmnd(eid, command, 3);
   write(eid, "data message", 12);       /*send the data*/
}
```

Now assume that you are expecting to receive 50 bytes of data from another device on the bus. The code below allows the interface to receive character data from a device at bus address 5. The integer variable *MLA* contains the bus address of the interface.

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

   eid = open("/dev/raw_hpib", O_RDWR);
   MLA = hpib_bus_status(eid, 7) + 32;   /*calculate MLA*/
   command[0] = 63;                      /*the UNLISTEN command*/
   command[1] = 64 + 5;                  /*the talk address of device at*/
                                         /*address 5                   */
   command[2] = MLA;                     /*the listen address of interface*/
   hpib_send_cmnd(eid, command, 3);
   len = read(eid, buffer, 50);          /*store the data in "buffer"*/
   buffer[ len ] = '\0';                 /*terminate with NULL for printf*/
   printf("Data read is: %s", buffer);   /*print message*/
}
```

Clearing HP-IB Devices

There are two HP-IB commands for resetting devices to their pre-defined, device-dependent states. The first one is the DEVICE CLEAR command which causes all devices that recognize the command to be reset, whether they are addressed or not.

Thus, to reset all of the devices on an HP-IB accessed through a interface file with an entity identifier *eid*, you can use the following code:

```
:
command[0] = 20;              /* the 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 with an HP-IB address of 7, you can use the following code:

```
:
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);
:
```

Servicing 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 Service Request (SRQ) line on the bus. To determine if SRQ is being asserted, you check the status of the line, wait for SRQ, or set up a interrupt handler for SRQ. The *hpib_status_wait* routine allows you to write code that waits until the SRQ line is asserted before it continues. To specify that you want the program to wait until the SRQ line is asserted, *hpib_status_wait* must be invoked as follows:

```
hpib_status_wait(eid, 1);
```

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

The following code illustrates the use of *hpib_status_wait*:

```
#include <fcntl.h>
main()
{
   int eid;
   eid = open("/dev/raw_hpib", O_RDWR);
   io_timeout_ctl(eid,10000000);
   if (hpib_status_wait(eid, 1) == 0)
      service_routine();            /*SRQ is asserted; service the request*/
   else
      printf("Either a timeout or an error occurred");
}
```

Another solution is to periodically check the value of the SRQ line with *hpib_bus_status*. To check the SRQ line with *hpib_bus_status*, the call looks like this:

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

where *eid* is the entity identifier for the open interface file and *1* indicates that you want the logical value of the SRQ line returned. The routine returns 1 if SRQ is asserted, 0 if it isn't, and −1 if an error occurred.

The most practical way to monitor the SRQ line is to set up a interrupt handler for that condition (see the "Interrupts" section of Chapter 2, "General-Purpose Routines").

The Service Routine

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.

In many cases, devices requesting service **clear** the SRQ line when they are serially polled. They see the poll as an acknowledgement from the Active Controller to the device that the request has been seen and the Active Controller is responding.

If there is more than one device on the bus and the SRQ line is asserted, any one of the devices could be asserting the line. The Active Controller must then determine which of the devices needs service. There are two strategies for doing this:

- Serial poll each device until you find the one that is requesting service. 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 (that is capable) is programmed to respond on a different data line. However, since there can be 15 devices on the bus and there are only 8 data lines, it is sometimes necessary to have several devices respond on the same line.

  If several devices are programmed to respond on the same parallel poll line and the parallel poll shows that line asserted, the Active Controller must then serially poll each of these devices until it finds the one that is requesting service.

Thus, the Active Controller usually takes one of two approaches in response to seeing the SRQ line asserted: it can conduct a serial poll or it can conduct a parallel poll. In some cases the Active Controller may need to take both types of polls. The DIL routines that conduct these polls are *hpib_ppoll* and *hpib_spoll*. How these routines are used is discussed next.

Parallel Polling

The parallel poll is the fastest means of gathering device status when several devices are connected to the bus. Each device (with this capability) can be programmed to respond with one bit of status when parallel polled, making it possible to obtain the status of several devices in one operation. If a device responds affirmatively (**I need service**) to the parallel poll, more information as to its specific status can be obtained by conducting a serial poll of the device.

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

Configuring Parallel Poll Responses

Certain devices can be remotely programmed by the Active Controller to respond to a parallel poll. However, other devices require that the response be configured locally. Refer to the documentation for the device whose response you want to configure to find out if remote configuration by the Active Controller is possible.

The Active Controller remotely configures a device's parallel poll response by sending the HP-IB command PARALLEL POLL CONFIGURE followed by PARALLEL POLL ENABLE. The combination of these two commands tells devices addressed as listeners to respond to any future parallel polls on a particular data line and with a particular logic level. Some devices may implement a local form of this message (for example, jumpers) that can not be changed remotely by the Active Controller.

There are 16 different PARALLEL POLL ENABLE commands, each configuring a response on a specific data line and at a specific level. The 8-bits of the command have the following binary form:

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

where:

L indicates the logic sense of the response (e.g., *1* means that the device will respond with 1 when it needs service).

X indicates the data line on which the device will respond.

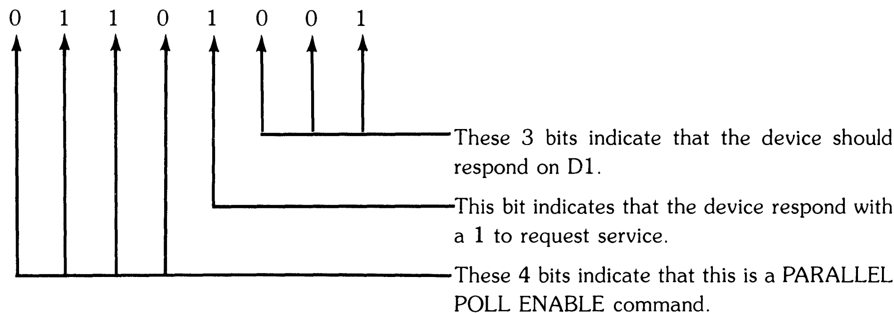
For example, given that the parallel response lines are labeled D0 to D7, a PARALLEL POLL ENABLE command with a decimal value of 104 (01101000 in binary) tells the addressed device to respond to parallel polls on data line D0 with a 1 when it needs service.

The following code shows how you can configure a device at bus address 5 to respond to a parallel poll by asserting data line D1 high when it needs service.

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

   eid = open("/dev/raw_hpib", O_RDWR);
   MTA = hpib_bus_status(eid, 7) + 64;  /*calculate MTA*/
   command[0] = MTA;         /*talk address of interface*/
   command[1] = 63;          /* the UNLISTEN command*/
   command[2] = 32 + 5;      /* the listen address for device at*/
                             /* address 5                      */
   command[3] = 5;           /* the PARALLEL POLL CONFIGURE command*/
   command[4] = 105;         /* the PARALLEL POLL ENABLE command*/
   hpib_send_cmnd(eid, command, 5);
}
```

Notice that the bit pattern for the PARALLEL POLL ENABLE command 105 used above is:



When the interface is the Active Controller, it can configure its own parallel poll response by addressing itself as both the talker and the listener. However, the configuration has no effect until the interface is no longer the Active Controller. 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.

The Active Controller disables a device from responding to any future parallel polls by first sending a PARALLEL POLL CONFIGURE command followed by PARALLEL POLL DISABLE. All devices that are currently addressed as listeners are disabled.

In the previous example a device at bus address 5 was configured to respond to parallel polls on D1. To disable the same device from responding you can use:

```
:
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 the parallel poll responses of devices on the HP-IB have been configured (either remotely or locally), the Active Controller can conduct a parallel poll with *hpib_ppoll.*

The *hpib_ppoll* routine 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 on a previously configured line. If an error occurs while the poll is being taken, a −1 is returned by the routine.

hpib_ppoll is invoked as follows:

```
hpib_ppoll( eid);
```

where *eid* is the entity identifier for the open interface file connected to the bus.

The code below indicates how you can interpret the byte returned by *hpib_ppoll.* Assume that a device at address 6 was previously configured to respond to a parallel poll by placing a 1 on D0 if it needed service. Assume the device at address 7 was configured to respond similarly on D1. If these are the only two devices able to respond to a parallel poll, you only care about the values of the 2 least significant bits of the integer returned by *hpib_ppoll.* The actual service routines have been left out of the example.

```
#include <fcntl.h>
main()
{
  int eid, status, byte;
  eid = open("/dev/raw_hpib", O_RDWR);

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

Errors During Parallel Polling

hpib_ppoll returns a −1 if any one of the following error conditions are true:

- The timeout defined by *io_timeout_ctl* occurred before all of the devices responded.
- The computer's interface is not the Active Controller.
- The *eid* entity identifier does not refer to a raw HP-IB interface file.
- The *eid* entity identifier does not refer to an open file.

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

Errno Value	Error Condition
EBADF	*eid* 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 or a timeout occurred.

Waiting For a Parallel Poll Response

The *hpib_wait_on_ppoll* routine allows you to wait for a specific parallel poll response from one or more devices. The effect of this is similar to waiting for the assertion of the SRQ line with *hpib_status_wait* (see the section *Servicing Requests*, presented earlier). With *hpib_wait_on_ppoll* you can wait for a specific device to request service; while *hpib_status_wait* is interrupted when any device requests service.

hpib_wait_on_ppoll is called with the form:

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

where *eid* is the entity identifier for the open interface file, *mask* is an integer whose binary value indicates on which parallel poll lines you are waiting for a request, and *sense* is an integer whose binary value indicates on which of these lines the request will use negative logic (device responds with 0 when it wants service). The routine returns the response byte **XOR**-ed with the *sense* value and **AND**-ed with the *mask*, unless an error occurs, in which case it returns a −1.

Calculating the mask

The routine *hpib_wait_on_ppoll* only looks at the least significant byte of the *mask* integer; therefore, the integer's remaining bytes can contain anything. For simplicity, the examples in this discussion set the upper bytes to zeros.

The *mask* value is determined as follows:

1. Decide which of the parallel poll lines (the 8 data lines) you want to wait for a request for service on. Assume that the lines are labeled D0-D7.
2. Set up an 8-bit binary number where the bits associated with the lines whose assertion you want to wait for are set to 1 and all of the other 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. This is the *mask* integer you should use with *hpib_wait_on_ppoll.*

For example, assume 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 the parallel poll line D0 and device B has been configured to respond on line D4. The binary value of the *mask* that you will use is:

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

The decimal value of this number is 17; the *mask* that you will use is 17.

Now consider a *mask* of 0. It indicates that you do not want to wait for a request on any of the parallel poll lines, meaning that a call to *hpib_wait_on_ppoll* using a *mask* of 0 has no effect.

Calculating the sense

The routine *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 zeros.

The *sense* value is determined as follows.

1. Decide which of the parallel poll lines (the 8 data lines) you want to wait for a request for service on. Assume that the lines are labeled D0-D7.
2. Determine which of these lines will indicate a request for service with a 0. This means that you must know the *sense* with which the associated devices are configured to respond to parallel polls.
3. Set up 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 the other 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. This is the *sense* integer you should use with *hpib_wait_on_ppoll.*

Refer back to the example given for calculating the *mask* value. You know that device A is configured to respond on line D0 with a 1 when it wants service, but device B is going to request service with a 0 on line D4. The binary value of the *sense* that you will use is:

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

The decimal value of this number is 16; the *sense* that you will use is 16.

If all of the devices on the bus respond to parallel polls with a 1 to request service, then the *sense* value can always be 0, no matter which parallel poll lines you are waiting for. If, on the other hand, all of the devices request service with a 0, then the *sense* value can always be 255 (11111111 in binary). You need only calculate a different *sense* value if devices on the bus respond with different levels.

Example

Assume that you want to use *hpib_wait_on_ppoll* to wait until one of the four devices on a bus are requesting service. Your 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.* You want to wait for responses on lines D0, D1, D2, and D3; therefore, the *mask* value is:

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

Since the four devices on the bus use both positive and negative logic, you must calculate the *sense* value. The devices responding on lines D1 and D2 use 0 to request service; therefore, the *sense* value is:

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

Now that you have the *mask* and *sense* values you can write the code that makes the call to *hpib_wait_on_ppoll*:

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

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

In the code above, for *service_routine* to be executed, one of the four devices must be requesting service with their parallel poll response. *Service_routine* should contain code that services all of the devices, 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 byte of status is returned by the specified 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, or a printer being out of paper. The particular response of each device depends on the device.

Not all devices can respond to a serial poll. To find out if a particular device can be serially polled, consult its documentation. Trying to serially poll a device that cannot respond causes a timeout or suspends your program indefinitely.

The Active Controller cannot serial poll itself.

Unlike the parallel poll responses, serial poll responses cannot be configured remotely by the Active Controller. They are device-dependent and you must refer to a device's documentation to see how it responds.

Conducting a Serial Poll

The *hpib_spoll* routine performs a serial poll of a specified device. It is called with the form:

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

where *eid* is the entity identifier for the open interface file and *address* is the bus address of the device to be polled. The routine 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, one bit always supplies the same information. Given that the status byte's bits are labelled D0-D7, D6 always indicates whether or not the device is requesting service by asserting the SRQ line.

The code below illustrates how *hpib_spoll* can be used to find out if a device at bus address 5 is requesting service. It does this by asserting SRQ (it only looks at D6).

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

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

Errors During Serial Poll

The *hpib_spoll* routine returns a −1 indicating an error if any of the following conditions are true:

- The addressed device did not respond to the serial poll before the timeout defined by *io_timeout_ctl* occurred.
- The computer's interface is not the Active Controller.
- The *eid* entity identifier does not refer to an HP-IB raw interface file.
- The *eid* entity identifier does not refer to an open file.
- Address is outside the range [0,30].

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

Errno Value	Error Condition
EBADF	*eid* did not refer to an open file.
ENOTTY	*eid* did not refer to a raw interface file.
EIO	The device polled did not respond before the timeout or the interface was not the Active Controller.
EINVAL	Invalid bus address.

Passing Control

The current Active Controller can pass the active control capability to a **Non-Active Controller** with the *hpib_pass_ctl* routine. A **Non-Active Controller** is a device capable of becoming Active Controller, and in most cases this means it is a computer.

hpib_pass_ctl is called as follows:

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

where *eid* is the entity identifier for the open interface file (that is currently the Active Controller) and *address* is the bus address of a Non-Active Controller. Once the call is completed, the Non-Active Controller is the new Active Controller and the interface is a Non-Active Controller.

The *hpib_pass_ctl* routine only passes active control capability, it does not 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. However, the computer still loses active control. This means the bus no longer has an Active Controller. If this happens, the System Controller must assume the role of Active Controller with *hpib_abort* (see *The System Controller's Duties* section) or *io_reset.*

No error is returned by *hpib_pass_ctl* if the device that you address does not accept active control. There is also no direct way to determine in advance if a given device can accept active control. However, if the computer immediately requests service after passing control and a timeout occurs before the request is acknowledged, possibly the active control wasn't accepted. There is no way for the computer, after initiating *hpib_pass_ctl*, to see if active control is accepted.

Errors While Passing Control

The routine *hpib_pass_ctl* returns a −1 indicating an error if any of the following error conditions are true:

- The computer's interface is not the Active Controller.
- The *eid* entity identifier does not refer to an HP-IB raw interface file.
- The *eid* entity identifier does not refer to an open file.
- Address is outside the range [0,30].

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

Errno Value	Error Condition
EBADF	*eid* 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.
EINVAL	Invalid bus address.

The System Controller

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

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

Determining System Controller

To find out if your computer's HP-IB interface is the System Controller, use the *hpib_bus_status* routine. It must be called as follows:

```
hpib_bus_status(eid, 3);
```

where *eid* is the entity identifier for the open interface file and *3* indicates that you want to find out if it is the System Controller. The routine returns a 1 if it is the System Controller, a 0 if it isn't, and a −1 if an error occurs.

The code that follows prints a message indicating if the interface is the System Controller:

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

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

System Controller's Duties

The System Controller of an HP-IB bus has three major functions:

- It assumes the role of Active Controller whenever it is powered on or 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* carries out the following actions:

- It terminates activity on the bus by pulsing the Interface Clear line (IFC). This results in all talkers and listeners on the bus being unaddressed.
- It sets the REN line so that devices on the bus will be placed in their remote state when they are addressed as listeners.
- It clears the ATN line if it was left set by the previous Active Controller.
- The System Controller then becomes the bus's new Active Controller.
- Returns devices on the bus to their local state.

The routine leaves the SRQ line unchanged, which means any device requesting service before *hpib_abort* is executed is still requesting service when the routine 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 bus's Active Controller attempts to pass active control to another device that does not accept active control. This happens if the device addressed to receive control is not another controller. As a result the bus is left without any Active Controller and the System Controller must assume that role using *hpib_abort.*

hpib_abort is called as follows:

```
hpib_abort(eid);
```

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

hpib_ren_ctl

With *hpib_ren_ctl* you can enable or disable the REN line on the HP-IB. If the line is enabled, all devices that are capable of remote operation (interpreting HP-IB commands) can be placed in the 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.

When the System Controller is powered on or reset, the REN line is enabled by default. It is also enabled if the System Controller executes *hpib_abort.*

To use *hpib_ren_ctl* on a particular HP-IB , the computer must be the System Controller of 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 the open interface file and *flag* is an integer. If *flag* is zero, the REN line is disabled. If it has any other value, then REN is enabled.

Errors During hpib_abort and hpib_ren_ctl

hpib_abort and *hpib_ren_ctl* both return a -1 indicating an error if any of the following error conditions are true:

- The computer's interface is not the System Controller.
- The *eid* entity identifier does not refer to an HP-IB raw interface file.
- The *eid* entity identifier does not refer to an open file.

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

Errno Value	Error Condition
EBADF	*eid* did not refer to an open file.
ENOTTY	*eid* did not refer to a raw interface file.
EIO	The interface was not the System Controller.

The Computer As a Non-Active Controller

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

Determining the Controller's Status

The *hpib_bus_status* routine allows you to determine information about the interface card and the HP-IB. It can be used by any controller on the bus, independent of whether or not it is the Active Controller or System Controller. In the previous discussions about the Active and System Controllers, the routine is mentioned briefly. The discussion that follows should give you a broader look at the routine's uses.

hpib_bus_status is called with the form:

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

where *eid* is the entity identifier for the 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 0-7 where the values indicate the following questions:

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

If the value of *status_question* is in the range 0-6, the routine returns a 1 if the answer to the question is yes, or a 0 if the answer is no. If the value of *status_question* is 7, the routine returns the bus address of the computer's interface. If *status_question* has any other value, a −1 is returned, indicating an error.

For example, to determine if your interface is a Non-Active Controller on the bus, use the routine call illustrated by the following code:

```
:
if ((status = hpib_bus_status(eid, 4)) == -1)
   printf("Error occurred while checking status");
else if (status == 0)
     printf("Computer is a Non-Active Controller");
else
     printf("Computer is the Active Controller");
:
```

Requesting Service

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

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

where *eid* is the entity identifier for the 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 routine. Given a bit labeling of D0-D7, D6 of the lower byte sets the SRQ line. The defined values for the remaining 7 bits are application-dependent. This section only discusses the setting and clearing of the SRQ line with D6 (integer value of 64).

To request service you can invoke *hpib_rqst_srvce* as follows:

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

   eid = open("/dev/raw_hpib", O_RDWR);
   hpib_rqst_srvce(eid, 64);   /*Bit 6 of serial poll response is set*/
                                /*and SRQ is asserted                */
}
```

Note that by setting *response* to 64 the only information that the Active Controller receives when it serially polls your computer is that you are asserting the SRQ line.

The routine *hpib_rqst_srvce* returns a 0 if it executes correctly or a −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 (e.g. *hpib_rqst_srvce(eid, 0)*). After your serial poll response is configured, your computer's interface responds automatically to any serial polls from the Active Controller.

Note that if another device is asserting SRQ also, the line is still asserted after your request is removed.

If you try to request service and you are the Active Controller, the SRQ line is not set. However, if you then pass active control to another computer, the *response* that you specified with *hpib_rqst_srvce* is remembered and the SRQ line is set.

When the Active Controller sees the SRQ line asserted, it usually polls the devices on the bus to find out who is requesting service. To determine which device (or devices) is requesting service, the Active Controller conducts a parallel poll. Configuring your computer's response to a parallel poll is discussed in the next section.

If a device responds to a parallel poll with an **I need service** message, the Active Controller can perform a serial poll to determine what service action is required. If several devices are configured to respond to a parallel poll on the same line and the Active Controller sees that line is requesting service, it must perform a serial poll of each of the devices to find out which one is requesting service.

Errors While Requesting Service

Hpib_rqst_srvce returns a −1 indicating an error if either of the following error conditions are true:

- The *eid* entity identifier does not refer to an HP-IB raw interface file.
- The *eid* entity identifier does not refer to an open file.

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

Errno Value	Error Condition
EBADF	*eid* did not refer to an open file.
ENOTTY	*eid* did not refer to a raw interface file.

Responding to Parallel Polls

Before your computer can respond to a parallel poll from the Active Controller, its response must be configured. This can be programmed remotely by the Active Controller (see *The Active Controller* section) or locally using *hpib_card_ppoll_resp*.

Configuring a parallel-poll response of a device involves:

- Specifying the logic sense of the response (i.e. whether a 1 means the device does or doesn't need service).
- Specifying which data line the device responds on. More than one device can be configured to respond on the same line.

To locally configure how your computer responds to parallel polls, call *hpib_card_ppoll_resp* as follows:

```
hpib_card_ppoll_resp(eid, flag);
```

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

Calculating the Flag

The *flag* value is found by first forming an 8-bit binary number and then using the decimal value of that number. The binary number's bits have the following meaning:

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

where:

S sets the sense of the response if allowed by the hardware. If *S* is a 1, then the device responds with a 1 when it requires service.

P is a 3-bit binary number that specifies which of parallel poll response lines (D0-D7) the device responds on if allowed by the hardware.

Limitations of hpib_card_ppoll_resp

There are some hardware limitations on using *hpib_card_ppoll_resp* to configure parallel poll responses. You should refer to the Appendix for your system to find out if any restrictions apply. If there are restrictions on your system, you may find it easier to configure the interface's parallel poll response remotely with the Active Controller. Note the Active Controller can configure its own response, but the response only has effect when it passes active control.

Errors While Configuring Response

The routine *hpib_card_ppoll_resp* returns a −1 indicating an error if any of the following error conditions are true:

- The *eid* entity identifier does not refer to an HP-IB raw interface file.
- The *eid* entity identifier does not refer to an open file.
- **Series 500 Only:** The interface's parallel poll response cannot be programmatically controlled.

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* did not refer to an open file.
ENOTTY	*eid* did not refer to a raw interface file.
EINVAL	The interface cannot respond on the line indicated by *flag* (**Series 500 Only.**).

hpip_ppoll_resp_ctl

The *hpip_ppoll_resp_ctl* function allows the user to determine how the computer will respond to the next parallel poll. There are two ways to respond to a parallel poll. Responding favorably indicates to the controller that the computer wants to be serviced. Responding unfavorably indicates the computer does not need the Active Controller's attention.

The 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 the *response_value* is an integer that indicates the response to use. If *response_value* is non-zero then the computer will respond favorably to the next parallel poll. A zero *response_value* will respond unfavorably to the next parallel poll.

Disabling Parallel-Poll Response

The routine *hpib_card_ppoll_resp* also allows you to disable your interface from responding to parallel polls made by the Active Controller. This is done by setting bit D4 of the routine's flag value. When D4 is 0 the interface is enabled to respond to parallel polls, and when it is 1 the interface's parallel poll response is disabled. Thus, a flag value of 16 disables the response.

For example, the code:

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

disables the interface with the entity identifier *eid* from responding to any parallel polls.

Accepting Active Control

If your computer is a Non-Active Controller, the current Active Controller may pass active control to you. Your computer's interface accepts active control automatically; however, you must design an interfacing program to recognize when this happens.

The *hpib_bus_status*, *hpib_status_wait*, and *io_on_interrupt* routines allow recognizing the computer has become the Active Controller.

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.

hpib_status_wait is called as follows:

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

where *eid* is the entity identifier for the 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 Waiting For
1	Wait until the SRQ line is asserted
4	Wait until this computer is the Active Controller
5	Wait until this computer is addressed as a talker
6	Wait until this computer is addressed as a listener

Now imagine a situation where the current Active Controller is programmed to know that when your computer requests service, it is to pass active control to you. The following code shows how you can program your computer to request service and then wait until it becomes the bus's new Active Controller.

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

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

   if (hpib_status_wait(eid, 4) == -1)   /*wait until Active Controller*/
   {
      printf("Error while waiting for status");
      exit(1);
   }
   :                            /*Computer is now the Active Controller*/
}
```

Notice for *hpib_status_wait* to have returned a −1 (due to a timeout occurring), you would have had to set a timeout value, using *io_timeout_ctl*, after opening the interface file. Since this wasn't done in the example above, no timeout occurs.

Errors While Waiting on Status

hpib_status_wait returns a −1 indicating an error if any of the following error conditions are true:

- A timeout occurred before the condition the routine was waiting for became true.
- The *status* specified has an invalid value.
- The *eid* entity identifier does not refer to a raw HP-IB interface file.
- The *eid* entity identifier does not refer to an open file.

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

Errno Value	Error Condition
EBADF	*eid* did not refer to an open file.
ENOTTY	*eid* did not refer to a raw HP-IB interface file.
EINVAL	*status* contains an invalid value.
EIO	The specified condition did not become true before a timeout occurred.

Determining When You Are Addressed

As a Non-Active Controller you may be addressed by the Active Controller and become a bus talker or listener for data transfer. The DIL routines *hpib_bus_status*, *hpib_status_wait*, and *io_ on_interrupt* allow you to find out if the computer's interface is currently being addressed.

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

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

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

In the above call to *hpib_bus_status*, *eid* is the entity identifier for the interface and *5* indicates that you are asking if it is a bus talker. The routine returns a 1 if the answer is yes and 0 if the answer is no.

To find out if the interface is currently addressed as a bus listener use the following:

```
:
if (hpib_bus_status(eid, 6) == 1)
{
   printf("the interface is addressed as a listener");
   read(eid, buffer, 12);            /*do the data transfer*/
}
else
   printf("the interface is not addressed as a listener");
:
```

If you need to wait until the interface is addressed as either a talker or listener and then handle a data transfer, use the DIL routine *hpib_status_wait*. When you call the routine, you specify the entity identifier of the interface and the bus condition that you want to wait on:

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

As with *hpib_bus_status*, with a condition value of 5 the routine waits for the interface to be addressed as a talker. With a condition value of 6 the routine waits until it is a listener. How long the routine waits for the specified condition is controlled by the timeout value that you have previously set for the entity identifier with *io_timeout_ctl* (see discussion in Chapter 2, "General-Purpose Routines"). The routine returns a 0 if the condition became true or a −1 if a timeout (or an error) occurred first.

In the example below, the program waits for the interface to become a bus listener, and then it reads a 50-byte message.

```
#include <fcntl.h>
main()
{
   int eid, len;
   char buffer[51];                          /*storage for message*/
   eid = open("/dev/raw_hpib", O_RDWR);
   io_timeout_ctl(eid, 5000);

   if (hpib_status_wait(eid, 6) == -1)
   {
       printf("Either a timeout or an error occurred");
       exit(1);
   }

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

Note that a timeout value is set for the interface file's entity identifier in the code above so the program does not hang while waiting for the interface to be addressed as a bus listener.

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

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

   eid = open("/dev/raw_hpib",O_RDWR);
   cause_vec.cause = LTN;
   io_on_interrupt(eid, cause_vec, handler);
   :
}
handler(eid, cause_vec);
int eid;
struct interrupt_struct cause_vec;
{
   if (cause_vec.cause == LTN)
       read(eid, data, 50);
}
```

Buffering I/O Operations

The DIL routine *hpib_io* allows you to perform structures of HP-IB I/O operations for both sending HP-IB commands and transferring data. The computer's HP-IB interface must be the bus's Active Controller before this routine can be used.

A call to *hpib_io* has the form:

```
#include <dvio.h>
/* on the Integral PC, the include directive would be:
 *
 *      #include <libdvio.h>
 */
main()
{
   int eid;
   struct  iodetail *iovec;
   int iolen;
   :
   hpib_io(eid, iovec, iolen);
   :
}
```

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

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

Iodetail: The I/O Operation Template

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

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

Each of the components of *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 is to be performed on the data pointed to by the *buf* component. You determine its value by **OR**-ing constants from a set defined in the include file *dvio.h.* The constants that you can choose from are:

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 only has effect when it is **OR**-ed with HPIBREAD, while HPIBATN and HPIBEOI only have effect when they are **OR**-ed with HPIBWRITE (but not both at the same time).

The *mode* component allows you to specify under what conditions 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, the detection of the termination character determined 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* of the structure pointed to by *iovec* with:

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

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

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

The Count Component

The *count* is an integer determining the maximum number of bytes that will 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.

Set a maximum number of bytes for a structure's data transfer with:

```
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 that holds the data that will be transferred during a read operation (HPIBREAD) or is written to during a write operation (HPIBWRITE). Note the array's size limit is defined by the structure's *count* component.

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

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

Allocating Space

Before you can build *iodetail* structures for your I/O operations, you need to allocate space for them in memory. 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.

Below is the code for such a routine, *io_alloc*:

```
struct iodetail *io_alloc(n)
int n;
{
  char *malloc();
  return((struct iodetail *) malloc(sizeof(struct iodetail) * n));
}
```

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

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 that your computer's HP-IB interface is at HP-IB address 30 and it is the bus's Active Controller. You want to send a data message to a device at HP-IB address 7 and then receive a message from the same device using *hpib_io*. Four *iodetail* structures are required to do this:

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 illustrates how the 4 structures can be built and then implemented.

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

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


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


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

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

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


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


/*Build structure 4 -- Receiving data message*/
   temp++;                /*increment structure pointer*/
   temp->mode = HPIBREAD; /*read data; reaching count value terminates read*/
   temp->count = 10;      /*you expect a 10-byte message*/
   temp->buf = buffer[3];


/*Implement the I/O operations stored in the iodetail structures*/
   eid = open("/dev/raw_hpib", O_RDWR);

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


 /*Print data message you received from the device. Note temp still*/
 /*points to the last iodetail structure and the last structure did the read
*/

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

One comment about the C language: routine parameters are passed by value and not by reference; therefore, after you execute *hpib_io* the *iovec* parameter still points to the first *iodetail* structure, just as it did before the routine executed. Thus, another way to print out the data message, 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 of the I/O operations specified in the array of *iodetail* structures complete successfully, *hpib_io* returns a 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 find out during which *iodetail* structure operation the error occurred, look at the structures' *count* components. The *hpib_io* routine updates the *count* component of the structure that caused the error to be a −1. Once you have located a structure with a count of −1, you know that all of the structures previous to it were completed successfully and all of the structures after it were not executed at all.

For example, assume that you have built an array of 10 *iodetail* structures to execute a sequence of I/O operations. The following code executes the operations and then checks for errors. If an error occurs, the code prints the number of the structure that caused it (for instance, the first structure in the array is number 1).

```
#include <fcntl.h>
#include <dvio.h>
main()
{
   int FOUND, number, eid;
   struct  iodetail *iovec, *temp;
   :
   /*space is allocated for the 10 structures and then they are*/
   /*built. "Iovec" is left pointing to the first structure*/
   :
   eid = open("/dev/raw_hpib", O_RDWR); /*open the interface file*/
```

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

Notes

Controlling the GPIO Interface 4

This chapter briefly describes the actions you take to configure your GPIO interface before it can be accessed from a program using the DIL routines. It then discusses the limitations and capabilities that DIL provides for controlling the GPIO interface.

Configuring Your GPIO Interface

On Series 200/300 and 500 computers, the GPIO interface is configured via switches on the interface card. However, the Integral PC's HP 82923A GPIO interface is set using DIL routines—not switches.

Configuring the Integral PC GPIO

On the Integral PC, the HP 82923A GPIO interface is set using DIL routines. The functions that can be configured are:

- data logic sense (use *gpio_normalize* routine)
- data handshake mode (use *gpio_handshake_ctl* routine)
- delay time (use *gpio_delay_time_ctl* routine).

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

Setting the Interface Switches for Series 200/300 and 500

The GPIO interface card **for Series 200/300 and 500 computers** has several switches that allow you to configure your interface. These are fully described in the interface's installation manual. The functions they configure are:

- the data logic sense
- the data handshake mode
- the input data clock source.

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

Default Configuration and Switch Settings for the Series 800 Model 840 GPIO

The Series 800 Model 840 supports the HP 27114A AFI (Asynchronous FIFO Interface) card as its GPIO interface. Some features are set by default, and some features can be set by switches. The default configuration for the AFI card sets the following features:

- data logic sense: trigger on leading edge
- data handshake mode: full

The AFI card has several switches that allow you to configure the following features:

- delay time
- even or odd parity

Set the switches according to the directions found in the AFI installation manual.

NOTE

On some systems, the GPIO interface's select code is determined by a switch setting on the interface card. Refer to the appropriate hardware-specific appendix to see if a switch configuration is required. If a switch setting is not required, then the select code is determined by the I/O slot in which you place the interface card.

Creating the GPIO Interface File

Once you have set the necessary switches on your GPIO interface, you must install the card in your computer and create an interface file for it. Chapter 2, "General-Purpose Routines" discusses the creation of interface special files. You must create an interface file before you can access the interface from HP-UX.

Limitations on Controlling the Interface

The Device I/O Library (DIL) routines allow you to use a GPIO interface to communicate with devices that are not supported on your HP-UX system. They do not provide you with full control of the interface and because of this, you are faced with the following limitations:

- You do not have direct access to the interface's handshake lines: the Peripheral Control line (PCTL), the Peripheral Flag line (PFLG), and the Input/Output line (I/O).
- You cannot read the value of the Peripheral Status line (PSTS).
- (**Series 500 Only**) You cannot recognize interrupts sent by the peripheral on the External Interrupt Request line (EIR).

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

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

Using the DIL Routines

Several of the DIL routines can be used to control the GPIO interface. These are divided into two groups:

- general purpose routines used with either an HP-IB or GPIO interface
- GPIO routines; routines specifically designed to be used with a GPIO interface.

The general purpose routines are listed and described in Chapter 2, "General-Purpose Routines," and you should refer there for more information. They are used in this chapter to illustrate various aspects of controlling the GPIO interface from an HP-UX process.

There are two DIL routines that are restricted to use with a GPIO interface:

- *gpio_get_status*
- *gpio_set_ctl.*

These two routines allow you to use the four special purpose lines that are available on the interface for any purpose desired. The *gpio_get_status* routine reads the two lines controlled by the peripheral (STI0 and STI1) and *gpio_set_ctl* sets the values of the two lines controlled by the computer (CTL0 and CTL1). These two routines are described later in this chapter in the section *Using the Special Purpose Lines.*

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

- reset the interface
- perform data transfers
- use the interface's 4 special purpose lines
- control the data path width and data transfer speed
- set a timeout for data transfers
- set a read termination character
- get the termination reason
- set up the interrupts
- enable or disable interrupts.

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

Resetting the Interface

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

- the Peripheral Reset line (PRESET) is pulsed low
- the PCTL line is placed in the clear state
- if the DOUT CLEAR jumper is installed, the Data Out lines are all cleared (set to logical 0)
- interrupts are disabled on Series 200/300.

The lines that are left unchanged are:

- the CTL0 and CTL1 output lines
- the I/O line
- the Data Out lines, if the DOUT CLEAR jumper is not installed

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

- the read termination character is cleared
- the timeout value is set to 0
- the width for all ports is set to 8 bits
- normalization is set to positive true
- the delay time is set to 1 μ-sec
- the handshake mode is set to 1
- the data lines are set to 0
- speed is set to the flag transfer mode
- the I/$\overline{\text{O}}$ line remains unchanged.

Performing Data Transfers

Using the DIL routines *read* and *write* you can transfer bytes of ASCII data to and from the GPIO interface. The following code illustrates using these routines to first write 16 bytes of data and then read 16 bytes.

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

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

Using the Special-Purpose Lines

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

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

Driving CTL0 and CTL1

The call to *gpio_set_ctl* has the following form:

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

where *eid* is the entity identifier for the open GPIO interface file and *value* is an integer whose least significant two bits are mapped to CTL0 and CTL1.

To illustrate:

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

Both CTL0 and CTL1 are asserted low; thus, in the above example both lines are pulled low. This logic polarity cannot be changed. To raise both of the lines, call *gpio_set_ctl* with:

```
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 the open GPIO interface file. The routine returns an integer whose least significant two bits are the values of STI0 and STI1.

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're both set*/
    :
 /*insert code that handles case when both STI0 and STI1 are asserted*/
else if (bits == 1)                  /*just STI0 is asserted*/
    :
 /*insert code that handles case when STI0 is asserted*/
    :
else if (bits == 2)                  /*just 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*/
```

Note that STI0 and STI1 are asserted low; thus, when the value returned by *gpio_get_status* has one of its two least significant bits set, the associated special-purpose line is low.

Controlling the Data Path Width

The DIL routine *io_width_ctl* allows you to specify two different data path widths for your GPIO interface: 8 bits and 16 bits. The call has the following form:

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

where *eid* is the entity identifier for the open GPIO interface file and *width* is either 8 or 16. If a different *width* value is specified, the routine returns an error of −1 and *errno* is set to *EINVAL*. The GPIO interface defaults to an 8-bit path when its file is first opened.

The code below illustrates data transfers using a 16-bit data path.

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

Since the interface's data path is 16 bits, 2 ASCII characters are transferred for each handshake cycle involved. In the first 16-bit transfer, *d* is sent in the upper byte and *a* is sent in the lower. The actual logic level of the GPIO data output lines depends on how the lines have been configured.

Controlling the Transfer Speed

You can request a minimum speed for the data transfer across a GPIO interface using *io_speed_ctl*. Your system rounds the speed that you specify up to the nearest defined speed. If you specify a speed that is faster than your system allows, the highest allowable speed is used. Refer to Chapter 2, "General-Purpose Routines," for more information on using this routine. Again, the Series 500 and the Series 800 always provide DMA; therefore, the routine *io_speed_ctl* is ineffective on those systems.

In Case of a Timeout

If you have previously set a timeout value for the data transfer entity identifier, reaching the timeout after attempting a transfer will cause an error condition. If a timeout does occur, the DIL routine that you called to implement the transfer returns −1 and sets *errno* to EIO. When a timeout occurs you should reset the GPIO interface with the *io_reset* routine before attempting the transfer again.

Read Terminations

Determining Why a Read Operation Terminated

The *io_get_term_reason* routine, described in Chapter 2, "General-Purpose Routines," is used to discover why the last read performed on a particular entity identifier terminated. It tells you which of the following caused the termination:

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

Specifying a Read Termination Pattern

Chapter 2, "General-Purpose Routines," describes the routine *io_eol_ctl* which allows you to specify a character or string of characters, known as a read termination pattern, that when encountered during a read will terminate the read operation on a particular entity identifier for the GPIO interface file.

Interrupts

Chapter 2, "General-Purpose Routines," describes the routines *io_on_interrupt* and *io_interrupt_ctl.* These routines allow you to set up and control interrupt handlers for the GPIO status line or a particular *eid* for the GPIO interface file.

Interrupt-Driven Transfer Mode

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

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

Series 500 Dependencies

A

The following information, specific to the Series 500, is discussed in this appendix:

- the location of the DIL routines
- information about creating the special file for the interfaces that you plan to access with DIL routines
- the relationship between entity identifiers and file descriptors
- the restrictions imposed by the hardware on using the DIL routines
- information about how you can improve the performance of your I/O process

Location of the DIL Routines

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

The GPIO Interface

The **GPIO** (General Purpose Input/Output) interface is a very flexible parallel interface that allows communication with a variety of devices. On the Series 500, 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.

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

Special-Purpose Lines

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

Data Handshake Methods

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

Data-In Clock Source

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

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

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

Creating the Interface Special File

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

Creating an Interface File

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

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

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

pathname

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

major_number

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

Major Number	Interface
12	HP 27110A/B HP-IB Interface
18	HP 27110A GPIO Interface
37	Internal 550 HP-IB Interface.

minor_number

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

```
0xScAdUV
```

where:

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

`Sc` a two-digit hexadecimal value specifying the select code of the interface card. The select code corresponds to the I/O slot in which the interface card resides.

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

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

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

Creating an HP-IB Interface File

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

- the pathname is */dev/raw_hpib*
- the HP-IB interface is internal—the major number is 12
- the card is placed in slot 2, giving a select code 02—i.e., the `Sc` component of the minor number is 02
- the special file must be a **raw** special file in order to use DIL library routines 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 12 0x021f00
```

To further illustrate the use of *mknod*, suppose you have two HP-IB HP 27110A interface cards (major number = 12) installed in slots 2 and 3. The following *mknod* commands set up a special file for the interface at select code 02 (*/dev/raw_hpib1*) and select code 03 (*/dev/raw_hpib2*):

```
mknod /dev/raw_hpib1 c 12 0x021f00

mknod /dev/raw_hpib2 c 12 0x031f00
```

Creating a GPIO Interface File

Now suppose you have a GPIO interface that you want to access with the DIL routines on the same Series 500 computer.

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

Assume that you have placed the interface in the I/O slot on your Series 500 corresponding to select code 04. The following *mknod* command will create the appropriate special file, named */dev/raw_gpio*:

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

Determining The Bus Address of the Interface Card

The HP 27110A/B card always assumes bus address 30 when it is the Active Controller. If control is passed, then it assumes the address specified by the cards switch setting. However, the *hpib_bus_status* routine always returns the correct bus address.

Effects of Resetting (via io_reset)

For an HP-IB interface on a Series 500 computer, resetting involves clearing REN, pulsing its Interface Clear line (IFC), and resetting REN; for a GPIO interface the Peripheral Reset line (PRESET) is pulsed. The routine also causes the interface to self-test. If it fails its test, the routine returns a −1; if the interface successfully resets and completes its self-test, the routine returns a 0.

Entity Identifiers

On the Series 500, 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 routines *dup*, *fcntl*, and *pipe*, in addition to *open*.

Restrictions Using the DIL Routines

This section presents some restrictions on using the DIL routines on the Series 500 computers. These restrictions are organized under the routine to which they apply. The routines are presented in alphabetical order.

hpib_bus_status

A bug in the HP 27110A HP-IB interface card can cause an erroneous report of the state of the SRQ line. There is a small window when *hpib_bus_status(eid, 1)* reports that the line is clear when in reality it is set. Since the routine will never report that the line is set when in reality it is clear, **OR**-ing together successive readings of the state of the SRQ line minimizes the possibility of error. **OR**-ing five successive readings gives you a result that is approximately 99% accurate. This bug has been fixed in the HP 27110B card.

On the Series 500, it is possible to look at the SRQ line with *hpib_bus_status* and not see it asserted when it actually is. Because of this, you should check the SRQ line at least 5 times before determining whether or not it is asserted. If it is seen true any one of the 5 times, then the line is asserted (it will never be seen asserted when it actually isn't). For example:

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

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

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

hpib_card_ppoll_resp

The HP 27110A/B HP-IB interface cards do not support programmatic configuration of their parallel poll response. The parallel poll response is set and enabled by the *hpib_card_ppoll_resp* routine. The default *sense* of the HP 27110A/B interface's parallel poll response is always 1. If the interface's address is 7 or less, the address determines the response's line number as follows: given that the bus data lines are labeled D0 through D7, they correspond to addresses 7 through 0, respectively. For instance, the parallel poll response of an HP 27110A/B with address 0 is a 1 on data line D7. If its address is 7 then it responds with a 1 on line D0. If the address of the interface is greater than 7, there is no default line for it to respond on. Therefore, unless its response is configured remotely by the Active Controller, it can not respond at all.

If you want the interface to respond with a sense of 0 or on a different line than HP 27110A/B defaults to, you must configure it remotely with the Active Controller

hpib_rqst_srvce

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

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

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

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

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

clears it.

hpib_send_cmnd

The HP 27110A/B HP-IB and Series 550 Internal HP-IB interface cards send all the commands you specify with this routine, with odd parity. 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 parity.

hpib_status_wait

The *hpib_status_wait* routine, when processing, holds off all other activity on that interface card. Other processes attempting to access the interface card will hang. It is strongly recommended that a non-zero timeout be in effect before calling *hpib_status_wait.*

hpib_wait_on_ppoll

The *hpib_wait_on_ppoll* routine, also, holds off all other activity on the interface card. Again, other processes attempting to access the interface card will hang and it is recommended that a non-zero timeout be in effect before calling *hpib_wait_on_ppoll.*

io_get_term_reason

Normally, this routine can indicate multiple reasons for a read termination by the values of the least significant three bits in its returned value:

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

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

The *io_get_term_reason* routine on the Series 500 has a limitation when a read is terminated for multiple reasons; it can only indicate one termination cause at a time. If a read terminates for multiple reasons, the value returned by *io_get_term_reason* is the value of the highest numbered reason. Thus, on the Series 500 the routine can only return a 0, 1, 2, or 4 (or a −1 if the routine itself fails). For instance, if a 4 is returned, you know that a device-imposed termination condition occurred, but you do not know if the byte count was reached or if a termination character was read as well.

On the Series 500, if you set a termination character for a GPIO interface that is using a 16-bit data path, only an 8-bit termination character is set (the least significant byte of the match value). During read operations, if the termination character is then seen as the lower byte in a data transfer, everything works correctly; both the upper and lower bytes of the transfer are received and the count of received bytes is incremented by two. However, if the termination character is seen as the upper byte of the transfer, both the upper and lower bytes are still read. The count of received bytes is only incremented by one though, indicating that the termination character was in the upper byte.

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 500 the timeout is rounded up to the nearest 10-millisecond boundary. For example, if you specify a timeout of 155000 microseconds (155 milliseconds), the effective timeout is rounded up to 160 milliseconds.

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

io_speed_ctl

The Series 500 always provides DMA for the fastest possible I/O speeds. Therefore, *io_speed_ctl* has no affect on the Series 500.

io_width_ctl

Although this routine is designed to be used on any interface, the path width that you specify with it must be supported on the particular interface. On the Series 500, only the GPIO interface allows you to change data path widths and only two widths are currently supported: 8 bits and 16 bits. The routine returns an error if you access a GPIO interface with any width besides 8 or 16 bits or if you access any other interface with a width other than 8 bits.

Performance Tips

The performance of your I/O process on a Series 500 that uses DIL routines can be improved by following the basic guidelines listed below.

- Use buffers to hold data that you write to an interface. Transferring data that you have previously stored in a buffer is faster than if you specify the data string when you invoke the transfer. For example, the data transfer performed by the code:

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

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

  is faster than the data transfer performed by the code:

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

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

- Make the number of bytes transferred divisible by the number of bytes per word that your system supports. Data transfers, both reading and writing, are faster if the number of bytes involved in the transfer falls on a word boundary. The Series 500 supports 4-byte words; therefore, the following code has an optimized performance because the byte counts are divisible by 4.

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

- If you are the super-user, you can use the *memlck(2)* routine (see *HP-UX Reference: Section 2*) to lock your I/O process's address space into physical memory. Data transfer times are reduced because they are carried out directly from the user area and do not have to be first moved to the system area. However, you cannot lock an arbitrarily large amount of space for your process since there is a point at which your system's performance will begin to degrade.

- For processes running with an effective user ID of super-user, it is possible to lock the process in memory with *plock(2)* (see *HP-UX Reference*). This lock is different than *memlck* (as mentioned above). *plock(2)* informs the system that the process code, data, or both are not to be swapped out of memory. The following example illustrates the use of *plock*:

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

- Use auto-addressing for all read and write operations. (See the section "Setting up Talkers and Listeners" of Chapter 3, "Controlling the HP-IB Interface," for details.)

- Increasing the system priority of an I/O process can be accomplished by using *rtprio(2)*. *rtprio* requires the process to be running with an effective user *ID* of super-user. The real time priorities available with *rtprio* are non-degrading priorities. Caution must be observed when using real time priorities since one can increase their priority above system processes. This may cause undesirable behavior. For example, requesting a real time priority in the range of 0-63 places your process in a higher priority than the DIL interrupt handler system process. This means that interrupts could be lost if there is not sufficient CPU resource available. The following example places the calling process at the lowest (least important) real time priority:

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

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

Series 200/300 Dependencies B

The following information, specific to Series 200/300 computers, is discussed in this appendix:

- the location of the DIL routines
- information about creating the special file for the interfaces that you plan to access with DIL routines
- the relationship between entity identifiers and file descriptors
- the restrictions imposed by the hardware on using the DIL routines
- information about how you can improve the performance of your I/O process
- information on how to simulate i/o interrupt programming on Series 200/300 computers.

Location of the DIL Routines

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

Linking DIL Routines

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

The GPIO Interface

The **GPIO** (General Purpose Input/Output) interface is a very flexible parallel interface that allows communication with a variety of devices. On Series 200/300 computers, the interface sends and receives up to 16 bits of data with a choice of several handshake methods. External interrupt and user-definable signal lines provide additional flexibility.

The GPIO interface is comprised of the following lines:

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

Data Lines

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

Handshake Lines

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

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

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

Special-Purpose Lines

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

Data Handshake Methods

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

Data-In Clock Source

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

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

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

Creating the Interface Special File

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

Creating the Special File

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

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

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

pathname

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

major_number

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

Major Number	Interface
21	HP-IB Interface
22	GPIO Interface

minor_number

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

```
OxScAdUV
```

where:

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

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

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

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

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

Creating an HP-IB Interface File

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

- the pathname is */dev/raw_hpib*
- because the interface is HP-IB, the major number is 21
- the card's select code switches are set to select code 2—i.e., the `Sc` component of the minor number is 02
- the special file must be a **raw** special file in order to use DIL library routines with it; therefore, the `Ad` portion of the minor number must be 31 (1f in hexadecimal).

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

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

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

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

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

Creating a GPIO Interface File

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

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

Assuming that you have set the interface select code switches to 04 on the Series 200/300 GPIO card, the following *mknod* command will create the appropriate special file, named */dev/raw_gpio*:

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

Effects of Resetting (via io_reset)

For an HP-IB interface on Series 200/300 computers, resetting involves clearing REN, pulsing its Interface Clear line (IFC), and resetting REN; for a GPIO interface the Peripheral Reset line (PRESET) is pulsed. If it fails, the routine returns a −1; otherwise the routine returns a 0.

Entity Identifiers

On Series 200/300 computers, an entity identifier for a file used by a DIL routine is equivalent to an HP-UX file descriptor. This means that you can obtain entity identifiers for your interface files with the system routines *dup*, *fcntl*, and *creat*, in addition to *open*.

Restrictions Using the DIL Routines

This section presents some restrictions on using the DIL routines on Series 200/300 computers. These restrictions are organized under the routine to which they apply. The routines are presented in alphabetical order.

hpib_io

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

hpib_send_cmnd

The Series 200/300 HP-IB interface card uses odd parity when you send commands via *hpib_send_cmd*. To do this, it overwrites the most-significant bit of each command byte with a parity bit. This should not cause a problem since all HP-IB commands use only 7 bits, and the eighth is free for use as a parity bit.

hpib_status

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

io_interrupt_ctl

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

io_on_interrupt

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

io_reset

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

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

io_speed_ctl

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

IMPORTANT

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

---

io_timeout_ctl

This routine allows you to set a time limit for I/O operations on an entity identifier associated with an interface file. The timeout value that you specify is a 32-bit long integer that indicates the length of the timeout in microseconds. However, the resolution of the effective timeout is system-dependent. On the Series 200/300 computers the timeout is rounded up to the nearest 20-millisecond boundary. For example, if you specify a timeout of 150000 microseconds (150 milliseconds), the effective timeout is rounded up to 160 milliseconds.

Performance Tips

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

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

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

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

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

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

Simulating Interrupts for the HP-IB Interface

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

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

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

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

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

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

Some additional points about simulating interrupts in this way are:

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

Simulating Interrupts on the GPIO Interface

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

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

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

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

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

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

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

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

   }
}
```

Integral PC Dependencies — C

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

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

Location of the DIL Routines

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

The GPIO Interface

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

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

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

Creating an Interface Special File

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

GPIO Interface Files

The special files for GPIO interfaces have the following form:

`/dev/gpio`*GPIO_port.IO_port*

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

HP-IB Interface Files

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

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

Unloading the DIL Drivers

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

Interrupts

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

Controlling the HP-IB Interface

Limitations on the HP-IB Interface

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

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

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

The Computer as a Non-Active Controller

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

Non-Standard DIL Routines

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

General-Purpose Routines

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

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

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

Non-Standard HP-IB Routines

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

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

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

Non-Standard GPIO Routines

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

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

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

Restrictions Using the DIL Routines

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

hpib_bus_status

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

hpib_card_ppoll_resp

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

hpib_ppoll_resp_ctl

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

io_eol_ctl

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

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

io_reset

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

io_speed_ctl

GPIO

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

DMA transfers are not available on the Integral PC.

Interrupt-Driven Transfer Mode

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

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

HP-IB

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

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

io_timeout_ctl

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

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

io_width_ctl

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

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

GPIO Data Path Widths

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

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

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

open(2)

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

read(2) and write(2)

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

Series 800 Model 840 Dependencies D

The following information, specific to the Device I/O Library (DIL) on Series 800 Model 840 computers, is discussed in this appendix:

- compiling programs that use DIL routines
- accessing the special files for the interfaces that you plan to use with DIL
- creating special files for the interfaces that you plan to use with DIL
- DIL routines affected by the Series 800 Model 840 hardware
- DIL support of HP-IB auto-addressed files
- improving performance of DIL programs

Compiling Programs That Use DIL

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

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

Accessing the Interface Special Files

The Series 800 Model 840 kernel is shipped with a default I/O configuration. This means a default set of special files is made for you. For example, the */dev/hpib* directory contains special files created for use with HP-IB instruments connected to the HP 27110B HP-IB interface. The special file */dev/gpio0* is created for use with instruments or peripherals connected to the HP27114A Asynchronous FIFO interface (AFI). The *insf* command is used to install these special files all at one time. *Mknod* could also be used to create them one at a time. For more information on *insf* and *mknod* refer to the *HP-UX Reference.*

Major Numbers

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

Minor Numbers and Logical Unit Numbers

Drivers use minor numbers to map the hardware I/O cards to their locations in the Model 840 I/O backplane. The default I/O configuration shipped with your Model 840 creates special files accessing a subset of the available backplane slots. For the HP-IB card, two slots are available for instrument I/O, and one slot is available for the AFI card. Slot information is accessed through the device's *logical unit* number. The logical unit number is mapped into the special file's minor number. For HP-IB special files, the HP-IB bus address is also mapped into the minor number.

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

```
0x00LuBa
```

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

The minor number syntax for an AFI special file is:

```
0x00Lu00
```

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

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

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

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

Listing Special Files

The Series 800 Model 840 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 Model 840 backplane.

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

```
$ lssf /dev/hpib/0a16
instr0 lu 0 bus address 16 address 8.2.16 /dev/hpib/0a16
```

where *instr0* is the name of the instrument HP-IB driver, the logical unit number is *0*, the HP-IB bus address is *16*, and the backplane address of the HP-IB card is *8.2.16.* This says that the CIO channel card is in mid-bus address *8*, and the HP-IB card should be in slot *2* of that CIO channel. There are 12 CIO slots available, numbered 0-11. The last digit, in this case *16*, is the HP-IB bus address of the device *0a16.*

The default HP-IB special files are set up for cards in slot 2 or slot 7 of the CIO channel at mid-bus address 8. A special file for each possible bus address (0-31) is made for each card. The special files for the card at slot 2 all have a logical unit number of 0, and the special files for the card in slot 7 all have a logical unit number of 1.

The default GPIO special file is set up for an AFI card in slot 5 of the CIO channel at mid-bus address 8, and uses a logical unit number of 0.

For more information on *lssf* refer to the *HP-UX Reference.*

Naming Conventions for Interface Special Files

If your Series 800 Model 840 computer was configured correctly, the special files discussed above will already have been created.

By convention, HP-IB special files reside in the */dev/hpib* directory. Also by convention, the default special files for the HP-IB *raw bus* (a HP-IB card itself) are named */dev/hpib/***X**, where **X** is the bus's logical unit. *Auto-addressed* files are named */dev/hpib/***X***a***Y**, where **X** is the logical unit, *a* stands for an auto-addressed file, and **Y** is the file's associated HP-IB bus address (see the "DIL Support of HP-IB Auto-Addressed Files" section of this appendix).

The naming convention for the GPIO default special files is */dev/gpio***X**, where **X** is the device's logical unit.

If you cannot locate the default special files on your system, refer to the next section for how to create them.

Creating Interface Special Files

If the special files you need for HP-IB or GPIO are not available on your system, you can use the *mksf* command to create them. *Mksf* is a high-level command implemented for the Series 800 Model 840, that can be used instead of *mknod.* Like *lssf*, *mksf* frees you from having to know the major number and minor number format. *Mksf* makes the special file creation process consistent for all classes of devices. The syntax of *mksf* is:

`mksf -d` **driver** -l **lu** *other_flags...* **sfname**

where **driver** is the name of the driver associated with the special file, **lu** is the file's logical unit, and **sfname** is the name of the special file you wish to create.

Each class of device can have additional class-dependent attributes (such as the bus address for an HP-IB auto-addressed file).

For HP-IB devices, the driver is *instr0.* Thus, to create a special file named */dev/bus* for HP-IB lu 1, you use the command:

```
mksf -d instr0 -l 1 /dev/bus
```

When creating auto-addressed HP-IB special files, you add another option **-a** to associate the address with the device. For example, to create an auto-addressed special file called */dev/plotter*, at bus address 7 on HP-IB lu 2, you could type:

```
mksf -d instr0 -l 2 -a 7 /dev/plotter
```

For the AFI card, the driver is *gpio0.* Thus, to create a special file named */dev/afi* for GPIO lu 0, you could use the command:

```
mksf -d gpio0 -l 0 /dev/afi
```

For more information on *mksf* or *mknod*, refer to the *HP-UX Reference.*

Hardware Effects on DIL Routines

The HP-IB card supported on the Series 800 Model 840 is the HP 27110B HP-IB interface; the GPIO card is the HP 27114A Asynchronous FIFO Interface (*AFI*).

This section presents some restrictions on using the DIL routines on Series 800 Model 840 computers. These restrictions are organized under the DIL routine to which they apply. The routines are presented in alphabetical order. A list of *errno* error names can be found in section two of the *HP-UX Reference*. *Errno* numeric values are defined in the file */usr/include/sys/errno.h*.

hpib_rqst_srvce

The *hpib_rqst_srvce* routine only permits bit 6 of the serial poll *response* to be set. If *hpib_rqst_srvce* is called with a *response* having bit 6 set, the interface sends <01000000> (64 decimal) in response to serial poll; if bit 6 is not set in *response*, the interface sends <10000000> (128 decimal). See "The Computer as a Non-Active Controller" in Chapter 3.

io_eol_ctl

The AFI driver does not support pattern matching on reads; all *io_eol_ctl* calls return -1 and set *errno* to **EINVAL**.

io_reset

When an HP-IB interface is reset via *io_reset*, the card's parallel poll response is set to 0; its serial poll response is set to 128; its HP-IB address is read off the hardware switches; and the card is put on-line. Any enabled interrupts are preserved. If the card is configured as system controller, then Interface Clear (IFC) is pulsed and Remote Enable (REN) is asserted.

When an AFI interface is reset via *io_reset*, each of the three control output lines is reset to zero, the incoming Attention Request (ARQ) is disabled, the ARQ flip flop is cleared, the ARQ enable flip flop and the handshake to the peripheral are disabled, and the FIFO buffer is flushed out.

io_speed_ctl

The *io_speed_ctl* routine is not supported on Series 800 Model 840 computers; transfer is always done via DMA.

io_timeout_ctl

On Series 800 Model 840 computers, the timeout you specify via *io_timeout_ctl* is rounded up to the nearest 10-millisecond boundary. For example, if you specify a timeout of 125000 microseconds (125 milliseconds), the effective timeout is rounded up to 130 milliseconds.

DIL functions, *read*, or *write* requests that time out, return a value of -1 and set *errno* to either **ETIMEDOUT** or **EINTR**. If the request can be aborted normally, then *errno* is set to **ETIMEDOUT** . Otherwise, the HP-IB card is reset and **EINTR** is returned.

io_width_ctl

The only allowable data path width for HP-IB devices is 8. AFI devices support 8-bit and 16-bit data paths. If you specify any other width, *io_width_ctl* returns an error indication.

Return Values for Special Error Conditions

On specific error conditions, the Series 800 Model 840 sets *errno* values which are different from what is expected from the DIL as documented in the HP-UX Standard. For example, when any request times out, *errno* is set to **ETIMEDOUT** ("connection timed out") or instead of setting it to **EOI**. Also, upon HP-IB requests that require the interface to be the active controller or the system controller, set *errno* to **EACCES** ("permission denied"). Requests that are aborted due to system power failure set *errno* to **EINTR** ("interrupted system call"); in addition, your process receives the signal **SIGPWR**, which indicates recovery of system power.

DIL Support of HP-IB Auto-Addressed Files

As noted in Chapter 3 in the section called "Setting Up Talkers and Listeners," one class of HP-IB special files, known as *auto-addressed* files, are associated with a given address on the bus. For *read* and *write* requests to these files, addressing is done automatically; that is, the sequence of talk and listen bus commands is generated for you.

In general, the DIL functions are not defined for auto-addressed files. On the Series 800 Model 840, however, many of them are implemented, but with more device-oriented actions.

IMPORTANT

The DIL Standard does not currently specify a functional definition for the support of auto-addressed files. When support for auto-addressed files becomes part of the DIL Standard, the specific functionality implemented may differ from the implementation described here for the Series 800 Model 840. Please keep this in mind when developing programs which take advantage of this new functionality.

The following table shows which DIL functions are supported on auto-addressed files. Entries in the first column work the same on both auto-addressed and non-auto-addressed (also called *raw bus*) files. Entries in the second column are somewhat different for auto-addressed files; entries in the third column are not supported on HP-IB auto-addressed files and will return an error indication if used.

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

Those functions in the second column, which operate differently on raw bus and auto-addressed special files, are discussed below.

hpib_card_ppoll_resp

Calling *hpib_card_ppoll_resp* on an auto-addressed file does not configure the HP-IB interface card; rather, it configures the device associated with the file with the appropriate addressing and Parallel Poll configuration commands.

hpib_io

For those *iodetail* structures that send commands (by setting the *mode* flag to HPIBWRITE or HPIBATN), *hpib_io* prefixes the command buffer *buf* with the appropriate device addressing (see *hpib_send_cmd*, below). For data transfers (with *mode* set to HPIBREAD or HPIBWRITE) using auto-addressed files, the addressing is also done for you.

hpib_ren_ctl

Setting REN (by setting the *flag* parameter to a non-zero value) on an auto-addressed file addresses the associated device *before* asserting REN. Clearing REN (by setting *flag* to a zero) addresses the device and sends it a Go To Local command, in lieu of clearing REN.

hpib_send_cmd

Sending HP-IB commands to an auto-addressed file via *hpib_send_cmd* does the appropriate device addressing for you. The *command* buffer you pass down to the device is preceded by the commands necessary to remove any previous listeners on the bus, address the Active Controller to talk, and configure the file's associated device to listen.

hpib_spoll

Performing a serial poll on an auto-addressed file polls the associated device; any bus address passed via the *ba* argument is ignored.

hpib_wait_on_ppoll

For auto-addressed files, the *mask* argument is ignored; only the address associated with the device is polled. In addition, the *sense* argument only specifies the sense of the particular device's assertion. Successful completion of the *hpib_wait_on_ppoll* request implies that the device responded to parallel poll.

io_on_interrupt

The only allowable interrupt for auto-addressed files is SRQ.

Performance Tips

DIL performance improvements for the Series 800 Model fall into two categories: those that keep your process from waiting for resources, and those that actually improve your I/O performance. The first three of the tips described below fall into the first category; the last two are in the second category.

Process Locking

Normally, the operating system swaps processes in and out of memory; you can circumvent this swapping by using the *plock* system call.

If you are running as the super-user (or have the **PRIV_MLOCK** capability), you can use *plock* to lock your process in memory; *plock* prevents the system from swapping out the process's code, data, or both.

The following example illustrates its use:

```
 #include <sys/lock.h>
 int plock();

 main() {

plock(PROCLOCK); /* lock text and data segments into memory */
:
plock(UNLOCK);    /* unlock the process */
 }
```

Refer to *plock(2)* and *getprivgrp(2)* in the *HP-UX Reference* for more information.

Setting Real-Time Priority

The operating system schedules processes based on their priority. Under normal circumstances, the priority of a process drops over time, allowing newer processes a greater share of CPU time. You can assign a higher priority to your process and keep its priority from dropping by using the *rtprio* system call.

If you are running as the super-user (or have the **PRIV_RTPRIO** capability), you can use *rtprio* to give your process a real-time priority. Real-time processes run at a higher priority than normal user processes; they get preempted only by voluntarily giving up the CPU or by being interrupted by a higher priority process or interrupt.

You must be careful when using real-time priorities because you can increase your priority above those of important system processes. The following example places the calling process at the lowest (least important) real-time priority:

```
#include <sys/rtprio.h>
#define  ME 0 /* a zero process ID means this process */
int rtprio();

main() {
    rtprio(ME, 127);           /* Turn on real-time priority for ME  */
    :
    rtprio(ME, RTPRIO_RTOFF); /* Turn off real-time priority for ME */
}
```

Refer to *rtprio(2)* and *getprivgrp(2)* in the *HP-UX Reference* for more information.

Preallocating Disc Space

If your process is reading large amounts of data and writing it to a file, you can block while the operating system allocates disc space. However, you can allocate disc space in advance by using the *prealloc* system call. The following example opens a file and preallocates 65536 bytes of space for that file:

```
#include <fcntl.h>
#define MAX_SIZE 65536
int prealloc();

main() {
    int eid;

    eid = open("data_file", O_WRONLY);
    prealloc(eid, MAX_SIZE);    /* preallocate space to write into */
    :
}
```

Refer to *prealloc(2)* in the *HP-UX Reference* for more information.

Reducing System Call Overhead

Most DIL function calls you make on the Series 800 map into system calls. Therefore, you can cut down on operating system overhead by using fewer library calls. In particular, use auto-addressed files for all read and write operations, rather than using an extra call to *hpib_send_cmd* to do addressing.

Setting Up Faster Data Transfers

Because of the I/O architecture of the Series 800, data transfers run more efficiently if your data buffers are aligned on a page boundary. The number of bytes per page is defined as **NBPG** and can be referenced by including *<sys/param.h>*. The following example shows how to allocate and page-align a data buffer:

```
#include <sys/param.h> /* defines NBPG and roundup(x, y)        */
#define REAL_SIZE 1024 /* amount of memory we want to page-align */
char *malloc();

main() {
    char *malloc_ptr, *align_ptr;

    :
    malloc_ptr = malloc(NBPG + REAL_SIZE);   /* allocate memory      */
    align_ptr  = roundup(malloc_ptr, NBPG);  /* and round up the ptr  */
  /* in future data transfers, use align_ptr      */
    :
    free(malloc_ptr);  /* when we're done with the data */
}
```

In addition, even count transfers run more quickly than odd count transfers.

Character Codes

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

STD-LL-60182

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

Character Codes (cont.)

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

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

Index

a

accessing interface special files 140
active control, accepting 77
active controller 41, 43, 45–46, 52
addressed commands 36, 37
allocating space 87
ASCII character codes 153–154
asserted lines 9
ATN 11
auto-addressed files 47, 147–149

b

buffered I/O operations 83–90
buffered I/O operations, locating errors 90
bus commands 37
bus management control lines 11

c

calculating listen addresses 48–49
calculating talk addresses 48–49
call incompatibility 5
calling DIL routines from FORTRAN 5
calling DIL routines from Pascal 4
character codes 153–154
clearing HP-IB devices 54
closing an interface special file 17
configuring the integral PC GPIO 93
control passing 67
controller 8
controller, active 41, 43, 45–46, 52
controller, nonactive 41, 71–82, 134
controller status 71–72
controller, system 41, 68–71
controlling I/O parameters 23–28
creating interface special files 119, 133, 144

d

data handshake methods ... 104, 117
data lines ... 104, 116
data path width ... 23, 25, 100
data transfers ... 23, 52, 98, 152
data-in clock source ... 105, 118
DAV ... 9, 10
determining active controller ... 46
determining controller status ... 71–72
determining why a read terminated ... 29–30, 101
DEVICE CLEAR ... 38
DIL drivers ... 133
DIL routines ... 3–5, 14, 40–42, 96, 103, 115, 123, 132
DIL routines, hardware effects ... 145
DIL routines, nonstandard ... 135
disabling parallel poll response ... 59, 76
disabling read termination pattern ... 28
disc space ... 151

e

EBADF ... 45, 61, 66, 68, 74, 76, 79
eid ... 14
EINVAL ... 66, 68, 76, 79
EIO ... 24, 45, 61, 66, 68, 79
enabling local control ... 51
ENOTTY ... 45, 61, 66, 68, 74, 76, 79
entity identifier (eid) ... 14, 122
EOI ... 11
errno ... 19
error checking routines ... 19–21
error handlers ... 21
errors during parallel polling ... 60
errors during serial polling ... 66
errors in buffered I/O operations ... 90
errors while configuring response ... 75
errors while passing control ... 67
errors while requesting service ... 74
errors while sending commands ... 44–45
errors while waiting on status ... 78
external declaration ... 4

f

flag-driven mode ... 102
FORTRAN calls to DIL routines ... 5

g

general-purpose routines ... 14–15
GO TO LOCAL ... 39
GPIO interface ... 12, 93–102, 103–105, 116–118, 132

h

handshake lines ... 9, 104, 117
handshake process ... 7
HP-IB commands ... 36–39
HP-IB devices ... 8
HP-IB interface ... 8–12
hpib_abort ... 40
HPIBATN ... 85
hpib_bus_status ... 40, 136
hpib_card_ppoll_resp ... 40, 110, 136, 149
HPIBCHAR ... 85
HPIBEOI ... 85
hpib_eoi_ctl ... 40
hpib_io ... 40, 123, 149
hpib_pass_ctl ... 40
hpib_ppoll ... 40
hpib_ppoll_resp_ctl ... 40, 136
HPIBREAD ... 85
hpib_ren_ctl ... 40, 149
hpib_rqst_srvce ... 40, 110, 145
hpib_send_cmd ... 149
hpib_send_cmnd ... 40, 111, 123
hpib_spoll ... 40, 149
hpib_status ... 123
hpib_status_wait ... 40, 111
hpib_wait_on_ppoll ... 40, 111, 149
HPIBWRITE ... 85

i

IFC ... 11
integral PC dependencies ... 131–138
integral PC interrupts ... 31
interface functions ... 7
interface special files ... 13, 105, 140
interrupt-driven transfer mode ... 102
interrupts ... 31–34, 102, 134
interrupts, Series 500 ... 31
interrupts, simulating for GPIO interface ... 128
interrupts, simulating for HP-IB interface ... 126
io_burst ... 40
io_eol_ctl ... 15, 23, 136, 145
io_get_term_reason ... 15, 29, 111
io_interrupt_ctl ... 15, 33, 123
io_on_interrupt ... 15, 32, 123, 149
io_request ... 136
io_reset ... 15, 22, 124, 145
io_speed_ctl ... 15, 23, 112, 124, 137, 146
io_timeout_ctl ... 15, 23, 112, 124, 137, 146
io_width_ctl ... 15, 23, 112, 138, 146

l

linking DIL routines ... 3, 116
listen addresses ... 48–49
listener ... 47
listeners ... 8
listing special files ... 142
local control ... 51
local control, locking out ... 51
LOCAL LOCKOUT ... 38
local state ... 50

m

memory allocation ... 87

n

naming conventions for special files 143
NDAC 9, 10
nonactive controller 41, 71–82, 134
nonstandard DIL routines 135
not asserted lines 9
NRFD 9, 10

o

onionskin routine 5
opening an interface special file 15–16
opening the HP-IB interface file 42

p

PARALLEL POLL CONFIGURE 39
PARALLEL POLL DISABLE 39
PARALLEL POLL ENABLE 39
parallel polling 57–68, 74
parallel polling, disabling 59, 76
parameter-passing irregularities 5
Pascal calls to DIL routines 4
passing control 67
PIR 31
polling, parallel 57–68, 74
polling, serial 65–66
preallocating disc space 151
process locking 150

r

read termination 29–30, 101
read termination pattern 23, 26–28
reading 18–19
real-time priority 151
remote enable line (REN) 36
remote state 50
REN 11, 36
resetting an interface 22, 97
resetting devices 54

s

secondary commands 36, 37
SELECTED DEVICE CLEAR 39
sending HP-IB commands 43–45
SERIAL POLL DISABLE 38
SERIAL POLL ENABLE 38
serial polling 65–66
Series 200/300 dependencies 115–129
Series 500 dependencies 103–114
Series 500 interrupts 31
Series 800 Model 840 dependencies 139–152
service request 54, 73
setting data path width 25
setting interface switches 93
setting I/O timeout 23–24
setting read termination pattern 26–28
setting real-time priority 151
setting transfer speed 25–26
setting up talkers and listeners 47
simulating interrupts 126, 128
space allocation 87
special files 15, 17, 105, 119, 133, 140
special-purpose lines 104, 117
SRQ 11
SRQ line 55
status byte message 65
system call overhead 152
system controller 41, 68–71

t

talk addresses 48–49
talk and listen addresses 36, 37
talkers 8, 47
terminating character 11
termination character 111
timeout, setting 23
transfer speed 23, 25–26, 101
transferring data 52, 98
TRIGGER 39, 52
triggering devices 52

u

universal commands 36, 37
UNLISTEN 38
unload_gpio 133
unload_hpib 133
unloading DIL drivers 133
UNTALK 38

w

writing 18–19

Table of Contents

Chapter 1: Overview

Program Overview 1
Manual Overview 2
Chapter 1: Overview 3
Chapter 2: Hardware Configuration 3
Chapter 3: Software Configuration 3
Chapter 4: Uucp File Structure 3
Chapter 5: Uucp Daemons 3
Chapter 6: Using The Uucp Facility 4
Chapter 7: The X.25 Network 4
Chapter 8: Log, Status and Cleanup 4
Chapter 9: Problems 4
Where To Start? 5
If You Are a User 5
If You Are the System Administrator 5
Additional Networks 6
The RJE Network 6
The Local Area Network 7
HP AdvanceNet 8

Chapter 2: Hardware Configuration

Series 500 ASI Card 9
Series 500 MUX Cards 9
HP-UX Series 800 MUX Card 10
HP-UX Series 200 and 300 MUX Cards 10
Modem Connections 13
Direct Connections 15
Making a Special Connector 27

Chapter 3: Software Configuration

Software Loading and Setup 29
General Startup Information 30
Creating a TTY Device File 32
Naming Your Node 34
Uucp Login 35
Getty Entries 36

Editing the Library Files for Uucp ... 37
Additional Uucp Information ... 38

Chapter 4: Uucp File Structure
Examples of uucp Data Transfer ... 40
Transfer Single File Between Local and Remote System ... 40
Transfer Multiple Files Between Local and Remote System ... 41
Uux Command Sequences ... 43
Spool Directory ... 44
The Public Area ... 44
The uucp Directory ... 44
Workfiles ... 44
Data Files ... 47
Image Data Files ... 47
Data Execution Files ... 48
Execution Files ... 49
Typical Execution File ... 52
Lockfiles and Temporary Files ... 52
Log Files ... 53
Binary Files ... 54
Library Files ... 54
L.cmds File ... 55
Security Sequence-Checking Files SEQF and SQFILE ... 56
USERFILE ... 57
L-devices File ... 61
L-dialcodes File ... 62
Dialit.c ... 63
Dialit File ... 67
L.sys File ... 68
ADMIN File ... 72

Chapter 5: Uucp Facility Daemons
Running The Uucp Facility ... 73
Invoking uucp Daemons ... 75

Chapter 6: Using the Uucp Facility
Syntax Information ... 77
Pathnames ... 77
Option Separators ... 78
The Cu Command ... 79
Cu Command with a Modem Connection ... 79
Cu with Direct Connection ... 80

After Connection 81
Using the uucp Command 83
General uucp Snytax 83
Sending Files To a Remote System 84
Receiving Files From Remote Systems 85
Forwarding through Several Systems 85
Uucp Command Errors 88
Using the uux Command 89
General uux Syntax 89
Example 90
Uux Error Numbers 91
Miscellaneous Commands 92
Using uuclean 92
Using uulog 93
Using uuname 94
Using uupick 95
Using uustat 96
Using uusub 98
Using uuto 100
Using the Mail Facility 101
Notes 102

Chapter 7: The X.25 Network
Description of X.25 103
Packet Switching Network 103
Public Data Network 105
Configuring uucp for X.25 106
Prerequisites 107
Installing the HP 2334A 107
Remote and Local Off-line Configuration 110
Preparing for Configuration 112
Configuration Procedure 113
Notes 132

Chapter 8: Log, Status and Cleanup
Logging Information 133
The LOGFILE file 133
The SYSLOG file 135
The DIALLOG file 136
Status 137
Cleanup 138

Chapter 9: Problems

Bad Connections 141
Out of Space 141
Out-of-date Information 141
Abnormal Termination 141
Notes 142

Appendix A: Log Entry Messages

/usr/spool/uucp/DIALLOG 143
Meaning of Entries 143
Sample Entries 144
Message Interpretations 144
/usr/spool/uucp/LOGFILE 147
Meaning of Entries 147
Sample Entries 148
Message Interpretations 148
/usr/spool/uucp/SYSLOG 153

Index

Overview 1

The HP-UX *uucp* facility is a set of programs which exchange information between HP-UX and UNIX[1] or UNIX-like systems. Note that the program *cu* which is part of the *uucp* facility enables you to talk to non-UNIX systems. *Uucp* programs can be used to transfer files and execute commands to and from a remote system, to transfer files from a remote system to another remote system, and to send and receive mail. You can also forward mail and files through intermediate nodes. All systems involved must use an HP-UX or UNIX operating system, be on the *uucp* network and have the *uucp* facility installed. The network consists of workstations connected with either direct or modem connections. The *uucp* facility is easy to use, fast, reliable and cost-effective.

Program Overview

There are three main programs in the *uucp* facility: *cu*, *uucp* and *uux*, and there are also many auxiliary programs: *uuclean*, *uulog*, *uuname*, *uupick*, *uustat*, *uusub*, *uuto*, *uusnap*, and *uuls*. The *uucp* and *uux* programs operate in the background mode leaving your terminal free for other uses. Both your local system and the remote system must use an HP-UX or UNIX operating system, except when you are using *cu* which allows you to use a non-UNIX operating system. The rest of this section is a brief overview of the three main *uucp* programs.

Cu is an acronym for *call UNIX*. With the *cu* program you can interactively log onto any other UNIX and many non-UNIX systems. *Cu* provides a way for you to check your communications link and transfer ASCII files, but implements no error checking.

Uucp is an acronym for *UNIX-to-UNIX copy*. With the *uucp* program you can have the source_file and/or the destination_file reside on remote systems. To specify remote source or destination files you simply include the remote system name with the file name.

The information necessary to contact the remote system as well as the security access information is kept in a set of files on both systems. Once you have placed this information in the appropriate files, the *uucp* facility can automatically establish the remote connection and protect your data files from unauthorized use.

[1] UNIX is a trademark of AT&T Technologies.

All data transferred is checked for errors and re-transmitted should an error occur. This makes the *uucp* facility a reliable method of information exchange.

The *uux* command is the acronym for *UNIX-to-UNIX execution.* With *uux* you can only execute those commands which the remote system gives you permission to execute. The remote system has a list of these commands in its *L.cmds* file.

Although the *mail* command is a local HP-UX command, mail can also be used with the *uucp* facility. You can send mail to remote systems or forward mail through several remote systems to a final destination system.

Manual Overview

This manual contains nine chapters that address the following subject areas:

- Introduction and overview
- Hardware configuration
- Software configuration
- *Uucp* file system
- *Uucp* daemons
- Using *uucp* commands
- X.25 Networks
- *Uucp* log, status, mail, and clean-up information
- Possible uucp problems and solutions.

Chapter 1: Overview

This chapter gives an overview of the programs covered in this manual, as well as a brief description of each chapter included in this manual. You are also provided with directions on how to use this manual as a system administrator or system user.

Chapter 2: Hardware Configuration

This chapter describes the hardware installation steps which must be taken by the System Administrator to configure HP 9000 computers for the *uucp* facility.

Chapter 3: Software Configuration

This chapter outlines the *uucp* software configuration tasks the system administrator must perform after hardware is installed.

Chapter 4: Uucp File Structure

This chapter describes the files that are used by *uucp* facilities to implement remote communication.

Some of the files are created automatically in the */usr/spool/uucp* directory by *uucp* programs as they handle information transfers. By listing this directory you can verify the status of the transfer by identifying these files. Understanding the file structure also helps in case you have a problem in transferring data.

The system administrator must edit many of the files in this directory (see Software Configuration chapter) to specify:

- How your HP-UX Computer can contact each remote system
- How, when and if systems on the *uucp* network can contact you
- Access permission restrictions for files and commands.

Chapter 5: Uucp Daemons

Uucp daemons are the programs that do the work of the *uucp* facility. These daemons are automatically run when the *uucp* or *uux* programs are operating in the background mode. You can also invoke them interactively.

Chapter 6: Using The Uucp Facility

This chapter illustrates the use of *uucp* facility programs to:

- Send files and commands to a remote system
- Receive files and commands from a remote system
- Send and receive mail
- Monitor status, log and access information
- Clean up old or unwanted files.

Chapter 7: The X.25 Network

This chapter provides you with a brief discussion on what the X.25 Network is, and it explains how to set up *uucp* for X.25 communications.

Chapter 8: Log, Status and Cleanup

This chapter discusses how the system logs information about each transaction, how you can check on job or system status and how you can clean up old or unwanted files.

Chapter 9: Problems

The commonly encountered problems, their solutions and debugging information are presented in this chapter.

Where To Start?

If You Are a User

If your System Administrator has set up the hardware and software configurations on your system, go to the chapter, "Using The Uucp Facility".

If You Are the System Administrator

If you are System Administrator for the *uucp* facility, you should first contact the System Administrator at each remote system you want to communicate with and obtain the following information:

- The remote node name
- Whether this will be a direct (hardwired) or a modem (telephone) connection
- What calendar/clock times the remote system will accept communications from other systems (specifically yours)
- The remote system's incoming telephone number (modem connections)
- The data (baud) rate
- Your login name and password, if any, on the remote system.

This information must be incorporated into your *uucp* files to establish connection and maintain proper protections.

Next, set up your hardware and software configurations as described in the chapters, "Hardware Configuration" and "Software Configuration".

When these tasks are complete, your system can respond to *uucp* facility commands. The features described in the chapter, "Log, Status and Cleanup" should be started as soon as possible to monitor *uucp* activities and keep your file storage area free of old or unwanted data. These features should be used periodically to keep file space from getting cluttered and to maintain clean system operation.

Become familiar with the entire contents of this manual so you clearly understand the operation of *uucp* facility processes. You should also be familiar with the HP-UX file and command system. *HP-UX Concepts and Tutorials* manuals, the *HP-UX Reference*, and HP-UX *System Administrator Manual* for your system provide more information about HP-UX operation.

Additional Networks

This section is intended to introduce you to HP networks other than the ones found in this manual. It provides a brief explanation of each network and gives references to documentation for these networks. The additional networks are as follows:

- RJE
- LAN
- HP AdvanceNet.

The RJE Network

The RJE Emulator package enables your HP 9000 HP-UX computer to communicate with remote computers and peripherals that support IBM 2780/3780 Remote Job Entry (RJE) data transmission protocols. The emulator can also communicate with other IBM 2780/3780 compatible devices for file transfer.

The supported features of the RJE Emulator include:

- Binary synchronous communications with EBCDIC transmission codes
- Data transmission at up to 19 200 bits per second
- Space compression and expansion in 3780 mode only, thereby raising the effective throughput rate
- Transparent mode, which allows all possible EBCDIC combinations to be used as data
- Full- or half-duplex operation
- Programmable modem timeout value
- RJE/send subsystem adapted from System III UNIX MRJE
- Tracing of data
- Print formatting utility using IBM print conventions
- Auto answer or manual originate
- MSV2 protcol.

The following IBM 2780/3780 features are not supported:

- Interactive mode (Instead of ACK, a message is transmitted.)
- Multipoint transmission
- 6-bit transcode
- Bell messages
- Hardwired connections.

For more information on RJE networking, consult the *RJE User's Guide.*

The Local Area Network

A Local Area Network (LAN) is a way of connecting multiple systems together in a limited geographical area. For example, a Local Area Network can consist of systems attached to a single length of cable. A Local Area Network can also be formed by connecting a central system to each of the other systems in the building. Cabling present in a Private Branch Exchange (PBX) telephone system can also be used to interconnect devices within a site. PBX cabling forms a unique data transmission network unlike other LAN configurations; therefore, references to LANs in this manual do not include PBX LANs unless directly stated.

A LAN is not simply a connection of hardware; software controls the interaction and transmission of data between systems on the network. There are many different kinds of Local Area Networks; however, there are a few main characteristics they all share. These characteristics are:

- Limited geographic coverage
- Single-organization ownership
- High data rate.

For more information on the LAN network, read the *NS/9000, LAN User's Guide* and the *NS/9000, LAN Node Manager's Guide.*

HP AdvanceNet

HP AdvanceNet is an HP networking strategy which offers the following:

- Size Alternatives and growth paths
- Easy-to-use network
- Compatibility with:
  Industry standards,
  De facto standards.

Because HP AdvanceNet is based on industry standards, it is able to provide communication between HP computers and other vendors' computer products based on the same standards. Links currently defined up through Level 3 of the Open System Interconnection (OSI) model provide a common protocol for simplified communication in a multi-vendor environment.

HP AdvanceNet provides system-to-system communication capability within or between HP product lines, such as the HP 1000, HP 3000, and HP 9000 multi-user computer systems. These communication capabilities range from user-level services such as virtual terminal and file transfer to physical links such as point-to-point, PBX, X.25, and satellite.

For more information on HP AdvanceNet, read portions of the LAN documentation mentioned in the previous section which include HP AdvanceNet. Also read information on HP AdvanceNet found in these manuals:

- *NS/1000 User/Programmer Reference Manual*
- *NS/3000 User/Programmer Reference Manual.*

Hardware Configuration 2

This chapter discusses how to interconnect and configure hardware for use by *uucp* facility programs. *Uucp* always assumes the presence of standard RS-232C modem signals.

Series 500 ASI Card

The HP 27128A Asynchronous Serial Interface card contains the necessary firmware for modem signalling. If you are using a modem connection, follow the directions in the installation manual shipped with the card.

Before inserting the ASI card:

1. Be sure the power to your computer is off
2. Set switches two and eight of node address switches to their ON (down) position
3. Use the male cable (HP 27128A Opt. 001).

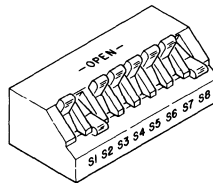


Figure 2-1. Switch Settings

Series 500 MUX Cards

The HP 27130A/B 8-channel Multiplexer card supports up to eight RS-232C-compatible devices. It consists of an interface card and an RS-232C connection panel with eight connectors. It is recommended for direct connection of terminals, providing slightly higher performance than the HP 27140A Modem MUX at a slightly lower per-port cost. The HP 27130A/B also supports terminal clusters up to 100m distant with customer-fabricated cables are used, provided the operating environment does not present electrical noise problems. The HP 27130A/B cannot be used for modem connections.

The HP 27140A 6-channel Multiplexer card supports up to six RS-232C/CCITT V.22-compatible devices. It consists of an interface card and an RS-232C connection panel. The HP 27140A is recommended when one or more ports are being connected to a modem or directly connected to another HP 27140A MUX card.

Before you insert the HP 27130A/B or HP 27140A card:

1. Be sure that the power to your computer is off
2. No card configuration is required. Cables and adapters for various equipment combinations are discussed later in this chapter. For other equipment combinations, contact your HP Sales and Service Office for assistance.

HP-UX Series 800 MUX Card

The HP 27140A option 800 (6 port MUX with four meter cable) is used on the Series 800 Model 840 computer. You must have **option 800** to have supported asynchronous serial RS-232-C connections. Installation instructions are in the *HP27140A Asynchronous 6-Channel Multiplexer Hardware Reference Manual* which is packaged with the MUX card. Read the instructions carefully or contact your HP Sales and Service Office for installation.

HP-UX Series 200 and 300 MUX Cards

The HP 98626A, HP 98642A, HP 98644A, or HP 98628A interface card can be used for the *uucp* facilities on your Series 200 or 300 computer. Before you insert the interface card:

1. Disconnect power to the computer
2. Set all U3 switches on the HP 98626A interface card to 1. If you are using the HP 98628A interface card, you need to set two switches on the row of 8 switches labeled DEFAULTS on the interface card. The two switch settings for the HP 98628A interface card are as follows: set switch 4 to 1 and set switch 5 to 0.
3. Set switches 1, 4, and 8 to 1 and the remaining switches to 0 on the HP 98644A interface card. Also cut or remove the remote jumper on the HP 98644A interface card.

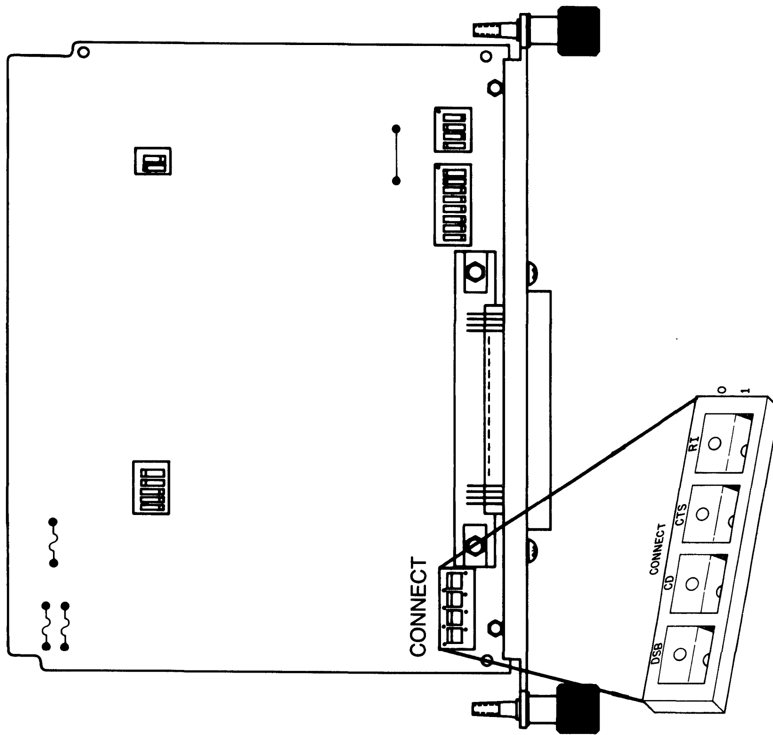


Figure 2-2. HP 98626 Switch Settings

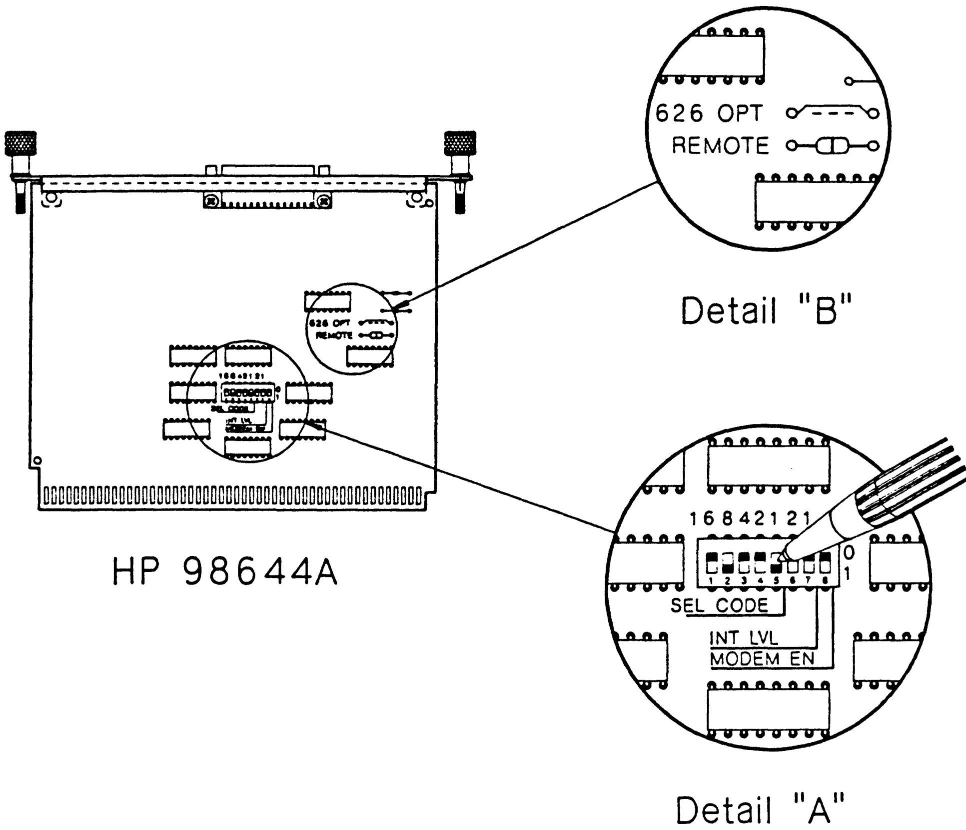


Figure 2-3. HP 98644 Switch Settings

Modem Connections

If the distance between your system and the remote system is more than about 15 meters (50 feet) or if noise on a direct connection line becomes a problem, a modem connection should be used.

Modem connections for the Series 500 computers require the HP 27128A Asynchronous Serial Interface (ASI) card whose firmware revision number is 27128-80005 or greater. Set switches 2 and 8 down for modem control. The HP 27140A (6-Channel Multiplexer) can also be used with the Series 500 for modem connection. There are no switch settings on this card which need to be made for modem use. For information on installing this card, read the installation manual for your HP 27140A interface.

Modem connections for Series 200/300 computers require either the HP 98626A or HP 98644A serial interface card or the HP 98628A datacomm interface card. Set all three switches to the "CONNECT" position. The HP 98642A (4-Channel Multiplexer) can also be used with Series 200/300 for modem connection (but only one of three ports). There are no switch settings on this card which need to be made for modem use. For information on installing this card, read the installation manual for your HP 98642A card.

For Series 500 computers, use a modem cable (HP 27128A Opt. 001) to connect the ASI card to the modem and use the HP 92219Q cable for connecting an HP 27140A card to a modem. For Series 200/300 computers, use a modem cable with male end (HP 98626/28A Opt. 001, "DTE cable") to connect either the serial or datacomm interface card. Note that you **cannot** use an interface card with a DCE (female) end since this does not carry through modem signals.

For Series 800 computers, use the HP 27140A option 800 (6-Channel Multiplexer with four meter cable) with the HP 92219Q cable to connect the MUX to a modem.

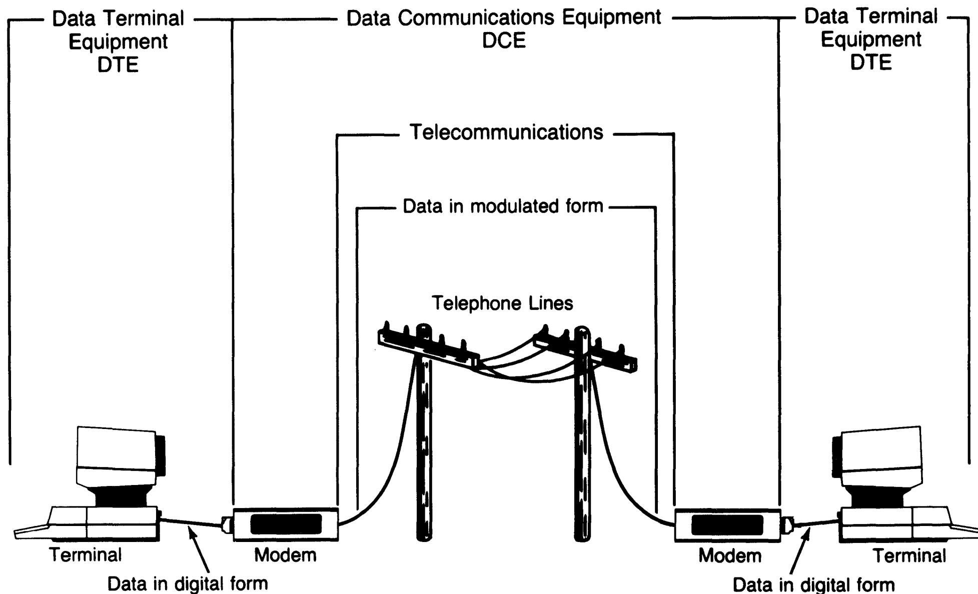


Figure 2-4. Typical Modem Connection

HP-UX modem connections can use the same line for both incoming and outgoing calls. No special modification is necessary; the DTE (male) cable end can be connected to the modem. When you use the *mknod* command to associate a special (device) file with the interface card on a specified select code, a flaging mechanism assigns the line as either an incoming or outgoing port. Refer to the section, "Creating a TTY Device File", in the "Software Configuration" chapter of this manual for more information.

Many companies sell modem devices which are compatible with the *uucp* facility. Contact your nearest HP Sales Office for further information.

Direct Connections

Direct connections can be made between two interface cards, two multiplexers, or a multiplexer and an interface card on two different systems. Direct connections can be implemented for any combination of Series 200/300 and Series 500 computers, and for all combinations of serial interfaces and multiplexer (MUX) cards. This section provides several examples of equipment combinations and the required cabling. Note that two DTE (male) cable ends cannot be connected together, so a special adapter connector or cable must be provided. Sufficient details are provided for building the needed adapters or cables for most situations.

Here is an example of directly connecting a Series 200/300 computer with either a HP 98626A or HP 98628A interface card to a Series 500 computer with an HP 27130A/B (8-channel multiplexer).

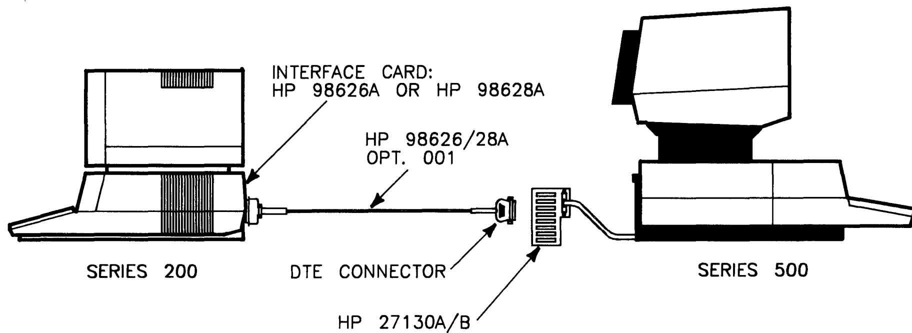
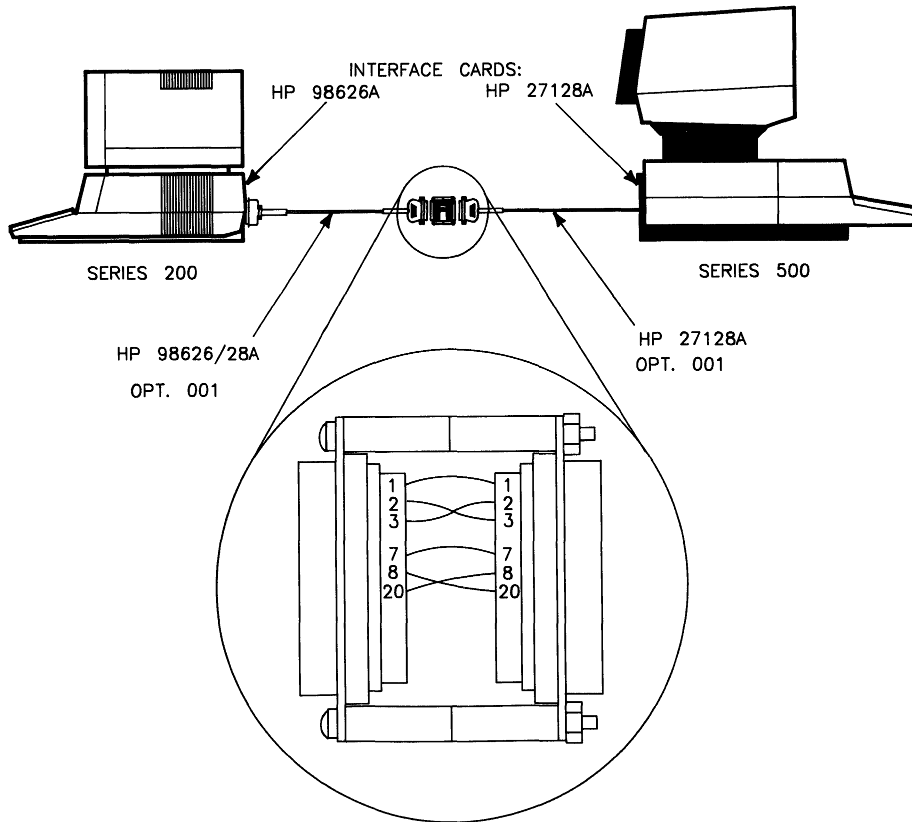


Figure 2-5. Series 200/300 Serial Interface to Series 500 8-Channel MUX

Here is an example of directly connecting a Series 200/300 computer with either an HP 98626A or HP 98628A interface card to a Series 500 computer with an ASI (HP 27128A) card. The first figure shows a direct connection for connection origination in one direction only (*getty* on one end only), while the second figure shows a direct connection for connection origination in either direction (*gettys* on both ends at the same time).



Fiture 2-6. Series 200/300 Serial Interface to Series 500 ASI (unidirectional)

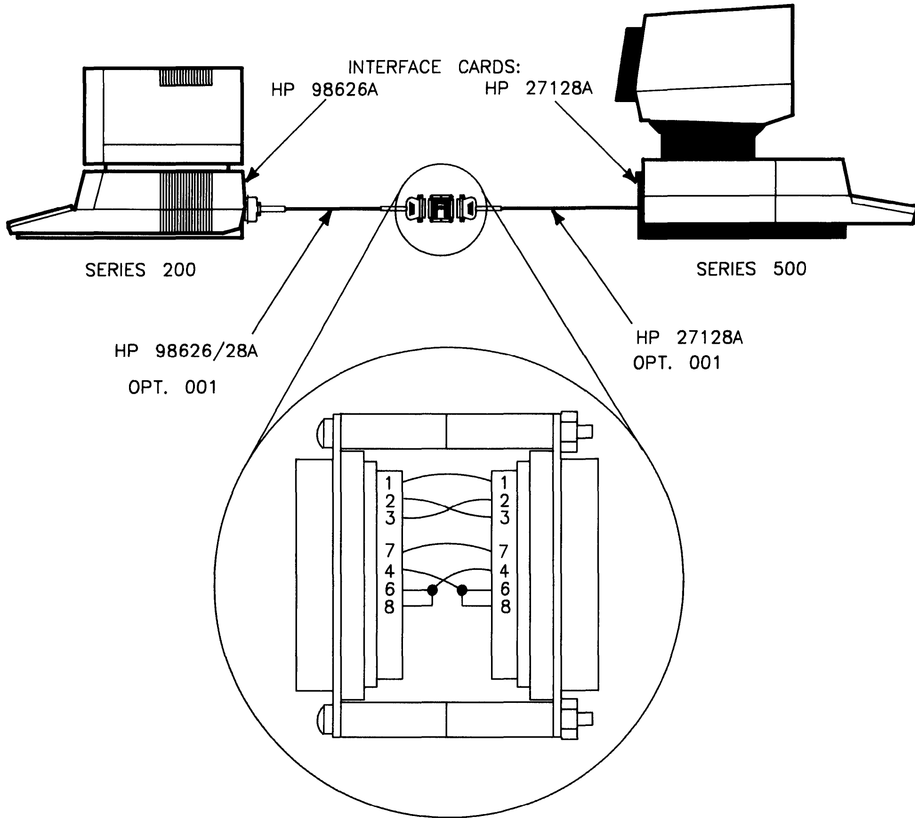


Figure 2-7. Series 200/300 to Series 500 ASI (bidirectional)

Here is an example of directly connecting a Series 200/300 computer with either an HP 98626A or HP 98628A interface card to a Series 200/300 computer with either an HP 98626A or HP 98628A interface card.

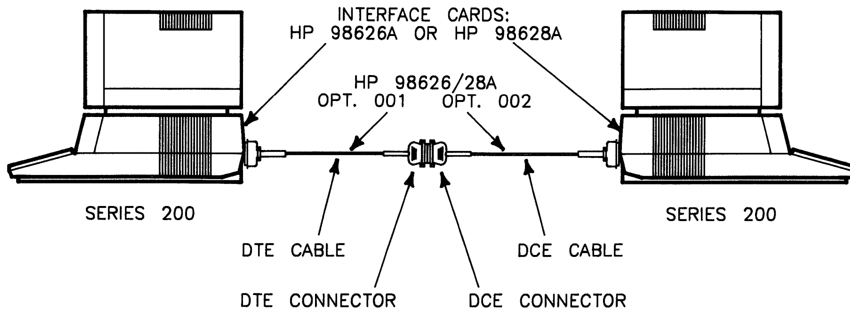


Figure 2-8. Series 200/300 to Series 200/300 Serial I/O

Here is an example of directly connecting a Series 500 computer with an ASI (HP 27128A) card to a Series 500 computer with an HP 27130A/B (8-channel multiplexer).

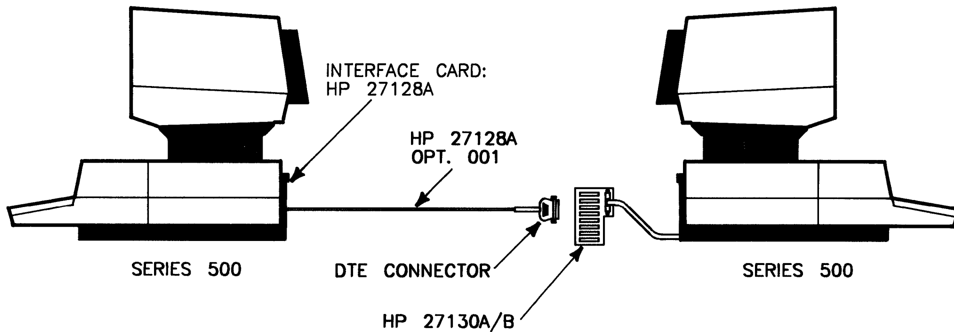


Figure 2-9. Series 500 ASI to Series 500 8-channel MUX

Here is an example of directly connecting a Series 500 computer with an HP 27130A/B (8-channel multiplexer) to another Series 500 computer with an HP 27130A/B (8-channel multiplexer).

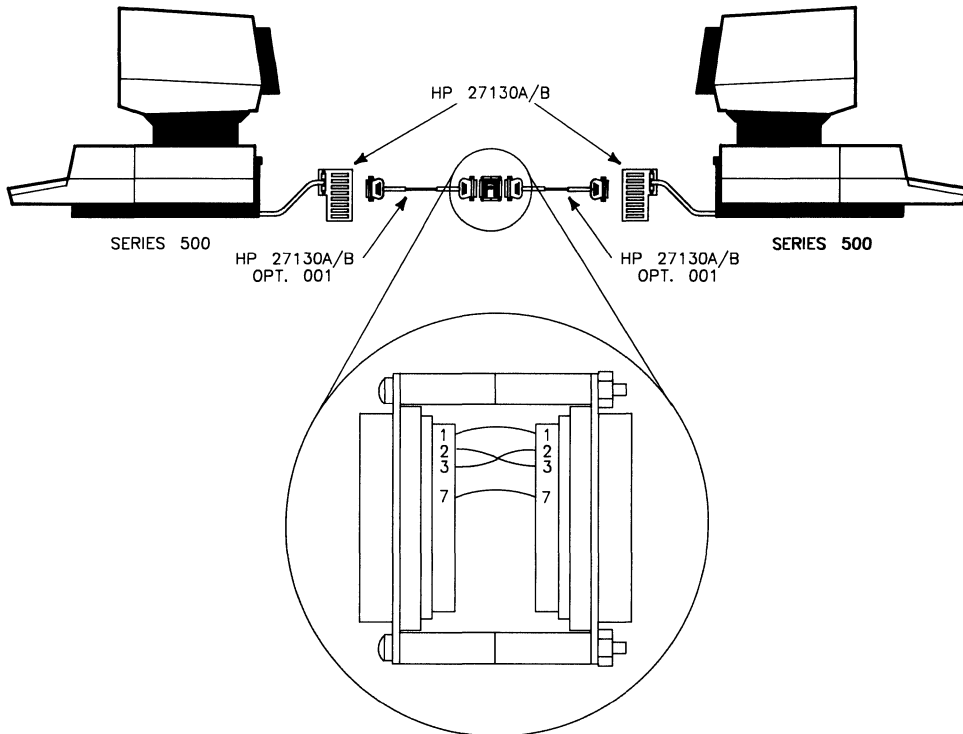


Figure 2-10. Series 500 8-channel MUX to Series 500 8-channel MUX

Here is an example of directly connecting a Series 500 computer with an ASI (HP 27128A) interface card to another Series 500 computer with an ASI (HP 27128A) card. The first figure shows a direct connection for communication in one direction, and the second figure shows a direct connection for communication in both directions.

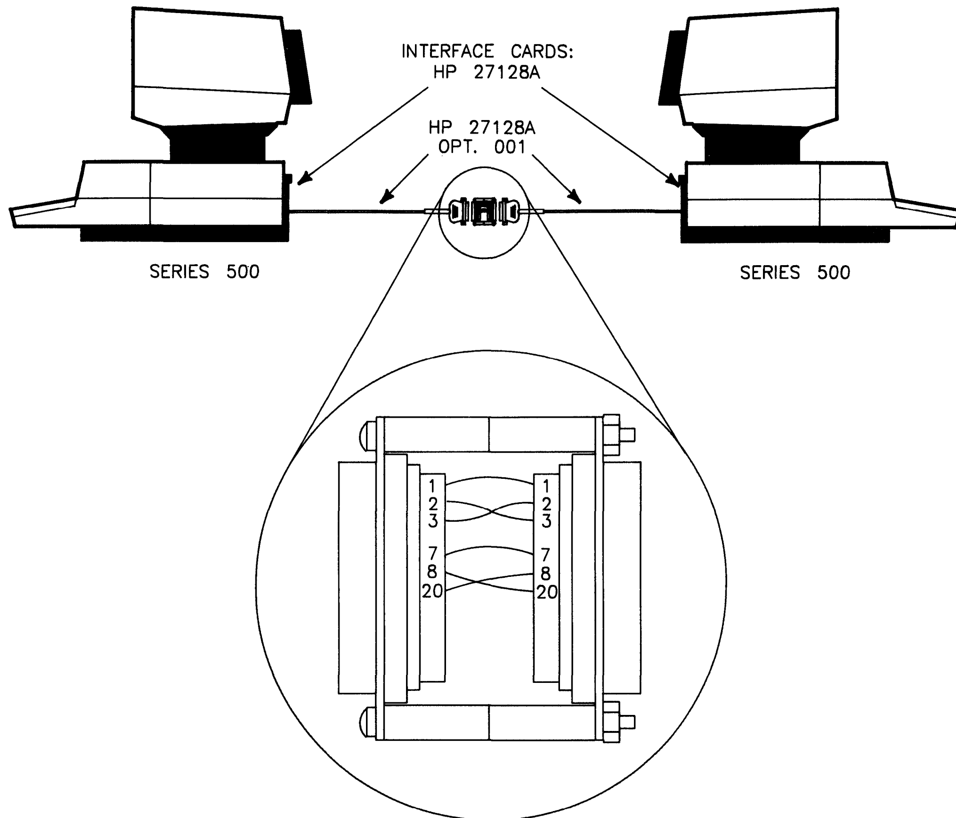


Figure 2-11. Series 500 ASI to Series 500 ASI (unidirectional)

Figure 2-12. Series 500 ASI to Series 500 ASI (bidirectional)

The direct connections shown here are between an HP serial interface card and an HP 27140A (6-channel modem multiplexer), between an HP 27130A/B (8-channel multiplexer) and HP 27140A, and between an HP 27140A and another HP 27140A. The following diagrams show direct connection for transmission in only one direction where one device is ACTIVE (sends information) and the other device is PASSIVE (waits for the information).

The diagrams given in the following three figures **do not** use the special connection as shown in previous examples. The direct conections shown here are made by the use of a cable and connectors as labeled in the diagrams. It is up to you to make the cable. The wires on the ends of this cable are connected to the pins on the connectors as shown.

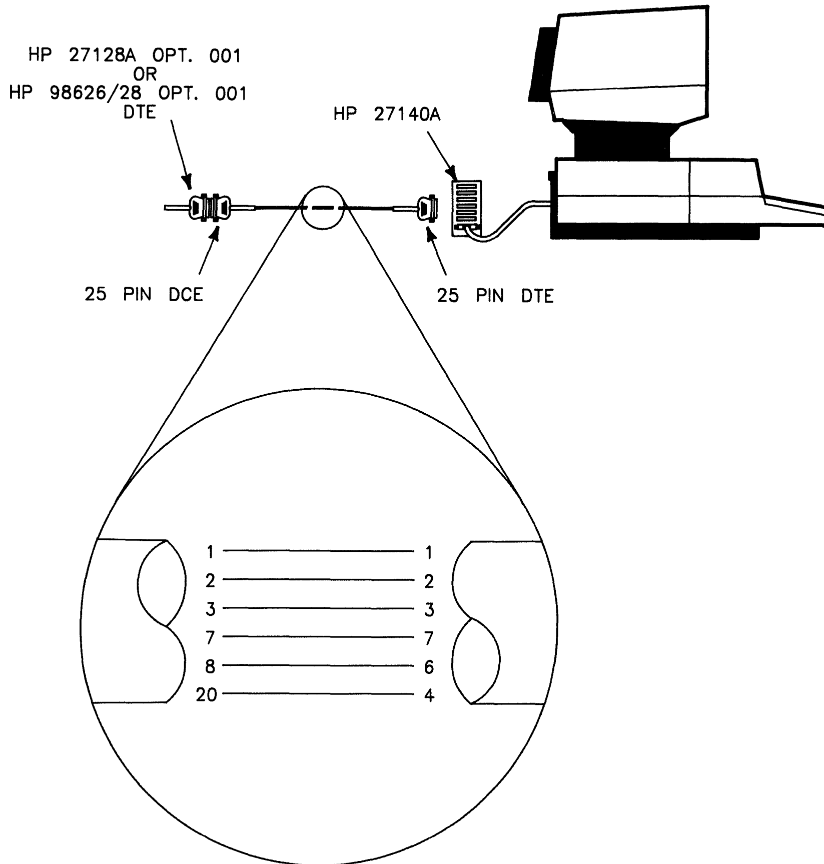


Figure 2-13. Serial Interface card to HP 27140A 6-channel MUX (unidirectional)

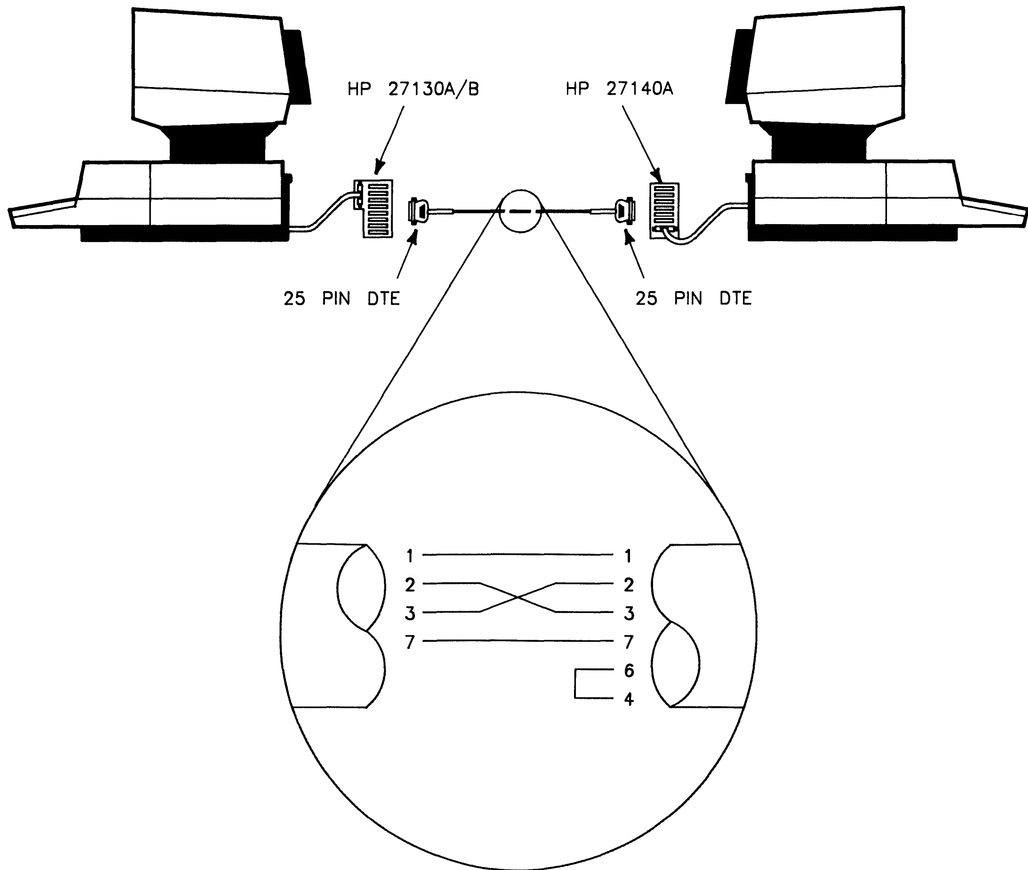


Figure 2-14. HP 27140A 8-channel MUX to HP 27130A/B 6-channel MUX (unidirectional)

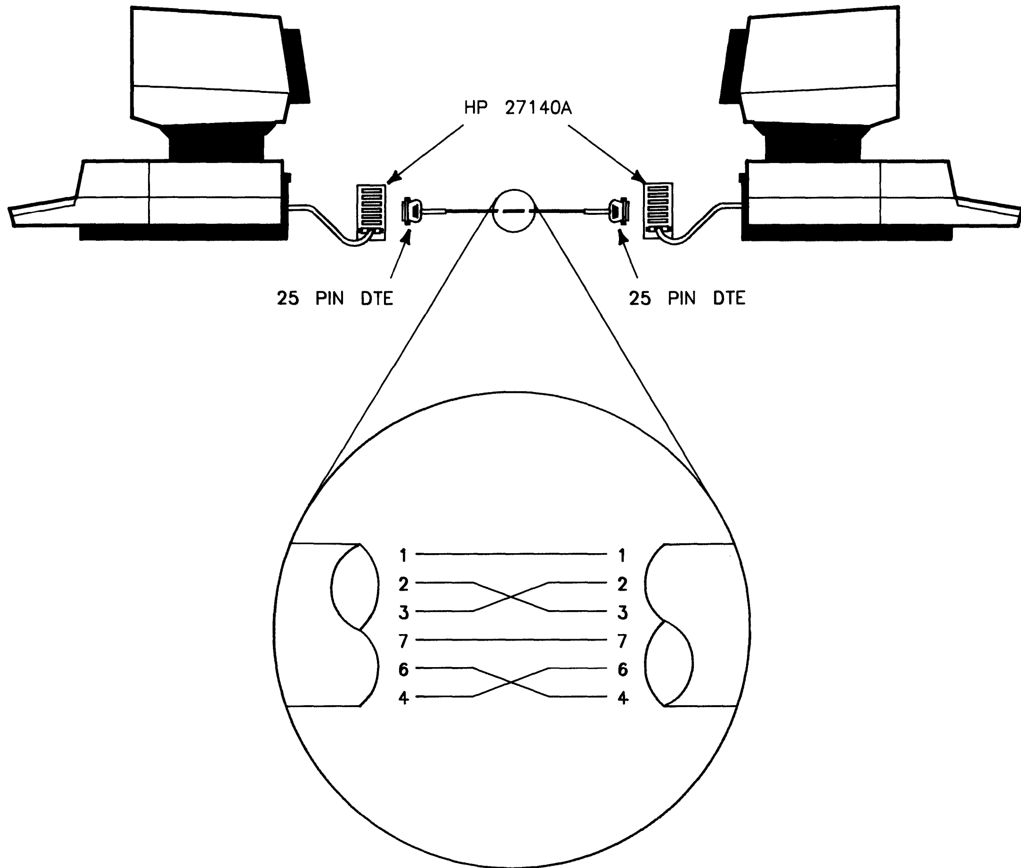


Figure 2-15. HP 27140A 6-channel MUX to HP 27140A 6-channel MUX (unidirectional)

The direct connections shown here are between an HP serial interface card and an HP 27140A (6-channel modem multiplexer), and between an HP 27140A and another HP 27140A. The following diagrams show direct connection for transmission in two directions where either device may be ACTIVE (sends information) or PASSIVE (waits for the information).

The two diagrams shown **do not** use the special connector as given in previous examples. The direct conections shown here are made by the use of cables and connectors as labeled in the diagrams. The wires on the ends of the cables are connected to the pins on the connectors as shown.

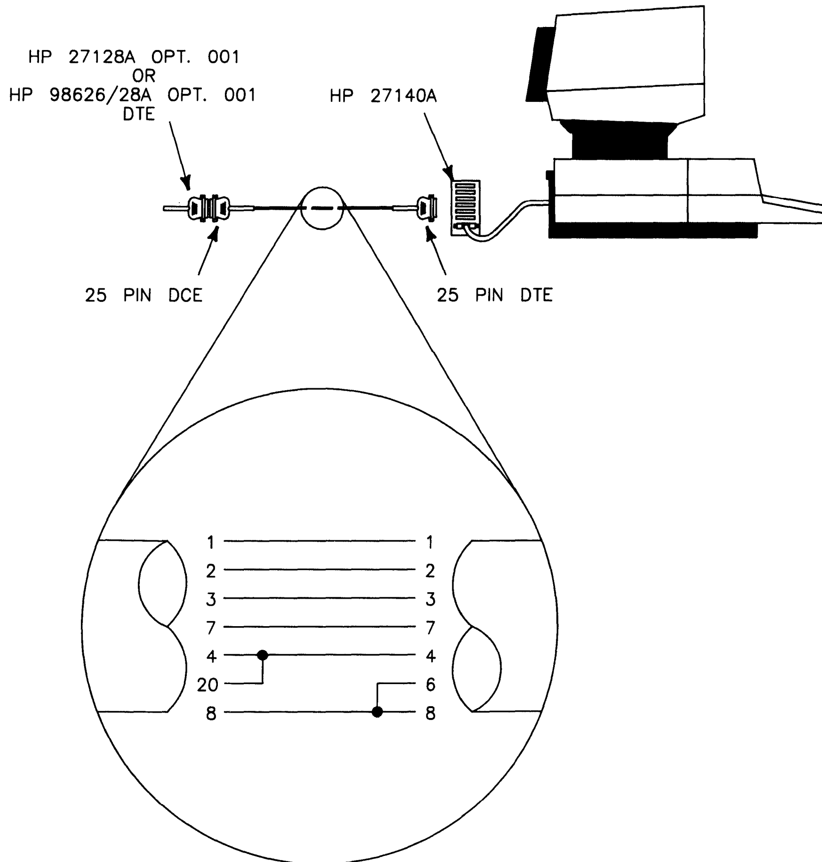


Figure 2-16. Serial Interface card to HP 27140A 6-channel MUX (bidirectional)

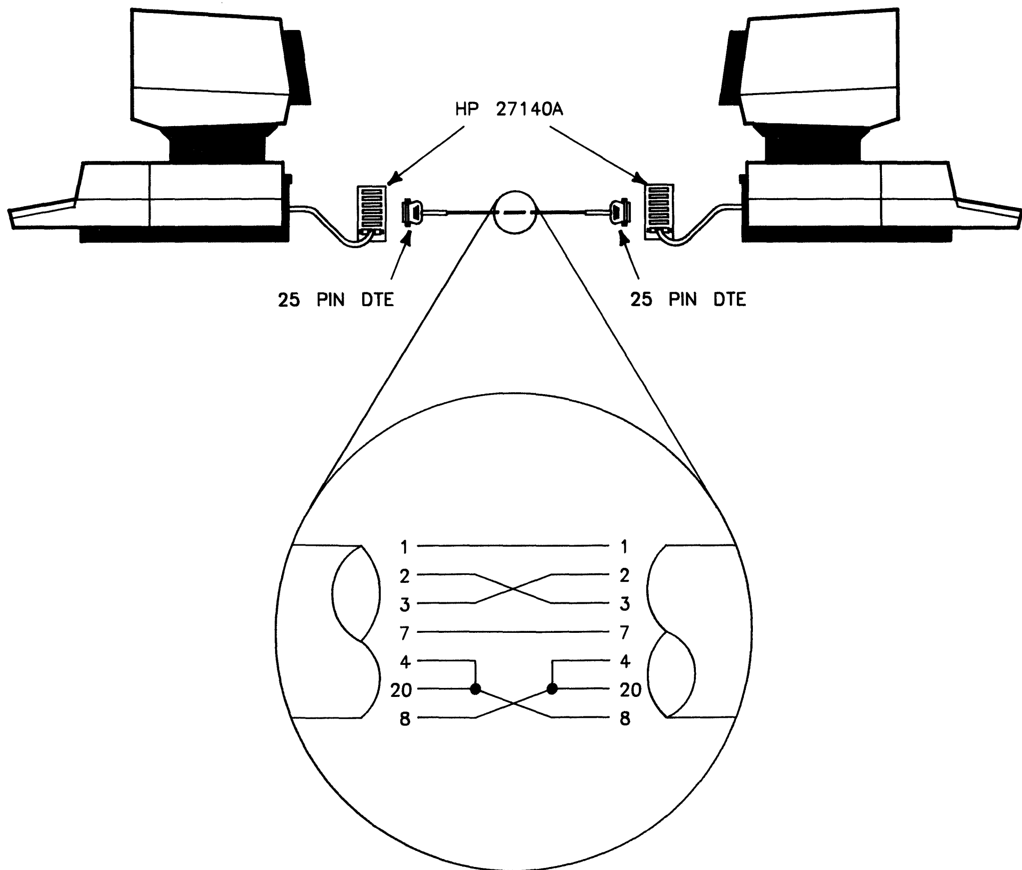


Figure 2-17. HP 27140A 6-channel MUX to HP 27140A 6-channel MUX (bidirectional)

You should be aware of these additional considerations when making direct connections:

- Series 200/300 computers can use an HP 98642A (4-channel multiplexer) in place of an HP 98626 or HP 98628 interface card; however, you must treat the modem port as an HP 98626A or HP 98628A connection, and you must treat HP 27130A/B (8-channel multiplexer) ports as multiplexers with 3-wire cables.
- Series 200/300 computers making direct connections using an HP 98644A card should treat this card as if it were an HP 98626A card.

The "Getty Entries" section of the chapter, "Software Configuration" describes how to make each line coming into your HP 9000 an ACTIVE (you can initiate calls) or a PASSIVE (you must wait to be contacted) line.

Making a Special Connector

The "special connector" is not a part-numbered item that can be purchased from HP. This connector has to be made using special parts. The following parts or usable equivalents are needed:

- Two DCE connectors (25-pin RS-232C female, HP part number 1251-0063 or equivalent) connectors
- Two 1½ inch long machine screws (HP part number: 2200-0125) with nuts and lock washers. These machine screws should be of the proper size to fit through the holes on both sides of the connectors and should not be so long they keep you from plugging in the conncector
- Four .625 inch metal (HP part number: 0380-0010) spacers
- 8 - one inch long pieces of 24 gauge electrical wire (note the wires should have ¼ of an inch of insulation stripped from each end).

The "special connector" should look like the following example when put together:

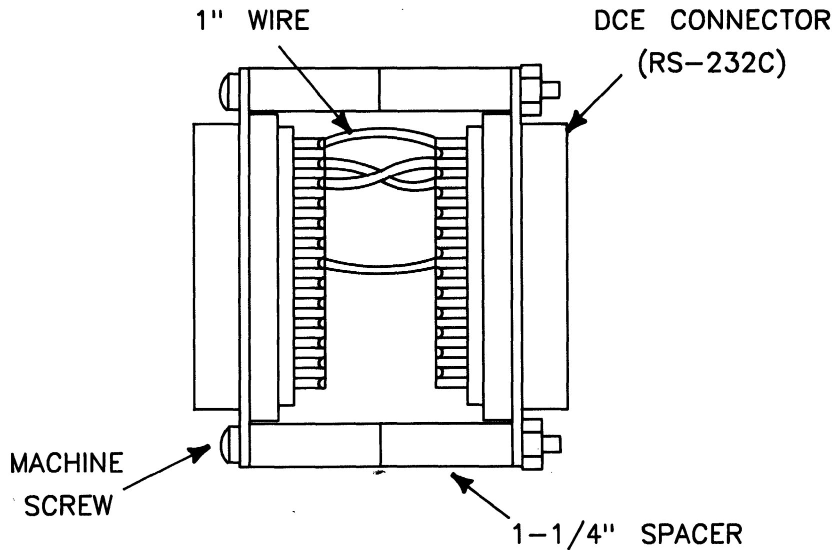


Figure 2-18. Special Connector

Wiring diagrams have been given on previous pages in this chapter to help you wire your "special connector". The *HP 9000 Series 500 Configuration Information and Order Guide* provides an alternate method for making direct connections with other systems. To find the appropriate information in this guide, read the section, "Uucp Connection" found in *Appendix I.*

Software Configuration 3

This chapter discusses the software configuration necessary for your HP 9000 computer to use *uucp* facility programs.

Software Loading and Setup

This section discusses the software steps the System Administrator must take to use an HP-UX computer as a node on the *uucp* network. Basically you must specify how and which other nodes on the network can contact you and how and which other nodes you can contact. **You must complete these steps to completely configure the *uucp* facilities.**

Seven main areas are covered:

- describing the boot and login processes
- creating a device file
- naming your node
- *uucp* login
- setting up a *getty* entry
- editing the necessary files.

General Startup Information

This section describes the boot and login processes. The tasks you need to perform for each file and command mentioned are discussed in later sections of this chapter.

Loading the Operating System

You must have the HP-UX operating system loaded in the boot area of your systems hard disc. Refer to the HP-UX *System Administrator Manual* for your particular computer for information about loading an operating system.

Loading Optional Drivers (Series 500 only)

This section uses the HP 27140A (6-channel modem multiplexer) to explain how to load an optional driver. This same procedure may be used to load other necessary drivers with the exception of Series 200 serial communication drivers which are loaded when the system is loaded. On the Series 800, the drivers are already installed upon boot-up.

The *HP27140.opt* driver is not installed when HP-UX is installed. If you need to use this driver, use the following procedure to install it:

1. Before installing the *HP27140.opt* driver, test to see it has not been previously installed by typing:

   osck -v /dev/sys_disc [RETURN]

   This lists the drivers which have already been installed with your system. If the *HP27140.opt* driver has been installed, skip the remainder of this procedure and continue reading in this section. If the driver has not been installed continue with the steps in this procedure.

2. Determine which HP 9000 Series 500 system model number you are using from this list (e.g. 97078C):

   - 97070C Model 520 single-user system
   - 97078C Model 520 multi-user system for 32 users
   - 97079C Model 530/540/550 single-user system
   - 97080C Model 520 multi-user system for 16 users
   - 97088C Model 530/540/550 multi-user system for 32 users
   - 97089C Model 530/540/550 multi-user system for 16 users

3. Type:

   `oscp -a /system/970xxA/HP27140.opt /dev/sys_disc` RETURN

   where `-a` says append to an existing operating system from a list of ordinary files, and put the resulting system in the boot area. The file `/system/970xxA/HP27140.opt` is the driver being copied to the boot area of the operating system. **The xx in this file is a two digit number taken from the last two digits of the system model number given in the above list of model numbers**. For example, in the system model number *97070A* the *xx* would be the digits *70*. The special file `/dev/sys_disc` identifies the system disc.

4. Verify that the file has been copied into the boot area by typing:

   `osck -v /dev/sys_disc` RETURN

5. Re-boot your system to make the *HP27140.opt* driver active.

You can use this same procedure to load any other drivers that you might need for your particular system application. Just change the system model number (for example, `97070C`) and the driver name (for example, *HP27130.opt*).

Final Sequence of Events Once the System is Loaded

Once the HP-UX operating system configuration is loaded, its initialization process begins and executes the file `/etc/init`. The init program reads the `/etc/inittab` file to find all incoming ports. You need to add an entry, called a `getty` (get terminal) entry, in this `inittab` file for each incoming *uucp* line. The operating system can then regularly check all incoming lines to see if another system is trying to communicate with you. The `getty` entries are therefore needed only if this line is used PASSIVEly or ACTIVEly by your local system. The `getty` entry specifies the special file name of your dial-in line as well as its communication speed. The getty command also causes the "`login:` " prompt to be sent to the calling computer when a connection is established.

The *init* program then starts the */etc/rc* shell script. This script causes many things to happen, among them setting your system's node name and executing the */etc/cron* program, which runs commands on a scheduled basis. You should use the */etc/cron* program to regularly compact and clean up some of the files used by the *uucp* facility, for example, files used to log transactions (see the chapter, "Uucp Facility Daemons").

Creating a TTY Device File

For creating a TTY device file on the Series 800, refer to the *Series 800 System Administrator's Manual.*

A device file must exist in the `/dev` directory to associate each device connected to your Series 200/500/800 computer with a special file name. This is done with the *mknod* command.

To create a `tty`, `cul`, or `cua` device file, make an entry of the form:

`mknod /dev/name c 31 0x`*ScAdnn*

where:

- The file `/dev/name` is either `/dev/ttydXX`, `/dev/culXX`, or `/dev/cua`. "`XX`" are characters used to differentiate between the device files for the various communication ports (for example, ttyd01 and ttyd02). The file naming convention `ttydXX` is for dial-up lines and the file naming convention `ttyXX` is for hardwire connections.
- The characters `Sc` identify the 2-digit hexadecimal select code of the communication port's interface.
- If the communcation port is connected via an HP 27130A/B (8-channel multiplexer) or HP 27140A (6-channel modem multiplexer), the characters `Ad` specify a 2-digit hexadecimal port number on the multiplexer. If it is connected via an HP 27128A Asynchronous Serial Interface, the characters `Ad` should both be zero.
- The characters `nn` represent one of the following for the HP 27128A card and the HP 27140A modem multiplexer. For the HP 27130A/B, all lines are `00` because it has no modem capabilities.
  - 00 for the incoming device file.
  - 01 for the outgoing device file.
  - 02 for the incoming device file if the modem obeys CCITT protocols.
  - 03 for the outgoing device file if the modem obeys CCITT protocols.

An example for the Series 800 would be to use `mknod` to create the special filename (Driver is 1):

```
mknod tty5p3 c 1 0x200503  #incoming
mknod cul5p3 c 1 0x100503  #outgoing
```

Note `ttyd`, `tty`, `cua`, and `cul` files use driver:

31 when using the HP 27128A serial interface card or the HP 27130A/B 8-channel multiplexer.

29 when using the HP 27140A 6-channel modem multiplexer.

1 when using serial interface cards on a Series 200 or 800 computer.

To illustrate, suppose you have inserted an HP 27128A Asynchronous Serial Interface into your computer at select code 0. You next connect a modem to it so this port may be used as a dial-in and dial-out port. The following *mknod* commands create the necessary device files:

```
mknod /dev/ttyd02 c 31 0x020000    #incoming device file
mknod /dev/cul02 c 31 0x020001     #outgoing device file
mknod /dev/cua02 c 31 0x020001     #autodial device file
```

Depending on whether or not you have the directory */etc* set up in your directory path, you would execute the *mknod* command with the */etc* prefix. For example:

```
/etc/mknod /dev/ttyd02 c 31 0x020000
```

Check to be sure you have both read and write access to the */dev* device file for lines used as out going ports. The *ll* command lists file characteristics, for example:

```
ll /dev/ttyd02
```

displays the following information:

```
crw-rw-rw- 1 root other 31 0x020000 May 13 14:37 /dev/ttyd02
```

showing read/write access for everyone.

Change the protection to read/write access if necessary with the *chmod* command, for example:

```
chmod 666 /dev/file_name
```

where:

`666` is the HP-UX code for read/write access.

The *uucp* `/dev` files `cul` and `cua` are normally readable and writeable by everyone.

Remove any old entries from the */dev* directory which have the same select code as the cards you now intend to use for ingoing or outgoing *uucp* calls.

For more information on the *mknod* command, read the chapter, "The System Administrator's Toolbox" in your *System Administrator Manual.* For more information on the *chmod* command consult *chmod*(1) in the *HP-UX Reference.*

Naming Your Node

Every node on the *uucp* network must have a unique nodename. Local Area Network (LAN) nodenames are independent of the *uucp* nodenames.

Since nodenames are frequently typed, a carefully planned convention can help all users identify and remember system nodenames. Try to avoid extraneous characters such as hyphens, numbers or upper-case letters.

To determine if your node has a name and if so what the nodename is, type:

```
uname -n
```

or

`uuname -l` (if uucp has been installed)

or

```
hostname
```

There are two ways to name your node:

1. use the *hostname* command in the system script */etc/rc* when you want this node name to come up with every change of state or power-up;
2. execute the *hostname* command as the superuser (`root`) when you want the name to last until you change your state or power-down your system.

Using the */etc/rc* system script is the recommended procedure.

Uucp Login

When you installed your HP-UX system, this entry was made in */etc/passwd* to provide a login for *uucp*:

```
uucp::5:1::/usr/lib/uucp:/bin/sh
```

You need to change this to allow a remote system to contact you and initiate the process *uucico* which takes care of any work requested.

This login should have a password for security reasons. An example of the initial entry prior to setting the password is:

```
uucpln::5:5::/usr/spool/uucppublic:/usr/lib/uucp/uucico
```

where:

`uucpln`	is the *uucp* login name and is restricted to a maximum of eight characters
`::5:5::`	are conventions used by HP-UX to designate the user and group access restrictions. They have nothing to do with the mode bits
`/usr/spool/uucppublic`	is a public area normally accessible to everyone and used here as the login directory
`/usr/lib/uucp/uucico`	is a program which must be started when you login as a uucp user.

Getty Entries

You need a *getty* entry in your */etc/inittab* file for each line which is used as a *uucp* login port for your Series 200/500/800 computer.

The following is an example of the format used for the *getty* entries in */etc/inittab*:

```
/etc/getty [-h] [-t timeout] line [speed]
```

where:

`-h`	forces a hangup on the line by setting the speed to zero before setting the speed to the default or specified speed
`-t timeout`	specifies that `getty` should exit if the open on the line succeeds and no one types anything in the specified number of seconds
`line`	is the name of a `tty` line in `/dev` to which `getty` is to attach itself
`speed`	is a label to a `speed` and `tty` definition in the file `/etc/gettydefs`.

An example for a direct connection at 9600 baud is:

```
/etc/getty tty02 9600
```

An example for a modem connection at 1200 baud is:

```
/etc/getty ttyd02 1200
```

Editing the Library Files for Uucp

Before you read this section, you need to have a basic understanding of where various files and directories are located in the HP-UX file system. The specific directories and files you should be aware of are listed in the following diagram:

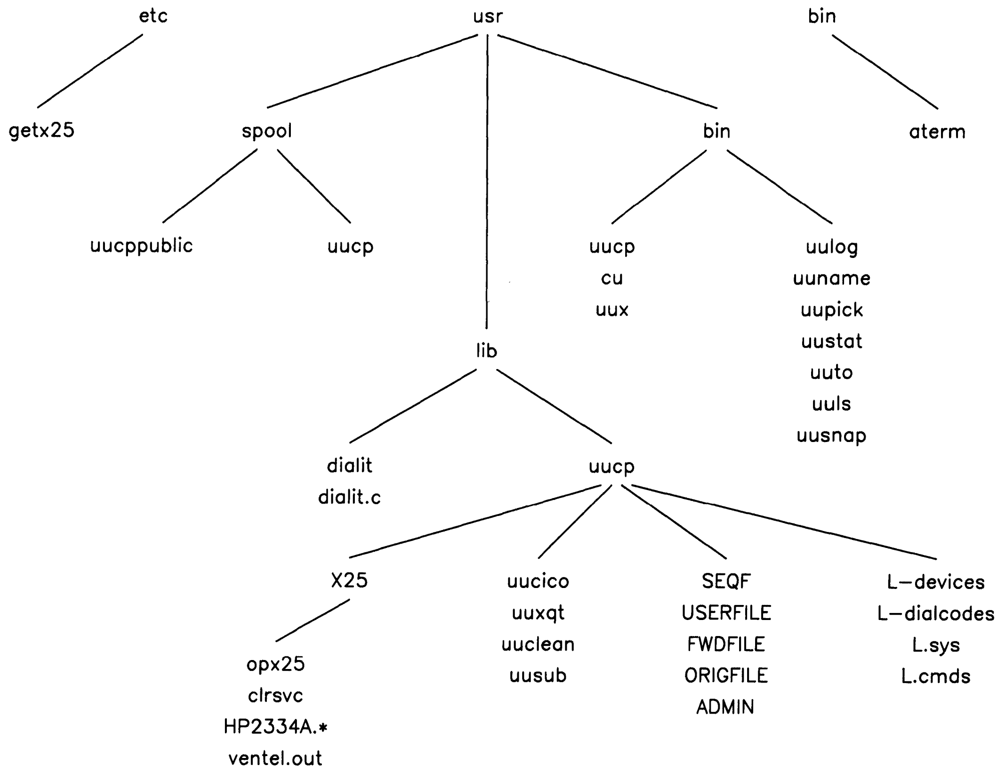


Figure 3-1. Files/Directories Installed

The directory */usr/lib/uucp* contains program modules which the *uucp* commands need to use. These program modules initiate and carry on all communications with remote systems and perform the remote execution of commands.

You must edit the following files:

- `FWDFILE` — to provide a list of systems your system can forward to or through
- `ORIGFILE` — to provide a list of systems which may forward through your system
- `L.sys` — to specify the remote system parameters
- `L-devices` — to provide a list of valid devices
- `dialit.c` — to modify the C source code for a dialing routine. *Dialit.c* then must be compiled and directed to the *dialit* file.
- `L-dialcodes` — if you use special characters in the modem phone number
- `USERFILE` — if you want security protection for a file(s)
- `L.cmds` — if you want to list the commands a remote system can execute with *uux* on your local system.

The System Administrator must edit these library files before any *uucp* communications take place. The "Library Files" section of the chapter, "Uucp File Structure" in this manual discusses how you place information specific for your needs into the files. Reading this entire chapter can provide you with a better understanding of the *uucp* facility processes and can help you should problems occur.

Additional Uucp Information

The chapter, "Log, Status and Cleanup" in this manual discusses the methods by which you can monitor status and log information, as well as the ways you can rid your file structure of old or unwanted files. It is recommended that you begin implementing these features as soon as possible.

Uucp File Structure

4

Three HP-UX directories contain files used to implement *uucp* facilities:

- */usr/spool/uucp* (spool directory for background processing)
- */usr/bin* (directory containing executable command files)
- */usr/lib/uucp* (library of programs used only by uucp, and configuration files).

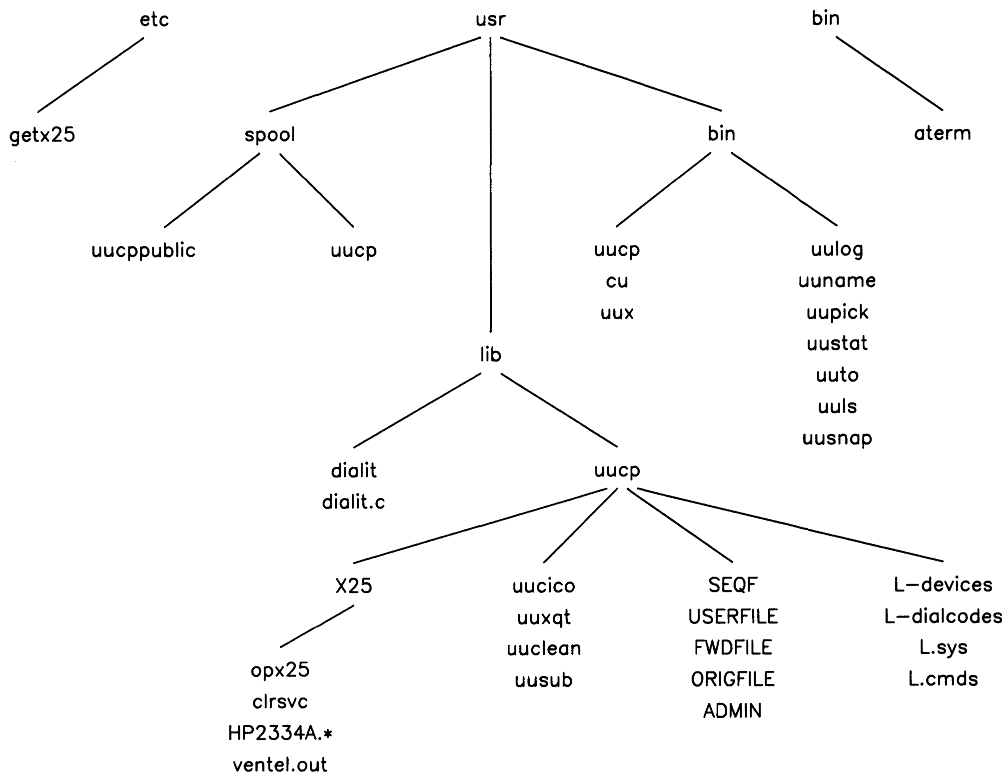


Figure 4-1. Files in Directories

Examples of uucp Data Transfer

The next three examples illustrate the dynamics of the *uucp* data transfer. Refer to these examples when reading about the *uucp* file structure.

Transfer Single File Between Local and Remote System

`uucp` *local_source_file remote_dest_file*

or

`uucp` *remote_source_file local_dest_file*

This example shows the transfer of one file from a local system to a remote system.

First, make sure the source_file is readable by everybody[1].

1. The local system checks its **USERFILE** to verify the user has access to the source_file and that this file is readable by everybody.[1] If the user is not found, the default is "everybody" (if the last field of USERFILE is set, as recommended). Usually, no one specifies many users in USERFILE because it limits who can do the transferring.
2. A workfile (`C.`) is set up in the local spool directory.
3. A prompt is sent to your terminal display indicating that you can now perform other operations while *uucp* continues with the transfer (*uucp* is in the background).
4. *Uucp* checks the local `L.sys` file for information about connecting the device to the remote system.
5. *Uucp* checks the `L-devices` file to determine whether the device from the *L.sys* entry is a valid device (with speed that matches).
6. Uucp locks the remote system and device line using `LCK.` files (in `/usr/spool/uucp`).
7. If this is a modem device, the `dialit` program executes a dialing routine.
8. *Uucp* attempts to login on remote system according to the information in the `L.sys` file (device filename) and the `L-devices` file (type of connection).

[1] When transferring files in the opposite direction (remote system to local system) the dest_file is on the local system. The local system checks its **USERFILE** to verify that the local user has access permission to the dest_file/path if it already exists and that the dest_file is writeable. A dest_file is created if none exists.

9. The remote system checks its **USERFILE** to determine whether your local system should be called back to verify your identity (this example assumes callback is not required).
10. The local system now sends a request for work (one line of `C.`*workfile*) to the remote system.
11. The remote system checks its **USERFILE** to verify that your local system has permission to access to the remote dest_file (if the destination file already exists, it must be writeable).[2]
12. The source_file is sent to a temporary (`TM`) file on the remote sysem (if transmission proceeds without error, the `TM` file is copied into the *remote_dest_file.*).
13. The local and remote systems disconnect if no further work needs to be done.

Transfer Multiple Files Between Local and Remote System

`uucp -C` *local_source_file remote_dest_file*

This example illustrates multiple transfers from both the local and remote systems. A **user** on the local system initiates the *uucp* command. This example differs from the first one in that the callback option as well as multiple work orders on both systems are discussed.

First, make sure the source_file is readable by everybody[3].

1. The local system checks its **USERFILE** to verify that the user has access to the source_file and that this file is readable by everybody.[3] If the user is not found, the default is "everybody" (if the last field of USERFILE is set, as recommended). Usually, no one specifies many users in USERFILE because it limits who can do the transferring.
2. A workfile (`C.`) and a source_file (`D.`) are placed in the the local spool directory.
3. A user prompt is sent to the terminal, and *uucp* continues operation as a background process. The local system is the master.

---

[2] When transferring files in the opposite direction (remote system to local system) the source_file is on the remote system. The remote system now checks its **USERFILE** to verify that the local system has access permission to the source_file/path and that the source_file is readable by everybody.

[3] When the dest_file is on the local system, the local system would check its **USERFILE** to verify that the local user has access permission to the dest_file/path if it already exists and that the dest_file is writeable. A dest_file is created if none exists.

4. *Uucp* checks the local `L.sys` file for information on how to connect the device to the remote system.
5. *Uucp* checks the `L.devices` file to determine whether the device from the `L.sys` entry is a valid device (with speed that matches).
6. *Uucp* locks the remote system and device line using `LCK.` files (in `/usr/spool/uucp`).
7. If this is a modem device, the `dialit` program now executes a dialing routine.
8. *Uucp* now logs into the remote system according to the information in the `L.sys` file (device filename) and the L-devices file (type of connection). The remote system is the slave, so the master waits for the return message "`Shere`" indicating that the slave is ready to continue. If you want to see the "`Shere`" message appear on the terminal, you turn the debugging option on in `uucico`, option `-x9`.
9. The slave (remote) checks its `USERFILE` to see if the master (local) should be called back to verify its identity. If callback is required, the slave signals the master to hangup. The master disconnects. The slave changes to the master role and initiates a return call. When the return connection is completed, the two computers have reversed roles. The computer that initiated the callback is now the master for the remainder of this example.
10. The new master sends a request for work (`S`, `R`, or `X` line of the `C.`*workfile*) to the slave.
11. The slave checks its `USERFILE` to verify that master's system_name can access to the dest_file. If the destination file already exists, it must be writeable.[4] If access is not permitted, the slave sends a "`NACK`" and the master puts a "Remote access to file/path denied" message in the `LOGFILE`.
12. The slave sends the acknowledge (`ACK`) message, "`SY`", and *local_source_file* is sent to the slave's temporary (`TM`) file. The master transfers the bits in 64 byte packets. The slave retrieves each packet, verifies the checksum, and puts the data in the `TM` file in `/usr/spool/uucp`. The slave must `ACK` each packet; if the master does not receive the `ACK` from the slave in a specified time interval, the master retransmits the packet. This is done up to five times. If no `ACK` is received the transmission is aborted ("g" protocol). If transmission proceeds without error, the TM file is copied into the slave's dest_file and the master deletes the `D.*` file if one exists.

[4] When the source_file is on the remote system, the remote system now checks its `USERFILE` to verify that the local system has access permission to the source_file/path and that the source_file is readable by everybody.

13. The master checks for additional work. If there is additional work, go back to step 10.
14. The master sends a hangup message, "`H`" to the slave who then checks its spool directory for `C.*` files for master.

    If there is work on slave for master, the slave sends a hangup no, "`HN`", message to the master, the master and slave change roles, and go to step 10.
15. The slave sends a hangup message, "`HY`", to the master and they both disconnect.

Uux Command Sequences

`uux` *command_string*

This example illustrates the *uux* command sequence of actions:

1. A workfile (`C.*`) and two data (`D.*`) files are set up in the local spool directory. One data file has an X grade and becomes an execution file on the remote system. The other data file(s) contains any file(s) the command requires.

Steps 3 through 13 are the same as in the second example.

When a request for work is sent to remote system, the file **`D.aaXbb`** becomes an execution file (`X.*`) on the remote spool directory. Any other data files are also transferred. At this point both systems can disconnect and the execution daemon *uuxqt* starts.

14. The remote system checks its **`L.cmds`** file to verify your local system may execute the specified *command_string* on the remote system.
15. The remote system executes the command.
16. The remote system notifies your local system by mail of the execution status.

Spool Directory

The Public Area

The public area, `/usr/spool/uucppublic`, is the area with general access privileges. This area can be used to receive files from a remote system that does not have access to any specified path on your system. Each time you use the "`sys_name!~/file_name`" as your pathname, the system stores "`file_name`" in "`/usr/spool/uucppublic/file_name`" on the remote system "`sys_name`".

The uucp Directory

The `/usr/spool/uucp` directory contains workfiles, data files, log files and system status files. At the time of initial installation of your system, this directory is empty. These files are automatically created when the *uucp* facilities are used.

Workfiles

Workfiles have a `C.` prefix and are work orders to copy data files. When you use *uucp* or *uux* commands or the remote *mail* command, these workfiles are automatically created. A child process of the parent *uucp* scans the spooler directory and in chronological order, taking the oldest work order first, processes whatever is asked for in the background mode.

Workfile names contain the information indicating which systems must be contacted to perform the work requested. The following figure shows the general form of a `C.workfile`.

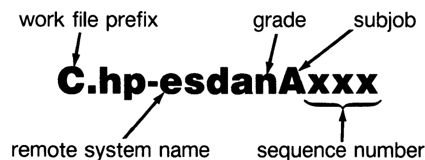


Figure 4-2. Workfile Name

where:

C.	is always the workfile prefix
remote system name	is the name of the remote system to contact (name cannot exceed seven characters)
grade	is a work-sequencing mechanism (the higher the grade, the sooner the work is done because workfiles are processed in alphabetical sequence. The highest grade is A and the lowest is z.)
sequence number	is the job number associated with the workfile and is assigned by the system (the sequence number is used with the uustat command)
sub-job	is the character used to differentiate among files having the same sequence number. These are sub-jobs of the request.

You can alter the grade by using the **-g** option (following **-g** with the desired grade) with the *uucp* command. Workfiles created by the *uux* command have a grade of "A" because command execution has a higher priority than file transfers.

Workfile names tell the *uucp* facility which system to contact, while workfile contents tell the facility exactly what work must be done. Each line in the workfile is separated into eight fields, as explained below.

A workfile contains one or more lines having the following form (only seven fields are shown):

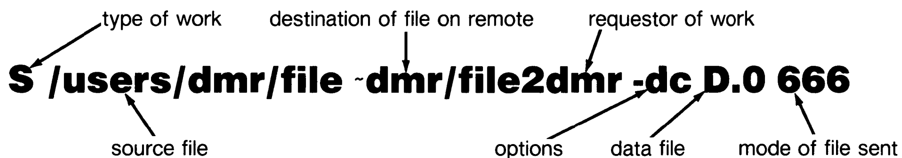


Figure 4-3. Workfile Contents

where:

type of work can be the following:

S – send a file from your system to the desired remote system;
R – copy a file from the remote system onto your local system;
X – send a request to the remote system for processing (the remote generates the C file with S lines specifying file(s) to be processed).

source file is the source file of the copy in either direction. The pathname on a line with an X type of work does not have an explicit file name because the remote system is sending the file. When you have an R type of work the pathname includes a filename on the remote system, but does not necessarily include the complete path name.

destination is the destination of the file copied. R types of work have local destinations, while S or X types of work specify a copy to a local or remote location. The destination is expanded on either the local or remote to the login directory if you use "~" (tilda).

user is the login name of the user who requested the work.

options is a list of command options and begins with a minus sign (-). If you use the **-dc** options (these are the the default options) you specify with the "d" that the system make any directories needed on the destination and with "c" that the system use the source file(s) to copy from. A "C" option indicates that a copy of the source file exists as D. file in the */usr/spool/uucp* directory.[5]

data file is the data file to be copied. If you chose a "c" option, this data file should be D.0; if you chose the "C" option, the file is `D.*`. Note that a D.0 data file uses the source file that is current at the time of transfer (background processing may involve a delay that provides a potential opportunity for the file to be altered by another process before the transfer occurs) while a `D.*` data file uses the source file in its current state as of the time of the request.[5]

mode is the mode of the source file sent in an "S" type of work. This can have read, write and/or execute mode capabilities.

[5] If you want to modify a file while an original copy is transmitted across the network, use the -C option. This forces a copy of the file to be queued on the spool directory to await its turn for transmission. The -c option takes a copy of the file only when it is time to transmit that file. The default option is -c.

The eighth field only exists if an "n" exists in the option list. This option causes a user on a remote system to receive mail when a file being sent to him has arrived. This field holds the login name of the user who is notified.

Each workfile can contain up to 20 entries (lines beginning with "X", "S", "R" or some combination).

Data Files

Data files begin with the prefix "D.". There are two types of data files: image and execution.

Image Data Files

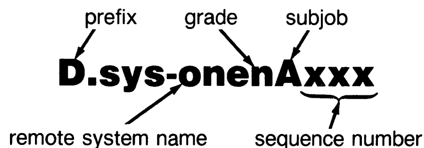


Figure 4-4. Image Data File Name

where:

prefix	is always "D".
system name	is normally the name of the remote system where the file is being sent if the D.* file is a "copy" of the file on your system. If the data file is to become an execution file on the remote system, the system name field is your local system, and the grade field contains an "X".
grade	is again the work sequencing number. See grade under workfiles. The grade of these files is usually "n" if generated by a *uucp* command or "B" if generated by a remote *mail* command
sequence number	is a four digit job number associated with the data file
sub-job	is the character used to differentiate among files having the same sequence number. These are sub-jobs of the request.

The image data file (`D.0`) generated with the -c (default) option for the *uucp* command does not really exist because the data is gathered at the time of transfer. The image data file generated with the -C option contains a copy of the data file at the time the *uucp* command was invoked.

Data Execution Files

Data execution files contain the necessary information for executing a command on the remote system. This command is requested locally by a *uux* command or remote mail. The following figure illustrates data execution filename fields.

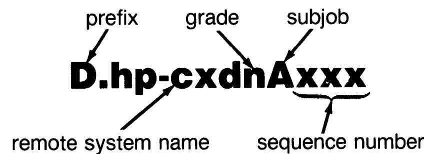


Figure 4-5. Data Execution Filename

where:

prefix	is always "D"
system name	is the name of the local system where the file was generated
grade	is always "X" denoting a data file which, when transferred to the remote system, becomes an execution file
sequence number	is again the job number associated with the data file
sub-job	is the character used to differentiate among files having the same sequence number. These are sub-jobs of the request.

Data execution files become execution files (`X.*`) when they are transferred to the remote system. As with all data files, their contents remain the unchanged; only their name is altered. The next section describes the contents of these files.

Execution Files

Execution files found on the spool directory */usr/spool/uucp* are the product of data execution files that were copied from another system. These files are created by *uucp* when the request for work is transferred.

The following figure illustrates their fields.

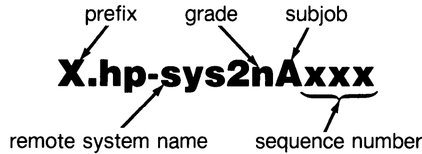


Figure 4-6. Execution File Name

where:

prefix	is always "X"
system name	is always the name of the remote system that initiated the transfer
grade	is "X" for execute
sequence number	is the job number associated with this file
sub-job	is the character used to differentiate among files having the same sequence number. These are sub-jobs of the request.

An typical execution file would contain these five lines:

- a user line (has a "U" prefix)
- a required file line (has a "F" prefix)
- a required standard input information line (has an "I" prefix)
- a required standard output information line (has an "O" prefix)
- a command line (has a "C" prefix).

F, I and O lines are required **only** if their respective files are required for the command execution.

The discussion that follows shows examples of the general form of the line, a specific example, then a discussion of the line fields. The line prefix is not included in the discussion but is shown in the line examples.

User Line

Syntax: `U` *user source_system*

Example: `U kls hp-dcx`

where:

user	is the user login name for the user on the remote system who issued the command
source_system	is the name of the remote system where this execution file originated.

There should be only one "U" line per execution file.

NOTE

You can route an execution file through intermediate nodes, but the information concerning its origin is lost. The last intermediate system routed through and the login used for uucp are shown on the user line in this file.

Required File Line

Syntax: `F` *required_file <source>*

Example: `F D.hp-dccB278`

where *required_file* has two fields:

- data filename as it should appear in your local */usr/spool/uucp* directory
- source from which the data filename above was copied.

An execution file may contain zero or more "F" lines.

Standard Input Information Line

Syntax: `I` *D.file_for_standard_input*

Example: `I D.hpdccB278`

where:

file_for_standard_input is used if the original command used the standard input file for its parameters or if a redirection is specified.

If the remote *mailx* command used the standard input file for its contents you would use this line.

Only one standard-input information line is allowed in an execution file, if present.

Standard Output Information Line

Syntax: `O` *file_for_standard_output system_where_directed*

Example: `O /dev/lp hpdcdx`

where:

file_for_standard_output — is the file or device where standard output is to be sent

system_where_directed — is the system name where the file resides.

Command Line

Syntax: `C` *command arguments*

where:

command — is the command to be execute

arguments — is an optional field that can consist of any options or filenames the command supports. For a further discussion of command parameters refer to the *HP-UX Reference* manual.

The command **must** exist in the remote command security file `/usr/lib/uucp/L.cmds`. The command must also exist in the `/usr/bin` or `/bin` directory unless an explicit and fully qualified pathname is given. The system automatically searches the `/usr/bin` and `/bin` directories when you issue a command, so it is not necessary to explicitly specify the complete pathname for the command.

If you do use an explicit pathname for a *uux* command, that pathname must exist in the `/usr/lib/uucp/L.cmds` file.

Typical Execution File

Here is an example of the contents of a typical execution file:

```
U dmr hpcnoa
F D.testsysA1569 DIALLOG
F D.testsysA1570 LOGFILE
F D.testsysA1571 LOG-WEEK
C pr DIALLOG LOGFILE LOG-WEEK
```

The `DIALLOG` source file for the first required file "F" line is also one of the "C" command line arguments.

Lockfiles and Temporary Files

Lockfiles are created to provide you with exclusive access to a system while you are using it. When you want to copy a file to or from a remote system, lockfiles lock out any other system from trying to communicate with that remote system. When a log file is being updated, locking that file ensures the file cannot be overwritten during the update.

Here are some examples of lockfiles:

`LCK..hpdsd`	lock call to system hpdsd; conversation in progress
`LCK.LOG`	lock LOGFILE for making an entry
`LCK..cul04`	lock /dev/cul04 while conversation is going on
`LCK.SQ`	lock the SQ file while the sequence number is being updated.

If a *uucp* program is aborted abnormally, one or more lockfiles may still exist, preventing future communication through use of *uucp*. They can be deleted by using the */usr/lib/uucp/uuclean* program that is discussed in the chapter, "Log, Status and Cleanup".

Uucp programs use temporary files to hold data being received until the entire transmission has been completed without error. The temporary file is then copied into the destination file, and the temporary file is automatically deleted.

Examples of temporary (TM) filenames are:

TM.<pid>.<count>

TM.00216.001

LTMP.* (Used and cleaned up internally by uucp programs.)

dummy

Log Files

The log summary file, named LOGFILE, is used to record all *uucp* connections (whether local or remote), the names of files transferred, the completion or failure of the transfers, the success or failure of autodial attempts, and the status of the *uux* commands. It is also used to record the time at which transfers took place.

The SYSLOG file is used to record the number of bytes sent or received from a system and the number of seconds used in the transfer. This file is used to report traffic statistics between connections.

The DIALLOG file records information about the modem used, the telephone number dialed and the result of the dialing. An example of a DIALOG file is shown in the chapter, "Log, Status and Cleanup".

The ERRLOG file records information about any errors encountered during communication processes.

Binary Files

The directory */usr/bin* contains the command files: *cu, uucp, uux, uulog, uuname, uupick, uustat* ,*uuto*, *uusnap*, and *uuls.*

Every time you issue one of these commands the system looks in the */usr/bin* directory for an executable file with the appropriate command name and executes it.

Library Files

When the *uucp* programs were installed in their proper directories, several files that supply remote connection information were created in the **/usr/lib** directory.

The System Administrator must edit each of these files except **SEQF** to add information obtained earlier from the remote systems contacted and information specific to local system needs.

The directory **/usr/lib/uucp** contains program modules needed by *uucp* commands. These program modules initiate and manage all communication with remote systems, and perform remote command execution.

The files (including complete pathnames) are:

/usr/lib/uucp/L.cmds	List of allowed *uux* commands
/usr/lib/uucp/FWDFILE	List of systems your system can forward through
/usr/lib/uucp/ORIGFILE	List of originating systems that can forward through your system
/usr/lib/uucp/SEQF	Keeps track of local sequence numbers
/usr/lib/uucp/USERFILE	Gives protection information for local users and remote systems
/usr/lib/uucp/L-devices	Defines devices and types of connections possible
/usr/lib/uucp/L-dialcodes	Contains strings that are abbreviations for telephone number prefixes
/usr/lib/dialit	Contains modem dialing sequences
/usr/lib/dialit.c	C language source file for *dialit*

/usr/lib/uucp/L.sys	Contains information concerning what systems your system knows about and how to make a connection
/usr/lib/uucp/ADMIN	Contains comments for each system that has access to yours.

L.cmds File

The `L.cmds` file determines what commands can be executed from remote machines using *uux* on your local system. This file can be edited to add new commands or delete old commands.

In this example of the contents in a `L.cmds` file:

```
rmail,node1,node2,node3
pr,node1
col
lp,node12
```

limits remote execution to four commands. System nodes are also included with some commands. This technique limits access to certain commands to specific system nodes. Where no system node is specified with the command, the command can be used by all systems in the network that have access to your system.

For security reasons, you probably want users on other systems to only be able to send mail to users on your system; you do not want remote users to read your local system *mail.* The `rmail` parameter permits execution of the receive mail command on your local system.

Only one command is permitted per line. The complete pathname does not have to prefix the command if the command resides in `/bin` or `/usr/bin`. If not, the full pathname must be provided. For example:

/BIN/*my_command*

indicates that the command, *my_command*, found in the /BIN directory is a valid command.

All commands found in the `L.cmds` file are executable by all remote systems if the systems have been assigned special permission to use these commands.

Security Sequence-Checking Files SEQF and SQFILE

The file *SEQF* is used by *uucp* programs to record the general sequence numbers used in workfile and data filenames. No external management is needed for this file.

The sequence number file *SQFILE* was not created automatically when *uucp* was installed on the system. It must be created manually by the System Administrator. The sequence number records the number of transactions between your local system and a remote system. Each remote system that you want to implement sequence checking with must have an entry for your system in its *SQFILE*. For example, to initiate sequence checking, *SQFILE* on sys1 has an entry for sys2, and the corresponding *SQFILE* on sys2 has an entry for sys1:

SQFILE on sys1	SQFILE on sys2
sys2	**sys1**

Thereafter, each time a transaction is made, the sequence number in the **SQFILE** on both systems is incremented. The **SQFILE** is used for explicit sequence checking between remote systems. These numbers must match before a transaction can be made.

For example, when you attempt to make a connection to a remote system and the sequence check fails, a message similar to the following is given:

```
dmr hpfcla (5/23-9:37-24748) HANDSHAKE FAILED (BAD SEQ)
```

The **BAD SEQ** message indicates a bad sequence number and the two systems are "out of sync".

Sequence files are used as security measures to ensure that *uucp* transactions are with the specified remote machine. Both machines keep a record of the name of the remote machine, a count of the number of transactions and the time of the most recent transaction. These files are updated after every transaction and compared. If the files on the two machines disagree, the connection is terminated and one of the files must be corrected manually to bring them back into agreement.

USERFILE

The **USERFILE** specifies the type of access permission that is granted to both local users and remote systems; this is the major security tool for the *uucp* facility. **All users should read this section.**

Three methods can be used to implement system and file security:

- Require a remote system be called back to verify its identity
- Check the user login name
- Restrict file access paths available to remote systems

A **USERFILE** line entry has four main fields:

```
<user>,<system_name> [c] <pathname>
```

There must be a blank or tab between the **user,system_name** field, the **c** option if used and each **pathname**.

where:

user — associates the access permission for a named user on your local system or is used to determine whether a call back is required for the remote system

system_name — determines the access permissions for a remote system logged into your local system

[c] — is an optional field indicating that the remote system logged in as **user** should be called back. When the remote system tries to log into your local system, the remote system is called back to verify its identity

pathname — is a list of pathnames separated by blanks. This is the critical list which gives the **user** or **system_name** access along the specified paths. If the pathname ends at a directory, all files and sub-directories in that directory are accessible.

Entries must be in **both** USERFILE and */etc/passwd.* For every entry in USERFILE, there must be a corresponding entry in */etc/passwd.* The user ID field must be unique and not zero in */etc/passwd.* Here is an example:

```
user.hpdsd, hpdsd pathname
```

requires in */etc/passwd*:

```
user.hpdsd::5:5::x:xuucico
```

where the first `5` is a unique user ID field.

At the time you install your system `USERFILE` contains:

```
uucp, /
, /
```

which provides unlimited access for all users on your local system and for all remote systems logging into your local system. You can restrict the access by editing this file.

The following example shows the contents of a typical `USERFILE`:

```
dmr,hp-sys1   /users/dmr /dev/null
emd,hp-sys2   /usr/spool/uucppublic /dev/null
kls,hp-sys3 c /
uucp,         /usr/spool/uucppublic /dev/null
,            /usr/spool/uucppublic /dev/null
```

The fourth line provides access to the two specified paths: */usr/spool/uucppublic* and */dev/null* for all *uucp* users on your local system and all remote systems not previously specified. */dev/null* is the default input and output file for the *uux* command. The last line gives all not previously specified users on your local system access to the same two paths. You will probably want to include these permissions in your `USERFILE`.

The following lists show the searching sequence for `USERFILE`:

USERFILE SEARCH

(file on remote system)

1. Check the user field. In order for a remote system to log onto your system, its *user* name **must** appear in the system's *USERFILE*. It is also recommended that you have the remote system's *system_name* field in your *USERFILE*, but it is not necessary. If the remote system, when communicating with your system, has the correct *user* field but the *system_name* field was left blank, there must **not** be a line before it in your *USERFILE* that contains the correct *system_name* field and the incorrect *user* field, or permission for the remote login will be denied. For example, when a remote system calling you has a *user* field of **uucp** and a *system_name* field of **remote1** and your *USERFILE* contains the following:

   ```
   test1,remote1 /pathname /pathname
   uucp,         /pathname
   ```

   login permission will be denied to the remote system. In this example, your system tests the remote system's *user* field (**uucp**) and finds it is incorrect; however, the remote system's *system_name* field (**remote1**) is correct and your system sets a flag and continues to search for the correct *user* field. When the correct *user* field is found and *system_name* field is blank, log in permission is denied to the remote system.

2. Check the **system_name** field for system access to the path/file for each file transfer request.

   (If no **system_name** match is found, use the first blank or null **system_name** field if and only if that line is not the same line used for a blank user field.)

USERFILE SEARCH

(file on local system)

1. Check the path/file access for local user permission (if no user match is found, use the first blank or null user field).
2. Check to verify that the path/source_file is readable or the destination_file path is readable and the destination file is writeable.

The **USERFILE** is searched sequentially for the **user**. If the search does not find the **user**, the **first** entry encountered with a null user entry is employed. **Be cautious**. The first **user** entry that is null defines the access permission for **all** users who are not specifically named. If you put a null **user** field before some named **user** fields, the search finds the null field first and stops searching; the named users after the null user field are never searched. When a match or null field is found **and** the user is a remote system, the *uucp* program checks to see whether a callback is required. If so all activity stops until the remote system is called and its identity verified. If the user is on your local system, the *uucp* program checks the **USERFILE** for pathnames that user is granted access to. If no match or null field is found for the user, an access denied message is generated.

The next sequentially checked field is the remote **system_name** field. This field specifies access permission for remote systems to any pathnames on the match or null system_name line. All users on a specific remote system are restricted to the same path(s). If the remote system_name does not have permission to either read a source file or write a destination file, an access denied message is again generated.

Note that the **user** and the **system_name** fields are divorced from each other. *Uucp* starts its search process at the first line of the **USERFILE** for **each** field. For remote systems, the **user** field is only searched for the callback option. *Uucp* starts at the beginning of **USERFILE**, searching for a **system_name** field that restricts remote systems to the paths specified on that line. The users field for **local users** is combined with the paths described on the specified line when restricting access to that path.

Although there may be several lines with the same system name, *uucp* must find either the name of the remote system, or a null system name on the system name line before the information transfer is permitted.

Users on a local system must also have permission to access their own files when using *uucp* facility commands.

When *uucp* reaches a null or blank user field before finding the wanted user name, that line is used for local user pathname access or remote system callback. If *uucp* reaches the same line in its system_name search and finds a blank or null system_name, that line **cannot** be used to grant remote system access to the pathnames specified. *Uucp* continues its **system_name** search on succeeding lines. For example:

```
,       /dev/null
,hpsys1 /
```

gives **hpsys1** access to all paths on your local system. It **does not** give all unmatched users on all unmatched systems access to the path /dev/null.

When the *uuxqt* daemon encounters an X.* file in the /usr/spool/uucp directory, the L.cmds file is checked to determine whether the command can be executed by **all** remote systems. The USERFILE is then searched to find the first **null** system field for path access permission.

L-devices File

The L-devices file defines the devices and types of connections that are valid for **both** the *uucp* facility and the *cu* terminal emulator. Each entry describes a connection type, the /dev entries for the connection, and the speed at which the connection is made.

When this file is automatically created at installation time, the file contains the following lines as examples for you:

```
<type> <cul> <cua> <speed> <proto>
DIR tty04 0 9600
DIR tty12 0 9600
ACUVADIC3450 cul03 cua03 1200
DIR tty09 0 1200
ACUVENTEL212 cul06 cua06 300
```

where:

type denotes the type of connection. This may be a direct connection indicated by DIR or an autodial modem connection, automatic calling unit (ACU) indicated by the string, ACUmodem_name, the name of the autodial modem in use. The maximum length of the string is 19 characters. This modem connection must have an autodial routine in /usr/lib/dialit.c

cul is the */dev* entry name for the main data line you specified when you used the *mknod* command. This line is used to transmit all data once the connection is made with the remote system. It is recommended, but not required, that you use an entry_name with a cul prefix for dialout lines and tty for dialin lines

cua contains a zero (0) if the line refers to a direct connection (DIR) or contains the name of a /dev entry name if the line refers to a modem connection (ACUmodem_name). It is this line which is passed to the dialit routine[6]

[6] The entries for <cul> and <cua> can be the same if you have only one physical line connection. You can also have two entry_names in the */dev* directory with the same select code for dialout lines

speed specifies the speed of the data communications line associated with the `L-devices` entry (*uucp* allows speeds of 300, 1200, 2400 4800 and 9600 baud). The speed you choose depends on the restrictions of the remote system

proto a single letter specifying the protocol to be used on that line.

You need to to define your own direct and modem connection devices by inserting appropriate entries similar to the examples given.

Order is important when using the *cu* command. For example:

```
cu -ltty09 -s1200 dir
```

causes the system to look at its direct connection nodes for `tty09`. In the above list of `L-devices` contents, the first matching entry specifies 9600 baud, so the line connection is opened at 9600 baud, even though you explicitly stated 1200 baud speed, `-s1200`. The `-s` speed option has no effect with *cu* if a direct connection is specified; once a rate is found it can only be changed using the following command line:

```
~!stty 1200 < /dev/tty09
```

This order restriction does not apply to the *uucp* command.

L-dialcodes File

The L-dialcodes file specifies telephone prefix translations. Each entry in this file contains two fields: an identifying string and the string number you want substituted when you use the identifying string. For example:

```
boston 131-149
```

When you want to use a modem connection to contact a remote node, your system looks in the L.sys file for the phone number of that remote node. If you have used the string, `boston`, in the phone field for that remote system, your local system then looks in the `L-dialcodes` file for the translation of `boston` into an actual telephone number, `131-149`.

These abbreviations for telephone number prefixes can reduce some time, typing and mistakes.

Dialit.c

The `dialit.c` file is the C language source of the `dialit` module used by *uucp* to perform the autodialing needed to contact a remote system with a modem connection. This file was installed in the */usr/lib* directory when you used the *optinstall* command.

Four example dialing routines are supplied for you in `dialit.c`. You may need to edit these routines or modify the USERSUPPLIED ROUTINES section for your specific needs. Comment lines in `dialit.c` and the following chart will help you modify these routines. The procedure headers in `dialit.c` contain information about modem configuration.

Table 4-1.

For this modem	Use this name in the L.sys device field and the L-devices type field
VENTEL 212	ACUVENTEL212
VADIC 3450	ACUVADIC3450
HP 35141A	ACUHP35141A
HP 92205A	ACUHP92205A
HAYES	ACUHAYESSMART
HP37212A	ACUHP37212A

The programs contained within this module are examples of autodialing routines for selected modems currently on the market. Hewlett-Packard makes no claim as to the validity or reliability of this code. These programs are not supported products, but simply examples for our customers. Their compatibility with future products is not guaranteed.

In certain areas, especially those with tone dialing, the numbers may be dialed faster than the PBX can respond to. In this case, you should insert a "-" or any defined pause signal between the numbers dialed.

The following code is an example of part of the `dialit.c` program:

```
static char Uni_id[] = "@(#)20.1";
/* UNISRC_ID:   @(#)dialit.c    20.1      84/03/22    */
/*****************************************************************
 *    (c) Copyright 1984 Hewlett Packard Co.
 *        ALL RIGHTS RESERVED
 *****************************************************************/
/************************************************************************
 *            * * * * D I S C L A I M E R * * * *
 *  The programs contained with in this module are examples of autodial-
 *  ing routines for selected modems currently on the market. H.P.
 *  makes no claim as to the validity or reliability of the code in this
 *  module. These programs are not supported products, but simply examples
 *  for our customers. Their compatibility with future products is not
 *  guaranteed.
 ************************************************************************/
/***************************************************************************
 *  This module consists of:
 *     main routine - this routine is the main entry point into the module.
 *     The usage of this routine is:
 *               dialit <modemtype> <cua> <phone> <speed>
 *     Where:
 *        modemtype - is the name of a modem know in the Modem structure
 *                    along with a user supplied routine to do the
 *                    autodialing. The standard is for the modemtype to
 *                    be a name of the form:
 *                             ACUmodemname
 *                    such as:
 *                             ACUVENTEL212
 *                    This convention is followed since this is the form
 *                    expected by both uucp and cu which utilize this program
 *                    to perform their autodialing.
 *
 *        cua - This must be the full pathname of the /dev entry over which
 *              the auto dial sequence is to be sent to the modem. In the
 *              case of uucp and cu this entry is pulled from the L-devices
 *              file. NOTE: that in the L-devices file the full pathname is
 *              not given. But uucp and cu do expand it before calling this
 *              module.
 *
 *        phone - The phone number to be called by the autodial modem. The
 *                phone number may consist of digits, '=' and '-' only.
 *                The special characters are mapped to wait for secondary
 *                dialtone(if implemented on the modemtype) or 5 second
 *                pause respectively.
 *
```

```
 *        speed - This argument is the speed desired for transmission,
 *                i.e. 1200,300,etc. The inclusion of this parameter
 *                allows you to configure the cua line. If the dial
 *                routine is called from cu or uucp the line has already
 *                been configured.
 *
 *     sendstring routine - writes the designated string to the device
 *          whose descriptor was sent it.
 *
 *     await routine - will read from the designated device a sequence of
 *          characters until a certian string is recognized or a specific
 *          number of characters is read.
 *
 *     ckoutphone routine - scans the phone string and checks for invalid
 *          characters and determins a delay time used for alarm timeout
 *          purposes when calling the remote machine.
 *
 *     map_phone routine - map the characters '=' and '-' which mean wait
 *          for a secondary dial tone and pause respectfully to their
 *          actual character representation for given modems.
 *
 *     log_entry - make an entry into the DIALLOG which resides in /usr/spool
 *          /uucp.
 *
 *     make_entry - called by log_entry. makes the actual entry in the logfile.
 *
 *     prefix - tests a string to determine if it begins with a given prefix.
 *
 *     mlock - lock the logfile so only one process may write to it at a time.
 *
 *     remove_lock - remove the logfile lock and allows another process to
 *          access the log.
 *
 *     close_log - cleans up any temporary log files created and closes the
 *          log file.
 *
 *   USERSPECIFIED ROUTINES:
 *    these routines are supplied by the users of the uucp package. Each
 *    routine is written for a specific type of autodialer modem and must
 *    have an entry in the Modems structure.
 *
 *********************************************************************/
#include <stdio.h>
#include <termio.h>
#include <setjmp.h>
#include <sys/types.h>
#include <signal.h>
#include <ctype.h>
#include <fcntl.h>
```

```
#include <sys/stat.h>
#include <sys/dir.h>
#include <pwd.h>

#define FILENAMESIZE 15
#define MAXFULLNAME  100
#define MAXMSGSIZE 256
#define SAME     0
#define FALSE    0
#define TRUE    -1
#define FAIL    -1
#define SUCCESS  0
#define MAXRETRIES  3
#define PREFIX "DIAL."
#define LOG_LOCK  "/usr/spool/uucp/LCK..DIAL"

jmp_buf Sjbuf;
char *modemtype;             /* modem name as entered in the L-devices file
int alarmtout();

/****************************************************************
 *  The following structure Modems is used in determining which user
 *  supplied routine is to be used for autodialing given a specific
 *  modem type.  Each user specified routine must have at least one entry
 *  in this structure and each modem type used for autodialing must have
 *  only one entry in the structure.
 *
 *  To add additional modem types and routines simply add them to the
 *  initialization of modem[].
 ****************************************************************/
int vad3450(),                     /* function name for vadic3450*/
    ventel212(),                   /* ventel 212+3 function */
    hp35141_autodial(),            /* hp support link modem */
    hayes_smart();                 /* hayes smartmodem1200 function */

struct Modems{
      char *name;                  /* modem name */
      int (*modem_fn)();           /* function to call */
} modem[] = {
     "ACUVADIC3450",    vad3450,
     "ACUHP35141A",     hp35141_autodial,
     "ACUVENTEL212",    ventel212,
     "ACUHAYESSMART",   hayes_smart,
     "ACUHP92205A",     hayes_smart,
      0,
};
```

There are lines at the beginning of dialit.c that define the integer functions used and the module functions that can be called by the dialit program.

Locate the lines that define the integer functions and add your modem name. For example:

```
int hays1200();
```

Next define your modem connection by adding its name and function to the modem[] structure. For example:

```
"ACUHAYS1200",   hays1200(),0
```

Now locate the USER-SUPPLIED-ROUTINES section at the end of this module and add the code necessary for your specific type of modem. Note that the await routine now has a parameter specifying string length.

After you have finished modifying the `dialit.c` source code, you **must** compile the code and store the compiled version in the `dialit` file.

Dialit File

Note that the `dialit` pathname does not include the *uucp* sub-directory name.

The `dialit` file contains the object code necessary to implement autodial sequences necessary for the specific modem you are using. When the *uucp* program checks the `L-devices` file and finds the device for the remote system uses a modem connection, the `dialit` routine is called to perform the autodialing and an entry is made in the `DIALLOG` file.

Refer to the `dialit.c` section above for information on modifying the contents of the `dialit` file.

L.sys File

The `L.sys` file is used by the uucp facility to provide information on which systems can be contacted, when the remote system allows communication, whether the connection to the remote system is direct or modem, the baud rate (speed) of the data communications link, and how to log in on the remote system.

Here is an example of what an automatically installed *L.sys* file might look like:

```
<sysname> <time>[,<retry>] <dev> <speed> <phone> <logininfo>
sys1 Any,1 tty04 9600 tty04 login:-EOT-login: uucp
sys2 Mo0800-1730,10 tty06 1200 tty06 login:-BREAK-login: access
sys3 Wk0800-0600 tty03 1200 tty03 login: network password: hpdcd
sys4 Any,5 ACUVADIC3450 1200 555-1212 login:uucp
sys5 Any,5 ACUVENTEL212 300 999-7777 login: sys5 ssword: uucp
sys6 Any,5 ACUHP2334A 9600 f/123456789 login: sys6 ssword: h9
```

where:

sysname is the remote name of the system whose contact name is contained on the entry line. This name may be up to seven characters. If you have alternate means of communicating with a certain system, you can have more than one entry in the `L.sys` file with the same sysname. This file is searched sequentially to determine if the requested system name matches a sysname. If you are a PASSIVE (receiving calls) system with respect to **sysname**, the remaining five fields are blank. Specifying the name permits queuing work

`time` gives the time of day as well as the days sysname allows communication. Days of the week are specified by: *Su*, *Mo*, *Tu*, *We*, *Th*, *Fr*, and *Sa*. *Wk* specifies weekday (Monday through Friday, and *Any* specifies any day of the week). The day specification is followed by the times permitted in 24-hour format. If the day specification is omitted, *Any* is assumed, by default. For example:

```
0800-0600
```

allows calls from 8:00 AM to 6:00 AM the following day which is equivalent to any time except between 6 and 8 AM. This example:

```
Su Mo Tu 0600-2300
```

allows calls Sunday, Monday and Tuesday between 6:00 AM and 11:00 PM. If time of day is not specified, *all times permitted* is assumed by default.

You can also specify "`NEVER`" which means you do not want to call a system but that system may want to call you (you need an entry in `L.sys`).

`[,retry]` is an optional field indicating the waiting time in minutes between a failed call connection to a remote system and the next retry. The default waiting time is the minimum required wait of 5 minutes. If you specify less than 5 minutes, the default of 5 minutes is substituted.

Note that this retry time specifies the **wait** time only. It does not specify that a retry dial operation will be attempted

`dev` indicates whether a direct (`DIR`) entry or a modem (`ACUmodemname`) entry must be found in the *L-devices* file. This field is the same as the `<cul>` field of the corresponding `DIR` entry in *L-devices* file

`speed` specifies the speed (baud rate) at which communications take place. The union of the `<dev>`, `<speed>` and `<type>` fields in the `L-devices` file are searched for the proper entry to use

phone is the telephone number of the remote system to be called during the login connection process. For direct connections the phone field is the same as the dev field. The telephone number can contain a string, such as **boston** that is used as a search string when scanning the *L-dialcodes* file where it is translated into the associated telephone prefix such as **999**.

Samples of permitted characters are:

- Digits zero (0) through nine (9),
- Equal sign (=): wait for secondary dial tone, and
- Minus sign (-): pause for five seconds.

These are generic examples of the characters; your modem may use different characters that you must map to the above meanings in the dialit file.

Maximum allowable length of the translated telephone number is 29 characters.

Uucp can be used on two types of communication links, so two protocols are provided so you can obtain efficient use of both link types. The *f-protocol* is used with X.25 (see the chapter, "The X.25 Network") lines while the *g-protocol* is suitable for regular telephone lines. Note that the corresponding *f* or *g* protocol character is prefixed to the phone number. For example,

```
f/555-3111
```

which says *f-protocol* is being used and

```
g/555-3111
```

says *g-protocol* is being used.

You can specify *uucp* protocol that includes the use of both *f* and *g* protocol. For example:

```
fg/cul05
```

Here, `fg/` specifies that *f* or *g* protocol can be used, but *f* is given higher priority.

If there is no protocol character is specified, *uucico* defaults to *gf-protocol.*

logininfo is a field containing the information necessary for logging onto the remote system. This field should contain the prompt you expect to receive from the remote system, followed by a space and the response you are expected to give. For example, suppose you expect the prompt: **login** and your response **sys5** followed by the prompt: **password** and your response **abcxyz**. Your *L.sys* file entry becomes:

```
login: sys5 password: abcxyz
```

where **sys5** is the user name on the remote system, and must be in the remote system's USERFILE.

Login information is given as a series of fields and subfields in the format:

```
[expect send]
```

where **expect** is the string expected to be read and **send** is the string to be sent when the expect string is received.

The expect field may be made up of subfields of the form:

```
expect[-send-expect]...
```

where the **send** is sent if the prior **expect** is not successfully read and the **expect** following the **send** is the next expected string. For example, **login-@-login** expects **login**. If a **login** is received, the program goes on to the next field; if it does not get the **login** it sends **null** followed by a new line, and then expects **login** again.

NOTE

After the connection to a remote system is made, you may need to add an "**@**" before the login message is received; this is most common on direct connections. If you use:

```
"" @ gin:-@-login: sys5 password: abcxyz
```

"**@**" is the line 'kill' character before logging in. **abcxyz** represents a valid password, and should be replaced with a valid password string for the remote system. This is a very reliable method for restoring order to an uncooperative new connection to a remote UUCP facility.

When `L.sys` was installed, the protection mode, 444, was "readable by everybody". Since this file may contain proprietary information, you can change its mode to 400 so that only the owner, *uucp*, can read the contents. Be sure that *uucp* remains the owner because *uucp* **must** be able to read `L.sys` in order to handle data transfers.

The **send** string can also contain:

`\s`	blank
`\d`	1 second delay
`\c`	no carriage return on the **send** string
`\N`	null character
`\\`	backslash
`EOT`	two CONTROL-Ds
`BREAK`	send a break

A correctly structured **logininfo** field should resemble the following:

```
login:-\d\d\d@\c-login: XYZ ssword: Ply.
```

ADMIN File

The ADMIN file is used to keep comments for each system with access to yours. Each entry is formatted as: *sysname* <tab> *description*, where the description should be a useful comment about the system that will be displayed when **uuname** is used. Also, be sure you separate with a tab, not a space. If the other system is never called by the current system (the word "NEVER" exists in the `L.sys` entry for that system), then an entry in the description should be put in `ADMIN` indicating the other system calls the current system (but not the other way around). The file is as follows:

```
/usr/lib/uucp/ADMIN
```

Uucp Facility Daemons 5

Uucp daemons are programs that perform the necessary operations for the *uucp* facility to transfer information. This chapter discusses daemons and their specific functions.

Running The Uucp Facility

When you execute a *uucp* command two things happen:

- Workfiles are set up in the */usr/spool/uucp* directory
- A child process is spawned. The child process invokes the *uucp* communications daemon:

  ```
  /usr/lib/uucp/uucico -r1
  ```

Uucico is the mnemonic for UNIX-to-UNIX copy-in copy-out. The **-r1** option specifies that *uucico* act in the **master** role.

Uucico scans directory */usr/spool/uucp* for the workfile having the highest grade. At least one workfile exists in the spool directory because the *uucp* command that spawned *uucico* also set up a workfile (unless *uucico* was started manually as in the example above).

Next *uucico* examines the system names, either local or remote, that are part of the source and destination file names. If a remote system is specified, *uucico* looks in the *L.sys* file to determine how to contact the remote system. For modem connections, *L.sys* tells *uucico* what type of modem to use, the telephone number, the data transfer speed (baud rate), and the login information. *Uucico* now looks in file */usr/lib/uucp/L-devices* to determine if the modem to use is a valid device. The `L-devices` entry (which must match the speed in the `L.sys` file) tells *uucico* which data communication line (*/dev/line_entry*) is connected to the modem.

Uucico then checks to see if another *uucp* facility is using the line. If not, *uucico* creates lock files for the line in directory */usr/spool/uucp* as well as for the remote system it is trying to call. These lock files implement a binary semaphore mechanism.

Uucico now spawns a child process which in turn invokes the */usr/lib/dialit* program when making the actual call to the remote system. Once on-line or connected, *dialit* returns a status report to the parent *uucico* process.

Uucico now uses the login information in the `L.sys` file to attempt to login on the remote system. There must be an entry in file */etc/passwd* on the remote system of the form:

```
uucp::5:5::/usr/spool/uucppublic:/usr/lib/uucp/uucico
```

It is important to note two special things about this */etc/passwd* entry:

1. The login directory for *uucp* purposes **must** be */usr/spool/uucppublic.*
2. The */usr/lib/uucp/uucico* daemon **must** be invoked instead of the normal shell */bin/sh*; if this is not done, communication can never take place.

At this point, *uucico* on the system that originated the call is the master because it was invoked with the **-r1** option. Another *uucico* is automatically invoked on the remote system where it functions as the slave. The slave sends the master a message `Shere`, meaning, "slave is here".

When the master receives the `Shere` message, a series of messages is sent back and forth to establish correct handshake and communication protocol.

Refer to the second example at the beginning of the "Uucp File Structure" chapter for a description of the conditions that result in activating a reversal of master/slave roles.

Once protocol and handshaking are established, the master sends a request and waits for the approval of that request by the slave. If the request is approved, transfer of the file begins. The file is broken into 64-byte packets each of which includes a checksum used to verify that the packet has been received without error. If a packet is not correctly received (checksum test failed), it is re-transmitted up to five times. After the fifth unsuccessful try, *uucico* assumes a bad connection and terminates the connection.

A new connection is established when another *uucico* is invoked.

If transfers are being handled without exceeding the retransmission limit, the master continues transmitting requests to the slave until there is no more work for that system. The master then sends the slave a hangup request. When the slave receives the hangup request it scans its spool directory for any work files that have the master's (remote) destination. If none are found, the slave returns a hang-up OK message to the master and the connection is terminated. If the slave has work for the master, a hang-up denial message is sent indicating to the master that the slave has work to send. At this point,

the roles of master and slave are switched. The new master starts sending requests to the new slave. When all work has been sent, the data communication (datacomm) link is disconnected.

Upon completion of transfers and disconnection of the datacomm link, each *uucico* daemon (master and slave) spawns a child process and dies. The child process invokes the */usr/lib/uucp/uuxqt* execution daemon. *Uuxqt* scans directory */usr/spool/uucp* for any execution files created by the *uucico* transfer (remember that the initial workfile with a grade of "X" becomes an execution file upon transfer. If any execution files are found, *uuxqt* attempts to execute them, then *uuxqt* terminates.

A file is the smallest unit that can be transfered by *uucp.* If a connection is terminated because a packet could not be successfully transferred and a new connection is made, the packet cannot be retried; the entire file transfer must be rerun. When errors occur, short files are more efficient because the necessity of retransmitting all of a large file when a transmission failure occurs near the end of the file is avoided. On the other hand, splitting a large file into smaller files then recombining them at the other end also requires time, so file splitting overhead and retransmission time must be balanced against each other when planning system operation and tuning communication procedures for best overall efficiency.

Invoking uucp Daemons

Although *uucp* daemons are normally invoked as a result of *uux* or *uucp* commands, either you as a user or another *uucp* daemon can also initiate them.

Uucico Daemon

The syntax for invoking the *uucico* daemon is:

```
/usr/lib/uucp/uucico -r1 [-ssystem_name] [-xn] &
```

where:

`-r1`	invokes *uucico* as the master
`-ssystem_name`	is an optional parameter indicating the node name of the system you want to contact
`-xn`	is an optional parameter providing debug or error information (n is a digit between 1 and 9 where higher values print more information)
`&`	is used to execute the process in the background so your terminal is not tied up.

Uuxqt Daemon

The *uuxqt* module performs local command execution of execution (`X.*`) files from remote systems. The syntax is:

```
/usr/lib/uucp/uuxqt [-xn] &
```

where:

`-xn`	is the same as for *uucico* above
`&`	runs the process in the background so your terminal is not tied up.

Uudemons

Uudemons are script files that are used to:

- Periodically clean up certain files such as log files.
- Communicate with systems that are waiting for you to contact them.

These script files are normally executed by entries in file */usr/lib/crontab.*

The *uudemon.hr* script that is shipped with most *uucp* facilities cleans up old status files and lock files and polls all systems for which you have work pending on an hourly basis. If you use the **-ssystem** option *uudemons* forces a call to the system specified. This is necessary for PASSIVE only systems (systems waiting for contact from another system) that cannot initiate communication with you. You can edit this file by replacing **-s<nodename>** with the system name you want polled.

Using the Uucp Facility 6

You can begin using *uucp* commands as soon as the *uucp* facility has been installed on your system, including hardware configuration, software configuration and neccessary file editing. Refer to the early chapters of this tutorial for installation and configuration information.

You must be logged in on your local system before you can use these commands.

Syntax Information

Pathnames

Just as you must use unique names to reference files on your local system, unique *system_name!pathname/filenames* are needed to identify files on remote systems. The system name is separated from the complete pathname to that file by a ! character, which is sometimes referred to as "bang".

An example of a complete pathname (fully qualified) to a file is:

```
rem_node!/usr/sys_dir/your_dir/your_file
```

You also have the option of using the ~ character to represent a login directory. If you use **~your_user_name**, the remote system expands this to the **your_user_name** login directory, for example:

```
rem_node!~your_user_name/your_file
```

would be expanded by the remote system to:

```
rem_node!/usr/sys_dir/your_dir/your_file
```

if the login directory for **your_user_name** is **/usr/sys_dir/your_dir**.

If you specify **~/file_name**, the *uucp* facility uses the public area in the spool directory: **uucppublic**.

If you do not use any pathname after the system name, the **current local directory** or the remote login directory is used.

The pathname syntax for files within the **current local directory** supports any shell meta-characters like:

`*` a "wild card" character indicating zero or more characters

`?` any single character

`[...]` ending character(s) for the file.

The *uucp* command allows you to exchange files with other systems by using remote systems as links into the system you wish to obtain information from or send it to. For example, you might type a command line that looks like this:

```
uucp message node1!node2!node3!/usr/spool/uucppublic/file_name
```

This sends a **message** to the system **node3** (by way of **node1** and **node2**) and places it in the default directory */usr/spool/uucppublic* on that system. The **message** is placed in a file named *file_name*. Forwarding files through several systems is discussed in greater detail later in this chapter.

Option Separators

Square brackets indicate optional parameters; spaces and the - character are **required between each option**.

The Cu Command

The *cu* ("call UNIX") command manages interactive communications with HP-UX or UNIX remote computers, as well as with non-UNIX remote computers. It functions as an asynchronous terminal emulator.

The *cu* command is used interactively to transmit messages to and from remote systems and to transfer ASCII files. You can also use the *cu* command to interactively contact a remote system to verify that the connection is working properly before you invoke the *uucp* or *uux* commands.

Cu tries each line in the `L-devices` file (which specifies acceptable devices and types of connections) until it finds a match with the parameters specified or until it runs out of entries.

Cu Command with a Modem Connection

The syntax for using the *cu* command with a modem connection is:

```
cu [-sspeed] [-l line] [-q] [-h] [-m] [-t] [-o|-e] tel_num|sysname
```

where:

-sspeed	specifies the transmission speed of 110, 150, 300, 1200, 2400, 4800, or 9600 baud. The default is 300 baud.
-lline	specifies the modem device name to override searching for the first available device with the correct speed
-q	enables the ENQ/ACK handshake
-h	emulates local echo. The remote system expects your terminal to be in half-duplex mode.
-m	ignores modem controls
-t	is for adding CR to LF on output to remote (for terminals)
-o|-e	designates even or odd parity. The default is no parity.
tel_num	is the telephone number of the remote system. Only digits, "-" meaning pause and "=" meaning wait for secondary dial tone are allowed.
sysname	gives the name of a system that appears in *L.sys.*

Some examples of using the cu command with a modem connection:

`cu -s1200 -e -q 555-1212`	(direct dial line, 1200 baud)
`cu -s1200 -o -q 9=555-1212`	(dial from PBX line, 1200 baud)

The second example is for a private branch exchange that requires dialing "9" and waiting for a secondary dial tone before dialing the number.

These messages are displayed on your CRT as the connection is made:

autodialing - please wait
Connected
login: (from remote system)

If the autodial failed, the message:

```
Connect failed: autodial failed
```

is generated or (in some cases) a reason for the failure is given.

You need to know how to login on the remote system in order to respond correctly to its **login**: prompt.

Cu with Direct Connection

The syntax for the cu command with a **direct connection** is:

```
cu [-h] [-q] [-m] [-t] [-o|-e] -lline dir|sysname
```

where:

-h	emulates local echo (default is full duplex)
-q	enables the ENQ/ACK handshake
-m	ignores modem controls
-t	is for adding CR to LF on output to remote (for terminals)
-o|-e	designates even or odd parity (default is no parity)
-l	specifies the device name and is a mandatory parameter
dir	is a character string that must be used for a direct connection
sysname	gives the name of a system that appears in *L.sys*.

With a direct connection the `-sspeed` parameter is ignored. The *cu* facility uses the speed field in the `L-devices` file.

For example, if you type:

```
cu -ltty09 -s1200 -m dir
```

and the line in the valid devices file for the direct connection of tty09 is:

```
DIR tty09 0 9600
```

the line specified by tty09 is opened at 9600 baud; **not** 1200 baud. To change to 1200 baud, execute:

```
~!stty 1200 < /dev/tty09
```

Here are some examples of using the *cu* command for a direct connection:

```
cu -ltty09 dir
cu -h -ltty09 dir (remote expects you to be in half-duplex mode);
cu -o -ltty09 dir (odd parity)
```

The login prompt appears on your CRT if the connection is made correctly.

After Connection

Cu reads data from the standard input file and passes it to the remote system when the transmit process is active. *Cu* accepts data from the remote system and passes it to the standard output file when the receive process is active.

Transmitted lines beginning with a "~" have special meanings:

~%cd path change directory (default is $HOME)

~. terminate the connection

~%b transmit a break character (you can also use "~%break")

~! [command] escape to local shell and return by EOT. If you specify a command on this line, the local shell executes the command and returns.

~& [command] run the command, but kill the *cu* "receive" process and restore it later

~$ [command] run the command locally and send its output to the remote system

~%take from [to] copies the file "from" on the remote HP-UX or UNIX system to the file "to" on your local system. If you do not use the optional [to] parameter the file "from" on the remote system is copied to a file with the same name, "from", on your local system and created if none exists.

~%put from [to] copies the file "from" on your local system to the file "to" on the remote HP-UX or UNIX system. If you do not use the optional [to] parameter the file "from" on your local system is copied to a file with the same name, "to", on your remote system.

~%<file upload file with prompt handshake

~%setps xy set prompt sequence to **xy** (default DC1)

~%setpt n set prompt timeout to **n** seconds

~%set tell what the current timeout and prompt are

~~... send the line "~..." to the remote system

~%nostop toggle the DC3/DC1 input control protocol off and on

~%>file begin output diversion to file

~%>>file append to file

~%> end any active output diversion.

To transfer the ASCII file, "My_file", on your local system to a file named "My_rem_file" on the remote system, type:

```
~%put My_file My_rem_file
```

Do not press any key on your keyboard while transferring files with "%take" or "%put". The facility may transmit incorrect data or be left in an unstable state.

Do not terminate the *cu* program while a communication is in process.

Using the uucp Command

The *uucp* command is a background program that is used to transfer files to and from remote HP-UX or UNIX systems. *Uucp* creates work files and data files in the `/usr/spool/uucp` directory for later processing.

General uucp Snytax

The general syntax for the *uucp* command is:

`uucp [options]` *source_file(s) destination_file*

for *uucp* to copy the *source_file(s)* to the *destination_file.*

The following options can be used:

-c	use the source file **when** copying out (rather than making a copy of the source file at the time the command is issued and storing it on the spool directory for later processing–this is the default)
-C	make a copy of the source file at the time the command is given and store it on the spool directory
-d	make all necessary directories for the file (this is the default)
-esys	send the uucp command to the remote system sys. This option is only successful if the remote system allows the *uucp* command in its `L.cmds` file.
-f	do not create intermediate directories
-ggrade	request a grade for work sequencing (a grade of "A" specifies "do this work first, "z" specifies last and "n" is the default)
-m	send mail to the requester when the copy is complete
-nuser	send mail to notify the user on the remote system that a file was sent
-r	create the files necessary for the transfer to take place, but do not invoke uucico to call the remote system.

The *source_file(s)* must exist and both that file and its path name must be readable by everyone.

The **destination file** does not have to exist; *uucp* creates a file by that name if none exists. If the destination file already exists, it must be writable by everyone and the pathname must be readable and writeable by everyone. Your System Administrator can change the access permissions to files and paths.

For example, if you type:

```
uucp /usr/sys_dir/user_dir/file1 hpsys1!~uucp/file2
```

A workfile with a name similar to *C.hpsys1n2270* is created in */usr/spool/uucp*, whose contents resembles:

```
S /usr/sys_dir/user_dir/file1 ~uucp/file2 sys_dir -dc D.0 444
```

Do not use *uucp* to copy a **local** source file to a **local** destination file. **Do** make sure the directory is readable, writable, and executable, or `uucp` will put the file in `/usr/spool/uucppublic` and give you an error message (in the form of a mail message).

Sending Files To a Remote System

The following examples send single or multiple files to a remote system. To send *file1* in the current directory to */users/hpfsd/file2* on remote system `hpsys1`, use:

```
uucp file1 hpsys1!/users/hpfsd/file2
```

To send */usr/all.exp/cmd/file1* on the local system to */users/hpfsd/file2* on the remote system `hpsys1`, use:

```
uucp /usr/all.exp/cmd/file1 hpsys1!/users/hpfsd/file2
```

Using the tilde (~) character as a login directory substitution:

```
uucp ~kls/file1 hpsys1!~uucp/file2
```

sends *file1* on the login directory for *kls* to *file2* on the remote system, `hpsys1`, in the login directory for *uucp*. Note that in this example, ~ is used to represent both the local login directory and the remote login directory. In this case, `~uucp` after `hpsys1!` in the destination file field is used, not as the *uucp* command, but as a typical name for a remote login directory.

A copy is made of *file1* at the time that the *uucico* daemon performs the file transfer. Since this is a background operation, the transfer occurs at some later time after you invoke the *uucp* command. Therefore, if any process modifies *file1* after you invoke *uucp* but before the time of transfer, the modified version, not the original, is transferred to the remote system.

If you need to send a copy of a file in its current state to a remote system then continue to modify that file, use the -C option. For example:

```
uucp -C -m ~kls/file1 hpsys1!~uucp/file2
```

copies *file1* onto the spool directory where it is held until it is transferred to *file2* on the remote system.

Receiving Files From Remote Systems

The following examples show how to use the *uucp* command to request that a file on a remote system be copied to your local system.

```
uucp -m hpsys2!~ems/prog ~my_login/BIN/
```

requests the file *prog* from the login directory for **ems** on the remote system, **hpsys2**, be copied into a file of the same name in the *BIN* directory in directory *my_login* on the local system. The **-m** option requests mail acknowledging that the copy is complete.

```
uucp hpsys2!*.[ab] my_login
```

fetches all files ending in **a** or **b** in the login directory on the remote system, **hpsys2** and places them in the subdirectory, *my_login* on your local system.

All files in the **current** directory can be sent with a * character or a subset of these files can be sent with `*.[qualifiers]`. * cannot be used to represent entire pathnames.

Forwarding through Several Systems

To perform file forwarding on your system, special permissions must be set up in the following files:

`L.sys` — Contains information that determines how *uucp* will automatically reach other systems, and whether remote systems will be able to login to your system.

`FWDFILE` — A subset of `L.sys` that provides a list of systems through which your system may forward files.

`ORIGFILE` — Contains a list of originating system nodes that are allowed to forward files through your system. For example, if system nodes B, C, and D are part of your *uucp* network and they have included your system node A in their `ORIGFILE` then you can send a copy of **file_name** to system node D by typing,

```
uucp B!C!D!file_name [RETURN]
```

The `L.sys` file should be set up as explained in the "Uucp File Structure" chapter. The subset of `L.sys` called `FWDFILE` should be set up with a list of the system nodes that provide forwarding access to your system. The `ORIGFILE` should be set up with a list of system nodes that have forwarding access through your system. System node names are used to identify a given system within a network of systems using *uucp*. Consider the following network and table showing how permissions could be set up within these files for the network shown:

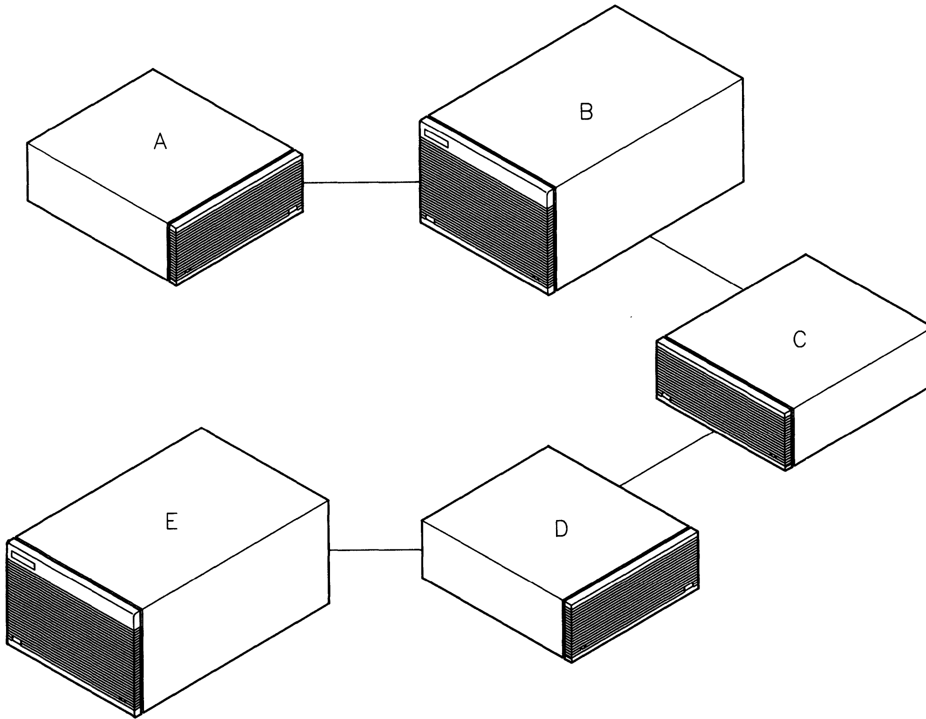


Figure 6-1. Network of Systems Using Uucp

The letter characters used in the following table represent system node names.

Table 6-1.

File Names	B	C	D	E
L.sys	A, C	B, D	C, E	D
FWDFILE	C	D		
ORIGFILE		B	C	

As indicated in the column for system **D**, systems **C** and **E** can use *uucp* to communicate with **D** because both systems have been included in the *L.sys* file on **D**. **D** cannot forward files through other systems because it has no system node names in its **FWDFILE** file. However, **C** can forward files through **D** because **C** is included in **D**'s **ORIGFILE** file.

Here is a typical example of sending a message (file) through a series of remote nodes:

```
uucp message node1!node2!node3!/usr/spool/uucppublic/filename
```

Anyone desiring to send you a **message** (file) from a remote system would use the same format as shown above, where:

message — is the file being sent

node1 — is the name of the first node (nearest the originating system node) in the forwarding chain, and **node2**, **node3**, . . ., etc. are the remaining nodes that are to forward the message until it reaches its destination. Node names used must specify the exact transmission path from the first system following the originating system through the final destination system. Each node name is followed by an exclamation point (**!**).

/usr/spool/uucppublic — is a directory open for writing by everyone. This directory is the depository for messages from remote systems, the destination for transmitted messages, and the source directory from which received files are retrieved by the destination user.

file_name — is the name given to the file that is being sent to a remote system when it is placed in the destination directory */usr/spool/uucppublic.*

Uucp Command Errors

If an error occurs in the *uucp* command transfer, a message identifying the problem is generated on the standard output device, normally your CRT.

"1" is the only error number ever generated. This indicates one of the following conditions:

- You have no right to access this file
- The file does not exist
- The file cannot be copied
- The system name given is incorrect.

Using the uux Command

The *uux* command gathers zero or more data files from various systems, executes a command on a specified system, then sends the standard output file for that execution to a file on the system on which the command was executed.

General uux Syntax

The general syntax for using the *uux* command is:

`uux [options]` *command_string*

where options can be:

`-`	uses standard input to get the data for the command
`-z`	requesting that the remote system be notified by mail only if the command execution failed
`-r`	creates the files necessary for the transfer to take place, but does not invoke *uucico* to call the remote system
`-n`	requesting no mail notification for the remote system.

The *command_string* must be enclosed in double quotes ("*command_string*") if an input or output diversion for the *command_string* is specified. If quotes are omitted, the shell tries to redirect the input/output of the *uux* command. For example:

```
uux "sys2!pr !ems/cmd/file1 > sys2!myoutput"
```

requests that the *pr* command be executed on remote system `sys2`. Your *ems/cmd/file1* on the local system (`sys1`) is printed by `sys2` to its *myoutput* file. At the time the *uux* facility is evaluating the command string, the *myoutput* file must also be accessible to the system **originating** the *uux* command (`sys1`).

Note two things illustrated in this example:

- All local files must be prefixed with "`!`"
- The system where output is redirected should be the same system on which the command is executed.

These files are created in the */usr/spool/uucp* directory:

`C.sys2AAxxx` is the workfile containing the lines:

```
S D.sys2Bxxxx D.sys2BxAxx ems - D.sys2BAxxx 666
S D.sys1XAxxx X.sys1XAxxx ems - D.sys1XAxxx 666
```

`D.sys1XAxxx` is an execution grade data file containing the lines:

```
U ems sys1
F D.sys2BAxxx file1
O myoutput sys2 O
C pr -n file1
```

`D.sys2BAxxx` contains a copy of the *file* to be printed.

Example

To compile a Pascal program on your local HP-UX system and have the results of that compilation directed to a file, a command string is used that resembles:

```
pc pas_file.p > pas_com
```

To execute this command on a remote system, command string merely includes the *uux* command and name(s) of the local or remote systems. For example:

```
uux "hpsys1!pc !/users/cmd/pas_file.p > hpsys1!~/pas_com"
```

requests that **hpsys1** execute the *pc* command on the Pascal program file *pas_file.p* on the local system, then place the results of the compilation in **hpsys1**'s public area in the file **pas_com**. Note that *system!file* where the output is redirected should be the same system that executed the command.

Before you can execute a command on any remote system, you must have permission from the remote system to the command. The list of all the commands a system permits to be executed by another system are contained in the **L.sys** file. You are notified by mail if the requested command on the remote system was not allowed.

The usual file naming conventions stated in the beginning of this chapter apply to the *uux* command file names with these exceptions:

- All local files must be prefixed with "!"
- Output files must have their parentheses escaped: `\(output_file\)`.

For example:

```
uux hpsys1!uucp hpsys2!/usr/file1 \(hpsys1!/usr/file2\)
```

sends a *uucp* command to `hpsys1` to copy *file1* from `hpsys2` to *file2* on `hpsys1`. Preceding the left and right parenthesis by a \ character tells the shell to interpret the parentheses literally. The parentheses tell the shell not to gather *file2* on `hpsys1` as a data file for the *uucp* command, but rather to use *file2* as the output file.

Uux Error Numbers

The *uux* command has several errors that are printed as error numbers rather than the usual error messages. These error numbers and their interpretations are:

`1` You have no right to access a file, the file does not exist, or you cannot copy the file

`2` The pathname cannot be properly expanded

`101` You specified an invalid system name

`102` The size of the parameters given to *uux* exceeds the maximum length specified in the `BUFSIZ` variable.

Miscellaneous Commands

This section describes use of the commands: *uuclean*, *uulog*, *uuname*, *uupick*, *uustat*, *uusub*, and *uuto*. They are presented and discussed in alphabetical order (for other related commands, see the *HP-UX Reference*, `uucp`(1)).

Using uuclean

The *uuclean* command scans a directory for files with a specified prefix and deletes those that are older than the specified number of hours. If you have a backlog of jobs that cannot be transmitted to other systems, they should be cleaned up so that the file space can be reused. You can also have the *uuclean* command remove lock and status files that are no longer needed.

These cleanup activities can be routinely executed by shell scripts started by the *cron* program. Refer to the chapter, "Log, Status and Cleanup" for a detailed discussion.

General syntax for *uuclean* is:

```
uuclean [options]
```

where options can be:

−d*directory* is an alternate directory to scan for files instead of the default spool directory

−p*pre* is the file prefix (up to ten prefixes may be specified. If no **−p** option is specified, **all** files older than the **−n*time*** hours are deleted)

−n*time* is the time in hours where files older than *time* are deleted (the default is 72 hours)

−m sends mail to the owner of a file when that file is deleted.

Here is an example use of *uuclean*:

```
/usr/lib/uuclean -pLOG -pLCK -n24
```

This removes all files starting with `LOG`, such as `LOGFILE`s and all files starting with `LCK`, such as `LCKFILE`s that are more than 24 hours old.

The "Log, Status and Cleanup" chapter contains examples of shell scripts that use *uudemons* to implement *uuclean* commands.

Using uulog

The *uulog* command displays a summary log of *uucp* and *uux* transactions. If you use the *uulog* command without any options, the information in the temporary log files (LOG.*) is appended to the main LOGFILE. These LOG.* files are created only if LOGFILE is locked when the *uucp* facility attempts to make an entry. *Uulog* then gathers information into the LOGFILE in directory */usr/spool/uucp*, then prints it.

General syntax for *uulog* is:

```
uulog [-ssys_name] [-uuser_name]
```

where:

-ssys_name	prints information about work involving system **sys_name**
-uuser_name	prints information about work done for the specified **user_name**.

Uulog then displays log information with each printed line showing user, system, (date, time, PID_number), status, and action.

For example, if you type:

```
uulog -smit
```

a typical display for the system mit would resemble:

```
john mit (2/14-10:11-15486) SUCCESSFUL (AUTODIAL)
john mit (2/14-10:11-15486) SUCCEEDED (call to mit)
john mit (2/14-10:11-15486) OK (startup)
john mit (2/14-10:11-15486) REQUEST (S D.mitn2236 D.mitn2236 john)
john mit (2/14-10:12-15486) OK (conversation complete)
```

Invoking *uulog* without any parameters appends all temporary log files (*LOG.**) to the main *LOGFILE.*

Refer to the appendix for an alphabetical listing and interpretation of LOGFILE messages.

Using uuname

The *uuname* command returns the *uucp* name of your local system or the nodenames of remote systems known to your local system.

The general syntax for *uuname* is:

```
uuname [-l] [-v]
```

where:

-l returns your local system name.

For example:

```
uuname -l
```

returns:

```
My_node_name
```

and:

```
uuname
```

returns a list similar to:

```
mit
csu
hp-sys1
UCLA
ISU
```

-v returns a description for each system listed in the ADMIN file.

These are the names of the systems you can communicate with.

Using uupick

This command should be used with the *uuto* command.

You can use *uupick* command to accept or reject files transmitted to you with the *uuto* command on another system. *Uupick* searches */usr/spool/uucppublic* for files sent to your local system by *uuto.* For each file or directory found, *uupick* prints a message about the designated file on the standard output file. *Uupick* then waits for an answer indicating what you want to do with that file, reading the answer line from the standard input file. The cycle is repeated until *uupick* finds no more *uuto* files destined for you.

General syntax for uupick is:

```
uupick [-ssys_name]
```

where:

`-ssys_name`   searches `/usr/spool/uucppublic` for files sent only from `sys_name`.

When *uupick* is reading from the standard input file to determine the disposition of a file, the following translations are used:

Table 6-2.

Input Command	Interpretation
<carriage return>	Go on to next entry
d	Delete the entry
m<dir>	Move the file to the named directory <dir> (default is the current directory)[1]
a<dir>	Move all files sent from <sys_name> to the named directory <dir> (default is the current directory)
p	Print the contents of the file
q	Quit (stop)
EOT (CTRL-D)	Quit (stop)
!*command*	Escape to the shell to do **command**
*	Print a command summary

[1] Do not use "." to represent the current directory with the **m** or the **a** parameter. Use the default instead.

For example, if you type:

```
uupick
```

and **file_name** has been sent from **sys1**, this is printed on the standard output file:

```
from system sys1: file file_name
```

You could then continue with any of the options listed in the table above.

Using uustat

The uustat command initiates status inquiry for all jobs requested and for job control.

The general syntax for the *uustat* command is:

```
uustat [options]
```

where the available options are:

-chour	Remove status entries that are older than **hour** hours old (only the user initiating the *uucp* command or the super-user can invoke this option)
-jall	Report the status of all the *uucp* requests
-kjob_num	Kill (terminate) the uucp request whose job number is **job_num** (the terminated *uucp* request must belong to the person issuing the uustat command or the super-user. The **job_num** is supplied automatically by the *uucp* facility.)
-mmachine	Report the status of accessibility of **machine**. If **machine** is **all**, then the status of all machines known to the local *uucp* are displayed.
-ohour	Report the status of all *uucp* requests that are older than **hour** hours old
-yhour	Report the status of all uucp requests that are younger than **hour** hours old
-ssys	Report the status of all *uucp* requests involving system **sys**
-uuser	Report the status of all uucp requests issued by **user**
-v	Report the *uucp* status verbosely (this option is recommended because without it only the status code for each request is printed).

Using *uustat* without any options prints all job information in the non-verbose mode for the user you are logged in as.

If you type:

```
uustat -v -jall
```

here is an example of a typical display:

```
0923     rmd      hpfcla     04/25-11:00      04/25-13:01
REMOTE ACCESS TO FILE DENIED
COPY FINISHED, JOB DELETED
```

where:

`0923`	is the *uucp* job number
`rmd`	is the user issuing the uucp command
`hpfcla`	is the remote system
`04/25-11:00`	is when the uucp command was first issued
`04/25-13:01`	is the last status update
`REMOTE ACCESS TO FILE DENIED`	is part of the status report
`FINISHED, JOB DELETED`	is the remainder of the status report.

The status report indicates that the workfile was deleted without any data file being copied.

Using uusub

The *uusub* command defines the *uucp* subnetwork and monitors the connection and traffic among its members. This command is normally used by super-user or the system administrator.

General syntax for using *uusub* command is:

```
uusub [options]
```

where **options** could be:

-asys	add **sys** to the subnetwork (only one system can be added at a time)
-csys	exercise the connection to system **sys** by making a call to that system (**sys** may be **all** for all systems in the subnetwork)
-dsys	delete **sys** from the subnetwork
-f	flush (erase) the connection statistics
-l	report the statistics on connections
-r	report the statistics on traffic amount
-uhour	gather the traffic statistics over the past **hour** hours.

which when executed displays:

```
sys #call #ok time #dev #login #nack #other
```

where:

sys	is the remote system name
#call	is the number of times your local system tried to call **sys** since the last flush
#ok	is the number of successful connections
time	is the latest successful connect **time**
#dev	is the number of unsuccessful connections because of no available device
#login	is the number of unsuccessful connections because of login failure
#nack	is the number of unsuccessful connections because of no response
#other	is the number of unsuccessful connections because of other reasons.

You can define your subnetwork with the **-a** option, for example:

```
uusub -ahpdcd
uusub -ahprvd
uusub -ahpcnob
uusub -ahpfcld
```

You can then monitor this defined subnetwork by typing:

```
uusub -l
```

which results in an output resembling:

sysname	#call	#ok	latest-oktime	#noacu	#login	#nack	#other
hpdcd	26	9	(4/25-23:58)	0	0	17	0
hprvd	0	0	(4/21-15:30)	0	0	0	0
hpcnob	25	24	(4/25-23:57)	0	1	0	0
hpfcld	5	2	(4/25-23:56)	0	0	14	0

The meanings of the traffic statistics gathered with the -r option are:

```
sfile   sbyte   rfile   rbyte
```

where:

sfile is the number of of files sent

sbyte is the number of bytes sent over the period of time indicated in the latest *uusub* command with the **-uhour** option

rfile is the number of files received

rbyte is the number of bytes received.

The traffic statistics over the last two hours can be gathered by typing:

```
uusub -u2
```

The traffic statistics must be gathered before they can be reported with the **-r** option. For example, if you now type:

```
uusub -r
```

your output could be:

```
sysname      sfile       sbyte       rfile       rbyte
hpdcd          2          699          0           0
hprvd          0            0          4         584
hpfcla        66       266639         34      140784
```

Using uuto

The *uuto* command uses the *uucp* facility to send files to a specified destination. You can use the *uupick* command to "pick" disposition of the files sent by *uuto*.

General syntax for *uuto* is:

```
uuto [options] source_file destination
```

where:

options can be either:

- **-p** copy the `source_file` into the spool directory before transmission, or
- **-m** generate mail to the sender when the copy is complete

source_file is the source file(s)

destination is of the form: `sysname!user` where `sysname` is the name of the remote system and `user` is the user on the remote system you are sending the files to.

The source_file(s) are sent to */usr/spool/uucppublic* on `sysname`.

If you type:

```
uuto -p -m /users/rmd/file hpsys2!mark
```

this *uucp* command is generated:

```
uucp -d -C -m -nmark /users/rmd/file
          ~/receive/mark/hpsys1/
```

where the destination user is `mark` and `hpsys1` is the system **from** which the file is transferred.

Using the Mail Facility

You can use the *mail* command to send mail messages to other systems. For example, if you type:

```
mail remote_sys!name
Meet me in the lunch room.
Don't be late again!
[CTRL]-[D]
```

lines two and three are mailed to *name* on **remote_sys**.

When you specify remote systems with *mailx*, *uucp* uses the *uux* command sequence. A workfile with an **X** grade and one containing the actual mail message, are set up in directory **/usr/spool/uucp**.

You do not have to specify the entire pathname to the user receiving your mail message.

You can also forward mail through intermediate nodes.

```
mail remsys1!remsys2!remsys3!name
(Message typed in here.)
[CTRL]-[D]
```

Your mail is then forwarded through **remsys1** and **remsys2** before it reaches its final destination on **remsys3**.

You can mail entire ASCII files to a user on a remote system by using:

```
mail remote_sys!name <filename
```

Notes

The X.25 Network 7

This chapter provides a brief description of the X.25 Network and explains how to configure *uucp* software to work with X.25.

Description of X.25

X.25 is a worldwide standard protocol used in many Public Data Networks (PDNs). Public Data Networks are packet switching networks (PSN's).

Packet Switching Network

Before learning what a Packet Switching Network (PSN) is, you need to know what an X.25 packet is. X.25 packets are defined as serially transmittable strings of information containing the following fields:

ADDRESS	identifying the packet destination
DATA	data being transmitted, usually not more than 256 bytes
CHECKSUM	error detection information
SEQUENCE	ensures that data packets are handled in correct sequence
OTHER	other fields not discussed in this manual.

A Packet Switching Network consists of many nodes (also called stations) where each node has the ability to correctly interpret routing information contained in each packet and forward the packet to the next appropriate node in the network. The computer that is originating the packet combines routing and error checking information with the data being transmitted, then sends the packet to the nearest switching node where it is forwarded to its destination.

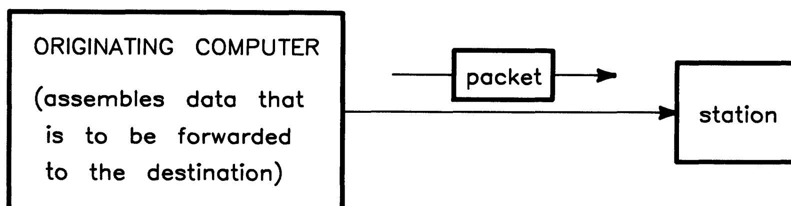


Figure 7-1.

Each switching node, in turn, examines address information in each packet as it is received, and determines automatically where to send it. The process is repeatedly continued until each packet reaches its correct destination.

Consider the following hypothetical stations and interconnections:

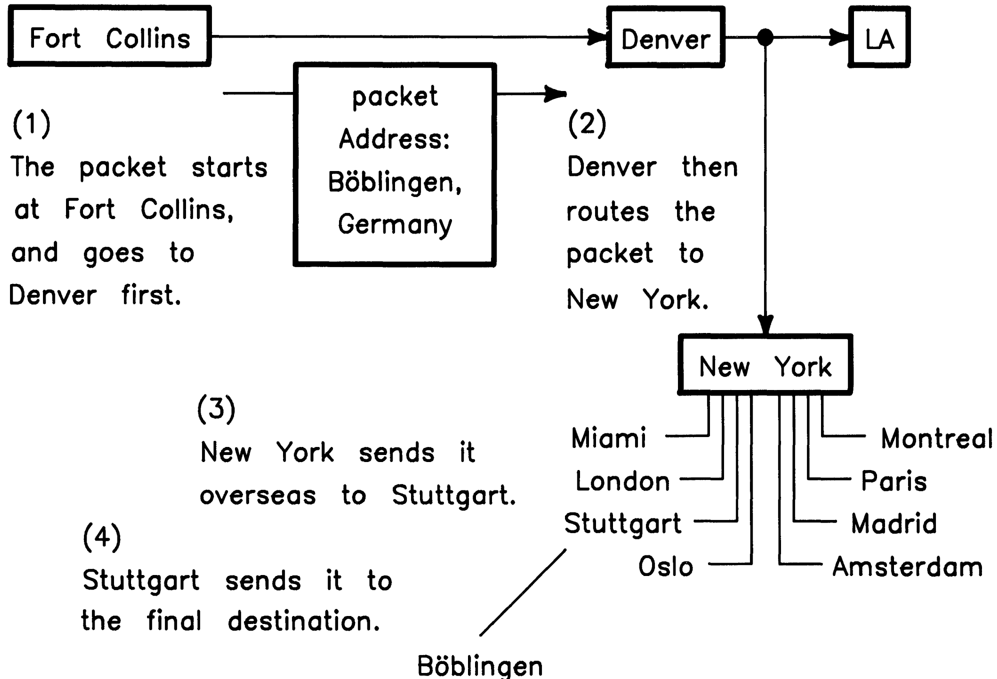


Figure 7-2.

In this example, the packet is sent from Fort Collins, Colorado to Denver. The Denver node determines that the packet should be forwarded to New York for transatlantic transmission. New York forwards the packet to Stuttgart, Germany, where it is forwarded to its final destination, Boeblingen.

When the receiving computer in Boeblingen accepts the packet, it disassembles the packet by first verifying that the address is correct, then verifying the checksum against the data to ensure that no data was lost enroute (if an error is detected, retransmission is requested). The sequence number is checked to make sure the packet did not arrive before a packet which preceded it. (Packet switching networks use many data transmission lines simultaneously, so it is not uncommon for packets to be received in incorrect sequence. If the sequence is incorrect, the receiving interface must hold the message until preceding

messages in the sequence arrive before passing the packet data to the computer.) If the data is in correct sequence and contains no errors, it is passed to the computer for use.

Public Data Network

A Public Data Network (PDN) is a packet switching network that each country has established to handle data traffic. Most countries have only one PDN, while a few have several. Public Data Networks are connected to each other by "gateways" which are really nothing more than packet switching connections between two switching nodes. For example,

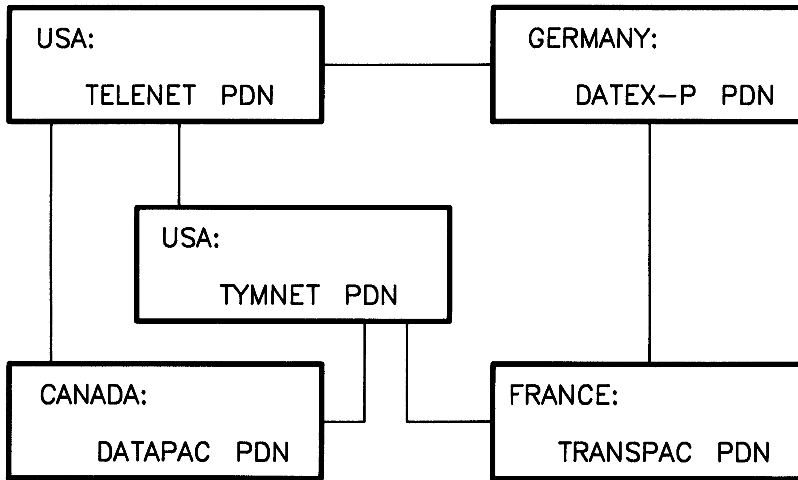


Figure 7-3. Public Data Network

Most nations have their own Public Data Network with gateways to PDNs in other countries. This worldwide interconnection of PDNs is known collectively as "The X.25 Network".

Configuring uucp for X.25

This section discusses configuring *uucp* for use with the X.25 Network. These topics are as follows:

- Prerequisites
- Installing the HP 2334A
- Remote and Local Off-line Configuration
- Preparing for Configuration
- Configuration Procedure.

The *HP 2334A MULTIMUX Reference and Service Manual* (HP part number: 02334-90001) covers the above topics in detail. If you are using the HP 2334A to connect to the X.25 Network, you need to read the *HP 2334A MULTIMUX Reference and Service Manual* after reading this manual.

The acronym "PAD", which stands for **P**acket **A**ssembler/**D**isassembler, needs to be defined before proceeding with this section of the chapter. A Packet Assembler/Disassembler is a device which takes a message to be transmitted and **assembles** it into transmittable X.25 data packets. Conversely, it receives X.25 packets and **disassembles** them into character streams for transmission to a terminal.

The HP 2334A is the device which provides for the interfacing of your HP-UX system to the X.25 Network. This device must be configured with the proper synchronous X.25 network parameters to enable communication across the synchronous network and the asynchronous protocol (by assigning PAD parameter values) for communication with connected asynchronous devices (or computer ports). These communication parameters must initially be defined off-line (i.e. when not communicating with the synchronous network or devices). They are saved automatically in permanent memory so that the HP 2334A does not require the configuration process each time the power is turned on.

The off-line configuration (or reconfiguration) of the HP 2334A is normally performed using a terminal which is directly connected to the HP 2334A device port A1. Alternatively the off-line configuration (or recofiguration) may be performed from a remote location by connecting the remote terminal to the HP 2334A device port A1 via a pair of asynchronous modems and a telephone line. Both of these off-line methods are covered in this section of the chapter.

Prerequisites

Before you can start configuration of X.25 you need to make sure the following prerequisites have been met.

- The *uucp* facility must already be working on your system
- You must have an understanding of your *uucp* file system
- You should know how to use the *uucp* commands.

Information for these prerequisites is covered in the chapters prior to this one. If you haven't read these chapters please do so and return to this chapter when you are done. You also need to use the unpacking list sent with your HP 2334A to verify that you have the necessary hardware to begin the installation of your HP 2334A device.

Installing the HP 2334A

This section covers the necessary steps for installing the HP 2334A. Line voltages, power supply settings and mounting the HP 2334A are covered in the manual, *HP 2334A Cluster Control Reference and Service Manual.* The topics included here are as follows:

- Power-on and CPU Switch Test
- Connecting Cables to an HP 2334A.

Power-on and CPU Switch Test

The HP 2334A has a power-on self test which is performed automatically at power-on and a CPU switch test which is perform manually by the user of the HP 2334A. The first test is covered in this manual, as the second one is not necessary for getting started on your HP 2334A. The test are as follows:

- An internal "power-on self test" is automatically performed whenever the HP 2334A is switched ON (1), or initialized using the reset button. This test is performed regardless of whether the HP 2334A is off-line or on-line. To observe this test, you need to remove the front panel of the HP 2334A. You can remove the front panel by placing your fingers under the lip on the top part of the front panel and pulling outward. Turn the unit on and observe the LEDs on the CPU card. The LEDs will blink off and on as the self test is being executed. For a description of this test, read the section in this chapter entitled, "Entering the CONFIGURE Mode".
- An off-line CPU Card Switch Test checks the operation of the CPU cards DIP switches. For an explanation of this test, read the section in the, "INSTALLATION" chapter of the, *HP 2334A MULTIMUX Reference and Service Manual* entitled, "CPU CARD SWITCHES TEST".

With the front panel off you need to set switch 8 of the CPU card's DIP switches to the upward position and all other switches should be set to the downward position. The front panel should be left off after you have completed this section.

Connecting Cables to an HP 2334A

The cable connections made in this section are for "local off-line configuration". Local off-line configuration is where you have a terminal directly connected (not through a modem) to the A1 port of your HP 2334A and the HP 2334A connected to the the X.25 Network. The next section in this chapter explains "remote" and "local" off-line configuration.

Before connecting any cables, you need to have the following cards insert in the HP 2334A:

- Synchronous Network Adapter Card
- Modem Control Adapter Card.

The Synchronous Network Adapter Card has already been installed in your unit and is located behind the front panel as shown in the diagram below.

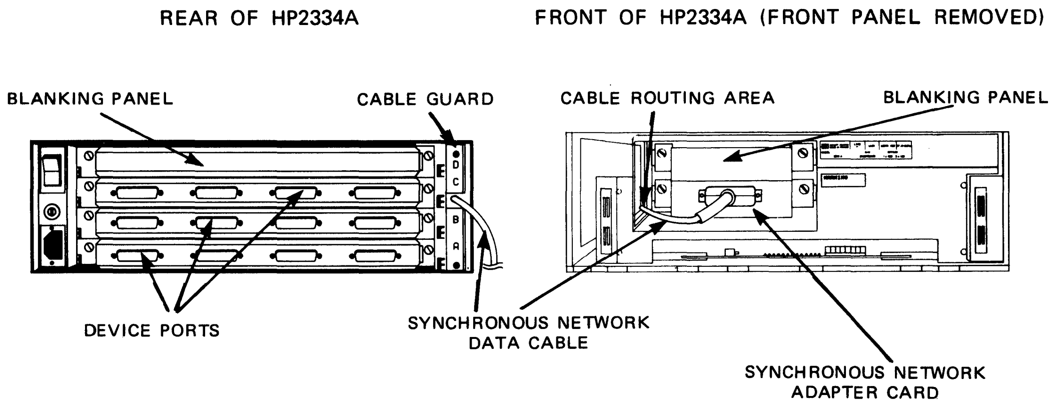


Figure 7-4. HP 2334A to Device and Synchronous Network Connection

The Modem Control Adapter Card (HP40261A) is installed in one of the Device Adapter Card slots located in the backplane of the HP 2334A as shown in the diagram below. The first adapter card is inserted in slot A and the next one is inserted into slot B and so on. Also, initial adapter cards come installed.

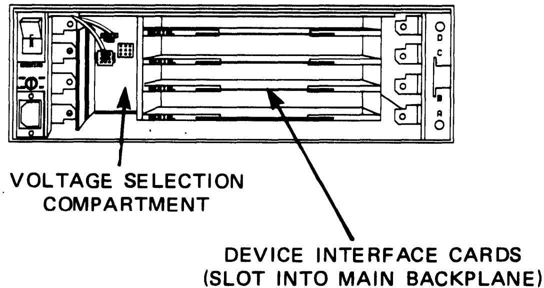


Figure 7-5. Adapter Card Slots

For local off-line configuration, you need to have an HP 40261A card inserted in slot A.

Connect an interactive terminal to port A1. Port A1 is located on the Modem Control Adapter Card which is in slot A of the backplane. The port is labeled number 1 on the Modem Control Adapter Card. The terminal is connect to port A1 by an RS-232C cable with a DTE connector.

The Synchronous Network Cable (HP part number: 02333-60008) is connected to the Synchronous Network Adapter Card using the following procedure:

1. Insure that the HP 2334A is switched OFF (0), then disconnect the power cord
2. Remove the HP 2334A front panel
3. On the right-hand side of the backplane (see Figure 7-4) remove the cable guard.

NOTE

Do not remove any of the Device Adapter Cards or blanking panels.

---

4. Pass the connector of the Synchronous Network Cable network into the data cable routing area, as shown in the diagram mentioned in step 3. This cable (part number: 02333-60008) is supplied with the HP 2334A.
5. From the front of the HP 2334A, carefully pull the cable through the data cable routing area and then plug the data cable connector into the Synchronous Network Adapter Card connector (see Figure 7-4) and secure it in position by tightening the two locking screws.
6. Insure that the cable is a loose fit in the routing area. Then replace the front panel.
7. At the rear of the HP 2334A, place the slot in the cable guard over the data cable. Then insert a plastic cable tie (i.e. tie wrap) through the two holes in the cable guard slot and secure the cable guard (this acts as a cable clamp). Replace the cable guard on the HP 2334A backplane and tighten the two cross-head screws to secure it.
8. Replace the power cord and switch ON the HP 2334A as required.

Remote and Local Off-line Configuration

Off-line configuration may be performed remotely by connecting a terminal to port A1 of the HP 2334A via a telephone line and two full duplex, asynchronous modems (at 1200 baud) as shown in the Figure 7-6 below. An operator is required at the HP 2334A location to perform certain simple actions:

- DIP switch setting
- Power-on/reset
- Reading the LEDs.

A second telephone line is necessary for conveying verbal instructions.

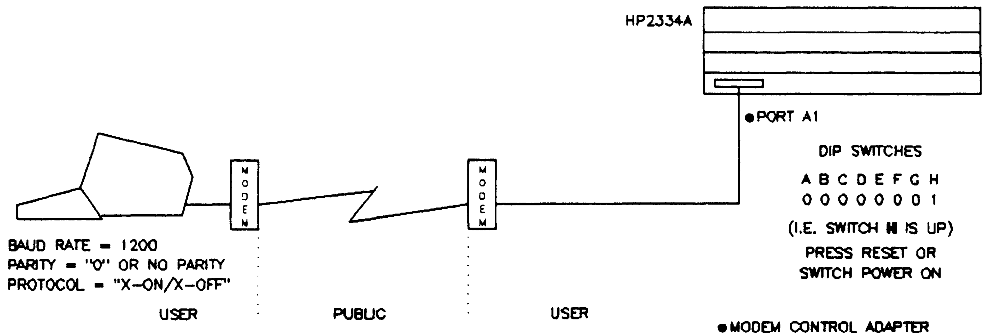


Figure 7-6. Remote Off-line Configuration

Remote off-line configuration is an important feature as it enables, for example, an experienced operator to perform off-line configuration of HP 2334As located at multiple remote sites (branch offices) without leaving the office.

The Modem Control Adapter Card (HP 40261A) provides modem interface ports A1 to A4 as remote configuration requires the use of asynchronous modems between the terminal and device port A1. Once communication is established between the terminal and the HP 2334A, the preparation and configuration procedure is the same as when configuring the HP 2334A locally (see the diagram below).

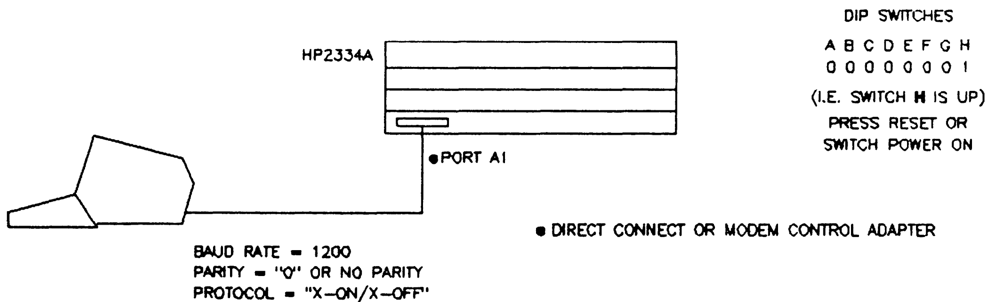


Figure 7-7. Local Off-line Configuration

The procedure for configuring the HP 2334A is explained in the remaining sections of this chapter.

Preparing for Configuration

The HP 2334A should be prepared for off-line configuration as follows (refer to the diagrams in the previous sections for connections and switch settings):

1. Connect an interactive terminal to device port A1. All other asynchronous devices may remain connected as required. The synchronous network may remain connected as required.
2. Set the terminal as follows:
   a. **BAUD RATE** — 1200 baud
   b. **DATA BITS** — 7
   c. **PARITY** — set for "0" or no parity
   d. **X-ON/X-OFF Handshake** — **ENABLED**
   e. **FULL DUPLEX**
   f. **REMOTE** mode
   g. **CHARACTER** mode (**BLOCK** mode **OFF**).
3. Make sure the HP 2334A is set to **CONFIGURE** mode by checking the **CPU** cards switch settings. The **CPU** cards **DIP** switches should be set as follows:

```
DIP Switch: A  B  C  D  E  F  G  H
   Setting: X  X  X  X  0  0  0  1  Where: 0 = DOWN / OFF
                                           1 = UP / ON
                                           X = ON or OFF
```

Switches A, B, C, and D may be set as required.

NOTE

When the HP 2334A is in CONFIGURE MODE **only** port A1 is enabled and it is pre configured at 1200 baud.

Configuration Procedure

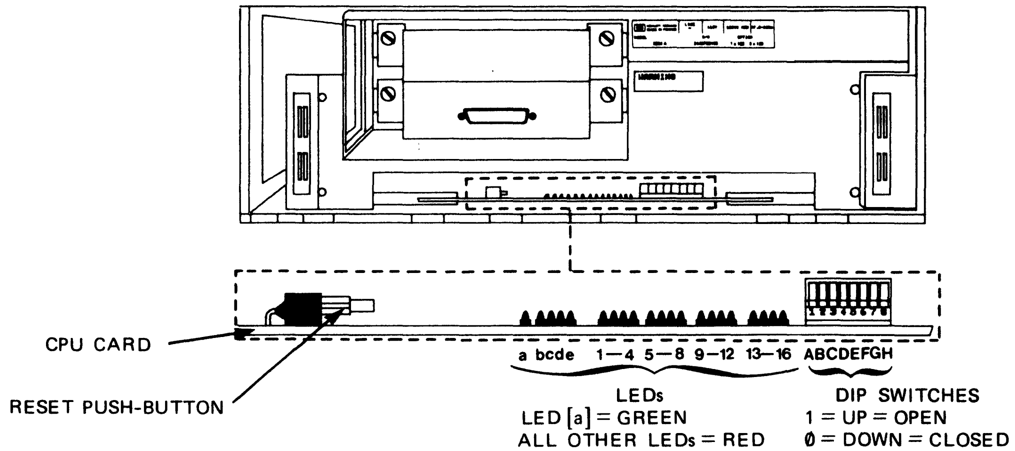
This section covers the following topics:

- Entering the CONFIGURE Mode
- Sample Off-line Configuration Listing
- Entering the HSA Command Mode
- Step-by-step Configuration Procedure for HSA
- Entering the UDP Command Mode
- Entering the ASG Command Mode
- Adding New Entries to the uucp Files
- Creating New Configuration Files.

Entering the CONFIGURE Mode

To enter **CONFIGURE** mode, simply press the **CPU** card's reset button (or if the HP 2334A is switched OFF, switch it ON). The front panel must be removed to get to the reset button. To remove the front panel, place your fingers in the channel on its upper edge and pull outward. The inside of the HP 2334A looks like the diagram shown below. If you use the ON/OFF switch, it is located on the backside of the HP 2334A.

HP2334A WITH FRONT PANEL REMOVED



NOTE:
1) SWITCHES A, B, C AND D ARE SENSED:
   A) CONTINUOUSLY DURING THE RUN MODE, CONFIGURE MODE, SWITCHES TEST, DEVICE LOOP BACK TEST AND MODEM LOOP BACK TEST.
   B) DURING THE SELF DIAGNOSTIC TESTS, ONLY AT POWER-ON OR AFTER RESET.
2) SWITCHES, E, F, G AND H ARE ONLY SENSED AT POWER-ON OR AFTER RESET.

Figure 7-8. CPU Card's Reset Button and LEDs

Once the above process has been executed the HP 2334A goes through a power-on self test that last approximately twenty seconds and causes the following to happen:

1. All of the 21 LEDs shown in the diagram above turn ON (illuminate) for one second, then they turn OFF (extinguish) for one second
2. Then the LEDs are individually illuminated, starting at LED **a** and going through to LED **16**.

3. The self diagnostic tests are then performed with LEDs **d** or **e** ON (illuminated) to indicate which test routine is active (LEDs **a** and **1** to **16** are OFF). See the diagram below.

```
      LED DISPLAY                TEST
[a]  [b]  [c]  [d]  [e]
 0    0    0    0    1   - CPU Card Test (10 sec. approx.)
 0    0    0    1    0   - Internal Bus and Device Interface
                           Cards Test (3 sec. approx.)

                           Where: 0 = LED OFF
                                  1 = LED ON
```

   The CPU Card Test is performed first and, if successful, the Internal Bus and Device Interface Cards are then tested.

If your CPU Card Test is **successful**, you may continue with the next section. However, if you had a **failure** or an error was detected, the HP 2334A halts and provides a failure indication as follows:

- LED **a** remains **off** (extinguished)
- LED **d** or **e** remains **on** (illuminated) indicating the failed test routine
- LEDs **1** to **16** provide a display indicating the type of failure
- LED **b on** indicates an invalid DIP switch setting.

If the HP 2334A halts due to a failure refer to the section, "User Troubleshooting" in the chapter, "OPERATION" or the chapter, "TROUBLESHOOTING", in your *HP 2334A MULTIMUX Reference and Service Manual.* If the failure cannot be corrected then contact the nearest HP Sales and Service Office.

Sample Off-line Configuration Listing

The HP 2334A automatically provides the initial off-line configuration listing as shown in this section; however, the listing shown in this section has been filled in with the correct field values for Levels I through III. The remaining fields, with the exception of the "HP 2334 CONFIGURATION X.25 SRA" field, can be filled in after you have read the *HP 2334A MULTIMUX Reference and Service Manual* and you are familiar with the HP 2334A.

The off-line configuration listing is divided into three groups they are as follows:

- The HP 2334A-to-Synchronous Network (X.25) configuration. This includes all the field entries made using the *HSA* command.
- The HP 2334A-to-Device (X.3 parameters) configuration. This configuration is for assigning ports to the various devices which are to be connected to the HP 2334A. The *ASG* command is used to assign PAD or CAS/PAD profiles to the HP 2334A asynchronous ports. Explanations for PAD and CAS/PAD can be found in the chapter, "CONFIGURATION" in the, *HP 2334A MULTIMUX Reference and Service Manual.*
- The hardware installed in the HP 2334A. This lists the ROMs on the CPU card and is followed by a list of the device "interface" and "adapter" card information. No data is displayed if a "device adapter card" is not inserted in the associated slot A, B, C, or D. The "device adapter cards" used with the HP 2334A are identified as follows:

  - Direct Connect Adapter Card (HP 40260A)
  - Modem Control Adapter Card (HP 40261A)

  The Direct Connect Adapter Card is not implemented for use with your HP 2334A.

The following listing is a sample of what your display would show had you already configured it to the values given. This listing assumes that you are in the United States of America and using TELENET.

```
HSA   02334-80320 . 02334-80330

 HP2334 CONFIGURATION X.25 LEVEL I

       ------------------------------------------------------------------
       -PHYS. LINK   : X.21bis DTE                 -LINE SPEED    : 9600
       ------------------------------------------------------------------

 HP2334 CONFIGURATION X.25 LEVEL II

       ------------------------------------------------------------------
       -NETWORK TYPE:  TEL, 12                     -EQUIP. TYPE   : LAP-B DTE
       -FRAME WINDOW:  7                           -TIMER  T1     : 3000  ms
       -RET. CNT N2 :  20                          -I-FRAME       : 131 bytes
       ------------------------------------------------------------------

 HP2334 CONFIGURATION X.25 LEVEL III

       ------------------------------------------------------------------
       -LOCAL ADDRESS : nnnnnnnnnnnnnss
       -WIND. SIZE IN : 2                          -WIND. SIZE OUT: 2
       -THROUGHPUT IN : 9600                       -THROUGHPUT OUT: 9600
       -PACK. SIZE IN : 128                        -PACK. SIZE OUT: 128
       -FIRST PVC     :                            -LAST PVC      :
       -FIRST SVC IN  :                            -LAST SVC IN   :
       -FIRST 2W SVC  : 1                          -LAST 2W SVC   : 64
       -FIRST SVC OUT :                            -LAST SVC OUT  :
       -FIRST POOL PRT: B4                         -LAST POOL PRT : B4
       -D-BIT         : NO
       -PKT. NUMBERING: 8
       -FAC. SUPPORTED:
       -PVC ASSOC. PRT:
       ------------------------------------------------------------------

 HP2334 CONFIGURATION X.25 LUG

       ------------------------------------------------------------------
       -REMOTE ADDRESS
       ------------------------------------------------------------------

 HP2334 CONFIGURATION X.25 SRA
```

```
        ------------------------------------------------------------------
        -REMOTE ADDRESS
        ------------------------------------------------------------------
  ASG
     Assignment for each port

     .    1    2    3    4
     D    2    2    2    2
     C    2    2    2    2
     B    2    2    2    2
     A    2    2    2    2
        -----------------------------------------------------------------

 CPU    02334-80300 . 02334-80310 .
 SC-D               .             .
 SC-C               .             .
 SC-B               .             .
 SC-A   05180-2039  . 05180-2040  . RS232MOD4 ports
```

The following is an explanation of the various sections of the configuration listing:

- The HSA refers to the Synchronous Network Adapter Card which fits into slot "A" of the Network Adapter card cage. This card cage is located behind the front panel of the HP 2334A as seen in the picture below. The two part numbers adjacent to HSA refer to two ROMs located on the CPU card.

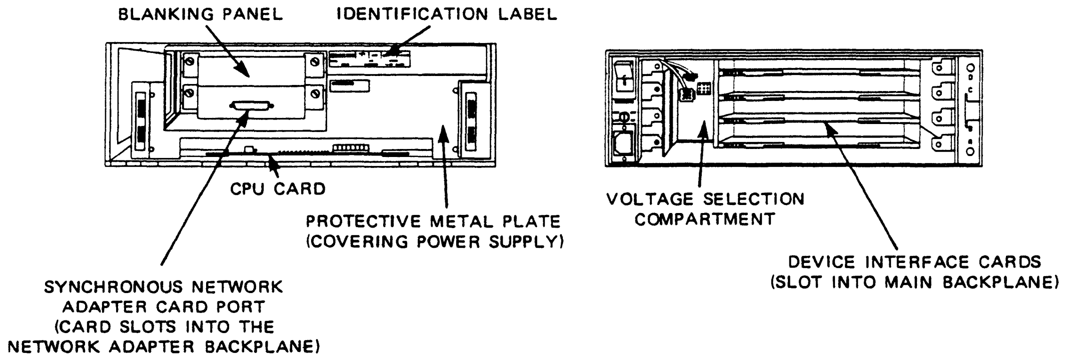


Figure 7-9. Front and Rear View of the HP 2334A

- Parameters associated with the various fields in Levels 1 through 3 have recommended values assigned to them. These values should be used to configure your HP 2334A for the first time.
- The two part numbers associated with the CPU toward the bottom of the configuration list refer to two more ROMs located on the CPU card
- The parameters `SC-D`, `SC-C`, `SC-B`, and `SC-A` refer to the Device Adapter Cards inserted in the slots D, C, B, and A respectively. These slots are located on the backplane of the HP 2334A. The values associated with these fields are:
  - The part numbers of the ROMs located on the Device Interface Cards
  - The type of card inserted into the backplane of the HP 2334A which is the HP 40261A card represented by this value: `RS232MOD 4 ports`.

The `nnnnnnnnnnnnnss` values located after various fields in the listing are local network addresses and remote addresses which may be up to 15 digits long the last two digits (`ss`) being the sub-address of the HP 2334A device ports. For a detailed explanation of this, read the sections, "X.25 Level 3" and "MSG, LUG and SRA CONFIGURATION" found in your *HP 2334A MULTIMUX Reference and Service Manual.*

Entering the HSA Command Mode

The HP 2334A-to-synchronous network configuration may be defined using the *HSA* command and an HP-UX supported interactive terminal connected to device port A1. The *HSA* command allows the user to configure all the X.25 parameters (Levels 1, 2, and 3), Symbolic Remote Address (SRA) facilities, and Local User Address (LUG) facilities.

Once the configuration listing, which has not been filled in, has been displayed and the user asterisk (*) prompt is obtained, execute:

```
HSA
```

The *HSA* command you just executed refers to the Synchronous Network Adapter Card mounted in slot "A" of the Network Adapter cage. The following is next displayed as a user prompt:

```
HSA:
```

Step-by-step Configuration Procedure for HSA

This section provides you with a step-by-step procedure for entering the correct values in the fields of the previously shown configuration listing. Some of the fields in the configuration listing are preset and are not mentioned in the following procedure. For a detailed explanation of this procedure, you need to read the chapter entitled, "CONFIGURATION" in your *HP 2334A MULTIMUX Reference and Service Manual.*

All commands are entered on the *HSA* command line which is indicated by the **HSA**: prompt. The step-by-step procedure is as follows:

1. Specify the data transmission rate on the synchronous network connetion, execute:

   LEVEL1

   The following message is displayed:

   Line_speed?

   Respond to it by entering:

   9600

2. Enter the fields for Level 2 of the configuration, entering:

   LEVEL2

   The following message is displayed:

   `NTK_type?`

   If you are in the United States of America, you would respond to the above prompt by typing in the response given below; otherwise, refer to your *HP 2334 MULTI-MUX Reference and Service Manual* for the proper response.

   TEL,12 [RETURN]

   This message is displayed:

   `Frame_window?`

   Respond to it by typing:

   7 [RETURN]

   This message is displayed:

   `Timer_T1?`

   Respond to it by typing:

   3000 [RETURN]

3. Enter the values for Level 3 of the configuration, enter:

   LEVEL3

   The following message is displayed:

   `Lcl Addr.?`

   Respond to it by typing:

   nnnnnnnnnnnnnss

   where **nnnnnnnnnnnnn** is the local network address which may be be up to 13 digits long and **ss** is the 2 digit sub-address.

The remainder of this message and response sequence is given in a tabular form. To use this table correctly, you should start with the message and response at the top of the table and work your way to the bottom. You will be reading the message in the left column and responding with the prompt in the right column. Press the RETURN key after typing in your response. There are some cases where you are asked just to press RETURN without entering a response.

Message Displayed	Response
Thrput in?	9600
Thrput_out?	9600
Wind sz in?	2
Wind sz out?	2
Def/mod vc tbl.?	no
Fst pvc?	RETURN
Fst svc in?	RETURN
Fst 2w svc?	1
Lst 2w svc?	64
Fst svc out?	Press RETURN
First pool port?	A1
Last pool port?	A1
Neg pk sz?	no
Neg wd sz?	no
Neg thrput?	no
Rev. char. acc.?	no
D-Bit?	no

To verify the field entries you made to your configuration listing, type the following after the **HSA**: prompt:

```
list
```

If a field entry is wrong, you will have to re-enter the command after the **HSA**: prompt (e.g. level1) which gets you the section containing the field that needs to be changed. Note that there is no way to step to the field you wish to correct. You must re-enter the correct values to all of the fields as the prompts for them appear.

To exit the HSA mode, press: RETURN

Entering the UDP Command Mode

The UDP primary user command is used to create or modify User Defined Profiles (UDPs). The HP2334A has several pre-defined BDPs (Basic Defined Profiles) which can be used for many standard applications, but certain configurations require a special sets of parameters to be defined (e.g. auto-speed, auto-parity or different flow control mechanisms). A good knowledge of the standard X.3 and local parameters is required to avoid creating erroneous UDPs.

To enter the UDP mode, enter:

```
udp
```

after the * prompt. The following message should appear:

```
Prof number?
```

Respond to this message by entering:

```
2
```

The following message should appear:

```
    -PROFILE : 2                        -FREE SPACE : 43 parameters

    -EXISTING PROFILES : 1,21,31,51,61,71,100,101,121,141

UDP:
```

In response to the `UDP:` prompt, you **must** type:

```
set 11:12,0:13,14:2
```

this defines profiles for the following "free spaces": 11 and 14 respectively. Note that 0 is a separator and not really a parameter.

To see the newly modified profile listing, type:

```
par?
```

The following will appear in your display:

```
PAR 1:1, 2:1, 3:2, 4:0, 5:1, 6:5, 7:21, 8:0, 9:0, 10:0, 11:12,12:1, 13:0, 14:0,
 15:1, 16:8, 17:24, 18:0, 0:13, 1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:128, 8:0, 9:0, 1
0:0, 11:0, 12:0, 13:3, 14:2, 15:0, 16:0, 17:0, 18:63, 19:255, 20:0, 21:0, 22:64,
 23:1, 24:0, 25:0
```

To exit UDP mode, type [RETURN].

The following will appear:

```
Prof number?
```

Respond to this prompt by typing [RETURN].

The * should appear in the display. You are now ready to proceed to the next section where you are to enter the ASG mode.

Entering the ASG Command Mode

The ASG primary user command is used to assign PAD or CAS/PAD profiles to the HP 2334A asynchronous ports. Note that a Remote PAD (associated with CAS/PAD) profile is automatically downloaded by a CAS/PAD profile and is not user assigned.

To enter the ASG command mode, type the following after the *:

```
asg
```

Respond to the `ASG`: prompt by typing:

```
list
```

this gives you a listing of the profile assignments of the ports. Your display should look like this:

```
Assignment for each port

.    1    2    3    4

D    1    1    1    1
C    1    1    1    1
B    1    1    1    1
A    1    1    1    1
```

Change all of the port profile assignments to 2 by typing:

```
a,b,c,d:2
```

Test to see that the port profiles have been changed type:

```
list
```

The display should look like this:

```
Assignment for each port


.    1    2    3    4

D    2    2    2    2
C    2    2    2    2
B    2    2    2    2
A    2    2    2    2
```

Exit the ASG mode by typing: [RETURN]

The * will appear in the display. Turn your HP 2334A OFF (0) to prepare for the next section.

Adding New Entries to the uucp Files

Before proceeding with this section, you need to remove the front panel cover and set switch 2 on the CPU card to the upward (ON) position. All other switches on the CPU card switch packet should be in the down position (OFF).

At this point your HP 2334A has been configured and connected to the X.25 Network. The remaining step-by-step procedure explains how to configure your HP-UX software for use with the X.25 Network.

1. You should have the HP-UX supported terminal which is connected to your HP 2334A set at a baud rate of 2400. Next turn its power on and observe the display the following prompt should appear:

   ```
   @
   ```

   You cannot make a direct connection to the HP 2334A ports through an HP 27130A (8-channel multiplexer) or three of the ports on the HP 98642A (4-channel multiplexer). The port you can use on the HP 98642A is port number 1. You can make direct connection to the HP 2334A ports through an HP 27140A (6-channel multiplexer).

2. All data communication cards should be configured just as if you were going to connect to a modem.

3. Set up the following HP-UX files for *uucp* use: */dev, /etc/inittab, /etc/passwd, /usr/lib/uucp/L.sys, /usr/lib/uucp/L-devices.* These files are set up in the same manner as was explained in the "Uucp File Structure" chapter of this manual with the exception of HP 2334A and X.25 naming conventions. The following are examples of how these files should be set up:

   a. You can have up to 16 terminals remotely or locally connected to the ports on the backplane of your HP 2334A. On the Series 200/300/500 the special device filename for each device on the HP2334A should be as follows:

      ```
      mknod /dev/x25.n c 1 0x000202
      ```

      On the Series 800, use `mknod` to create the special filename as follows:

      ```
      mknod /dev/x.25in  c 10x000202
      mknod /dev/x.25out c 10x000203
      ```

   b. The */etc/passwd* file should have the following entry already made for *uucp*:

      ```
      uucp:password:5:1::/usr/spool/uucppublic:/usr/lib/uucp/uucico
      ```

      where the `password` is assigned by the system administrator for security purposes.

   c. The */etc/inittab* file should contain entries similar to the following for each incoming port from the HP 2334A:

      ```
      00:2:respawn:/etc/getx25 x25.0 2 HP2334A
      ```

      where 2 is the run level. The command to be executed is `/etc/getx25`. The first parameter to the mentioned command is the special (device) file to be used. The second parameter is the speed indicator (baud rate) for getx25, a value of 2 is common. The final parameter is the name of the PAD device you are connected to: `HP 2334A`.

   d. The */usr/lib/uucp/L.sys* file contains entries similar to the following for each HP 2334A device you are able to communicate with on the X.25 Network:

      ```
      hpbm Any,5 ACUHP2334A 9600 f/45762 login:-EOT-login: uucp Password:xxxx
      ```

   e. The */usr/lib/uucp/L-devices* file contains entries similar to the following:

      ```
      ACUHP2334A x25.0 x25.0 2400
      DIR        x25.0 0     2400
      ```

4. If no changes were made to the files mentioned in step 3 then skip this step. However, if changes were made, enter system state 2 to execute the file changes. To do this, type:

   ```
   init 2
   ```

   On the Series 800 use *telinit*, which is documented in *init(1m)* in the *HP-UX Reference* manual.

5. To test to see if the *getx25's* entries are there, type:

   ```
   ps -ef
   ```

6. Execute the following command line:

   ```
   cu -l<line> -m dir
   ```

   where `<line>` is the device name (i.e. x25.0), without `/dev/`. Your display should display the following:

   ```
   Connected
   ERR
   @
   ```

7. You are now ready to test the your HP 2334A. To do this, type the following:

   local_network_address

   where *local_network_address* is an address to another unit which you have access to on the X.25 Network. You should receive a COM message indicating a circuit has been established. If you do not receive a COM message, try to reconfigure the HP 2334A. If this does not work, you should call your Local HP Sales or Service Representative for help.

Creating New Configuration Files

After you have successfully completed the steps in the last section and you decide that you want to talk to another kind of PAD, you have to write new configuration files (scripts). They must be placed in */usr/lib/uucp/X25*, and they must be named *.in, *.out, and *.clr, where * is the name of your "modem type" as specified to *uucp*, without the initial ACU. For example,

```
HP2334A.in
HP2334A.out
HP2334A.clr
```

You **do not** have to modify *dialit.c*, since that program now assumes that any unknown modem type is an X.25 PAD, and looks for the appropriate configuration file to control it.

Each configuration file (script) looks something like a shell file, though it's actually interpreted by *opx25*, a program that talks to the PAD and is thus like a telephone operator. You tell what characters to send, and which ones you expect back. Each file has a different purpose:

```
*.in    detect an incoming call
*.out   make an outgoing call
*.clr   clear the circuit (hang up)
```

As stated above the configuration files are like shell scripts and they are executed using the *opx25* command. The *opx25* command executes HALGOL programs which are scripts used for communicating with devices such as modems and X.25 PADs. The scripts are the configuration files covered in this section.

A configuration file (script) contains lines of the following type:

empty	Lines are ignored
beginning with /	Lines are ignored (comments)
ID:	Denotes a label. ID is limited to alphanumerics or "_".
send STRING	STRING must be surrounded by "". The text is sent to the device. Non-printable characters are represented as in C; if you don't know C, just represent each non-printing ASCII char as \DDD, where DDD is the octal ASCII character code. In the *.out file, \# in a string is taken to be the number being dialed.
break	Send a break "character" to the device

expect NUMBER STRING Here NUMBER is how many seconds to wait before giving up. 0 means wait forever, but this isn't advised. Whenever STRING appears in the input within the time allotted, the command succeeds. Thus, it isn't necessary to specify the entire string. For example, if you know that the PAD will send several lines followed by an "@" prompt, you could just use "@" as the string. The one exception is in the *.in file, where you need to specify at least the end of the expected input. If you just specify a substring in the middle, the rest of the input remains unread until the logger comes along. The logger reads junk, causing it to repeat its login prompt.

run program args The program (sleep, date, whatever) is run with the args specified. Don't use "" here. Also, the program is invoked directly (with execp), so wild cards, redirection, etc. are not possible.

error ID If the most recent expect or run encountered an error, go to the label ID

exec program args Like run, but doesn't fork

echo STRING Like send, but goes to stderr instead of to the device

set debug Sets the program in debug mode. It echoes each line to `/tmp/opx25.log`, as well as giving the result of each expect and run. This can be useful for writing new scripts.

set log `set log` will send subsequent incoming characters to `/usr/spool/uucp/X25LOG`. This can be used in the `*.in` file as a security measure, since part of the incoming data stream contains the number of the caller. There is a similar feature in *getx25*: it writes the time and the login name into the same logfile.

set numlog Like "set log", only better in some cases, since it sends only digits to the log file, and not other characters. For the `*.in` file, for example, "set numlog" gives you information about who has called, but in a compact form.

timeout NUMBER Sets a global timeout value. Each expect uses time in the timeout reservoir; when this time is gone, the program gives up (exit 1). If this command isn't used, there is no global timeout. Also, the global timeout can be reset any time, and a value of 0 turns it off.

exit NUMBER Exits with this value. 0 is success, anything else is failure. The `*.out` file should observe the convention that an exit value of 1 means you couldn't dial the number, and a value of 2 means that you couldn't get the prompt after dialing the number.

You can test configuration files, sort of, by running *opx25* by hand, using the argument "-f" followed by the name of the script file. The program in this case sends to, and expects from, standard output and input, so you can type the input, observe the output, and see messages with the *echo* command.

The content of a configuration file is the HALGOL program (script) you write and place in that file with a filename of your own choosing. These files are executed using the *opx25* command. Below is a list of the *opx25* command line and its options.

opx25 [**-f***script*] [**-c***char*] [**-o***number*] [**-i***number*] [**-n***string*] [**-d**] [**-v**]

where:

[**-f***script*] causes *opx25* to read the configuration file (`script`) as the input program. If `-f` is not specified then *opx25* reads standard input for the `script`.

[**-c***char*] causes *opx25* to use `char` as the first character in the input stream instead of actually reading it from the input descriptor. This is useful sometimes when the program that calls *opx25* is forced to read a character but cannot "unread" it.

[**-o***number*] causes *opx25* to use `number` for the output file descriptor (i.e. the device to use for *send*). The default is 1.

[**-i***number*] causes *opx25* to use `number` for the input file descriptor (i.e. the device to use for *expect*). The default is 0.

[**-n***string*] causes *opx25* to save this `string` for use when `\#` is encountered in a *send* command

[-d] causes *opx25* to turn on the debugging mode

[-v] causes *opx25* to turn on the verbose mode.

The following configuration file (script) is a sequence that could normally be accomplished by entering a set of PAD commands one at a time; however, to save time, this script was written. The configuration file (script) test a PAD connection to see if a virtual circuit is active at the time you are trying to communicate with it, and if there is a virtual circuit active it will clear it.

```
/ clear the HP2334A
timeout 20
/ ignore garbage
send "\021\021\r"
expectg 2 "**++**++**++**++**++**++**++**++**++**++**++**++**++**++**++"

cr:
        send "\r"
        expect 2 "@"
        error brk_clr
        exit 0
brk_clr:
        break
        run sleep 1
        expect 2 "@"
        error dle_clr
        send "CLR\r"
        expect 2 "@"
        error dle_clr
        exit 0

dle_clr:
        send "\020"
        expect 2 "@"
        error cr
        send "CLR\r"
        expect 2 "@"
        error cr
        exit 0
```

Notes

Log, Status and Cleanup 8

This chapter discusses the files that contain information about your transactions, files reflecting system status information, and the programs that compact or delete old or unwanted files. It also contains several shell scripts to implement cleanup operations.

Refer to the appendix for an alphabetical listing and interpretation of messages in `DIAL-LOG`, `LOGFILE` and `SYSLOG`.

Logging Information

The LOGFILE file

A `LOGFILE` is automatically created and maintained by *uucp* for logging all *uucp* communications and transactions. It is the dominant resource for determining the cause of a communications failure. It also keeps track of requests from the local or remote system, files transferred, completion or failure of transfers, success or failure of autodial, and the status of *uux* commands. The logfile is discussed in greater detail under the *uulog* command topic in the chapter "Using the Uucp Facility".

The following example segements from a weekly logfile show the type of messages that are usually found in the file. Each segment is preceded by an interpretation of the segment.

- Remote system called us:

```
user system (date-time-PID) log entry
uucp hpfcms (6/5-8:25-23221) OK (startup)
uucp hpfcms (6/5-8:25-23221) OK (conversation complete)
```

- Remote system `hpfclj` called us and sent job `5954`. We sent them jobs `5952`, `5958`, and `5956`. Note the user also changed:

```
uucp hpfclj (6/6-10:46-5183) OK (startup)
uucp hpfclj (6/6-10:46-5183) REQUESTED (S D.hpcnoaB5954 D.hpcnoaB5954 sww)
sww hpfclj (6/6-10:46-5183) COPY (SUCCEEDED)
sww hpfclj (6/6-10:46-5183) REQUESTED (S D.hpfcljX5952 X.hpflcjX5952 sww)
sww hpfclj (6/6-10:46-5183) COPY (SUCCEEDED)
sww hpfclj (6/6-10:46-5183) REQUESTED (S D.hpcnoaB5958 D.hpcnoaB5958 sww)
sww hpfclj (6/6-10:46-5183) COPY (SUCCEEDED)
sww hpfclj (6/6-10:47-5183) REQUESTED (S D.hpfcljX5956 X.hpfcljX5956 sww)
sww hpfclj (6/6-10:47-5183) COPY (SUCCEEDED)
sww hpfclj (6/6-10:47-5183) OK (conversation complete)
```

- Remote system `hpfclj` called us. We initiated two copies: `1917` and `1915`:

```
uucp hpfclj (6/6-13:39-6583) OK (startup)
dmr hpfclj (6/6-13:39-6583) REQUEST (S D.hpfcljB1917 D.hpfcljB1917 dmr)
dmr hpfclj (6/6-13:39-6583) REQUESTED (CY)
dmr hpfclj (6/6-13:39-6583) REQUEST (S D.hpcnoaX1915 X.hpcnoaX1915 dmr)
dmr hpfclj (6/6-13:39-6583) REQUESTED (CY)
dmr hpfclj (6/6-13:39-6583) OK (conversation complete)
```

- We called remote system `hpfcld`. No work requested.

```
root hpfcld (6/5-6:01-21531) SUCCESSFUL (AUTODIAL)
root hpfcld (6/5-6:01-23531) SUCCEEDED (call to hpfcld)
root hpfcld (6/5-6:01-23531) OK (startup)
root hpfcld (6/5-6:01-23531) OK (conversation complete)
```

- Autodial to remote system `hpfcla` failed:

```
root hpfcla (6/5-8:58-24969) FAILED (AUTODIAL)
root hpdcla (6/5-8:58-24969) FAILED (call to hpfcla)
```

- Autodial to remote system `hpdcd` worked but found no carrier on `hpdcd`.

```
root hpdcd (6/5-7:58-24493) FAIL (NO CARRIER DETECTED)
root hpdcd (6/5-7:58-24493) FAILED (call to hpdcd)
```

- Autodial to remote system `hpdcd` was successful. Login to `hpdcd` failed.

```
root hpdcd (6/5-8:58-24981) SUCCESSFUL (AUTODIAL)
root hpdcd (6/5-8:58-24981) LOST LINE (LOGIN)
root hpdcd (6/5-8:58-24981) LOST LINE (LOGIN)
root hpdcd (6/5-8:58-24981) FAILED (LOGIN)
root hpdcd (6/5-8:58-24981) FAILED (call to hpdcd)
```

- Local system tried to call `hpfcla`. `STST.*` file indicates that over ten tries were attempted. The dialing try to `hpfcla` was consequently stopped.

```
root hpfcla (6/5-20:56-127) NO CALL (MAX RECALLS)
root hpfcla (6/5-20:56-127) CAN NOT CALL (SYSTEM STATUS)
```

- Execution daemon *uuxqt* is sending mail from sww on remote system `hpfclj` to `rmd` and `dmr` on the local system.

```
uucp hpfclj (6/6-10:47-5375) sww XQT (PATH=/bin:/usr/bin;rmail rmd)
uucp hpfclj (6/6-10:47-5375) sww XQT (PATH=/bin:/usr/bin/;rmail dmr)
```

NOTE

If a connection or communication is initiated from your local system, the time in your `LOGFILE` is that of your local time zone.

If it is initiated from a remote system, the time in **your** `LOGFILE` is `EASTERN` time.

The SYSLOG file

The `SYSLOG` keeps track of the number of bytes transferred between systems and the time in seconds it took to complete the transfer. This file is used by *uusub* when reporting traffic statistics between various connections. The chapter, "Uucp File Structure", contains an example of this file.

The DIALLOG file

The DIALLOG file is created by the *dialit* module to log information about the modem used, the telephone number dialed, and the result of the dialing.

The ownership of DIALLOG is set at mode 600 (read permission for owner only) by the *dialit* module. Only the owner, *uucp*, should be able to read DIALLOG because confidential numbers are listed here.

The following segments are from a typical DIALOG file:

```
root ACUVENTEL212 (6/10-6:01-480) SUCCESSFUL (opening of /dev/cua04)
root ACUVENTEL212 (6/10-6:01-480) PHONE OK (phone # 9=226-1111, delay 50 secs)
root ACUVENTEL212 (6/10-6:01-480) MAPPED PHONE - SUCCESS (9&2261111)
root ACUVENTEL212 (6/10-6:01-480) SUCCESS (modem wake up)
root ACUVENTEL212 (6/10-6:01-480) REQUESTED (dial number - 9&2261111)
root ACUVENTEL212 (6/10-6:01-480) ONLINE (remote system)
root ACUVENTEL212 (6/10-6:01-480) SUCCESSFUL (autodial)
```

This section of the DIALLOG file indicates that the autodialing sequence was successful.

When the *uucico* daemon searched the L.sys file for the name of the remote system to contact, a remote system with a modem connection was found. *Uucico* then looked in the L-devices file which specified the line parameters to pass to the *dialit* routine. The *dialit* routine not only performed the autodialing sequence, but also made an entry in the DIALLOG file.

Status

The `STST` files are the system status files. These files are created in the spool directory by *uucico*, and contain information for each remote system, such as *login*, *dialup* or *retry* failures. If *uucp* is aborted, `STST.*` file remains in directory `/usr/spool/uucp`. When two systems are actively communicating, the files contain a `TALKING` status.

Each filename has the form:

```
STST.sys_name
```

where `sys_name` is the remote system name.

For ordinary failures such as *dialup* or *login*, the `STST.*` file prevents repeated tries for about 5 minutes. This is the default time, you can change this time for any system with the `time` field in the `L.sys` file.

When the maximum number of retries (10) is reached, the `STST.*` file must be manually removed before any future attempts to converse can succeed with that remote system. `STST.*` is normally deleted by the uudemons after six hours. However, you can find information about the transaction in `LOGFILE`.

You can use *uustat* to check on one or all jobs that have been queued. The identification printed when a job is queued is used as a key to query status of the particular job. For example:

```
uustat -j123
```

returns a message similar to:

```
123 user system 06/08-08:30 06/08-9:46
JOB IS QUEUED
```

You can also use *uustat* to check on the status of the last transfer to each system on the network, for example:

```
uustat -mall
```

returns a message resembling:

```
sys1 06/08-8:50 CONVERSATION SUCCEEDED
```

When sending files to a system that has not been contacted recently, the time of last access to that system can help you determine whether the system is in service.

You can use *uusub* to set up and monitor subnetwork statistics. Two files are created by *uucp*: **L_sub** and **R_sub**, which keep track of the defined subnetwork and their corresponding statistics.

Refer to the chapter, "Using The *Uucp* Facility" for more information about *uustat* and *uusub*.

Cleanup

The following shell scripts can be started from *cron* to routinely compact the log files, **SYSLOG** and **LOGFILE**, and to clean up any backlog of jobs that could not be transmitted to other systems.

Enter the following lines into a file having a name of your choosing:

```
56 6-22 * * * /usr/lib/uucp/uudemon.hr >> /usr/spool/uucp/DEMONLOG 2>&1
0 6 * * * /usr/lib/uucp/uudemon.day >> /usr/spool/uucp/DEMONLOG 2>&1
5 * *1 /usr/lib/uucp/uudemon.wk >> /usr/spool/uucp/DEMONLOG 2>&1
```

Next, type the following:

```
crontab filename
```

Spool Cleanup Script

```
May 29 15:18 1983  uudemon.hr Page 1

# UNISRC_ID: @(#)uudemon.hr     14.1    83/05/01
#*****************************************************************
#*    (c) Copyright 1983 Hewlett Packard Co.
#*        ALL RIGHTS RESERVED
#*****************************************************************

#
# This is an example of a daemon to run hourly.
# It cleans up bad spool entries and establishes
# communications with specified systems.  This daemon
# normally runs at 20 minutes past the hour.

/usr/bin/uulog
/usr/lib/uucp/uuclean -pLCK -n6

# The entries below are to contact the system specified
# on an hourly basis.
# Uncomment the line and replace <nodename> with the proper
# system names of the remotes you want to contact.  Add more
# entries or delete to match your situation.

# /usr/lib/uucp/uucico -r1 -s<nodename>
# /usr/lib/uucp/uucico -r1 -s<nodename>
# /usr/lib/uucp/uucico -r1 -s<nodename>
```

The **−s** system option forces a call to systems that may be PASSIVE (receives calls from other systems) only with respect to you.

Log System Cleanup Script

```
May 29 15:17 1983  uudemon.day Page 1

# UNISRC_ID: @(#)uudemon.day    14.1    83/05/01
#****************************************************************
#*    (c) Copyright 1983 Hewlett Packard Co.
#*        ALL RIGHTS RESERVED
#****************************************************************

#
# This is an example of a daemon which should run daily
# to clean up log system and call those remotes
# you wish once per day.  Normally run at 4:00 am.
#
/usr/bin/uulog
cat /usr/spool/uucp/LOGFILE >> /usr/spool/uucp/LOG-WEEK
/usr/lib/uucp/uuclean -pLOGFILE -n0
cat /usr/spool/uucp/SYSLOG >> /usr/spool/uucp/SYS-WEEK
/usr/lib/uucp/uuclean -pSYSLOG -n0
cat /usr/spool/uucp/DIALLOG >> /usr/spool/uucp/DIAL-WEEK
/usr/lib/uucp/uuclean -pDIALLOG -n0
#
# Below is the command to call up a system once per day.
# Replace <nodename> by the system name of the remote you want
# to contact, then uncomment the line.
#
#/usr/lib/uucp/uucico -r1 -s<nodename>
```

Weekly Logfile Cleanup Script

```
May 29 15:19 1983  uudemon.wk Page 1

# UNISRC_ID: @(#)uudemon.day    14.1    83/05/01
#****************************************************************
#*    (c) Copyright 1983 Hewlett Packard Co.
#*        ALL RIGHTS RESERVED
#****************************************************************

#
# This is an example of a daemon to be run once per week.
# The entries delete the old weekly log files and clean up
# the /usr/spool/uucp directory.
#
/usr/lib/uucp/uuclean -pTM -pC. -pD. -pLTMP -pLOG. -pdead -pX
/usr/lib/uucp/uuclean -pLOG-WEEK -pSYS-WEEK -n0
/usr/lib/uucp/uuclean -pDIAL-WEEK -n0
```

Refer to the *uudemons* section of the "Uucp Daemons" chapter.

Problems 9

This chapter discusses problems that are most commonly encountered when using *uucp* facilities.

Bad Connections

A bad connection is the most cause of *uucp* malfunctions. Both direct and modem connections occasionally experience difficulty when connecting to remote systems. If you examine the `LOGFILE` in the */usr/spool* directory, the remote entry usually points to a bad direct or modem line. Also verify that you are using compatible modems if a modem link interconnects the two computers. Use *cu* to interactively attempt a call to the other system over the problem line.

When the transaction cannot run to completion, the temporary file `TM.*` is not copied into the destination file so it remains in directory */usr/spool/uucp*.

Out of Space

When the disc containing directory */usr/spool* is out of space, work requests cannot be sent or received. This occurs when the system is heavily used or if non-transmittable files have not been cleaned up. If your situation does not require that a copy of the file be stored in the spool directory until transmission is ready to start (the `-C` option in the *uucp* command), use the default *-c* option instead.

Out-of-date Information

Passwords, logins and phone numbers for remote systems are sometimes changed without your knowing of the change. Be sure that your automatic dialing mechanism does not keep trying to dial an unreachable system.

You should not change the mode (protection) bits on *uucp* files that are command modules. For example, commands need an execution-by-everybody mode.

Abnormal Termination

DO NOT turn off power to your computer while *uucp* is running even though *uucp* may be running in background mode.

Do not press any key on your keyboard while you are using *cu* with "~%take" or "~%put".

Notes

Log Entry Messages A

This appendix provides an alphabetical listing and interpretation of messages found in DIALLOG, LOGFILE and SYSLOG files.

/usr/spool/uucp/DIALLOG

Dialit logs dialing status information in the DIALLOG file. Direct *uucp* connections do not involve dialing, so they do not produce DIALLOG file entries. This file grows very quickly. If a collision occurs between two processes wishing to append to the file, one of them starts a DIAL.<pid> file and writes all further messages to that file instead of to DIALLOG (pid is the process identification number).

All DIAL* files contain potentially sensitive information (i.e., phone numbers for other systems) and should be protected against unauthorized access. Therefore DIAL* files are owned by user *uucp*, and their protection mode is set as unreadable and unwritable by the public at the time they are created.

The *dialit* routine can be modified by users by recompiling the source.

Meaning of Entries

The information given here may not be applicable to a modified version of *dialit.*

The general format for an entry in the DIALLOG file is:

```
user cu_type (month/date-hour:min-pid) message
```

where:

user	is the user who requested the transaction
cu_type	is the calling unit type, e.g. a device as defined in /ust/lib/uucp/L-devices
month/date-hour:min	refers to the 24-hour time based on the timezone of the originator process (values can be set by the invoking parent process)

`pid` is the process identifier of the process performing the logging operation (useful for tracing individual process's log entries)

`message` is one of a number of message lines (refer to the "Message Interpretations" section below).

Sample Entries

These are some sample `DIALLOG` entries.

```
uucp ACUVENTEL (8/24-15:43-8307) SUCCESSFUL (opening of /dev/cul03)
uucp ACUVENTEL (8/24-15:43-8307) PHONE OK (phone # 3524-, delay...)
uucp ACUVENTEL (8/24-15:43-8307) MAPPED PHONE - SUCCESS (3524)
uucp ACUVENTEL (8/24-15:43-8307) SUCCESS (modem wake up)
uucp ACUVENTEL (8/24-15:43-8307) REQUESTED (dial number - 3524)
uucp ACUVENTEL (8/24-15:43-8307) ONLINE (remote system)
uucp ACUVENTEL (8/24-15:43-8307) SUCCESSFUL (autodial)
```

Message Interpretations

`ATTEMPTING (second modem wakeup)`
This is logged after the first wakeup attempt fails.

`BAD (phone number - <number>)`
The phone number the autodial module used is incorrect.

`ERROR (bad character in phone number <character>)`
<character> in phone number is not recognized by dialcodes or dialit module.

`FAILED (autodial)`
This message appears as a frequent companion with other `FAILED` messages. The autodial can fail to make a connection for many different reasons: invalid cua device, failure to open the cua special file, an incorrect phone number, no response from the dial prompt, the modem wakeup failed, the attempt to dial failed, the modem type is unknown, slow answer (with carrier), a busy number, or line noise.

`FAILED (connection with remote system)`
The autodial module failed to connect to a remote system. This is a secondary entry logged after some other failure.

`FAILED (dial of phone <phone_number>)`
`FAILED (dialing of phone number)`
The modem reported dial failure probably due to no answer fast enough (with carrier), a busy number, or line noise.

`FAILED (invalid cua device)`
The configuration of the cua device is incorrect. Check the mknod command, the getty entry, the L.sys and the L-devices special file names.

`FAILED (mapping of phone number)`
Special characters, such as "=", "-", in the phone number could not be interpreted by the dialcodes module.

`FAILED (modem wake up)`
The dialit program could not wake up and synchronize with the modem the first time it tried.

`FAILED (no response from dial prompt)`
After waiting for a length of time, there was no response from the dial prompt.

`FAILED (open of cua <device>)`
The *uucp* `facility could not open the call unit. Check the mknod` command special file, the getty entry and the `L.sys` and `L-devices` cua field entries.

`FAILED (second wake up)`
The dialit program could not wake up and synchronize with the modem the second time it tried.

`MAPPED PHONE - SUCCESS (<mapped phone number>)`
The dialit routine succeeded in mapping the phone number as given, containing special characters such as "-" (pause) and "=" (secondary dial tone) separators, into the form the modem understands. This is the form `<mapped phone number>` appears in.

`NOT ATTEMPTING (second wakeup)`
This message follows the "FAILED (modem wakeup)" for certain modems.

`NOT KNOWN (modem type specified)`
The *dialit* module does not recognize the modem device specified.

`NOT RECEIVED (modem parity message)`
The modem is not set for the proper parity.

`ONLINE (remote system)`
Dialit got a carrier signal from the remote modem.

`PHONE OK (phone # <original phone number>, delay <secs> secs)`
Dialit accepted the given phone number as valid, containing "-" (pause) and "=" (secondary dial tone) separators. This is the form `<original phone number>` appears in. `<secs>` is the total computed timeout that dialit allows the modem for dialing the phone number and returning a response.

`REQUESTED (dial phone - <mapped phone number>)`
Dialit instructed the modem to dial the phone number shown, for the first time. The format of `<mapped phone number>` is the actual form passed to the modem.

`RETRYING (dial number - <mapped phone number>)`
Dialit instructed the modem to dial the phone number shown, for the second time, after a failure. The format of `<mapped phone number>` is the actual form passed to the modem.

`SUCCESS (modem wake up)`
Dialit succeeded in resetting and synchronizing with the local modem.

`SUCCESS (modem wake up - second attempt)`
Dialit succeeded in resetting and synchronizing with the local modem on the second attempt.

`SUCCESSFUL (autodial)`
This is the last entry logged for a successful autodial. It means *dialit* terminated and returned "successful".

`SUCCESSFUL (dial phone <phone number>)`
The dialit module succeeded in dialing the `<phone number>`.

`SUCCESSFUL (opening of cua device <devicename>)`
Dialit managed to open the autodial device `<devicename>` to talk to the modem.

/usr/spool/uucp/LOGFILE

Uucp, uux, uucico, and *uuxqt* log status information here. The file grows very quickly. If a collision occurs between two processes wishing to append to the file, one of them starts a `LOG.<pid>` file and writes all further messages there instead of `LOGFILE`. If *uulog* is invoked with no arguments, it appends all `LOG.*` files to `LOGFILE` and then removes the `LOG.*`.

NOTE

If a `LOG.*` file is active (still in use) when this occurs, the process using it continues to hold the file open and write to it. Since `LOG.*` is unlinked when closed, all information written after the *uulog* executes is lost.

Meaning of Entries

The general format for a `LOGFILE` entry is:

```
user system (month/date-hour:min-pid) message
```

where:

`user`	is the name of the user requesting the transaction
`system`	is the name of the remote system (may be a null or undefined field if specified by the remote system)
`month/date-hour:min`	is the 24-hour time based on the transaction originator's time zone (the timezone could be set to any value by the invoking parent process but never gets set for transactions initiated by remote systems since their login shell is *uucico*. The times in the file may not be sequential because of old `LOG.*` files appended by *uulog*)
`pid`	is the process identifier of the logging process (useful for tracing individual process's log entries)
`message`	is one of a number of message lines (refer to the "Message Interpretations" section below).

Sample Entries

The following entries are sample entries from a `LOGFILE`.

```
uucp hp-pcd (8/24-14:34-7710) SUCCESSFUL (AUTODIAL)
uucp hp-pcd (8/24-14:34-7710) SUCCEEDED (call to hp-pcd )
uucp hp-pcd (8/24-14:34-7710) OK (startup)
uucp hp-pcd (8/24-14:34-7710) REQUEST (S D.hp-pcdB1170 ...)
uucp hp-pcd (8/24-14:35-7710) REQUESTED (CY)
uucp hp-pcd (8/24-14:35-7710) REQUEST (S D.hpfclaX1168 ...)
uucp hp-pcd (8/24-14:35-7710) REQUESTED (CY)
uucp hp-pcd (8/24-14:35-7710) OK (conversation complete)
```

Message Interpretations

`ACCESS (DENIED)`
A system tried to access a file for which it did not have file path access permission.

`BAD READ (expected <message> got <message>)`
Transaction terminated abnormally; `<message(s)>` indicate problem.

`CAN NOT CALL (SYSTEM STATUS)`
A `/usr/spool/uucp/STST.<nodename>` file still exists for this nodename. The system specified with the `[-ssys]` option could not be called.

`CAUGHT (SIGNAL N)`
An interrupt, hangup, quit or terminate signal was generated during the uucico operation. `N` is the signal number.

`COPY (FAILED)`
The system failed to copy a requested file.

`COPY (SUCCEEDED)`
The system succeeded in copying a requested file.

`DENIED (CAN'T OPEN)`
An unauthorized access to a protected file was requested.

`DONE (WORK HERE)`
All local copies are finished.

`FAIL (NO CARRIER DETECTED)`
After an otherwise successful *dialup*, *uucico* checked and found no carrier on a modem line, independent of *dialit.*

`FAILED (AUTODIAL)`
Could not dial another system for some reason; see `DIALLOG`.

`FAILED (CAN'T CREATE TM)`
The temporary file (used to hold data until the transfer has completed successfully) can not be created.

`FAILED (CAN'T READ DATA)`
The input file is protected and can not be opened.

`FAILED (DIALUP ACU write)`
An error occurred trying to access the modem.

`FAILED (DIALUP LINE open)`
The dial was completed but the line could not be opened.

`FAILED (LOGIN)`
Could not successfully negotiate the login sequence specified in the `/usr/lib/uucp/L.sys` file.

`FAILED (call to <nodename>)`
Usually a secondary entry, after a different failure entry.

`FAILED (conversation complete)`
Usually a secondary entry, after another failure occurred. This could be because a packet of information could not be transferred correctly or the connection had a problem.

`FAILED (startup)`
The local and remote systems could not agree on a protocol.

`<file name> XUUCP (DENIED)`
A request to copy a protected remote file was denied.

`HANDSHAKE FAILED (BADSEQ)`
The remote system sequence number on the local system does not match the local system sequence number on the remote system.

`HANDSHAKE FAILED (CB)`
Succeeded in logging in on a remote system, but the other system is set up to call this system back, so the connection failed. Usually the other system then calls back within a short time (usually on a cheaper line or at a higher baud rate).

`LOCKED (call to <nodename>)`
A `/usr/spool/uucp/LCK..<nodename>` file already exists for the nodename, due to another conversation already in process or a file left behind due to some sort of abort.

`LOST LINE (LOGIN)`
An error occurred during the login process.

`NO (AUTODIAL DEVICE)`
There is no available autodial device. A `/usr/spool/uucp/LCK..<devicename>` file already exists for every possible device, due to another conversation already in process or a file left behind due to some sort of abort.

`NO (DEVICE)`
A device having charactertics matching an entry in the `L-devices` file could not be found.

`NO CALL, MAX RECALLS`
The call was attempted ten times without success.

`NO CALL (RETRY TIME NOT REACHED)`
A `/usr/spool/uucp/STST.<nodename>` file still exists for this nodename, or, ... and in any case, the retry time specified in `/usr/lib/uucp/L.sys` has not yet been reached.

`NO WORK (<nodename>)`
Uucico was initiated for `<nodename>`, but there is no work pending for that system.

`OK (conversation complete)`
Normal end of conversation with a remote system.

`OK (startup)`
Normal start of conversation with a remote system. If this system is the master, this entry is preceded by other entries; if this system is the slave (it was logged into), this is the first entry for the conversation.

`PERMISSION (DENIED)`
An unauthorized access to a protected file was requested.

`PREVIOUS (BADSEQ)`
The call to the remote system failed because the remote system's sequence number for the local system did not match the local system's sequence number for the remote system.

QUE'D (<nodename>)
A copy (*uucp*, not *uux*) operation was queued on the local system, destined for **<nodename>**.

REQUEST (COPY FAILED <message>)
The message could be any of the following:
<no message> the reason is not given by remote system can't copy to directory/file - file left in *uucppublic* local access to path denied remote access to path/file denied remote system can't create temporary file system error - illegal *uucp* command generated.

REQUESTED (CY)
The request for work was completed.

REQUESTED (file user)
The **user** on a remote system requests the local **file** transferred.

REQUIRED (CALLBACK)
The remote system called requires that both systems hang up, the remote system then calls the original caller asking the originator to verify its identity.

REQUEST (S <source filename> <dest filename> <username>)
Start of a transfer from this system to a remote system.

return_number from system user (MAIL FAIL)
The *mail* command failed and the **return_number** was returned to the sender: **user** on **system**.

SUCCEEDED (call to <nodename>)
After successful autodial or direct connect, this entry indicates that the local system succeeded in logging in to the remote, but has not (yet) synchronized with the remote *uucico.*

SUCCESSFUL (AUTODIAL)
This is the first entry for an outbound conversation, where the local system is the master. It means the local system has succeeded in connecting with the remote, but has not (yet) logged in.

TIMEOUT (AUTODIALER)
The autodial module took longer than timeout value to make a successful connection to a remote system. The timeout value is 30 seconds or five times the length of the phone number, whichever is longer.

`TIMEOUT (DIALUP DN write)`
The autodial module took longer than two minutes to make a successful connection to a remote system.

`user XQT (DENIED command)`
The **user** tried to execute a **command** on a remote system which was not in the remote system's `L.cmds` file.

`user XQT (path)`
The file name which included a path was expanded to the full path name.

`/usr/spool/uucp/LCK.SQ (CAN'T LOCK)`
An error occurred five times trying to lock the sequence file.

`WRONG TIME TO CALL (<nodename>)`
The `/usr/lib/uucp/L.sys` file does not allow a call to the remote system at this time. This sort of entry appears if the `L.sys` line for the nodename does not parse properly, for reasons ranging from it containing a nodename only (the simplest way to queue work only for a remote site the local system is passive to) up to an error in the syntax given for the legal call times.

`XQT QUE'D (<command line>)`
A remote execute (*uux*, not *uucp*) operation was queued on the local system, destined for a remote system nodename.

`<username> XQT (PATH=<pathlist>;<commandline>)`
After *uucico* completes a conversation, it starts *uuxqt* to process all `/usr/spool/uucp/X.*` files. One such entry is logged for each `X.*` file processed.

/usr/spool/uucp/SYSLOG

Uucico logs information here about actual bytes transferred. The file grows quickly. In case of a collision some data may be lost.

The general form for a SYSLOG entry is:

```
user system (month/date-hr:min) (sec) direction data <bytes> bytes <secs> secs
```

where:

user	is the user who requested the transaction (may be a user from the remote system)
system	is the name of the remote system (may be null or undefined if specified by the remote system)
month/date-hr:min	is the 24-hour time based on the timezone given by the originator (the timezone could be set to any value by an an invoking parent process; it never gets set for transactions initiated by remote systems, since their login shell is *uucico.*)
sec	is the time in seconds of the current system clock (independent of any timezone and is useful for programs that deal with the data numerically);
direction	can be either "sent" or "received"
bytes	is the integer number of bytes transferred
secs	is the integer number of elapsed seconds for the transfer.

Notes

Index

a

ADMIN file 55, 72
ASI card installation for Series 500 9

b

binary files 54

c

cleanup script 138–140
cu command:
  data transmission 81–82
  description of 1, 79
  direct connection 80–81
  modem connection 79–80
cua device file, creation of 32–34
cul device file, creation of 32–34

d

daemons, *uucp* 73–76
data files:
  data execution files 48
  image data files 47–48
data transfer:
  multiple files between local and remote system 41–43
  single file between local and remote system 40–41
  uux command sequences 43
device files, creation of 32–34
Dialit file 54, 67
Dialit.c file:
  description of 54, 63
  program example 64–67
DIALLOG file 53, 136, 143–146

direct connections:
- 6-channel MUX to 6-channel MUX ... 21, 24, 25, 26
- 8-channel MUX to 6-channel MUX ... 21, 23
- Serial interface card to 6-channel MUX ... 21, 22, 25
- Series 200/300 serial to Series 200/300 serial ... 18
- Series 200/300 serial to Series 500 8-channel MUX ... 15
- Series 200/300 serial to Series 500 ASI ... 16–17
- Series 500 8-channel MUX to Series 500 8-channel MUX ... 19
- Series 500 ASI to Series 500 8-channel MUX ... 18
- Series 500 ASI to Series 500 ASI ... 20–21
- special connector ... 27–28
- special considerations ... 27

e

execution files:
- command line ... 49, 51–52
- example ... 52
- fields ... 49
- required file line ... 49, 50
- standard input information line ... 49, 51
- standard output information line ... 49, 51
- user line ... 49, 50

f

file structure, *uucp* ... 39

g

getting started ... 5

getty entries ... 31, 36

h

hardware configuration ... 9–28

HP AdvanceNet ... 8

HP 2334A configuration procedure:
- adapter cards ... 109–110
- adding new *uucp* entries ... 125–127
- ASG command mode operation ... 124–125
- configuration procedure ... 113–131
- CONFIGURE mode operation ... 113–115
- connecting cables to an HP 2334A ... 108–110

creating new configuration files 127–131
HSA command mode operation 119
installation 107–110
local off-line configuration 108–110, 111
power-on and CPU switch test 107–108
preparing for configuration 112
prerequisites 106–107
remote off-line configuration 110–111
sample off-line configuration listing 115–119
step-by-step HSA configuration procedure 120–122
UDP command mode operation 123–124

i

installing software 29–38
invoking *uucp* daemons 75–76

l

L-devices file 54, 61–62
L-dialcodes file 54, 62
LAN network 7
L.cmds file 54, 55
library files:
ADMIN 55, 72
Dialit 54, 67
Dialit.c 54, 63–67
editing for *uucp* 37–38
file names 54–55
L-devices 54, 61–62
L-dialcodes 54, 62
L.cmds 54, 55
L.sys 55, 68–72
SEQF 54, 56
USERFILE 54, 57–61
loading optional drivers 30–31
lockfiles 52, 73
log files:
cleanup 138–140
description of 53
DIALLOG 53, 136, 143–146
entry messages 143–153

LOGFILE . . . 53, 93, 133–135, 147–152
SYSLOG . . . 53, 135, 153
login name and password for *uucp* . . . 35
L.sys file . . . 55, 68–72

m

mail command . . . 2, 101
modem connections . . . 13–14
multimux interface installation (see HP 2334A configuration procedure) . . . 106–131
MUX card installation:
Series 200/300 . . . 10–12
Series 500 . . . 9–10
Series 800 . . . 10

n

networks:
HP AdvanceNet . . . 8
LAN (Local Area Network) . . . 7
PDN (Public Data Network) . . . 105
PSN (Packet Switching Network) . . . 103–105
RJE (Remote Job Entry) . . . 6–7
node names, *uucp* . . . 34

p

Packet Switching Network (PSN) . . . 103–105
pathnames . . . 77–78
problems:
abnormal termination . . . 141
bad connections . . . 141
out of space . . . 141
out-of-date information . . . 141
process spawning, *uucp* . . . 73–75
public area . . . 44
Public Data Network (PDN) . . . 105

r

RJE network . . . 6–7

s

security sequence-checking:
SEQF file 54, 56
SQFILE 56
separators required between *uucp* command options 78
serial I/O card installation for Series 200/300 10–12
software installation:
boot and login process 30–31
creating a *tty* device file 32–34
editing library files for *uucp* 37–38
getty entries 31, 36
loading optional drivers 30–31
node names 34
uucp login 35
spool directory:
public area 44
uucp directory 44
workfiles 44–47
status files, *uucp* 137–138
SYSLOG file 53, 135, 153

t

temporary files 52–53
troubleshooting 141
tty device file, creation of 32–34

u

USERFILE:
example 58
line entry 55–58
null user entry 60–61
search (local and remote systems) 59–61
uucico daemon 73–75
uuclean command 92
uucp:
command errors 88
command execution 73–75
command syntax 83–84
directory 44
file structure 39

forwarding files through several systems 85–87
invoking daemons 75–76
login 35
node names 34
option separators 78
pathnames 77–78
process spawning 73–75
program description 1–2
receiving files from remote system 85
sending files to a remote system 84–85
status files 137–138
X.25 configuration (see HP 2334A configuration procedure) 106–131
uudemons 76
uulog command 93
uuname command 94
uupick command 95–96
uustat command 96–97
uusub command 98–100
uuto command 100
uux command:
command description 2
command sequences 43
command syntax 89–90
error numbers 91
example 90–91
uuxqt daemon 75, 76

W

workfiles 44–47

X

X.25:
configuring *uucp* for (see HP 2334A configuration procedure) 106–131
description of 103
Packet Switching Network (PSN) 103–105
Public Data Network (PDN) 105

Table of Contents

Using Curses and Terminfo

Introduction 1
Display Data Handling 2
- Output Data Structure 2
- Applications Program Structure 3
- Applications Program Operation 5

Keyboard Input 6
- Keypad Character Handling 7
- Keyboard Input Program Example 9

Display Highlighting 10
Multiple Windows 13
- Pads 13
- Creating Windows 14
- Using Multiple Windows 14
- Subwindows 16

Multiple Terminals 17
Low-Level Terminfo Usage 19
A Larger Example 21
- Use of Escape in Program Control 22

Program Routines 23
- Program Structure Considerations 23
- Terminal Initialization Routines 24
- Option Setting Routines 25
- Terminal Configuration Routines 26
- Window Manipulation Routines 27
- Terminal Data Output Routines 28
- Window Writing Routines 28
- Window Data Input Routines 30
- Terminal Data Input Routines 30
- Video Highlighting Attribute Routines 31
- Miscellaneous Functions 32

curses Routines 33
Description of Routines 33
Terminfo Routines 52
Termcap Compatibility Routines 54

Program Operation 55
Insert/Delete Line 55
Additional Terminals 55
Multiple Terminals 56
Video Highlighting 57
Special Keys 59
Scrolling Regions 60
Mini-Curses 60
TTY Mode Functions 61
Example Programs 63
SCATTER 63
SHOW 64
HIGHLIGHT 65
WINDOW 66
TWO 68
TERMHL 70
EDITOR 72
Index

Using Curses and Terminfo

Introduction

This tutorial describes the operation of *curses(3x)* and *terminfo(5)*. It is intended for use by programmers who are interested in writing screen-oriented software using the *curses* package. *curses* uses *terminfo* when interacting with a given terminal in the system and when formatting display data for subsequent output to the terminal display.

curses is a versatile cursor and screen control package that has many capabilities. It is designed to efficiently utilize terminal screen control and display capabilities, thus limiting its demand for computer CPU resources. It can create and move windows and subwindows, use display highlighting features, and support other terminal capabilities that enhance visual interaction with display terminal users. All interaction with a given terminal is tailored to the terminal type which is obtained from the environment variable `TERM`).

curses also interacts with the terminal keyboard, and can handle user inputs. Its ability to handle keys that produce multi-character sequences (such as arrow keys) as ordinary keys can be used to add versatility to application programs.

Display Data Handling

Output Data Structure

curses uses data structures called windows to collect display text, then transfers the data structures to the terminal display screen during execution of *refresh* routines. Each window contains a two-dimensional data array for storing text and character highlighting attributes. Other data structures associated with the window contain the current cursor position and various pointers, and fill other *curses* needs.

Two windows are always present when *curses* is active. **Current screen** is named *curscr* for programming purposes, and represents the current screen. It is used as a reference when optimizing output operations to the CRT screen. The **standard screen** window, named *stdscr*, is the default destination for all text output operations that are not directed to a window specified in the function. Both *curscr* and *stdscr* have the same row and column dimensions as the physical display screen.

Additional program-definable windows can be created and dimensioned as programming needs dictate. Such windows can be any size, provided they do not exceed the row and/or column capacity of the physical display screen.

When a program requires a window that is larger than the available display screen, pads are used. Pads have the same structure and characteristics as a window, but they can be any size within the limits of reasonable memory usage (each pad requires two bytes of memory per character position in the pad, plus data structure overhead).

Text and Highlighting Data Format

Every window data structure contains, among other things, a two-dimensional array of 16-bit data words, each word corresponding to a displayable character in the window. Seven bits in each 16-bit word contain the 7-bit character code of the character associated with the corresponding screen display position. The remaining nine bits specify which highlighting attributes, if any, are to be used when the character is displayed. The window data structure also contains a set of current attributes that are used to form the attribute bits as each word is placed in the array by *addch* or its equivalent. If text highlighting is to be changed for a given character or set of characters, an update to the current attribute set must be performed by *attrset* (or its equivalent) before *addch* is performed. The beginning default attribute set disables all highlighting.

Applications Program Structure

Consider the following example of an application program structure that uses *curses*:

```
#include <curses.h>
. . .
    initscr(); /* Initialization */

    cbreak();/* Various optional mode settings */
    nonl();
    noecho();
. . .
    while (!done) {/* Main body of program */
       . . .
       /* Sample calls to draw on screen */
       move(row,col);
       addch(ch);
       printw("Formatted print with value %d\n", value);
       . . .
       /* Flush output */
       refresh();
       . . .
    }

    endwin();/* Clean up */
    exit(0);
```

Example of Program Framework for Using *curses*

One of *curses*' major advantages is its ability to optimize the process of updating terminal screen contents, thus reducing the demand for CPU and I/O resources by reducing the amount of data handling required for requested changes in displayed text. This is accomplished by comparing the current screen contents with the window being transferred, then transmitting only those text and control characters that are needed to most efficiently update the screen. Other screen contents remain undisturbed.

NOTE

Most terminals are equipped with hardware scrolling whose operating characteristics make it impossible to write characters in the extreme lower right-hand character position.

In order to optimize screen updates, *curses* must have access to a data base that reflects current screen contents. When an application program starts execution, the current screen is unknown. To provide a starting current screen reference, a screen clearing operation must be set up early in the program by a call to *initscr()* which identifies the terminal, initializes data structures, and enables the *clearok* option in *curses* so that the screen is cleared during the first *refresh* operation in the program. Upon completion of the first refresh operation, the terminal screen is an exact replica of the text stored in the current screen data base. Use of *initscr()* in a typical program is shown in the preceding sample program structure example.

When initialization is complete, other operating modes and options can be selected as dictated by program needs. Available operating modes include *cbreak()* and *idlok(stdscr, TRUE)* which are explained in detail later. During program execution, screen output is handled through routines such as *addch(ch)* and *printw(fmt,args)*. They are equivalent to *putchar* and *printf*, respectively, but use *curses* in addition to the usual other system facilities. Cursor and character positioning are performed by *move* and other similar calls.

All of the routines mentioned send their output to program-specified window data structures; not directly to the display screen. The window data structure represents all or part of a CRT display screen, and contains the following items:

- An array of characters to be displayed on the screen area defined by the window boundaries,
- Present cursor location,
- Current set of video attributes, and
- Various operating modes and options.

There is little need to be concerned with windows (unless you use several windows during program operation), except to recognize that the data structure corresponding to a given window acts as a buffer/data accumulator for display output requests.

Accumulated contents of a window data structure are sent to the display screen by use of *refresh()* or an equivalent function for windows and pads (functionally similar to a *flush*). *curses* considers many different ways of handling the output operation, taking into account the various available terminal characteristics, similarities between the current screen display and the desired pattern, and other factors. Refresh operations are usually handled using as few characters as possible, but not always.

When the application program is finished, certain clean-up operations should be performed before termination. While the amount of clean-up needed varies, depending on program structure and capabilities, termination should always include a call to *endwin()*. *endwin()* restores all terminal settings to their original state prior to program execution, places the cursor at the bottom left corner of the screen, and dismantles data structures that are no longer needed.

Among the example programs at the end of this tutorial is a program named *scatter* that reads a file and displays the file contents in random order on the CRT display screen. While some application programs assume that terminals have twenty-four 80-character lines of available display space, many terminals do not. To accommodate display terminals having various screen sizes, the variables `LINES` and `COLS` are defined by *initscr* to specify the current screen size. Application programs should always use screen-size variables rather than assuming a 24×80 display screen.

Applications Program Operation

During program operation, no data is output to the display terminal until *refresh* is called. Instead, program routines such as *move* and *addch* place data in a window data structure called *stdscr* (standard screen) that is maintained by *curses*. *curses* also maintains a replica of what is on the current physical screen in *curscr* for updating purposes.

When *refresh* or an equivalent function is called, *curses* compares the *curscr* window with what is presently contained in *stdscr* (or other specified window or pad). The results of the comparison are combined with terminal hardware capabilities to construct character streams that most efficiently update the physical display to the desired contents. Available terminal capabilities are considered while comparing *stdscr* and *curscr* so that the most efficient means of updating the screen can be determined. This sequence is referred to as cursor optimization, and is the basis for naming the *curses* package. During the update operation, *curscr* is also changed to reflect the contents of the updated screen.

Keyboard Input

curses capabilities include more than screen writing functions. Several keyboard input functions are also supported, including special handling of certain keys that normally generate a sequence of two or more characters (usually an escape code followed by a single character, but not always). Such keys can then be treated as ordinary single-character keys for improved programming versatility.

The most commonly used keyboard input function is *getch( )* which waits for the terminal user to type a character on the terminal keyboard, then returns the character to the calling program. *getch* is similar to *getchar*, except that it uses *curses* instead of other HP-UX facilities. *getch* is particularly useful in programs that use *cbreak( )* or *noecho( )* options because *getch* supports several terminal- and system-dependent options that are not accessible through *getchar*. Available *getch* options include:

- *keypad* enables programmers to use non-typing keys such as arrow keys, function keys, and other special keys that transmit escape sequences or other multi-character sequences as ordinary single-character keys. Keypad character code length requires 16-bit integer variables for storage.
- *nodelay* enabled option causes *getch* to return immediately with the value −1 if no input character is waiting. This avoids program delays that would otherwise result when no response from the terminal is available.
- *getstr* can be used to input an entire string of characters up to a newline instead of a single character. It also handles echo, erase, and kill character functions associated with the input operation.

Example programs at the end of this tutorial show how these options are used.

Keypad Character Handling

When *keypad* is enabled, keypad character sequence conversion tables in the *terminfo* data base are used to map keypad character sequences into corresponding single, 16-bit character form. Each supported keypad key must produce a unique character or character sequence when pressed. All convertible sequences must be included in the *terminfo* data base. If any sequence is absent from the table, it cannot be converted, so it is handled in unaltered form. The following special keys are assigned the values and names indicated. Some of the keys listed may not be supported on given terminals, depending on the terminal model and its internal operating characteristics, and whether the conversion sequence is in *terminfo*.

NOTE

Keypad character codes do not fit in a normal 8-bit data element. Therefore a *char* variable cannot be used. Use a larger (16-bit) variable for storing and handling keypad character codes.

Keypad Character Code Values

Character Name	Octal Value	Key name
KEY_BREAK	0401	Break key (unreliable)
KEY_DOWN	0402	Down Arrow key
KEY_UP	0403	Up Arrow key
KEY_LEFT	0404	Left Arrow key
KEY_RIGHT	0405	Right Arrow key
KEY_HOME	0406	Home Up (to upper left corner) key
KEY_BACKSPACE	0407	Backspace key (unreliable)
KEY_F0	0410	Function Key 0
. . .	. . .	. . .
KEY_F(n)	0410+(n)	Function Key (n)
KEY_DL	0510	Delete Line key
KEY_IL	0511	Insert Line key
KEY_DC	0512	Delete Character key
KEY_IC	0513	Insert Character or Enter Insert Mode key
KEY_EIC	0514	Exit Insert-character Mode Key
KEY_CLEAR	0515	Clear Screen key
KEY_EOS	0516	Clear to End-of-Screen key
KEY_EOL	0517	Clear to End-of-line key
KEY_SF	0520	Scroll Forward 1 Line
KEY_SR	0521	Scroll Reverse (backwards) 1 line
KEY_NPAGE	0522	Next Page key
KEY_PPAGE	0523	Previous Page key
KEY_STAB	0524	Set Tab key
KEY_CTAB	0525	Clear Tab key
KEY_CATAB	0526	Clear All Tabs key
KEY_ENTER	0527	Enter or Send key (unreliable)
KEY_SRESET	0530	Soft (partial) Reset key (unreliable)
KEY_RESET	0531	Reset or Hard Reset key (unreliable)
KEY_PRINT	0532	Print or Copy key
KEY_LL	0533	Home Down (to lower left) key

Keyboard Input Program Example

The example program **show** at the end of this tutorial contains an example use of *getch*. **Show** displays a file, one screen at a time; advancing to the next page each time the space bar is pressed. Nearly any exercise for *curses* can be created by constructing an input file that contains a series of 24-line pages, each page varying slightly from the previous page.

In the **show** program:

- *cbreak* is used so that only the space bar need be pressed (use of **RETURN** is unnecessary).
- *Noecho* is used to prevent the character transmitted by the space bar from being echoed during *refresh* calls so that echoed character does not alter vertical alignment of the display during refresh operations.
- *nonl* is called to enable additional screen optimization.
- *idlok* allows insert and delete line. This capability helps streamline updates in some instances, but produces undesirable effects in other cases. Therefore an option to allow or disallow the capability has been provided.
- *clrtoeol* clears from cursor to end of current line.
- *clrtobot* clears from cursor to end of current line, then clears all subsequent lines to the bottom of the screen.

Display Highlighting

curses supports nine highlighting attributes, each of which has a corresponding 16-bit integer constant named in the include file <**curses.h**>. The value of each constant is selected such that one bit (corresponding to the attribute) in the 16-bit integer is set while all other bits are cleared. Here is a list of the nine attributes with their corresponding enable-bit positions. The name and octal value of each constant is also shown (note that only six digits are needed to represent the 16-bit value; the leading zero identifies the constant as an octal value).

- Standout (bit 7):
  `A_STANDOUT` = 0000200
- Underlining (bit 8):
  `A_UNDERLINE` = 0000400
- Inverse Video (bit 9):
  `A_REVERSE` = 0001000)
- Blinking (bit 10):
  `A_BLINK` = 0002000
- Dim (bit 11):
  `A_DIM` = 0004000
- Bold (bit 12):
  `A_BOLD` = 0010000
- Invisible (bit 13):
  `A_INVIS` = 0020000
- No print or display (bit 14):
  `A_PROTECT` = 0040000
- Alternate Character Set (bit 15):
  `A_ALTCHARSET` = 0100000

addch and *waddch* store window characters as 16-bit data words where the lower seven bits (0-6) of each word contain the character code and the upper nine bits (7-15), when set, enable the corresponding display highlighting attributes when that character is displayed on a terminal. Each attribute bit corresponds to one of the highlighting functions listed above. Obviously, any selected highlighting feature that is not available on a given terminal cannot be used even though the capability is standard fare for *curses*. However, when a requested attribute is not available on a given terminal, *curses* attempts to identify and use a suitable substitute. If none is possible, the attribute is ignored.

Three other constants in <**curses.h**> are also useful:

- `A_NORMAL` (value = 0000000) can be used as an argument for *attrset* to disable all attributes. *attrset(A_NORMAL)* is equivalent to *attrset(0)*, but more descriptive.
- `A_ATTRIBUTES` has an octal value of 0177600. It can be used in a bit-level logical AND to remove character bits, isolating the attributes attached to a given character.
- `A_CHARTEXT` has an octal value of 0000177. It is useful in a bit-level logical AND to discard all except the lower seven bits of the data word; in effect, separating the character from its highlighting attributes.

curses maintains a set of **current attributes** for each window. Whenever text is being placed in a given window by the program, the current attribute bits for the selected window are added to each character of text data, forming a 16-bit word for each character handled. To select a specific combination of attributes, a program call to *attrset* (or *attron*) with new attribute values must precede text output to the window. This can be used to enable one or more attributes when all were previously disabled, disable all currently enabled attributes (*attrset(0)*), or change the current set to any other new current set.

To enable one or more attributes in the current set without altering other active or inactive attributes, call *attron*. A call to *attroff* performs the opposite function, disabling the selected attributes without disturbing any other attributes in the current set.

curses always uses current attribute values, so a call to *attrset*, *attron*, or *attroff* (or their related window functions) must be used whenever you begin, end, or change any selected highlighting option. Here is an example program segment that illustrates how to set a word in boldface then restore normal display attributes for remaining text:

```
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
. . .
refresh();
```

In this example, the space characters before and after the word **boldface** are included in text blocks outside (before and after) the *attrset* calls. This technique prevents *curses* from applying display highlights to the spaces, thus avoiding possible undesirable effects; especially in situations where *curses* attempts to substitute an alternative for unavailable highlighting features.

The attribute `A_STANDOUT` offers unique program flexibility. In many interactive programs, displayed text needs to be enhanced to attract attention. However, it is not critical that the text be displayed with specific attributes. Many multi-terminal systems contain various terminal models that do not support identical highlighting features. For versatility, `A_STANDOUT` uses the terminal characteristics stored in the *terminfo* data base to determine the most pleasing highlighting feature available on the terminal being addressed (usually bold or inverse video), then uses that feature when sending corresponding text to the selected window on the terminal display screen. Two functions, *standout()* and *standend()* are provided so you can conveniently enable and disable `A_STANDOUT` highlighting.

attrset can be used to select only one (such as A_BOLD, shown in the earlier example in this section) or multiple attributes (such as A_REVERSE and A_BLINK for blinking inverse video). To change only one attribute or a certain combination of attributes while leaving the others undisturbed, use *attron()* and *attroff()*.

The example program **highlight** at the end of this tutorial demonstrates typical use of attributes. The program uses a text file as input, and embedded escape sequences in the file to control attributes. In the example program, \U enables underlining, \B selects bold, and \N restores normal text. An call to *scrollok* allows the terminal to scroll if the text file exceeds the capacity of a single display screen. When *scrollok* is active, if any text extends beyond the lower screen boundary, *curses* automatically scrolls the internally stored window up one line, then calls *refresh* to update the terminal display screen each time a line of input text exceeds the lower screen boundary. The scrolling process continues until end-of-file is reached on the input file.

The **highlight** program comes about as close to being a filter as is possible with *curses*. It is not a true filter because *curses* interacts directly with the terminal screen. *curses*' ability to optimize interaction between HP-UX programs and terminals is inherently linked to its direct monitoring of the current CRT screen and the windows where display text is being held for output through *refresh* operations. This capability requires that *curses* clear the screen as part of the first *refresh* operation so that it has a known beginning reference condition, then maintain a continually up-to-date data structure that reflects current screen contents and cursor location.

Multiple Windows

A window is a data structure that represents all or part of the CRT display screen. It contains a two-dimensional array of 16-bit character data words, a cursor, a set of current attributes, and several flags. Each 16-bit character data word contains:

- A 7-bit character code in the lower seven bits, and
- A 9-bit video highlighting code in the upper nine bits. Each bit enables one of nine attributes when set, each attribute represented by one of the respective bits.

curses provides a full-screen window called **stdscr** and a set of functions that use **stdscr**. Another window called **curscr** that represents the current physical display screen is also provided.

It is important that you clearly understand that a window is only a data structure. Use of more than one window does not imply the presence of more than one terminal, nor does it involve more than one process. A window is nothing more than a data object that can be copied to all or part of the terminal screen. *curses*, as presently implemented, cannot handle windows that are larger than the available display screen (use pads for such applications).

Pads

Pads are data structures that are essentially identical to windows, except that they can be larger than the available terminal screen size, and, as a result, must be handled differently. For example, a special refresh function is required that knows how to transfer only a specified part of the total pad area to the current screen instead of the entire pad. Other window operations do not depend on the size of the structure, so they can treat windows and pads identically. In such instances, a single function supports pads and windows (such as *addch*, *delwin*, and similar functions).

Creating Windows

Additional windows can be created so that the applications program can maintain several different screen images. Images can then be alternated under program control as needs dictate. Windows can be useful in editors, games, and other applications such as when handling interactive processes involving multiple users on multiple terminals.

Overlapping windows can also be constructed so that changes to one window are easily copied onto the overlapping area of the second. Several *curses* routines have been provided specifically to handle such cases. *overlay* and *overwrite* copy one window onto the second, each handling the copy operation differently. *wrefresh* can be used to refresh the terminal screen, but in some cases it is operations that are equivalent to refresh, but which do not update the screen. This is done by using a series of calls to *wnoutrefresh* (or its equivalent for pads), followed by a single *doupdate* that copies the series of refreshes onto the physical screen in a single operation. This is readily provided because *refresh* is really a call to *wnoutrefresh* followed by a call to *doupdate*.

To create a new window, use the function:

newwin(`lines, cols, begin_row, begin_col`*)*

The newwin function call returns a pointer to the newly created window whose dimensions are `lines` by `cols`, and whose upper left-hand corner is positioned at screen location `begin_row` and `begin_col`.

Using Multiple Windows

All operations that affect *stdscr* have a corresponding function for use with other named windows. These functions' names are formed by adding the letter *w* in front of the *stdscr* function name. For example, the window function that corresponds to *addch* is named:

waddch(`mywin, c`*)*

To update the contents of the currently displayed screen to match the contents of a window, use:

wrefresh(`mywin`*)*

Whenever the boundaries of two or more windows overlap and thus conflict, the most recently refreshed window becomes the currently displayed screen in that area of the display area that is defined by the window size and location.

Any call to the non-w version of any window function (**stdscr** function calls) is converted to its w-prefixed counterpart. Thus, a call to *addch(*c*)* produces a call to *waddch(***stdscr**, *c)*, automatically adding the **stdscr** argument in the process.

The example program **window** at the end of this tutorial shows how windowing can be handled. The main display is kept in *stdscr*. When the user wants to put something else on the screen, a new window is created that covers part of the screen. A call to *wrefresh* on that window causes the window to be written over *stdscr* on the display screen. A subsequent call to *refresh* on *stdscr* causes the original window to be fully restored to the screen, eliminating the temporarily displayed window.

Examine the *touchwin* calls in **window** that precede refresh calls on overlapping windows. *touchwin* calls prevent optimization by *curses*, thus forcing *wrefresh* to completely overwrite the entire window area on the physical screen (previously displayed data is thus erased in the window area only). In some situations, if the *touchwin* call is omitted, only part of the window is written and existing information from a previous window may remain in the newly written window area.

For improved screen addressability, a set of move functions are available in conjunction with most common window functions. They produce a call to *move* before the other function is called, so that the cursor can be relocated before the window function is executed. Here are some examples:

- *mvaddch(row,col,ch)* is equivalent to *move(row,col); addch(ch)*
- *mvwaddch(row,col,win,ch)* is equivalent to *wmove(win,row,col); waddch(win,ch)*.

Refer to the *curses* routines section of this tutorial for more detailed descriptions of the window routines and their related move functions.

Subwindows

Subwindows can be created within any existing window or pad. Subwindows are identical to normal windows except that the subwindow's character data structure occupies the same memory locations as the corresponding character positions in the main window. This means that whenever a character is placed in a subwindow, the main window automatically contains the same character in the same location with the same highlighting attributes. In fact, as a result of shared character storage, any character stored in the character array automatically receives the current attributes for the window or subwindow through which it was stored, regardless of how many subwindows overlap the storage location. This feature greatly simplifies combining windows in a single display for some types of applications.

Each subwindow has its own cursor location, can be configured with a soft scrolling region, and generally has the same capabilities as any normal window, but, except for shared character storage, is completely independent of the original window it is associated with. Because of shared character data structures, *curses* does not allow deletion of any window (*delwin(win)* or pad that has one or more undeleted subwindows.

If subwindows are created within a pad, care must be exercised in the choice of correct refresh functions and other program characteristics to ensure correct data handling.

Multiple Terminals

curses can produce simultaneous output on multiple terminals. This capability is useful in single-process programs that access a common data base such as multi-player games. Output to multiple terminals is a complex issue, and *curses* does not solve all of the related programming problems. For example, it is the program's responsibility to determine the special file name for each terminal line and what type of terminal is connected to that line. The normal method, checking the environment variable `$TERM`, does not work because each process can only examine its own environment. Another issue that must be addressed is the case of multiple programs reading data from a single terminal line, a situation that produces race conditions which must be avoided because a program that wants to take over a terminal cannot arbitrarily stop whatever program is currently running on that terminal (particularly where security considerations make this action inappropriate, though it is appropriate for some applications such as inter-terminal communication programs).

Race conditions may or may not be a problem, depending on the overall relationships of running programs and processes. For example, if a *curses* program is looking for input from a terminal, there **must** be no other program looking for input from the same terminal (such as a shell). On the other hand, if two programs are sending output to the same terminal at the same time, the result is usually no worse than an unusable screen display. In any event, for interaction with the terminal to flow smoothly, conflicts in terminal access must be prevented.

A typical solution requires the user logged onto each terminal line to run a program that notifies the master program that the user is interested in joining the master program. The master program is given the notification program's process id, the name of the tty link, and the type of terminal being used. The notification program then goes to sleep until the master program finishes. During termination, the master program wakes up the notification program and all programs exit.

curses handles multiple terminals by always having a **current terminal**. All function calls always pertain to the current terminal. The master program should set up each terminal, saving a reference (pointer) to the terminal in its own variables. When it is ready to interact with a given terminal, the master program should set the current terminal (use *set_term*) according to program needs, then use ordinary *curses* routines.

Terminal references have type `struct screen *`. To initialize a new terminal, call *newterm(type,fd)*. *newterm* returns a screen reference to the terminal being set up. `type` is a character string that names the kind of terminal being used. `fd` is a stdio file descriptor to be used for input and output to the terminal (if only output is needed, the file can be opened for output only). The *newterm* call replaces the normal call to *initscr*.

To select a new current terminal, call *set_term(sp)* where **sp** is the screen reference returned by *newterm* for the terminal being selected. *set_term* returns a screen reference to the previous terminal.

A full set of windows and options must be maintained for each terminal according to program needs. Each terminal must be initialized separately with its own *newterm* call. Options such as *cbreak* and *noecho*, and functions such as *endwin* and *refresh* must be set (or called) separately for each terminal. Here is a typical scenario for sending a message to each terminal:

```
for (i=0; i <nterm; i++) {
    set_term(terms[i]);
    mvaddstr(0,0,"Important message");
    refresh();
}
```

The sample program **two** at the end of this tutorial contains a full example of how this technique is implemented. The program pages through a file, showing one page to the first terminal; the next page to the second. It then waits for a space character to be typed on either terminal, then sends the next page to the terminal that sent the space character. Each terminal has to be put into nodelay mode separately. Multiplexing is currently not implemented in *curses*(3X), so it is necessary to busy wait or call *sleep(1);* between each check for keyboard input. **two** waits one second between checks for available terminal keyboard characters.

two is only a simple example of two-terminal *curses*. It does not handle notification as described above; instead, it requires the name and type of the second terminal on the program procedure line. As written, **two** requires that the command *sleep 100000* be typed on the second terminal to put it to sleep while the program runs, and the the first-terminal user must have read and write permission on the second terminal.

Low-Level Terminfo Usage

Some programs need access to lower-level primitives than those offered by *curses*. For such programs, the ***terminfo*-level** interface is provided. This interface does not manage the CRT screen, but gives programs access to strings and capabilities that can be used to manipulate the terminal.

Use of *terminfo*-level routines is discouraged. Whenever possible, higher-level *curses* routines should be used instead, in order to maintain portability to other systems and handle a wider variety of terminal types. *curses* takes care of all of the anomolies, glitches, and personality defects present in physical terminals, but at the *terminfo* level they must be dealt with in the program. Also, there is no guarantee that the *terminfo* interface will not change with new releases of HP-UX, nor that it will be compatible with previous HP-UX releases.

There are two circumstances where use of *terminfo* routines is appropriate. On instance is where a special-purpose program sends a special string to the terminal (such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line). The second is when writing a filter. A typical filter performs one transformation on the input stream without clearing the screen or addressing the cursor. If this transformation is terminal-dependent and clearing the screen is inappropriate, *terminfo* routines are preferred.

A program written at the *terminfo* level uses the framework shown here:

```
#include <curses.h>
#include <term.h>
. . .
   setupterm(0,1,0);
   . . .
   putp(clear_screen);
   . . .
   reset_shell_mode();
   exit(0);
```

The call to *setupterm* handles initialization (*setupterm(0,1,0)* invokes reasonable defaults). If *setupterm* cannot determine the terminal type, it prints an error message and exits. The calling program should call *reset_shell_mode* before exiting.

Global variables with such names as *clear_screen* and *cursor_address* are defined during the call to *setupterm*. When outputting these variables, use calls to *putp* or *tputs* for better programmer control during output. Global variable strings should not be output to the terminal through *printf* because they contain padding information that must be processed. A program (*such as printf*) that transmits unprocessed strings will fail on terminals that require padding or use Xon/Xoff flow-control protocol.

Higher-level routines described previously are not available at the *terminfo* level. The programmer must determine output needs and structure programs accordingly. For a list of *terminfo* capabilities and their descriptions, see *terminfo(5)* in the *HP-UX Reference*.

The example program **termhl** at the end of this tutorial shows simple use of *terminfo*. It is similar to **highlight**, but uses *terminfo* instead of *curses*. This version can be used as a filter. The strings used to enter bold and underline mode, and to disable all highlighting attributes are demonstrated.

The program was made more complex than necessary in order to illustrate several *terminfo* properties. For example, *vidattr* could have been used instead of directly outputting *enter_bold_mode*, *enter_underline_mode*, and *exit_attribute_mode*. In fact, the program could easily be made more robust by using *vidattr* because there are several ways to change video attributes. However, this program was structured only to illustrate typical use of *terminfo* routines.

The function *tputs(cap,affcnt,outc)* adds padding information to the capability **cap**. Some capabilities contain strings such as $<20>, which means to pad for 20 milliseconds. *tputs* adds enough pad characters to produce the desired delay. **cap** is the string capability to be output; **affcnt** is the number of lines affected by the output (for example, *insert_line* may have to copy all lines below the current line, and may require time proportional to the number of lines being copied). By convention, **affcnt** is 1 if no lines are affected rather than 0 because **affcnt** is multiplied by the amount of time required per item, and a zero time may be undesirable. **outc** is the name of a routine that is to be called with each character being sent.

In many simple programs, **affcnt** is set to 1, and **outc** just calls *putchar*. For such programs, the *terminfo* routine *putp(cap)* is a convenient abbreviation. The example program **termhl** could be simplified by using *putp*.

Note the special check for the *underline_char* capability. Some terminals, rather than having a code to start underlining and a code to stop underlining, use a code to underline the current character. **termhl** keeps track of the current mode, and outputs *underline_char*, if necessary, whenever the current character is to be underlined. Low-level details such as this are a major reason why *curses* routines are preferred over *terminfo* routines. *curses* takes care of all the different terminal keyboard and display functions and highlighting sequences instead of forcing such details onto the application program.

A Larger Example

The example program **editor** is a very simple screen editor that has been patterned after the *vi* editor and illustrates how *curses* can be used for such applications. **editor** uses *stdscr* as a buffer for simplicity, whereas a more useful editor would maintain a separate data structure for editing operations, then display the pertinent contents of that separate structure on the screen. **Editor**, as written, requires a file size equal to screen size. It also cannot handle lines longer than the screen, and has no provision for control characters in the file.

Several program characteristics are of interest. The routine that writes the file back to the file system shows how *mvinch* is used to retrieve characters from given window positions. The data structure used does not provide for keeping track of the number of characters in a line nor the number of lines in the file, so trailing blanks are eliminated when the file is written out.

editor uses built-in *curses* functions *insch*, *delch*, *insertln*, and *deleteln*. These functions behave much like deleting characters and lines.

The command interpreter accepts not only ASCII characters, but also special (non-typing) keys. This is important — a good program accepts both. Defining the keyboard so that every special key has its function defined on a normal typing key as well provides a desirable increase in flexibility. The benefit for new users, for example, is that they can use arrow keys without having to remember that the same functions are available on h, j, k, and l keys in the normal typing area. On the other hand, an experienced user may prefer to keep his fingers on the home typing row where he can work faster, so the typing key equivalent of special keys is appreciated. Handling both classes of keys also widens the variety of terminals the program can interact with because some terminals may not be equipped with arrow or other special keys on the keyboard. Providing an ASCII character synonym for each special keypad key provides better overall program and system flexibility, and makes the program more salable and easier to learn.

Note the call to *mvaddstr* in the input routine. *addstr* is roughly equivalent to the *fputs* function in C. Like *fputs*, *addstr* does not add a trailing newline. It is equivalent to a series of calls to *addch*, using the characters in the string. *mvaddstr* moves the current cursor position to the specified location in the window before writing the string into the data structure.

The control-L command demonstrates a feature that most programs using *curses* should include. Frequently, an independent program operating beyond the control of *curses* may write something to the terminal screen, or some other event such as line noise causes the physical screen to be altered without *curses* being notified. In such a case, **CTRL-L** can be used to clear and redraw the current screen at the user's request. This is accomplished by a call to *clearok(curscr)* which sets a flag that causes the next *refresh* to clear the screen. A call to *refresh* follows immediately

so that the screen is immediately redrawn using the data in *curscr* so that there is no wait for other program activities or completion of a pending keyboard input. There is also no loss of current screen data.

Note also the call to *flash()* which flashes the screen (unless the terminal has no flashing capability, in which case it rings the bell instead). Replacing the bell with the flashing capability is useful in environments where the sound of the bell is objectionable or distracting. Still, there may be instances where an audible signal is still needed for certain purposes, even in quiet environments. In such cases, the *beep ()* routine can still be called instead whenever a real beep is preferred. If *beep* is called and the terminal is not equipped to process the call, *curses* substitutes the *flash* in its place if possible, and vice versa. Thus, a terminal with no beep capability receives a flash sequence when beep is called; a terminal that cannot flash receives a beep sequence when flash is called. If the terminal has neither capability, ... well, ... some situations do present certain limitations — do without or get a different terminal because both are ignored in such a case.

Use of Escape in Program Control

Another important programming practice is terminating the input command with control-D; not escape. It is very tempting to use escape as a command because the escape key is one of the few special keys that is available on nearly every terminal keyboard (return and break are the only others). However, using escape as a separate key introduces an ambiguity which is handled by *curses* as follows:

Most terminals use sequences of characters beginning with an escape character (called escape sequences) to control the terminal. They also use similar escape sequences to transmit special keys to the computer. If the computer sees an escape character from the terminal, it cannot immediately determine whether the user pressed the escape key, or whether a special key was pressed instead. *curses* handles the ambiguity by waiting for up to one second. If another character is received within the one-second time limit, the escape and second character are compared with possible escape sequences. If the character pair represents a valid possibility, the wait is extended for up to one more second, or until the next character is received. The cycle continues until a valid special key sequence is completed or a character is received that could not be part of a valid sequence (or the time limit expires). While this technique works well most of the time, it is not foolproof. For example, a user could press the escape key then press one or more other keys that represent a valid sequence before the time limits expired (less than one second between successive key strokes). *curses* would then think that a special key had been pressed. Another disadvantage is the inevitable delay from the time a key is pressed until it can be processed by the program when an escape key is pressed, possibly even accidentally.

Many existing programs use escape as a fundamental command which often cannot be changed without incurring the wrath of a large group of users. Such programs cannot make use of special keys without dealing with the aforementioned ambiguity, and must, at best, resort to a timeout solution. The pathway is clear. When designing new programs and updating older ones, avoid using the escape key for program control whenever possible.

Program Routines

This and the following sections describe *curses* routines that are available to programmers. In this section, the routines are discussed in groups by function in the context of program operation. The next sections list *curses*, *terminfo*, and *termcap* compatibility routines alphabetically for easy reference, and each is discussed in greater detail. Both are helpful as tutorial and reference information, expanding on the information contained in the *curses(3X)* and *terminfo(5)* entries in the *HP-UX Reference*.

The *curses* routines discussed in this section operate on pads, windows, and subwindows. In general, windows and subwindows are treated identically by most routines. Subwindows share character data structures with the original window, but have their own cursor location and other non-character data structures. Unless indicated otherwise, all references to windows during discussion of window routines apply equally to windows and subwindows.

Program Structure Considerations

All programs using *curses* should include the file *<curses.h>* which defines several *curses* functions as macros and establishes needed global variables as well as the datatype `WINDOW` (window references are always of type `WINDOW *`). *curses* also defines the `WINDOW *` constants *stdscr* (the standard screen that is used as a default for all routines that interact with windows) and *curscr* (the current screen, used as a reference for low-level operations when updating the current display or clearing and redrawing a scrambled display). The integer constants `LINES` and `COLS` are defined, and contain values equal to the number of available lines and columns in the physical display. The constants `TRUE` and `FALSE` are also defined with the values 1 and 0, respectively. Two additional constants are defined; the values returned by most *curses* routines. `OK` is returned when the routine was able to successfully complete its assigned task. `ERR` indicates that an error occurred (such as an attempt to place the cursor outside a defined window boundary or create a window larger than the physical screen); thus, the task was not successfully completed.

The include file `<curses.h>` that must be specified at the beginning of the program automatically includes `<stdio.h>` and an appropriate tty driver interface file, presently `<termio.h>`. Including `<stdio.h>` again in a subsequent program statement is harmless though wasteful, but including a tty driver interface file could cause a fatal error if the file is not the same as the one selected by *curses*.

Any program that uses curses should include the loader option

```
-lcurses
```

in its makefile, whether the program operates at the *curses* or *terminfo* level. If the program only needs *curses*' screen output and optimization capabilities, and no non-default windows are involved, you can improve output speed and processing efficiency by restricting the program to the mini-curses package. Mini-curses is selected by using the compilation flag

```
-DMINICURSES
```

Routines supported by mini-curses are marked by asterisks in the complete list of *curses* routines at the beginning of the *curses* Routines section of this tutorial. They are also similarly marked in the *HP-UX Reference* under *curses(3X)*.

Terminal Initialization Routines

Program entry and exit states must be handled correctly to maintain system integrity and proper terminal operation. If the program interacts with only one user/terminal, *initscr* should be the first function call in the program. It sets up the necessary data structures and makes sure that terminal handling and screen clearing are properly initialized. The program should call *endwin* before terminating, ensuring that the terminal is restored to its original operating state and the cursor is placed in the lower left corner of the screen. *endwin* also dismantles data structures and other program entities that were created by *curses* and are no longer needed.

If the program must interact with multiple terminals during operation, *newterm* should be used for each terminal instead of the single call to *initscr*. *newterm* returns a variable of type `SCREEN *` which should be saved and used each time that terminal is referenced. Two file descriptors must be present, one for input, and one for output. Use *endwin* for each terminal prior to program termination to restore previous terminal states and dismantle data structures that were created by *curses* and are no longer needed. During program operation with multiple terminals, *set_term* is used to switch between terminals.

Another initialization function is *longname* which returns a pointer to a static area containing a verbose description of the current terminal upon completion of a call to *initscr*, *newterm*, or *setupterm*.

Option Setting Routines

These routines set up options within *curses*. Arguments specify the window to which the option applies, and the boolean flag which must be **TRUE** or **FALSE** (not 1 or 0) specifies whether the option is enabled or disabled. Default for all functions in this group is FALSE (disabled).

- *clearok(win, boolean_flag)*, when set, clears and redraws the entire screen on the next call to *refresh* or *wrefresh*.

- *idlok(win, boolean_flag)*, when set, allows *curses* to use the insert/delete line features of the terminal if they are available. This feature tends to be visually annoying if used in applications where it is not really needed. Insert/delete character capabilities are always considered by *curses*, and are not related to insert/delete line considerations.

- *keypad(win, boolean_flag)*, when set, enables handling of special keys from the terminal keyboard as single values instead of character sequences.

- *leaveok(win,boolean_flag)*, when set, allows *curses* to ignore cursor position and relocation at the end of an operation. This feature helps simplify program operation when the cursor is not used or cursor position is not important.

- *meta(win,boolean_flag)*, when set, handles characters from the *(getch)* function as 8-bit entities instead of the usual seven. However, this feature has no value if other programs and networks interacting with the data can only pass 7-bit characters.

  This feature is useful for applications where an extended non-text character set is needed and the terminal has a meta shift key available. Curses takes whatever measures are needed to handle the 8-bit input, including the use of raw mode, if necessary. In most cases, the character size is set to 8, parity checking disabled, and 8th-bit stripping is disabled. For the data to continue unaltered, all programs using it must also be capable of handling 8-bit character codes.

- *nodelay(win,boolean_flag)*, when set, makes *getch* a non-blocking call. When enabled, *getch* returns immediately with the value −1 if no input is ready. If not enabled, the program hangs until a terminal key is pressed.

- *intrflush(win,boolean_flag)*, when set, flushes all output in the tty driver queue if an interrupt key (interrupt, quit, or suspend, if available on the system) is pressed on the terminal keyboard. While this capability provides faster interrupt response, the flush destroys the representative relationship between *curscr* and the current physical display contents.

- *typeahead(file_descriptor)*, when set, enables typeahead for the specified file where **file_descriptor** is the terminal input file. A file descriptor value of zero selects *stdin*; −1 disables typeahead checking.

- *scrollok(win,boolean_flag)*, when set, enables scrolling on the specified window whenever the cursor position exceeds the lower boundary of the window (or scrolling region, if set). Boundary crossing results when a newline occurs on the bottom line or a character is placed in the last character position of the bottom line. If *scrollok* is enabled, the window or scrolling region is scrolled up one line, and a *refresh* operation is performed to update the terminal screen. *idlok* must be enabled on the terminal to get a physical scrolling effect on the visible display. If *scrollok* is disabled, the cursor is left on the bottom line, and no advances are allowed beyond the last character position.

- *setscrreg(top,bottom)* and *wsetscrreg(win,top,bottom)* are used to set software scrolling regions within a given window. If this option and *scrollok* are both active, the scrolling region is scrolled up one line and *refresh* is called to update the screen whenever the cursor position is moved beyond the lower limit of the scrolling region in the window. To get a scrolling effect on the terminal screen, *idlok* must also be enabled.

Terminal Configuration Routines

These routines are used to set or disable various operating modes that are supported by the terminal being used.

- *cbreak()* and *nocbreak()* enable and disable single-character mode. When *cbreak* is enabled, characters are received and processed from the terminal keyboard as they are typed. When *nocbreak* is active, characters are held by the tty driver until a newline key is received before making the line available to the program. Interrupt and flow control characters are not affected by either option. *cbreak* enabled is the preferred operating mode for most interactive programs. Default is *nocbreak* active.

- *echo()* and *noecho()* enable and disable, respectively, direct echoing of characters back to the terminal display as they are received by the tty driver. When *noecho()* is used, incoming characters are transfered directly to the program without returning them to the terminal display. *noecho* can also be used to process incoming text under program control, then echo selected characters to a controlled area of the screen or not echo at all.

- *nl()* and *nonl()* select or disable conversion of newline characters into a carriage-return line-feed sequence on output and conversion of incoming return character(s) into newlines. By disabling newline conversions, *curses* can use line-feed capability more effectively, resulting in better cursor motion.

- *raw()* and *noraw()* select or disable raw mode. Raw mode is similar to *cbreak* in that characters are passed to the program as they are typed, but interrupt, quit, and suspend characters are not interpreted, so they do not generate a signal. Raw mode also handles characters as 8-bit entities. BREAK handling is not affected.
- *resetty()* and *savetty()* restore and save tty modes. *savetty* saves the current state in a buffer. *resetty* restores the terminal to the state that was obtained by the last previous call to *savetty*.

Window Manipulation Routines

Window manipulation routines are used to create, move, and delete windows, subwindows, and pads, and perform certain other operations. *newwin*, *newpad*, and *subwin* create new structures. *delwin* deletes window, pad, and subwindow structures, and *mvwin* relocates a window to a different area within the physical screen boundary. *touchwin*, *overlay*, and *overwrite* affect optimization and character replacement during refresh and window copying operations as follows:

- *touchwin* forces the entire window to be rewritten to the screen during refresh.
- *overlay* copies non-blank characters from one window onto the overlapping area of another.
- *overwrite* overwrites all characters from one window onto the overlapping area of another.

Pad functions are related to window functions, with some differences. Pads are essentially the same as windows but they can be larger than the available screen size for added flexibility. When a pad is larger than the physical display space, only part of the pad can be displayed at any given time. Therefore pads cannot be directly transferred to the terminal screen by use of window *refresh* functions. Pad refresh functions are used instead so that the appropriate area of the pad can be specified for display.

When a new window, subwindow, or pad is created, the function returns a pointer that should be stored in a variable for later use when accessing the window or pad. The returned variable then becomes the `win` argument for writing to the window (or pad), deleting the window (or pad), and for other text and cursor operations that include `win` as an argument. Except for *prefresh*, *pnoutrefresh*, and *newpad*, all pad operations use the appropriate window function for all text and cursor manipulations and other pad/window activities.

Terminal Data Output Routines

All data transfers from a pad or window to the terminal display are handled by pad and window refresh/update functions:

- *refresh()* and *wrefresh(win)* transfer the contents of the default or specified window to the current screen window and to the terminal display.
- *doupdate()* and *wnoutrefresh (win)* are used to accumulate several window copy operations to the standard screen window by using multiple calls to *wnoutrefresh (win)*, then transferring the current screen window to the terminal screen by calling *doupdate()*.
- *prefresh(...)* and *pnoutrefresh(...)* are equivalent to *wrefresh* and *wnoutrefresh*, except that the pad and area within the pad are specified. *pnoutrefresh* is followed by the *doupdate* function that is normally used with window updates.

Window Writing Routines

Placing Text in the Window

These routines are used to write data in windows, subwindows, and pads. Only the root function is listed here. Other related functions are listed with the root function in the alphabetical *curses* Routines section later in this tutorial.

Routines in this group that use the `win` argument operate on the *stdscr* window if `win` is not specified. The cursor can be relocated before a function is executed by adding *mv* onto the beginning of the function name. This produces a *move(y,x)* or *wmove(win,y,x)* call on the default or specified window associated with the function, followed by a call to the remaining window writing routine. Row (y) and column (x) coordinates begin with (0,0) in the upper left-hand corner of the window or screen (not (1,1)). Use of the *mv* prefix was also discussed earlier. See the section, Using Multiple Windows.

- *move(y,x)* and *wmove(win,y,x)* move the cursor in the given window or pad. *move(y,x)* is equivalent to *wmove(stdscr,y,x)*.
- *addch(ch)* and related functions (see *curses* routines section for related functions) write a single character in the given window or pad. *mv* prefixed to the base function name causes the current cursor/character position to be changed to the specified `y,x` location before the character is placed. Cursor position after the placement is determined by the type of character written.
- *addstr(str)* and related functions place the specified string in the selected window. *mv* prefixed to the base function name causes the current cursor/character position to be changed to the specified `y,x` location before the string is placed. Cursor position after the placement is determined by the characters contained in the written string.

- *erase()* and *werase(win)* place blanks in the entire window or pad, destroying all previous window contents.
- *clear()* and *wclear(win)* are similar to *erase()*. They erase the window by filling it with blanks, but they also call *clearok()* which clears the terminal screen on the next *refresh()* for that window.
- *clrtoeol()* and *clrtobot()* and their related window/pad functions erase the specified window/pad from the present cursor position to the end of the cursor line or to the end of the window or pad, respectively.

Inserting and Deleting Text in the Window

The following routines are used to insert and delete lines and characters in the window. These operations are performed on the window only, and have no effect on the terminal at the time of execution.

- *delch()* and related window and move routines delete a single character from the current or specified new cursor position.
- *deleteln()* and *wdeleteln(win)* remove the current cursor line from the default or specified window.
- *insch(c)* and related routines insert the specified character in front of the current cursor position and move succeeding text appropriately to accommodate the new character.
- *insertln()* and *winsertln(win)* insert a blank line at the present cursor line position and move the existing cursor line (and subsequent lines) down one position. The bottom line in the window is lost. The inserted line becomes the new cursor line.

Formatted Output to the Window

printw is functionally similar to *printf* except the output is handled by *addch* which places the formatted data in the window.

Miscellaneous Window Operations

scroll(win) is used to scroll a given window up one line each time the function is called. *box(win,vert,hor)* uses the specified characters to draw a box around the specified window. When the window is boxed, the top and bottom rows and left and right columns in the window are no longer available for normal text use.

Window Data Input Routines

Two functions are available that are used to obtain data from a given window. *getyx(y,x)* is used to obtain the present cursor position for use by the program. *inch()* and related functions can be used to retrieve any character in a given window. The returned character includes video highlighting attribute bits, each of which is set or cleared according to the original highlighting attributes that were stored with the character when it was written to the window.

Terminal Data Input Routines

getch and its related window and move routines are the basic building block for all program input from the terminal. *getch* handles individual characters, one at a time, returning a character as a 16-bit integer value each time it returns from a call.

If echo is enabled, *getch* also places each character at the current cursor position in the window associated with the function and updates the terminal screen with a *refresh* on the window as the character is received and processed (the cursor is advanced as each character is written to the window). If *noecho* is active instead, input character(s) are not placed in the window.

getstr and its related functions generate a series of calls to *getch* to read an entire line, one character at a time, up to the terminating newline character. The line is stored in the specified string before *getstr* returns to the calling program.

scanw and its related functions perform formatted processing on the input line after it has been placed in a special buffer used by *getstr*. (If echo is enabled, the string is also placed in the associated window, but only the characters stored in the buffer are used by *scanw*. When scanning is complete, the processed results string results are placed in the specified **args** variables.

Video Highlighting Attribute Routines

Each character written into a window is stored as a 16-bit word. Seven bits contain the character code; the remaining nine bits control video highlighting. As each word is stored, the 7-bit character code is combined (through a bit-level logical OR operation) with the current set of nine video highlighting attributes to obtain the 16-bit result. Video attribute routines are used to construct the current attribute set that is used during character storage.

Highlighting attributes can be specified as a complete set by using *attrset* or *wattrset*. Using 0 (or **A_NORMAL**) as an argument for *attrset* disables all highlighting.

Highlighting can be altered from the present state by turning individual attributes on or off without altering the state of other attributes in the set. This is done with *attron*, *attroff*, *wattron*, and *wattroff*.

As characters are stored in a given window, the current attributes are attached to each character. To change highlighting, attributes must be changed before the next character is written to the window. When deciding where to change highlighting attributes, remember that highlighting applies to non-printing space and tab characters as well as visible characters.

standout and *standend* provide easy access to the **A_STANDOUT** attribute. *standout* is equivalent to a call to *attron(A_STANDOUT)*, and adds **A_STANDOUT** to the currently active set of attributes (if any are active). However, *standend* is not the opposite. *standend* is equivalent to *attrset(0)*, not *attroff(A_STANDOUT)*. Thus, a call to *standout* with underlining on would maintain underlining until another highlighting call. *standend*, on the other hand, would not only terminate the previous *standout* call, but would terminate underlining as well.

Attribute functions and arguments must be logically conceived. For example, *attron(A_NORMAL)* and *attroff(A_NORMAL), though executable, do nothing because all bits in A_NORMAL* are cleared (value is zero). The bit-level logical OR of *attron* has no effect (all bits zero), and *attroff* is ineffectual because *A_NORMAL* is inverted (all bits set to 1) before a bit-level logical AND is used to clear the selected highlighting attribute.

Miscellaneous Functions

beep/flash

beep() and *flash()* are used to signal the terminal operator. If the terminal does not support the called function, the other is substituted where possible. Thus a call to *beep* flashes the screen if the terminal has no beep capability; a call to *flash* produces a beep if no flashing video capability is available.

Portability Functions

Several functions have been included to aid portability of *curses* between various systems:

- *baudrate()* returns the terminal datacomm line speed as an integer baud rate value. The returned value can then be used for program and system configuration purposes.
- *erasechar()* returns the terminal erase character that has been chosen by the user. This character is used to cancel the last previous character. Interactive programs should include cancellation capabilities so users can correct typographical errors during keyboard inputs.
- *killchar()* is similar to the erase character, but cancels the entire line where the character appears.
- *flushinp()* discards any typeahead characters when an interrupt character is detected. This enables users to interrupt a series of commands or other activities that have accumulated in the typeahead buffer and terminate the current process without waiting for the typeahead queue to empty. Normally used for aborts, this function and the related program structure must be handled carefully to ensure proper termination of program processes before the program exits.

Delay Functions

Delay functions are not highly portable, but are frequently needed by programs that use *curses*, especially real-time interactive response programs. Use of these functions should be avoided where possible:

- *draino(ms)* is used to reduce the amount of data being held in the output queue. The main purpose of this function is to keep the program (and keyboard) from getting ahead of the screen. With careful program design, use of this function should be unnecessary in most cases.
- *napms(ms)* suspends program operation for a specified time. It is similar to *sleep*, but offers higher resolution (resolution varies, depending on system resources). *napms* uses a call to *select* for its time base reference.

curses Routines

curses supports the following functions. Those marked with an asterisk are also supported by *Mini-curses* (some unmarked routines might work, but are not officially supported by *Mini-curses*. Proceed at your own risk if you try them).

*addch(ch)**
*addstr(str)**
*attroff(attrs)**
*attron(attrs)**
*attrset(attrs)**
*baudrate()**
*beep()**
box(win,vert,hor)
*cbreak()**
*clear()**
clearok(win,boolean_flag)
clrtobot()
clrtoeol()
*delay_output(ms)**
delch()
deleteln()
delwin(win)
doupdate()
draino (ms)
*echo()**
*endwin()**
*erase()**
*erasechar()**
fixterm()
*flash()**
*flushinp()**
getch()
getstr(str)
gettmode()
getyx(win,y,x)
*has_ic()**
*has_il()**
*idlok(win,boolean_flag)**
*inch()**
*initscr()**
insch(c)
insertln()
intrflush(win,boolean_flag)
keypad(win,boolean_flag)
*killchar()**
leaveok(win,boolean_flag)
longname()
*meta(win,boolean_flag)**
*move(y,x)**
*mvaddch(y,x,ch)**
*mvaddstr(y,x,str)**
mvcur(oldrow,oldcol,
  newrow,newcol)
mvdelch(y,x)
mvgetch(y,x)
mvgetstr(y,x,str)
mvinch(y,x)
mvinsch(y,x,c)
mvprintw(y,x,fmt,args)
mvscanw(y,x,fmt,args)
mvwaddch(win,y,x,ch)
mvwaddstr(win,y,x,str)
mvwdelch(win,y,x)
mvwgetch(win,y,x)
mvwgetstr(win,y,x,str)
mvwin(win,beg_y,beg_x)
mvwinch(win,y,x)
mvwinsch(win,y,x,c)
mvwprintw(win,y,x,fmt,args)
mvwscanw(win,y,x,fmt,args)
napms(ms)
newpad(num_lines,num_cols)
*newterm(type,fdout,fdin)**
newwin(num_lines,num_cols,
  beg_y,beg_x)
*nl()**
*nocbreak()**
nodelay(win,boolean_flag)
*noecho()**
*nonl()**
*noraw()**
overlay(win1,win2)
overwrite(win1,win2)
pnoutrefresh(pad,pminrow,
  pmincol,sminrow,
  smincol,smaxrow,
  smaxcol)
prefresh(pad,pminrow,
  pmincol,sminrow,
  smincol,smaxrow,
  smaxcol)
printw(fmt,args)
*raw()**
*refresh()**
*resetterm()**
*resetty()**
*saveterm()**
*savetty()**
scanw(fmt,args)
scroll(win)
scrollok(win,boolean_flag)
setscrreg(t,b)
setterm(type)
setupterm(term,filenum,errret)
*set_term(new)**
*standend()**
*standout()**
subwin(orig_win,n_lines,
  n_cols,beg_y,beg_x)
touchwin(win)
traceoff()
traceon()
typeahead(fd)
unctrl(ch)
waddch(win,ch)
waddstr(win,str)
wattroff(win,attrs)
wattron(win,attrs)
wattrset(win,attrs)
wclear(win)
wclrtobot(win)
wclrtoeol(win)
wdelch(win,c)
wdeleteln(win)
werase(win)
wgetch(win)
wgetstr(win,str)
winch(win)
winsch(win,c)
winsertln(win)
wmove(win,y,x)
wnoutrefresh(win)
wprintw(win,fmt,args)
wrefresh(win)
wscanw(win,fmt,args)
wsetscrreg(win,t,b)
wstandend(win)
wstandout(win)

Description of Routines

The *curses* package includes the following functions. Function names that are associated with operations on user-specified windows contain a *w* or *mvw* prefix, and the window must be included as a parameter in the function call. If no *w* or *mvw* prefix is present, or if the window is not specified in the parameter set, the operation is performed on the default window *stdscr*. Programs that use the *curses* package are subject to the normal rules of C compiler statement syntax.

Routines are listed alphabetically by function keyword which is printed in slanted bold type. When two or more functions are related to a common keyword, the root keyword is listed in bold, followed by a list of related function names in normal italics. The individual related functions are also included elsewhere in the list with references back to the root keyword where a detailed explanation of all keywords related to the root keyword is located.

addch(*ch*)
waddch(win,ch)
mvaddch(y,x,ch)
mvwaddch(win,y,x,ch)

Places the character `ch` in the window at the current cursor position for that window, then advances the cursor to the next position. If `ch` is a tab, newline, backspace, the cursor is moved appropriately, but no text is altered. If `ch` is a control character other than tab, newline, or backspace, the character is drawn using ^x notation (where `x` is a printable character preceded by ^ to indicate a control character – see *unctrl(ch)*). If the character is placed at the right margin, an automatic newline is performed. At the bottom of the scrolling region, the region is scrolled up one line if *scrollok* is enabled.

The *ch* parameter is an integer; not a character. *addch* performs a bit-level logical OR between the 16-bit character and the current attributes if any are active. Highlighting of individual characters can also be handled by the program if the current attributes are all zero (disabled) by performing an equivalent bit-level logical OR operation between the 7-bit character code in bit positions 0 through 6 and selected video attribute bits in bit positions 7 through 15 to create a single 16-bit integer representing the character and its associated highlighting attributes. If no highlighting attributes for the window are currently active, any attributes added to the character by the program or already present from the source are preserved. If any are active, they are added to the character and any attached attributes without altering other attributes. Thus, you can copy text (including attributes) from one place to another with *inch* and *addch*.

addch is used with *stdscr* window; *waddch* with window *win*; *mvaddch* moves the cursor to row `Y`, column `X`, then places the character at that location; *mvwaddch* is identical to *mvaddch*, but operates on a specified window `win`. If `win` is not specified, default is to *stdscr*. All row and column references are relative to the upper left corner whose corner character position is represented by row 0, column 0.

addstr(*str*)
waddstr(win,str)
mvaddstr(y,x,str)
mvwaddstr(win,y,x,str)

Places the character string specified by `str` at the current cursor position (*addstr* and *waddstr*) or at the specified location in the window (*mvaddstr* and *mvwaddstr*). String placement consists of a series of character placements using the *addch* routine. `str` must be terminated by a null character.

***attroff**(attrs)*
wattroff(win,attrs)

Disables the specified video highlighting attributes without affecting other attributes. Any or all of the following attributes can be specified (multiple attributes must be separated by the C logical OR operator, |, which performs a bit-level logical OR on all attributes specified in the function call): **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS** (invisible), **A_PROTECT**, and **A_ALTCHARSET**.

***attron**(attrs)*
wattron(win,attrs)

Enables the specified video highlighting attributes without affecting other attributes. Any or all of the following attributes can be specified (multiple attributes must be separated by the C logical OR operator, |, which performs a bit-level logical OR on all attributes specified in the function call): **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS** (invisible), **A_PROTECT**, and **A_ALTCHARSET**.

***attrset**(attrs)*
wattrset(win,attrs)

Enables the specified video highlighting attributes, and disables all others. Any or all of the following attributes can be specified (multiple attributes must be separated by the C logical OR operator, |, which performs a bit-level logical OR on all attributes specified in the function call): **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS** (invisible), **A_PROTECT**, and **A_ALTCHARSET**. *attrset(0)*, *attrset(A_NORMAL)*, and *standend()* (or *standend(win)*) are equivalent functions that disable all attributes (normal display). See *standend()*.

***baudrate**()*

Returns the terminal serial I/O datacomm speed. The value returned is the integer baud rate (such as 9600) rather than a table index value (such as B9600). If the baud rate is External A or External B, the value −1 is returned instead.

***beep**()*

Used to signal the terminal user with an audible signal. If no audible signal is available on the terminal, the screen is flashed instead (see *flash()*). If neither capability is available, no output is sent to the terminal.

***box**(win,vert,hor)*

Draws a box around the specified window. `vert` specifies the character to be used for left and right columns; `hor` specifies the character for top and bottom rows. Usable window space is reduced by two lines and columns when a box is present.

***cbreak**()*
nocbreak()

These functions place the terminal in and out of `CBREAK` mode, respectively. When *cbreak* (character-mode operation) is active, each typed character is immediately available to the program. If disabled (*nocbreak*), the tty driver holds characters until a newline character is received, then releases the entire line to the program (line-mode operation). Interrupt and flow control characters are not affected by *cbreak*; default is *nocbreak*, but most interactive programs that use *curses* run with *cbreak* enabled.

***clear**()*
wclear(win)

Similar to *erase* and *werase*, but *clearok* is also called so that the terminal screen is cleared by the next call to *refresh* for that window. *clearok* sets a flag to clear the screen, blanks are placed in the window, and the next call to *refresh* outputs a screen clearing operation or blanks or both to the terminal, depending on terminal capabilities.

***clearok**(win,boolean_flag)*

If set, the next *wrefresh* call for the specified window clears and redraws the entire screen (instead of just the area represented by the specified window). If `win` specifies *curscr*, the next call to *wrefresh* for **any** window clears and redraws the entire screen. This is useful when current screen contents are uncertain, or in some cases for a more pleasing visual effect.

***clrtobot**()*
wclrtobot(win)

Clears all character positions from the current cursor position to the right margin, and all lines below the current cursor line to the end of the window.

***clrtoeol**()*
wclrtoeol(win)

Clears all character positions from the current cursor position to the right margin. The rest of the window remains undisturbed.

***delay_output**(ms)*

See *terminfo* routines in the next section of this tutorial.

***delch**()*
wdelch(win)
mvdelch(y,x)
mvwdelch(win,y,x)

The character at the present cursor position is deleted. All remaining characters on the line to the right of the deleted character are moved left one position. Other lines are not disturbed. The operation is performed only on the window, and does not use the terminal hardware delete-character feature because no terminal operation has been performed.

***deleteln**()*
wdeleteln(win)

The present cursor line is deleted. All remaining lines in the window below the cursor line are moved up one position, leaving a blank line at the bottom of the window. This window operation does not interact directly with the terminal when performed, so no terminal hardware delete-line feature is used.

delwin(*win*)

Deletes the specified window and releases all memory associated with it. If the window contains subwindows, all subwindows must be deleted first.

doupdate()
wnoutrefresh(win)
pnoutrefresh(pad,...)

wnoutrefresh (or *pnoutrefresh*) and *doupdate* essentially divide *wrefresh* into two independent functions that can be called separately for more efficient handling of multiple output operations to windows and pads. In normal operation, *wrefresh(win)* calls *wnoutrefresh(win)* to copy the named window to the virtual screen, then uses *doupdate* to update the physical screen to match the virtual screen. When outputting multiple windows, *wnoutrefresh(win)* can be used successively, once for each window; followed by a single *doupdate()* to transfer the new screen to the terminal, probably with fewer characters transmitted. *pnoutrefresh* is used similarly when writing to pads.

draino(*ms*)

Suspends program operation until the output queue has been reduced sufficiently ("drained") so that the remaining characters can be transmitted in not more than `ms` milliseconds. For example, *draino(50)* at 1200 baud would suspend program execution until no more than 6 characters remain to be sent (6 characters @ 1200 baud require about 50 ms transmit time). This routine is used to keep the program (and thus the keyboard) from getting ahead of the screen. If the operating system does not support the I/O controls (ioctls) that are needed to implement *draino*, the value ERR is returned; otherwise OK is returned.

echo()
noecho()

Enables or disables echoing of characters by *getch* through the specified window and back to the terminal as each character is typed on the keyboard and subsequently processed by *getch*. Default is *echo* (enabled). In some interactive programs, it is preferable to suppress echoing by *getch* (*noecho*), then let the program place incoming characters in a controlled area of the screen or not return them at all, as needs dictate.

endwin()

endwin should always be called before exiting from a *curses*-based program. Restores tty modes, places the cursor in the lower left corner of the terminal screen, resets the terminal into the proper non-visual mode, and removes data structures that are no longer needed by the exiting program.

erase()
werase(win)

Copies blanks to every character position in the specified or default window. As each blank is stored in the window, the highlighting attribute bits are set to zero (disabled).

erasechar()

Returns the user's chosen erase character from the terminfo data base. The returned character should be interpreted by the program as an "erase previous character" command whenever it is received from the terminal.

fixterm()

Restores the current terminal to the state it was in prior to the most recent call to *resetterm()*. State information stored by the most recent previous call to *saveterm()* provides the needed restoration information. See *resetterm()*.

flash()

Used to signal the terminal user by flashing the screen. If the terminal has no screen flashing feature, the audible signal is sounded instead (see *beep()*). If neither capability is available, no output is sent to the terminal.

flushinp()

Discards any typeahead characters in the typeahead buffer (characters that have been typed on the terminal but are still waiting to be handled by the program.

getch()
wgetch(win)
mvgetch()
mvwgetch(win)

Takes a character from the terminal keyboard input buffer as a 16-bit integer, processes it, and returns it to the program as a 16-bit integer. Character processing and return conditions vary as follows:

If *mv* is placed in front of *getch* or *wgetch*, the cursor position for the selected window is moved to the specified location which becomes the new current cursor position. This operation is completed before any character processing begins.

If *echo* is active and the character is a normal typing character (keypad and meta characters are discussed later), the character is placed in the current cursor position by a call to *waddch* from *getch*. During character placement in the window, a bit-level logical OR in *waddch* attaches current highlighting attributes to the character. *waddch* is followed immediately by a call to *wrefresh* which updates the terminal screen with the echo character.

If an escape character is received, special timeouts are set up to determine whether the character is part of a multiple-character keypad sequence. See Use of Escape in Program Control topic earlier in this tutorial for a detailed discussion of how escape is handled.

If **meta** is enabled and the character is not a keypad sequence, the 16-bit input character is logical ANDed with octal 0377 to mask the upper bits to zero and return an 8-bit text character value. The eighth bit interferes with the *A_STANDOUT* highlighting attribute bit in the same position, so *noecho* is usually chosen for programs that operate with meta active.

If **meta** is not enabled, text characters are logical ANDed with octal 0177 to mask the upper bits to zero and return a 7-bit character value. Echoing is handled in the normal manner if enabled.

If **keypad** is **not enabled**, function key sequences are treated as individual characters and handled as normal text.

If **keypad** is **enabled**, each function key sequence (usually an escape sequence) is handled as a single-character keycode which is assigned a 16-bit integer value in a range beginning at 0401 (octal) and a name that starts with *KEY_* (a complete list of keypad character value and name definitions is included in the keypad discussion near the beginning of this tutorial). The character value is **not** placed in the window for echoing, even if echo is enabled.

If **nodelay** is active: if no input is available in the keyboard input buffer when *getch* is called, *getch* returns with the value −1 and no other action is taken. If *nodelay* is not active, the program hangs until text is available in the buffer. Depending on the current ***cbreak*** setting, text is made available to the program as each character is received (*cbreak*), or incoming characters are held by the tty driver until a newline is received then they are made available to the program (*nocbreak*).

getstr(*str*)
wgetstr(win,str)
mvgetstr(y,x,str)
mvwgetstr(win,y,x,str)

This routine is used to input an entire line from the terminal. It is equivalent to *getch*, except that it handles an entire string instead of single characters. Handling of each character is identical to *getch* except that text and meta characters are packed into the string variable *str* instead of being returned to the program as individual 16-bit integers. Keypad characters (except for kill, erase, key_left (left arrow), and backspace) are not recognized and cannot be handled through *getstr*.

During execution, *getstr* generates a series of calls to *getch* until a newline is received, at which time it returns. The 16-bit integers returned by successive calls to *getch* are stripped of their unneeded upper bits (except recognized keypad keys) before packing into a string variable beginning at the location identified by the character pointer *str*.

If **echo** is enabled, incoming string characters are also placed in the associated window (by *getch*) as they are received and processed, and echoed to the terminal (by *refresh*). If *noecho* is active, characters are not placed in the window; they are only placed in *str*.

gettmode()

(Get tty mode). Dummy entry point. Performs no useful function.

getyx(*win,y,x*)

Places the current cursor position of the specified window in the specified two integer variables y and x. This is a macro, so no & is necessary.

has_ic()

Returns a value indicating whether or not the terminal has insert/delete character capability. Zero value indicates the capability is not present; non-zero: capability present.

has_il()

Returns a value indicating whether or not the terminal has insert/delete line capability. Zero value indicates the capability is not present; non-zero: capability present.

idlok(*win,boolean_flag*)

Insert and Delete Line OK. If enabled, *curses* can use hardware insert/delete line capabilities when the terminal is so equipped. If disabled, *curses* does not use the capability. Use only when the program requires it (such as a screen editor). *idlok* is disabled by default because it tends to be annoying when used in applications where it is not really needed. If insert/delete line cannot be used, *curses* redraws changed portions of all lines that do not match the desired result.

inch()
winch(win)
mvinch(y,x)
mvwinch(win,y,x)

Returns the character located at the current or specified position in the specified window as a 16-bit integer. If any attributes are set for that position, their values are included in the value returned. To extract only the character or the attributes, perform a bit-level logical AND on the returned value, using the predefined constant **A_CHARTEXT** (octal 0177) or **A_ATTRIBUTES** (octal 0177600).

initscr()

The first function called in *curses*-based programs. Determines terminal type, and initializes *curses* data structures as appropriate. Also sets indicators so that the first call to *refresh* clears the terminal screen and updates *curscr* to reflect the cleared screen.

insch(c)
winsch(win,c)
mvinsch(y,x,c)
mvwinsch(win,y,x,c)

Inserts the character (byte, usually a 7-bit code) specified by **c** at the current cursor position position or at the specified location in the standard or specified window (current attributes are attached during the placement operation). All characters beginning at the insertion location are moved right one position for the remainder of the line. If the line is full, the rightmost character is discarded. This does not interact with the terminal so no hardware insert-character feature is used.

insertln()
winsertln(win)

Inserts a blank line between the current cursor line and the line above it. The current line and subsequent lines of text in the window are moved down one position, and the blank line becomes the new current cursor line. The bottom line of text is discarded if it cannot fit inside the window. This is a window operation that does not interact with the terminal, so no hardware insert-line feature is used.

intrflush
(win,boolean_flag)

Causes tty driver queue to be flushed on interrupt. When enabled, an interrupt, quit, or suspend keypress from the terminal flushes all output from the tty driver queue, providing a faster response to the interrupt. However, *curses* loses its record of what is currently displayed on the screen when the interrupt occurs. Disabling the option prevents the flush. Default is flush enabled. Requires proper support from the underlying driver.

***keypad**(win,boolean_flag)* Enables keypad character handling for the user terminal associated with `win`. When true, the terminal operator can press any key that generates multiple-character sequences (such as a function key), and *getch* returns a single 16-bit integer value representing the function key (the returned character must be handled as a 16-bit value). If *keypad* is disabled (default), *curses* handles keypad sequences as normal text. *keypad* also enables and disables keypad keys on the terminal if the terminal hardware is equipped to support such command sequences from the external computer.

***killchar**()* Returns the line-kill character chosen by the terminal user. This character, when typed by the user, is a command to the program to cancel the entire line being typed.

leaveok**(win,boolean_flag)* Upon completion of normal *refresh* operations (*leaveok* disabled) the terminal hardware cursor is placed at the current cursor location for the window being refreshed. A call to *leaveok(win,* **TRUE)* prior to *refresh* allows refresh operations to leave the terminal hardware cursor in any convenient position instead of updating it to the current window cursor position when refresh is finished. This is useful for applications where the cursor is not used because it reduces the need for cursor movements. When possible, the cursor is made invisible when *leaveok* is specified for the window. Once *leaveok* is set **TRUE** for a given window, it remains active for the duration of the program or until another call sets it **FALSE**.

***longname**()* Returns a pointer to a static area containing a verbose description of the current terminal. This static area is defined only after a call to *initscr*, *newterm*, or *setupterm*.

***meta**(win,boolean_flag)* When enabled, text characters are returned by *getch* as 8-bit character codes (masked by octal 0377) instead of 7-bit (masked by octal 0177) characters. Returns the value **OK** if the request succeeds; **ERR** if the terminal or system cannot handle 8-bit character codes.

meta is useful for extending the non-text command set in applications where the terminal has a meta shift key. *curses* takes whatever measures are necessary to arrange for 8-bit input. When *meta* is true, HP-UX sets datacomm configuration to 8-bit character length, no parity checking, and disables 8th-bit stripping. Remember that if any program or facility handling the data can only pass 7-bit codes or strips the 8th bit, 8-bit handling is not possible.

move(*y,x*)
wmove(win,y,x)

Places the cursor associated with the specified or default window at the specified row (`y`) and column (`x`) where the upper left corner of the window is row 0, column 0. The cursor is not moved on the display screen until a *refresh* or equivalent function is executed.

mvaddch(*y,x,ch*) — Same as *move(y,x); addch(ch)*. See *addch(ch)*.

mvaddstr(*y,x,str*) — Same as *move(y,x); addstr(str)*. See *addstr(str)*.

mvcur(*oldrow,oldcol, newrow,newcol*)

Optimally moves the cursor from (`oldrow`, `oldcol`) to (`newrow`, `newcol`). The user program is expected to keep track of the current cursor position. Unless a full-screen image is kept, *curses* must make pessimistic assumptions that sometimes result in less than optimal cursor motion. For example, if the cursor needs to be moved a few spaces to the right, the task could be accomplished by retransmitting the characters between the present and the desired position; but if *curses* cannot access the screen image, it cannot determine what those characters are.

mvdelch(*y,x*) — Same as *move(y,x); delch()*. See *delch()*.

mvgetch(*y,x*) — Same as *move(y,x); getch()*. See *getch()*.

mvgetstr(*y,x,str*) — Same as *move(y,x); getstr(str)*. See *getstr(str)*.

mvinch(*y,x*) — Same as *move(y,x); inch()*. See *inch()*.

mvinsch(*y,x,c*) — Same as *move(y,x); insch(c)*. See *insch(c)*.

mvprintw(*y,x,fmt,args*) — Same as *move(y,x); printw(fmt,args)*. See *printw(fmt,args)*.

mvscanw(*y,x,fmt,args*) — Same as *move(y,x); scanw(fmt,args)*. See *scanw(fmt,args)*.

mvwaddch(*win,y,x,ch*) — Same as *wmove(win,y,x); waddch(win,ch)*. See *addch(ch)*.

mvwaddstr(*win,y,x,str*) — Same as *wmove(win,y,x); waddstr(win,str)*. See *addstr(str)*.

mvwdelch(*win,y,x*) — Same as *wmove(win,y,x); addch(ch)*. See *delch()*.

mvwgetch(*win,y,x*) — Same as *wmove(win,y,x); wgetch(win)*. See *getch()*.

mvwgetstr(*win,y,x,str*) — Same as *wmove(win,y,x); wgetstr(win,str)*. See *getstr(str)*.

mvwin(*win,beg_y,beg_x*)

Moves the specified window so that the upper left-hand corner is located at character position (`beg_y`, `beg_x`). If the move causes any part of the relocated window to lie outside the physical screen boundary, the command is considered to be in error, and the window remains in its original location.

mvwinch*(win,y,x)* Same as *wmove(win,y,x); winch(win)*. See *inch()*.

mvwinsch*(win,y,x,c)* Same as *wmove(win,y,x); winsch(win,c)*. See *insch(c)*.

mvwprintw*(win,y,x, fmt,args)* Same as *wmove(win,y,x); wprintw(win,fmt,args)*. See *printw(fmt,args)*.

mvwscanw*(win,y,x, fmt,args)* Same as *wmove(win,y,x); wscanw(win,fmt,args)*. See *scanw(fmt,args)*.

napms*(ms)* Suspends program operation for `ms` milliseconds. *napms* is similar to *sleep*, but has higher resolution. The resolution actually provided depends on the resolution of available operating system facilities. If a resolution of at least 0.100 sec is not available, the routine rounds to the next higher second, calls sleep, and returns ERR. Otherwise the value OK is returned.

newpad*(num_lines, num_cols)* Creates a new **pad** data structure. A pad is similar to a window, but it is not restricted by physical screen size nor is it associated with a particular part of the screen. Pads are useful when a large window is needed and only part of the window will be displayed at any given time. Automatic refreshes from pads (such as scrolling or input echo) do not occur. Refresh cannot be used with a pad as an argument. Instead, the routines *prefresh* and *pnoutrefresh* are used. Pad refresh routines require additional parameters to specify what part of the pad to display, and where to display it on the screen.

newterm*(type,fpout,fpin)* Used instead of *initscr* in programs that output to more than one terminal. *newterm* should be called once for each terminal. It returns a variable of type `struct screen *` which should be saved for use as a reference to that terminal. Arguments are: a string defining the terminal type, a file pointer for the output file, and another for the input file if needed (interactive terminal).

newwin*(num_lines, num_cols,beg_y,beg_x)* Create a new window with the specified number of lines and columns whose upper left-hand corner is located at the specified row and column of the physical screen, and return a window pointer (the upper left-hand corner of the physical screen is row 0, column 0). If the number of lines and/or columns is specified as zero, the default value *LINES* minus `beg_y` and *COLS* minus `beg_x` is used instead. A screen buffer for the window is also created. To create a new full-screen window, use *newwin(0,0,0,0)*.

nl()
nonl()

Defines handling of newline characters. When enabled (*nl*), newline is translated into a carriage-return and line-feed on output, and carriage-return is translated into a newline character on input. *curses* initially enables newline, but if it is disabled by *nonl*, *curses* can make better use of line feed capability, resulting in faster cursor motion.

nocbreak()

See *cbreak()*.

nodelay
(win,boolean_flag)

Makes *getch* a non-blocking call. When enabled, if no input is ready, a call to *getch* returns −1. If disabled, *getch* hangs until a key is pressed.

noecho()

See *echo()*.

nonl()

See *nl()*.

noraw()

See *raw()*.

overlay*(win1,win2)*
overwrite(win1,win2)

Copies `win1` onto `win2` for all screen area where the two windows overlap. *overlay* copies only visible (non-blank) text, and does not disturb those `win2` character positions where `win1` is blank. *overwrite* copies all of overlapping `win1` onto `win2`, including blanks, thus destroying all original data in the overlapping area of `win2`.

overwrite*(win1,win2)*

See *overlay*.

pnoutrefresh*(pad, pminrow,pmincol, sminrow,smincol, smaxrow,smaxcol)*

See *prefresh*.

prefresh*(pad, pminrow,pmincol, sminrow,smincol, smaxrow,smaxcol)*

pnoutrefresh
(same parameters)

Analogous to *wrefresh* and *wnoutrefresh*, except that pads are involved instead of windows. Additional parameters specify what part of the pad and screen are to be used. **`pminrow`** and **`pmincol`** identify the upper left corner of the pad area to be displayed. **`sminrow`**, **`smincol`**, **`smaxrow`**, and **`smaxcol`** define the display boundaries on the physical screen. The lower right-hand corner of the pad area being displayed is calculated from the screen boundary parameters because both rectangles must be the same size. Both rectangles must lie completely within their respective structures.

printw(*fmt,args*)
wprintw(win,fmt,args)
mvprintw(y,x,fmt,args)
mvwprintw(win,y,x, fmt,args)

These commands are functionally equivalent to *printf*. Characters that would normally be output by *printf* are instead output by *waddch* on the associated window.

raw()
noraw()

Places the terminal in or out of raw mode. Raw mode is similar to *cbreak* mode in that characters are immediately passed to the user program as they are typed on the terminal keyboard, except that interrupt and quit characters are passed as normal text instead of generating a special interrupt signal. Raw mode handles all terminal I/O as 8-bit characters instead of 7. BREAK key behavior may vary, depending on the terminal.

refresh()
wrefresh(win)

These functions output window data to the terminal (other routines only manipulate data structures). *wrefresh* copies the named window to the physical screen on the terminal by using *wnoutrefresh(win)* followed by *doupdate()*, taking into account what is already on the screen in order to optimize the transfer. *refresh()* is similar, except it uses *stdscr* as the default screen. Unless *leaveok* is enabled, the cursor is placed at the location of the window cursor when the operation is complete.

resetterm()
saveterm()
fixterm()

resetterm restores the current terminal to the operating condition it was in when *curses* was started. The "current *curses* state" is saved by *saveterm()* for possible future use by *fixterm()*. *resetterm* and *fixterm* should be used in all shell escapes. Equivalent routines are also available at the *terminfo* level.

resetty()
savetty()

Restores (resets) the tty modes to those stored in the buffer by the last previous *savetty()* command. This means that only one set of states can be stored at any given time. See *savetty()*.

saveterm()

Preserves the current terminal *curses* state for possible future use by *fixterm*. See *resetterm()* and *fixterm()*.

savetty()

Saves the current state of the tty modes in a buffer for possible later use by *resetty()*. See *resetty()*.

scanw(*fmt,args*)
wscanw(win,fmt,args)
mvscanw(y,x,fmt,args)
mvwscanw(win, y,x,fmt,args)

Corresponds to *scanf(3S). Calls wgetstr* which inputs characters from the terminal and places them in a buffer until newline is received. When newline is received, the string in the buffer serves as input for the scan which processes the buffered string and places the result in the appropriate **args**. Uses *getch* for character input and echo handling.

scroll(*win*)

Scrolls the window up one line by moving the lines in the window data structure. As an optimization, if the window being scrolled is *stdscr*, and the scrolling region is the entire window, the physical screen is scrolled at the same time.

scrollok *(win,boolean_flag)*

Controls window handling when the cursor advances beyond the bottom boundary of the window or scrolling region due to a newline in the bottom line or a character placed in the last character position of the bottom line. If scrolling is disabled, the cursor is left on the bottom line (characters are accepted until the bottom line is full, but newlines are ignored). If the cursor crosses the bottom boundary while *scrollok* is enabled, a *wrefresh* is performed on the window, then the window and terminal are scrolled up one line. *idlok* must also be called before a physical scrolling effect can be produced on the terminal screen.

setscrreg(*t,b*)
wsetscrreg(win,t,b)

Sets up a software scrolling area in window `win` or *stdscr*. `t` and `b` are the top and bottom lines of the scrolling region (line 0 is the top line of the window). If this option and *scrollok* are both enabled, an attempt to move off the bottom margin causes all lines in the scrolling region to scroll up one line. Note that this process has nothing to do with the physical scrolling region capability that exists in some terminal has scrolling region or insert/delete line capabilities, they will probably be used by the output routines during refresh. *idlok* must be enabled before a scrolling effect can be produced on the terminal screen (see *scrollok*).

setterm(*type*)

Low-level interface used by old *curses* and included here for compatibility with earlier software.

setupterm(*term,filenum, errret*)

terminfo routine. See *terminfo* routines in the next section of this tutorial for description.

set_term(*new*) — Switches to a different terminal. The screen reference `new` becomes the new current terminal, and the function returns the previous terminal. All other calls affect only the current terminal. This function is used to handle multiple terminals interacting with a single program.

standend()
wstandend(win) — Equivalent to *attrset(0)* and *attrset(A_NORMAL)*. Turns off all video highlighting attributes for the default (*standend*) or specified (*wstandend*) window.

standout()
wstandout(win) — Equivalent to *attron(A_STANDOUT)*. Turns on the video highlighting attributes used for standout highlighting for the terminal being used. Does not alter other attributes in effect at the time. *standout* applies to the default window *stdscr*. *wstandout* affects the specified window.

subwin(*orig_win, num_lines,num_cols, beg_y,beg_x*) — Creates a new window containing the specified number of lines and columns within existing window `orig_win`. `beg_y` and `beg_x` specify the starting row and column position of the window on the physical screen (not relative to window `orig_win`). The subwindow uses that part of the main window character data storage structure that corresponds to its own area (each window maintains its own pointers, cursor location, and other items pertaining to window operation; only character storage is shared). Thus, the subwindow always contains character data (including highlighting attributes) that is identical to the data contained in the corresponding area of the original window, regardless of which window is the target of a write operation (highlighting bits are determined by the current attributes in effect for the window through which each character was stored). When using subwindows, it is often necessary to call *touchwin* before *refresh* in order to maintain correct display contents.

touchwin(*win*) — Discards optimization information on the specified window so that the entire window must be completely rewritten during refresh. This is sometimes necessary when using overlapping windows because changes to one window do not update the overlapping window structure in such a manner that a subsequent refresh operation can be handled correctly.

traceoff() — Dummy entry point. Performs no useful function.

traceon() — Dummy entry point. Performs no useful function.

typeahead*(fd)* Sets the file descriptor for typeahead check. `fd` is an integer obtained from *open* or *fileno*. Setting typeahead to −1 disables typeahead check. Default file descriptor is 0 (standard input). Typeahead is checked independently for each screen; for multiple interactive terminals, it should be set to the appropriate input for each screen. A call to *typeahead* always affects only the current screen.

unctrl*(ch)* Converts the character code represented by `ch` into a printable form if it is an unprintable control character. The converted character is returned as a two-byte character pair consisting of an alpha-numeric character preceded by ^ where (^) represents the control key, and the alpha-numeric character corresponds to the key that is normally pressed in conjunction with the keyboard **CTRL** key to produce the control character.

waddch*(win,ch)*	See *addch(ch)*.
waddstr*(win,str)*	See *addstr(str)*.
wattroff*(win,attrs)*	See *attroff(attrs)*.
wattron*(win,attrs)*	See *attron(attrs)*.
wattrset*(win,attrs)*	See *attrset(attrs)*.
wclear*(win)*	See *clear()*.
wcleartobot*(win)*	See *cleartobot()*.
wcleartoeol*(win)*	See *cleartoeol()*.
wdelch*(win)*	See *delch()*.
wdeleteln*(win)*	See *deleteln()*.
werase*(win)*	See *erase()*.
wgetch*(win)*	See *getch()*
wgetstr*(win,str)*	See *getstr(str)*
winch*(win)*	See *inch()*
winsch*(win,c)*	See *insch(c)*.
winsertln*(win)*	See *insertln()*.
wmove*(win,y,x)*	See *move(y,x)*.
wnoutrefresh*(win)*	See *doupdate()*.

wprintw(*win,fmt,args*)	See *printw(fmt,args)*.
wrefresh(*win*)	See *refresh()*. See also *doupdate()*.
wscanw(*win,fmt,args*)	See *scanw(fmt,args)*.
wsetscrreg(*win,t,b*)	See *setscrreg(t,b)*.
wstandend(*win*)	See *standend()*.
wstandout(*win*)	See *standout()*.

Terminfo Routines

delay_output*(ms)* Inserts a delay into the output stream for the specified number of milliseconds by inserting sufficient pad characters to effect the delay. This should not be used in place of a high-resolution *sleep*, but rather to slow down or hold off the output. Due to system buffering, it is unlikely that a delay can result in a process actually sleeping. `ms` should not exceed about 500 because of the large number of pad characters used to produce such delays.

putp*(str)* Outputs a string capability without use of an `affcnt` (see *tputs*). The string is sent to *putchar* with an `affcnt` of 1. It is used in simple applications that do not require the output processing capability of *tputs*.

setupterm*(term,filenum, errret)* Initializes the specified terminal. `term` is the character string representing the name or model of the terminal; `filenum` is the HP-UX file descriptor of the terminal being used for output; `errret` is a pointer to the integer in which a success/failure indication is returned. The values returned can be: 1 (initialize complete); −1 (*terminfo* data base not found); or 0 (no such terminal).

If 0 is given as the value of `term`, the default value of TERM is obtained from the enviroment. `errret` can be specified as 0 if no error code is wanted. If `errret` is default (0), and something goes wrong, `setupterm` prints an appropriate error message and exits rather than returning. Thus, a simple program can call *setupterm(0,1,0)* and not provide for initialization errors.

If the environment variable TERMINFO is set to a path name, *setupterm* checks for a compiled *terminfo* description of the terminal under that path before checking */usr/lib/terminfo*. Otherwise, only */usr/lib/terminfo* is checked.

setupterm uses `filenum` to check the tty driver mode bits, and changes any that might prevent correct operation of low-level *curses* routines. Tabs are not expanded into spaces because various terminals exhibit inconsistent uses of the tab character. If the HP-UX system is expanding tabs, *setupterm* removes the definition of the `tab` and `backtab` functions because they may not be set correctly in the terminal. Other system-dependent changes such as disabling a virtual terminal driver may also be made here, if deemed appropriate by *setupterm*.

ttytype (an array of characters) to the value of the list of names for the terminal in question. The list is obtained from the begining of the *terminfo* description.

Upon completion of *setupterm*, the global variable **cur_term** points to the current structure of terminal capabilities. A program can use two or more terminals at once by calling *setupterm* for each terminal, and saving and restoring **cur_term**.

nl() is enabled, so newlines are converted to carriage return-line feed sequences on output. Programs that use *cursor_down* or *scroll_forward* should avoid these two capabilities or disable the mode with *nonl()*. *setupterm* calls *reset_prog_mode* after any changes are made.

tparm*(instring,p1,p2,p3, p4,p5,p6,p7,p8,p9)*

Instantiates a parameterized string. Up to nine parameters can passed (in addition to the input string) that define what operations are to be performed on **instring** by *tparm*. The resultant string is suitable for output processing by *tput*.

tputs*(cp,affcnt,outc)*

Processes *terminfo(5)* capability strings for terminal devices. The padding specification, if present, is replaced by enough padding characters to produce the specified time delay. The resulting string is passed, one character at a time, to the routine *outc* which expects a single character parameter each time it is called. Often, *outc* simply calls *putchar* to complete its task. **cp** is the capability string, and **affcnt** is the number of units affected (such as lines or characters). For example, the **affcnt** for *insert_line* is the number of lines on the screen below the inserted line; that is, the number of lines that will have to be moved on the terminal. In certain cases, **affcnt** is used to determine the number of padding characters that must be created in the output string to produce the required delay(s), based on known terminal characteristics (obtained from the terminal identification data base).

vidattr*(attrs)*

Transmits the appropriate string to *stdout* to activate the specified video attributes which can include any or all of the following: **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_BLANK** (invisible), **A_PROTECT**, and **A_ALTCHARSET** (multiple attributes must be separated by the C logical OR operator |).

vidputs(*attrs,putc*) — Transmits the appropriate string to the terminal, activating the specified video highlighting attributes. `attrs` can include any or all of the following (multiple attributes must be separated by the C logical OR operator |): **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_BLANK** (invisible), **A_PROTECT**, and **A_ALTCHARSET**. `putc` is a *putchar-like* function. Previous highlighting attributes are preserved by this routine and restored upon return.

Termcap Compatibility Routines

Several routines have been included in *curses* that support programs written with calls to *termcap* routines. Calling parameters are the same as for equivalent *termcap* calls, but the routines are emulated using the *terminfo* data base. These routines may be removed in future releases of HP-UX.

tgetent(bp,name)	Obtains *termcap* entry for `name`
tgetflag(id)	Returns the boolean entry for `id`.
tgetnum(id)	Returns the numeric entry for `id`.
tgetstr(id,area)	Returns the string entry for `id` and places the result in `area`.
tgoto(cap,col,row)	Attaches `col` and `row` parameters to the capability `cap`.
tputs(cap,affcnt,fn)	Equivalent to the *terminfo* routine *tputs*. Parameters are identical for both cases.

Program Operation

This section describes how *curses* routines behave and how they are used in a typical programming environment.

Insert/Delete Line

The output optimization routines associated with *curses* use terminal hardware insert/delete line capabilities provided the routine

```
idlok(stdscr,TRUE);
```

has been called to enable the capability. By default, insert/delete line during refresh is disabled (`FALSE`); not for performance reasons (there is no speed penalty involved), but because experience has shown that not only is insert/delete line frequently not needed (especially in simple programs); it can sometimes be visually annoying when used by *curses*. Insert/delete character is **always** available to *curses* if it is supported by the terminal.

Additional Terminals

Curses can be used, even when absolute cursor addressing is not provided on the terminal, as long as the cursor can be moved from any location to any other location. *curses* considers available cursor control options such as local motions, parameterized motions, home, and carriage return.

curses is intended for use with full-duplex, alphamumeric, video display terminals. No attempt is made to handle half-duplex, synchronous, hard copy, or bitmapped terminals. Bitmapped terminals can be handled by programming the bitmapped terminal to emulate an ordinary alphanumeric terminal. This prevents *curses* from using the bitmap capabilities, but *curses* was not designed for bitmapping.

curses can also deal with terminals that have the "magic cookie" glitch in their display highlighting behavior. The term "magic cookie" means that changes in highlighting are controlled by storing a "magic cookie" character in a location on the screen. While this "cookie" takes up a space, preventing an exact implementation of what the programmer wanted, *curses* takes the extra character space into account, and moves part of the line to the right when necessary. In some cases, this unavoidably results in losing text along the right-hand edge of the screen, but *curses* compensates where possible by omitting extra spaces.

Multiple Terminals

Some applications require that text be displayed on more than one terminal at the same time from the same process. This is easily accomplished, even when the terminals are different types.

curses maintains all information about the current terminal in a global variable called

```
struct screen *SP;
```

Although the screen structure is hidden from the user, the C compiler accepts declarations of variables that are pointers. The user program should declare one screen pointer variable for each terminal that is to be handled. The routine:

```
struct screen *newterm(type,fdout,fdin)
```

sets up a new terminal of the specified `type` and output is handled through file descriptor `fdout`. This is comparable to the usual program call to *initscr* which is essentially equivalent to

```
newterm(getenv("TERM"),stdout)
```

A program that uses multiple terminals should call *newterm* for each terminal, and save the value returned as a reference to that terminal for other calls.

To change to a different terminal, call

```
set_term(term)
```

which returns the old value of variable `SP`. Do not assign to `SP` because certain other global variables must also be changed.

All *curses* routines always interact with the current terminal. *set_term* is used to change from one terminal to the next in a multi-terminal environment. When the program is ready to terminate, each terminal should be selected in turn by a call to *set_term*, then cleaned up with screen clearing and cursor locating routines, followed by a call to *endwin()* for that terminal. Repeat the sequence for each additional terminal used by the program. The example program **TWO** demonstrates the technique.

Video Highlighting

Video highlighting attributes can be displayed in any combination on terminals that support the various attribute capabilities. Each character position in screen data structures is allotted 16 bits: seven for the character code; the remaining nine for highlighting attributes, one bit per attribute. Each respective bit is associated with one of the following attributes: standout, underline, inverse video, blink, dim, bold, invisble, protect, and alternate character set. Standout selects the visually most pleasing highlighting method, and should be used by all programs that do not need a specific highlighting combination. Underlining, inverse video, blinking, dim, and bold are standard features on most popular terminals, though they are not usually all present on a single terminal (for example, no current terminal implements both bold and dim). Invisible means that visible characters are displayed as blanks for security reasons (such as when echoing passwords). Protected and Alternate Character Set are subject to the characteristics of the terminal being used. Invisible, protected, and alternate character set attributes are subject to change or substitution by *curses*, and should be avoided unless necessary.

When characters are stored, each character is combined with the **current attributes** variable associated with the window. The variable is formed by using one of the following routines:

attrset(attrs)	*wattrset(win,attrs)*
attron(attrs)	*wattron(win,attrs)*
attroff(attrs)	*wattroff(win,attrs)*
standout()	*wstandout(win)*
standend()	*wstandend(win)*

The following attributes can be specified in the `attrs` argument for corresponding attribute set/on/off routines.

A_STANDOUT	A_BLINK	A_INVIS
A_UNDERLINE	A_DIM	A_PROTECT
A_REVERSE	A_BOLD	A_ALTCHARSET

When specifying multiple attributes, they should be separated by the C logical OR operator (|). Thus, to specify blinking underline and disable all other attributes on the *stdscr* window, use `attrset(A_BLINK|A_UNDERLINE)`.

curses forms the current attributes word as follows:

- Each attribute (such as A_UNDERLINE) is stored as a 16-bit word where all bits are zero except the bit that represents the corresponding attribute in a stored character word (for example, 0000010000000000 controls blinking).

- All attributes forming the **attrs** argument are combined using the logical OR operator to create a single 16-bit word containing all attributes in the argument. For example, the three attribute words

  0000010000000000,
  0001000000000000, and
  0000001000000000 are combined to form
  0001011000000000 which identifies the new attributes.

- Three things can be done with the new attributes word. It can be used as the new current attributes (*attrset* or *wattrset*); or the new attributes can be added to any currently active attributes (*attron* or *wattron*), or deleted from the currently active attributes (*attroff* or *wattroff*).
- If *attrset* (or *wattrset*) was called, the routine stores the new attributes in the current attributes variable and returns. The previous set of current attributes is destroyed.
- If *attron* (or *wattron*) was called, the routine performs a logical OR of the current attributes with the new attributes, then places the result in the current attributes variable and returns. The revised current attributes variable contains all previously active attributes plus the new attributes.
- If *attroff* (or *wattroff*) was called, the routine inverts the new attributes, performs a logical AND on the inverted new attributes and the current attributes, then places the result in the current attributes variable and returns. The altered current attributes variable contains all previously active attributes except those specified in the call, which are now disabled.
- *standout* and *wstandout* obtain their highlighting attributes from the *terminfo* data base, then perform the same operation as *attron* prior to returning.
- *standend* and *wstandend* disable all attributes then return. They are equivalent to *attrset(0)* and *attrset(A_NORMAL)*.
- *attrset(0)* and *wattrset(win,0)* set the 16-bit current attributes variable value to zero which disables all attributes. **A_NORMAL** can be substituted for zero as an argument.

The preceding scenarios assume that the specified attributes are available on the current terminal. In every case, the *terminfo* data base is used to determine whether the selected attribute is present. If it is not, *curses* attempts to find a suitable substitute before forming the new attribute set. If the terminal has no highlighting capabilities, all highlighting commands are ignored.

Three other constants (defined in <curses.h>), in addition to the previously listed attributes are also available for program use if needed:

- `A_NORMAL` has the octal value 0000000, and can be used as an attribute argument for *attrset* to restore normal text display. *attrset(0)* is easier to type, but less descriptive. Both are equivalent.
- `A_ATTRIBUTES` has the octal value 0177600. It can be logically ANDed with a character data word to isolate the attribute bits and discard the character.
- `A_CHARTEXT` has the octal value 0000177. It can be logically ANDed with a character data word to isolate the character code and discard the attributes.

Special Keys

Most terminals have special keys, such as arrow keys, screen/line clearing keys, insert and delete line or character keys, and keys for user functions. The character sequences that such keys generate and send to the host computer vary from terminal to terminal. *curses* provides a convenient means for handling such keys through the use of *keypad* routines. Keypad capabilities are enabled by the call:

```
keypad(stdscr,TRUE)
```

during program initialization, or

```
keypad(win,TRUE)
```

when setting up and initializing other windows, as appropriate. When keypad is enabled, keypad character sequences are passed to the program by *getch*, but they are converted to special character values starting at 0401 octal (keypad character codes are listed in the keypad discussion early in this tutorial). Keypad codes are 16-bit values, and must not be stored in a *char* type variable because the upper bits must be preserved.

When keypad keys are used in a program, avoid using the escape key for program control because most keypad sequences begin with escape. If escape is used for program control, an ambiguity results that is not easily dealt with, and, at best, results in sluggish program response to all escape sequences as well as significant potential for incorrect program operation.

Scrolling Regions

Each window has a programmer-accessible scrolling region that is normally set to include the entire window. *curses* contains a routine that can be used to change the scrolling region to any location in the window by specifying the top and bottom margin lines. The routines are called by

```
setscrreg(top,bottom)
```

for the *stdscr* window, or

```
wsetscrreg(win,top,bottom)
```

for other windows. When the cursor advances beyond the bottom line in the region, all lines in the region are moved up one line (destroying the top line in the process) and a new line at the bottom of the region becomes the new cursor line. If scrolling has been enabled by a call to *scrollok* for that window, scrolling takes place, but only within the window boundary (if *scrollok* is not enabled, the cursor stays on the bottom line and no scrolling can occur). The scrolling region is a software feature only, and only causes a given window data structure to scroll. It may or may not translate to use of the hardware scrolling region that is featured on some terminals or hardware insert/delete line capabilities on the terminal.

Mini-Curses

All calls to *refresh* copy the current window to an internal screen image (*stdscr*). For simpler applications where window capabilities are not important and all operations can be handled by the standard screen, the screen output optimization capabilities of *curses* can be obtained through the low-level *curses* interface routines supported by *mini-curses*. Mini-*curses* is a subset of full *curses*, so any program that runs on the subset can also run on full *curses* without modification.

A complete list of commands is shown at the beginning of the *curses* commands section in this tutorial. Commands that are supported by *mini-curses* are marked with an asterisk (some that are not marked may also be accessible – if a program calls routines that are not, an error message showing undefined calls is produced by the compiler at compile time).

mini-curses routines are limited to commands that deal with the *stdscr* window. Certain other high-level functions that are convenient but not essential (such as *scanw*, *printw*, and *getch*) are not available, as well as all commands that begin with `w`. Low-level routines such as hardware insert/delete line and video attributes are supported, as are mode-setting routines such as *noecho*.

To access *mini-curses*, add **-DMINICURSES** to the CFLAGS in the makefile. If any routines are requested that are not available in *mini-curses*, an error diagnostic such as

```
Undefined:
m_getch
m_waddch
```

is listed to indicate that the program contains calls (in this case to *getch* and *waddch*) that cannot be linked because they are not available.

Remember that the preprocessor is involved in the implementation of *mini-curses*, so any programs that are compiled for use with *mini-curses* must be recompiled if they are to be used with full *curses*.

TTY Mode Functions

In addition to the save/restore functions *savetty()* and *resetty()*, other standard routines are provided by *curses* for entering and exiting normal tty mode.

- *resetterm()* restores the terminal to its state prior to *curses*' start-up.
- *fixterm* performs the equivalent of an *undo* on the previous *fixterm* on that terminal; it restores the "current curses mode" using the results of the most recent call to *saveterm()*.
- *endwin* automatically calls *resetterm*.
- Routines that handle control-Z (on systems that have process control) also use *resetterm()* and *fixterm()*.

Programs that use *curses* should use these routines before and after shell escapes, and also if the program has its own routines for dealing with control-Z. These routines are also available at the *terminfo* level.

Typeahead Check

When a user types something during a screen update, the update stops, pending a future update. This is useful when several keys are pressed in sequence, each of which produces a large amount of output. For example in a screen editor, the "forward screen" (or "next page") key draws the next screenful of text. If the key is pressed several times in rapid succession, rather than drawing several screens of text, *curses* cuts the updates short and only displays the last requested full screen. This feature is automatic, and cannot be disabled. It requires support by certain routines in the HP-UX operating system.

getstr

No matter whether echo is enabled or disabled, strings typed and input by *getstr* are echoed at the current cursor location. Erase and kill characters assigned by the user for his (or her) terminal are considered when handling input strings. Thus it is unnecessary for interactive programs to deal directly with erase, echo, and kill when processing a line of text from the terminal keyboard.

longname

The longname function does not require any arguments. It returns a pointer to a static storage area that contains the actual long (verbose) terminal name.

Nodelay Mode

The program call

```
nodelay(stdscr,TRUE)
```

puts the terminal in "no delay" mode. When nodelay is active, any call to getch returns the value −1 if there is nothing available for immediate input. This feature is helpful for real-time situations where a user is watching terminal screen outputs and presses a key when he wants to respond. For example, a program can be producing a text pattern on the screen while maintaining an open opportunity for the user to press certain keys to alter the output pattern, change cursor direction, or produce some other effect.

Example Programs

SCATTER

This program takes the first 23 lines from the standard input, then displays them in random order on the display terminal screen.

```
#include <curses.h>
#define    MAXLINES     120
#define    MAXCOLS      160
char    s[MAXLINES][MAXCOLS];    /* Screen Array */

main()
{
        register int  row = 0,
                      col = 0;
        register char c;
        int           char_count = 0; /* count non-blank characters */
        long          t;
        char          buf[BUFSIZ];

        initscr();
        for (row = 0; row < MAXLINES; row++)     /* initialize screen array */
                for (col = 0; col < MAXCOLS;  col++)
                        s[row][col] = ' ';

        row = 0;
        col = 0;
        /* Read screen in */
        while ( (c = getchar()) != EOF && row < LINES) {
                if (c != '\n' && col < COLS) {
                        /* Place char in screen array */
                        s[row][col++] = c;
                        if (c != ' ')
                                char_count++;
                } else {
                        col = 0;
                        row++;
                }
        }

        time(&t);       /* Seed the random number generator */
        srand((int)(t&0177777L));

        while (char_count) {
                row = rand() % LINES;
                col = (rand() >> 2) % COLS;
```

```
                if (s[row][col] != ' ' && s[row][col] != EOF) {
                        move(row,col);
                        addch(s[row][col]);
                        s[row][col] = EOF;
                        char_count--;
                        refresh();
                }
        }

        endwin();
        exit(0);
}
```

SHOW

This example program displays a file taken from the standard input, one screen at a time. Press the terminal space bar to advance to the next screen.

```
#include <curses.h>
#include <signal.h>
main(argc,argv)
        int     argc;
        char    *argv[];
{
        FILE    *fd;
        char    linebuf[BUFSIZ];
        int     line;
        void    done(),perror(),exit();

        if (argc != 2) {
                fprintf(stderr, "usage: %s file\n", argv[0]);
                exit(1);
        }

        if ( (fd = fopen(argv[1], "r")) == NULL) {
                perror(argv[1]);
                exit(2);
        }

        signal(SIGINT, done);
        initscr();
        noecho();
        cbreak();
        nonl();                     /* enable more screen optimization */
        idlok(stdscr,TRUE);       /* allow insert/delete line */

        while (1) {
                move(0,0);
                for (line = 0; line < LINES; line++) {
                        if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
```

```
                                clrtobot();
                                done();
                        }
                        move(line,0);
                        printw("%s", linebuf);
                }

                refresh();
                if (getch() == 'q')
                        done();
        }
}
        void
done()
{
        move(LINES-1, 0);
        clrtoeol();
        refresh();
        endwin();
        exit(0);
}
```

HIGHLIGHT

This example program displays text taken from the standard input. Highlighting is determined by embedded character sequences in the file. \U starts underlining, \B starts bold highlighting, and \N restores normal display characteristics.

```
#include <curses.h>

main(argc,argv)
        int     argc;
        char    *argv[];
{
        FILE    *fd;
        int     c,c2;

        if (argc != 2) {
                fprintf(stderr, "Usage: highlight file\n");
                exit(1);
        }

        fd = fopen(argv[1],"r");
        if (fd == NULL) {
                perror(argv[1]);
                exit(2);
        }

        initscr();
```

```
        scrollok(stdscr,TRUE);

        for (;;) {
                c = getc(fd);
                if (c == EOF)
                        break;

                if (c == '\\') {
                        c2 = getc(fd);
                        switch(c2) {
                        case 'B':
                                attrset(A_BOLD);
                                continue;
                        case 'U':
                                attrset(A_UNDERLINE);
                                continue;
                        case 'N':
                                attrset(0);
                                continue;
                        }

                        addch(c);
                        addch(c2);
                } else
                        addch(c);
        }

        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

WINDOW

This program demonstrates the use of multiple windows.

```
#include <curses.h>

WINDOW    *cmdwin;

main()
{
        int     i,c;
        char    buf[120];

        initscr();
        nonl();
        noecho();
```

```
cbreak();

cmdwin = newwin(3,COLS,0,0);    /* top 3 lines */
for (i=0; i < LINES; i++)
        mvprintw(i,0,"This is line %d of stdscr",i);

for (;;) {
        refresh();
        c = getch();
        switch(c) {
        case 'c':    /* Enter command from keyboard */
                werase(cmdwin);         /* clear window */
                wprintw(cmdwin,"Enter command:");
                wmove(cmdwin,2,0);
                for (i=0; i < COLS; i++)
                        waddch(cmdwin,'-');

                wmove(cmdwin,1,0);
                touchwin(cmdwin);
                wrefresh(cmdwin);
                wgetstr(cmdwin,buf);
                touchwin(stdscr);

                /*
                 * The command is now in buf.
                 * It should be processed here.
                 */
                erase();
                for (i=0; i < LINES; i++)
                        mvprintw(i,0,"%s",buf);
                refresh();
                break;
        case 'q':
                endwin();
                exit(0);
        }
}
```

TWO

This program shows how to handle two terminals from a single program.

```
#include <curses.h>
#include <signal.h>

struct screen *me, *you;
struct screen *set_term();

FILE    *fd, *fdyou;
char    linebuf[512];

main(argc,argv)
        int     argc;
        char    *argv[];
{
        int     done();
        int     c;

        if (argc != 4) {
                fprintf(stderr,"Usage: two othertty otherttytype inputfile\n");
                exit(1);
        }

        fd = fopen(argv[3],"r");
        fdyou = fopen(argv[1],"w+");
        signal(SIGINT, done);    /* die gracefully */

        me = newterm(getenv("TERM"),stdout,stdin);    /* initialize my tty */
        you = newterm(argv[2],fdyou,fdyou);    /* Initialize his/her terminal*/

        set_term(me);           /* Set modes for my terminal */
        noecho();               /* turn off tty echo */
        cbreak();               /* enter cbreak mode */
        nonl();                 /* Allow linefeed */
        nodelay(stdscr,TRUE);   /* No hang on input */

        set_term(you);
        noecho();
        cbreak();
        nonl();
        nodelay(stdscr,TRUE);

        /* Dump first screen full on my terminal */
        dump_page(me);

        /* Dump second screen full on his/her terminal */
        dump_page(you);
```

```
	for (;;) {	/* for each screen full */
		set_term(me);
		c = getch();
		if (c == 'q')	/* wait for user to read it */
			done();
		if (c == ' ')
			dump_page(me);

		set_term(you);
		c = getch();
		if (c == 'q')	/* wait for user to read it */
			done();
		if (c == ' ')
			dump_page(you);
		sleep(1);
	}
}

dump_page(term)
	struct screen *term;
{
	int	line;

	set_term(term);
	move(0,0);
	for (line=0; line < LINES-1; line++) {
		if (fgets(linebuf,sizeof linebuf,fd) == NULL) {
			clrtobot();
			done();
		}
		mvprintw(line,0,"%s",linebuf);
	}

	standout();
	mvprintw(LINES-1,0,"--More--");
	standend();
	refresh();	/* sync screen */
}

/*
 * Clean up and exit.
 */
done()
{
	/* Clean up first terminal */
	set_term(you);
	move(LINES-1,0);		/* to lower left corner */
	clrtoeol();		/* clear bottom line */
	refresh();		/* flush out everything */
```

```
        endwin();               /* curses clean up */

        /* Clean up second terminal */
        set_term(me);
        move(LINES-1,0);        /* to lower left corner */
        clrtoeol();             /* clear bottom line */
        refresh();              /* flush out everything */
        endwin();               /* curses clean up */

        exit(0);
}
```

TERMHL

This program is equivalent to the earlier example program **HIGHLIGHT**, but uses *terminfo* routines instead.

```
#include <curses.h>
#include <term.h>

int     ulmode = 0;     /* Currently underlining */

main(argc, argv)
        int     argc;
        char    *argv[];
{
        FILE    *fd;
        int    c,c2;
        int    outch();

        if (argc > 2) {
                fprintf(stderr, "Usage: termhl [file]\n");
                exit(1);
        }

        if (argc == 2) {
                fd = fopen(argv[1],"r");
                if (fd == NULL) {
                        perror(argv[1]);
                        exit(2);
                }
        } else {
                fd = stdin;
        }

        setupterm(0,1,0);
        for (;;) {
                c = getc(fd);
                if (c == EOF)
                        break;
```

```
                if (c == '\\') {
                        c2 = getc(fd);
                        switch(c2) {
                        case 'B':
                                tputs(enter_bold_mode,1,outch);
                                continue;
                        case 'U':
                                tputs(enter_underline_mode,1,outch);
                                ulmode = 1;
                                continue;
                        case 'N':
                                tputs(exit_attribute_mode,1,outch);
                                ulmode = 0;
                                continue;
                        }
                        putch(c);
                        putch(c2);
                } else
                        putch(c);
        }

        fclose(fd);
        fflush(stdout);
        resetterm();
        exit(0);
}

/*
 * This function is like putchar, but it checks for underlining.
 */
putch(c)
        int     c;
{
        outch(c);
        if (ulmode && underline_char) {
                outch('\b');
                tputs(underline_char,1,outch);
        }
}

/*
 * Outchar is a function version of putchar that can be passed to
 * tputs as a routine to call.
 */
outch(c)
        int     c;
{
        putchar(c);
}
```

EDITOR

This program is a very simple screen-oriented editor that is similar to a small subset of *vi*. For simplicity, the *stdscr* window is also used as the editing buffer.

```
#include <curses.h>
#define CTRL(c) ('c'&037)
main(argc,argv)
        int     argc;
        char    *argv[];
{
        int     i,n,l;
        int     c;
        FILE    *fd;

        if (argc != 2) {
                fprintf(stderr,"Usage: edit file\n");
                exit(1);
        }

        fd = fopen(argv[1],"r");
        if (fd == NULL) {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        cbreak();
        nonl();
        noecho();
        idlok(stdscr, TRUE);
        keypad(stdscr, TRUE);

        /* Read in the file */
        while ((c = getc(fd)) != EOF)
                addch(c);
        fclose(fd);

        move(0,0);
        refresh();
        edit();

        /* Write out the file */
        fd = fopen(argv[1],"w");
        for (l=0; l < LINES; l++) {
                n = len(l);
                for (i=0; i<n; i++)
                        putc(mvinch(l,i),fd);
                putc('\n',fd);    /* typo in AT&T manual */
        }
```

```
        fclose(fd);
        endwin();
        exit(0);
}
len(lineno)
        int     lineno;
{
        int     linelen = COLS-1;

        while (linelen >= 0 && mvinch(lineno,linelen) == ' ')
                linelen--;
        return linelen + 1;
}

/* Global value of current cursor position */
int     row,col;

edit()
{
        int     c;
        for (;;) {
                move(row,col);
                refresh();
                c = getch();
                switch(c) {    /* Editor commands */

                /* hjkl and arrow keys: move cursor */
                /* in direction indicated */
                case    'h':
                case    KEY_LEFT:
                        if (col > 0)
                                col--;
                        break;

                case    'j':
                case    KEY_DOWN:
                        if (row < LINES-1)
                                row++;
                        break;

                case    'k':
                case    KEY_UP:
                        if (row > 0)
                                row--;
                        break;

                case    'l':
                case    KEY_RIGHT:
                        if (col < COLS-1)
                                col++;
```

```
                        break;

                /* i: enter input mode */
                case    KEY_IC:
                case    'i':
                        input();
                        break;

                /* x: delete current character */
                case    KEY_DC:
                case    'x':
                        delch();
                        break;

                /* o: open up a new line and enter input mode */
                case    KEY_IL:
                case    'o':
                        move(++row,col=0);
                        insertln();
                        input();
                        break;

                /* d: delete current line */
                case    KEY_DL:
                case    'd':
                        deleteln();
                        break;

                /* ^L: redraw screen */
                case    KEY_CLEAR:
                case    CTRL(L):
                        clearok(curscr, TRUE);
                        refresh();
                        break;

                /* w: write and quit */
                case    'w':
                        return;

                /* q: quit without writing */
                case    'q':
                        endwin();
                        exit(1);
                default:
                        flash();
                        break;
                }
        }
}
/*
```

```
 * Insert mode: accept characters and insert them.
 * End with ^D or EIC.
 */
input()
{
        int     c;
        standout();
        mvaddstr(LINES-1, COLS-20, "INPUT MODE");
        standend();
        move(row,col);
        refresh();

        for (;;) {
                c = getch();
                if (c == CTRL(D) || c == KEY_EIC)
                        break;
                insch(c);
                move(row, ++col);
                refresh();
        }
        move(LINES-1, COLS-20);
        clrtoeol();
        move(row,col);
        refresh();
}
```

Notes

Index

a

addch . . . 2, 4, 10, 28, 35
addstr . . . 28, 36
alternate character set . . . 10
application program operation . . . 5
application programs structure . . . 3
arrow keys . . . 1, 59
attributes . . . 10, 11
attroff . . . 11, 31, 36, 58
attron . . . 11, 36, 58
attrset . . . 2, 10, 11, 36, 58

b

baudrate . . . 32, 36
beep . . . 22, 32, 36
blinking highlight . . . 10
bold highlight . . . 10
box . . . 29, 36

c

cbreak . . . 4, 9, 26, 37
clear . . . 29, 37
clearok . . . 4, 25, 37
clrtobot . . . 9, 29, 37
clrtoeol . . . 9, 29, 37
COLS . . . 5
configuration routines . . . 26
creating windows . . . 14
current attributes . . . 11
current screen . . . 2
current terminal . . . 17
curscr . . . 2
curses . . . 1
curses routines . . . 33
curses routines, introduction . . . 23
curses routines, listed description of . . . 33–51
curses.h . . . 10

d

data output routines 28
data-input routines:
  terminal data 30
  window 30
delay functions 32
delay_output 37, 52
delch 29, 37
deleteln 29, 37
deleting text 29
deleting text from windows 29
delwin 38
dim highlight 10
display highlighting 10
doupdate 28, 38
draino 32, 38

e

echo 26, 38
endwin 5, 24, 38, 61
erase 29, 39
erasechar 32, 39
ERR 23
escape sequences 22
escape used in program control 22
example programs:
  editor 21, 72
  highlight 12, 65
  scatter 63
  show 12, 64
  termhl 20, 70
  two 18, 56, 68
  window 15, 66

f

fixterm 39, 61
flash 22, 30, 39
flush 4
flushinp 32, 39
formatted output to windows 29

g

getch ... 6, 30, 39
getstr ... 30, 41, 62
gettmode ... 41
getyx ... 30, 41

h

half-bright highlight ... 10
has_ic ... 41
has_il ... 41
highlight escape sequences ... 12
highlighting attribute routines ... 31
highlighting data structure ... 2
highlighting displays ... 10
highlighting program operation ... 57

i

idlok ... 4, 9, 25, 41
inch ... 30, 42
include files ... 23
initialization routines ... 24
initscr ... 4, 24, 42
input routines ... 30
insch ... 29, 42
insert/delete line, program operation ... 55
inserting text ... 29
inserting text in windows ... 29
insertln ... 29, 42
intrflush ... 25, 42
introduction to curses routines ... 23
inverse video ... 10
invisible highlight ... 10

k

keyboard input ... 6
keyboard input program example ... 9
keypad ... 6, 7, 25, 43, 59
keypad character handling ... 7
keypad codes ... 8
killchar ... 32, 43

l

leaveok . . . 25, 43
LINES . . . 5
loader options . . . 24
longname . . . 24, 43, 62
low-level terminfo usage . . . 19

m

magic cookie . . . 55
manipulation routines . . . 27
meta . . . 25, 40, 43
mini-curses . . . 24, 60, 61
miscellaneous curses functions . . . 32
miscellaneous window operations . . . 29
move . . . 28, 44
multiple terminals . . . 17, 56
multiple terminals, program operation . . . 56
multiple types of terminals, dealing with . . . 55
multiple windows . . . 13, 14
mvaddch . . . 44
mvaddstr . . . 44
mvcur . . . 44
mvdelch . . . 44
mvgetch . . . 44
mvgetstr . . . 44
mvinch . . . 44
mvprintw . . . 44
mvscanw . . . 44
mvwaddch . . . 44
mvwaddstr . . . 44
mvwdelch . . . 44
mvwgetch . . . 44
mvwgetstr . . . 44
mvwin . . . 44
mvwinch . . . 45
mvwinsch . . . 45
mvwprintw . . . 45
mvwscanw . . . 45

n

napms 32, 45
newpad 45
newterm 17, 24, 45, 56
newwin 14, 45
nl 26, 46
no-print highlight 10
nocbreak 46
nodelay 25, 46
nodelay mode 62
noecho 9, 26, 46
nonl 9, 26, 46
noraw 27, 46

o

OK 23
option-setting routines 25
options 25
output data structure 2
overlay 14, 27, 46
overwrite 14, 27, 46

p

padding 2
pads 13
placing text in windows 28
pnoutrefresh 28, 46
portability functions 32
prefresh 28, 46
printw 4, 29, 47
program structure considerations 23
putp 52

r

race conditions 17
raw 27, 47
refresh 4, 12, 21, 28, 47
resetterm 47, 61
resetty 27, 47

s

saveterm . . . 47
savetty . . . 27, 47
scanw . . . 30, 48
screen size . . . 5
scroll . . . 48
scrolling regions in window or pad . . . 60
scrollok . . . 26, 48
scrollw . . . 29
setscrreg . . . 26, 48, 60
setterm . . . 48
set_term . . . 17, 18, 49
setupterm . . . 19, 24, 48, 52
special keys on terminals, keypad program handling . . . 59
standard screen . . . 2
standend . . . 31, 49
standout . . . 31, 49, 58
standout highlight . . . 10
stdscr . . . 2
struct screen . . . 56
structure considerations for programs . . . 23
sttron . . . 31
sttrset . . . 31
subwin . . . 49
subwindows . . . 16

t

TERM . . . 1
termcap compatibility routines . . . 54
terminal configuration routines . . . 26
terminal data output routines . . . 28
terminal data-input routines . . . 30
terminal initialization routines . . . 24
terminfo . . . 1
terminfo routines, listed description of . . . 52–54
terminfo-level access . . . 19
text data structure . . . 2
touchwin . . . 15, 27, 49
tparm . . . 53
tputs . . . 20, 53
traceoff . . . 49

traceon 49
tty mode functions 61
typeahead check 25, 50, 61

u

unctrl 50
underlining highlight 10
using multiple windows 14

v

vidattr 20, 53
video highlighting attribute routines 31
video highlighting, program operation 57
vidputs 54
vminsch 44

w

waddch 10, 14, 35, 50
waddstr 35
wattroff 31, 36, 58
wattron 31, 36, 58
wattrset 36, 58
wclear 29
wdeleteln 29
window 2
windows 13–16
windows:
  creating 14
  data-input routines 30
  formatted output to 29
  inserting and deleting text 29
  miscellaneous operations 29
  multiple 13
  placing text in windows 28
  subwindows 16
  window manipulation routines 27
  window writing routines 28
wmove 28
wnoutrefresh 28
wrefresh 14, 28
wsetscrreg 60
wstandout 58

Table of Contents

Overview

Who Will Use Native Language Support? 1
Manual Organization 2
Conventions Used In This Manual 3
Using Other HP-UX Manuals 4

Character Set Representation and Introduction to NLS

What Is NLS? 6
Scope of Native Language Support 7
Aspects of NLS Support 7
Prelocalized commands 10
Supported Native Languages and Character Sets 11
8-Bit Character Sets 11
16-Bit Character Sets 15
Native Languages 17

Configuring Native Language Support on HP-UX

File Hierarchy 21
Configuring Native Languages 22
Installing Optional Languages 22
Environment Changes 23
Accessing NLS Features 24
NLS HP-UX Commands 24
Library Support for NLS 24

Programming With Native Language Support

NLS Header Files 25
Library Routines 26
Convert date/time to string 26
Convert floating point to string 26
Get message from catalog 27
Information on user's native language 27
C routines to translate characters 28
C routines that classify characters 28
Non-ASCII string collation 29
Print formatted output with numbered arguments 30

Convert string to double precision number 31
Multi-byte Library Routines 31
Application Guidelines 32
Example C Programs 32
Example 1 32
Example 2 34

Message Catalog System
Introduction to the Message Catalog System 37
Creating a Message Catalog 39
Preview: Incorporating NLS into Commands 39
Following the Flow 40
Format of Source Message File 44
Printmsg, Fprintmsg, and Sprintmsg 46
Accessing Applications Catalogs 46
File System Organization and Catalog Naming Conventions 47
Prelocalization: Adding Native Language Support 48
7 Steps to Prelocalize an Example Program 48
Localization 51
Maintaining Programs and Message Catalogs 51

Native Language Support Library and Commands
Library Routines 53
Commands 56
NLS Files 57

Character Sets 59

Peripheral Configuration
European Character Sets 63
Japanese Character Sets 63
ISO 7-bit Substituion 64
Character Set Support by Peripherals 64

Glossary 69

Index 77

Overview

This manual describes what Native Language Support (NLS) is and how to use the NLS tools on your Hewlett-Packard computer.

Please use one of the reply cards at the back of this manual to tell us what was helpful, what was not, and why. Feel free to comment on depth, technical accuracy, organization, and style. Your comments are appreciated.

Who Will Use Native Language Support?

OEMs (Original Equipment Manufacturers), ISVs (Independent Software Vendors), applications programmers, and Hewlett-Packard Country Software Centers will be the primary users of Native Language Support (NLS). These are the people writing or translating programs for multi-national use.

This manual has been written with these users in mind.

Manual Organization

Overview
Defines the NLS user audience, explains the conventions used in the manual, and identifies other manuals referenced within this one.

Chapter 1: Character Set Representation and Introduction to NLS
Presents the basic description and scope of Native Language Support, Localization, and Internationalization. This includes the aspects of NLS (Character Set Support, Local Customs, and Messages), pre-localization, and the character sets as well as native languages supported.

Chapter 2: Native Language Support on HP-UX
Identifies the HP-UX directories and files in which the NLS tools reside, provides an installation guide for the optional languages, and identifies the library calls (and commands) that an applications programmer needs in order to access NLS features.

Chapter 3: Programming With Native Language Support
Presents the header files specific to NLS, a detailed description of the C library routines (with their syntax), and example C programs (with their command lines and output).

Chapter 4: Message Catalog System
Explains how local language message files are created and updated, where they are kept, and by what conventions they are named. This includes a diagram and description of the general flow of the message catalog system, ways to access catalogs by use of library routines, file naming conventions and an example of program output in a local language other than American English.

Appendix A: Native Language Support Library and Commands
Overview of NLS library routines and routines affected by NLS.

Appendix B: Character Sets
ASCII, Roman and Katakana character sets.

Appendix C: Peripheral Configuration
Table summarys of HP 9000 peripherals that support alternate character sets.

Conventions Used In This Manual

The following naming conventions are used throughout this manual.

- *Italics* indicate files and HP-UX commands, system calls, and subroutines found in the *HP-UX Reference* manual as well as titles of manuals. Italics are also used for symbolic items either typed by the user or displayed by the system as discussed below. Examples include */usr/lib/nls/american/prog.cat, date(1),* and *pty(4).* The parenthetic number shown for commands, system calls, and other items found in the *HP-UX Reference* is a convention used in that manual.
- **Boldface** is used when a word is first defined and for **general emphasis**.
- `Computer font` indicates a literal typed by the user or displayed by the system. A typical example is:

  ```
  findstr prog.c > prog.str
  ```

  Note that when a command or file name is part of a literal, it is shown in computer font and not italics. However, if the command or file name is symbolic (but not literal), it is shown in italics as the following example illustrates.

  `findstr` *progname* > *output-file-name*

  In this case you would type in your own *progname* and *output-file-name.*
- Environment variables such as LANG or PATH are represented in uppercase characters.
- Unless otherwise stated, all references such as "see the *nl_toupper(3C)* entry for more details" refer to entries in the *HP-UX Reference* manual. Some of these entries will be under an associated heading. For example, the *nl_toupper(3C)* entry is under the *nl_conv(3C)* heading. If you cannot find an entry where you expect it to be, use the *HP-UX Reference* Manual's Permuted Index.

Using Other HP-UX Manuals

This manual may be used in conjunction with other HP-UX documentation. References to these manuals are included, where appropriate, in the text.

- The *HP-UX Reference* manual contains the syntactic and semantic details of all commands and application programs, system calls, subroutines, special files, file formats, miscellaneous facilities, and maintenance procedures available on the HP 9000 HP-UX Operating System.
- The *HP-UX Portability Guide* documents the guidelines and techniques for maximizing the portability of programs written on and for HP 9000, Series 200, 300, and 500 computers running the HP-UX Operating System. It covers the portability of high level source code (C, Pascal, FORTRAN) and transportability of data and source files between commonly used formats.
- The *HP-UX System Administrator Manual* provides step-by-step instructions for installing the HP-UX Operating System software and for installing the optional NLS languages. It also explains certain concepts used and implemented in HP-UX, describes system boot and login, and contains the guide for implementing administrative tasks.

Character Set Representation and Introduction to NLS

1

The features of Hewlett-Packard **Native Language Support (NLS)** enable the applications designer or programmer to adapt applications to an end user's local language needs.

NLS provides the programmer with the ability to internationalize software. **Internationalization** is the concept of providing hardware and software which is capable of supporting the user's local language. NLS, along with Hewlett-Packard hardware, accomplishes this. **Localization** refers to the process of adapting a software application or system for use in different local environments or countries. It includes all changes that must be done repeatedly for each language or locale of interest as well as for each piece of software.

What Is NLS?

NLS provides the tools for an applications designer or programmer to produce localizable applications. These tools include architecture and peripheral support, as well as software facilities within the operating systems and subsystems. NLS addresses the internal functions of a program (such as sorting) as well as its user interface (which includes displayed messages, user inputs, and currency formats.)

An addition to providing the tools for programmers to develop applications in several languages, NLS also provides end users with the following features:

- NLS saves disc space by separating the resources required for a specific language product from the executable code. The components of these products are tables of data used by the software in the base product.
- NLS allows different users to use different languages, all on the same system.
- NLS permits users to specify the desired language at run time.

Users who are less technically sophisticated benefit from application programs that interact with them in their **native language** and conform to their **local customs**. **Native language** refers to the user's first language (learned as a child), such as Finnish, Portuguese, or Japanese. **Local customs** refer to local conventions such as date, time, and currency formats.

Programs written with the intention of providing a friendly user interface often make assumptions about the user's local customs and language. Program interface and processing requirements vary from country to country; sometimes even within a country. Most existing software does not take this into account, making it appropriate for use only in the country or locality for which it was originally written.

Scope of Native Language Support

NLS facilities allow application programs to be designed and written with a local language interface for the end user and for locally correct internal processing. The end user then interacts with localized programs.

For the USASCII-only user the system will appear unchanged. HP-UX commands which check the LANG environment variable work the same as before unless LANG is set (though exceptions to this may exist).

For the programmer and the system administrator, the interface has not changed. Most HP-UX interfacing, subsystems, programmer productivity tools, and compilers have not been localized. Applications programmers must still use American English to interact with HP-UX and its subsystems. For example, it is possible to write a complete local language application program using C, but the C compiler retains the English-like characteristics. For example, C keywords such as *main, if*, and *while*, and library calls such as *printf* are still in English.

Aspects of NLS Support

There are three aspects, or levels, of native language support included in HP-UX software. These three aspects, **Character Set Support**, **Local Customs**, and **Messages**, describe the extent of localization of an application. The applications programmer should consider each aspect carefully when creating software that is language independent.

Character Set Support

A major NLS objective is to provide the capabilities for adapting character sets and sequences to local language needs. This takes into account that character code size determines the maximum number of distinct characters contained in a set. The default set is 7-bit ASCII character set; all programs not localized use this character set. 7-bit ASCII is not sufficient to span the Latin alphabet used in many European Languages including upper- and lowercase, punctuation, and special symbols.

The 8th bit of a character byte is normally never stripped or modified. Hewlett-Packard has defined character sets with bytes in the range 0 to 255 for foreign languages instead of ASCII's 0 to 127. Using the extra bit allows expansion to support languages that have additional characters, accented vowels, consonants with special forms and special symbols.

For languages with larger character sets, such as **Kanji** (the Japanese ideographic character set based on Chinese), multi-byte character codes are required.

For more detail on the different character sets, refer to the section called "Supported Native Languages and Character Sets" in this chapter.

All sorting, shifting, and type analysis of characters is done according to the local conventions for the native language selected. While the ROMAN8 character set has uppercase and lowercase for most alphabetic characters, some languages discard accents when characters are shifted to uppercase. European French discards accents while Canadian-French does not. If there is no notion of **case** in the underlying language (such as Katakana), alphabetic characters are not shifted at all.

Each language uses its own distinct **collating sequences** (the sequence in which characters acceptable to the computer are ordered). The ASCII collation order is inadequate even for American dictionary usage. Different languages sort characters from the ROMAN8 set in different orders. For example, Spanish requires character pairs such as "ch" and "ll" to be sorted as single characters. Therefore, "ch" falls at the end of the sorted pairs "cg", "ci", and "cz", and "ll" similarly falls after "lk", "lm", and "lz". Certain **ideographic** character sets, which represent ideas by graphic symbols, can have multiple orderings. An instance of this is Japanese ideograms (use of graphic symbols to represent Kanji) which can be sorted in phonetic order; based on the number of strokes in the ideogram; or according, first, to the radical (root) of the character and, second, to the number of strokes added to the radical.

On the subject of directionality, the assumption that displayed text goes from left to right does not hold for all languages. Some Middle Eastern languages such as Hebrew are read from right to left; some Far Eastern languages use vertical columns, starting from the rightmost column.

Local Customs

Some aspects of NLS relate more to the local customs of a particular geographic area. These aspects, even when supported by a common character set, change from region to region. Consequently, date and time, number, currency information, and so on are presented in a way appropriate to the user's language. For instance, although Great Britain, the United States, Canada, Australia, and New Zealand share the English language, other aspects of data representation differ according to local custom.

The representation of numbers, variations in the symbol indicating the radix character (period in the U.S.), modification of the digit grouping symbol (comma in the U.S.), and the number of digits in a group (three in the U.S.), are all based on the user's native customs. For example, the United States and France both represent currency using periods and commas, but the symbols are transposed (2,345.77 vs. 2.345,77).

Currency units and how they are subdivided vary with region and country. The symbol for a currency unit can change as well as the symbol's placement. It can precede, follow, or appear within the numeric value. Similarly, some currencies allow decimal fractions while others use alternate methods for representing smaller monetary values.

Computation and proper display of time, 24- versus 12-hour clocks, and date information must be considered. The HP-UX system clock runs on Greenwich Mean Time (GMT). Corrections to local time zones consist of adding or subtracting whole or fractional hours from GMT. Some regions, instead of using the common Gregorian calender system, number (or name) the years based upon seasonal, astronomical, or historical events. For example, in Arabic, time of day is measured from the previous sunset; in India, the calendar is strictly lunar (with a leap month every few years); in Japan years are based upon the reign of the emperor.

Names for days of the week and the months of the year also vary with language. Rules for abbreviations also differ. Ordering of the year, month, and day, as well as the separating delimiters, is not universally defined. For example, October 7, 1986 would be represented as *10/7/1986* in the U.S. and as *7.10.1986* in Germany.

The chapter "Programming With NLS" describes the library routines used to access these local customization features.

Messages

The need to customize messages for different countries is perhaps the most significant justification for implementing Native Language Support. The user can choose the language for prompts, response to prompts, error messages, and mnemonic command names at run time. Thus it is not necessary to recompile source code when translating messages to another language. Keep in mind the syntax of another language may force a change in the structure of the sentence if messages are built in segments (using *printf(3S)*). For example, in German, *"output from standard out and file"* becomes *"Aus und sammlung aus dem standarden ausgabe"*, which translates literally to *"out and file from standard output."*

To do this, user messages must be put in a **message catalog** from which they are retrieved by special library calls. The chapter "Message Catalog System" explains how to create and access message catalogs.

As an example, a fully localized version of *pr* (the HP-UX print command) would

- never strip the 8th bit of a character code
- properly format the date in each page header
- use the message catalog system to select user error messages.

Prelocalized commands

Prelocalization is program modification that uses language-dependent library routines not limited to 7-bit character processing. These routines are enhanced to ensure the proper handling of 8-bit data.

Localization consists of taking the prelocalized program and adding the necessary message catalogs and tables to make it run in a particular language (such as French).

Prelocalization allows the message catalogs and tables to be specified at run time, rather than having the information hard-coded and compiled into the programs.

A **localized message file** contains messages in the desired native language. Some HP-UX commands have been enhanced to check for localized message files.

To prelocalize source code, you would replace original HP-UX commands and routines with commands and routines that incorporate NLS. For example, the routine *ctime* would be replaced with the NLS enhanced version *nl_ctime*.

Supported Native Languages and Character Sets

HP-UX NLS is based on 18 languages and 6 character sets. Facilities to handle these character sets are built into the operating system. Tables and files associated with supported languages will be available through Hewlett-Packard sales offices.

Within NLS, each supported language is associated with a 7-bit, 8-bit, or 16-bit character set (one character set may support several languages).

The 7-bit character set is called USASCII. This character set is the traditional computer ASCII character set. Before the introduction of NLS, the only widely supported character set was ASCII, a 128-character set designed to support American English text. ASCII uses only seven bits of an 8-bit byte to encode each character. The eighth or high order bit is usually zero, except in some applications where it is used for other purposes. For this reason, ASCII is referred to as a "7-bit" code.

The 8-bit and 16-bit character sets are described below.

8-Bit Character Sets

The 8-bit character sets are comprised of an ASCII character set for values from 0 to 127, and non-ASCII characters for values from 128 to 255. These 256 unique values permit encoding and manipulation of characters required by languages other than American English and are referred to as **8-bit compatible** or **extended** character sets. These sets have five distinct ranges: 0 to 31 and 127 are **control codes**; 32 is **space**; 33 to 126 are **printable characters**; 128 to 160 and 255 are **extended control characters**; and 161 to 254 are **extended printable characters** (see Table 1-1.) New printable characters are added by defining code values in the range 161 to 254.

Table 1-1. 8-bit Character Set Structure

COL BIT	8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
ROW BIT	5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
4 3 2 1		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0 0 0 0	0	CONTROL CODES (0-31)		SP						EXTENDED CHARACTERS (128-160)							
0 0 0 1	1																
0 0 1 0	2																
0 0 1 1	3																
0 1 0 0	4				USASCII GRAPHIC (printable) CHARACTERS (33-126)								EXTENDED PRINTABLE CHARACTERS (161-254)				
0 1 0 1	5																
0 1 1 0	6																
0 1 1 1	7																
1 0 0 0	8																
1 0 0 1	9																
1 0 1 0	10																
1 0 1 1	11																
1 1 0 0	12																
1 1 0 1	13																
1 1 1 0	14																
1 1 1 1	15								127								255

NLS supports four 8-bit character sets: ROMAN8, KANA8, GREEK8, and TURKISH8. There are also line drawing and math character sets supported by Hewlet-Packard that are not a part of the NLS system. The ROMAN8 and KANA8 character sets are shown in the appendix "Character Sets".

NLS 8-bit character sets support all ASCII characters in addition to the characters needed to support several Western European-based languages and Katakana. The exception to this is that the graphic (on the keyboard/overlay) for back slash ("\") in KANA8 is yen ("¥")

Since NLS uses 8-bit character sets in character data, every bit in an 8-bit byte has significance. Application software must take care to preserve the eighth (high order) bit and not allow it to be modified or reused for any special purpose. Also, no differentiation should be made between characters having the eighth bit turned off and those with it turned on, because all characters have equal status in any extended character set.

Peripherals play a key role in a system's ability to represent a particular language. Sometimes, even within a single document, several character sets are needed. For example, this document's tables needed line drawing characters; another section contains a German example. Hewlett-Packard peripherals (generally) use the *8-bit Character Set Support Model* to handle multiple character sets (see Figure 1-1).

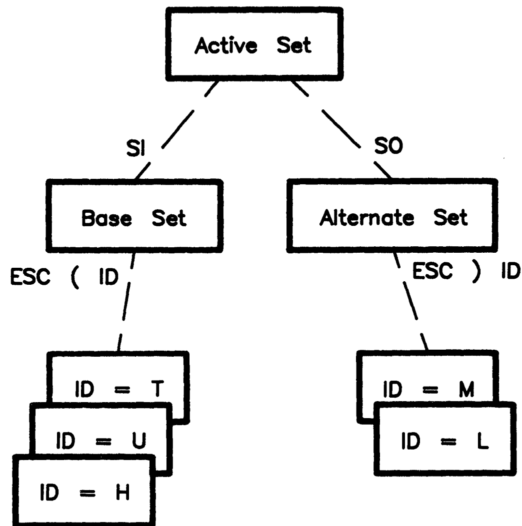


Figure 1-1. 8-bit Character Set Support Model for Peripherals

Each rectangle in Figure 1-1 represents a collection (or set) of 256 character code values in the form shown in Table 1-1. The appendix called "Character Sets" contains tables of characters along with their associated ID values.

The **Active Set** is the one the printed, plotted, or displayed on the terminal. S_I (shift in) and S_O (shift out) characters are used to invoke or activate the **Base** or **Alternate** character set. The Base Set is the language-oriented set while the Alternate Set is for special symbols. The escape sequences E_C(*ID* and E_C) *ID* are used to designate, from the collection of available character sets, the Base and Alternate Set. *ID* designates **ID Field** in this context; see Table 1-2 for a list of character sets with their ID Field numbers. All sets in this model are 8-bit character sets.

Table 1-2. Character Set ID Numbers

8-bit Character Set Name	ID Field
Start up Base/Default Set	@
GREEK8 Character Set	8 G
KANA8 Character Set	8 K
LINEDRAW8 Character Set	8 L
MATH8 Set	8 M
TURKISH8 Character Set	8 T
ROMAN8 Character Set	8 U

16-Bit Character Sets

Asian languages require character sets with more than 256 values. To provide more than 256 values, the character sets must use more than one byte to represent characters. 16-bit character sets are comprised of two-byte characters.

An Asian character is identified by the first byte having the high order bit on. The tool *firstof2* provides the developer the ability to easily identify Asian characters (16-bit) from ASCII. Refer to the chapter called "Programming with Native Language Support" for more details on *firstof2*.

Japanese is currently the only language requiring a 16-bit character set that is supported by HP-UX. Japanese uses the 16-bit character set called JAPAN15.

HP uses two types of 16-bit character encoding schemes: HP-15 for use within the operating system and HP-16 for use outside the operating system. As shown in Figure 1-2, any 16-bit characters within HP-UX system (i.e., for applications programs, text files, etc.) use a method of encoding the characters called HP-15. For I/O a method of encoding called HP-16 is used. The conversion between HP-15 and HP-16 is performed by the HP-UX system.

Unless you are writing your own input servers or printer models, you need only be concerned with the HP-15 encoding scheme. Only HP-15 is described in this manual.

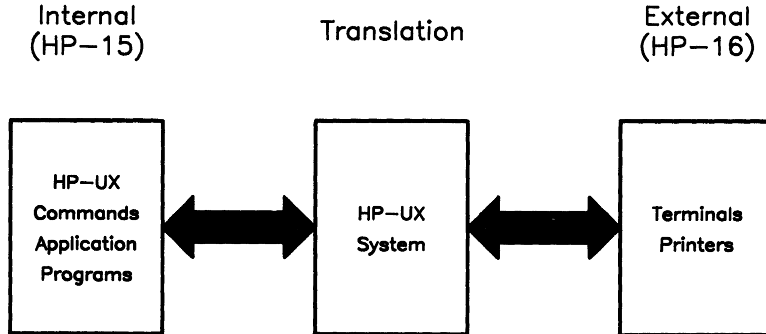
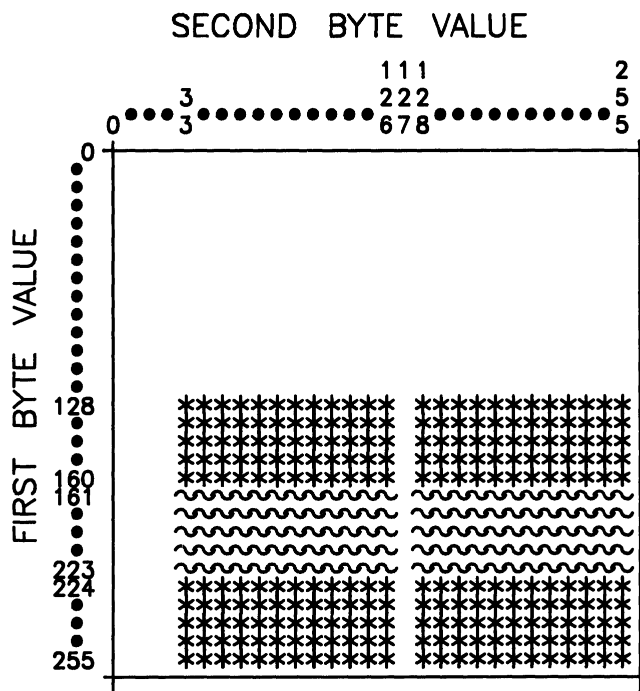


Figure 1-2. Use of 16-bit character sets within HP-UX

Table 1-3 depicts the 16-bit code space of HP-15. The complete range of values (0-255) of the first byte is shown along the vertical axis, and the same range for the second byte is shown horizontally. The HP-UX system will support character code values in the indicated area as either 16-bit or 8-bit. In the case of Japanese, those shown as "*" are 16-bit, those shown as "~" are 8-bit. Byte values outside the indicated area are **always** treated as 8-bit characters.

Table 1-3. Usable character values for HP-15



Hewlett-Packard has developed tools to help parse data so you can determine what characters your data contains. These library routines are described in the appendix "Native Language Support Library and Commands".

Native Languages

Each supported native language is based on one of the six character sets (USASCII, GREEK8, KANA8, ROMAN8, TURKISH8, and JAPAN15). They consist of several language-dependent characteristics defined in various system tables and accessed by C library routines and HP-UX commands. These characteristics include rules on upshifting, downshifting, date and time format, currency, and collating sequence.

Hewlett-Packard has assigned a unique **language name** and **language number** to each language included in NLS (see Table 1-4). In some cases, Hewlett-Packard has introduced more than one **supported language** corresponding to a single **natural language**. For example, NLS supports both French and Canadian-French because upshifting is handled differently in French and Canadian-French.

Each of the supported languages can also be considered a **language family** which is applicable in several countries. German, for example, can be used in Germany, Austria, Switzerland, and any other place it is requested.

In addition to the native languages supported, an artificial language, `native-computer`, represents the way the computers dealt with languages before the introduction of NLS. Whenever `native-computer` is used in a native language function, the result is identical to that of the same function performed before the introduction of NLS. NLS library calls with the language parameter equal to 0 will always work correctly, even when no native languages have been configured on the system.

Table 1-4. Supported Native Languages and Character Sets

Language Number	Language Name	Character Set
00	n-computer (native computer)	USASCII
01	american	ROMAN8
02	c-french (canadian french)	ROMAN8
03	danish	ROMAN8
04	dutch	ROMAN8
05	english	ROMAN8
06	finnish	ROMAN8
07	french	ROMAN8
08	german	ROMAN8
09	italian	ROMAN8
10	norwegian	ROMAN8
11	portuguese	ROMAN8
12	spanish	ROMAN8
13	swedish	ROMAN8
41	katakana	KANA8
61	greek	GREEK8
81	turkish	TURKISH8
221	japanese	JAPAN15

Notes

Configuring Native Language Support on HP-UX 2

File Hierarchy

Prelocalized HP-UX commands and C library routines for NLS are in standard directories (*/bin*, */usr/bin*, and */usr/lib*), but language-dependent features reside in directories and files created specifically for NLS:

- The language configuration file, */usr/lib/nls/config*, is a file containing all the native languages that can be configured into a system. Your system has a table like this:

  ```
    0 n-computer
    1 american
    2 c-french
    3 danish
    4 dutch
    5 english
    6 finnish
    7 french
    8 german
    9 italian
    10 norwegian
    11 portuguese
    12 spanish
    13 swedish
    41 katakana
    61 greek
    81 turkish
   221 japanese
  ```

  Your computer is always configured for **native-computer**, language number 0 (see Figure 1-4).

  Unless you have purchased and installed the language, your computer will not actually have the language's files (refer to the section "Installing Optional Languages" later in this chapter). The "*config*" file is used by *langinfo* routines; it must be present before prelocalized commands can work correctly.

- The following directories are of the form */usr/lib/nls/$LANG* where *$LANG* is a native language (such as *american*).
  - */usr/lib/nls/$LANG/collate8* contains the collating sequence for a given language.
  - */usr/lib/nls/$LANG/ctype* contains information on character set type for the language *$LANG*.
  - */usr/lib/nls/$LANG/info.cat* contains language-dependent information used by *langinfo*.
  - */usr/lib/nls/$LANG/shift* has shift tables (uppercase to lowercase or vice-versa).

Configuring Native Languages

To use a language other than `native-computer` (the default language on HP-UX) you must purchase the support software for the optional language and update the environment accordingly.

Installing Optional Languages

HP-UX is shipped with only the default language (`native-computer`). Other languages (such as German) must be ordered as an option from your Hewlett-Packard sales office. A language includes the tables for collating, upshifting, downshifting, and includes character type and language information. Not all character sets are supported on all peripherals, so peripherals which support the desired character set must be obtained.

NLS includes the library header files and routines (described in Chapter 3), and message catalog system (described in Chapter 4). Message catalogs for HP-UX commands are available in `native-computer` language. You can use these as a basis for translation to local catalogs.

To install a language, use the *update* command, as explained in the chapter of the *HP-UX System Administrator's Manual* entitled "The System Administrator's Toolbox". *Update* automatically installs the language support files in the correct directory as described in the previous section "File Hierarchy".

After a language is installed, the NLS language-specific information can be used by any application program requesting it.

Environment Changes

To support NLS, changes to the user environment within HP-UX are needed. A new environment variable **LANG** (LANGuage) was created during installation. LANG specifies the language you want to use. The variable **TZ** (Time Zone), which allows input about different zones, needs to be changed.

LANG

LANG is an environment variable that must be set to the native language you desire. LANG contains the language name in American English text. It is used to select the character set, lexical order, upshift and downshift tables, and other conventions that vary with language and locality. LANG can be set in */etc/profile* as a default native language, or it can be set by any individual user in *.profile* or *.login*.

An example *.profile*, setting LANG to `american`, is:

```
LANG=american
export LANG
```

For *.login* use:

```
setenv LANG american
```

If LANG is not set, or is set to an invalid language string, a warning message will be issued and all programs using LANG default to the native computer language.

TZ

TZ is a variable that holds time zone information. TZ allows fractional offsets from GMT (Greenwich Mean Time is the international basis of standard time). Specification of daylight savings time is taken into account as well as name differences and starting and ending date differences.

Accessing NLS Features

On HP-UX, all NLS features are optional. These features must be requested by the applications programmer through library calls or interactively by the user through a localized HP-UX command. The C library routines used for NLS can also be accessed from Pascal and FORTRAN. A description of how to access C library routines from Pascal and FORTRAN is documented in the *HP-UX Portability Guide.*

NLS HP-UX Commands

There are several HP-UX commands that were created specifically to access the message catalog features. They are described in detail in the chapter "Message Catalog System".

- *findstr*— find strings in programs not previously localized for inclusion in message catalogs.
- *gencat*— generate a formatted message catalog file.
- *insertmsg*— use output from *findstr* to both create a preliminary message file and to create a new C program with calls to the message file.
- *findmsg*— extracts strings from prelocalized C programs for inclusion in message catalogs.
- *dumpmsg*— reverse the effect of *gencat*; take a formatted message catalog and make a modifiable message catalog source file.

Library Support for NLS

There are several C library routines access the language tables and message catalogs (see the appendix "Native Language Support Library and Commands"). These are documented in the chapter "Programming With Native Language Support".

Programming With Native Language Support 3

This chapter describes the NLS header files and the C library routines that are used by Native Language Support (NLS). Two example programs are also provided.

NLS Header Files

There are three header files in */usr/include* specific to NLS: *msgbuf.h, nl_ctype.h,* and *langinfo.h.*

Library Routines

Most NLS library routines have counterparts within the standard HP-UX system. These routines produce similar results; but, instead of assuming standard formats, they use NLS-specific parameters to format information as the user prefers to see it.

NLS Library routines are listed below. Routines that have counterparts in the standard C library are mentioned, but not described in detail. Other NLS routines that were added to the C library are described in more detail. Manual pages for all these routines are included in the *HP-UX Reference.* NLS routines are discussed in this chapter in the same sequence as in the *HP-UX Reference*, Section 3.

Convert date/time to string

```
nl_ctime(clock, format, langid)
nl_asctime(tm, format, langid)
```

The *nl_ctime* routine extends the capabilities of *ctime* in two ways. First the *format* specification allows the date and time to be output in a variety of ways. *format* uses the field descriptors defined in *date(1).* If *format* is the null string, the *D_T_FMT* string defined by *langinfo(3C)* is used. Second, *langid* provides month and weekday names (when selected as alphabetic by the format string) to be in the user's native language. The *nl_asctime* routine is similar to *asctime*, but like *nl_ctime* allows the date string to be formatted and the month and weekday names to be in the user's native language. However, like *asctime*, it takes *tm* as its argument.

See *ctime(3C)* for these commands.

Convert floating point to string

```
nl_gcvt(value, ndigit, buf, langid)
```

The *nl_gcvt* routine differs from *gcvt* only in that it uses *langid* to determine what the radix character should be. If *langid* is not valid, or information for *langid* has not been installed, the radix character defaults to a period.

See *ecvt(3C)* for these routines.

Get message from catalog

```
getmsg(fd, set_num, msg_num, buf, buflen)
```

where *fd* is the file descriptor pointing to the catalog (file) containing the messages, *set_num* is the set number designating a group of messages in the catalog, *msg_num* is the message number within that set, *buf* is the character array that will hold the returned message, and *buflen* is the number of bytes of the message that can be put into *buf*. The function itself returns a pointer to the character string in *buf*. If *fd* is invalid or *set_num* or *msg_num* are not in the catalog, it returns a pointer to an empty (null) string.

See *getmsg(3C)* for more information.

Information on user's native language

```
langinfo(langid, item)
langtoid(langname)
idtolang(langid)
currlangid()
```

where *langid* is language information and *item* is one of several types of definitions. Refer to the *langinfo(3C)* manual page for a complete list of *items*. Some examples of *item* are:

`D_T_FMT`	string for formatting *date(1)*, *nl_ctime*, and *nl_asctime.*
`DAY_1`	"`Sunday`" in English
⋮	
`DAY_7`	"`Saturday`" in English
`MON_1`	"`January`"
⋮	
`MON_12`	"`December`"
`RADIXCHAR`	"decimal point" ("`,`" on the European Continent)
`THOUSEP`	separator for thousands
`YESSTR`	affirmative response for [y/n] questions
`NOSTR`	negative response for [y/n] questions
`CRNCYSTR`	symbol for currency preceded by "-" if it precedes the number, "+" if it follows the number. (e.g., "`-DM`" for Dutch, "`+ Kr`" for Danish.)

The command *langinfo* retrieves a null-terminated string containing information unique to a language or cultural area.

The *idtolang* routine takes the integer *langid* and returns the corresponding character string (language name) defined in the *langid(7)* manual page. If *langid* is not found, an empty string is returned. The routine *langtoid* is the reverse of *idtolang.* The *currlangid* routine looks for a LANG variable in the user's environment. If it finds it, it returns the corresponding integer (language number) listed in *langid(7).* Otherwise it returns 0 to indicate a default to ASCII **native-computer**.

See *langinfo(3C)* for more information.

C routines to translate characters

```
nl_toupper(c,langid)
nl_tolower(c,langid)
```

These routines are similar to the routines in *conv(3C).* They function the same way, but use a second parameter whose value is expected to be one of the values defined in *langid(7).* If *langid* has not been installed or if shift information for *langid* has not been installed, *toupper* and *tolower* is used for characters below 127, while characters 127 and above are returned unchanged (*toupper* and *tolower* are used with ASCII character set only).

See the *nl_conv(3C)* manual page for this routine; see also *conv(3C).*

C routines that classify characters

All these routines have the same parameter list:

routine(c, langid)

where *routine* is any of the routines in *nl_ctype.*

nl_isalpha	*c* is a letter
nl_isupper	*c* is an upper case letter
nl_islower	*c* is a lower case letter
nl_isalnum	*c* is an alphanumeric (letter or digit)
nl_ispunct	*c* is a punctuation character (neither control nor alphanumeric)
nl_isprint	*c* is a printing character
nl_isgraph	*c* is a printing character, like *nl_isprint* except false for space

These routines classify character-coded integer values by using the tables in */usr/lib/nls*. The command *langid* is as defined in *langid(7)*. Each returns non-zero for true, zero for false. All are defined for the range -1 to 255. If *langid* is not defined or if type information for that language is not installed, *isalpha, isupper,* etc. from *ctype(3C)* is used, returning 0 for values above 127.

If the argument to any of these routines is not in the domain of the function, the result is undefined.

See the *nl_ctype(3C)* manual page for more information.

Non-ASCII string collation

```
strcmp8(s1, s2, langid, status)
strncmp8(s1, s2, n, langid, status)
strcmp16(s1, s2, file_name, status)
strncmp16(s1, s2, n, file_name, status)
```

The command *strcmp8* compares string *s1* and *s2* according to the collating sequence specified by *langid* (the language number). An integer greater than, equal to, or less than *0* is returned, depending on whether the collation of *s1* is greater than, equal to, or less than that of *s2*. If *langid* or the collation sequence file is not installed, the native machine collating sequence is used. The command *strncmp8* makes the same comparison but looks at only *n* characters. The *strcmp16* and *strncmp16* commands are similar, but use the 16-bit Japanese collating sequence. The *file_name* argument is currently ignored and should always be the null string literal (`""`).

If an abnormal condition is encountered the integer pointed to by *status* is set to one of the following non-zero values: `ENOCFILE`, `ENOCONV`, `ENOLFILE`. These values are defined in */usr/include/langinfo.h*

See the *nl_string(3C)* manual page for more information.

Print formatted output with numbered arguments

```
printmsg (format [ , arg ] ... )
fprintmsg (stream, format [ , arg ] ... )
sprintmsg (s, format [ , arg ] ... )
```

The conversion character *%* used in *printf* is replaced by the sequence *%digit$*, where *digit* is a decimal digit *n* in the range 1-9. The conversion should be applied to the *n*th argument, rather than to the next unused one (you specify which parameter you want this conversion applied to). All other aspects of formatting are unchanged. All conversions must contain the *%digit$* sequence, and it is your responsibility to make sure the numbering is correct. All parameters must be used exactly once.

See also *printf(3S)*.

Example

The following example prints a language-independent date and time format.

```
printmsg(format, weekday, month, day, hour, min);
```

For American usage *format* would be a pointer to the string:

```
"%1$s, %2$s %3$d, %4$d:%5$.2d"
```

producing the output:

```
Sunday, July 3, 10:02.
```

For German usage, *format* would be a pointer to the string:

```
"%1$s, %3$d %2$s %4$d:%5$.2d"
```

which outputs:

```
Sonntag, 3 Juli 10:02.
```

Note that the values of the strings are not modified, only the order. If the German format is used with the American data, the output would be:

```
Sunday, 3 July 10:02
```

Convert string to double precision number

```
nl_strtod(str, ptr, langid)
nl_atof(str, langid)
```

The *nl_strtod* and *nl_atof* commands are similar to the standard routines, *strtod* and *atof*, but use *langid* to determine the radix character. If *langid* is not valid, or information for *langid* has not been installed, the radix character defaults to a period.

See *strtod(3C)* for these commands.

Multi-byte Library Routines

Multi-byte library routines and macros were created to help you parse multi-byte character data. The library routines are: *langinit, firstof2, secof2, byte_status*, and *charadv.* The macros are: `FIRSTof2`, `BYTE_STATUS`, `CHARADV`, `SECof2`, `CHARAT`, `ADVANCE`, `PCHAR`, `PCHARADV`, and `CHARADV`

The routine, *langinit*, must be called before any of the other 16-bit tools are called. *langinit* initializes a table specific to the specified language name. The rest of the 16-bit tools parse data to determine if the character is 1 or 2 bytes, advance the pointer to the next character, or return a character.

Refer to the *nl_tools_16(3c)* manual page for information on these commands.

Application Guidelines

When writing an application program, do not use hard-coded message statements. Store all messages to the user in a separate message catalog where they can be accessed via NLS library commands. This allows users who prefer other native languages to modify the messages to to fit their own needs.

The library routines provided for NLS guarantee correct and standard conversions to formats in all supported native languages. You can also create any formats or tables that are beyond those supported by HP to fit your specific needs.

Example C Programs

Here are two example C programs that show how to use some of the library routines described in this chapter.

Example 1

This C program is representative of changes to *ctime* that adapt it for NLS. The library routines *nl_conv(3C)*, *nl_ctype(3C)*, and *nl_string(3C)* are handled in a similar manner.

```
#include <langinfo.h>
main ()
     {
      int langid;
      long timestamp;

      langid = currlangid();

      time(&timestamp);
      printf("%s", ctime(&timestamp));
      printf("%s", nl_ctime(&timestamp, "", langid));
     }
```

The command lines used are:

```
LANG=american
export LANG
cc test_ctime.c -o test_ctime
test_ctime
```

The output is:

```
 Tue Apr 24 15:56:34 1990
 Tue, Apr 24, 1990 15:56:34 PM
```

The command lines to change the language to French are:

```
LANG=french
export LANG
test_ctime
```

The output is:

```
Tue Apr 24 15:56:34 1990
mar 24 avr 1990 15H56 34
```

Example 2

This C program uses the *printmsg(3C)* routines to output the same message in a variety of ways.

```
#include <stdio.h>
main()
     {
     char *a = "Hello,";
     char *b = "world!";
     char buf[100];

     printf("Hello, world!\n");
     printf("%s %s\n", a, b);

     printmsg("Hello, world!\n");
     printmsg("%1$s %2$s\n", a, b);
     printmsg("%2$s %1$s\n", a, b);

     fprintf(stdout, "Hello, world!\n");
     fprintf(stdout, "%s %s\n", a, b);

     fprintmsg(stdout, "Hello, world!\n");
     fprintmsg(stdout, "%1$s %2$s\n", a, b);
     fprintmsg(stdout, "%2$s %1$s\n", a, b);

     sprintf(buf, "Hello, world!\n");
     printf("%s", buf);
     sprintf(buf, "%s %s\n", a, b);
     printf("%s", buf);

     sprintmsg(buf, "Hello, world!\n");
     printf("%s", buf);
     sprintmsg(buf, "%1$s %2$s\n", a, b);
     printf("%s", buf);
     sprintmsg(buf, "%2$s %1$s\n", a, b);
     printf("%s", buf);
     }
```

The command lines used are:

```
cc test_pmsg.c -o test_pmsg
test_pmsg
```

The output is:

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
world! Hello,
Hello, world!
Hello, world!
Hello, world!
Hello, world!
world! Hello,
Hello, world!
Hello, world!
Hello, world!
Hello, world!
world! Hello,
```

Notes

Message Catalog System 4

This chapter explains how localized message files are created and updated, where they are kept, and naming conventions.

Introduction to the Message Catalog System

To simplify the localization process, applications programmers should write programs that do not require recompiling when they are localized. If the code can remain unmodified, the functionality of an application is not affected when translations are made. This reduces support problems because only one version of the application exists. This also minimizes the possibility of introducing additional bugs into the product and reduces the time required to localize.

Localizable programs use text (prompts, commands, messages) from an external message catalog file. This allows text to be translated (part of the localization process) without modifying program source code or recompiling. If the external message catalog file is inaccessible for any reason (such as accidentally removed or not yet created), you can either use the internally stored messages written in the original language or you can write your program to default to the `n-computer` message catalog when the desired language's message catalog is missing.

A message catalog system is used to separate strings such as prompts and messages from the main code of a program. This makes it very easy for another country to translate the information and have the program run properly without modifying the program's source code. The HP-UX message catalog system uses HP-UX commands to help create the catalogs and C library routines to access those catalogs. Message catalog commands work only with the C programming language, but the library routines can be accessed from C, Pascal, and FORTRAN programs.

The message catalog commands are:

- *findstr* - find strings for inclusion in message catalogs
- *gencat* - generate a formatted message catalog file
- *insertmsg* - use output from *findstr* to both create a preliminary message file and to create a new C program with calls to the message file (*getmsg* calls).

The C library routines specific to message catalogs are:

- *getmsg* - get a message from the catalog
- *printmsg, fprintmsg, sprintmsg* - print formatted output with numbered arguments

The steps an applications programmer would take to simplify the localization process are:

- modify existing programs using *findstr, insertmsg*
- maintain message catalogs using *findmsg, gencat*
- translate message catalogs using *dumpmsg, gencat*

Creating a Message Catalog

To make a program easier to localize, string literals such as the error messages and prompts should be placed in a separate file that is accessed by the program at run time. (Hard-coded messages can be left in; they are useful in source for clarifying code.) This way a program can easily access any localized message file without modification of the program. Hewlett-Packard has developed a set of tools to extract print statements from C programs. This set of tools is referred to as the **Message Catalog System**.

Preview: Incorporating NLS into Commands

The general flow of the message catalog system is diagrammed in Figure 4.1. The three HP-UX commands: *findstr, insertmsg,* and *gencat* extract messages from C programs and build a message catalog. The filenames are *prog.c, prog.str, prog.msg,* and *prog.cat.* (They can be named anything you prefer. Names, discounting the *.c* suffix, should be equal to or less than 9 characters in length. The suffixes used here are only a suggested naming convention.)

The name *prog.c* represents any C program containing hard-coded messages. The name *prog.str* represents an intermediate file containing all strings from the source file surrounded by double quotes (" "). The new C program is named *nl_prog.c* (where *prog.c* is the original C program) that includes a message file instead of hard-coded messages. The final object file produced by compiling *nl_prog.c* is *prog.* The file *prog.msg* contains the numbered messages and sets that are used to generate the final message file. The final message file is *prog.cat.*

An example session is described later in this chapter in the section "7 Steps to Prelocalize an Example Program".

Following the Flow

The next sections describe in detail the steps used when creating a message catalog (see Figure 4-1).

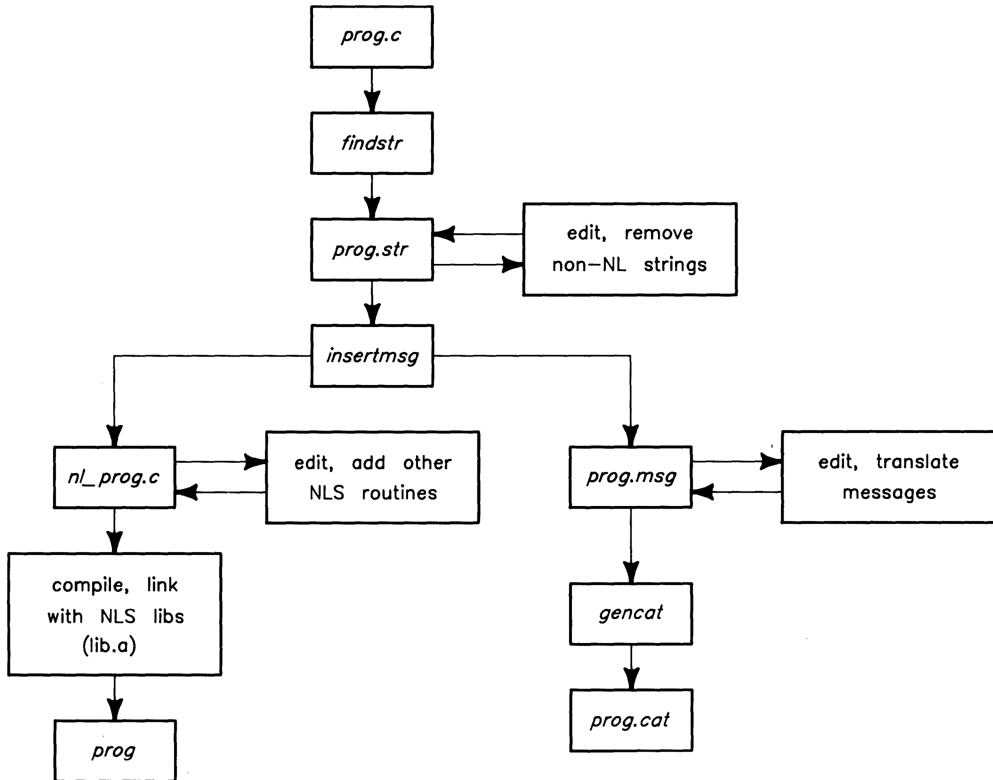


Figure 4-1: Flow of the Message Catalog

findstr

findstr examines files of C source code (*prog.c* in this case) for string constants that do not appear in comments. These strings, along with the surrounding quotes, are placed on standard output. Each extracted string is preceded by the file name, start position in the file, and string length. The output should be redirected to a file for editing.

Syntax

```
findstr prog.c > prog.str
```

prog.str

prog.str, the output from *findstr* which is created when the user redirects output from *findstr* into a file, contains all quoted strings that do not appear in comments from the C program (*prog.c*) used as input to *findstr*. This includes error messages, format statements, system calls, and anything else that is surrounded in double quotes. Preceding the strings is a copy of the filename (*prog.c*), from which the strings came, followed by the *byteposition* and *bytecount*. The file *prog.str* can be called any name. Message files should contain nothing but messages, so you must edit *prog.str* to remove all other types of quoted strings.

Format

prog.c *byteposition* *bytecount* "string"

The parameters *byteposition* and *bytecount* apply to the source program at the time *findstr* is run. Any changes to *prog.c* will invalidate these numbers. Do not modify these parameters.

insertmsg

insertmsg uses *prog.c* and *prog.str* to create both the new C source file (*nl_prog.c*) and a file (*prog.msg* which is redirected standard out) containing the messages for translation into local languages. *prog.msg* is used by *gencat*.

Syntax

```
insertmsg prog.str > prog.msg
```

Here, *prog.str* is the edited output from *findstr* (see above section on *prog.str*). The routine *insertmsg* creates a new file (*nl_prog.c*), for each file named in *prog.str*. For this example, all the lines in *prog.str* refer to *prog.c*.

These lines

```
#ifndef NLS
#define nl_msg(i, s) (s)
#else NLS
#define NL_SETN   1  /* set number */
#include <msgbuf.h>
#endif NLS
```

are inserted by *insertmsg* at the beginning of each new file (in this case *nl_prog.c*). Then for each line in *prog.str*, it surrounds the string with an expression of the form:

```
nl_msg(1, "Hello, world\n");
```

where *1* is the message number.

This is expanded at runtime by a macro in *msgbuf.h.* Then *insertmsg* redirects (to standard out) message catalog source. This is generally redirected into a file so that *gencat* can be used to generate the actual message catalog. If *insertmsg* doesn't find the opening or closing double quote where it expects it in *prog.str*, it prints "`insertmsg exiting : lost in strings file`" and dies. If this happens check the strings file to make sure that the lines kept there haven't been altered. Rerunning *findstr* on *prog.c* reconstructs *prog.str* to its unedited form.

output from insertmsg

There are two branches from *insertmsg*: the new ".c" file (*nl_prog.c*) and the messages going to **stdout** (assumedly redirected into a file, referred to here as *prog.msg*).

nl_prog.c

This is the new source of your program. It consists of all the source in the original program, with the messages in *prog.str* changed to be of the form shown above, and an additional *#define* and *#include* statement at the beginning of the file.

You must now edit the file *nl_prog.c* to insert the following:

```
#ifdef NLS
nl_catopen("prog");
#endif NLS
```

where *prog.cat* is the final message file (*.cat* is appended to *prog* by the *nl_catopen* macro). If a set number other than *1* is desired (for merging several message catalog files, separating them by set number only), change the *NL_SETN* define statement shown in the previous section's code, accordingly.

prog.msg

This is what *insertmsg* places on **stdout** to be used as the input to *gencat.* This file needs to be edited to define the `$set` number to match the `NL_SETN` in *nl_prog.c* (i.e. you must insert the `$set` line). Messages in this file are automatically numbered from `1` upward, in the same order as they appear in the file *prog.str.* The same number is placed in the call to *nl_msg* (the macro placed around the message by *insertmsg*).

findmsg also generates this same output on standard out. Although, unlike *insertmsg*, it does not produce a modified C source file. Instead, it acts on the modified source file (*nl_prog.c* in the example in figure 4-1).

Example of a modified *prog.msg* file

```
$set 1
1 Good morning
2 error, monday morning
$set 2
15 Hello, world!
16 Thank goodness it's Friday!!
17 CRASH
```

gencat

gencat generates a formatted message catalog (*prog.cat*) from the information in *prog.msg.*

Syntax

```
gencat prog.cat prog.msg ...
```

The *prog.msg* file consists of sets of messages along with comments which are merged into a formatted file (*prog.cat*) that *getmsg* can access. If *prog.cat* does not exist, it is created. If it exists, the new messages are included in the original *prog.cat* unless the set and message numbers collide, in which case the new supersedes the old. See the section on *prog.msg* for details on the input file format. If a message source line has a number but no text then the existing message corresponding to this number is deleted from the catalog.

prog.cat

prog.cat is the final message catalog, created by *gencat*, which is then accessed by the new source program. *gencat* is a binary file and cannot be read directly by a user.

The file *prog.cat* will be stored as */usr/lib/nls/n-computer/prog.cat* where *n-computer* is the value of LANG when this file is accessed and "prog" is the program name string entered into the *nl_catopen* statement. You must be logged in as super user to place the file in that directory.

Multiple commands may share the same physical file or share the same name in the *nl_catopen* macro. Each message catalog name (program name with *.cat* appended) must be linked to the same file. Messages can be distinguished, either by set number or by message number.

prog

prog is the object file produced by compiling *nl_prog.c*. Do not confuse this file with "prog" called by *nl_catopen* that has *.cat* appended.

Format of Source Message File

All lines in the message file must begin in column 1. The source message catalog may contain lines of the following form:

$set n *comment*

This line, followed by the message text lines, specifies the set number of the following messages until the next **$set**, **$delset**, or end of file appears. The *n* denotes the set number (1 to 255). Set numbers must be in ascending order within a single source file. Any string following the set number is treated as comment.

$delset n *comment*

This line deletes an entire message set from the existing catalog file. The *n* denotes the set number (1 to 255). Any string following the set number is treated as comment. Set numbers must be in ascending order within a single source file.

To delete an entire message set, place the directive

```
$DELSET set_number
```

at the beginning of a line between sets.

$ *comment*
This line is used as a comment line.

m *message text*
The *m* denotes the message number (1-32767). If *message text* exists, the message is stored in the catalog file with the set number specified by **`$set`** and message number *m*. If the *message text* does not exist, the message corresponding to the set number and message number is deleted from the existing catalog file. Message numbers must be in ascending order within a single set.

Certain special characters are used in the text strings; certain non-graphic characters and the backslash "\" can be specified using the escape sequences shown in table 4-1:

Table 4-1: Escape Sequences

Description	Symbol	Sequence
newline	NL(LF)	\n
horizontal tab	HT	\t
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
bit pattern	ddd	\ddd

The escape sequence *ddd* consists of backslash followed by 1, 2, or 3 octal digits which are used to specify the value of the desired character. If the character following a backslash is not one of those specified, the backslash is ignored. Backslash "\" is also used to continue a string to the next line. The following two lines are considered a single string:

```
1 This line continues \
to the next line.
```

which is equivalent to:

```
1 This line continues to the next line.
```

Note that, in this case, backslash "\" must immediately precede the newline character.

Printmsg, Fprintmsg, and Sprintmsg

The library routines *printmsg, fprintmsg,* and *sprintmsg* are derived from their counterparts in *printf(3S)*, with the understanding that the conversion character *%* is replaced by the sequence *%digit$*. *Digit* is the decimal *n*, in the range 1 to 9, and indicates that this conversion should be applied to the *n*th argument, rather than to the next unused one. All conversion specifications must contain the *%digit$* sequence, and numbered correctly. All parameters must be used exactly once. These commands are used to handle the message catalog system with messages, having two or more parameters, that may need to be reordered.

Accessing Applications Catalogs

Message catalogs are accessed from any supported language program, such as C, Pascal, or FORTRAN, using C library routines. These C library routines consist of some new library functions and some altered, pre-existing C library routines.

All HP-UX shell commands and C library routines that are associated with NLS or that have been changed due to NLS are documented in the *HP-UX Reference.*

To use the C library routines from a Pascal or FORTRAN program please refer to the relevent language and portability manuals.

File System Organization and Catalog Naming Conventions

Any application that has been localized into several languages has separate message catalogs (files) for each language. The routine *nl_catopen* assumes the message file is under */usr/lib/nls/$LANG/filename.cat* where *$LANG* is the the language contained in the LANG environmental variable and *filename* is the name of the file specified in the call to *nl_catopen* in the source program (usually the program name).

Only the root user can write in the directory */usr/lib/nls.*

For example, original, unlocalized data might be stored in a file whose full path name is */usr/lib/nls/n-computer/prog.cat.* The file */usr/lib/nls/german/prog.cat* would contain the same data modified for German, and */usr/lib/nls/spanish/prog.cat* would contain Spanish data. It is the responsibility of the application program to determine (at run time) which file to open.

Prelocalization: Adding Native Language Support

Suppose you have the following C program, *hello.c*, and you want to localize the output. The source file of *hello.c* looks like this:

```
main()
/* This program prints a greeting and the date */
{
printf("hello, world\n");
system("date");
}
```

7 Steps to Prelocalize an Example Program

1. **Execute** *findstr*, redirecting the output to *hello.str.*

   ```
   $findstr hello.c > hello.str
   ```

2. **Edit** *hello.str.* The file *hello.str* contains all the strings from *hello.c* that are surrounded by double quotes. It contains the following lines:

   ```
   hello.c 67 16 "hello, world\n"
   hello.c 93 6 "date"
   ```

   The file *hello.str* needs to be edited so it contains only messages that should appear on the screen. Notice that *date* is enclosed with double quotes, but should **not** be included in the message file. Edit *hello.str* so it contains only the line:

   ```
   hello.c 67 16 "hello, world\n"
   ```

3. **Execute** *insertmsg*, redirecting output to a file called *hello.msg.*

   ```
   insertmsg hello.str > hello.msg
   ```

   In addition to the messages output to *hello.msg*, *insertmsg* creates the new source file, *nl_hello.c*, which contains the original source plus a new *#define* line and *#include* line, plus an altered message line. Your directory should now contain the following files relating to this example:

   ```
   hello.c     hello.msg     hello.str     nl_hello.c
   ```

4. **Edit** *nl_hello.c.* The file currently looks like:

```
#ifndef NLS
#define nl_msg(i, s) (s)
#else NLS
#define NL_SETN   1 /*set number*/
#include <msgbuf.h>
#endif NLS
main()
/* This program prints a greeting and the date */
{
     printf((nl_msg(1, "hello, world\n")));
     system("date");
}
```

The macro *nl_msg* is expanded at compile time (see section on *insertmsg*). Both the set number and the message number is set to *1.*

The file needs to be edited so it refers to the final message file. Decide now what you want to call the final message file (in this example it will be called *hello.cat*) and insert the folloowing lines into the program body of *nl_hello.c* (within *main()*):

```
#ifdef NLS
nl_catopen("hello");
#endif NLS
```

The above lines open a file called *hello.cat* in a directory corresponding to the native language defined in the LANG environmental variable. If LANG is not defined, the hard-coded messages in the source are used. This means that you never need to change the source code. You simply need to change the value of LANG and create a message file stored in */usr/lib/nls/$LANG/hello.cat* if you wish to localize *hello.c* for a new language.

The final source file looks like this:

```
#ifndef NLS
#define nl_msg(i, s) (s)
#else NLS
#define NL_SETN   1 /*set number*/
#include <msgbuf.h>
#endif NLS
main()
/* This program prints a greeting and the date */
{
#ifdef NLS
nl_catopen("hello");
#endif NLS
printf((nl_msg(1, "hello,world\n")));
system("date");
}
```

5. **Edit** *hello.msg* to define **$set** to match the set number in *nl_hello.c*, if different. It should be the same unless you are creating a message file other than the one created by *insertmsg.* The file *hello.msg* looks like:

```
$set 1
1 hello, world\n
```

6. **Execute** *gencat* specifying the file *hello.cat* used in step 4 as the output file. The input file is *hello.msg.*

```
gencat hello.cat hello.msg
```

   the file *hello.cat* should then be moed to */usr/lib/nls/n-computer/hello.cat.*

   Note: unless you (or your system administrator) change the access permissions you must be superuser to write under the */usr/lib/nls* directories.

7. Compile *nl_hello.c* to include the NLS code. For example:

```
cc -DNLS -o hello nl_hello.c
```

   Often the modified source becomes the master copy. This may be done by moving the "nl" version to the original, like this:

```
mv nl_prog.c prog.c
```

   This destroys the original.

Localization

You now have a localizable program. If your native language is English, you also have a localized message file. If your native language is something other than English, you still need to localize the message file. Let's say your native language is German, and rather than printing the message "`hello, world`" to the screen, you wish to print "`Guten Tag Welt, wie geht es dir?`".

Edit *hello.msg* or create a new file to read:

```
$set  1
1 Guten Tag Welt, wie geht es dir?\n
```

Execute *gencat* by typing in:

```
gencat hello.cat hello.msg
```

Store the new *hello.cat* message file in */usr/lib/nls/german/hello.cat* and change your LANG environment variable to *german.*

When you re-execute the program, it will automatically use the German message file rather than the American English message file. Execute *hello* to verify that it works. If the LANG variable is not defined, or the message catalog does not exist, the hard-coded message will appear. While this discussion and example was done in English, there is no reason that the same exercise could not be done with German literals which must then be translated into English or another language.

Maintaining Programs and Message Catalogs

Generally programs are larger than the simple examples used here. Rather than separately maintaining source for the program (*nl_prog.c* as originally named or *prog.c* if moved to replace the original) and the message catalog (*prog.msg*), the message catalog can be extracted from program source using *findmsg(1).* Unlike *findstr*, *findmsg* returns only those messages which are to be taken from the message catalog.

Message catalog source (*prog.msg*) can be extracted or **uncompiled** from a message catalog (*prog.cat*) using *dumpmsg(1).* This is useful in situations where C program source code is not available. Also if an error is found in the message catalog, *dumpmsg* can be used to get message catalog source. Then the correction can be made to that source and the corrected messages **compiled** back in using *gencat.* (If this message catalog was extracted from C program source, it should be corrected also.)

Notes

Native Language Support Library and Commands A

Library calls and commands, described in the appropriate *HP-UX Reference* manual pages, have been added to HP-UX to facilitate the development of fully localized programs.

Library Routines

The NLS library routines are included in the standard C library */usr/lib/libc.a*. Table A-1 lists the C library routines for NLS. For more details refer to the appropriate page in the *HP-UX Reference* or the the chapter "Programming with Native Language Support".

Table A-1. NLS Library

Name([1])	Description
catread(3C)	adds MPE/RTE style support to getmsg
nl_ctime	time conversion routines (see *ctime(3C)* manual page)
nl_gcvt	convert binary numbers to string numerics (see *ecvt(3C)* manual page)
nl_conv(3C)	character casefolding routines
nl_ctype(3C)	character classification
getmsg(3C)	get native language message from catalog
langinfo(3C)	get native language information
nl_string(3C)	string comparison routines
printmsg, fprintmsg, sprintmsg	print formatted numeric output (see the *printmsg(3C)* manual page).
nl_strtod, nl_atof	convert string numeric to binary number routines (see the *strtod(3C)* manual page).
langinit	initialize the table for multi-byte parsing (see the *nl_tools_16(3C)* manual page).
firstof2, secof2	returns true of byte is first (or second) of 2-byte character (see the *nl_tools_16(3C)* manual page).
byte_status	returns 0, 1, or 2 indicating single byte character, second byte of 2-byte character, or first byte of 2-byte character. (see the *nl_tools_16(3C)* manual page).

[1] The location of this command in the *HP-UX Reference.*

Other HP-UX system and library calls are 8-bit compatible, with the following exceptions. Localized versions exist for many of these (shown in table A-1) and should be used for new program development.

Table A-2. Non-NLS HP-UX System and Library Calls

Name([1])	Description
atof(3C)	convert ASCII string numerics to various binary forms
conv(3C)	ASCII character casefolding routines
ctime(3C)	date and time conversion routines
ctype(3C)	character classification routines
ecvt(3C)	convert binary number to ASCII string numeric
qsort(3C)	quick sort
regex(3X)	regular expression compile/execute
string(3C)	character string operations

[1] The location of this command in the *HP-UX Reference*.

Commands

The commands listed in table A-3 were created by Hewlett-Packard specifically for NLS. They are described in more detail in the appropriate manual page in *HP-UX Reference* and in the chapter "Message Catalog System".

Table A-3. NLS Commands

Name([1])	Description
dumpmsg	Reverse the effect of *gencat*; take a formatted message catalog and make a modifiable message catalog source file (see the *findmsg(1)* manual page).
findmsg(1)	Extract strings from prelocalized C programs for inclusion in message catalogs.
findstr(1)	Find strings in programs not previously localized for inclusion in message catalogs.
gencat(1)	Generate a formatted message catalog file.
insertmsg(1)	Uses output from *findstr* to both create a preliminary message file and to create a new C program with calls to the message file.

Other HP-UX commands may have NLS support to some degree. In the *HP-UX Reference* entry for a command, there is a catagory called "International Support". This category indicates to what level the command supports NLS. The possible values are:

8-bit data — The command accepts, and correctly processes, files containing 8-bit data. For example, *vi*.

16-bit data — The command accepts, and correctly processes, files containing 16-bit data. For example, *vi*.

8-bit filename — The command accepts files with names written in an 8-bit language. For example, *cut*.

16-bit filename — The command accepts files with names written in a 16-bit language. For example, *vi*.

custom — The command formats output appropriate to the user's language. For example, the *date* command formats the output to the language set in the `LANG` environment variable.

[1] The location of this command in the *HP-UX Reference*.

messages — The command has the capability of accessing a message catalog. For example, *cat.*

8-bit string — The command will correctly parse through 8-bit strings and comments. An example of this is *cc*, which allows source files to have 8-bit strings in comments.

NLS Files

In addition to library routines and commands, one system file was created for NLS. The file, *tztab*, is a time zone adjustment table for *date* and *ctime*. See the *HP-UX Reference* for more details.

Notes

Character Sets B

This section provides the table for the following character sets:

- ASCII
- ROMAN8
- KANA8

Table B-1. ASCII Character Set

				b_7	0	0	0	0	1	1	1	1
				b_6	0	0	1	1	0	0	1	1
				b_5	0	1	0	1	0	1	0	1
b_4	b_3	b_2	b_1		0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	‘	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	’	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Table B-2. ROMAN8 Character Set (ID=8U)

				b8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				b7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				b6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				b5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
b4	b3	b2	b1		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	NUL	DLE	SP	0	@	P	‘	p				¯	â	Å	Á	Þ
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q			À	Ý	ê	î	Ã	þ
0	0	1	0	2	STX	DC2	"	2	B	R	b	r			Â	ý	ô	Ø	ã	•
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s			È	°	û	Æ	Đ	µ
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t			Ê	Ç	á	å	ð	¶
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u			Ë	ç	é	í	Í	¾
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v			Î	Ñ	ó	ø	Ì	—
0	1	1	1	7	BEL	ETB	’	7	G	W	g	w			Ï	ñ	ú	æ	Ó	¼
1	0	0	0	8	BS	CAN	(	8	H	X	h	x			´	¡	à	Ä	Ò	½
1	0	0	1	9	HT	EM	)	9	I	Y	i	y			`	¿	è	ì	Õ	ª
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z			^	¤	ò	Ö	õ	º
1	0	1	1	11	VT	ESC	+	;	K	[	k	{			¨	£	ù	Ü	Š	«
1	1	0	0	12	FF	FS	,	<	L	\	l	|			˜	¥	ä	É	š	■
1	1	0	1	13	CR	GS	-	=	M	]	m	}			Ù	§	ë	ï	Ú	»
1	1	1	0	14	SO	RS	.	>	N	^	n	~			Û	ƒ	ö	ß	Ÿ	±
1	1	1	1	15	SI	US	/	?	O	_	o	DEL			₤	¢	ü	Ô	ÿ	

Table B-3. KANA8 Character Set (ID=8H)

b4	b3	b2	b1		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				b8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				b7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				b6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				b5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	0	0	0	0	NUL	DLE	SP	0	@	P	‘	p				ー	タ	ミ		
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q			。	ア	チ	ム		
0	0	1	0	2	STX	DC2	"	2	B	R	b	r			「	イ	ツ	メ		
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s			」	ウ	テ	モ		
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t			、	エ	ト	ヤ		
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u			・	オ	ナ	ユ		
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v			ヲ	カ	ニ	ヨ		
0	1	1	1	7	BEL	ETB	’	7	G	W	g	w			ァ	キ	ヌ	ラ		
1	0	0	0	8	BS	CAN	(	8	H	X	h	x			ィ	ク	ネ	リ		
1	0	0	1	9	HT	EM	)	9	I	Y	i	y			ゥ	ケ	ノ	ル		
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z			ェ	コ	ハ	レ		
1	0	1	1	11	VT	ESC	+	;	K	[	k	{			ォ	サ	ヒ	ロ		
1	1	0	0	12	FF	FS	,	<	L	¥	l	|			ャ	シ	フ	ワ		
1	1	0	1	13	CR	GS	-	=	M	]	m	}			ュ	ス	ヘ	ン		
1	1	1	0	14	SO	RS	.	>	N	^	n	~			ョ	セ	ホ	゛		
1	1	1	1	15	SI	US	/	?	O	_	o	DEL			ッ	ソ	マ	゜		

Peripheral Configuration C

European Character Sets

For European languages, many HP peripherals support the Hewlett-Packard ROMAN8 character set. ROMAN8 is a full superset of ASCII and offers 88 additional local language symbols. Older HP peripherals may use the HP **Roman Extension** set, which is a subset of ROMAN8. Roman Extension is missing ROMAN8 Characters À thru Ï, Ù, Û, Ç. ¥ , § . *f* . Á thru ±.

See the ROMAN8 character set in the appendix "Character Sets".

Japanese Character Sets

Many HP peripherals support an alternate 8-bit character set known as KANA8. The first 128 codes in the KANA8 set are JASCII (same as ASCII except substitutes "¥" for "\") and the last 128 codes are Katakana.

ISO 7-bit Substituion

ISO7 stands for International Standards Organization 7-bit character substitution. For each ISO7 language, certain ASCII character codes infrequently used in ordinary text (such as those for "|" and "{") are designated to generate different local-language symbol (such as "ø" or "æ" in Danish). Unfortunately, the designated ASCII codes represent special characters often used in HP-UX (and all other UNIX and UNIX-like systems). The use of ISO 7-bit substitution is neither recommended nor supported.

Character Set Support by Peripherals

ROMAN8 terminals can simultaneously display any characters in their set. Their keyboards have keycaps only for the specified local language, but you can enter any ROMAN8 character by use of the [Extend char] key. You can also use most 8-bit terminals in ISO7 mode (see discussion above).

Plotter ROM (internal) fonts are normally used for **draft-quality** plots. **Final** plots are normally done with host-generated (software) vector fonts. DGL/9000 graphics presently generate only ASCII characters.

Some printers are capable of context-sensitive letters, so some shapes may vary.

The following tables summarize the character set support of HP 9000 peripherals. Not all peripherals are available on all HP 9000 computers; check with your HP Sales Representative. Also, this list may not be complete. Again, check with your HP Sales Representative for new peripherals, or new options to existing peripherals. The **Ordering Information** column indicates what action you must take to obtain a peripheral which is not ASCII.

Table C-1. 8-Bit Terminals

Peripheral Device	Character Set(s) Support	Ordering Information	Comments
HP 98700H Display Sta.	ASCII only	Product suffix	
HP 110	ROMAN8 Std.	Product suffix	
HP 45610B (HP 150)	ROMAN8 Std.	Product suffix	
HP 45650BV (HP 150)	ARABIC8 Std.	Product suffix	
HP 45650BT (HP 150)	HEBREW8 Std.	Product suffix	
HP 2392A	Roman extension, ARABIC8 Std.	Keyboard option	Missing Á thru ±.
HP 2393A	ROMAN8 Std.	Keyboard option	
HP 2397A	ROMAN8 Std.	Keyboard option	
HP 2622A	Roman Ext. Std.	Keyboard option	
HP 2622J	KANA8 Std.		Cannot combine an accent with a vowel.
HP 2623A	Roman Ext. Std.	Keyboard option	Cannot combine an accent with a vowel.
HP 2623J	KANA8 Std	NA	Cannot combine an accent with a vowel.
HP 2624B Terminal	--------	-----------	Not recommended for NLS
HP 2625A Terminal	ROMAN8 Std.	Keyboard option	
HP 2626A/W	Roman Ext. Std.	Keyboard option	Cannot combine an accent with a vowel.

Table C-1. 8-Bit Terminals (Cont.)

Peripheral Device	Character Set(s) Support	Ordering Information	Comments
HP 2627A	Roman Ext. Std.	Keyboard option	
HP 2628A	ROMAN8 Std.	Keyboard option	
HP 2647F Terminal	ASCII only	NA	
HP 2703A Terminal	Roman Ext. Std.	Keyboard option	

Table C-2. 16-Bit Terminals

Peripheral Device	Character Set(s) Support	Ordering Information	Comments
HP 35714A	Character Mode Kanji Terminal	NA	

Table C-3. 8-Bit Printers

Peripheral Device	Character Set(s) Support	Ordering Information	Comments
HP 2225A *ThinkJet*	ROMAN8, ARABIC8, HEBREW8 Std.	NA	
HP 2563A Printer	ROMAN8 Std.	NA	
HP 2565A Printer	ROMAN8 Std.	NA	
HP 2566A Printer	ROMAN8, KANA8, ARABIC8 Std.	NA	
HP 2601A Printer	Substitution	Accessory	Change print wheel
HP 2602A Printer	Substitution	Accessory	Change print wheel
HP 2603A Printer	ROMAN8 Std.	NA	
HP 2608S Printer	Roman Ext., KANA8 Std.	Option 002	
HP 2631B	Roman Ext., KANA8 Std.	Formerly Option 009	
HP 2631G	ROMAN8, KANA8 Std.	NA	
HP 2671A/G	ROMAN8, KANA8 Std.	NA	
HP 2673A	Roman Ext. Std.	NA	
HP 2680A	Roman Ext. Std.	NA	Series 500 only
HP 2686A *LaserJet*	ROMAN8 Std.	Font cartridges may be available.	
HP 2686A *LaserJet +*	ROMAN8 Std.	Downloadable fonts, font cartridges may be available.	
HP 2688A	ROMAN8 Std.	Downloadable fonts.	Series 500 only, not all fonts ROMAN8

Table C-3. 8-Bit Printers (Cont.)

Peripheral Device	Character Set(s) Support	Ordering Information	Comments
HP 2932A	ROMAN8, KANA8 Std.	NA	
HP 2933/34A	ROMAN8, KANA8 Std.	Font cartridges are available for Arabic, Hebrew, Greek, Turkish	
HP 82906A	ROMAN8 Std.	NA	
HP 97090A	Roman Ext. Std.	NA	Series 500 only
HP 9876A	Roman Ext. Std.	NA	

Table C-4. 16-Bit Printers

Peripheral Device	Character Set(s) Support	Ordering Information	Comments
HP 35713	Kanji	NA	
HP 35719A	Kanji	NA	Does not support HP-16
HP 35720A	Kanji	NA	Does not support HP-16
HP 4163A	ROMAN8, KANA8, Japanese	NA	

Table C-5. 8-Bit Plotters

Peripheral Device	Character Set(s) Support	Ordering Information	Comments
HP 7470A	ISO7 only	NA	
HP 7475A	ISO7 only	NA	
HP 7580A	ISO7 only	NA	
HP 7585A	ISO7 only	NA	
HP 7586A	ISO7 only	NA	

Glossary

16-bit character sets — a character set that uses two bytes to encode characters. This allows representation of up 35 344 characters, as would be needed to support Chinese, Japanese, and Korean languages.

8-bit character sets — a character set that uses all eight bits of a single byte to encode characters. These character sets are designed so the range 0 to 127 are ASCII, with the exception of the "\" character in Kana8 which is replaced by the yen symbol. Non-ASCII characters appear in the range 161 to 254.

applications program — a program performs a specific application.

applications programmer — a person who writes programs for an end-user.

ASCII — American Standard Code for Information Interchange. A 128-character set represented by 7-bit binary values. (ASCII does not define the value of the eighth bit.)

bit — a contraction of BInary digiT. A bit can have a value of 0 or 1.

byte — a unit of data storage consisting of 8 bits. A byte can represent one ASCII, KANA8, GREEK8, TURKISH8, or ROMAN8 character.

character — a language unit, usually consisting of 7 (ASCII), 8 (KANA8, ROMAN8, GREEK8, TURKISH8), or 16 (JAPAN15) bits.

character set — a grouping of graphic (visible) symbols and control characters, each represented by a unique binary value occupying a fixed amount of storage. Character sets contain the necessary alphanumeric and other characters required to support languages.

collating sequence — the ordering sequence assigned to characters or a group of characters when they are sorted and ordered by a computer.

command — a program which is executed by the shell command interpreter. Arguments following the command name are passed to the command program. You can write your own command programs, either as compiled programs or as shell scripts (written in the shell command language).

command interpreter — a program that reads lines typed at the keyboard or from a file, and interprets them as requests to execute other programs. The command interpreter for HP-UX is called the shell.

comment — an expression used to document a program or routine that has no effect on the execution of the program.

compiler — a program that translates a high-level language into machine-dependent form.

control character — a member of a character set that produces action in a device other than a printed or displayed character. In ASCII, control characters are those in the code range 0 thru 31, and 127. Most control characters are generated by simultaneously pressing a displayable character key and [CTRL].

default search path — the sequence of directory prefixes that *sh, time,* and other HP-UX commands apply when searching for a file known by an incomplete path name. It is defined by PATH in *environ*. *login* sets PATH = *:bin:/usr/bin*, which means that your working directory is the first directory searched, followed by */bin*, followed by */usr/bin*.

device — a piece of peripheral equipment, usually used to input or output data.

directory — a file used to catalog other files on a mass storage medium. Each directory contains entries for its own unique files. The directory information includes name, type, length, location, and protection.

downshifting — a pheripheral's provision for producing lowercase letters by using the [Shift] key (on most keyboards).

editor — a program that allows you to create and modify text files based on text and commands entered from a terminal.

end-user — a person who uses existing programs and applications.

environment — the set of conditions (such as your working directory, home directory, and type of terminal you are using) that exist when you log in.

file name — a sequence of 14 or fewer characters which uniquely identifies a file in a directory. Any character except "/" can be used.

GREEK8 — the Hewlett Packard supported 8-bit character set for the Greek language.

hp-8 — Hewlett Packard's implementation of the ISO's (International Standard Organization) 8-bit character code set.

hp-15 — a Hewlett-Packard encoding scheme for 16-bit character sets.

hp-16 — a Hewlett-Packard encoding scheme for 16-bit character sets.

ideogram — the use of graphic symbols to represent ideas.

ideographic — representing an idea by use of a character or symbol rather than a word; the use of ideograms.

Internationalization — the process of making software and hardware usable to users outside the United States. Native Language Support and Localization are two key factors of Internationalization.

ISO7 — International Standards Organization 7-bit character substitution. The character graphics associated with some less-used ASCII codes are changed to other characters needed for a particular language.

JAPAN15 — the Hewlett Packard supported 16-bit character set for the Japanese language.

KANA8 — the Hewlett Packard supported 8-bit character set for support of phonetic Japanese (Katakana).

Kanji — the Japanese ideographic character set based on Chinese characters. The set consists of roughly 50,000 characters.

Katakana — the Japanese phonetic character set typically used in formal writing. The set consists of 64 characters including puntuation.

LANG — the Unix environment variable (LANGuage) that should be set to the American English name of the native language desired.

library — a set of subroutines contained in a file that can be accessed by a user program.

library routine — one of a collection of programs within the HP-UX operating system. Each routine performs a unique task.

local customs — refers to a region's local conventions such as date, time, and currency formats.

localization — the adaptation of software for use in different countries or local environments.

message catalog — the external file containing prompts, responses to prompts, error messages, and mnemonic command names in the user's native language.

message catalog system — a set of tools developed by Hewlett-Packard to extract print statements from C programs and place them in the message catalog.

native language — a person's or user's first language (learned as a child) such as Japanese, Finnish, or American English.

natural language — the spoken or written language as opposed to a computer implementation of a language.

NLS — Native Language Support. The Hewlett-Packard model that provides capabilities for reducing or eliminating the barriers that would make HP-UX difficult to use in a native language.

operating system — a program which manages the computer's resources. It provides the programmer with utilities, including I/O routines, peripheral-handling routines, and high-level languages.

parameter — in a program, a quantity that may be given different values. It is usually used to pass conditions or selected information to a subroutine that is used by different main routines or by different parts of one main routine. Its value frequently remains unchanged throughout any one such use.

path name — a sequence of directory names separated by slashes (/), and ending in a file name (any type).

peripheral — a device connected to the computer's processor that is used to accept information from or provide information to an external environment.

prelocalization — modification to application programs before compilation to make use of language-dependent library routines and to ensure that 8-bit data can be handled properly.

program — a sequence of instructions to the computer, either in the form of a compiled high-level language or a sequence of shell command language instructions in a text file.

prompt — a character displayed by the system on a terminal indicating that the previous command has been completed and the system is ready for another command. It is usually a "$" or "%", but can be redefined to any character string.

psuedo-teletype — a pair of interconnected character devices; a master device and a slave device. Anything written on the master is given to the slave as input and anything written on the slave is presented as input to the master.

pty — abbreviation for psuedo-teletype.

radix character — the actual or implied character that separates the integer portion of a number from the fractional portion.

ROMAN8 — the Hewlett Packard supported 8-bit character set for Europe.

root directory — the highest level directory of the hierarchical file system, in which other directories are contained. In HP-UX, the "/" refers to the root directory.

shell — the shell is both a command language and a programming language that provides the user-interface to the HP-UX operating system.

shell script — a sequence of shell commands and shell programming language constructs, usually stored in a text file, for invocation as a user command (program) by the shell.

space — a blank character. In ASCII a space is represented by character code 32 (decimal).

standard input — the source of input data for a program. The default standard input is the terminal keyboard, but the shell may redirect the standard input to be from a file or pipe.

standard output — the destination of output data from a program. The default standard output is the terminal CRT, but the shell may redirect the standard output to be a file or pipe.

string — a connected sequence of characters, words, or other elements.

supported language — the computer-implemented version of a written or spoken language.

syntax — the rules governing sentence structure in a spoken language, or statement structure in a computer language such as that of a compiler program.

teletype — a trademark for a form of teletypewriter.

teletypewriter — a peripheral for telegraphic data communication with a computer.

TURKISH8 — the Hewlett Packard supported 8-bit character set for the Turkish language.

upshifting
a pheripheral's provision for producing uppercase letters by using the [Shift] key (on most keyboards).

USASCII
A less common name for ASCII. See ASCII.

variable
a storage location for data.

working directory
the directory in which you currently reside. Also, the default directory in which path name searches begin, when a given path name does not begin with "/".

Notes

Index

a

access permissions under */usr/lib/nls* ... 50
accessing NLS features ... 24
active character set ... 14
ADVANCE macro ... 31
alternate character set ... 14
application guidelines ... 32
applications catalogs ... 46
ASCII ... 7, 8, 11

b

16-bit character encoding schemes (HP-15, HP-16) ... 15
16-bit character set ... 11, 15, 16
7-bit character set ... 11
8-bit character set ... 11, 12, 13
8-bit character set support model ... 14
base character set ... 14, 15
byte_status library routine ... 31
BYTE_STATUS macro ... 31

c

C library routines ... 24, 25, 38
case ... 8
character set *(see also 7-bit, 8-bit, 16-bit)*:
  description ... 11, 23
  ID numbers ... 15
  support ... 7
  type ... 22
character type ... 22
charadv library routine ... 31
CHARADV macro ... 31, 31
CHARAT macro ... 31
classify characters ... 28
collating sequence ... 8, 17, 22
control codes ... 11

conv(3C) library routine ... 28
creating a message catalog ... 39, 40
ctime(3C) library routine ... 26
ctype(3C) library routine ... 29
currency ... 9, 17
currlangid library routine ... 27, 28

d

date format ... 17, 26
default native language ... 23
`$delset` ... 44
double precision number ... 31
downshifting ... 17, 22, 23
dumpmsg command ... 24, 51

e

ecvt(3C) library routine ... 26
end user ... 6, 7
environment changes ... 23
escape sequences ... 45
extended character set ... 11
extracting message catalog source ... 51

f

file system organization ... 47
findmsg command ... 24
findstr command ... 24, 38, 39, 41, 48, 51
firstof2 library routine ... 15, 31
`FIRSTof2` macro ... 31
format of source message files ... 44
FORTRAN ... 24, 37, 46
fprintmsg library routine ... 30, 38

g

gencat command ... 38, 39, 41, 42, 43, 44, 50, 51, 244
getmsg command ... 38
getmsg library routine ... 27
GREEK8 character set ... 13, 15, 17
Grenwich Mean Time ... 23

h

hard-coded messges ... 39
header files ... 25
HP-15 ... 15, 16, 17
HP-16 ... 15

i

idtolang library routine ... 27, 28
incorporating NLS into commands ... 39
insertmsg command ... 24, 38, 39, 41, 42, 43, 48
installing optional languages ... 22
internationalization ... 5

j

JAPAN15 character set ... 15, 17
Japanese ... 15, 16, 29

k

KANA8 character set ... 13, 15, 17
Kanji ... 8

l

LANG environment variable ... 23, 44, 47, 49, 51
langinfo library routine ... 21, 22, 27, 27, 28, 28
langinfo.h ... 25
langinit library routine ... 31
langtoid library routine ... 27, 28
language name ... 17
language number ... 17
language tables ... 24
language-dependent information ... 22
languages ... 11
lexical order ... 23
library calls ... 24
library header files ... 22
library routines ... 26
linedrawing character set ... 13, 15
local customs ... 6, 7, 9

localization 5, 10, 37, 38, 51
localized command 24
localized message file 10

m

manual conventions 3
math character set 13, 15
message catalog 10, 22, 24, 27, 37, 39, 40, 46, 47, 51, 51
message catalog commands 38
messages 7
msgbug.h 25
multi-byte character codes 8
multi-byte library routines 31
multi-byte macros 31

n

native language 17
`native-computer` 18, 21, 22, 37, 44
natural language 17
nl_asctime library routine 26
nl_atof library routine 31
nl_conv library routine 28
nl_ctime library routine 26
nl_ctype(3C) library routine 29
nl_ctype.h 25
nl_gcvt library routine 26
nl_isalnum library routine 28, 29
nl_isalpha library routine 28, 29
nl_isgraph library routine 28, 29
nl_islower library routine 28, 29
nl_ispunct library routine 28, 29
nl_isupper library routine 28, 29
NLS definition 6
NLS header files 25
NLS support 7
nl_string(3C) library routine 29
nl_strtd library routine 31
nl_tolower library routine 28
nl_tools_16 (3C) manual page 31
nl_toupper library routine 28

non-ASCII string collation 29
number representation 9

p

Pascal 24, 37, 46
PCHAR macro 31
PCHARADV macro 31
peripherals 13, 14
prelocalization 10, 48
prelocalized commands 21
printf(3S) library routine 30
printmsg (3C) library routine 30
printmsg library routine 38
printmsg(3C) library routine 34
programmer interface 7

r

radix character 26
ROMAN8 character set 13, 15, 17

s

SECof2 macro 31
$set 43, 44, 45, 50
shifting 8
sorting 8
sprintmsg library routine 30, 38
strcmp16 library routine 29
strcmp8 library routine 29
strncmp16 library routine 29
strncmp8 library routine 29
strtod(3C) library routine 31
supported character sets 19
supported languages 11, 17, 19

t

time format 9, 17, 26
time zone 9, 23
TURKISH8 character set 13, 15, 17
two-byte character 15
TZ environment variable 23

u

upshifting 17, 22, 23
USASCII character set 11, 17
/usr/lib/nls access permissions 50

MANUAL COMMENT CARD

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

HP Part Number 97089-90062 September 1986

Please help us improve this manual. Circle the numbers in the following statement that best indicate how useful you found this manual. Then add any further comments in the spaces below. **In appreciation of your time, we will enter your name in a quarterly drawing for an HP calculator.** Thank you.

The information in this manual:

Is poorly organized	1	2	3	4	5	Is well organized
Is hard to find	1	2	3	4	5	Is easy to find
Doesn't cover enough	1	2	3	4	5	Covers everything
Has too many errors	1	2	3	4	5	Is very accurate

Particular pages with errors? ______________________________

Comments: ______________________________

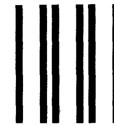
Name: ______________________________

Job Title: ______________________________

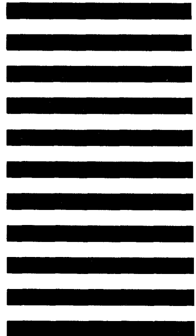
Company: ______________________________

Address: ______________________________

☐ Check here if you wish a reply.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

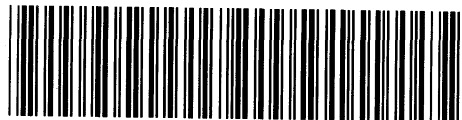
POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

HEWLETT
PACKARD

HP Part Number
97089-90052

Microfiche No. 97089-99052
Printed in U.S.A. 9/86



97089-90652

For Internal Use Only